Efficient Maintenance of Ephemeral Data

Albrecht Schmidt and Christian S. Jensen

Department of Computer Science, Aalborg University, Denmark
{al, csj}@cs.aau.dk

Abstract. Motivated by the increasing prominence of loosely-coupled systems, such as mobile and sensor networks, the characteristics of which include intermittent connectivity and volatile data, we study the tagging of data with so-called *expiration times*. More specifically, when data are inserted into a database, they may be stamped with time values indicating when they expire, i.e. when they are regarded as stale or invalid and thus are no longer considered part of the database. In a number of applications, expiration times are known and can be assigned at insertion time. We present data structures and algorithms for online management of data stamped with expiration times. The algorithms are based on fully functional treaps, which are a combination of binary search trees with respect to a primary attribute and heaps with respect to a secondary attribute. The primary attribute implements primary keys, and the secondary attribute stores expiration times in a minimum heap, thus keeping a priority queue of tuples to expire. A detailed and comprehensive experimental study demonstrates the well-behavedness and scalability of the approach as well as its efficiency with respect to a number of competitors.

1 Introduction

We explore aspects of implementing an extension to Codd's relational data model [5] where each tuple in a relation is timestamped with an *expiration time*. By looking at a tuple's timestamp, it is possible to see when the tuple ceases to be part of the current state of the database. Specifically, assume that, when a tuple r is inserted into the database, it is stamped with an *expiration time*, $t^{\exp}(r)$. Tuple r is thus considered part of the current state of the database from the time of insertion until $t^{\exp}(r)$. Expiration-time semantics now ensures that operations, most prominently queries, do not see tuples that have expired by the time associated with a query. Our study is motivated by the emergence and increasing prominence of data management applications which involve data for which the expiration time is known at the time of insertion, updates are frequent, and the connectivity of the data sources that issue the updates is intermittent. Applications which involve mobile networks, sensor networks, and the Internet generally qualify as examples.

Data produced by sensors that measure continuous processes are often short-lived. Consider a sensor network of temperature sensors that monitor a road network. It may be assumed that a temperature measurement is valid for at most a fixed number of minutes after it is measured, or the duration of validity may be determined by more advanced computations in the sensor network. A central database receives temperature measurements stamped with expiration times. A measurement from a sensor then automatically disappears if the sensor does not issue a new temperature measurement before the old measurement expires. While a temperature sensor network may be relatively static in nature, mobile devices that frequently log on to and log off from access points form a more dynamic network. In such a context, it is natural to tag records that capture log-ons with expiration times so that a session can be invalidated by the server after a period of inactivity. Closely related examples are cookies and session keys used where only stateless protocols like HTTP [20] are used for communication. Similarly, availability tactics like *heartbeats* [3], where devices periodically emit (heartbeat) messages to indicate that they are still online, go together with expiration times in a natural way. For a more detailed discussion about the role of expiration times in query processing see [17].

By adding the notion of expiration time to a database management system (DBMS), designers can help simplify software architectures and reduce code complexity while retaining transparent semantics. A user of an expiration time-enabled SQL engine needs not be aware of the new concept, as expiration time ∞ can be assumed for tuples for which no expiration time is provided explicitly. A further benefit of the integration of expiration times into a DBMS is that the number, and thus cost, of transactions especially in distributed systems can often be reduced significantly because no explicit delete statements to 'clean-up' previous transactions need be issued; since transaction costs in these settings are often an important bottleneck, overall system performance can increase significantly.

The *contributions of this paper* are as follows. (1) Motivated by the ubiquity of loosely-coupled distributed systems with unstable connections such as mobile and sensor networks, we argue that DBMS support for expiration time benefits applications, as pointed out above. (2) The main technical contribution of this paper are online main-memory algorithms and data structures that are capable of handling data expiration efficiently on a variety of devices; this implies that expired data are automatically removed as early as possible from the database without the need for user interaction. (3) A comprehensive experimental study offers insight into resource consumption and other performance characteristics such as scaling behaviour, response times, and throughput.

The remainder of the paper is structured as follows. The next section briefly outlines the assumed (simple) extension to the relational model and discusses functional treaps from an algorithmic point of view. Section 3 presents the results of a comprehensive evaluation of the performance characteristics of treaps and a comparative study of the performance of treaps with respect to various competing data structures; it also covers a variety of functional issues in relation to expiration times and the use of treaps. After a review of related work, the final section concludes the paper and identifies promising directions for future research.

2 Treaps in Detail

2.1 Setting

We assume the following basic setting for our research: data sources emit tuples stamped with expiration times. A relational view of these sources is provided, where only current, i.e. unexpired, tuples are exposed to queries. Thus, expiration times can be seen as a database-internal function $t^{\exp}$: *tuples* $\rightarrow$ *timestamps* from tuples to timestamps. Assuming that a database db^{exp} of tuples with expiration times is given and that the time associated with a query q is given by τ_q, then the tuples seen by q are: $\{\, r \,|\, r \in db^{\mathrm{exp}} \wedge t^{\mathrm{exp}}(r) > \tau_q \}$. It is a fundamental decision to associate expiration times with tuples. Arguably, they could be associated with other constituents of the relational model, including individual attribute values, attributes or other schema elements. This design decision is motivated by a desire for clear semantics, simplicity, and practicality [17, 18].

2.2 Overview

A treap is a combination of a tree and a heap: with respect to a (primary) key attribute, it is a binary search tree; with respect to a second, non-key attribute, it is a heap. The idea we elaborate on in the remainder of this paper is to use the key attribute for indexing while managing expiration times using the second non-key attribute.

We use the term (*fully*) *functional* [16] for a data structure if an update of the data structure produces a new version, both physically and logically, without altering the original. Functional data structures enable concurrent access through versioning [4]. In particular, by using a functional treap then, as long as only one thread updates the treap (like any other fully functional data structure), concurrent (read) access can be implemented with a minimum of locking, which is desirable in a main-memory environment.

```
class Node;
class Inner extends < $k, t, v$ > Node {
  left child: Node;
  key: $k$;
  expiration: $t$;
  data: $v$
  right child: Node;
} /* Instantiate with: Inner($l_c, k, t, c, r_c$) */
class Leaf extends Node {};
  /* Instantiate with: Leaf */
```

Fig. 1. Treap node data type

The structure of a treap node is shown in Fig. 1. The layout of the tree is binary: each node has a left and a right child, a key, and an expiration time; it also has a value field, which may contain arbitrary data such as non-key attributes.

2.3 Example

We proceed to exemplify how functional treaps can be used to support expiration times efficiently. Focus lies on eager removal of expired data.

Figure 2 shows the construction of a treap given the following sequence of (key, expiration time) pairs to be inserted: $(1, 7), (2, 6), (3, 6), (4, 0), (5, 7), (6, 6), (7, 8)$. A pair (key, time) denotes a tuple with key 'key' and associated expiration time 'time'. Note that, for the time being, we assume that the key and the expiration time are statistically independent; in [18], we discuss in more detail what happens when we do not make this assumption; we now just remark that we can use a hash function on the key to achieve independence. The last step in Fig. 2 consists of removing the root node from the treap, i.e. carrying out an expiration, for example at time 1. The algorithms that do the actual work are discussed in the sequel.

Before presenting algorithms, however, we have a quick look at the notation used in this paper. First, the functional nature is reflected in the code by the absence of

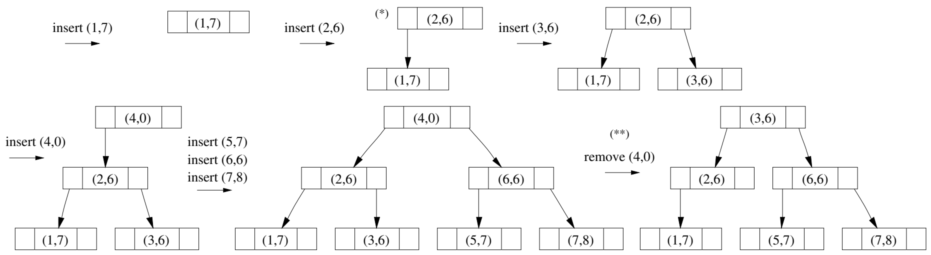


Fig. 2. Example treap

the assignment operator and, instead, the allocation of new objects with the **new** keyword whenever an update is performed. The function **new** t $(a_1, a_2, \ldots, a_n)$ allocates a new object of type t and initialises it by calling the respective constructor with the arguments $a_1, a_2, \ldots, a_n$. Second, extensive use of ML or Scala-style *pattern matching* [15] is made to bind parts of complex, nested data structures to variables in a concise manner avoiding combinations of nested if-statements. For example, assuming the class definitions of Fig. 1, using the second treap in the first row of Fig. 2 (marked (*)) and the find function of Fig. 3 then the first clause of Fig. 3 is executed as follows. Assume we want to find the node with the key 1, i.e. we call 'find $(treap, 1)$' where *treap* is bound (using the constructor notation) to 'Inner (Inner (Leaf, 1, 7, $\perp$, Leaf), 2, 6, $\perp$, Leaf)'. If we match against it the pattern 'Inner (_, k, _, item, _) **when** (key $= k$)' (line 3, Fig. 3), the following variable bindings are created: $k = 2, \text{item} = \perp$ ($\perp$ denotes a non-applicable variable in our case, i.e. we do not use the data field in this example). The underscore '_' in a constructor denotes a 'don't care' variable that is present in the class, but for which no binding is created. Since k is bound to 2 and the function argument key to 1, the **when** clause evaluates to false and the pattern does not match. However, the pattern in line 4 matches, and the following bindings are created: left $=$ Inner (Leaf, 1, 7, $\perp$, Leaf), right $=$ Leaf, and $k = 2$. Since the **if** statement evaluates to true, function find is called recursively and terminates successfully.

```
1 function find (node, key) =
2   match node with
3   | Inner (_, $k$, _, item, _) when (key $= k$) $\rightarrow$ item
4   | Inner (left, $k$, _, _, right) $\rightarrow$
5     if (key $< k$)
6     then find (left, key)
7     else find (right, key)
8   | Leaf $\rightarrow$ raise exception (Key is not in treap)
```

Fig. 3. Lookup of a primary key

If a treap is well-balanced, i.e. structurally similar to a height-balanced binary tree, it is guaranteed that we can execute look-up queries in logarithmic time. Treaps being also heaps implies that nodes with minimal expiration times cluster at the treap root. If the root node has expired, i.e. its expiration timestamp e is smaller than the current time, we can simply remove it.This is advantageous because we can keep the amount of stale data to a minimum using an eager deletion policy: as long as the root node is stale we remove it. Since stale data cluster at the root, no search is required. Furthermore, this strategy has the advantage that we essentially only need one procedure for both expiration and deletion: indeed, expiration is implemented as a small wrapper.

2.4 Operations on Treaps

This section introduces the most important operations on treaps. Since we are not aware of any other work that presents algorithms for the functional variant of treaps, we describe the operations in some detail.

Maintaining Balance. Like many other balanced tree structures, the insert and delete functions of treaps maintain balance through order-preserving node rotations. The insert function only rotates nodes on the path from a leaf, namely the newly inserted node, to the root. The delete function uses rotations to move an interior node to the leaf level without violating the order of the tree. Due to the functional nature of our kind of treap, rotations during inserts and deletes are implemented by slightly different code: for insertion (Fig. 4), the local function that implements the rotations is called *rebalance* (see second line in figure). We remark that it is the way the treap is rebalanced that destroys locality and makes it hard to adapt treaps to paginated secondary memory data structures.

Insertion. Insertion is a two-stage process. First, we insert a pair (key, time) as if the treap was a functional binary tree on key. We then execute rotations to re-establish the heap property (while retaining the binary tree property), which may have been violated. Insertion works as displayed in Fig. 4; it illustrated in several places in Fig. 2.

```
function insert (tree, key, time, item) =
  local function rebalance (node) =
    match node with
    | Inner(Inner($s_1, u, t', i', s_2$), $v, t, i, s_3$) when ($t > t'$) →
      new Inner($s_1, u, t', i'$, new Inner($s_2, v, t, i, s_3$))
    | Inner($s_1, u, t, i$, Inner($s_2, v, t', i', s_3$)) when ($t > t'$) →
      new Inner(new Inner($s_1, u, t, i, s_2$), $v, t', i', s_3$)
    | _ → node;
  match tree with
  | Inner(_, $k$, _, _, _) when (key = $k$) → tree
  | Inner(left, $k, t, i$, right) →
    if (key < $k$)
    then rebalance (new Inner(insert(left, key, time, item), $k, t, i$, right))
    else rebalance (new Inner(left, $k, t, i$, insert(right, key, time, item)))
  | Leaf → new Inner(Leaf, key, time, item, Leaf)
```

Fig. 4. Insertion into treaps

The second phase allocates new memory as it re-establishes the heap property. This fact and because the function runs through the tree twice (top to bottom for insertion and bottom to top for rebalancing) may seem to make insertion a comparatively expensive operation; however, since the first phase already populates the CPU caches with the nodes needed in the second phase, the overhead is not too large. The performance figure later in this paper quantify the cost of insertion relative to expiration. The amortised cost of insertion is $O(\log n)$ time where n is the number of elements stored in the treap [19]. Additionally, each insertion also allocates $O(\log n)$ memory by producing a new version of the data structure; however, since we use node-copying [4] to implement concurrency rather than provide access to historical versions of the data, memory management automatically reclaims $O(\log n)$ memory per insertion once it is not used by other threads anymore. Thus, for single-threaded applications the overall memory requirements per insertion are not higher than for conventional treaps. In the case of multi-threaded applications, old treap versions are reclaimed as soon the owning thread terminates. For practical workloads, this usually implies that algorithms does not incur a memory overhead.

Removal and Expiration. Like insertion, removal is a two-stage process [18]. The first step consists of locating the node that contains a given key. The second step includes executing rotations so that the node sifts down and eventually becomes a leaf. After this has happened, it is simply discarded. Removal of a key is also exemplified in Fig. 2 (marked (**) in Fig. 2). Like insertion, it is purely functional. By repeatedly calling the deletion function as long as the root node is expired, we can eagerly remove all stale data from the treap. Since the removal algorithm returns a new version of the treap just like insert, the discussion of resource requirements is similar to the discussion of insertion.

Other Operations. Depending on the area of application, other operations on treaps make sense as well. For example, we can use full traversals of a treap to create snapshots of the current state of the database for statistics, billing, etc. Furthermore, if the less-than relationship between keys returns sensible values, range queries on keys can be used to quickly extract ordered intervals from the indexed keys. These operations are implemented exactly as for binary trees, so no code is provided here. However, a performance evaluation of full traversals is presented in the next section.

Concurrency Issues. The tactics used to achieve concurrency is *versioning* [4], implemented by the node-copying method [7]. This implies that each modification to the data structure produces a new version; the previous version can be then garbage-collected once all pointers to it become stale. Therefore, treaps are not ever-growing data structures since only, besides the most recent version, only versions currently in use are kept in memory. Thus, only one thread is allowed to update the data structure, but any number of threads can read from it. As pointed out earlier, this type of design pattern can be implemented in a nearly locking-free manner and provides for concurrent operations at the cost of increased memory allocation and deallocation but not increased overall memory usage. Modern generational garbage collectors [6] are optimised for this kind of allocation pattern and provide favourable performance. Despite the increased memory allocation and deallocation activity the overall storage requirements are asymptotically not higher than traditional single-version implementation (assuming a ‘standard’ database setting with a finite number of threads all of which feature finite running times).

3 Experiments and Evaluation

This section reports on empirical studies of the performance of treaps. We take the following approach: First, we present the formal framework of our evaluation which

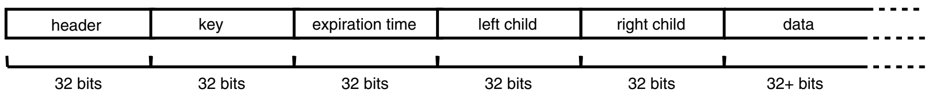


Fig. 5. Physical layout of an internal treap node

allows to reproduce results. Then we examine the performance of treaps for a diverse range of workloads. Lastly, we compare treaps to a number of competitors, including Red-Black trees and AVL trees.

3.1 Experimental Setup

The experiments were carried out on a PC running Gentoo Linux on an Intel Pentium IV processor at 1.5 GHz featuring 512 MB of main memory available; no hard disk was used during the experiments. The CPU caches comprise 8 KB at level 1 and 512 KB at level 2. The compiler used was gcc/g++ 3.2. The performance data from which the graphs displayed in this section were gathered from experiments lasting over 42 hours of runtime on a single machine. Figure 5 displays the physical layout of an internal treap node in our implementation. We fixed the size of the data field to 32 bits for our experiments. All relevant data are inlined, so to access a key or expiration time, we do not have follow a pointer, but we can read it locally in the record. This has been done mainly to improve cache utilisation [1]; in general, however, the data field may contain a pointer to non-local data. In order to explore the full potential and the limitations of treaps, we generated synthetic data to get the data volume needed to test the behaviour of treaps in the limit. The sensor and network hardware available to us are unable to deliver the data volumes necessary to determine the performance limitations of the data structure. Figure 6 plots counts of tuples across time and exemplifies a workload we used.

The dashed line indicates the numbers of tuples that arrive at each particular point in time. For example, the peak at approx. 20,000 milliseconds denotes that 4,000 tuples arrive during the respective interval and have to be inserted into the treap. Without support for expiration time, the network traffic would approximately double, and each spike indicating the arrival of new data would be followed by a spike indicating the deletion of the very data comprising the first spike (assuming that all data expire a fixed duration after their insertion).

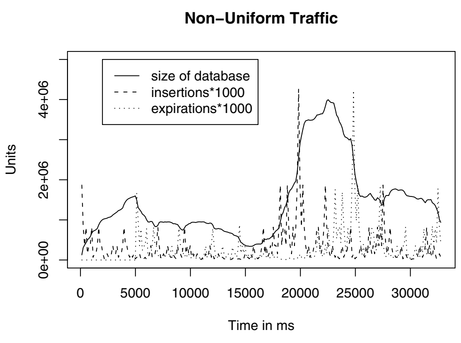


Fig. 6. Database size and operation for non-uniform traffic

We use the B-Model data generator proposed by Wang et al. [22], which is well suited for our purposes. This generator is capable of generating workloads while consuming only a fraction of available system resources. Thus, it provides enough performance not to flaw results. To make it fit our purposes, we extend the generator to work with four input parameters rather than the original three parameters. The three original parameters, b, l, and N, are the bias, the aggregation level, and the total volume, respectively; we refer to this model as $\mathrm{BModel}(b, l, N)$. The bias b describes the roughness of the traffic, i.e. how irregular it is and how pronounced the peaks are. The aggregation level l measures the resolution at which we observe the traffic. The parameter N equals the sum of all measurements and specifies the total amount of traffic. The new, fourth parameter is a random variable describing the distribution of the expiration times of arriving network traffic, i.e. the time interval we consider the $\mathrm{BModel}(b, l, N)$ arriving items valid.

To get an impression of both maximum throughput and response to extremely bursty traffic, we divided our experiments into two parts. We first consider uniform traffic, i.e. $\mathrm{BModel}(0.5, l, N)$. This is done to capture how treaps respond to continuous high workloads. Since the versioning semantics call for frequent allocations of memory, we can expect efficient memory management to be a key factor. Next, we consider $\mathrm{BModel}(b, l, N)$, $b \in]0.5, 1.0[$, i.e. bursty traffic. This is done to estimate how well treaps act under workloads with more or less pronounced peaks. In these settings, minimum and maximum throughput are of interest. Examining treaps in this context is a first step towards the consideration of stochastic quality-of-service guarantees. For experiments which try to illustrate scaling behaviour, N is the parameter used to generate databases of different size. However, when we talk about the size of a database, e.g. about 4 M tuples in Fig. 7(a), we mean the average number of unexpired tuples residing in the database, potentially after some bootstrap.

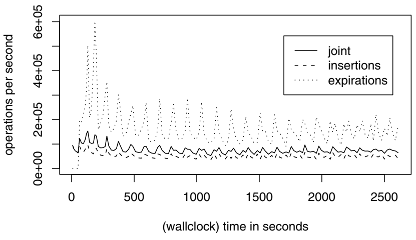
3.2 Discussion of Treap Performance

We now turn our attention to Figs. 7(a) through 8(b), which describe the performance of the data structure under different stress patterns and for different workloads. We first investigate the performance of updates; then we turn our attention to querying.

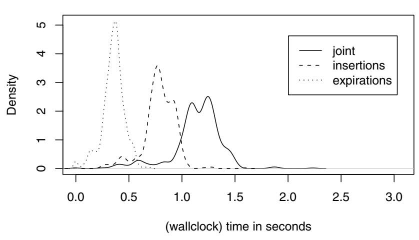
Insertions and Expirations. We first examine the behaviour of treaps under a uniform workload with insertions into a four million tuple database, i.e. after an initial bulk load, the database consists of four million tuples on average, with insertions and expirations basically cancelling out each other.

Figure 7(a) shows the throughput for such a setting. The conspicuous peaks are mainly due to comparatively cheap memory allocation cost after major garbage collections. Notice that the dotted line representing expiration remains, once expirations set in, above the dashed line representing insertions; thus, expirations are cheaper than insertions for large databases. This reflects the structure of the insertion algorithm, requiring to traversal from the root to a leaf and back. On the other hand, expiring the root only requires sifting the node to the leaf level before discarding it. Thus, expirations also require fewer memory allocations than insertions.

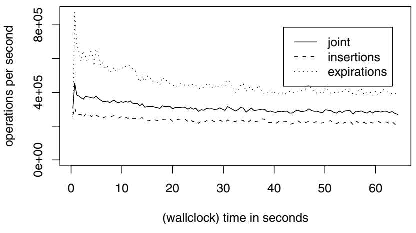
Since treaps only guarantee amortised performance, it is also interesting to learn to what degree the costs of the individual operations differ. Due to the high throughput, which may exceed 100,000 operations per second on our platform, is very hard to monitor the cost of an atomic operation without influencing the result to a degree that



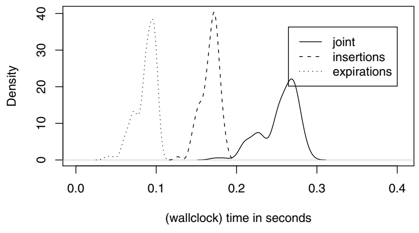
(a) Throughput under uniform traffic ($b = 0.5$), 4 M tuples



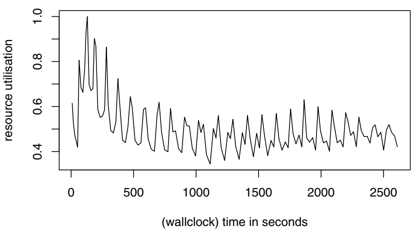
(b) Probability density functions for uniform traffic ($b = 0.5$), 4 M tuples



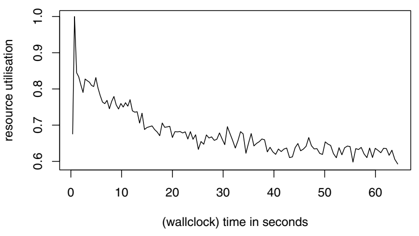
(c) Throughput under uniform traffic ($b = 0.5$), 40,000 tuples



(d) Probability density functions with uniform traffic ($b = 0.5$), 40,000 tuples



(e) Resource utilisation for run displayed in Fig. 7(a)



(f) Resource utilisation for run displayed in Fig. 7(c)

Fig. 7. Performance impressions, resource consumption, and peak performance

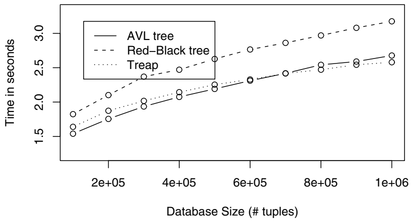
renders it unusable. Therefore, we move to a higher level of aggregation and consider the lengths of intervals containing a fixed number of insertions and expirations. This number was fixed to 80,000 for the experiments. While this rather large number theoretically may obscure the variance in the costs of individual operations, we did not experience this problem and found it a good trade-off between unobtrusiveness and intuition.

In Fig. 7(b), the Probability Density Function (PDF) of such an analysis displayed. We note that the local maximum of the solid line representing expirations at 0.0 indicates that the system wants to expire data, but there is no stale data present. This results in an operation with nearly zero cost. Figure 7(b) also shows that expirations are cheaper than insertions by a factor of approx. two and that the overall cost of a joint operation, insertions and expirations combined, is reasonable. Disregarding the phantom expirations of nearly-zero cost, one can assume that the execution time of 80,000 insertions lies between about 1.0 and 1.4 seconds. This suggests that treaps behave reliably and predictably in practical settings.

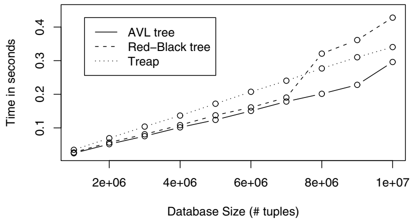
To provide some evidence that the performance peaks in Fig. 7(a) are indeed memory management-related, we concentrate on the results reported in Fig. 7(c). This time, the database contains 40,000 tuples on average. Now, the local maxima in the graph are noticeably less pronounced than in the large database. It turns out that because of the small size, major re-organisations of the storage space can be avoided, keeping the cost of individual memory allocations on about the same level. This is also reflected in the PDF displayed in Fig. 7(d). The bandwidths of insertions, expirations, and of the joint PDF are smaller in both absolute and relative terms. Figure 7(e) and 7(f) concern resource utilisation. It can be seen that most of the time, the treap is able to insert data at about half the maximum possible rate and that memory management causes some pronounced spikes. Again, for smaller databases the spikes remain less pronounced.

Uniform traffic can be considered the worst case for treaps in the sense that it always has to deliver as much performance as possible. In the case of non-uniform traffic, we can expect the system to consume few resources when there are few operations, while running resource-intensively when the numbers of operations peak. This is also demonstrated in Fig. 6. The straight line indicates the database size. Since we use an eager expiration and removal policy, the line also reflects the number of valid, i.e. non-expired, tuples in the database. A note on the choice of the parameter $b = 0.695234$ for non-uniform traffic: we chose b in this range because it is typical for Web traffic [22], a scenario which is probably closest to our area of application.

Retrieval. Concerning retrieval performance, Fig. 8(a) presents the cost profile of uncorrelated lookups while varying the database size. The graph shows that the number of lookups per second decreases as the database size grows and is thus consistent with $O(\log n)$ key lookup complexity. Figure 8(b) illustrates how expensive it is to traverse a treap in an in-order fashion. Traversal is an interesting operation, as it can be used for creating snapshots, computing joins, etc. The operation is linear in the size of the database, but benefits from caching: the path from the treap root the current node is very likely to be resident in the cache hierarchy. Thus, the operation is surprisingly fast; traversing a one million tuple database takes about one-third of a second on our test



(a) Scaling of 100,000 uncorrelated lookups for AVL trees, Red-Black trees, and treaps



(b) Scaling of complete in-order traversal of AVL trees, Red-Black trees, and treaps

Fig. 8. Generated traffic; lookup and traversal in database containing up to 16 M tuples

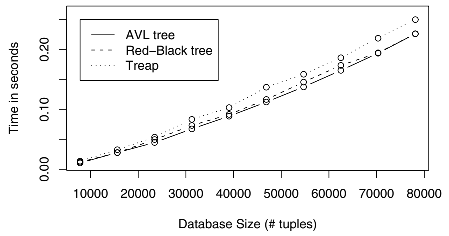
platform. This indicates that the versioning semantics of our treaps does not impede full traversals since the number of arriving tuples is certainly limited. Figure 8 also displays the performance of the same operations on AVL trees and Red-Black trees, which are what we compare treaps against in the following subsection.

3.3 Comparing Treaps to Competitors

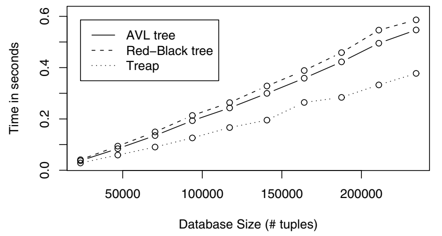
To estimate the performance and resource consumption of the treap index relative to other data structures, we compared the behaviour of treaps to a number of competing approaches. We use the following methodology: Besides requiring appropriate competitors to provide an index on the key attribute, we distinguish between structures which support *eager* expiration and structures which do not. Eager expiration implies that we can remove expired data in a timely fashion from the data structure so that, for example, ON EXPIRATION triggers can fire as soon as the item becomes stale and not at some arbitrary, later point in time. Thus, eager expiration calls for priority queue-like access to the data in addition to the index on the key values. We achieved this by combining the index structures with priority queues. Since our context requires us to work with main-memory data structures, we chose AVL trees and Red-Black trees [11] as competitors to treaps. To support expiration on these structures, we applied (1) periodic cleansing strategies, and (2) priority queue-supported, eager expiration strategies to both data structures. Since treaps may require us to apply a hash function to key values and, thus, may not support range queries under certain circumstances, we also compared the performance of treaps to main-memory hash tables [11]. Again, we use plain hash tables as well as heap-supported hash tables for eager expiration.

Maintenance Costs. To measure how dynamic a database instance is at a given point in time we look at how many tuples of a snapshot would expire during a given interval. Formally, we introduce the notion of *Rate of Expiration* (RoE), which is defined as the ration between expired data and the sum of expired and current data, in a given time interval. Thus, the Rate of Expiration is number between 0 and 1 (or 0% and 100%) which captures how dynamic or how static a particular database state is by relating the number of tuples expirations in a given time interval to the size of the database. Note that the RoE does not take into account insertions and expiration from insertions; it only measures the decay of a database state. An RoE of 100% would imply that, during the interval d, all data expire, whereas an RoE of 0% implies that there are no expirations. We note that expiration time-enabled data structures in general appear particularly useful when RoE is relatively low, i.e. a significant part of the database does not expire in the interval of interest; high RoEs imply that the we have to dispose of large parts of a database, which in turn implies that we have to scan a large part of the data—those to be expired—which we can do anyway without supporting data structures.

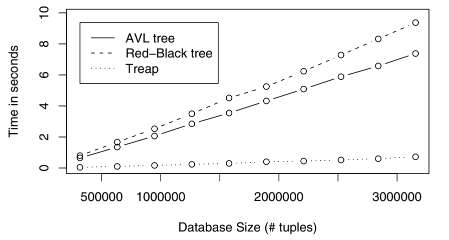
Figure 9 compares the cost of maintaining treaps to the cost of maintaining the other well-known data structures which were adapted to support expiration time. The RoE varies from very dynamic 100% to much more static 1%. It turns out that treaps never perform significantly worse than the other data structures but scale much better, both in terms of memory requirements and processor time, for databases with relatively small



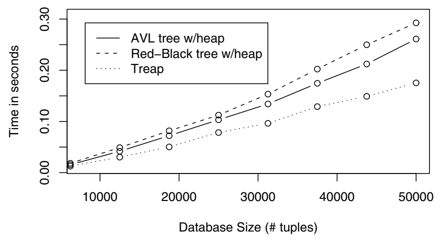
(a) RoE = 100%



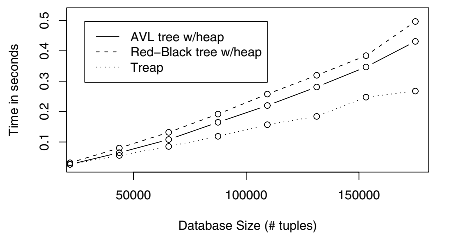
(b) RoE = 5%



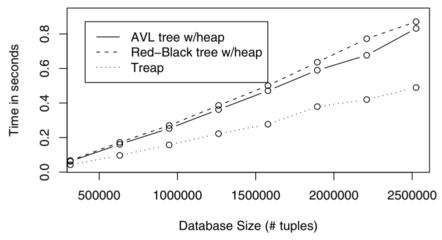
(c) RoE = 1%



(d) RoE = 100%



(e) RoE = 5%



(f) RoE = 1%

Fig. 9. Performance comparison to AVL and Red-Black Trees with/without supporting heaps

Rate of Expiration, which we consider a typical case. In more detail, Figs. 9(a)–9(c) illustrate that treaps outperform AVL trees and Red-Black trees for RoEs of 5% and 1%, whereas they incur only a small overhead for an RoE of 100%. Note that in this case expiration is done on AVL and Red-Black using traversals at the end of each interval, but no additional memory is needed for supporting data structures; thus, expiration is not eager. However, eager expiration can be implemented with supporting heaps as shown in Figs. 9(d)–9(f). Note that in these figures AVL and Red-Black trees are combined with heaps to support eager expiration. However, this incurs a memory overhead for these data structures so that, given a fixed-size main memory, a treap could index more than twice the data size than the competitors. Nevertheless, treaps outperform the other data structures in all cases although not as clearly as in the heap-less experiment.

Query Performance. This subsection considers the question what *query* performance (rather than the cost of maintenance) treaps feature in comparison to AVL and

Red-Black trees. As mentioned earlier, Fig. 8 shows the performance for two important query primitives used frequently in data management: traversals and lookup queries. It turns out that, as Fig. 8(a) shows, treaps consistently outperform Red-Black trees and are *en par* with AVL trees with respect to point queries or lookups. For small databases AVL trees exhibit a slightly better performance where treaps are slightly ahead for larger databases. Similarly, for scanning the data set in sort order, treaps perform slightly worse than both, Red-Black and AVL trees, for small databases; for large databases, they are again ahead of Red-Black trees as Fig. 8(b) shows. However, the important point here is that the probabilistic performance guarantees of treaps do not incur a significant (if at all) penalty on query performance.

Further Issues. The full version of this paper [18] covers several practically relevant issues. Most notably, it relaxes the assumption of statistical independence between key values and expiration times, by introducing a hash function to enforce the independence. Additionally, the full version also compares treaps to expiration time-enabled hash tables and discusses main-memory performance issues. Many other issues are also discussed in somewhat more detail than presented here.

4 Related Work

At the level of query languages and data models, which are not the focus of this paper, the concept of expiration time relates to the concept of vacuuming [9]. With vacuuming, it is possible to specify rules that delete data: when the preconditions, e.g. related to time, in the head of a rule, are met, the data identified by the body of the rule are logically deleted from the database. Like expiration time, vacuuming separates logical deletion from physical deletion. But whereas expiration times are explicitly associated with the tuples in the database, vacuuming specifies which data to delete in separate rules. We believe that the techniques presented in this paper may be relevant for the efficient implementation of time-based vacuuming. Stream databases [2], on the other hand, allow users to specify query windows; in this sense, they take an approach which is opposite to expiration times, which let the data sources declare how long a tuple is to be considered current. Some works that refer to the term “expiration” are slightly related to expiration time and thus this paper’s contribution. Expiration has been used in the context of view self-maintenance: Here the problem is which data that can be removed (“expired”) without this affecting the results of a predetermined set of queries (views) [8].

The use of expiration time has been studied in the context of supporting moving objects [21]. The idea is that locations reported by moving objects that have not been updated explicitly for some time are considered inaccurate and should thus be expired. The R^{EXP}-tree extends the R-tree to index the current and anticipated future positions of two- and three-dimensional points, where the points are given by linear functions of time and are associated with expiration times. We are not aware of any related research on main memory based indexing that incorporates expiration time.

Okasaki [16] offers a very readable introduction to purely functional data structures. Our primary data structure, the (*functional*) *treap*, is described and analysed in substantial detail by Seidel and Aragon [19]; it was first introduced by McCreight [14]. Later, treaps were primarily seen and interpreted as randomised search trees [19]. Treaps have been used in a number of contexts; however, we are not aware of any time-related applications. Heaps are a classical data structure in computer science [11]. In this paper, we technically achieve concurrency on functional treaps through versioning [4] by implementing the node-copying method [7]. In a database context, Lomet and Salzberg [13] present versioning supporting variants of B-trees and discuss related issues. Finally, we remark that distributed garbage collection also shares similarities with expiring data in databases; especially eager collection is sensible when scarce resources have to be freed up.

5 Conclusion and Future Work

This paper argues that expiration time is an important concept for data management in a variety of application areas, including heartbeat patterns in mobile networks and short-lived data. It presents a functional, or versioned, variant of the previously proposed treap along with algorithms for supporting data with expiration time, and it argues that this is an efficient main-memory index for data with expiration times. Through comprehensive and comparative performance experiments, the paper demonstrates that its proposal scales well beyond data volumes produced by current mobile applications and thus is suited for advanced applications.

Data expiration is an important and natural concept in many volatile application settings where traditional ACID semantics are not appropriate. Often, devices such as mobile phones, PDAs, sensors, and RFID tags experience intermittent connectivity, but also do not need a full-blown transaction system for many tasks. In these settings, data management applications can benefit from the underlying platform being expiration time-enabled. Benefits include lower transaction workloads, reduced network traffic, and the ability to free memory occupied by stale data immediately.

Support in the underlying platform for expiration time also have the potential of simplifying application logic by removing the need for "clean-up" transactions. The paper demonstrates through experiments that a functional treap, which is a binary tree with respect to a key and a heap with respect to the expiration time, is an effective tool for handling expiration times in main-memory settings.

Several interesting directions for future research exist in relation to the support for expiration time in data management. When data are not as short-lived as assumed in this paper, it might be beneficial to develop strategies for extending standard secondary-memory data structures, e.g. heap files, B-trees, and hash files, with expiration time support. We anticipate that expiration for secondary-memory structures requires strategies different from those presented in this paper. Furthermore, to take full advantage of database management system technology, expiration times have to be sensibly integrated into SQL's isolation levels and transaction system.

Acknowledgments. The authors would like to thank Peter Schartner for helpful comments and pointers as well as Laurynas Speicys, Simonas Šaltenis, and Kristian Torp for helpful discussions and their collaboration. C. S. Jensen is also an adjunct professor in Department of Technology, Agder University College, Norway.

References

1. Ailamaki, A., DeWitt, D., Hill, M.: Data page layouts for relational databases on deep memory hierarchies. The VLDB Journal **11** (2002) 198–215
2. Arasu, A., Babu, S., Widom, J.: CQL: A Language for Continuous Queries over Streams and Relations. Proc. DBPL (2003) 1–19
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison Wesley, (2003)
4. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley (1987)
5. Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Comm. ACM **13** (1970) 377–387
6. Diwan, A., Tarditi, D., Moss, J.: Memory System Performance of Programs with Intensive Heap Allocation. ACM TOCS **13** (1995) 244–273
7. Driscoll, J., Sarnak, N., Sleator, D., Tarjan, R.: Making Data Structures Persistent. Journal of Computer and System Sciences **38** (1989) 86–124
8. Garcia-Molina, H., Labio, W., Yang, J.: Expiring Data in a Warehouse. Proc. VLDB (1998) 500–511
9. Jensen, C.§.: Vacuuming. The TSQL2 Temporal Query Language (1995) 447–460
10. Jensen, C. S., Lomet, D.: Transaction Timestamping in (Temporal) Databases. Proc. VLDB (2001) 441–450
11. Knuth, D.: The Art of Computer Programming, vol. 3, Sorting and Searching. Addison Wesley (1998)
12. Lehman, T., Carey, M.: Query Processing in Main Memory Database Management Systems. Proc. ACM SIGMOD (1986) 239–250
13. Lomet, D., Salzberg, B.: Access Methods for Multiversion Data. Proc. ACM SIGMOD (1989) 315–324
14. McCreight, E.: Priority Search Trees. SIAM Journal on Computing **14** (1985) 257–276
15. Odersky, M., *et al.*: The Scala Programming Language. http://scala.epfl.ch (2005)
16. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
17. Schmidt, A., Jensen, C. S., Šaltenis, S.: Expiration Times for Data Management. IEEE ICDE (2006, to appear)
18. Schmidt, A., Jensen, C. S.: Efficient Management of Short-Lived Data. Technical Report (2005) http://arxiv.org/abs/cs.DB/0505038
19. R. Seidel and C. Aragon. Randomized Search Trees. *Algorithmica*, 16(4/5): 464–497, 1996.
20. The World Wide Web Consortium. HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/ (2005)
21. Šaltenis, S., Jensen, C. S.: Indexing of Moving Objects for Location-Based Services. Proc. IEEE ICDE (2002) 463–472
22. Wang, M., Chan, N., Papadimitriou, S., Faloutsos, C., Madhyastha, T.: Data Mining Meets Performance Evaluation: Fast Algorithms for Modeling Bursty Traffic. Proc. IEEE ICDE (2002) 507–516

References containing URLs are valid as of 6 December 2005.