Design, Implementation, and Performance of the LHAM Log-Structured History Data Access Method

Peter Muth
Dept. of Comp. Sc., Univ. of the Saarland
D-66041 Saarbruecken, Germany,
muth@cs.uni-sb.de

Patrick O'Neil
Dept. of Math. and Comp. Sc.,
UMASS–Boston, Boston MA, 02125–3393
poneil@cs.umb.edu

Achim Pick, Gerhard Weikum,
Dept. of Comp. Sc., Univ. of the Saarland
D-66041 Saarbruecken, Germany,
{pick, weikum}@cs.uni-sb.de

Abstract

Numerous applications such as stock market or medical information systems require that both historical and current data be logically integrated into a temporal database. The underlying access method must support different forms of "time-travel" queries, the migration of old record versions onto inexpensive archive media, and high insert and update rates. This paper introduces a new access method for transaction-time temporal data, called the Log-structured History Data Access Method (LHAM) that meets these demands. The basic principle of LHAM is to partition the data into successive components based on the timestamps of the record versions. Components are assigned to different levels of a storage hierarchy, and incoming data is continuously migrated through the hierarchy. The paper discusses the LHAM concepts, including concurrency control and recovery, our full-fledged LHAM implementation, and experimental performance results based on this implementation. A detailed comparison with the TSB-tree, both analytically and based on experiments with real implementations, shows that LHAM is highly superior in terms of insert performance while query performance is in almost all cases at least as good as for the TSB-tree; in many cases it is much better.

1 Introduction

For many applications maintaining only current information is not sufficient; rather, historical data must be kept to answer all relevant queries. Such applications include, for example, stock market information systems, risk assessment in banking, medical information systems, and scientific database applications. Temporal database systems [Sno90, Tan93] aim to support this kind of applications. In this paper, we consider a special type of temporal databases, namely, *transaction-time databases*, where multiple versions of a record are kept. Updating a record is implemented by inserting a new record version. Each record version is timestamped with the commit-time of the transaction that updated the record. The timestamp is considered to be the start time for a record version. The end time is implicitly given by the start time of the next version of the same record, if one exists. Records are never physically deleted; a logical deletion is implemented by creating a special record version that marks the end of the record's lifetime.

Indexing temporal databases is an important and challenging problem, mainly because of the huge amount of data to be indexed and the various "time-travel" types of queries that have to be supported. An equally important requirement is an access method's ability to sustain high insert/update rates. This requirement arises, for example, in data warehouses, in scientific databases that are fed by automatic instruments, or in workflow management systems for keeping workflow histories. Also, many banking and stock market applications exhibit such characteristics. For example, consider the management of stock portfolios in a large bank. For each portfolio, all buy and sell orders must be tracked. Based on this data, in addition to querying the current contents of a portfolio, queries asking for the history of a specific portfolio in a given time interval as well as queries asking for statistical data over certain portfolios can be supported. The results of these queries are important for future decisions on buying or selling stocks.

To keep the further explanation simple, assume that placing a sell or buy order is tracked by inserting a record version in a portfolio-history table. Assuming 1000 orders per second, we have 1000 inserts into the history table per second. Further assume, we want to index the history table by using a B^+-tree on the customer ID, and we want to keep the history of the last 7 days online. Given 24 business hours for worldwide orders per day and records of 48 Bytes, we have about 28 GB of index data. This translates into 3.5 million blocks, 8KB each, at the leaf level of the B^+-tree. Assuming, for simplicity, that orders are uniformly distributed among portfolios, repeated references to the same block are on average 3,500 seconds $\approx$ 1 hour apart. According to the five-minute rule [GP87], this does not justify main memory residence. As a consequence, it is highly unlikely that an insert operation finds the leaf node that it accesses in the buffer. Instead, inserting a new record causes two I/Os on the leaf level of the B^+-tree, one for writing some leaf node back to the database in order to free buffer space, and one for bringing the leaf node where the new record version is to be inserted into the buffer. Given 1000 inserts per second, we have 2000 I/Os per second, disregarding splits and the higher levels of the tree. Optimistically assuming that a single disk can serve 100 I/Os per second, we need 20 disks to sustain the insert rate of the application, but the data fits on two disks.

The above arguments hold for all index structures that place incoming data immediately at a final position on disk. The log-structured history access method LHAM, introduced in this paper, addresses this problem by initially storing all incoming data in a main memory component. When the main memory component becomes full, the data is merged with data already on disk and migrated to disk in a bulk fashion, similar to the log-structured file system approach [RO92] - hence the name of our method. At the same time, a new index structure on disk, containing both the new and the old records, is created. All I/O operations use fast multi-block I/O. In general, components may exist on different levels of a storage hierarchy. If a component becomes full, data is migrated to the component on the next lower level. This basic approach has been adopted from the LSM-tree method [OCGO96], a conventional (i.e., non-temporal) single-key access method. An analysis of LHAM as well as experimental results gained from our implementation show that LHAM saves a substantial amount of I/Os on inserts and updates.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

For the above example, an LHAM structure with a main memory component of 144MB and two disk components with a total size of 30GB is sufficient. This translates into two disks for LHAM, in contrast to 20 disks if a B$^+$-tree-like access method were used.

The basic idea of an earlier form of LHAM has been sketched in [OW93]. The current paper presents the comprehensive design of a full-fledged access method and its implementation. The contribution of this paper is threefold:

- We give a detailed presentation of the LHAM concepts, including a discussion of synchronization issues between concurrent migration processes, called *rolling merges*, and transactional concurrency control and recovery. The performance of inserts in terms of required block accesses is mathematically analyzed.
- We present a full-fledged LHAM implementation for shared-memory multiprocessors using the Solaris thread library. The entire prototype comprises 24,000 lines of C code (including monitoring tools) and has been stress-tested over several months.
- To validate the analytic results on insert performance and to evaluate the query performance of LHAM, we have measured LHAM's performance against the TSB-tree, which is among the currently best known access methods for temporal data. We present detailed experimental results in terms of required block accesses and throughput for different insert/update loads, different query types, and different LHAM configurations. Our results provide valuable insight into the typical performance of both access structures for real life applications, as opposed to asymptotic worst-case efficiency.

The paper is organized as follows. Section 2 discusses related work. Section 3 presents the principles of LHAM in terms of time partitioning the data, data migration, and query processing. In Section 4, we discuss the implementation of LHAM, its internal architecture, rolling merge processes for data migration, and the synchronization of these processes. Concurrency control and recovery are discussed in Section 5. Section 6 contains the results of our experimental performance evaluation. We compare the experimental results for our implementations of LHAM and the TSB-tree in detail. Section 7 concludes the paper. Appendix A briefly introduces the TSB-tree.

2 Related Work

As for "time-travel" queries, LHAM supports exact match queries as well as range queries on key, time, and the combination of key and time. Temporal index structures with this scope include the TSB-tree [LS89, LS90], the MVBT [Bec96], the Two-Level Time Index [EWK93], the R-tree [Gut84], and the Segment-R-tree[Kol93], a variant of the R-tree specifically suited for temporal databases. Temporal index structures like the Snapshot Index [TK95], the Time Index [EWK93, EKW91] and the TP-Index [SOL94] aim only at supporting specific query types efficiently. Comparing them with other index structures is only meaningful based on a specific kind of application. Among the index structures with a general aim, the TSB-tree has demonstrated very good query performance [ST94]. Therefore, we have chosen the TSB-tree as the yardstick against which LHAM is compared. In terms of asymptotic worst-case query performance, the TSB-tree guarantees logarithmic efficiency for all query types whereas LHAM is susceptible to degradation under specifically constructed "adversary scenarios". However, such degradation is extremely unlikely under realistic scenarios as our systematic performance study shows. For almost all query types, the realistic performance of LHAM is at least as good as for the TSB-tree, for many cases even substantially better because of better data clustering and potential for multi-block I/O.

Most proposals for index structures on temporal data are not specifically targeted at a good insert performance, exceptions being [Jag97] and [BSW97]. [Jag97] discusses an approach for efficient insertion of non-temporal, single-dimensional data into B$^+$-trees. Similar to LHAM, a continuous reorganization of data is proposed. The approach can be characterized as a generalization of an N-way merge sort. Like in LHAM, incoming data is stored in a main memory buffer first. When this buffer gets full, it is written to disk, organized as a B$^+$-tree (a hash-based scheme is also discussed). After K such trees are created, they are merged by a K-way merge into a new B$^+$-tree. N-1 levels, each consisting of K B$^+$-trees, are considered. Each K-way merge propagates the data to the next level. The final level N contains the target B$^+$-tree, denoted *root B$^+$-tree*. After K B$^+$-trees have been accumulated at level N-1, these K trees together with the root tree are merged in a K+1-way merge into a new root tree. This approach supports the efficient insertion of data, but penalizes queries significantly, as a query has too look up all $N*K$ component trees.

In terms of this approach, LHAM can be considered to perform a 2-way merge sort whenever data is migrated to the next of n components in the LHAM storage hierarchy. At each level of this hierarchy, only a single B$^+$-tree exists (unless a merge is currently performed, which creates temporary trees). In contrast to [Jag97], LHAM components implement a partitioning of the time dimension. In [Jag97], all $N*K$ B$^+$-trees may have overlapping key ranges, whereas components in LHAM cover disjoint areas in key-time space (with a single exception to make archiving more efficient, an issue which is not addressed at all in [Jag97]). Depending on its time range, a query in LHAM needs to access only a subset of the n LHAM components, so that query execution in LHAM is more efficient.

[BSW97] presents an approach for bulk-loading multi-dimensional index structures, e.g., R-trees. Their approach can be considered as the "opposite" of an N-way merge sort. The idea is to create unsorted sequences of records, where each sequence covers a subset of the dataspace that is disjoint to the subsets covered by the other sequences. A number of sets of record sequences form a balanced tree, called the *buffer tree*, with each set of sequences defining a level of the tree. Incoming records are migrated through this tree until they reach the leaf level, which is structurally equivalent to the leaf level of the target index structure. Similarly to LHAM and [Jag97], the migration of records can be implemented very efficiently. After all data has been migrated to the leaf level of the buffer tree, its higher levels are discarded, and a new buffer tree for building the next higher index level of the target index structure is created. This process is repeated until the root of the target index structure is built. As the approach of [BSW97] is intended for bulk loading only, it disregards query performance until the target index structure is completely built. The problem is to search the potentially large unordered sequences of records at the index levels of the buffer tree. While a query can easily navigate through the buffer tree, the sequential search inside a record sequence is expensive. LHAM provides a substantially better query performance.

3 Principles of LHAM

LHAM is an index structure for transaction-time databases. It indexes record versions in two dimensions; one dimension is given by the conventional record key, the other by the timestamp of the record version. A record version gets its timestamp at the time of insertion as the transaction time of the inserting transaction. The timestamp cannot be changed afterwards. Updating a record is cast into inserting a new version. Deleting a record is performed by inserting a new record version indicating the deletion. As a conse-

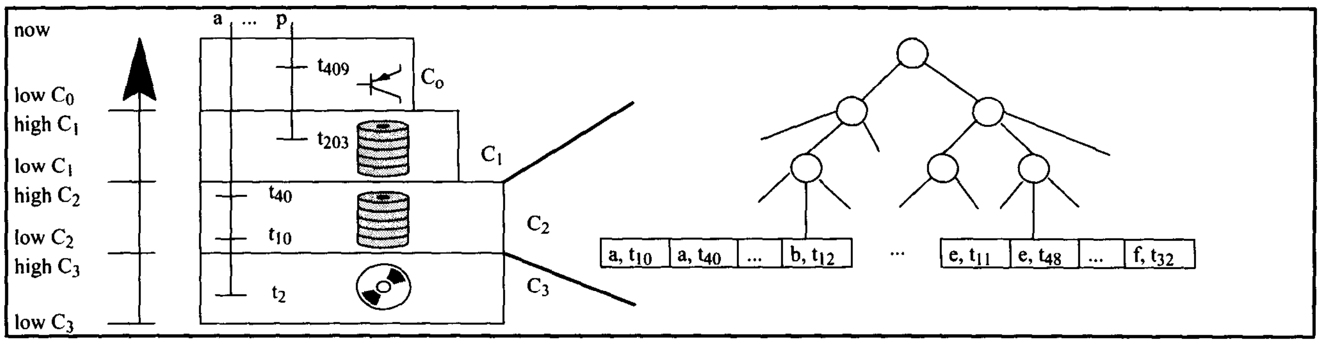

Fig. 1: LHAM Component Organization And B$^+$-tree Inside Component

quence, all insert, update, and delete operations are performed by inserting record versions.

Unlike virtually all previously proposed index structures, LHAM aims to support extremely high insert rates that lead to a large number of newly created record versions per time unit. Furthermore, while many other temporal index structures emphasize the efficiency of exact-match queries and range queries for either key or time, LHAM aims to support exact-match queries as well as all types of range queries on key, on time, and on the combination of both. Note that there is actually a tradeoff in the performance of time range queries versus key range queries, as the first query type benefits from clustering by time whereas the latter benefits from clustering by key. LHAM strives for a flexible compromise with respect to this tradeoff.

3.1 Partitioning the Time Dimension

The basic idea of LHAM is to divide the entire time domain into successive intervals and to assign each interval to a separate storage *component*. The series of components, denoted as C_0, C_1, ..., C_n, constitutes a partitioning of the history data based on the timestamp attribute of the record versions. A component C_i contains all record versions with timestamps that fall between a low-time boundary, low_i, and a high-time boundary, $high_i$, where $high_i$ is more recent than low_i. For successive components C_i, and C_{i+1}, components with lower subscripts contain more recent data, so low_i is equal to $high_{i+1}$. Component C_0 is stored in main memory and contains the most recent record versions from the current moment (which we take to be $high_0$), back to time low_0. Components C_1 through C_k reside on disk, and the rest of the components C_{k+1}, ..., C_n are stable archive components that can be stored on write-once or slow media (e.g., optical disks). Typically, the number k of disk components will be relatively small (between 1 and 3), whereas the number n-k of archive components may be large, but archive components will probably consist of a month worth of record change archives.

The overall organization of LHAM is depicted in the left part of Fig. 1. In the example, the history of two records is shown. The record with key a has been inserted at time t_2, and was updated at times t_{10} and t_{40}. Its original version as of time t_2 has migrated to archive component C_3, the other record versions are currently stored in disk component C_2. The record with key p has been inserted at time t_{203}, which now falls into the time interval covered by component C_1. Record p has a recent update at time t_{409}, the corresponding record version is still in main memory component C_0.

Inside each component, record versions are organized by a conventional index structure for query efficiency. In principle, every index structure that supports the required query types and efficiently allows the insertion of record versions in batches can be used. Different index structures can be used for different components. For the sake of simplicity, we have chosen B$^+$-trees for all components. An example B$^+$-tree is shown in the right part of Fig. 1, containing the record versions of record a at times t_{10} and t_{40}. The key of the B$^+$-tree is formed by concatenating the conventional record key and the timestamp of a record version. Therefore, the record versions are ordered according to their conventional key first, followed by their timestamp. Using this ordering is a drawback for time range queries, as record versions are clustered primarily by key. However, this drawback is typically compensated by partitioning data by time according to the component time intervals. We will consider other index structures inside of LHAM components in the future. A key requirement is their ability to support bulk loading of data.

The purpose of the more recent disk components, and especially the main memory component, is to support high insert rates. Inserting a new record version into the main memory component does not take any I/O (other than logging for recovery purposes, which is necessary anyway, see Section 5 for details). I/O is needed when a component becomes full. In this case, data is migrated to the next component. Providing a highly efficient migration by moving data in batches is the key to LHAM's good performance.

In the "standard" variant of LHAM, there is no redundancy among components. A record version is stored in the component whose low and high time boundaries include the version's timestamp. However, some versions are valid beyond the high time boundary of the component, namely, when the next more recent version for the same record key is created after the component's high time boundary. Especially for long-lived versions, it can be beneficial for query performance to keep such a version redundantly in more than one component. Redundancy is especially attractive for the usually much slower archive components. LHAM supports both redundancy-free and redundant partitioning.

3.2 Inserts and Migration of Data

Newly created record versions are always inserted into the main memory component C_0, consisting of a 2-3- tree or similar memory based key-lookup structure. They eventually migrate through disk components C_1 ... C_k, consisting of B$^+$-tree-like structures, and eventually arrive on archive media. There is no migration among archive components, as these are often write-once or too slow for data reorganizations. However, record versions reaching an age where they are no longer of interest may occasionally be purged from the on-line archive storage. This can be achieved easily with LHAM, because of the time boundaries between components, and the natural placement of components one after another on archive media such as optical disk platters.

The data migration from more recent to older components is accomplished by a process denoted *rolling merge*, following the idea of the LSM-tree [OCGO96], a log-structured access method for conventional, one-dimensional key access. For each pair of successive components C_i and C_{i+1}, $i < k$, a rolling merge process, denoted $RM_{i/i+1}$, is invoked each time component C_i becomes full. Its invocation frequency depends on how often the amount of data in C_i reaches a maximum triggering size. When the rolling merge process starts, a migration boundary m_i is chosen, that will become the new time boundary between C_i and C_{i+1} after the rolling merge is finished. The appropriate value for m_i, relative to low_i and $high_i$, depends on the growth rate of C_i and, thus, (by recursion) ultimately on the insert rate of the database. The rolling merge process $RM_{i/i+1}$ scans the leaf nodes of the tree in component C_i in order of key and timestamp, and migrates all record versions of C_i that have a timestamp smaller than m_i into component C_{i+1}, building a new tree there. It terminates when C_i is completely scanned, and at this point, low_i and $high_{i+1}$ are both set to m_i.

The rolling merge from the oldest disk component C_k does not really merge data into component C_{k+1}. Rather, this migration process builds up a complete, new archive component. This new archive component is then called C_{k+1}, and the previous archive components C_{k+1} through C_n are renumbered into C_{k+2} through C_{n+1}. As access to archive components is typically very slow, we choose to use the partitioning scheme with redundancy when deciding which versions are moved into the component. So an archive component contains all record versions whose validity interval overlaps with the component's time interval given by its low and high time boundaries. Note that in this case a new archive component C_{k+1} may contain versions that already exist in C_{k+2} and possibly older archive components, if these versions are still valid after the low-time boundary low_{k+1} of the new archive component (which is equal to the old low_k value). This scheme makes the archive components "self-contained" in that all queries with timestamps between the component's low and high boundary can be performed solely on a single component. Also, when an archive component is taken off-line, the redundancy ensures that all versions that remain valid beyond the high time boundary of the off-line component are still available in the appropriate on-line component(s). As archive components are built by the rolling merge from component C_k to component C_{k+1}, C_k has to store all redundant record versions needed for creating the next archive component. Redundant record versions in C_k need not be accessed by $RM_{k-1,k}$, as redundant versions are only created and accessed by $RM_{k,k+1}$ when a new archive component is built. Hence, only for $RM_{k,k+1}$ additional I/O is required to read and write the redundant versions in C_k. In the analysis of insert costs in the next section, we will see that the overhead of redundancy in terms of additional space and I/O is typically low.

Rolling merges avoid random disk accesses that would arise with moving record versions one-at-a-time. Rather, to achieve good I/O efficiency in maintaining the internal component index structure (i.e., B^+-trees in our case), a rolling merge reads both the source and the destination component sequentially in large multi-block I/Os, and the data from both components is merged to build a new index structure in the destination component again written sequentially in large multi-block I/Os. With multi-block I/O, instead of reading and writing single blocks, multiple contiguous blocks on disk are read and written in a single I/O operation, which is significantly faster than performing single random I/Os (see Section 6). The total number of block accesses required for a merge process is the same as for scanning both components two times. Contrast this with the much higher number of much slower random disk I/Os for migrating record versions one-at-a-time. In addition, our algorithm allows us to keep the data perfectly clustered all the time, with almost 100% node utilization, which in turn benefits range queries and index scans.

3.3 Analysis of Insert Costs

We derive a formula for the cost of inserting new record versions into LHAM in terms of the number of block accesses required. The formula implies that for minimizing the block accesses required, the space-capacity ratios should be the same for all pairs of successive components. This leads to a geometric progression between the smallest component C_0 and the largest disk component C_k. All archive components are assumed to have the capacity of C_k, which allows the migration of all record versions stored in C_k to an archive component in a single rolling merge. When record versions are stored redundantly (see Section 3.1), the capacity of C_k must be larger than defined by the geometric progression. In the worst case, one version of each record stored in LHAM has to be kept in C_k. However, with an average of s record versions per record residing in all non-archive components together, the space overhead for storing one redundant version per record in C_k is $1/s$ times the total size of the non-archive components. As a typical temporal database is expected to store more than a few versions per record, this is a small space overhead.

We derive the number of block accesses required to insert a given amount of data into LHAM by counting the block accesses needed to migrate the data through the components of LHAM. This approach is similar to the one presented for the LSM-tree [OCGO96]. However, in [OCGO96], the authors idealistically assumed a perfect steady-state balance in that the insertion rate in bytes per second matches the migration rate between all LSM components at any time. As a consequence, the actual filling degree of each component is constant and close to 100 percent all the time.

This assumption is unrealistic in practice because of fluctuations in the rate of incoming data. Also it is hard to keep the migration rate of the rolling merges truly constant, as the disk(s) typically have to serve additional, often bursty load like concurrent queries. So in a realistic environment, a rolling merge cannot be assumed to start again immediately after it finishes its previous invocation. Instead, rolling merges should be considered as reorganization events with a varying frequency of occurrence. This leads to actual component sizes (i.e., filling degrees) that vary over time. Immediately after a rolling merge has migrated data from a component C_i to component C_{i+1}, C_i will be almost empty. After sufficiently many rolling merges from C_{i-1} to C_i, component C_i will then become (close to) full again before the next rolling merge from C_i to C_{i+1} is initiated. So, if we merely assume that the time points of initiating the filling rolling merges from C_{i-1} to C_i are uniformly distributed over time, then C_i is on average half full. Thus, a "randomly arriving" rolling merge from C_{i-1} to C_i needs to merge the C_{i-1} data with a 50 percent full C_i component on average. This consideration is fully confirmed by our experimental findings in Section 6.

As all data is inserted into main memory component C_0 first, and as all rolling merges access data in terms of complete disk blocks instead of single record versions, the total number of block accesses depends only on the number of blocks required to store the data, not on the number of records stored in the database. As usual, we disregard the non-leaf levels of the B^+-trees here. Assume all record versions fit on $block_{tot}$ leaf nodes, including space-fragmentation overhead. We assume a LHAM structure with k components on disk, a component size ratio of r_i between components C_{i-1} and C_i, r_0 being the size of component C_0 in blocks. Let l_i denote the number of rolling merges taking place between components C_{i-1} and C_i until all data is inserted and finally migrated to archive component C_{k+1}, and let $1/s$ be the space overhead for redundancy in component C_k with respect to the total size of all non archive components C_0 ... C_k. We obtain for the total number of accessed blocks $block_{access}$:

$$block_{access} = l_1(r_0 + r_0r_1) + l_2(2r_0r_1 + r_0r_1r_2) + \ldots + l_k\Big(2\prod_{i=0}^{k-1} r_i + \prod_{i=0}^{k} r_i\Big) + l_{k+1}\Big(2\prod_{i=0}^{k} r_i\Big) + l_{k+1}\Big(\frac{2}{s}\sum_{j=0}^{k}\prod_{i=0}^{j} r_i\Big) \quad (1)$$

Note that for emptying component C_0, no I/O is required, which leads to the term $(r_0 + r_0r_1)$ rather than $(2r_0 + r_0r_1)$. For the final migration to archive media, the data is only read from component C_k and written to the archive component. The last term represents the block accesses needed to read and write the redundant record versions in C_k. As discussed in section 3.2, redundant records of C_k are not accessed by $RM_{k-1,k}$. The number l_i of rolling merges taking place between components C_{i-1} and C_i is given by:

$$l_i = \frac{block_{tot}}{\prod_{j=0}^{i-1} r_j} \qquad (2)$$

By substituting (2) into (1), we obtain:

$$block_{accesses} = block_{tot}(2k + 1 + \sum_{i=1}^{k} r_i) + l_{k+1}(\frac{2}{s} \sum_{j=0}^{k} \prod_{i=0}^{j} r_i) \qquad (3)$$

In order to tune the component capacity ratios r_i, we adopt the procedure of [OCGO96]. For the sake of simplicity, we assume the redundancy overhead to be constant instead of depending on the component size ratios, and assume that the main memory available for component C_0 and the size of the last disk component C_k are already fixed. [OCGO96] shows that under these assumptions, the number of blocks accessed is minimized if all component size ratios r_i are equal to a constant value r. Substituting all r_i of equation (3) by r, we obtain:

$$block_{accesses} = block_{tot}(k\,(2 + r) + 1) + l_{k+1}(\frac{2}{s} \sum_{j=0}^{k} r_{j+1}) \qquad (4)$$

For a component size ratio r of at least two, the number of block accesses is bounded by:

$$block_{accesses} \leq block_{tot}(k\,(2 + r) + 1 + \frac{4}{s}) \qquad (5)$$

As an example, consider again the stock portfolio scenario presented in the introduction. We assume the insertion of 604,800,000 record versions of 48 Bytes each into LHAM, representing a constant insertion rate of 1000 record versions per second over a 7 day period, and a total size of 28GB of data. Assume that we use two disk components. Main memory component C_0 has a size of 144MB, C_1 has 2GB and C_2 has 28GB. This translates into a component size ratio of 14. Assuming the placement of two orders for each portfolio per day on average, we obtain an overhead ratio for storing redundant data in component C_k of $\frac{1}{2 * 7}$. As we have about 31GB of data online, this leads to an additional space requirement of 2.2GB for redundant data on disk. With about 3,500,000 blocks of 8KB size to insert, according to equation (5), we need less than 115,900,000 block accesses for data migration, including 1,000,000 block accesses for redundant data, which is obviously negligible. Note that these numbers represent the number of block accesses needed to insert record versions into an already fully populated database, i.e., containing the data of the past days. Inserts into an empty database would cause even less block accesses. With a TSB-tree, on the other hand, we estimate 1,209,600,000 block accesses for inserting 604,800,000 record versions, 2 block accesses per insert. So the cost of the TSB-tree is more than ten times higher than the cost of LHAM, not even considering the additional gain from LHAM's multi-block I/O.

3.4 Query Processing

In general, query processing may require searching multiple components. LHAM maintains a (small) global directory of the low-time and high-time boundaries of all components, and keeps track of the number n of the last archive component. The directory is used to determine the relevant components that must be searched for queries.

For "time-travel" queries with a specified timepoint or time range, LHAM needs to retrieve all record versions that are valid at this point or within this time range, respectively. A record version resides in the component whose time-range covers its creation timestamp, but the version may also be valid in more recent components. Thus, LHAM must possibly search components later in time than the query specifies. Because of the size progression of components and their placement in the storage hierarchy, the search starts with the most recent component that could possibly hold a query match and proceeds along the older components until no more matches are possible (in a "time-travel" query for a specified key value) or all components have been searched (if none of the components uses redundancy) . For example, with the data of Fig. 1, the query "Select ... Where KEY = 'a' As Of t_{203}" has to search the disk components C_1 and C_2. Similar considerations hold for range queries. The redundant partitioning option (see Section 3.1) allows us to bound the set of components that must be searched. In the concrete LHAM configuration considered here with the redundancy option used for the last disk component C_k, queries with a time range not overlapping the time interval of archive components need not access archive components.

Having to search multiple components may appear as a heavy penalty from a superficial viewpoint. In practice, however, we would have only a small number of non-archive components, say three or four, one of which is the main memory component. Our experiments show that the absolute query performance of LHAM is very competitive even when multiple components need to be searched (see Section 6.1.2).

For searching within a component, the component's internal index structure is used. When using a B$^+$-tree on the concatenation of record key and version timestamp, exact-match queries can be answered with logarithmic performance. Time range queries for a given key are also efficiently supported, as all versions of a record are clustered by time. On the other hand, key-range queries with a given timepoint or a small time range are penalized with the chosen B$^+$-tree organization. However, even this query type does not perform too badly, since our approach of building the B$^+$-trees only by rolling merges provides relatively good clustering by key also. If there are only a few record versions per key, we may still be able to process the query with a few block accesses or even less than a single block access per key. In addition, the clustering again allows us to use multi-block I/O, which is not possible in most other indexing methods for temporal data as they do not cluster the data accordingly.

4 Implementation of LHAM

4.1 System Architecture

LHAM has been fully implemented in C on SUN Solaris. As the rolling merges between different components can be executed in parallel, but need careful synchronization to guarantee consistency of data, we have decided to implement them as Solaris threads. Threads communicate by shared variables and are synchronized by semaphores of the thread library. Fig. 2 shows the overall LHAM architecture. Each rolling merge is implemented by four threads as indicated by the small shaded boxes in Fig. 2 and explained in detail in the next subsection. Queries are implemented by separate threads for each component that is accessed. An additional thread performs the insertion of new data into component C_0.

Data read from disk is cached by LHAM in two kinds of buffers. Single-block buffers cache index nodes of B$^+$-trees and leaf nodes if read by single-block I/Os, i.e., by queries. For leaf nodes of B$^+$-trees accessed by rolling merges or by range queries, multi-block buffers are read and written by multi-block I/Os. The buffer replacement strategy for both buffers is LRU.

4.2 Inserts and Rolling Merges

Figure 3 shows two components C_i and C_{i+1}, with a rolling merge currently migrating data from C_i to C_{i+1}. During an ongoing rolling merge, both the source and the destination component consist of

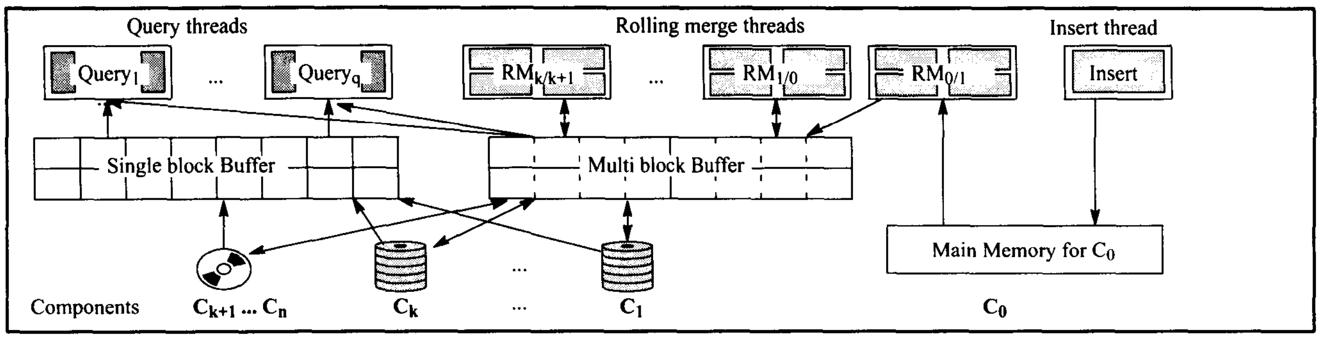

Fig. 2: LHAM Architecture

two B$^+$-trees, an *emptying tree* and a *filling tree*. The emptying trees of both components are the B$^+$-trees that exist at the time when the rolling merge starts. The filling trees are created at that time. A separate thread is assigned to each tree, responsible for emptying or filling the tree. To perform the migration, a *cursor* is circulating in key followed by timestamp order through the leaf level of the emptying and filling trees of components C_i and C_{i+1}, as depicted in Fig. 3. In each step of the rolling merge, the record versions coming from the emptying trees are inspected. If a record version of the emptying tree is younger than the migration time m_i, it is moved to the filling tree of C_i. The cursor of the emptying tree of C_i is advanced to the next record version. If it is decided to migrate the record version, the version is compared, based on its key and timestamp, with the next record version of the emptying tree of C_{i+1}. The smallest of both record versions is moved to the filling tree of C_{i+1} and the corresponding cursor is advanced.

Each time the cursor advances past the last record version of a multi-block, the next multi-block is read from disk by performing a multi-block I/O. The emptied multi-block is returned to free-space management. When a multi-block buffer of a filling tree becomes full, a multi-block I/O is issued to write it to disk. A new multi-block is requested from free-space management. So free blocks are dynamically transferred within and, if possible, also among components. The entire rolling merge process terminates when both emptying trees become empty.

Using multi-block I/O significantly reduces operating system overhead, as less I/O operations are issued, and also reduces disk overhead in terms of seeks and rotational delays. Even with modern disks using track read-ahead and caches for both reads and writes, the benefit of multi-block I/O is significant. We have measured a speedup of 2 for LHAM when using multi-block I/Os of four blocks per I/O operation in our system (see Section 6).

Rolling merges have to be synchronized when they operate on the same component in parallel. This is the most complex situation in LHAM but very common, as emptying a component usually takes a long time and it must be possible to migrate data into it in parallel. Instead of creating different sets of emptying and filling trees, two rolling merges share a tree in the jointly accessed component. The tree chosen depends on which of the rolling merges was first in accessing the shared component. Figure 4 shows both possible situations. In Fig. 4a, the rolling merge $RM_{i-1/i}$ was first, in Fig. 4b, $RM_{i/i+1}$ was first and has later been joined by $RM_{i-1/i}$. The shared trees are indicated in the figure by the larger boxes. They are used as both filling and emptying trees.

A problem arises if the cursors of both rolling merges point to the same record version. This means that the shared tree became empty. In this case, the rolling merge that empties the shared tree has to wait for the other rolling merge to fill the tree with some record versions again. Assume, $RM_{i/i+1}$ waits for $RM_{i-1/i}$. On average, $RM_{i/i+1}$ has to go through r records in C_{i+1} before it consumes a record in C_i. $RM_{i-1/i}$ is much faster in producing new records for C_i, as C_{i-1} is smaller than C_i again by a factor of r. Hence, the assumed blocking of $RM_{i/i+1}$ by $RM_{i-1/i}$ rarely occurs. However, the opposite situation, i.e. $RM_{i-1/i}$ waits for $RM_{i/i+1}$, is highly likely to occur. It is depicted in Fig. 4b. Assume that the shared tree becomes empty. In order not to block $RM_{i-1/i}$ until $RM_{i/i+1}$ produces new records, we allow $RM_{i-1/i}$ to *pass* $RM_{i/i+1}$. The goal of passing is to change trees between $RM_{i-1/i}$ and $RM_{i/i+1}$ until we have a situation as shown in Fig. 4a, allowing both rolling merges to continue. Without passing, both rolling merges would continue at the same speed, which is not acceptable for $RM_{i-1/i}$.

Passing is implemented by logically exchanging the trees between rolling merges as shown in Fig. 5. All trees in components C_{i-1} and C_{i+1} remain unaffected by passing. In the following, we discuss the passing process on a conceptual level. We start in the upper left part of Fig. 5 with the shared tree being empty, as this triggers passing. We then virtually delete the empty shared tree and consider the roles of the remaining two ones. The emptying tree now serves as an emptying tree for both rolling merges. The same is true for the filling tree. This is depicted in the upper right part of Fig. 5. Our goal was to let $RM_{i-1/i}$ continue without waiting for new records from $RM_{i/i+1}$. This is possible now as the emptying tree of $RM_{i-1/i}$ is non-empty. But records moved between the emptying and the filling tree in C_i by $RM_{i-1/i}$ have to be accessed by $RM_{i/i+1}$ for possible migration later. Hence, we have to introduce a new filling tree for $RM_{i-1/i}$, which is used as an emptying tree by $RM_{i/i+1}$.

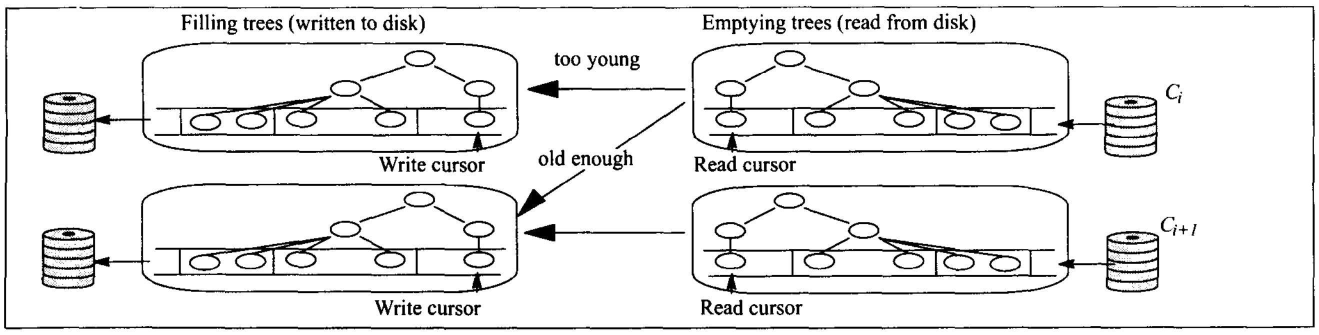

Fig. 3: Rolling Merge in LHAM

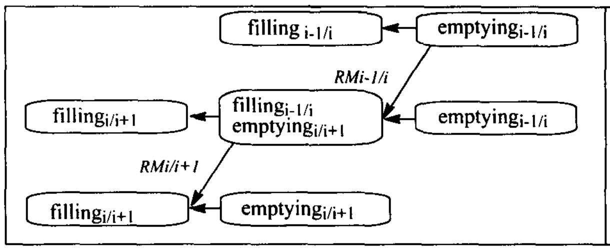

Fig. 4a: $RM_{i/i+1}$ Joining $RM_{i-1/i}$

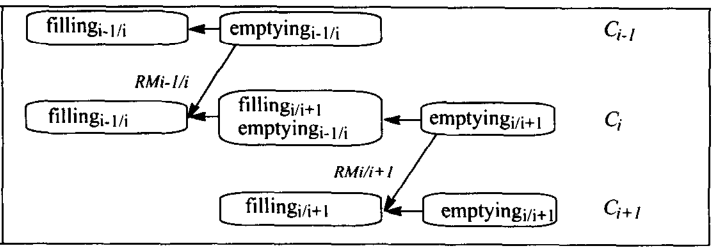

Fig. 4b: $RM_{i-1/i}$ Joining $RM_{i/i+1}$

$RM_{i/i+1}$ keeps its old filling tree. This is shown in the lower part of Fig. 5. We now have exactly the same situation as shown in Fig. 4a. $RM_{i-1/i}$ can continue without waiting for new records from $RM_{i/i+1}$.

4.3 Execution of Queries

All queries are first split into subqueries according to the components that need to be accessed. Each subquery is implemented by a separate thread (see again Fig. 2). In principle, all subqueries can be executed in parallel. This scheme would have the best response time, but may execute some subqueries unnecessarily. Consider for example a query which retrieves the most recent version of a record with a given key. It is possible that this version has already been migrated to the last (disk) component. In this case, all (disk) components have to be accessed to find the most recent version of the record. However, recent record versions will most likely be found in recent components. So accessing only the main memory component C_0 could be sufficient in many cases. Hence, for overall throughput it is best to execute the subqueries sequentially and stop further execution of subqueries as soon as the query result is complete. The performance results presented in section 6 are obtained based on this execution strategy.

A concurrent execution of inserts and queries may cause rolling merges and queries to access a component at the same time. Our approach to ensure good query performance is to prioritize disk accesses of queries over disk accesses by rolling merges. Rolling merges can be suspended whenever they finish the processing of a multi-block of the emptying tree of the destination component, providing a fine disk-scheduling granule.

5 Concurrency Control and Recovery

Concurrency control and recovery issues in LHAM depend on the type of component involved. The main memory component C_0, the disk components C_1 to C_k, and the archive components C_{k+1} to C_n have different requirements. For component C_0, inserts have to be made atomic and durable, and inserts have to be synchronized with queries. For the other components, we do not have to deal with insertions of new data, but only with the migration of existing data, which makes concurrency control and recovery easier. Except for the first archive component C_{k+1}, all archive components are static in terms of the records they store and in terms of their time boundaries. For them, concurrency control and recovery are not required.

5.1 Concurrency Control

We assume transactional predicate locking on key ranges and time ranges on top of LHAM, possibly using advanced low-overhead implementation tricks [GR93, Moh96, Lom93, KMH97]. Hence, concurrency control inside LHAM only has to guarantee consistent access to records (i.e., short-duration locking or "latching"). This includes records under migration between components. We discuss concurrency control issues for each type of component separately.

(1) **Main memory component C_0:** Because no I/O is taking place when accessing C_0, there is little need for sophisticated concurrency control protocols. So standard locking protocols for the index structure used in C_0 can be employed, e.g. tree-locking protocols when C_0 is organized as a tree [GR93, Moh96, Lom93].

(2) **Disk components C_1 to C_k:** Synchronization issues among different rolling merges that access a common disk component have already been discussed in Section 4.2. The only problem left is to synchronize queries with concurrent rolling merges. Interleaved executions of rolling merges and queries are mandatory for achieving short query response times. A query may have to access between one and three index structures (B^+-trees in our case) inside a single component. As discussed in Section 4.2, these index structures are emptied and filled in a given order according to the records' keys and timestamps. This dictates an order for accessing the index structures by queries. Queries have to look up emptying trees before filling trees. Records under migration are not deleted from emptying trees before they have been migrated into the corresponding filling tree. This guarantees that no records are missed by the query. Short term latches are sufficient to protect multi-blocks that are currently filled by a rolling merge from access by queries. Queries do not have to wait for these multi-blocks to become available, they can safely skip them as they have already read the records stored there while looking up the corresponding emptying tree. The only drawback of this highly concurrent scheme is that a record may be read twice by the same query, namely in both the

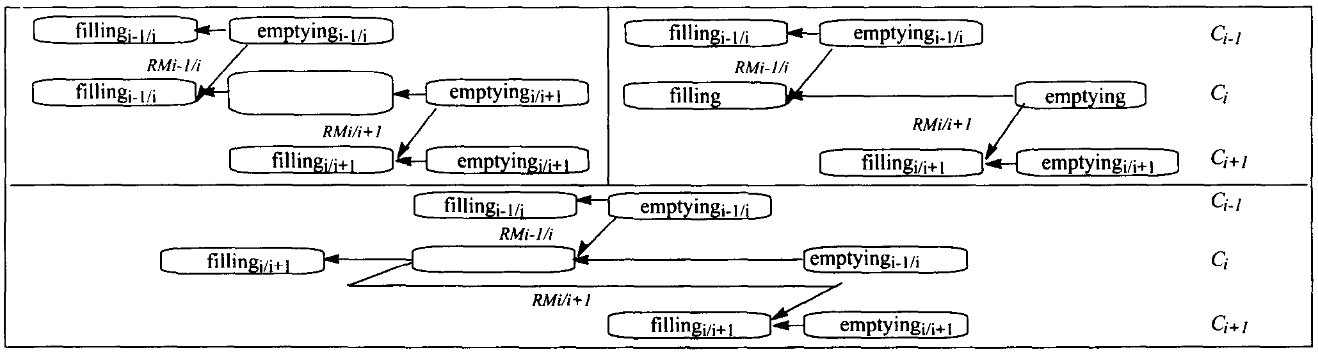

Fig. 5: $RM_{i-1/i}$ Passing $RM_{i/i+1}$

emptying and the filling tree. However, this should occur very infrequently, and such duplicates can easily be eliminated from the query result.

As discussed in Section 4.3, queries start with the most recent component that could possibly hold a query match and then continue accessing older components. During query execution, time boundaries of components may change as records migrate to older components. We have to make sure that no query matches are missed because of a concurrent change of boundaries (i.e., all components containing possible matches are indeed looked up). A change of the boundaries of the most recent component accessed by a query may cause this component to not intersect the query timerange anymore. This will not affect the correctness of the query result, however. On the other hand, a change of the boundaries of the oldest component to be looked up (as determined at query start time) may cause more components to intersect with the query time range. Hence, the oldest component that the query needs to access must be determined dynamically during query execution. Short-term latches on the corresponding data structure in the global LHAM directory are sufficient to correctly cope with these issues.

(3) **Archive components C_{k+1} to C_n:** Records are not migrated between archive components. Instead, the archive grows by creating a new archive component. In terms of concurrency control, an archive component under creation is treated like a disk component. All other archive components are static in terms of their records as well as their time boundaries; so no concurrency control is necessary here. Dropping an archive component causes a change in the global LHAM directory, again protected by a short-term latch.

5.2 Recovery

Similar to the discussion of concurrency control above, we distinguish between the main memory component, the disk components, and the archive components. We restrict ourselves to crash recovery (i.e., system failures); media recovery is orthogonal to LHAM. In general, we need to log all changes to the global LHAM directory that are made whenever a component's time boundaries are changed after finishing a rolling merge. In addition, as we discuss below, logging is necessary only for inserts into the main memory component C_0.

(1) **Main memory component C_0:** All newly inserted records are subject to conventional logging, as employed by virtually all database systems. As records in C_0 are never written to disk before they are migrated to the first disk component, C_0 has to be completely reconstructed during recovery. As C_0 only consists of the most recent records, they will be found in successive order on the log file, resulting in small reconstruction times. If necessary (e.g., when C_0 is exceptionally large), the reconstruction time could be further reduced by keeping a disk-resident backup file for C_0, and lazily writing C_0 blocks to that file whenever the disk is idle (i.e., using a standard "write-behind" demon). Then standard bookkeeping techniques (based on LSNs and a "dirty page list") [GR93] can be used to truncate the log and minimize the C_0 recovery time.

After a record has been migrated to component C_1, it must no longer be considered for C_0 recovery. This is achieved by looking up the most recent record in component C_1 before the C_0 recovery is started. Only younger records have to be considered for reconstructing C_0. Even if the system crashed while a rolling merge from C_0 to C_1 was performed, this approach can be used. In this case, the most recent record in the filling tree of C_1 is used to determine the oldest record that has to be reinserted into C_0 during recovery. During normal operation, the C_0 log file can be periodically truncated using the same approach.

(2) **Disk components C_1 to C_k:** No logging is necessary for migrating records during a rolling merge. Only the creation of emptying and filling trees, the passing of rolling merges as discussed in Section 4.2, the deletion of trees, and changes to time boundaries of components have to be logged.

In order to not lose records that were being migrated at the time of a crash, records are not physically deleted from emptying trees (i.e., their underlying blocks are not released back to the free space management) before they have been migrated into the corresponding filling tree and their newly allocated blocks are successfully written to disk. So we use a careful replacement technique here [GR93] that allows us to correctly recover without having to make a migration step an atomic event. As a consequence, reconstructing the filling and emptying trees during warmstart may create redundant records that will then be present in an emptying and in a filling tree. The number of such redundant records is limited by the size of a multi-block and thus negligible, as only records of a single multi-block per tree and rolling merge have to be reconstructed. Hence, the duplicates can easily be deleted after the trees have been recovered. At the same time, looking up the oldest records of the filling trees and the youngest records of the emptying trees allows reconstructing the rolling merge cursors as shown in Fig. 3, and restarting the rolling merges after the component structures have been reestablished.

(3) **Archive components C_{k+1} to C_n:** Except for the first archive component C_{k+1}, archive components are not subject to recovery. Analogously to concurrency control, the first archive component is treated like a disk component.

In summary, concurrency control and recovery in LHAM are relatively straightforward and very efficient. We either use conventional algorithms, e.g., for logging incoming data, or very simple schemes, e.g. for synchronizing queries and rolling merges. In particular, migrating data by rolling merges does not require migrated data to be logged. Only changes to the LHAM directory require additional logging. This causes negligible overhead.

6 Performance Measurements

In this section, we present experimental performance results from our implementation of LHAM. The results are compared with the analytical expectations for the insert costs. In addition, we compare LHAM to an implementation of the TSB-tree (see Appendix A for a brief review of the TSB-tree), considering both insert and query performance. Note that all experimental results are obtained from complete and fully functional implementations of both LHAM and the TSB-tree, as opposed to simulation experiments. Therefore, we are able to compare actual throughput numbers based on real-time measurements.

6.1 Experimental Results

Our testbed consists of a load driver that generates synthetic data and queries, and the actual implementations of LHAM and the TSB-tree. All measurements were run on a Sun Enterprise Server 4000 under Solaris 2.51. CPU utilization was generally very low, indicating a low CPU-overhead of LHAM. LHAM did not nearly utilize the full capacity of a single processor of the SMP machine. Thus, we restrict ourselves to reporting I/O and throughput figures. Our experiments consist of two parts. In the first part, we investigate the insert performance by creating and populating databases with different parameter settings. Migrations to archive components were not considered. As discussed in the analysis of LHAM's insert costs, the effect of archive components on the insert performance in terms of redundancy is expected to be negligible. In the second part of our experiments, we measure the performance of queries against the databases created in the first part.

6.1.1 Performance of Inserts

In all experiments, we have inserted 400,000 record versions. The size of record versions was uniformly distributed between 100 Bytes and 500 Bytes. This results in 120MB of raw data. The size of a disk block was 8KB in all experiments, for LHAM and the TSB-tree. We used an LHAM structure of 3 components with a capacity ratio of 4; component capacities were 8MB for C_0, 32MB for C_1, and 128MB for C_2. Both disk components resided on the same physical disk. We used a buffer of 1MB for blocks read in a multi-block I/O and a buffer of 1MB for single blocks. This results in a total of 10MB main memory for LHAM. For fair comparison, the TSB-tree measurements were performed with the same total amount of main memory as a node buffer. For LHAM, we have varied the number of disk blocks written per multi-block I/O, in order to measure the impact of multi-block I/O on the insert performance.

We are fully aware of the fact that this data volume merely constitutes a "toy database". Given the limitations of an academic research lab, we wanted to ensure that all experiments were run with dedicated resources in a controlled, essentially reproducible manner. However, our experiments allow us to draw conclusions on the average-case behavior of both index structures investigated. From a practical point of view, these results are more important than an analytic worst-case analysis, which is independent of the parameters and limitations of actual experiments, but provides only limited insights into the performance of real-life applications.

The structure of the TSB-tree depends on the ratio between logical insert and update operations. All experiments start with 50,000 record versions and a logical insert/update ratio of 90% to initialize the database. For the remaining 350,000 record versions, the logical insert/update ratio is varied from 10% inserts up to 90% inserts. Keys were uniformly distributed over a given interval. Logical deletions were not considered. The load driver generated record versions for insertion as fast as possible; so the measured throughput was indeed limited only by the performance of the index structure. The most important performance metrics reported below are the throughput in terms of inserted record versions per second, and the average number of block accesses per inserted record version.

Table 1 lists these values for both LHAM and the TSB-tree, plus other detailed results. The table shows that LHAM outperforms the TSB-tree in every respect. As the structure of LHAM is independent of the logical insert/update ratio, we do not distinguish different ratios for LHAM. Using 8 blocks per multi-block I/O, the throughput of LHAM was always more than 6 times higher than the throughput of the TSB-tree. The benefits of using even larger multi-blocks were small. Additional experiments showed that this is due to limitations in the operating system, which probably splits larger I/Os into multiple requests.

The block accesses required by LHAM and the TSB-tree match our analytical expectations very well. To store 120 MB of data, we need at least 15,000 blocks of 8KB. Using formula (5) and disregarding the terms for the migration to archive media, we expect LHAM to need 180,000 block accesses for inserting the data. In reality, LHAM needs 185,905 block accesses. To further confirm this behavior, we have run additional experiments with a larger number of smaller components, leading to more rolling merges. These experiments have reconfirmed our findings and are omitted for lack of space. The TSB-tree was expected to need about 800,000 block accesses for inserting 400,000 record versions if no node buffer were used. In reality, the experiments show that with 10 MB of buffer for 120MB of data, we need about 600.000 block accesses, depending on the ratio between logical inserts and updates.

LHAM consumed significantly less space than the TSB-tree. The total capacity of the three LHAM components was 168MB, but only 122MB were actually used. This is the benefit of the almost perfect space utilization by LHAM, based on building the B^+-trees inside the components in a bulk manner without the need for splitting leaf nodes. The TSB-tree, on the other hand, consumed between 275 MB and 313 MB, again depending on the logical insert/update ratio. The space overhead of the TSB-tree is caused by redundant record versions and by a lower node utilization due to node splits, similar to conventional B^+-trees. Note however that keeping redundant record versions is an inherent property of the TSB-tree, which is necessary for its good query performance, particularly its logarithmic worst case efficiency.

6.1.2 Queries

We have investigated the performance of four different types of queries:

(1) *<key, timepoint>*,
(2) *<key range, timepoint>*,
(3) *<key, time range>*, and
(4) *<key range, time range>*.

For *<key, timepoint>* queries we have further distinguished between queries with timepoint = *now* (i.e., the current time) and queries with a randomly chosen timepoint. We used the databases as described in the previous sections, i.e., 400,000 record versions with different logical insert/update ratios. In contrast to the insert performance, the query performance of LHAM is affected by the logical insert/update ratio. We give results for the number of block accesses required per query and the (single-user) throughput in queries per second.

	LHAM	**TSB-tree**		
		10% Inserts	**50% Inserts**	**90% Inserts**
Throughput (Inserts/sec)	1/4/8 block(s) per I/O: 146.8 / 304.9 / 348.4	54.4	49.1	45.3
Total number of I/Os	1/4/8 block(s) per I/O: 185905/46983/23663	597802	632587	623770
#Blocks Read/Written	84920 / 100985	282100 / 325702	296551 / 336036	292705 / 331065
#Blocks Accessed per Insert	0.46	1.49	1.58	1.56
Total Database Size (MB)	122	275	323	313
Component Sizes (MB)	$C_0/C_1/C_2$: 1/21/101	Curr./Hist. DB: 64/211	Curr./Hist. DB: 148/175	Curr./Hist. DB: 192/121

Table 1: Insert Performance

Queries of Type <*key, timepoint*>

Fig. 6a shows the average number of block accesses for a query that searches the current version of a given key, plotted against the ratio of logical inserts vs. updates during the creation of the database. Because the TSB-tree can access a given version by reading a single leaf node only, it requires one block access for a query of this type. With the given buffer, index nodes can almost always be found in the buffer. LHAM needs more block accesses here, as no redundancy is used. If the current record version is found in component C_0, no I/O is required. If it is found in C_1, a single block access is sufficient. If a record has not been updated for a long time, it will be stored in component C_2. This requires a lookup of C_0, C_1, and C_2 and requires two block accesses. For a high logical insert/update ratio, this will often be the case. Note, however, that we expect a typical temporal database application to have a rather low logical insert/update ratio, say 10 to 20 percent, resulting in relatively many versions per record. Also note that the absolute performance is good anyway; so this query type is not a major performance concern. Fig. 6b shows the (single-user) throughput achieved by LHAM and the TSB-tree for this type of query. The curve in Fig. 6b is similar to the curve in Fig. 6a, as LHAM cannot benefit from multi-block I/O for this type of query.

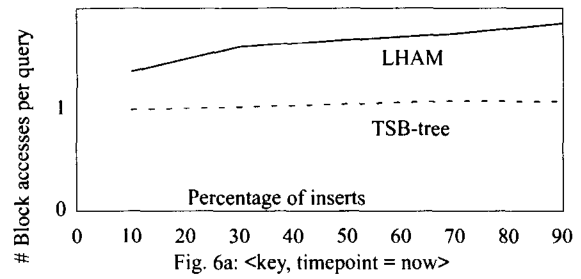

Fig. 6a: <key, timepoint = now>

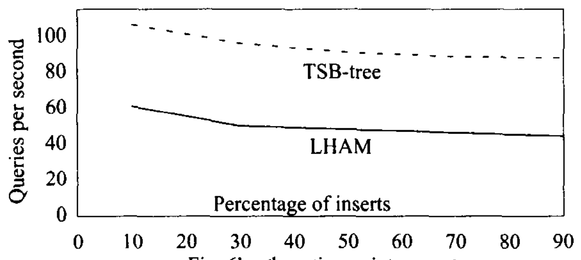

Fig. 6b: <key, timepoint = now>

The situation changes when we consider arbitrary timepoints instead of solely the current time. Fig. 7a and Fig. 7b show again the block accesses required, and the throughput for <*key, timepoint*> queries, but the timepoint is now uniformly distributed over the interval from the oldest record version in the database to *now*. LHAM now performs almost as good as the TSB-tree, because for older data, LHAM often needs to access only component C_2.

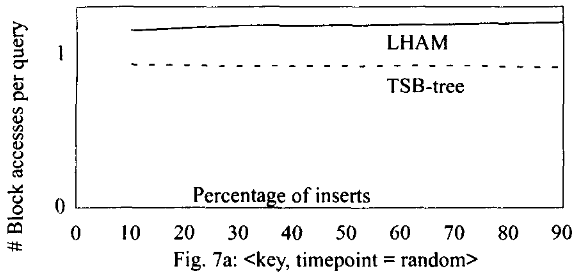

Fig. 7a: <key, timepoint = random>

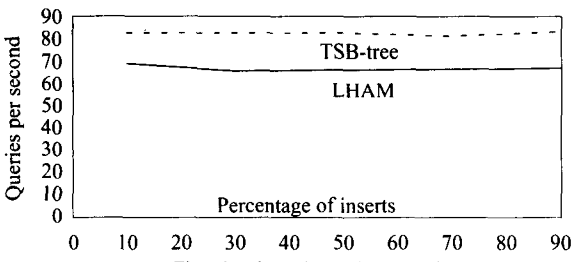

Fig. 7b: <key, timepoint = random>

Queries of Type <*key range, timepoint*>

The performance of <*key range, timepoint*> queries with a key range of 10% of all keys and a timepoint of *now* is shown in Fig. 8a and Fig. 8b. Varying the width of the key range has shown similar results, and choosing a random timepoint rather than *now* has yielded even better results for LHAM. For lack of space, we limit the presentation to one special setting. The results of LHAM are independent of the logical insert/update ratio. This is the case because LHAM has to access all blocks with keys in the given range in all (non-archive) components. Note that the required block accesses by LHAM do not depend on the number of components, but only on the total size of the (non-archive part of the) database. LHAM benefits from multi-block I/O, as shown by the different throughput rates for different numbers of blocks per multi-block I/O in Fig 8b. The performance of the TSB-tree highly depends on the database chosen. When the logical insert/update ratio is low, the current database is small and the number of required block accesses is low. The higher the logical insert/update ratio, the larger the current database and the more block accesses are needed. Fig. 8b shows that even with a small current database, the throughput of the TSB-tree is lower than the throughput of LHAM if multi-block I/O with 8 blocks per I/O is used. Note again that the TSB-tree is inherently unable to exploit multi-block I/O in the same manner due to the absence of clustering. When the current database is large, LHAM outperforms the TSB-tree even without multi-block I/O.

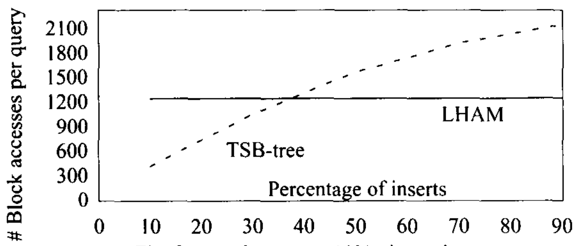

Fig. 8a: <key range 10%, timepoint = now>

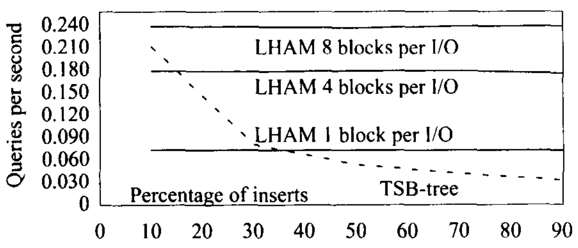

Fig. 8b: <key range 10%, timepoint = now>

Queries of Type <*key, time range*>

Fig. 9a and Fig. 9b show the performance of <*key, time range*> queries with a time range of 50%. Varying the width of the time range has led to similar results, which are omitted here for lack of space. LHAM outperforms the TSB-tree in terms of block accesses per query as well as throughput for all database settings. As LHAM stores all record versions with the same key in physical proximity, only one or two block accesses are need for each query. In general, LHAM benefits from multi-block I/O for this type of query. However, with only one or two blocks read per query for the databases in our experiments, using multi-block I/O would waste some disk bandwidth. Keeping statistics about the data would enable us to make appropriate run-time decisions on single-block vs. multi-block I/Os.

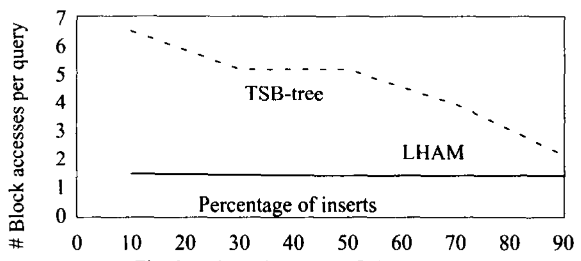

Fig. 9a: <key, time range 50%>

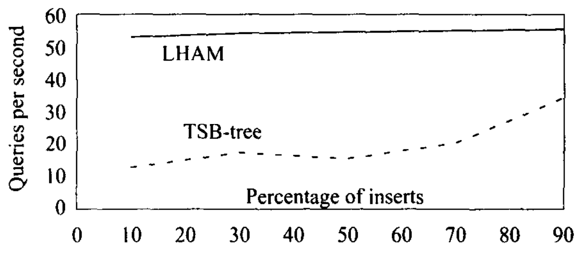

Fig. 9b: <key, time range 50%>

Queries of Type <*key range, time range*>

Finally, we consider the performance of <*key range, time range*> queries. Fig. 10a and Fig. 10b show the results for a key range of 10% and a time range of 10%. The results are similar to <*key range, timestamp*> queries as shown in Fig. 8a and Fig. 8a. Again, similar results have been obtained for other settings of the range widths.

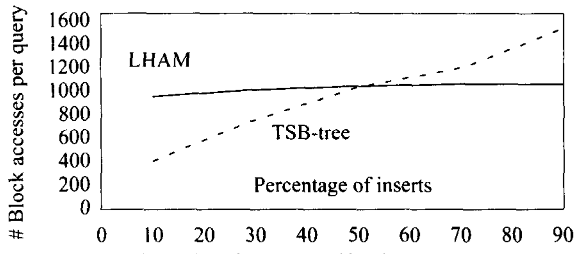

Fig. 10a: <key range 10%, time range 10%>

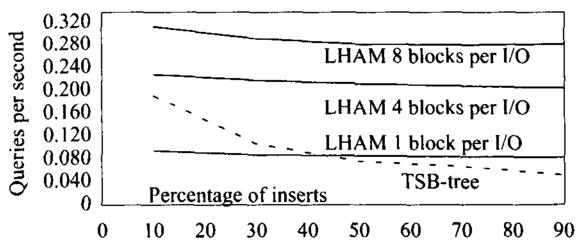

Fig. 10b: <key range 10%, time range 10%>

6.1.3 Multi-User Performance

We have also performed experiments with queries and inserts (and rolling merges), running concurrently. By prioritizing queries over rolling merges, query performance remained almost unaffected by concurrent rolling merges. The insert throughput, on the other hand, is adversely affected only when the system becomes overloaded. An overload occurs if the data rate of incoming data becomes higher than the data rate that can be sustained by the rolling merges in the presence of concurrent queries. Thus, the expected query load must be taken into account when configuring the system. Insert costs as analyzed in Section 3.3 determine the I/O bandwidth, i.e., the number of disks, necessary to sustain the insert load. Additional disks are required for the the query load.

In our current implementation, rolling merges are initiated when the amount of data stored in a component reaches a fixed threshold. In a multi-user environment, it would be beneficial to invoke rolling merge multi-block I/Os whenever the disk would be idle, even if the threshold is not yet reached.

7 Conclusions

Our experimental results based on a full-fledged implementation have demonstrated that LHAM is a highly efficient index structure for transaction-time temporal data. LHAM specifically aims to support high insertion rates beyond what a B^+-tree-like structure such as the TSB-tree can sustain, while also being competitive in terms of query performance. In contrast to the TSB-tree, LHAM does not have good worst-case efficiency bounds. However, our experiments have shown that this is not an issue under realistic, "typical-case" workloads. LHAM's average-case performance is consistently good. A number of extensions of LHAM require further studies that we are working on:

- First, redundancy between disk components can be used to improve query performance for those queries which would otherwise have to search too many components. Depending on the expected query load for a given period of time, the redundancy option for a component can even be turned on and off dynamically each time a rolling merge migrates data out of it. The information about the resulting redundancy which can be exploited by queries can easily be kept in the global LHAM dictionary.
- Second, the current implementation of the global LHAM dictionary only keeps track of the time partitioning in terms of the components' low and high time boundaries. This can be generalized to allow some form of key-partitioning also. Such a scheme requires a multi-dimensional global directory structure -which should still be very small- as well as rolling merge processes with more than two source trees and more than two destination trees.
- Finally, we have so far considered only B^+-trees as index structures for the data within an LHAM component. Using specific index structures for temporal data here (e.g., the TSB-tree) could further enhance query performance. However, whatever index structure is chosen must provide a means for bottom-up bulk loading of record versions during a rolling merge, in order to achieve the "batch-processing" benefit and the gain from multi-block I/O in the merge process.

References

[Bec96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, "An Asymptotically Optimal Multiversion B-tree", *VLDB Journal*, Vol.5, No. 4, 1996

[BSW97] J. v. d. Bercken, B. Seeger, P. Widmayer, "A Generic Approach to Bulk Loading Multidimensional Index Structures", *Proc. VLDB Conference*, 1997

[EKW91] R. Elmasri, V. Kim, G. T. J. Wuu, "Efficient Implementation for the Time Index", *Proc. Data Engineering*, 1991.

[EWK93] R. Elmasri, G. T. J. Wuu, V. Kim, "The Time Index and the Monotonic B^+-tree", in [Tan93]

[GP87] J. Gray, F. Putzolu, "The Five Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time", *Proc. SIGMOD*, 1987

[GR93] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

[Gut84] A. Gutman, "R-trees: A Dynamic Index Structure for Spatial Searching", *Proc. SIGMOD*, 1984

[Jag97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, R. Kanneganti, "Incremental Organization for Data Recording and Warehousing", *Proc. VLDB Conference*, 1997

[Kol93] C. P. Kolovson, "Indexing Techniques for Historical Databases", in [Tan93]

[KMH97] M. Kornacker, C. Mohan, J.M. Hellerstein, "Concurrency and Recovery in Generalized Search Trees", Proc. SIGMOD, 1997

[Lom93] D. Lomet, "Key Range Locking Strategies for Improved Concurrency", Proc. VLDB, 1993

[LS89] D. Lomet, B. Salzberg, "Access Methods for Multiversion Data", *Proc. SIGMOD*, 1989

[LS90] D. Lomet, B. Salzberg, "The Performance of a Multiversion Access Method", *Proc. SIGMOD,* 1990

[Moh96] C. Mohan, "Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM, in: V. Kumar (Editor), Performance of Concurrency Control Mechanisms in Centralized Database Systems, Prentice-Hall, 1996

[OCGO96] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, "The Log-Structured Merge-Tree (LSM-tree)", *Acta Informatica*, Vol 33, No. 4, 1996.

[OW93] P. O'Neil, G.Weikum, "A Log-Structured History Data Access Method", *5th International Workshop on High Performance Transaction Systems (HPTS),* Asilomar, California, 1993

[RO92] M. Rosenblum, J. K. Ousterhout, "The Design and Implementation of a Log Structured File System", *ACM Transactions on Computer Systems,* Vol. 10, No. 1, 1992

[Sno90] R. Snodgrass, "Temporal Databases: Status and Research Directions", *SIGMOD Record*, Vol. 19, No. 4, 1990

[SOL94] H. Shen, B. C. Ooi, H. Lu, "The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases", *Proc. Data Engineering,* 1994

[ST94] B. Salzberg, V. J. Tsotras, "A Comparision of Access Methods for Time Evolving Data", Technical Report NU-CCS-94-21, Northeastern University, Boston, 1994

[Tan93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass (Editors), "*Temporal Databases: Theory, Design, and Implementation,*" Benjamin/ Cummings Publishing Company, 1993

[TK95] V. J. Tsotras, N. Kangelaris, "The Snapshot Index: An I/O-Optimal Access Method for Timeslice Queries", *Information Systems,* Vol. 20, No. 3, 1995

Appendix A: Brief Review of the TSB-tree

The TSB-tree is a B^+-tree-like index structure for transaction-time databases [LS89, LS90]. It indexes record versions in two dimensions; one dimension is given by the conventional record key, the other by the timestamp of the record version. Its goal is to provide good worst-case efficiency for exact match queries as well as range queries in both time dimension and key dimension.

Nodes and Node Splits

Basically, each leaf node covers a two-dimensional interval, i.e., a "rectangle" in the data space, whose upper bounds are initially "open" (i.e., are interpreted as infinity) in both dimensions. A node is represented by a pair of key and timestamp, defining the lower left corner of the rectangle that it covers. The area covered by a rectangle becomes bounded if there exists another leaf node with a higher key or time value as its lower left corner. A leaf node contains all record versions that have a (key, timestamp) coordinate covered by its rectangle. Two types of nodes are distinguished: current nodes and historical nodes. Current nodes store current data, i.e., data that is valid at the current time. All other nodes are denoted historical.

As all data is inserted into current nodes, only current nodes are subject to splits. Current nodes can be split either by key or by time. A record version is moved to the newly created node if its (key, timestamp) coordinates fall into the corresponding new rectangle. The split dimension, i.e., whether a split is performed by key or time, is determined by a split policy. We have used the *time-of-last-update (TLU)* policy for all our experiments, which does a split by time unless there is no historical data in the node, and performs an additional split by key if a node contains two thirds or more of current data. The split time chosen for a split by time is the time of the last update among all record versions in the node. The TLU policy achieves a good tradeoff between space consumption, i.e. the degree of redundancy of the TSB-tree, and query performance. This is shown in [LS90] and has been confirmed by our own experiments.

A non-leaf index node stores a set of index terms. An index term is a triple consisting of a key, a timestamp, and a pointer to another index node or a leaf node. Like the open rectangle defined for each leaf node, an index term also covers an open rectangle, defined by key and timestamp as the lower left corner. Other index terms with higher key or timestamp bound this area. Index node splitting is similar to leaf node splitting. We have again adopted the TLU split policy. For the subtle differences concerning restrictions on the split value for time splits, the reader is referred to [LS89].

Searching

Searching in TSB-trees can be viewed as an extension to the search procedure for B^+-trees. Assume we are searching for a record version (k, t) with key k and timestamp t. At each level of the tree, the algorithm first discards all index terms with a timestamp greater than t. Within the remaining terms it follows the index term with the maximum key value being smaller than or equal to key k. This process recursively descends in the tree and terminates when a leaf node is found. A similar algorithm is used for range queries. Instead of following a single index term while descending the tree, a set of index terms to follow is determined. Because of the redundancy employed in the TSB-tree, the worst case performance of queries is logarithmic in the number of record versions stored (including the redundant ones). There is no guaranteed clustering, however, neither in key nor in time dimension.