The VLDB Journal (2000) 8: 199–221

The VLDB Journal

The LHAM log-structured history data access method

Peter Muth[1], **Patrick O'Neil**[2], **Achim Pick**[1], **Gerhard Weikum**[1]

[1] Department of Computer Science, University of the Saarland, D-66041 Saarbrücken, Germany
e-mail: {muth, pick, weikum}@cs.uni-sb.de
[2] Department of Mathematics and Computer Science, University of Massachusetts at Boston, Boston, MA 02125-3393, USA
e-mail: poneil@cs.umb.edu

Edited by O. Shmueli. Received March 4, 1999 / Accepted September 28, 1999

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 insertion and update rates. This paper presents an 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.

Key words: Index structures – Storage systems – Performance – Temporal databases – Data warehouses

1 Introduction

1.1 Problem setting

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 very large data warehouses [CACM98], in scientific databases that are fed by automatic instruments [MSS95], or in workflow management systems for keeping workflow histories [AGL98]. 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 simplify the explanation, 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, 8 KB 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 s $\approx$ 1 hour apart. According to the 5-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. Of course, this scenario exaggerates to make its point as clear as possible; in practice, applications often exhibit skewed access patterns such that additional memory could be leveraged for caching frequently accessed leaf nodes. The point is that even more realistic applications may need more than a minimum number of disks to sustain the I/O throughput.

1.2 Contribution

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*. So LHAM essentially leverages techniques for the bulk-loading of index structures to continuously construct and maintain an online index in a highly efficient manner.

In general, *components* may exist on different levels of a *storage hierarchy*. When 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. For the above example, an LHAM structure with a main-memory component of 144 MB and two disk components with a total size of 30 GB is sufficient. This translates into two disks for LHAM, in contrast to 20 disks if a B^+-tree-like access method were used.

Two-dimensional key-time queries are supported by LHAM through partitioning the data of the various components based on time intervals. For space efficiency, record versions that are valid across the timespans of multiple components are stored only in one component according to the version's creation time; that is, LHAM usually employs a *non-redundant partitioning* scheme. The exception from this rule are components that may reside on archive media such as tapes or WORM units; these components are formed based on a redundant partitioning scheme, where versions are redundantly kept in all *archive components* in which they are valid. Within each component, record versions are organized in a B^+-tree using the records' key attribute and the version timestamp as a logically concatenated search key. Based on this storage organization, LHAM provides consistently good performance for all types of range queries on key, on time, and on the combination of both.

The basic idea of an earlier form of LHAM has been sketched in [OW93], and a full-fledged implementation has been described in [MOPW98]. The current paper is an extended version of [MOPW98]. In addition to the performance experiments and the analysis of insertion costs already reported there, the current paper includes a mathematical average-case cost analysis for key-time (point) queries, an extended discussion of concurrency control and recovery issues, and an outline of various, promising extensions and generalizations of the LHAM method. The paper's major contributions are the following.

- 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 insertions in terms of required block accesses is mathematically analyzed. We further present analytical results on the average-case cost of key-time point queries.
- We present a full-fledged LHAM implementation for shared-memory multi-processors 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 insertion and key-time point-query performance, and to evaluate the full spectrum of query types, 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.
- We discuss several promising extensions and generalizations of LHAM, namely, supporting a tunable mix of non-redundant and redundant partitionings between components, the use of general multi-dimensional index structures within a component, and more general forms of the LHAM directory for more flexible partitioning schemes.

1.3 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 [ST99]. 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 certain conditions. However, our experimental studies indicate that such degradation occurs only in specifically constructed "adversary" scenarios and is rather unlikely under more realistic loads. For almost all query types, the average performance of LHAM is at least as good as for the TSB-tree, for many cases, even substantially better because of better data clustering, both within and across pages, and potential for multi-block I/O.

Most proposals for index structures on temporal data are not specifically targeted at high insertion rates, the only exceptions being [Jag97] and [BSW97]. Both approaches use continuous on-line reorganizations of the data like LHAM to improve the performance of insertions. Because of their strong relationship to LHAM, we discuss both of these approaches in detail in Sect. 8, after having presented the LHAM approach and its performance.

1.4 Outline of the paper

The paper is organized as follows. Section 2 presents the principles of LHAM in terms of time-based partitioning of the data, data migration, and query processing. Section 3 develops analytic models for the amortized costs of insertions and average-case costs of key-time point queries. In Sect. 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 Sect. 5. Section 6 briefly reviews the TSB-tree as a major competitor to our approach. Section 7 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 8 compares the LHAM method to similar, more recently proposed approaches. Section 9 discusses extensions and generalizations of LHAM. Section 10 concludes the paper.

2 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 consequence, 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.

2.1 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 typi-

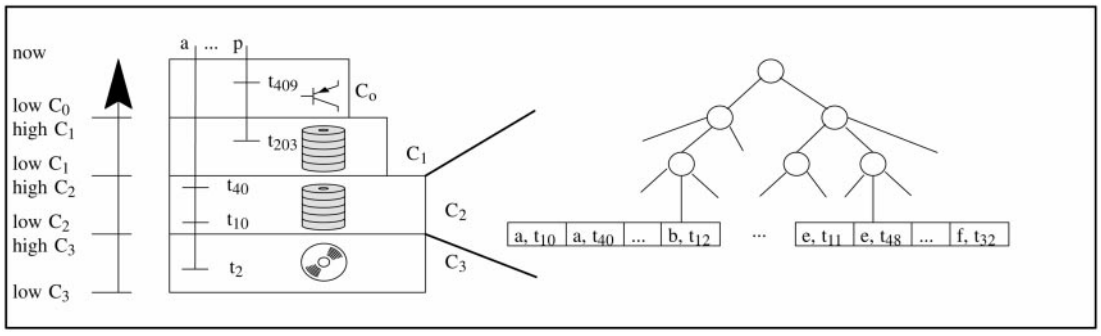

Fig. 1. LHAM component organization and B$^+$-tree inside component

cally 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 Sect. 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. The redundancy option can even be limited to certain keys or key ranges if, for example, the update rates are skewed across the key domain.

2.2 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, $low_i \leq m_i \leq high_i$, 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 (i.e., older) 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. Figure 2 shows an example of three successive rolling merges between components C_0 and C_1. Whenever component C_0 becomes full, i.e., its box becomes filled with record versions (the white, grey, and black rectangles), a new rolling merge is started as indicated by the arrows. To maintain the record order in the tree of C_1, whose collation order is by key and time, with key being the primary criterion and time being secondary, records migrating from C_0 to C_1 are merged with records already stored in C_1. This is indicated by the differently shaded rectangles.

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

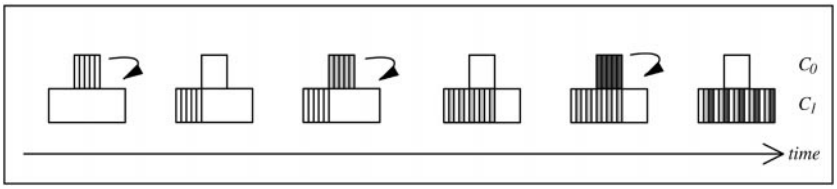

Fig. 2. Rolling merges between C_0 and C_1

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 Sect. 7). 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, both within and across pages, all the time, with almost 100% node utilization, which in turn benefits range queries and index scans.

2.3 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 earlier 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). Figure 3 shows the algorithm for a key/time-range query for the time interval from t_{low} to t_{high} in pseudo-code notation. The algorithm first determines the youngest component that needs to be searched. Components are then searched from younger to older components, until a component with one of the following stopping conditions is reached. (1) The component contains a matching record version with a timestamp that is already outside the query's time range; then this version is the last record version that qualifies. (2) The component stores records redundantly and older components do not overlap the query's time range. The search finishes with this component. Within each affected component, the functions $SearchTree$ and $NextFromTree$ are invoked to initiate a standard range query on a B$^+$-tree and collect the qualifying records.

"Time-travel" queries for key ranges rather than individual keys proceed analogously. The only difference is that this time-range/key-range query type requires maintaining a list of the oldest version encountered so far for each key that falls into the specified key range. This is necessary to determine the oldest version of a key that matches the query's time-range condition. Assuming that keys are rather short and key ranges are typically limited in width, LHAM simply keeps this list in memory.

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 Sect. 2.1) 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_2, 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 Sect. 7.1.2).

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 perfor-

```
Search (key, t_low, t_high) {
  StartComponent = 0;
  for i = 0 to n {                                  // determine youngest component to search in
    if ( low_i  > t_high) {                         // component too young
      StartComponent++;
    else
      break;                                        // must search this and possibly subsequent components
  }
  for i = StartComponent to n {                     // this loop searches the relevant components
    rec = SearchTree (key, t_low, t_high);
    while ( rec != NULL) {                          // most recent record version for given interval found
      addResult  (rec);                             // add record version to result list
      rec = NextFromTree(key, t_low, t_high);
      if (rec != NULL) & (rec.timestamp ≤ t_low)    // this is the oldest record version that qualifies
        return;                                     // older record versions can be ignored
    }
    if ( i+1 ≥ k) & (high_i+1     t_low)            // next component uses redundant partitioning
                                                    //   and does not overlap the query's time range
      return ;                                      // search is complete
  }
}
```

Fig. 3. Pseudocode for query with given key and time range

mance. Time-range queries for a given key are also efficiently supported, as all versions of a record are clustered by time both within a page and across leaf pages of the index. 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 across pages 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 accordingly cluster the data across pages.

3 Analysis of insertion and query costs

3.1 Amortized costs of insertions

We derive a formula for the amortized 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 Sect. 2.1), 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% 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 timepoints 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% full C_i component on average. This consideration is fully confirmed by our experimental findings in Sect. 7.

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 $C_0, \ldots, C_k$. We obtain for the total number of accessed blocks $block_{accesses}$:

$$\begin{aligned} block_{accesses} = {} & l_1(r_0 + r_0 r_1) + l_2(2r_0 r_1 + r_0 r_1 r_2) + \ldots \\ & + l_k \left(2 \prod_{i=0}^{k-1} ri + \prod_{i=0}^{k} r_i \right) + l_{k+1} \left(2 \prod_{i=0}^{k} r_i \right) \\ & + l_{k+1} \left(\frac{2}{s} \sum_{j=0}^{k} \prod_{i=0}^{j} r_i \right) \end{aligned} \quad (1)$$

Note that for emptying component C_0, no I/O is required, which leads to the term $(r_0 + r_0 r_1)$ rather than $(2r_0 + r_0 r_1)$. Further note that, on average, all other components are 50% full, so that the second summand of the above term is indeed $r_0 r_1$ rather than $2r_0 r_1$. The same holds for the subsequent terms. 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 Sect. 2.2, 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

$$l_i = block_{tot} / \prod_{j=0}^{i-1} r_j \ . \quad (2)$$

By substituting Eq. 2 into Eq. 1, we obtain

$$\begin{aligned} block_{accesses} = {} & block_{tot} \left(2k + 1 + \sum_{i=1}^{k} r_i \right) \\ & + l_{k+1} \left(\frac{2}{s} \sum_{j=0}^{k} \prod_{i=0}^{j} r_i \right) \ . \end{aligned} \quad (3)$$

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 Eq. (3) by r, we obtain

$$\begin{aligned} block_{accesses} = {} & block_{tot}(k(2 + r) + 1) \\ & + l_{k+1} \left(\frac{2}{s} \sum_{j=0}^{k} r^{j+1} \right) \ . \end{aligned} \quad (4)$$

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

$$block_{accesses} \leq block_{tot} \left(k(2 + r) + 1 + \frac{4}{s} \right) \ . \quad (5)$$

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 28 GB of data. Assume that we use two disk components. Main-memory component C_0 has a size of 144 MB, C_1 has 2 GB and C_2 has 28 GB. 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 $1/(2 * 7)$. As we have about 31 GB of data online, this leads to an additional space requirement of 2.2 GB for redundant data on disk. With about 3,500,000 blocks of 8 KB size to insert, according to Eq. (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.

3.2 Average-case cost of key-timepoint queries

In this subsection, we derive a mathematical model for the average-case performance of key/timepoint queries in terms of their expected number of block accesses. Recall that the worst-case performance of such queries can degrade because of the non-redundant partitioning of the time dimension between disk components. A time-travel query for a given key and timepoint would first be directed to the component into which the query's timepoint falls, but if the component does not contain an older or exact-match version for this key, the search will have to be extended into the older components until a version is found or the oldest disk component has been searched. However, we expect this worst case behavior to be infrequent. Rather, we would usually expect that a key/timepoint query needs to look up only one or, at worst, two components. To verify these expectations, we have developed the following analytical average-case cost model, and we have also addressed this issue in our experimental studies reported in Sect. 7. Although the worst-case versus average-case consideration is an issue also for queries with key ranges or time ranges, we concentrate on point queries,

as they already capture the essence of the problem. Our analytical model could be extended to range queries, but this is not elaborated here. For tractability of the analysis, we assume that no rolling merge is in progress while the query executes.

We concentrate on the main-memory component C_0 and the k disk components C_1 through C_k, assuming fixed component sizes c_0 through c_k in terms of allocated blocks with a geometric size ratio r so that $c_k = c_0 * r^k$. For simplicity, we assume that record versions are of fixed size, so that the block sizes of components can be directly translated into the numbers of record versions that can reside in a component, v_0 through v_k, by multiplying the c_i values with the number of record versions per block. The entire LHAM database is assumed to contain m different keys, and we further assume that all operations are logical updates of existing keys, i.e., insertions of new versions for already existing keys. This simplifying restriction is justified by the observation that most real-life applications (e.g., data warehouses) are dominated by logical updates rather than logical insertions of new keys. The update operations are assumed to be uniformly distributed across all m keys.

The final assumption that we make for tractability of the cost analysis is that logical updates arrive according to a Poisson process, a standard assumption in many performance analyses that is justified whenever the operations are submitted from a large number of independent sources (e.g., phone calls). The arrival rate of this Poisson process is denoted λ; thus, the interarrival time between two successive update operations is exponentially distributed with mean value $1/\lambda$. Since Poisson processes that are split into several branches are again Poisson processes, the uniform selection of keys for updates leads to an exponentially distributed time between successive updates of the same key with mean value m/λ.

From the given component sizes v_0 through v_k and the exponentially distributed interarrival times, we can now infer the expected timespan that is covered by a component. The timespan $\Delta_0 = high_0 - low_0$ (with $high_0 = now$) covered by C_0 is implicitly given by the constraint that the total number of versions created in time Δ_0, which is given by $\Delta_0 * \lambda$, must fit into the capacity v_0 of the component; hence, $\Delta_0 * \lambda = v_0$, or equivalently: $\Delta_0 = v_0/\lambda$. By the same token we derive the general equation

$$\Delta_i = high_i - low_i = v_i/\lambda = r^i * v_0/\lambda \ . \tag{6}$$

Thus, the total timespan covered by the non-archive components of the LHAM database is

$$\begin{aligned}\Delta_{total} &= high_0 - low_k = \sum_{i=0}^{k} \Delta_i = \frac{v_0}{\lambda} \sum_{i=0}^{k} r^i \\ &= \frac{v_0}{\lambda} \ \frac{1 - r^{k+1}}{1 - r} \ .\end{aligned} \tag{7}$$

Now consider a key-timepoint query for randomly chosen key x and timepoint t that falls into the timespan of C_i, i.e., $low_i \leq t \leq high_i$. We are interested in the probability that this point query finds a version for key x that is no younger than t and resides in C_i itself, i.e., a version that must have been created between low_i and t. Denote this relevant time interval by $\Delta_{i,t} = t - low_i$. The number of versions for key x that have been created within time $\Delta_{i,t}$ is Poisson distributed with parameter $\Delta_{i,t} * \lambda/m$ (i.e., the length of the time interval multiplied by the update rate for key x), and the probability that at least one version for x has been created within $\Delta_{i,t}$ is 1 minus the probability that no such version exists. We abbreviate this probability as $p_{i,t}$:

$$\begin{aligned}p_{i,t} &:= P\ [most\ recent\ version\ of\ x\ preceding\ t \\ &\qquad has\ been\ created\ after\ low_i] \\ &= 1 - e^{-\frac{\lambda}{m}\Delta_{i,t}} \ .\end{aligned} \tag{8}$$

This consideration can be generalized, leading us to the probability that the most recent version of x that precedes time t resides in component C_j with $k \geq j > i$:

$$\begin{aligned}p_{j,t} &:= P\ [most\ recent\ version\ of\ x\ preceding\ t \\ &\qquad has\ been\ created\ between\ low_j\ and\ high_j] \\ &= P\ [at\ least\ one\ version\ for\ x\ in\ \Delta_j] \\ &\quad * \Big(\prod_{\mu=i+1}^{j-1} P\ [no\ version\ for\ x\ in\ \Delta_\mu]\Big) \ * \ (1 - p_{i,t}) \\ &= (1 - e^{-\frac{\lambda}{m}\Delta_j}) \ * \ \Big(\prod_{\mu=i+1}^{j-1} e^{-\frac{\lambda}{m}\Delta_\mu}\Big) \ * \ e^{-\frac{\lambda}{m}\Delta_{i,t}} \ .\end{aligned} \tag{9}$$

As the next step, we can now easily compute the probability that the query for timepoint t needs to search z components, with $1 \leq z \leq k - i + 1$, namely

$$\begin{aligned}&P\ [query\ for\ time\ t\ between\ low_i\ and\ high_i \\ &\qquad needs\ to\ search\ z\ components] \\ &= p_{i+z-1,t} \ .\end{aligned} \tag{10}$$

Since we are not particularly interested in a specific timepoint t, we should now consider t as a random variable that varies across the total timespan Δ_{total} of all components. We assume that t is uniformly distributed within the time interval under consideration. Thus, as long as we restrict t to fall into the timespan of C_i, we obtain the probability for having to search z components, averaged over all such queries, as

$$\begin{aligned}q_{i,z} &:= P\ [a\ random\ query\ initially\ directed \\ &\qquad to\ C_i\ needs\ to\ search\ z\ components] \\ &= \int_{low_i}^{high_i} \frac{1}{high_i - low_i} p_{i+z-1,t}\ dt \ ,\end{aligned} \tag{11}$$

where the first factor in the integral accounts for the uniform distribution of t within the timespan of C_i. With t uniformly varying across Δ_{total}, the geometric ratio of the component sizes incurs a skew towards more queries directed to older and larger components. This effect is taken into account by the geometrically increasing factors Δ_i/Δ_{total} and corresponding integration ranges in the following sum of integrals:

$$\begin{aligned}q_z &:= P\ [a\ random\ query\ needs \\ &\qquad to\ search\ z\ components] \\ &= \sum_{i=0}^{k} \frac{\Delta_i}{\Delta_{total}} q_{i,z}\end{aligned}$$

$$= \sum_{i=0}^{k} \left(\frac{\Delta_i}{\Delta_{total}} \int_{low_i}^{high_i} \frac{1}{high_i - low_i} p_{i+z-1,t}\ dt \right) . \quad (12)$$

Finally, the expectation value of the number of components that need to be searched by a random key-timepoint query is derived as

$$E\ [number\ of\ components\ to\ be\ searched\ by\ a\ random\ query] = \sum_{z=1}^{k+1} z q_z\ . \quad (13)$$

So, by simply substituting the various expressions into the final formula above, we have derived a closed form for the average-case number of components that need to be searched. Within each component C_i, the number of necessary block accesses can be estimated as 1, since all levels of the component's B$^+$-tree except the leaf level are typically cached in memory and we are considering point queries (i.e., need to access only a single leaf node). So formula 13 gives also the expected number of block accesses for a random point query.

An interesting special case of the considered class of key-timepoint queries are those queries where the specified timepoint is the current time *"now"*, i.e., queries that are interested in the most recent version for a given key. The expected number of components to be searched, and thus the expected number of block accesses, can be derived from our more general analysis in a straightforward manner. With $t = now$ we know that the component to which the search is initially directed is C_0, and the query's timepoint is always identical to $high_0$. Then the average cost for such a query is obtained by specializing formula 11 in that we consider only $q_{0,z}$ and substitute t in formula 9 by $high_0$. Then the summation according to formula 12 becomes obsolete, and we can directly substitute into formula 13, thus arriving at

$$E\ [number\ of\ components\ to\ be\ searched\ by\ a\ query\ with\ random\ key\ and\ time = now] = \sum_{z=1}^{k+1} z q_{0,z}[t/high_0]\ , \quad (14)$$

where $q_{0,z}[t/high_0]$ denotes $q_{0,z}$ with t substituted by $high_0$.

As an example, consider an LHAM configuration with one memory component C_0, two disk components C_1 and C_2 (i.e., $k = 2$) with record versions of length 300 bytes, a block size of 8 KB and component sizes c_0 = 8 MB, c_1 = 32 MB, c_2 = 128 MB (i.e., a size ratio of $r = 4$), which translate into the following approximate number of record versions in each of the three components: $v_0 = 25,000$, $v_1 = 100,000$, $v_2 = 400,000$. We have chosen these particular figures as they closely match those of our experiments in Sect. 7. The LHAM database contains $m = 50,000$ different keys, and we assume that it has been populated with an update rate of $\lambda = 50$ updates per second; so each key has received an update every 1000 s on average. The timespans covered by the three components then are Δ_0 = 500 s, Δ_1 = 2000 s, and Δ_2 = 8000 s. With these figures, we can derive the expected number of block accesses for a random key-timepoint query whose time parameter lies in component C_0, C_1, C_2 as 1.89, 1.43, and 1, respectively, resulting in an overall expected cost of 1.123 block accesses per query. For random queries that are initially directed to C_0, i.e., those that are most likely to be penalized, the probabilities of having to search 1, 2, 3 components are 0.213, 0.524, 0.081, respectively. In other words, the vast majority of queries has to search at most two components, even under the rather unfavorable condition in our scenario that the C_0 size is fairly small in comparison to the number of different keys. For queries to current versions, i.e., queries whose time parameter is *"now"*, the expected number of components to be searched is 1.69; as C_0 does not require any disk I/O, the expected number of disk block accesses is even lower, namely, approximately 1.3. Overall, these figures are much more favorable than the worst case of having to search all three components. With more and larger components, the expected cost for point queries would be even lower.

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. Figure 4 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 5 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 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.

To perform the migration from an emptying to a filling tree, a separate thread is assigned to each tree. 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. 5. 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

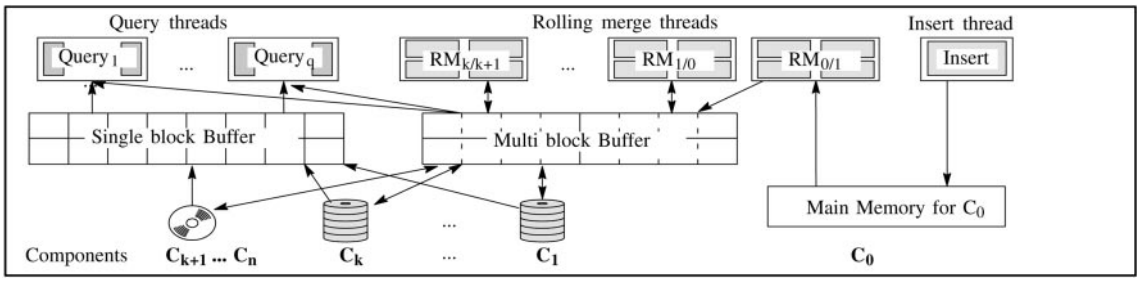

Fig. 4. LHAM architecture

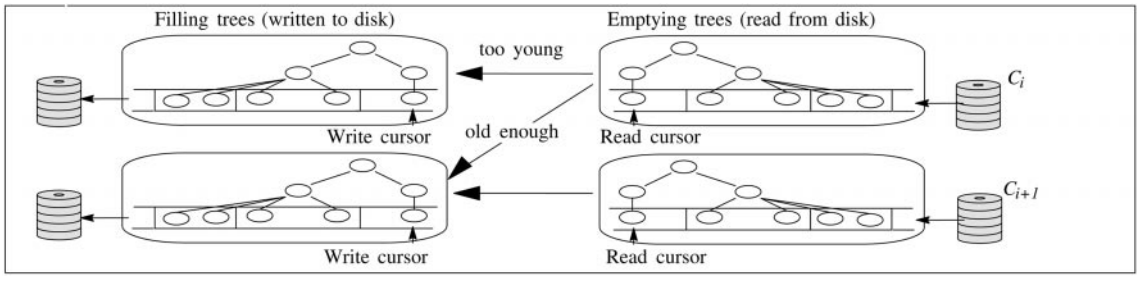

Fig. 5. Rolling merge in LHAM

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 Sect. 7).

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 6 shows both possible situations. In Fig. 6a, the rolling merge $RM_{i-1/i}$ was first, in Fig. 6b, $RM_{i/i+1}$ 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. 6b. 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. 6a, 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. 7.

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. One can view the emptying and the filling trees inside a component as a single "conceptual tree", as the various trees cover adjacent intervals of the record-version space ordered by key and time. The rolling-merge cursors define these intervals, i.e., they define the trees. We start in the upper left part of Fig. 7, with the

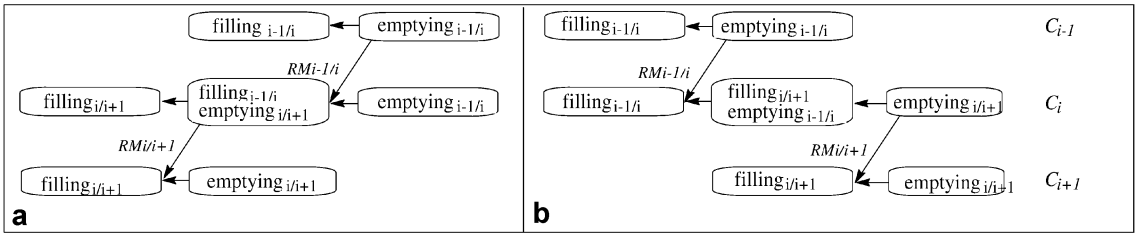

Fig. 6. Joining of rolling merges. **a** $RM_{i/i+1}$ joining $RM_{i-1/i}$, **b** $RM_{i-1/i}$ joining $RM_{i/i+1}$

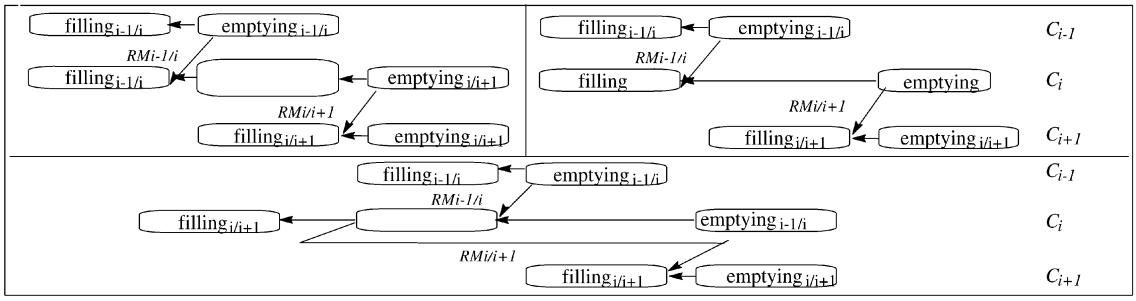

Fig. 7. $RM_{i-1/i}$ passing $RM_{i/i+1}$

cursors pointing to the same record version, i.e., the shared tree being empty. This triggers passing. Next, the rolling-merge cursor of $RM_{i-1/i}$ advances into the emptying tree of $RM_{i/i+1}$ and the original emptying tree of $RM_{i-1/i}$ is virtually deleted as shown in the upper right part of Fig. 7. If $RM_{i-1/i}$ is still faster than $RM_{i/i+1}$, the rolling-merge cursor of $RM_{i-1/i}$ passes the cursor of $RM_{i/i+1}$, they do no longer point to the same record version, and a new tree is virtually created (in the actual implementation, the empty emptying tree is reused). This is shown in the lower part of Fig. 7. We now have exactly the same situation as shown in Fig. 6a. $RM_{i-1/i}$ 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. 4). 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 Sect. 7 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. Conflict testing involves testing two-dimensional intervals for disjointness, which is not hard to implement and does not incur tremendous run-time overhead either. As an optimization, it would nevertheless be desirable to leverage advanced low-overhead implementation tricks along the lines of what [GR93, Moh96, Lom93,

KMH97] have developed mostly for single-dimensional indexes, but this would a subject of future research. 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 Sect. 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 Sect. 4.2, these index structures are emptied and filled in a given order according to the records' keys and timestamps. This suggest the following order for accessing the index structures inside a component: 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 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 Sect. 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 time-range 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 Sect. 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 warm start 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. 5, 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 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.

6.1 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.

Figure 8 shows the rectangles representing the leaf nodes of an example TSB-tree storing account values. The key value min denotes the lower end of the key space. The lower left leaf node is identified by (min, t_0), i.e., it contains record versions with key $\geq min$ and timestamp $\geq t_0$. It is bounded in the key and time dimension by the adjacent nodes $(Chris, t_0)$ and (min, t_{10}), respectively, meaning that it does not contain record versions with key $\geq Chris$ or with a timestamp $\geq t_{10}$. Leaf nodes $(Dave, t_9)$, $(Chris, t_0)$ and (min, t_{10}) are current nodes, since they contain record versions being valid at current time. All other nodes are historical nodes. Note that the two dots at $(Dave, t_9)$ and $(Alex, t_{10})$ represent redundant entries due to splits of current nodes. The entry $(Dave, t_9, 80)$ has to be copied to the current node since it is valid at both times being covered by historical node $(Dave, t_0)$ as well as current node $(Dave, t_9)$. We will discuss details of splitting below.

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].

Figure 9 shows the TSB-tree indexing the data of Fig. 8. We assume the capacity of all nodes being three entries, i.e., each index node and leaf node has at most three index terms or record versions, respectively. The root node points to index nodes (min, t_0) and $(Dave, t_0)$, which contain entries for nodes that are covered by the corresponding time and key range. Index node (min, t_0) contains entries for leaf nodes (min, t_0), (min, t_{10}) and $(Chris, t_0)$ since they contain record versions with a key value $\geq min$ and with timestamp $\geq t_0$. All other leaf nodes are referenced by index node $(Dave, t_0)$ as the keys of all record versions stored in the leaf nodes are greater than or equal to $Dave$.

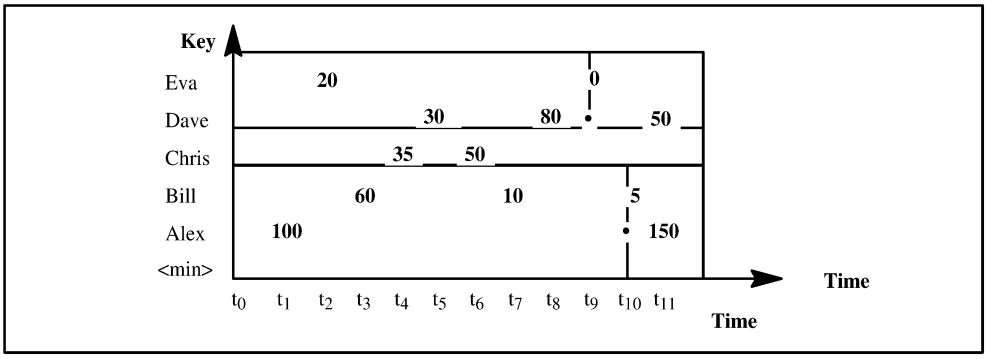

Fig. 8. Rectangles representing leaf nodes of a TSB-tree

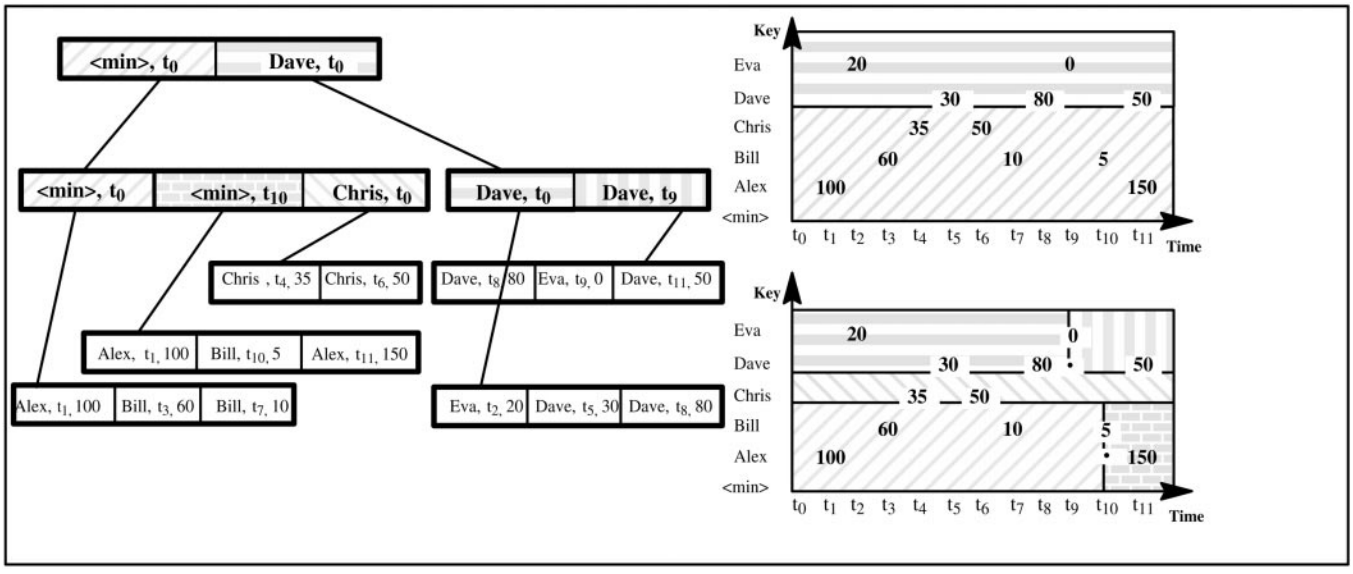

Fig. 9. TSB-tree nodes and covered rectangles

6.2 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.

In the example of Fig. 9, looking for the balance of Eva's account as of time t_6 is done as follows. Starting at the root, node the two index terms (min, t_0) and $(Dave, t_0)$ have to be examined. As $Dave$ is the maximum key value less than Eva, the algorithm follows the pointer of index term $(Dave, t_0)$. At the index node, the term $(Dave, t_9)$ is discarded because its timestamp is greater than t_6 and thus it does not contain relevant data. So the algorithm follows entry $(Dave, t_0)$ and finally returns $(Eva, t_2, 20)$, indicating that the account of Eva was updated to 20 at time t_2 and was not changed until t_6.

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 across pages, however, neither in key nor in time dimension.

7 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 with our implementation 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.

7.1 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.

7.1.1 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 120 MB of raw data. The size of a disk block was 8 KB in all experiments, for LHAM and the TSB-tree. We used an LHAM structure of three components with a capacity ratio of 4; component capacities were 8 MB for C_0, 32 MB for C_1, and 128 MB for C_2. Both disk components resided on the same physical disk. We used a buffer of 1 MB for blocks read in a multi-block I/O and a buffer of 1 MB for single blocks. This results in a total of 10 MB 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 eight blocks per multi-block I/O, the throughput of LHAM was always more than six 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 8 KB. 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 120 MB 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 168 MB, but only 122 MB 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.

7.1.2 Queries

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

1. $< key, timepoint >$,
2. $< keyrange, timepoint >$,
3. $< key, timerange >$, and
4. $< keyrange, timerange >$.

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

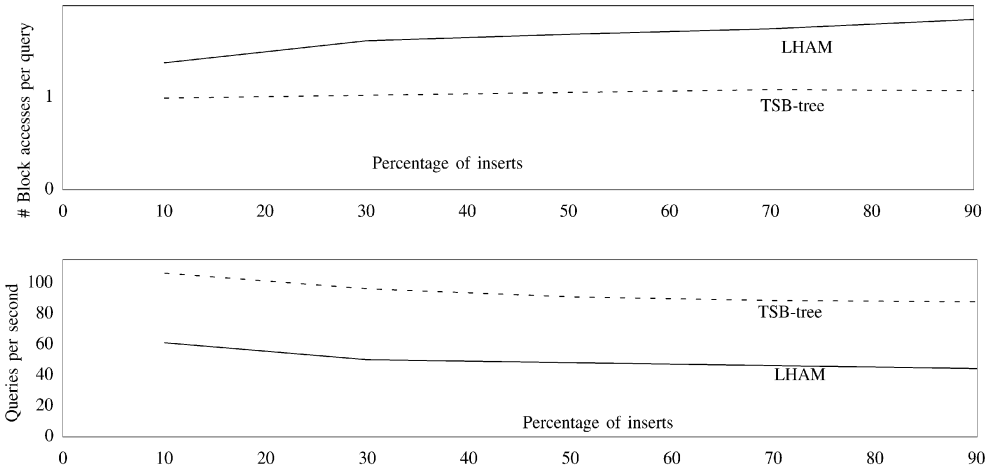

Fig. 10. Performance of queries of type <key, timepoint=now>

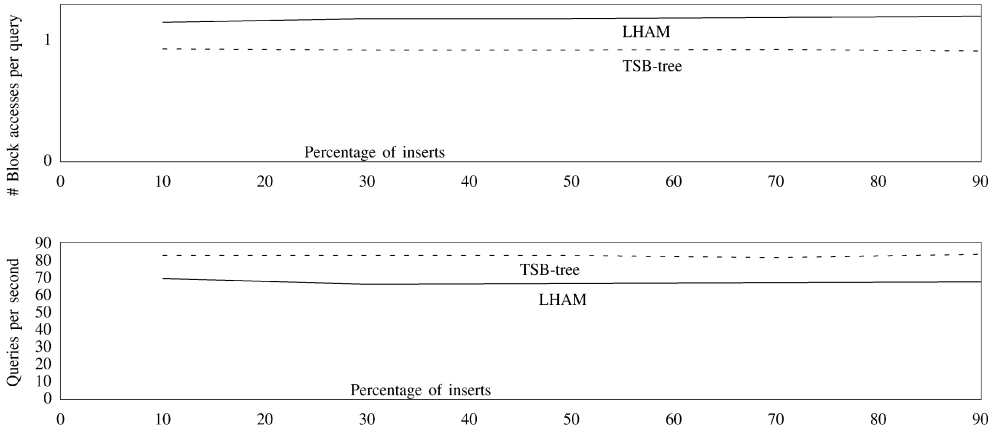

Fig. 11. Performance of queries of type <key, timepoint=random>

ratio. We give results for the number of block accesses required per query and the (single-user) throughput in queries per second.

Queries of type $< key, timepoint >$

Figure 10 shows in its upper chart 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 10–20%, resulting in relatively many versions per record. The case with 10% insertions is closest to the update-only setting that we have mathematically analyzed in Sect. 3.2. There we predicted an expected number of approximately 1.3 block accesses for a random point query with time parameter now. The actually measured figure for 10% insertions is 1.37; so our analytical prediction is reasonable. Finally, note that the absolute performance is good anyway; so this query type is not a major performance concern. In its lower chart, Fig. 10 shows the (single-user) throughput achieved by LHAM and the TSB-tree for this type of query. The curve in this chart is similar to the upper chart in Fig. 10, as LHAM cannot benefit from multi-block I/O for this type of query.

The situation changes when we consider arbitrary timepoints instead of solely the current time. Figure 11 shows 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. This effect has been predicted with reasonable accuracy by our mathematical analysis in Sect. 3.2. There we derived an expected number of 1.123 block accesses for a random key-time point query, whereas the actually measured number is 1.15 (for 10% insertions).

Table 1. Insert performance

	LHAM	TSB-tree 10% inserts	TSB-tree 50% inserts	TSB-tree 90% inserts
Throughput (Inserts/s)	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	Current/Hist.DB: 64/211	Current/Hist.DB: 148/175	Current/Hist.DB: 192/121

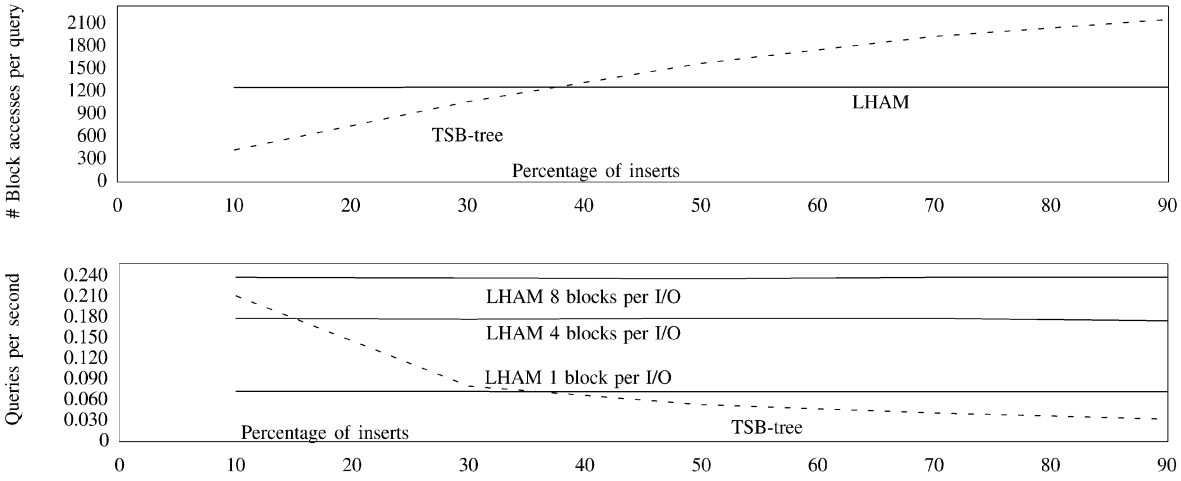

Fig. 12. Performance of queries of type <key range 10%, timepoint=now>

Queries of type $< keyrange, timepoint >$

The performance of $< keyrange, timepoint >$ queries with a key range of 10% of all keys and a timepoint of now is shown in Fig. 12. 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 the lower chart of Fig. 12. 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. Figure 12 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 across pages. When the current database is large, LHAM outperforms the TSB-tree even without multi-block I/O.

Queries of type $< key, timerange >$

Figure 13 shows the performance of $< key, timerange >$ 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 needed 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.

Queries of type $< keyrange, timerange >$

Finally, we consider the performance of $< keyrange, timerange >$ queries. Figure 14 shows the results for a key range of 10% and a time range of 10%. The results are similar to $< keyrange, timestamp >$ queries as shown in Fig. 12.

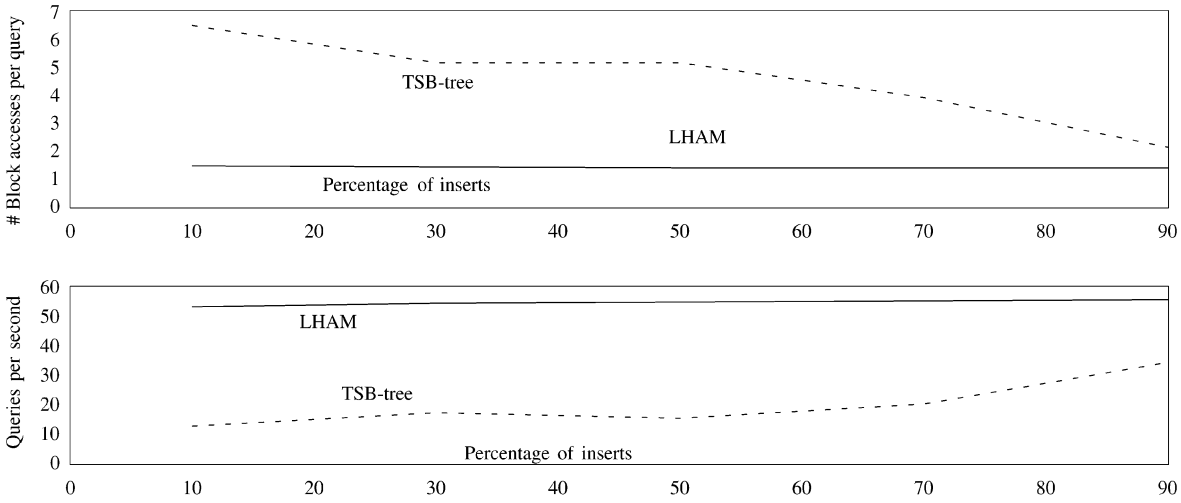

Fig. 13. Performance of queries of type <key, time range 50%>

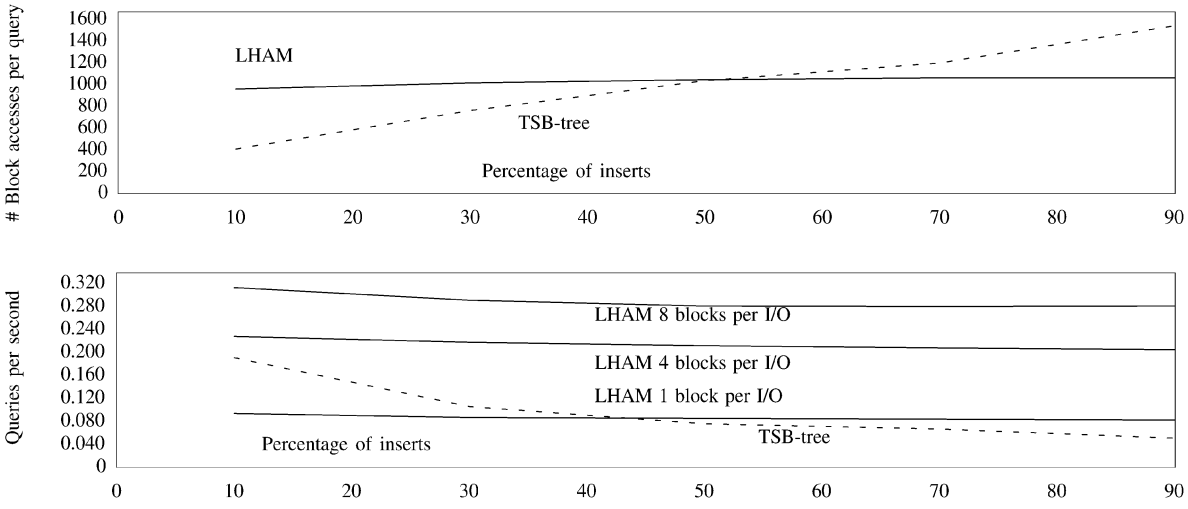

Fig. 14. Performance of queries of type <key range 10%, time range 10%>

Again, similar results have been obtained for other settings of the range widths.

7.1.3 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 Sect. 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.

7.2 Discussion

LHAM's major strength, as evidenced by the presented experiments, is its high throughput for insertions and record updates. Even without multi-block I/O, LHAM outperforms standard index trees like the TSB-tree by a factor of 3; with multi-block I/O, the measured gain exceeded a factor of 6. Note that it is LHAM's specific approach of using rolling merges, similar to bulk-loading techniques for index structures, that provides the opportunity for multi-block I/Os; such opportunities are usually absent in conventional index structures.

As for queries, our experiments have shown that LHAM is competitive to one of the best known index structures for transaction-time temporal data, the TSB-tree. For exact-match queries with a given key and timepoint, the TSB-tree still has an advantage over LHAM, but the conceivably worst case that LHAM needs to search all (non-archive) components is atypical. Rather the experiments have shown that LHAM is on average only 10–30% slower than the TSB-tree, which nicely coincides with our analytical predictions (see Sect. 3.2).

For time-range queries with a given key, i.e., so-called "time-travel" queries, LHAM even outperforms the TSB-tree by a significant margin, because its rolling merges provide for better clustering of a key's record versions across multiple blocks and LHAM can then intensively exploit multi-block I/O for such queries. On the other hand, for key-range queries with a given timepoint, LHAM loses this advantage and is outperformed by the TSB-tree when the average number of versions per key becomes sufficiently high.

A similar tradeoff situation has been observed in the experiments for the most general query type with both key ranges and time ranges. Depending on the width of the ranges and the average number of versions per key, LHAM may still win but can also be outperformed by the TSB-tree under certain circumstances. Especially, when the number of keys that fall into the query's key range becomes small, the TSB-tree is the clear winner. In the extreme case when all matches to a key-range query reside in a single block, LHAM may have to pay the penalty of searching multiple components, whereas the TSB-tree could almost always access the qualifying versions in a single disk I/O.

Queries may be adversely affected by rolling merges. In the worst case, a query may have to traverse three index trees within a single component, namely, the filling tree fed from the next higher storage level and an emptying and filling tree for the rolling merge to the next lower level. However, this extreme case can arise only when the query executes while two rolling merges across three consecutive components are in progress simultaneously. As LHAM generally aims to trade a major improvement in insert/update performance for a moderate potential degradation in query performance, this effect is not surprising. In the measurements performed with concurrent updates, the adverse effects from ongoing rolling merges were rather small, however, so that this conceptual penalty for queries appears acceptable.

All our measurements have focused on the non-archive components. With archive components that are formed by redundant partitioning, LHAM's performance should remain unaffected. However, its space overhead would drastically increase, ultimately consuming considerably more space than the TSB-tree. Note, however, that there is an inherent space-time tradeoff in the indexing of versioned records and multi-dimensional data in general. We do advocate redundant partitioning for archive components in LHAM mainly to make these components "self-contained" in the sense that they completely cover a given time period. This way, LHAM can easily purge all versions for an "expired" time period by discarding the appropriate components and the underlying WORM media or tapes. The same kind of "garbage collection" is not easily accomplished with a structure like the TSB-tree. When this issue is considered less important, then LHAM could be configured with non-redundant partitioning for archive components as well.

8 Comparison to similar approaches

In this section, we discuss two approaches for the bulk loading of index structures which use techniques similar to LHAM.

8.1 Stepped merge

[Jag97] presents a method for efficient insertion of non-temporal, single-dimensional data into B$^+$-trees. Similarly to LHAM, a continuous reorganization scheme is proposed, based on m-way merging. Like LHAM, incoming data is stored in a main-memory buffer first. When this buffer becomes full, it is written to disk, organized as a B$^+$-tree (alternatively, a hash-based scheme is discussed in [Jag97], too). This is repeated m times, each time creating a new, independent B$^+$-tree. Figure 15 illustrates the further processing. After m B$^+$-trees have been created, they are merged by an m-way merge into a new B$^+$-tree. N levels, each consisting of m B$^+$-trees, are considered. Each m-way merge migrates the data to the next level. The final level N contains a single target B$^+$-tree, denoted root B$^+$-tree in [Jag97]. Whenever m B$^+$-trees have been created at level $N-1$, these m trees together with the root tree itself are merged by an $(m+1)$-way merge into a new root tree. For this merge, the root B$^+$-tree is both source and destination, as indicated by the self-referencing arc in Fig. 15. The stepped-merge approach supports the efficient insertion of data, but penalizes queries heavily, as each query may have to look up all $N*m$ component trees.

In terms of this approach, LHAM can be characterized as performing a two-way merge (of the stepped merge's level-$(N-1)$-to-level-N kind) whenever data is migrated to the next component in the LHAM storage hierarchy. A rolling merge from component C_i to component C_{i+1} reads both components and writes back a new component C_{i+1}. Similar ro the root B$^+$-tree of [Jag97], component C_{i+1} is both source and destination of the merge. At each level of the LHAM component hierarchy, only a single B$^+$-tree exists (unless a rolling merge is in progress, which creates temporary trees).

In contrast to [Jag97], LHAM components implement a partitioning of the search space (with respect to the time dimension). In [Jag97], all $N*m$ B$^+$-trees may have overlapping key ranges, whereas (memory and disk) components in LHAM cover disjoint areas in key-time space. Depending on its time range, a query in LHAM needs to access only a few of the LHAM components, so that query execution in LHAM is much more efficient. Furthermore, the stepped-merge approach does not consider the time dimension at all, and also disregards archiving issues.

8.2 Buffer-tree-based bulk loading

[BSW97] presents an approach for the bulk loading of multi-dimensional index structures, e.g., R-trees. The basic idea

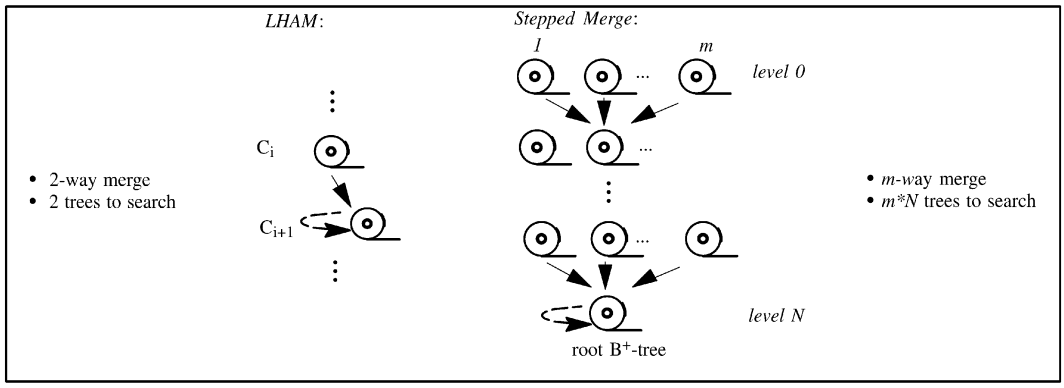

Fig. 15. Illustration of LHAM vs. stepped merge

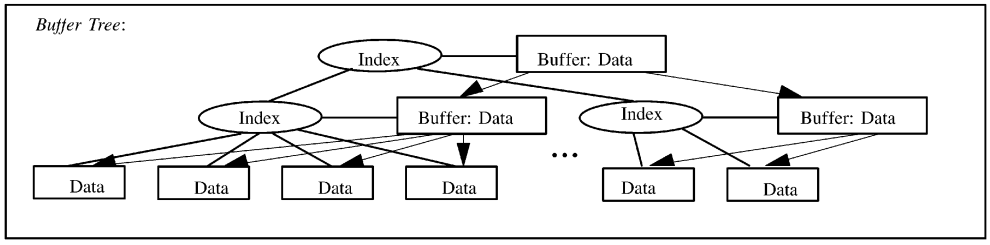

Fig. 16. Illustration of a buffer tree

is to create unsorted sequences of records, where each sequence covers a (multi-dimensional) range of the data space that is disjoint to the ranges covered by the other sequences. A set of such record sequences is managed in a special balanced tree, the buffer tree, where each tree node corresponds to a disk page but is extended with a multi-page disk-resident buffer (hence the name of the tree). Such a buffer tree is sketched in Fig. 16; this data structure has originally been introduced for batched dynamic operations in computational geometry by [Arge95]. Incoming records are migrated through the buffer tree until they reach the leaf level, which corresponds to the leaf level of the target index structure to be built.

Similarly to LHAM and [Jag97], the efficient migration of records is the key property. Incoming data is first inserted into the buffer of the root node. When the buffer becomes full, its data is partitioned according to the key ranges defined by the index nodes on the next lower level of the tree. This process is repeated until the leaf level is reached. A full buffer is read sequentially from disk, and the partitions for the next lower level of the tree are built in working areas in main memory. Whenever a working area becomes full, it is appended to the corresponding lower level node buffer by a single disk write. This way, all data is read and written only once at each level of the buffer tree, resembling the component stages of LHAM.

After all data has been migrated to the leaf level of the buffer tree, the leaves of the bulk loader's target index are completely built and the higher levels of the buffer tree are discarded. Then, 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 with regard to queries against a buffer tree lies in the fact that one would have to search the potentially large unsorted sequences of records at the index levels of the buffer tree, i.e., in the node buffers. While a query can easily navigate through the buffer tree, the sequential search inside a node buffer is expensive, as several disk blocks have to be read. In contrast, LHAM provides substantially better query performance.

9 Extensions and generalizations of LHAM

9.1 Tunable redundancy between disk components

In LHAM as presented in this paper, querying record versions that have not been updated for a long time may force the query to access several components until the record version is found. Both our analytical and experimental results have shown that this is not a severe problem under usual conditions where most records are updated frequently. However, in special applications with a high skew in the update rates of different records, query performance would be susceptible to degradation. A solution is to store record versions redundantly, like the TSB-tree. A record version that is under migration from component C_i to component C_{i+1} can simply be kept redundantly in component C_i (unless its timestamp happens to equal the new time boundary between C_i and C_{i+1}). During the next rolling merge, the redundant record version is deleted from C_i if there is a newer version of the same record under migration to C_{i+1}. In this case, the redundant record version is replaced by the new one. Otherwise, it is kept in component C_i. So there is always at most one redundant version per component for the same key.

The obvious caveat about this scheme is its increased space consumption, as well as its increase in the number of I/Os during the rolling merges. For almost all keys, a redundant version is stored in each component, and this version

has to be read and written during rolling merges just like the non-redundant record versions. On the other hand, we could now guarantee that each key-timepoint query would have to access only a single component.

The overhead introduced by the redundancy can be reduced if we drop the requirement that querying a single record version must access no more than a single component. Limiting the number of visited components to a small constant would already avoid the worst case situation that a query has to access all components C_0 to C_k, while at the same time substantially reducing the redundancy overhead in terms of space and I/Os.

Assume we want a key-timepoint query to access no more than two components in the worst case. This can be achieved by using the scheme of storing redundant records as sketched above, with a single modification: a redundant copy of a record version in component C_i is created during a rolling merge only if there is no successor of this record version left in C_i after the rolling merge has finished. It is easy to see that, with the modified redundancy scheme, no more than two components have to be accessed by a key-timepoint query. Suppose that the queried record version is neither present in the first nor in the second component searched. In this case, the second component does not contain any version of the record, as such a version would match the query. However, in this case, our redundancy scheme would have created a redundant copy of the record version in the second component.

In the case of uniform update rates across keys and reasonably sized comonents that are able to store at least one version for each key, the modified redundancy scheme would not create any redundant versions at all. For records with low update rates, however, the scheme improves query performance by limiting the number of components that have to be accessed.

The above-sketched form of selective redundancy can be exploited also to improve the query performance for keys that are frequently queried, but only infrequently updated. It is possible to dynamically replicate versions of such keys in higher level components, even if they had already been migrated to lower components. So the partitioning invariant given by the low_i and $high_i$ boundaries of a component C_i can be deliberately relaxed to keep older versions of "hot" keys closer to C_0 or even in the main-memory component itself.

9.2 Choice of index structures within LHAM components

For key-range queries for a single timepoint, using a B^+-tree within each LHAM components is by far not an optimal solution, as record versions are primarily clustered by key, so that multiple versions for the same key lie in between successive keys. In principle, we are free to use another index structure within components; even different structures for different components are conceivable. The problem, however, is that such alternative structures have to be efficiently read, merged with a second structure of the same kind, and rebuilt during each rolling merge. This requires the indexed data to be readable and writable in a predefined order while creating the merged data in the same order. In addition, the data should be read and written using multi-block I/O, and each block should be read and written only once during an entire rolling merge.

Using a TSB-tree, for example, the above properties can be satisfied only for current nodes. For historical nodes, there is no scan order such that each node of the two source trees is read only once during a rolling merge. Obviously, this problem can be addressed by buffering nodes during the rolling merges. Unfortunately, there is no sufficiently small upper bound for the number of necessary node buffers (where "sufficiently small" would be logarithmic in the data volume).

We are currently investigating other index schemes, where multi-dimensional space-filling curves such as the Peano curve (also known as Z-order) or Hilbert curve can be superimposed on the data [Fal96]. Such curves define a linear order for multi-dimensional data. During a rolling merge, record versions can then be read and written in the order given by the space-filling curve. The same is true for writing the merged structure back to disk, as record versions are inserted in curve order. Based on this idea, it appears to be feasible to employ an R-tree within each LHAM component, with the additional benefit that LHAM's rolling merges would automatically provide good clustering, both within and across pages, based on the Peano or Hilbert curve. The systematic study of such approaches are a subject of future research.

9.3 A two-dimensional LHAM directory

In the current LHAM version, the LHAM directory partitions only the time axis. Therefore, each LHAM component covers the whole key space. This can be a potential drawback, especially when the update rate of records in different key ranges varies widely, i.e., if there are hot (i.e., frequently updated) and cold (i.e., infrequently updated) regions of keys. Partitioning the data also by key would allow LHAM to use key-range-dependent values for the component's time boundaries. In cold key regions, the timespans covered by components should be long, whereas they would be rather short for hot key ranges. This would reduce the number of components that queries have to access, or, if redundancy is used, the number of redundant record versions.

To this end, a two-dimensional LHAM directory would be needed. As the directory would still be a very small, memory-resident data structure, the implementation of such an extended directory does itself not pose any problems. However, a two-dimensional directory would increase the complexity of rolling merges. Unless all components involved in a rolling-merge partition the key space in the same way, rolling merges now require more than a single source and a single key component. If the key partitioning is constructed by key splits of components like in the TSB-tree, we obtain specific structural constraints for the directory layout in the two-dimensional space. In particular, rolling merges will then again have a single source component, but multiple destination components. As a consequence, there will only be a single time boundary betweeen the involved components. This variant seems to be most promising.

Queries can strongly benefit from such a two-dimensional LHAM directory, since the number of components they need

to access could be reduced. In addition, the performance of key-range queries with a component organized as a B^+-tree can be improved. When a component stores too many versions of the same key, it can simply be split into two components with the same key range, without affecting other components with the same timespan, but mostly holding records with lower update rates.

With a two-dimensional LHAM directory along the sketched lines, the number of components a record lives in until it finally migrates to archival components depends on its key. The number of I/Os required to perform a rolling merge now depends on the sizes of all involved components, and can no longer be determined by a single size ratio. This would obviously call for a more advanced mathematical analysis.

10 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 "typical-case" workloads. LHAM's average-case performance is consistently good.

Target applications for LHAM include very-large-data data warehouses. In such settings, often the batch window for periodically loading or refreshing the warehouse can be a severe limitation, thus not being able to build (or maintain) all otherwise desired indexes. LHAM's excellent insertion throughput would substantially shorten the time for index maintenance, thus making tight batch windows feasible. Some warehousing applications, especially in the telecommunication industry, require continuous, near-real-time maintenance of the data. Once the update rate and the size of the warehouse exceed certain manageability thresholds, such a warehouse may not afford any conventional indexes at all. Again, LHAM is much better geared for coping with very high insertion/update rates against terabyte warehouses.

References

[AGL98] Agrawal R, Gunopulos D, Leymann F (1998) Mining Process Models from Workflow Logs. In: Schek HJ, Saltor F, Ramos I, Alonso G (eds) Proc. Int. Conf. on Extending Database Technology (EDBT), 1998, Valencia, Spain. Springer, Berlin Heidelberg New York, pp 469–483

[Arge95] Arge L (1995) The Buffer Tree: A New Technique for Optimal I/O Algorithms. In: Akl S, Dehne F, Sack JR, Santoro N (eds) Proc. Int. Workshop on Algorithms and Data Structures, 1995, Kingston, Ontario, Canada. Springer, Berlin Heidelberg New York, pp 334–345

[Bec96] Becker B, Gschwind S, Ohler T, Seeger B, Widmayer P (1996) An Asymptotically Optimal Multiversion B-tree. VLDB J 5(4): 264–275

[BSW97] Bercken J van den, Seeger B, Widmayer P (1997) A Generic Approach to Bulk-Loading Multidimensional Index Structures. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proc. VLDB Conf, 1997, Athens, Greece. Morgan Kaufmann, San Mateo, California, pp 406–415

[CACM98] Special Section on Industrial-Strength Data Warehousing. Commun ACM 41(9):28–67

[EKW91] Elmasri R, Kim V, Wuu GTJ (1991) Effcient Implementation for the Time Index. In: Proc. Data Engineering Conf, 1991, Kobe, Japan, IEEE Computer Society, Los Alamitos, Californ, pp 102–111

[EWK93] Elmasri R, Wuu GTJ, Kouramajian V (1993) The Time Index and the Monotonic B^+-tree. In: Tansel AU, Clifford J, Gadia S, Jajodia S, Segev A, Snodgrass R (eds) Temporal Databases: Theory, Design, and Implementation. Benjamin Cummings, New York, pp 433–456

[Fal96] Faloutsos C (1996) Searching Multimedia Databases By Content. Kluwer Academic, Amsterdam

[GP87] Gray J, Putzolu F (1987) The Five-Minute Rule for Trading Memory for Disc Accesses and the 10-Byte Rule for Trading Memory for CPU Time. In: Dayal U, Traiger IL (eds) Proc. SIGMOD Conf, 1987, San Francisco. California. ACM, New York, pp 395–398

[GR93] Gray J, Reuter A (1993) Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, California

[Gut84] Guttman A (1984) R-trees: A Dynamic Index Structure for Spatial Searching. In: Yormark B (eds) Proc. SIGMOD Conf, 1984, Boston, Massachusetts. ACM, New York, pp 47–57

[Jag97] Jagadish HV, Narayan PPS, Seshadri S, Sudarshan S, Kanneganti R (1997) Incremental Organization for Data Recording and Warehousing. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proc. VLDB Conf, 1997, Athens, Greece. Morgan Kaufmann, San Mateo, California, pp 16–25

[Kol93] Kolovson CP (1993) Indexing Techniques for Historical Databases. In: Tansel AU, Clifford J, Gadia S, Jajodia S, Segev A, Snodgrass R (eds) Temporal Databases: Theory, Design, and Implementation. Benjamin Cummings, New York, pp 418–432

[KMH97] Kornacker M, Mohan C, Hellerstein JM (1997) Concurrency and Recovery in Generalized Search Trees. In: Peckham J (eds) Proc. SIGMOD Conf, 1997, Tucson, Arizona. ACM, New York, pp 62–72

[Lom93] Lomet D (1993) Key Range-Locking Strategies for Improved Concurrency. In: Agrawal R, Baker S, Bell DA (eds) Proc. VLDB Conf, 1993, Dublin, Ireland. Morgan Kaufmann, San Mateo, California, pp 655–664

[LS89] Lomet D, Salzberg B (1989) Access Methods for Multiversion Data. In: Clifford J, Lindsay BG, Maier D (eds) Proc. SIGMOD Conf, 1989, Portland, Oregon. ACM, New York, pp 315–324

[LS90] Lomet D, Salzberg B (1990) The Performance of a Multiversion Access Method. In: Garcia-Molina H, Jagadish HV (eds) Proc. SIGMOD Conf, 1990, Atlantic City, New Jersey. ACM, New York, pp 353–363

[Moh96] Mohan C (1996) Concurrency Control and Recovery Methods for B^+-Tree Indexes: ARIES/KVL and ARIES/IM. In: Kumar V (ed) Performance of Concurrency Control Mechanisms in Centralized Database Systems. Prentice Hall, Englewood Cliffs, N.J., pp 248–306

[MOPW98] Muth P, O'Neil P, Pick A, Weikum G (1998) Design, Implementation, and Performance of the LHAM Log-Structured History Data Access Method. In: Gupta A, Shmueli O, Widom J (eds) VLDB Conf, 1998, New York City, New York. Morgan Kaufmann, San Mateo, California, pp 452–463

[MSS95] Proceedings of the 14th IEEE International Symposium on Mass Storage Systems, 1995, Monterey, California. IEEE Computer Society, Los Alamitos, California

[OCGO96] O'Neil P, Cheng E, Gawlick D, O'Neil E (1996) The Log-Structured Merge-Tree (LSM-tree). Acta Informatica 33(4): 351–385

[OW93] O'Neil P, Weikum G (1993) A Log-Structured History Data

Access Method. In: 5th Int. Workshop on High-Performance Transaction Systems (HPTS), 1993, Asilomar, California

[RO92] Rosenblum M, Ousterhout JK (1992) The Design and Implementation of a Log Structured File System. ACM Trans Comput Syst 10(1): 26–52

[Sno90] Snodgrass R (1990) Temporal Databases: Status and Research Directions. ACM SIGMOD Rec 19(4): 83–89

[SOL94] Shen H, Ooi BC, Lu H (1994) The TP-Index: A Dynamic and Effcient Indexing Mechanism for Temporal Databases. In: Proc. Data Engineering Conf., 1994, Houston, Texas. IEEE Computer Society, Los Alamitos, California, pp 274–281

[ST99] Salzberg B, Tsotras VJ (1999) A Comparison of Access Methods for Temporal Data. ACM Comput Surv: in press

[Tan93] Tansel AU, Clifford J, Gadia S, Jajodia S, Segev A, Snodgrass R (Eds) (1993) Temporal Databases: Theory, Design, and Implementation. Benjamin Cummings, New York

[TK95] Tsotras VJ, Kangelaris N (1995) The Snapshot Index: An I/O-Optimal Access Method for Timeslice Queries. Inf Syst 20(3): 237–260