TS-Trees: A Non-Alterable Search Tree Index for Trustworthy Databases on Write-Once-Read-Many (WORM) Storage*

Jian Pei
Simon Fraser University
jpei@cs.sfu.ca

Man Ki Mag Lau
Simon Fraser University
mklau@sfu.ca

Philip S. Yu
IBM T. J. Watson Research Center
psyu@us.ibm.com

Abstract

Trustworthy data processing, which ensures the credibility and irrefutability of data, is crucial in many business applications. Recently, the Write-Once-Read-Many (WORM) devices have been used as trustworthy data storage. Nevertheless, how to efficiently retrieve data stored in WORM devices has not been addressed sufficiently and thus remains a grand challenge for large trustworthy databases. In this paper, we describe a trustworthy search tree framework (called TS-tree), which is a simple yet effective non-alterable search tree index for trustworthy databases. It can take the role of B-trees in trustworthy databases to answer various queries including range queries. It is efficient and scalable on large databases. A systematic simulation verifies our design.

1 Introduction

Many regulations, such as Sarbanes-Oxley Act by the Congress of the USA, the USA Patriot Act, HIPAA, the European Union Data Protection Directive, and SEC Rule 17a-4 by the Securities and Exchange Commission, require that electronic data records must be trustworthy. To meet the regulations on trustworthy data storage, the Write-Once-Read-Many (WORM) devices have been used. In May 2003, the Securities and Exchange Commission (SEC) clarified that any electronic storage system could be used, provided that it meets the *non-rewritable* and *non-erasable* requirements. In the light of the regulations, both the optical WORM storage systems and the hard-drive-based WORM storage systems such as IBM TotalStorage DR550 and IBM Tivoli Storage Manager for Data Retention can be used.

Conceptually, in a WORM device, a unit of storage (e.g., a byte or a word) can be in one of the two states: *unset* or *set*. A unit is in this state if the unit has not been used yet, i.e., no data is stored into this unit. A unit is set when a data entry is written into it. Once a unit is set, it cannot be rewritten or altered.

While most of the existing techniques focus on how to fulfill the non-rewritable and non-erasable requirements, how to efficiently retrieve the data stored in WORM devices remains a grand challenge and has not been addressed sufficiently. Without a proper index, searching a large database by linear scans can be costly. However, to make the access to a non-rewritable and non-erasable database trustworthy, the index itself must also be non-rewritable and non-erasable. Otherwise, the data stored in the WORM storage can be hidden or altered in the query results. Clearly, most of the existing indexing techniques in database systems do not satisfy the requirement – they are neither non-rewritable nor non-erasable in incremental maintenance.

In a pioneering study [9], Zhu and Hsu established the linchpin of trustworthy indexes as the following five requirements. First, *once a record is committed to a WORM storage, the record and the path to access the record must also be committed and immutable*. Second, *the index must be incrementally maintainable, and be scalable to large databases*. Third, *the data must be searchable using the index efficiently*. Fourth, *the cost of the index in space must be reasonable*. Last, *a record should be destroyed when it is expired*. They also proposed the *Generalized Hash Tree (GHT)* techniques, which index non-rewritable and non-erasable records on WORM devices by hashing.

Although GHT satisfies the requirements on being trustworthy and can handle probing queries well, *GHT is not an ordered index and cannot efficiently handle many database queries other than probing ones due to the inherent limitation of hashing indexes*. For example, it is hard to use GHT for range queries, that is, finding all records whose indexed attribute values are in a given range.

Ordered indexes are critical for many database operations. However, the task of designing a trustworthy ordered index for non-rewritable and non-erasable data is challenging. Generally, such a design needs to achieve three goals

*The research of Jian Pei is supported in part by NSERC Grant 312194-05 and NSF Grant IIS-0308001. The research of Man Ki Mag Lau is supported in part by an NSERC Undergraduate Student Research Award and NSERC Grant 312194-05. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

simultaneously. First, *we need to maintain a virtual sorted list of records*. How to maintain the virtual sorted list against updates without changing any access paths to the existing records is far from trivial. The dynamic updates on the existing access paths during incremental maintenance prevent most of the existing indexes from being used for trustworthy purposes.

Second, *we need to design a (relatively) balanced data structure*. If the index is far from balanced, then the search cost for some records can be high.

Last, *we need to provide an effective mechanism to detect adversarial changes*. Even with the database non-alterable, an adversary may still be able to hide some data items by adding some other new items in a way bypassing the normal index maintenance methods. Thus, a highly trustworthy index should provide a mechanism to detect such changes effectively.

In this paper, we present a relatively balanced search tree index TS-tree as an effective solution. The idea is simple yet effective: *We use a probabilistic method to accommodate records in a search tree structure such that the tree is balanced in expectation*. Moreover, *we provide a real time mechanism to detect any adversarial changes at the query time*. We report an extensive simulation to evaluate the performance of TS-trees using both synthetic data sets and real data sets. *To the best of our knowledge, this is the first ordered index for trustworthy databases, and the first method that provides adversary detection in real time.*

The remainder of the paper is organized as follows. In Section 2, we review related work. In Section 3, we consider a rudimentary simple search tree method. We develop the TS-tree structure in Section 4, and present the adversary detection mechanism in Section 5. Section 6 gives algorithms for answering queries using TS-trees. An extensive simulation study is reported in Section 7.

2 Related Work

A *Trustworthy archive* is to create concrete proof and precise details about the events that have been occurred and recorded.

In [6], Hsu and Ong proposed an end-to-end trustworthy process. The central idea is that the trustworthy process prohibits any change to the history. In [9], Zhu and Hsu developed the Generalized Hash Tree (GHT) technique as a non-alterable index for trustworthy databases. The general idea is to use a universal hash function $h(x) = ((ax + b) \bmod p) \bmod r$ to hash the records level by level and form a tree, where p is a prime number chosen such that all the possible keys are less than p, r is the size of the target range, $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$. Insertions or lookups of a record start from the root node of the tree. If unsuccessful, the process recurses at one or more of its children subtrees. If a record cannot be inserted into any of the existing nodes, a new node is created and added to the tree as a leaf. While the GHT techniques are effective, one serious problem remains unsolved. Due to the inherent limitation of hashing indexes, GHT can only work well for probing queries.

Indexing is the fundamental practice in database research and development. Among many others, ordered index such as balanced search trees (e.g., B-tree, B+-tree and their variations and extensions) and hashing indexes are fundamental in both theory and practice. .

There exist previous attempts to build ordered indexes on WORM storage. A natural idea is to use copies (i.e., versions) of the index structure. Some previous studies such as [8, 3] developed methods to copy only the changed nodes in the index structure, i.e., the unchanged parts are shared by versions. As analyzed in [9], when the copying-based methods are used for WORM storage, if a node r is copied, any node having a pointer to r must also be copied. In other words, any paths from the root of a search tree to any changed nodes must be copied. The space overhead is non-trivial. Moreover, an adversary may omit or counterfeit entries in the ongoing copying of newly modified paths.

To avoid the frequent copying, the idea of node splitting was exploited in *write-once B-tree* [4], *multi-version B-tree* [2], and *append-only trie* [7]. When a node overflows, it is split into new nodes and pointers are used in its parent node to override the earlier pointer. Again, in the context of WORM storage, the semantics of "overriding" raises a chance for an adversary to tamper the index.

In real applications, an adversary may want to hide some historical records that were already stored in a trustworthy database. In the context of trustworthy indexes, an *adversarial attack* on a record in a trustworthy database tries to change the corresponding access path in the trustworthy index so that the target record cannot be located using the index. As an assurance of trustworthy indexing services, it is important for trustworthy indexes to detect adversaries effectively. To the best of our knowledge, the adversary detection issue has not been technically addressed in the previous studies about trustworthy indexes or indexes on WORM devices.

For example, in GHT, an adversary may change the hashing function at some level to hide the data in some subtrees. No method is provided to detect the adversarial changes unless one scans the whole database and verifies the GHT index.

3 Simple Search Trees

To keep our discussion easy to follow, we present the ideas and algorithms using binary search trees. The ideas and the methods can be easily generalized to arbitrary k-nary (i.e., k-way) search trees. We will report the experimental results on k-nary trees in Section 7.

A value in the indexed attribute is called a *key*. In a query (i.e., lookup), the index attribute value to be searched is called the *search key*.

First, let us consider *whether we can build non-rewritable and non-erasable search trees*. That is, at this moment we ignore the requirement of being relatively balanced for the search trees.

We can start from an empty tree, which is a trivial search tree for an empty set, to construct a search tree incrementally. To honor the non-rewritable and non-erasable requirements, at each step, we cannot change the access path of any existing node. Instead, we can only extend the current tree by adding new branches. In other words, we cannot balance the tree by adjusting the relative positions of the existing nodes in the tree.

As an important advantage, *a simple search tree is immune to adversarial attacks on records already in the tree*. That is, once a record is inserted into a simple search tree, its access path is completely fixed and thus cannot be changed later. As described in Section 5, this property can be used to detect adversarial attacks.

A simple search tree satisfies the non-rewritable and non-erasable requirements. However, the shape of a simple search tree is very sensitive to the arriving order of tuples.

4 TS-tree

4.1 Ideas

Suppose we are building a binary search tree, and the keys are real numbers uniformly distributed in range $[l, u]$. Consider the first tuple t_1 whose key is v_1. *If we put v_1 at the root of the search tree, what is the probability that the left subtree and the right subtree of the root would be balanced, i.e., have the same number of nodes in the future?*

Intuitively, if v_1 is about the mean of the range, i.e., $v_1 \approx \frac{l+u}{2}$, then putting v_1 at the root node has a good chance to lead to a tree in the future whose left and right subtrees have similar numbers of nodes.

On the other hand, if v_1 is close to l, the left end of the domain, then putting the tuple at the root node may lead to a small left subtree and a large right subtree in the future. In other words, the search tree in the future may be poorly balanced. Thus, it would be better to create a root node with an unfilled key value and not associated with any tuple at this moment, and accommodate v_1 and tuple t_1 in the left subtree. Symmetrically, if v_1 is close to u, the right end of the domain, then it would be better to create a root node with an unfilled key value and not associated with any tuple at this moment, and accommodate v_1 and tuple t_1 in the right subtree.

The critical idea is that *we accommodate a tuple in a search tree according to its probability to balance the numbers of nodes in its left and right subtrees*. By such a probabilistic arrangement, we hope to achieve a compact yet relatively balanced search tree index in expectation.

4.2 TS-tree Construction: the Framework

Assume that the index attribute is in the domain of $[l, u]$ and follows a distribution p, that is, $\int_l^u p(x)dx = 1$. In practice, we can estimate the distribution of data roughly by domain knowledge or sampling the data.

For any node c in a search tree, c is called *balanced* (in expectation) if the expected number of nodes in the left subtree of c equals the expected number of nodes in the right subtree of c.

The search tree is initiated as an empty tree. When the first tuple t_1 comes, let v_1 be its key. If v_1 is put at the root node, then the probability that a future tuple is in v_1's left subtree is $p_{left} = \int_l^{v_1} p(x)dx$. Similarly, the probability that a future tuple is in v_1's right subtree is $p_{right} = \int_{v_1}^u p(x)dx$. In addition, we have $p_{left} + p_{right} = 1$.

The probability that the expected number of nodes in v_1's left subtree is not equal to the expected number of nodes in v_1's right subtree is

$$p_{diff} = |p_{left} - p_{right}| = |2 \cdot p_{left} - 1|. \quad (1)$$

The smaller the difference between p_{left} and p_{right}, the better that v_1 is chosen as the root node. Thus, we can use $(1 - p_{diff})$ as the probability to put v_1 as the root node of the search tree and link t_1 to the root.

If v_1 is chosen as the root, then the first tuple is accommodated and we are ready to insert the second tuple. Otherwise, we create a root node that is not linked to any tuple and the key value at the root node is unfilled. The left subtree is expected to store tuples with index attribute values in range $[l, w)$ and the right subtree is to store tuples with index attribute values in range $(w, u]$, where

$$\int_l^w p(x)dx = \int_w^u p(x)dx = 0.5. \quad (2)$$

Value w is the *mean* of the distribution.

Therefore, we put t_1 into the left subtree of the root node if $p_{left} < 0.5$, and put t_1 into the right subtree of the root node if $p_{right} < 0.5$ (i.e., $p_{left} > 0.5$). Recall that if $p_{left} = 0.5$, v_1 will be taken as the root node since $1 - p_{diff} = 1$.

Let us consider the situation where $p_{left} < 0.5$ and v_1 is not chosen as the root. Since at this moment, the left subtree is empty, whether v_1 should be accommodated as the root of the left subtree (i.e., the left child of the root node of the whole tree) depends on its probability to be balanced for the future tuples in range $[l, w)$. Technically, this can be determined by comparing p_{left} and 0.25. We can make a decision according to probability $1 - \frac{|p_{left} - 0.25|}{0.25}$. The recursion can go on until either t is chosen at some node or a

Function `find-m-left(r)`
// find m_{left} of node r in a TS-tree
// the key is in domain $[l, u]$
`BEGIN`
1. `IF` r has a right child r_{right}
2. `THEN` return `find-m-left(`r_{right}`)`;
3. `ELSE-IF` r stores a key value v
4. `THEN` return v;
5. `ELSE-IF` r has a left child r_{left}
6. `THEN` return `find-m-left(`r_{left}`)`;
7. `ELSE` return l;

`END`

Figure 1. Finding m_{left}.

termination condition is reached. The termination condition will be discussed in Section 4.3.

Similarly, if $p_{left} > 0.5$ and v_1 is not chosen as the root, we can accommodate v_1 at some node in the right subtree of the root.

Suppose a search tree T is built. When a new tuple t comes, whose key is v, we need to insert the new tuple into the tree.

We first compare v and the key value v_0 at the root node of tree T. Two cases may happen.

If v_0 is unset, i.e., the root node has not stored a key value yet, then we need to determine whether v is good to be stored at the root node. In order to be stored at the root node, v must satisfy two requirements as follows. First, v should be close to value w in Equation 2. Second, $m_{left} < v < m_{right}$, where m_{left} and m_{right} are the largest key value in the left subtree and the smallest key value in the right subtree of the root node, respectively.

If the root node does not have a left child, then the largest key value in the left subtree of the root node is l. Otherwise, we can call `find-m-left(`r_{left}`)` in Figure 1 to search for the largest key value m_{left} from the left subtree of the root node, where r_{left} is the left child of the root node. Symmetrically, we can search for the smallest index attribute value m_{right} from the right subtree of the root node.

Similar to the discussion in Section 4.2, we can also compute p_{diff} (Equation 1) for v and use $(1 - p_{diff})$ as the probability to store v at the root node.

If v does not satisfy the second condition, or v is not chosen as the root node, then we check whether $v \leq m_{left}$ or $p_{left} = \int_l^v p(x)dx < 0.5$. If so, then t will be inserted into the left subtree. Otherwise, t will be inserted into the right subtree.

If v_0 is set, then we compare v and v_0 to decide whether tuple t should be inserted into the left subtree or the right subtree. Inserting a tuple into a subtree can be conducted as the recursion of the above process. Particularly, if a subtree is empty, then the subtree is expanded as described in Section 4.2.

Generally, the scope of a node r in a TS-tree can be determined as follows. First, the scope of the root node is $[l, u]$. Second, for node r, if the key value stored in the parent of r is v_p, and the scope of the parent of r is $[l', u']$, then the scope of r is $[l', v_p]$ if r is the left child, and $[v_p, u']$ if r is the right child. Last, for node r, if the key value stored in the parent of r is unset, and the scope of the parent of r is $[l', u']$, then the scope of r is $[l', w]$ if r is the left child, and $[w, u']$ if r is the right child, where w is the mean in range $[l', u']$. That is, $\int_{l'}^{w} p(x)dx = \int_{w}^{u'} p(x)dx = \frac{1}{2}\int_{l'}^{u'} p(x)dx$.

During the insertion, when a node r which has not stored a key value yet is met, we can accommodate the being inserted tuple at r using the probability that the key of the tuple can balance the subtrees of r in its scope.

4.3 Controlling the Size of a TS-Tree

Consider the insertion of the first tuple as described in Section 4.2. *In the worst case, how long the path would be from the root to the node storing the tuple?*

If the domain of the index attribute is categorical with cardinality n, then the length of the path is $\lceil \log_2 n \rceil$ in the worst case. If the domain of the index attribute is infinite, then, the length of the path is also infinite in the worst case.

Clearly, the tree size in the worst case is highly undesirable. *Can we confine the tree size such that it is linear to the expected number of tuples in the database?*

Suppose that the expected number of tuples in the database is m. For a node r in the search tree whose scope is $[l_r, u_r]$, we can compute the probability p_{index} that a tuple is indexed by the subtree rooted at r as $p_{index} = \int_{l_r}^{u_r} p(x)dx$.

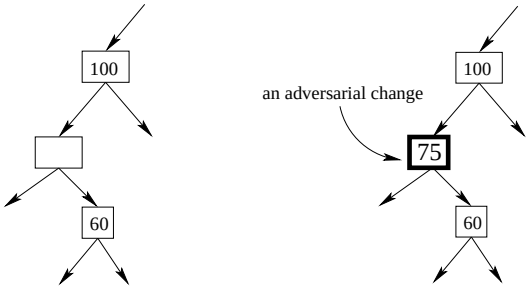
Intuitively, if p_{index} is very small, further pushing a key value v deep into the search tree does not bring any benefit, since likely no tuple will be indexed by the subtree rooted at r. Thus, we can stop growing the tree and simply adopt v at the current node if $p_{index} \leq \frac{1}{\alpha \cdot m}$, where $\alpha \geq 1$ is a space reservation ratio which is a fudge parameter set by the user, and m is the expected number of tuples in the database.

For any node r with $p_{index} \leq \frac{1}{\alpha \cdot m}$, all tuples falling into the subtree rooted at r will be organized in a simple search tree as discussed in Section 3.

Often, the size of a large database cannot be predicted accurately. Moreover, a database may grow over time. To handle such incrementally growing databases, we can increase the estimation of the number of tuples m periodically. Please note that parameters m and r do not affect the correctness of a TS-tree, and thus they can be changed dynamically.

5 Adversary Detection

If data records are inserted into a TS-tree following the tree construction and maintenance algorithms, a TS-tree is



(a) A segment of a TS-tree. (b) An adversarial change.

Figure 2. An example of adversarial changes.

guaranteed as a correct trustworthy index on the underlining non-alterable database. However, if an adversary bypasses the TS-tree maintenance method and inserts an obstructing entry, a TS-tree may be in an inconsistent state such that some data entries may not be properly accessible using the index. It is important to provide a mechanism to detect such adversarial attacks in acceptable cost.

In a TS-tree, the following consistent constraint is maintained by the tree construction and maintenance algorithms.

Property 1 (Consistent constraint) *In a TS-tree, let u and v be two nodes with key values u and v, respectively. $u \neq v$. Then, one of the following holds: (1) if u is in the left subtree of v then $u < v$; (2) if u is in the right subtree of v then $u > v$; or (3) if u and v are in the left and right subtrees of a common ancestor, then $u < v$.* ■

An adversary may break the consistent constraint and thus temper the index if the tree maintenance method can be bypassed.

Example 1 (Adversary) Consider the segment of a TS-tree shown in Figure 2(a).

If an adversary wants to hide the record with key 60, she/he can put one obstructing record of key value 75 into the TS-tree as the parent of the target tuple, as shown in Figure 2(b). This is an adversarial attack: it bypasses the TS-tree maintenance method and makes the TS-tree inconsistent. After the change, a query looking for key value 60 using the tempered TS-tree may miss the record. ■

For trustworthy indexes, an *adversarial attack* on a record in a trustworthy database is try to change the corresponding access path in the trustworthy index so that the target record cannot be accessed using the index.

In a TS-tree, an adversarial attack makes a subtree violates the consistent constraint. Thus, a straightforward approach to detecting adversarial attacks is to test a TS-tree thoroughly against the consistent constraint. That is, we conduct a depth-first search of the tree and check every node. Then, any inconsistency can be identified.

For example, a depth-first search of the tree in Figure 2(b) would find out that for the node with key 75, the minimum value in its right subtree is less that 75. Thus, there is an adversarial attack.

While the full-tree test method is correct, it may not be efficient and practical.

Interestingly, no adversarial changes can be made to a simple search tree described in Section 3, because once a record is inserted into a simple search tree, all nodes in the access path from the root to the record have been fixed.

Therefore, we can naturally combine the two structures as a double-tree approach. *The trustworthy database is organized as a simple search tree.* That is, all data records are stored and linked to form a simple search tree. This tree structure is used for verification only and not for query answering. *A TS-tree is built as a trustworthy secondary index.* Each node in a TS-tree with a filled key value will be linked to the corresponding node in the simple search tree.

The double-tree approach can provide real-time verification to detect adversarial changes that may affect the result of any query.

Suppose a query is looking for a key value k, and cannot find it in the TS-tree. Then, we should also look for the maximum value l_1 that is less than k and the minimum value l_2 that is greater than k from the TS-tree. The method will be explained in Section 6. Then, we should search the simple search tree for all nodes in the range of $[l_1, l_2]$, and find out whether a record of key value k exists. If such a record is found, then an adversarial change was made.

Note that this checking of simple search tree is not needed if the result set is not empty when looking for a key value of k. By construction, there cannot be partial hiding of tuples with a given value.

6 Query Answering Using TS-Trees

In this section, we discuss how to use TS-trees to answer typical database queries. Particularly, we focus on probing queries and range queries, which are essential in many database services. Given a search key on the index attribute, a *probing query* searches for the tuples that have the same key value as the search key in the database. Given a range $[a, b]$, a *range query* searches for the tuples whose keys are in the range.

The probing query answering algorithm using a TS-tree is very similar to that of a regular search tree, except that, for a tree node with an unset key value, we need to search the scopes of its left and right subtrees to determine whether one of the subtrees should be searched.

Technically, given a search key, we start the search from the root of the TS-tree. When we meet a node whose index value is set, by comparing the search key and the key value in the node, we can determine whether a match is found or we need to search the left subtree or the right subtree. When we meet a node whose index value is unset, we need to search m_{left} and m_{right}, the largest key value in the left

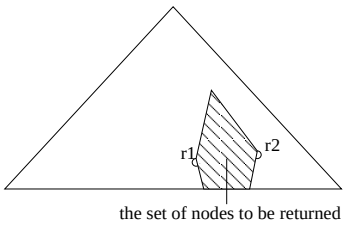


Figure 3. Answering range queries.

subtree and the smallest key value in the right subtree, respectively (Section 4.2). If the search key is less than or equal to m_{left}, the left subtree should be searched. Symmetrically, if the search key is greater than or equal to m_{right}, the right subtree should be searched. Otherwise, there exists no tuple in the database matching the search key.

Given a range $[a, b]$, the range query can be answered in two steps. First, we find the node r_1 with the least key value in the database that is larger than or equal to a. In the second step, we traverse in depth-first manner the right subtree of r_1, as well as the right ancestors of r_1 and their descendants, until a node r_2 is met whose index attribute value is b or over. Here, a node u_1 is a *right ancestor* of u_2 if u_2 is either the left child of u_1 or a descendant of the left child of u_1. Figure 3 illustrates the idea.

All the nodes searched, except for r_2 if its key value is over b, contain either an unset key value or a key value in range $[a, b]$. thus, all the tuples associated with those nodes should be returned. On the other hand, any tuples indexed by the TS-tree but not fall into this subset of nodes must have a key value outside the range.

We can extend the probing query answering algorithm to find r_1, the node with the least key value in the database that is larger than or equal to a.

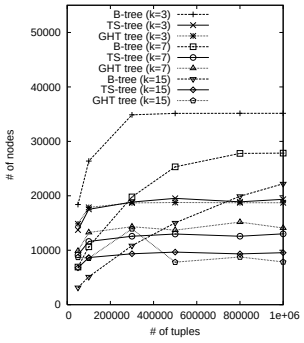
We call v the *nearest right neighbour* of q if v is the least key value in the database that is greater than or equal to a given search key q. Similar to answering a probing query, we search the TS-tree from the root. Generally, when a node r is met, two cases may happen.

On the one hand, if r stores a key value v_r, then we compare q and v_r. If $v_r = q$, then v_r is the answer. If $q < v_r$, then we need to check the left subtree of r. If r has a left subtree and $q \leq m_{left}$, then the left subtree should be searched. Otherwise, v_r is the answer. If $q > v_r$, then we need to check the right subtree of r. If r does not have a right subtree, then the nearest right neighbour with respect to q does not exist. Otherwise, the right subtree should be searched.

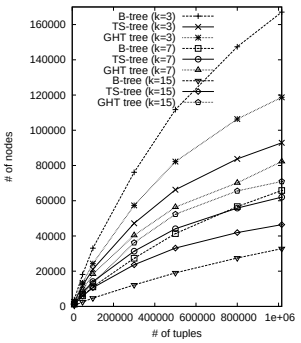
On the other hand, if r does not store a key value, then we check the subtrees of r. If r has a left child r_l with scope $[l_l, u_l]$, and $q \leq u_l$, then the left subtree should be searched. If r does not have a left child, or q does not fall in the scope of the left child of r, then the right child of r should be checked. If r has a right child r_r with scope $[l_r, u_r]$ and $q \leq u_r$ then the right subtree should be searched. If r does not have a right child or $q > u_r$, then the nearest right neighbour with respect to q does not exist.

Property	B-tree	GHT	TS-tree
Index type	Ordered	Hashing	Ordered
Probing queries	Yes	Yes	Yes
Range queries	Yes	No	Yes
Trustworthy	No	Yes	Yes

Figure 4. Comparison between B-tree, GHT and TS-tree indexes.



(a) Number of nodes on synthetic data sets



(b) Number of nodes on real data set Weather

Figure 5. Number of nodes versus database size.

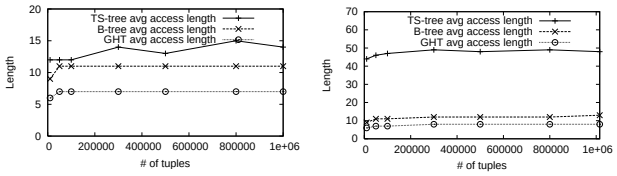
7 Simulation Results

In this section, we report a systematic simulation to empirically evaluate the effectiveness and the efficiency of our TS-tree design. All the experiments are run on a PC computer with an Intel Pentium 4 2.0 GHZ CPU and 785,904 KB main memory, running Microsoft Windows XP operating system.

We compare three methods: the B-tree, the GHT [9] and the TS-tree developed in this paper. Some important properties of the three methods are compared in Figure 4. *Among the three methods, only TS-trees are trustworthy and can answer both probing queries and range queries efficiently*.

We report the experimental results on both synthetic and real data sets. In the synthetic data sets, the key values follow the uniform distribution in range $[1, 1000000]$. The data sets contain $1,000,000$ tuples. We also use samples of various size of the synthetic data sets in the experiment. Since the range of the domain of key values is the same as the number of tuples in the synthetic data sets, using various samples and the full data set can illustrate the trend of performance of the indexes against the incremental maintenance.

To examine the effectiveness of the indexes on real data sets, we use the Weather data set [5], which contains $1,015,367$ tuples on 9 dimensions. The dimensions with



(a) Average access length on synthetic data sets

(b) Average access length on real data set Weather

Figure 6. Average access length versus database size.

the cardinalities of each dimension are as follows: station-id $(7,037)$, longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8), and brightness (2). We build an index using the lexicographic order on all attributes in the data set. That is, for tuples t_1 and t_2, $t_1 < t_2$ if there exists a dimension D_i such that $t_1.D_i < t_2.D_i$ and $t_1.D_j = t_2.D_j$ for any $(j < i)$. In the real data set, the domain is 1.45×10^{25} and is much larger than the number of tuples in the database. The distribution of the tuples is unknown and TS-trees uses uniform distribution to construct the index, which achieve good performance.

We also test using three cases of block size in the indexes: the ones that can hold 3, 7 and 15 key values each node for B-trees and TS-trees, respectively. Correspondingly, the GHT indexes use the blocks of the same size as B-trees and TS-trees to construct the hashing buckets. In other words, in a test, all the three methods use the blocks of the same size to construct the trees.

We test the size of the indexes on the number of tuples in the database. Figure 5(a) and (b) show the results on the synthetic data set with uniform distribution and the Weather data set, respectively. In each figure, we plot the three cases of different node size, $k = 3, 7, 15$, respectively, where k is the maximal number of keys per node in B-trees/TS-trees. We set the space reservation factor in TS-trees to 1.0 by default. From the figures, we make the following observations.

First, the number of nodes in all three indexes follows a sublinear trend in our experiments. This strongly suggests that *TS-trees and GHT trees for trustworthy data bases have a scalability similar to B-trees*.

Second, *TS-trees can be smaller than B-trees*. In fact, in most experiments on the synthetic data sets (Figure 5(a)), when the number of tuples is large, the TS-trees are smaller than the B-trees. The key values in our synthetic data sets are in range $[1, 1000000]$ and follow the uniform distribution. Thus, when the number of tuples is close to one million, many key values in the range appear. Since the space reservation factor is set to 1, many nodes in the TS-trees are filled and the TS-trees is relatively campact. On the other hand, in order to be balanced, a B-tree may contain many nodes that are not full. That may lead to the situations where a TS-tree is smaller than a B-tree.

On the real data set Weather, the domain of the keys is much larger than the number of tuples in the database. Figure 5(b) shows an interesting tradeoff. When the tree nodes are small and thus can hold a small number of keys (e.g., $k = 3$ in the figure), TS-trees are substantially smaller than B-trees. When the tree nodes are large (e.g., $k = 15$ in the figure), B-trees are smaller than TS-trees. When $k = 7$, the number of nodes in both trees are similar. With larger nodes, a TS-tree may have more unfilled fields and thus may cost more in space.

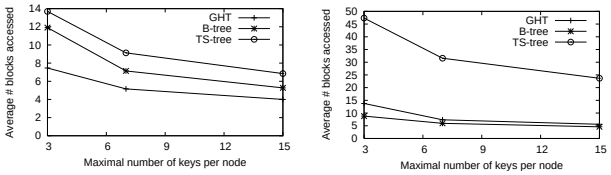
Third, between GHT trees and TS-trees which one is smaller depends on the data distributions. In our experiments, GHT trees and TS-trees have similar size on the synthetic data sets, and GHT trees are a little bit smaller in most cases. However, on the real data set Weather, the TS-trees are clearly smaller than the GHT trees.

Last, for all the three methods, the larger the nodes, the smaller the number of nodes. That is intuitive.

The length of the access paths in the indexes is critical for the efficiency of query answering. For a tree T, we define the *average access length* as $L(T) = \frac{\sum_{v \in T} l(v)}{|v \in T|}$, where $l(v)$ is the length of the path from the root node to v. In other words, under the assumption that each key value stored in a tree T will be queried with the same probability, $L(T)$ is the expected access length of all key values indexed in the tree. We argue that $L(T)$ is a better measurement of expected query efficiency than $H(T)$ if T is not balanced. It is easy to see $L(T) \leq H(T)$ if $H(T) > 1$.

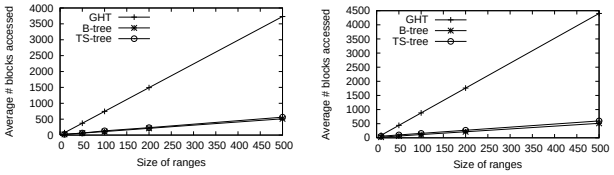
Figure 6(a) and (b) show, on the synthetic data sets following the uniform distribution and the Weather data set, respectively, the average access length of the three indexes with respect to the number of tuples in databases. The results are consistent. First, for all the three indexes, the average access length increases as the database becomes larger and larger. However, the increase is very moderate. Second, while the height of the TS-trees can be large in the worst case, the average access length is quite moderate. Third, the average access path in TS-trees is only up to 4 times of the average access path in the corresponding B-trees. In other words, the TS-trees are relatively balanced. Last, GHT trees are the shortest among the three indexes. This is the inherent advantage of hashing indexes and is consistent with the results in [9].

We notice that the average access length is substantially different on the synthetic data set and the Weather data set, though the numbers of tuples in these two data sets are similar. By a close examination of the resulting TS-tree, we find that the right subtree of the root is much larger than the left subtree. In other words, the distribution on the Weather data set is far from uniform. However, even in such a case, the average access length is only degraded slightly. the average access length is quite stable with respect to the number of tuples in the database. In fact, although the global distri-



(a) Probing query answering on the synthetic data set (b) Probing query answering on real data set Weather

Figure 7. Probing query answering performance versus the size of each node.



(a) Range query answering on the synthetic data set (b) Range query answering on real data set Weather

Figure 8. Range query answering performance versus the size of the ranges.

bution is far from uniform, in a relatively small subrange which corresponds to a lower level subtree, the distribution may have a good chance to be close to uniform.

Figure 7 evaluates the probing query answering performance using the three indexes. For each index, we count the number of nodes accessed during the process of answering a probing query, and assume that each node is stored in a distinct block. We issue a probing query for each tuple in the database and report the average number of blocks read in query answering. Moreover, we examine the average number of blocks read with respect to different size of nodes. Again, we use three sizes: the ones which can hold 3, 7 and 15 keys for B-trees and TS-trees, respectively.

The figures clearly show that all the three indexes can answer probing queries with lower I/O cost when larger blocks are used. Moreover, GHT has the best performance and B-tree has the performance similar to GHT. This is again consistent with the results in [9]. While the cost of TS-trees is the highest, the difference between TS-trees and B-trees as well as GHT becomes smaller when the block size becomes larger. Comparing to query answering using B-trees, the major extra cost using TS-trees is twofold: a TS-tree can be higher than a B-tree and we may have to search m_{left} and m_{right} for some subtrees during the query answering. Again, although GHT may have good query answering performance, it cannot answer range queries efficiently since it is not an ordered index.

To evaluate the range query answering performance, we issue range queries with ranges of different size. To use a GHT for range query, we issue a probing query for every key value in the range. Both the complete synthetic data set and the Weather data set are used. The average number of blocks accessed are calculated. The results are shown in Figure 8.

Both B-trees and TS-trees are efficient for answering range queries, while the GHTs are much slower. The difference becomes even bigger with larger ranges. For both B-trees and TS-trees, once the least key in the range is found, the other keys in sequel can be found by browsing the nodes in a depth-first search manner. But for GHTs, a probing query is needed for each possible key value in the range. The performance of B-trees and TS-trees are close, while B-trees are a little bit better. The blocks with unfilled key values in TS-trees may lead to visiting more blocks in TS-trees than in B-trees. However, as the size of B-trees and TS-trees are close to each other, the blocks with unfilled key values occur only small percentage in the query answering.

8 Conclusions

Trustworthy data processing is essential for many critical applications. In addition to trustworthy data storage, we often need trustworthy indexes for searching large trustworthy databases. In this paper, we describe a simple yet effective ordered index structure TS-trees, which are efficient, scalable to large databases, and can support many database tasks including probing queries and range queries.

References

[1] J. A. Aslam. A simple bound on the expected height of a randomly built binary search tree. In *Technical Report TR2001-387*, Dartmouth College Department of Computer Science, 2001.

[2] B. Becker et al. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[3] J R Driscoll et al. Making data structures persistent. In *STOC '86*.

[4] Malcolm C Easton. Key-sequence data sets on indelible storage. *IBM J. Res. Dev.*, 30(3):230–241, 1986.

[5] Hahn et al. *Edited Synoptic Cloud Reports from Ships and Land Stations over the Globe. 1982-1991*. Available at http://cdiac.esd.ornl.gov/, 1994.

[6] W. W. Hsu and S. Ong. Fossilization: A process for establishing truly trustworthy records. In *Technical Report RJ 10331*, IBM Almaden Research Center, 2004.

[7] P. Rathmann. Dynamic data structures on optical disks. In *ICDE'84*.

[8] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[9] Q. Zhu and W.W. Hsu. Fossilized index: The linchpin of trustworthy non-alterable electronic records. In *SIGMOD'05*.