An Efficient Multiversion Access Structure

Peter J. Varman and Rakesh M. Verma

Abstract—An efficient multiversion access structure for a transaction-time database is presented. Our method requires optimal storage and query times for several important queries and logarithmic update times. Three version operations—inserts, updates, and deletes—are allowed on the current database, while queries are allowed on any version, present or past. The following query operations are performed in optimal query time: key range search, key history search, and time range view. The key-range query retrieves all records having keys in a specified key range at a specified time; the key history query retrieves all records with a given key in a specified time range; and the time range view query retrieves all records that were current during a specified time interval. Special cases of these queries include the key search query, which retrieves a particular version of a record, and the snapshot query which reconstructs the database at some past time. To the best of our knowledge no previous multiversion access structure simultaneously supports all these query and version operations within these time and space bounds. The bounds on query operations are worst case per operation, while those for storage space and version operations are (worst-case) amortized over a sequence of version operations. Simulation results show that good storage utilization and query performance is obtained.

Index Terms—Transaction-time database, multidimensional data, access methods, data structures, indexing, I/O complexity.

1 Introduction

Several applications in commercial and manufacturing enterprises need access to past versions of data. In many conventional databases (called snapshot databases in [9]) only the most current version of the data is stored. As records are updated or deleted, previous versions of the records are lost from the database. In several applications, however, it is necessary to retain and maintain access to the old versions of these records, so that queries about the past states of the database can be answered. A transaction-time database [22] provides automatic storage and management of old (versioned) data. For definitions and concepts related to temporal databases see [3], [9], [17], [19], [21], [23].

As an illustrative example consider a bank database tracking the transactions associated with different accounts. Fig. 1a shows a hypothetical snapshot relation in this database; only the most recent balance for an account is stored in the relation. In contrast, in a transaction-time database a new version of a record is created with each update as shown in Fig. 1c, following the transactions noted in Fig. 1b. Notice that each record is now timestamped with the time at which the update took place. This is also assumed to be the time at which the new data values take effect and the old ones become obsolete. As a consequence, several queries that were not possible in the snapshot database can now be made. For instance, one can ask for the balance in account No. 2321000 on May 3, 1993, or for the balance history of account No. 2321000 between January and May 1993. A global query may ask for the balance in all accounts with account numbers between 2321000 and 2323000 at some time in the past, or to report all accounts with balances between $1,000 and $2,000 on May 1, 1993. It is necessary to organize the data so that queries such as these can be answered efficiently, while allowing for database-modifying operations (also referred to as version operations) that insert new records into the database, and update or delete current records. Such a structure is referred to as a *multiversion access structure* or MVAS.

We present an efficient multiversion access structure for transaction-time databases [23] in this paper. Version operations `insert`, `update`, and `delete` are permitted on the current data, and a rich variety of query operations are supported. The storage, as well as the time complexities for the query operations are asymptotically optimal. The time complexity for version operations is optimal for a method that simultaneously supports all these query operations optimally. We use the phrase *simultaneously optimal* for a set of queries to mean that the storage is optimal, the query times are optimal for the given set of queries, and the time for version operations matches the lower bound on update time for *at least* one of the queries in this set.

A brief comparison with related work is provided below. More detailed comparisons are provided in Section 8. First, most multiversion access structures (exceptions are noted and discussed in Section 8) do not explicitly handle the `delete` operation. Among designs handling deletes, the worst-case storage bounds of our design significantly improves (by about 85 percent) the best previously reported bound [2]. This is achieved by a combination of using more storage-conservative overflow and underflow policies, and coming up with a tighter (and nontrivial) analysis. Second, we present a new design for handling the *key-history* query, that retrieves all versions of a specified key in a specified time-interval. Most previous methods (see [18]) are inefficient for this query. Access lists [1] and the ST-tree [7] achieve optimal query time for the key-

- *P.J. Varman is with the ECE Department, Rice University, Houston, TX 77251. E-mail: pjv@rice.edu. Part of this work was performed while Varman was visiting at the School of Applied Science, Nanyang Technological University, Singapore.*
- *R.M. Verma is with the Department of Computer Science, University of Houston, Houston, TX 77204. E-mail: rmverma@cs.uh.edu.*

Manuscript received Oct. 3, 1994; revised June 12, 1995.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number 104404.1.

(a)

Acct. No.	Curr. Bal.
2321000	1500.00
2322000	2550.00
2323000	10000.00
⋮	⋮

(b)

Time	Acct. No.	Transaction
5/1/93	2321000	Cred. 500.00
5/3/93	2322000	Deb. 200.00
5/4/93	2321000	Deb. 150.00
⋮	⋮	⋮

(c)

Time	Acct. No.	Balance
4/30/93	2321000	1500.00
5/1/93	2321000	2000.00
5/4/93	2321000	1850.00
4/30/93	2322000	2550.00
5/3/93	2322000	2350.00
4/30/93	2323000	10,000.00

Fig. 1. Example snapshot and transaction-time database relation.

history query, but have much larger worst-case storage requirements.

Finally, like the related design of [2], our structure is simultaneously optimal for *key-range* queries that retrieve a range of keys at a given time, and *time-range* queries that retrieve all keys current in a time range.

The rest of the paper is organized as follows. Section 2 gives the definitions used in the paper, and states the worst-case time bounds for different operations. In Section 3, we describe the access structure and the version operations; and query operations are discussed in Section 4. The analysis of the storage requirements is presented in Section 5; and in Section 6, we describe simulation results to measure average storage utilization and query performance. In Section 7, we describe augmenting the structure with modified access lists to handle key-history queries. In Section 8, we discuss previous work on the design of multiversion access structures and compare it to the method described in this paper. The paper concludes with a summary in Section 9.

2 Definitions

A record is the basic unit of information. The records may contain the actual data or may be pointers to the objects of interest. In either case we refer to these as *data records*. Two types of operations may be performed on the database: *version* operations, which are operations that modify the database; and *query* operations that retrieve selected data records. The version operations are *insert*, *update*, and *delete*. The term *version number* is used to keep track of the total number of version operations performed on the database [4]; every version operation increases the version number by one.

Each version operation has a timestamp associated with it. The timestamps of successive version operations must form a monotonically increasing sequence, but otherwise there is no restriction on them. Version operations can be performed on only the latest version of records, i.e., on the current database. However, queries may be posed about objects present at any time, past, or current. Such a model is referred to as a transaction-time database [9], [23]. Note that timestamps are not necessarily consecutive integers, and there is no direct relationship between version numbers and timestamps except that events with later version numbers have higher timestamps than events with earlier version numbers. Finally, as expected for transaction-time data, we assume step-wise constant behavior, i.e., the value of a record with a given key remains unchanged between successive updates to the record with that key.

Each record has a unique `key` and a pair of timestamps `start` and `end`. The lifespan of the record is from `start` to `end`. The special timestamp value $ is used to indicate the present time. A record is created at some time t_0 using an `insert` operation and its lifespan is from t_0 to $. The record remains current (or live) until the time at which it is either `updated` or `deleted`. If the record is updated at time t_1, a new version of the record is created that is live from t_1 to $, and the `end` timestamp of the previous version becomes t_1. If the record is deleted at time t_1, then no further version of the record is created. A record is said to be current (or live) at time i if i is greater than or equal to its `start` timestamp and less than its `end` timestamp. A *present version* record is one which is current at the present time, time $.

Fig. 2 shows these definitions graphically. Records with keys A, B, D, and C are inserted at times 1, 2, 3, and 4, respectively. At time 5, A gets updated and its new version (denoted as (A, 2)) remains current until time 8, when another update is made to it. Similarly, B gets updated at times 7 and 12; and D at times 6 and 10. C gets deleted at time 9 and no new version of it is therefore created. The current entries at time 3 are (A, 1), (B, 1), and (D, 1). If the current time is 13, then the present version entries are (A, 4), (B, 3), and (D, 3).

Several of the most important and frequent query operations discussed in the literature, can be performed using the proposed access structure. For this set of queries our structure is simultaneously optimal. In the following, let B denote the size of a block (in records), N denote the current version number, N_t the number of records that are current at time t, R the number of records in the output of a query, and w the minimum number of records current at time t that are present in every block along a search path. We will show later that w is at least as large as $B/5$, even for a worst-case sequence of operations. We denote the number of block accesses needed for the query by T. The constants in the Θ expressions given below are all very small (more precise expressions are derived in Section 4):

- `Key Search`: Retrieves the record with key `K` current at time `t`. $T = \Theta(\log_w(N/B))$.
- `Key-Range Search`: Retrieves the records with keys in the range $[K_1 \ldots, K_2]$ at time `t`. $T = \Theta(\log_w(N/B) + R/B)$.
- `Key-History Search`: Retrieves the records with key `K` current in the time range $[t_1 \ldots, t_2]$. This operation retrieves the history of the object with key `K` during a time span from t_1 to t_2. Using the C-list described in Section 7, we obtain $T = \Theta(\log_w(N/B) + R/B)$.
- `Snapshot`: Retrieves all records that were current at time t_1. This is an important special case of the Key-Range Search operation.
- `Time-Range View`: Retrieves all records that were current in the time interval $[t_1 \ldots, t_2]$. This is a generalization of a Snapshot Query to all time points in an interval. $T = \Theta(\log_w(N/B) + R/B)$. (Applying the Snapshot operation successively to all time points in the range is much less efficient in general.)

The amortized time required for any of the insert, update and delete operations at time t is $\Theta(\log_w(N_t))$. This is sufficient to update all the different structures that are maintained to handle all the above query operations with the stated time bounds. For convenience, in the rest of the paper the time for version operations is to be understood as amortized time. We end this section by summarizing known bounds for these operations. Using the lower-bound results in [18] an I/O-optimal method for the Key-Range search problem requires $O(N/B)$ space, $O(\log_B N + R/B)$ query time, and logarithmic update processing. Our structure matches the bounds on space, update, and query times for this problem. For the Snapshot problem, the best known lower bounds [18] are $O(N/B)$ space, $O(\log_B N + R/B)$ query time, and $O(1)$ updating. Our method matches the bounds on space and query time, but uses logarithmic update time. Note that any single structure that supports *both* Key-Range and Snapshot queries, must require logarithmic update time, as is achieved by our method. We are unaware of any deterministic solution to

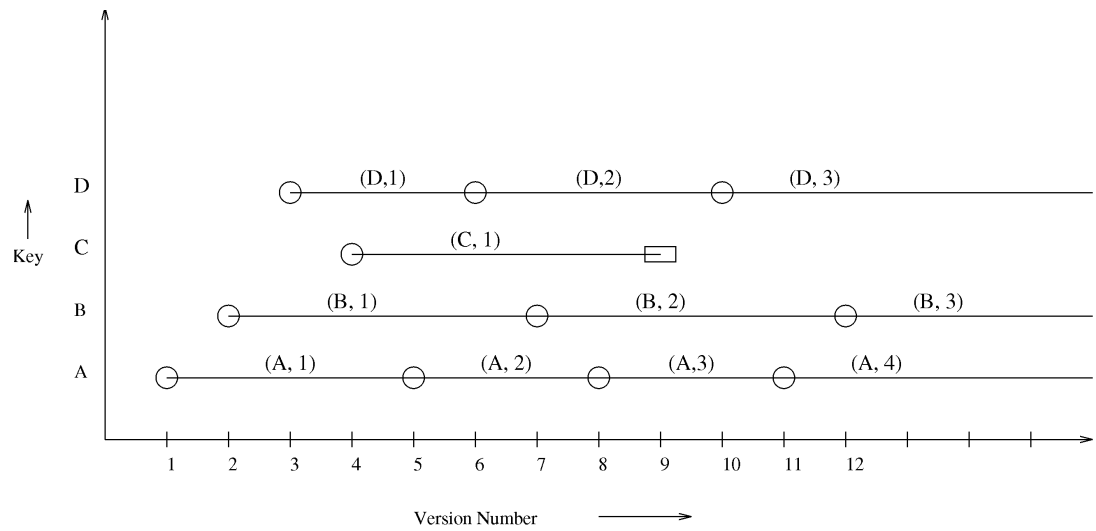


Fig. 2. Illustrating the model.

the Snapshot problem (when considered in isolation) that matches the lower bounds stated above. Recently, Tsotras and Kangelaris [25] presented a randomized optimal solution to this problem. They use a randomized update algorithm based on hashing, which achieves $O(1)$ *expected* amortized update time, and deterministically optimal query time and space. For the key-history query, a lower bound for query-time [18] is $\Omega(\log_B N + R/B)$, which is matched by our structure. The lower-bound on update time for the key-history query is open. We conjecture that logarithmic update time is needed by any deterministic algorithm.

Note that query times depend on N the total number of version operations, while times for version operations (since they only need to access the current database) depend on N_t, the number of records that are live at the current time. This is achieved by keeping a pointer to the root of the current database. Using this pointer the current database can be accessed directly. Whenever the root of the current database changes, the pointer is updated.

3 Multiversion Access Structure

In this section, we describe the structure of the multiversion access structure (MVAS) and explain the fundamental version operations: *insert*, *update*, and *delete*. The time complexity of the version operations is analyzed in Section 3.2. The MVAS is made up of a modified multiversion B+-tree ($\mathcal{T}$). The leaves of $\mathcal{T}$ are *data nodes* and contain the data records. The interior nodes of $\mathcal{T}$ are *index nodes* and contain index records. Fig. 3 shows a hypothetical MVAS with four data nodes, and one index node. For convenience we assume that index nodes and data nodes hold the same number of records. This a pessimistic assumption so if it does not hold our performance will improve.

A data record has the fields [`key, start, end, info`] with obvious semantics. An index record has the fields [`key, start, end, ptr`]; `ptr` is a pointer to a node at the next level in $\mathcal{T}$; the node pointed to will contain keys which are no smaller than `key`, `start` is the time at which the referenced node was created and `end` is the time at which the referenced node died (see Section 3.1). For simplicity, in Fig. 3 a data record is indicated by only its `key` and `start` time. Thus, node *U* contains four data records: `key A` created at time 1 (`(A, 1)`), `key B` created at time 2 (`(B, 2)`) and updated at time 4 (`(B,4)`), and `key D` created at time 3 (`(D, 3)`). Similarly, the index records in the index block are represented only by the `key`, `start`, and `ptr` fields. For instance, the index entry `(A, 5)` points to block *V*, that was created at time 5, and whose keys are no smaller than `A`.

A data record is said to be *live* if its `end` timestamp is \$, i.e., it has not been updated, deleted, or copied to another node. A block containing some live records is a *live* block; else it is *dead*. An index record is *live* if it points to a live block at the next level and *dead* otherwise. In Fig. 3, blocks *U* and *V* are dead, while *X* and *Y* are live. Of the records in *V*, `(A, 1)`, `(B, 4)`, and `(A, 6)` died because they were updated (by `(A, 6)`, `(B, 7)`, and `(A, 8)`, respectively), while `(B, 7)` died when it was copied into block *Y*. In block *X*, record `(D, 3)` is dead, but the other two records are live.

3.1 Version Operations of MVAS

In this section, we describe the *insert*, *update*, and *delete* operations on the MVAS. Assume, for simplicity, that both data and index blocks hold B records. Two parameters—T_H and T_L—are used to control the block occupancy, where $B \geq T_H \geq 4T_L$ (the reasons for these constraints are clarified below).

To *insert* a new record or *update* an existing record, the MVAS is searched starting from the current root to locate the appropriate data block into which the record must be added. (Details of the search procedure are provided in Section 4.) If there is space in the block, the record is simply added to it. If not, an `overflow` occurs, and new blocks are created according to the following rules (let L_A be the number of *live* records in an overflowing block, say A, including the record being inserted):

- `Case 1`. $L_A > T_H$: Create two new blocks `C` and `D`. Copy all the *live* records from `A` into `C` and `D` based on key range, with each of `C` and `D` getting an equal (or differing by one) number of keys.
- `Case 2`. $2T_L \leq L_A \leq T_H$: Create one new block `C`. Copy all *live* records from `A` to `C`.
- `Case 3`. $T_L < L_A < 2T_L$: Identify a live sibling block, `B` (i.e., a live block with adjacent key range). If no such sibling exists, then handle the overflow as in Case 2 above. Otherwise, let L_B be the number of live records in `B`.
  - `A)` If $L_B < 3T_L$: If ($L_A + L_B > T_H$), copy all the live records from `A` and `B` into two new blocks as in `Case 1`; else copy these combined records into one new block as in `Case 2`. Mark `B` as `dead`.

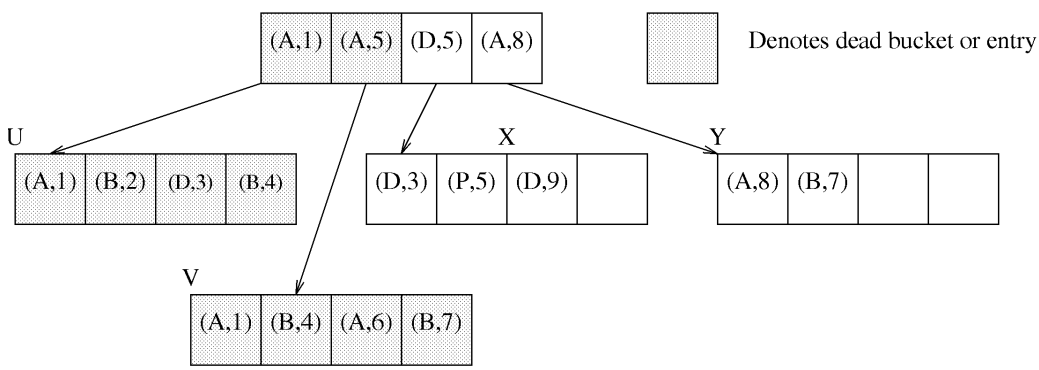


Fig. 3. The Multiversion Access Structure.

B) If $L_B \geq 3T_L$: Copy the L_A live records from A and $(2T_L - L_A)$ records from B with adjacent key values into one new block. Mark these records in A and B as copied.

In each case, A is marked as dead in its parent, and its end timestamp updated to the current version number. Note that B is marked dead only in Case 3A when *all* its live records are copied to another node; in Case 3B only a few (at most $T_L - 1$) live records will be copied from B, which will still retain at least $2T_L + 1$ live records after that. In Case 3B, the range of keys in B may change if its lowest key is copied. Then, a new index entry for B with the new lowest key and current start timestamp is inserted in its parent; the old index entry for B has its end timestamp set to the current version number. This ensures that subsequent searches for the copied records lead to the new block rather than to B. Index records for the newly created block(s) are inserted into A's parent. Note that adding index records may in turn cause an overflow of the index block. This is handled in the same manner as the overflow of a data block just described. Note that Case 3 above does not arise in a WOBT [5] or TSBT [13], since these designs do not treat deletes. Also note that our overflow policy substantially differs from that of [2]. The only time when a sibling block cannot be found is in the rare event that the current database consists of only a single block (the overflowing block). In this case, the live records are copied into a new block as in Case 2.

Fig. 4 shows an example of Cases 1 and 2 for a data block overflow assuming $B = 5$ and (just for illustration) $T_H = 3$. Initially, at time 5 there are three live records (A, 1), (C, 4), and (B, 5). At time 6, the record *D* is inserted in the data block, causing an overflow. Since the total number of live entries (including the one being inserted) is 4, two new blocks are created, and the live records split equally by key range. Two new entries are added to the index node; note that both of these have start time 6, indicating the creation time of the data blocks to which they point. The first index record now points to a dead data block. The second example shows the insertion of a record with key B at time 6. Since there is already a record with key B, this is an update and the total number of live entries on overflow is 3. All these three are copied to a new data block.

Deleting a record at time *i* changes the end field of that record to *i*. The record is now dead. If the delete causes the number of *live entries* in the block to fall to T_L, the block is said to underflow. To avoid fragmenting the live entries over too many blocks, the underflowing block A is merged with a live sibling block B, should one exist, according to the following rules (as noted before, the only time such a sibling cannot be found is in the (pathological) case that the current database consists of only one block (the one which has just underflowed); in this case, no action is taken on an underflow; in the general case, let the sibling B contain L_B *live* entries, and N_T *total* entries):

- Case 1. $L_B \geq 3T_L$: Create a new block C. Copy the T_L live entries of A into C. Mark A as dead. Copy T_L live entries with adjacent key values from B into C, and mark these records in B as copied.

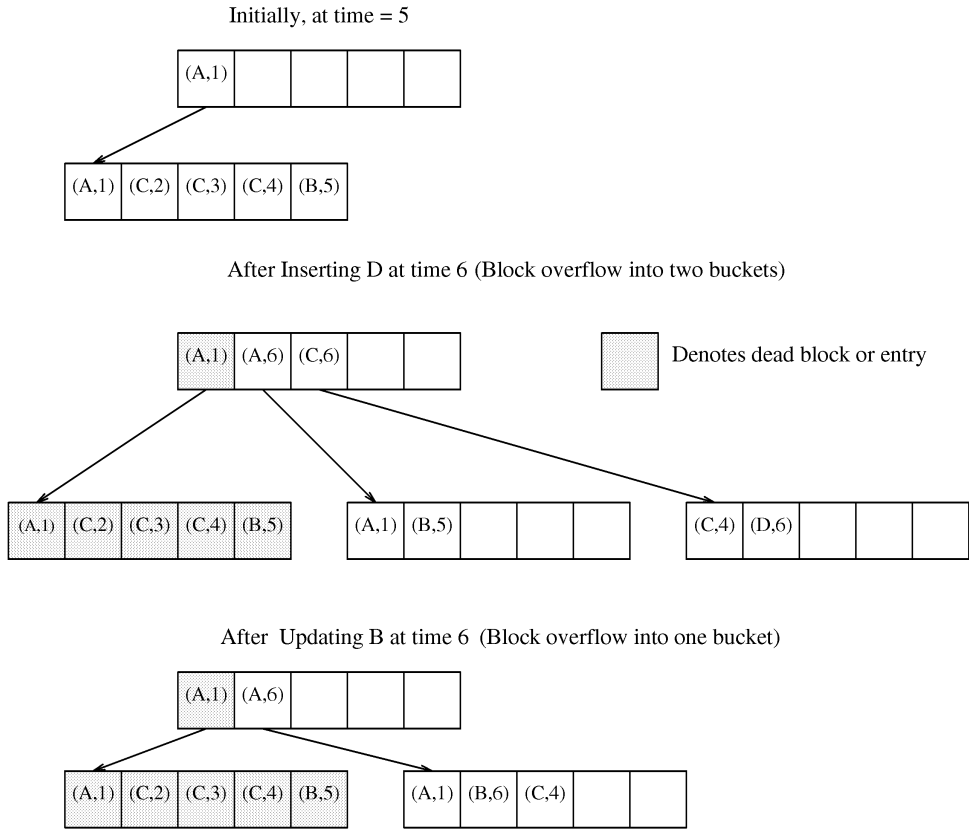


Fig. 4. Illustrating inserts and updates (end times are not shown).

- `Case 2`. $L_B < 3T_L$ and $N_T > (T_H + 1)$: Create a new block `C`. Copy the T_L live entries of `A` into `C`. Mark `A` as dead. Copy all live records from `B` to `C`, and mark `B` as dead.
- `Case 3`. $L_B < 3T_L$ and $N_T \leq (T_H + 1)$: Copy the T_L live entries of `A` into `B`, and mark `A` as dead.

The index entry for `A` (and in `Case 2` for `B` also) is updated to indicate that it is pointing to a dead block. In `Cases 1` and `2`, an index entry for the new block `C` is added to `A`'s parent. In `Cases 1` and `3`, if the lowest key of `B` is changed, a new index entry for `B` with the new lowest key and current `start` timestamp is inserted in the parent of `B`; the old index entry for `B` has its `end` timestamp set to the current version number.

Inserts or deletes of index records are handled in a manner similar to data records. A delete of an index record occurs when the block it points to dies. Note that this is accompanied in `Cases 1` and `2` by the insertion of a new in-dex entry, corresponding to a newly created block. The num-ber of live index entries in a block decreases in `Case 2` when two blocks die and only one new block is created, and in `Case 3` when one block dies and no new block is created.

A special case arises when the smallest key of a block is deleted, and this was the smallest search key at the next (and possibly more) higher index levels. When such a block is split, the smallest key of the newly created block will be larger than its old value. The new index record pointing to the new block must be given a `key` value, the same as the originally deleted smallest key, rather than the new larger one. Otherwise, *all index blocks* for which the original key is the smallest live key in that block would need to be updated. By retaining the original key value in the index record, we avoid the space and time overhead of updating all such index levels. Note that if all levels were not updated, then insertion of keys with values between the deleted value and new smallest value would not proceed correctly. Insisting that the smallest key value in an index record is *not changed* by insertion of new index records, provides a simple method that permits correct operation without the overhead.

Overflow and underflow for index blocks are defined as for data blocks, and are handled in the same way. The root of the current database (current root) is tracked separately. When a version operation causes the current root to die, then if one new block is created, this block becomes the current root. If two new blocks are created, then the current root is the parent of the new nodes. If due to block consolidations the current root is left with only one live child, then the child node becomes the current root.

Note that although the MVAS structure is a directed acyclic multigraph, there is always a unique search path from the root for any key K at any time i. Hence, we use the tree notation such as parent, sibling, etc., in the descriptions. Note that other MVAS designs like the WOBT [5], the TSBT [13], and the design in [2], all result in a DAG, rather than the multigraph of our design. In other words, creation of a new index entry is always accompanied by the creation of a new block. In our design, a new index entry can point to the same physical block with possibly modified range. This reuse of blocks provides for more efficient storage usage in our design.

We end this section by stating formally a set of invariants maintained by the nodes (both data and index) of the MVAS. The parameters T_H and T_L are chosen to satisfy the following invariants for any sequence of inserts, updates, and deletes. In addition, in `Case 3` of `delete` we need to ensure that there is enough space in `B` to copy in the T_L live entries of block `A`. By choosing $T_H = 4B/5$ and $T_L = B/5 - 1$ this condition, and those implied by the invariants below, are satisfied. We assume that the *total* number of *live* records in the MVAS will *never* fall below $2T_L$. Note that otherwise, there is only one live block in the MVAS. Rather than treat this situation (which will rarely, if ever, arise in practice) as a special case, we assume the existence of at least $2T_L$ live records in the database, if necessary, by inserting $2T_L$ dummy records that can never be deleted at the start.

`Invariants`:

1) When a block is created it has at least $2T_L$ live records, and no more than T_H live records. This can be seen as follows. When two new blocks are created by `Case 1` of `insert`, the number of records in each block is at most $\lceil (B+1)/2 \rceil$, which is less than T_H for all $B \geq 5$. When two new blocks are created by `Case 3A` of `insert`, the number of records in each block is less than $(2T_L + 3T_L)/2 < B/2 < T_H$, for all $B \geq 5$. When one new block is created by an insert, it has at most T_H records in `Case 2` and `Case 3A`. In `Case 3B`, the new block is created with exactly $2T_L$ records. When a new block is created by a `delete`, it will have exactly $2T_L$ records in `Case 1`, and $L_B + T_L$ records in `Case 2`. Since $T_L < L_B < 3T_L$, this is at least $2T_L$ and no more than $4T_L < T_H$.
2) Every live block has at least $T_L + 1$ live records. This follows because a block is created with at least $2T_L$ live records. When the number of live records falls to T_L, the block dies.
3) When a block underflows, at least T_L version operations must have been done on it (including the delete triggering the underflow). (See Lemma 5.2 in Section 5.) When a block underflows, at most one new block is created.
4) When a block overflows, at least $B - T_H + 1 \geq T_L$ operations must be done on it (including the operation triggering the overflow). (See Lemma 5.4 in Section 5.) When a block overflows, at most two new blocks are created.

3.2 Time Complexity of Version Operations

In this section, we analyze the number of block accesses required for the version operations. The worst-case analysis of space is in Section 5. The $(i + 1)th$ version operation (insert, update, or delete) performs a search in version i and then modifies at least one data block of $\mathcal{T}$, say A. If A overflows or underflows, then up to two other data blocks in $\mathcal{T}$ have to be created or modified. On an overflow or an underflow, up to two new entries will be sent up to the parent of A, say PA. This may trigger an overflow or underflow of PA, and again up to two other blocks may have to be created

or modified at this level. In the worst case, these changes can propagate up to the root of $\mathcal{T}$. At each index level, up to three blocks may need to be modified or created.

Note that overflows or underflows at the data level can occur only after at least T_L version operations. Since each underflow or overflow can send at most two index records up to the next level; therefore, underflows or overflows can occur at index level l after at least $(T_L)^l / 2^{l-1}$ version operations (here data blocks are at level 1). Therefore, the amortized cost for structural changes on $\mathcal{T}$ for the $(i + 1)th$ version operation is at most

$$3 \sum_{i=1}^{\log_w N+1} 1 / 2(2 / T_L)^i,$$

where N is the total number of version operations so far. Since B is larger than 20, $2 / T_L < 1$, and hence this is at most a constant $c < 3/2$.

4 Query Operations

In this section, we describe the procedures for the query operations and analyze their time complexity.

To `search` for a key K that is valid at time t, we begin at the root block. Ignore the records with `start` timestamps larger than t and the records with `end` timestamps smaller than t. From the remaining records choose the one with the largest key value less than or equal to K; if there are several such records, choose the one with the largest `start` timestamp. Follow the pointer to the next level and then apply the same procedure at this level. Note that this process leads to a unique path down the MVAS to a data block. The record will be found in the data block if it exists; otherwise, there was no record with key K valid at time t. The number of block accesses required is $\lceil \log_w N \rceil$, where N is the total number of version operations until the time of this search operation. Note that $w \geq B/5$.

To `search` for a key range $[K_1, \ldots, K_2]$ that is valid at time t, we begin at the root block as before. For every key K in the range, we follow the same procedure as described above for a single key. This will result in a number of pointers that need to be followed to the next level of the index. Follow this procedure at every level of the index. For efficiency, duplicate elimination may be employed to prevent redundant accesses to the same block (these occur because of overlap between index blocks). By Invariant 2 in Section 3.1, the number of block accesses required is $\lceil \log_w N \rceil + O(R/B)$, where R is the number of records in the output of this query. This is because the keys that are valid at time t are clustered by key range into blocks, each block having a minimum occupancy (at least $B/5$ for our method) of keys valid at time t.

The snapshot query is answered optimally since it is the special case of the key range query, where the key range is the entire range.

The time-range query between $[t_1, \ldots, t_2]$ retrieves all records that were valid in the time range t_1 through t_2. To do this efficiently, data blocks at the leaf level of the MVAS are linked in time-sequence order, i.e., in the order in which they were created.[1] Logically, the time-range query can be considered to be composed of two subqueries: retrieval of all records valid at time t_1 (a snapshot query at t_1) and retrieval of all records created between times t_1 and t_2. The first subquery has been discussed above. For the second, notice that all records created between t_1 and t_2 will either have been placed in a block that is live at t_1 (and hence retrieved by the first subquery), or in a block created between t_1 and t_2. These latter blocks are linked together in the time order of their creation. Hence, after locating the first block created after t_1, we follow the link pointers until we encounter a block with start time greater than t_2. The first block created after t_1 can be found by examining the blocks retrieved by the snapshot subquery; the block with the smallest `end` timestamp will be the first block to die after t_1, and the block(s) created on its death, will be the first block(s) created after t_1. The number of block accesses required is $\lceil \log_w N \rceil + O(R/B)$, where R is the number of records in the output of this query.

For the key-history search query, we need to retrieve all records with key K that were valid between times t_1 and t_2. The simplest method (in the absence of deletes) is to maintain backward pointers in every data block to the block whose split (overflow or underflow) created it [13]. In this scheme, first locate the record with key K that is valid at t_2; then follow the backward pointers until a record with key K and `end` time less than t_1 is found. If key-history queries are relatively infrequent, then the simplicity of this scheme makes it attractive to implement. However, worst-case performance can be poor in this scheme, since each qualifying record could require a separate block access, resulting in a complexity of $\lceil \log_w N \rceil + O(R)$ block accesses rather than the optimal $\lceil \log_w N \rceil + O(R/B)$, if all qualifying records were clustered together. An additional problem with this scheme arises when deletes are allowed; the record with key K may have been deleted before t_2, but could have several versions between t_1 and t_2. Some other mechanism is needed to access the last version of K before its deletion. In Section 7, we describe an optimal solution for the key history query based on access lists integrated into the MVAS. However, since the trade-off is slightly increased times for version operations (although still $O(\log_w N_t)$), this proposal should be evaluated within the context of a given application, based on the frequency of the key-history query.

5 Analysis of Space

In this section, we prove an upper bound on the worst-case space requirements of the MVAS for any sequence of version operations. We perform an amortized analysis of the space. We first show that if the version number is N, the number of data blocks is no more than $5N/(B-5) + 1$.

The idea of the proof is as follows. Every time a version operation is performed on a block we charge a space of $1/T_L$ blocks for that operation. This accumulated space is held in reserve until the block dies, and we need to create a new block or pair of blocks. We will show that we will always have accumulated enough storage to pay for the

1. If two blocks are created at the same time, the tie may be broken arbitrarily.

blocks we need to create. This will show that for N version operations no more than N/T_L blocks are needed.

A new block is created only due to either the overflow or the underflow of another block. We model a block as a node of a rooted, directed binary tree. The root of the tree is the initial block. Every time an overflowing block is copied to two new blocks, we add two child nodes to it. Each node of such a pair is referred to as a *two-node*. Every time an overflowing or underflowing block is copied to one new block, add one node as the child; such a singleton child is called a *one-node*. The number of nodes in this tree is the number of data blocks. The sum of the number of version operations done to each block is the current version number.

Note that in certain cases like in `Case 2` of `delete` and `Case 3A` of `insert` two blocks die (the overflowing or underflowing block and a sibling block which is killed) during the creation of a new block or pair of blocks. In these cases, only the overflowing or underflowing node becomes the parent of the new node(s) created in the tree. The node corresponding to the killed sibling block is not affected; that node will permanently remain a leaf in the tree as it is now dead, and cannot overflow or underflow in future. Similarly, in cases such as `Case 1` of `delete` and `Case 3B` of `insert` one blocks dies, and the newly created block becomes the child node of the node which dies. The sibling block from which some records are copied is not affected. Finally, note that when records are simply copied into a sibling block, as in `Case 3` of `delete`, we do not add any children to the tree.

We will assume that the *total* number of *live* records in the MVAS will *never* fall below $2T_L$, so that the Invariants of Section 3.1 are satisfied. In particular, for purposes of simplifying the proof, it will be assumed that the first data block is created with $2T_L$ live records, which are never deleted. This allows us to handle the first node like every other node; otherwise, this node would need to be treated separately, causing an unwarranted complexity in the proof. Recall that the thresholds are $T_L = B/5 - 1$ and $T_H = 4B/5$. We begin with the main theorem that provides the structure of the proof. Individual results used in the theorem are then proved in subsequent lemmas.

THEOREM 5.1. *The number of blocks required by data nodes of $\mathcal{T}$ in the* worst case *is less than* $5N/(B-5) + 1$, *after any sequence of N version operations.*

PROOF. Every version operation brings in a charge of $1/T_L$ blocks. We show that between the time that a node is created and the time it dies, sufficient charge will have been accumulated to pay for the new block(s) that need to be created. The first node is created without any version operations having been performed; from the statement of the theorem, when $N = 0$ there is one data node. By our assumption, this node like all nodes that are subsequently created will satisfy the Invariants of Section 3.1.

Let α be a node that dies. If α *dies* on a `delete`, at most one new block is created. From Lemma 5.2, α can always pay for its child.

If *dies* on an `insert`, either one or two new blocks may be created. In the first case, Corollary 5.5 shows that α can pay for the single block it creates. If α creates two blocks, we consider several cases individually. If α was *created* by a `delete`, Lemmas 5.7 and 5.8 show that there is sufficient accumulated charge to pay for two new blocks.

If α was *created* by an `insert`, we consider the two cases when α is a two-node or a one-node separately. If α is a two-node, Lemma 5.6 shows that it can pay for the two new blocks. Finally, Lemma 5.9 shows that if α is a one-node, there is sufficient accumulated charge to pay for the two new blocks. □

LEMMA 5.2. *Let v be a node that dies due to a* `delete`. *When v dies it must have accumulated a charge of at least one block.*

PROOF. Let v be created with N records; note $N \geq 2T_L$. Since v dies on a `delete`, it must have only T_L live entries when it dies. Hence, at least $(N - T_L)$ records in v must have died since its creation. These records must have died because of either (i) a delete operation on a record in v, or (ii) because some of v's records were copied to another node while leaving v alive (i.e., the other node performs `Case 1` of `delete` or `Case 3B` of `insert`). Now, (ii) can occur only if v has at least $3T_L$ live records, and at most T_L records are copied at any time. Consequently, at least $2T_L - T_L = T_L$ records in v must have died due to Case (i); i.e., there must be at least T_L deletes (version operations) done on v before it dies. Since, each version operation brings in a charge of $1/T_L$ block, v will accumulate a total charge of at least one block before it dies. □

LEMMA 5.3. *Let v be a node that dies by* `Case 3` *of procedure* `delete`. *Then, each record copied to the sibling block will bring in a charge of at least* $1/T_L$ *blocks.*

PROOF. By definition, v will dump its T_L live records to a sibling block when it dies. From Lemma 5.2, v will have accumulated a total charge of at least one block before it dies. Since, it does not need to buy any new blocks, this entire accumulated charge can be transferred to the sibling block. Thus, the sibling receives T_L records, and a transferred charge of at least one block. Hence, each record dumped brings in a charge of at least $1/T_L$ blocks. □

Note that the *transfer* of charge to another block when records are copied, as implied by the above Lemma, occurs only in certain situations. In particular, the block which transfers the charge always dies. This simplifies the bookkeeping required in the proof, since a live block from which records have been copied (as in `Case 1` of `delete` for example) does not transfer any of its charge. Secondly, the charge is transferred only to a block which is already alive (as in `Case 3` of `delete`). If a new block is being created (as in `Case 3B` of `insert`), then we never transfer accumulated charge of the copied records to the new block. Note `Case 3` of `delete` is the *only* time when we transfer charge to another block by copying records to it.

LEMMA 5.4. *Let v be a node that dies due to an* `insert`. *If v was created with N records, then on its death it must have accumulated a charge of at least* $5 - (N-6)/T_L$ *blocks.*

PROOF. Since v dies on an `insert` operation, at least $(B - N + 1)$ records must have been added to v before its death. The number of records in a node v increases only due to either (i) an insert or update operation into v, or (ii) because records are dumped to it by a sibling block (i.e., the sibling block does `Case 3` of `delete`). From Lemma 5.3, every record dumped into v by (ii) brings in a charge of at least $1/T_L$ block. Also, each version operation brings in a charge of $1/T_L$ block. Hence, v will accumulate a total charge of at least $(B - N + 1)/T_L = 5 - (N - 6)/T_L$ blocks before it dies. □

COROLLARY 5.5. *A node that dies on an* `insert` *must have accumulated a charge of at least one block on its death.*

PROOF. Every node is created with no more than $T_H = 4T_L + 4$ records (Invariant 1, Section 3.1). Hence, in Lemma 5.4, since $N \leq 4T_L + 4$, the charge accumulated by the node is at least $5 - (4T_L + 4 - 6)/T_L = 1 + 2/T_L > 1$ block. □

LEMMA 5.6. *Let v be a two-node that is created by an* `insert` *and dies on an* `insert`. *When v dies it must have accumulated a charge of at least two blocks.*

PROOF. Since v is a two-node, there are two cases to consider: either v was created by `Case 1` or by `Case 3` of procedure `insert`. Let N be the number of records with which v was created. If v is created by `Case 1`, then $N \leq (B/2 + 1)$. From Lemma 5.4, v will accumulate a total charge of at least $5 - (B/2 - 5)/T_L = 5/2 + 5/2T_L > 2$ blocks by the time it dies.

If v is created by `Case 3A` of `insert`, we argue as follows. The total number of records contributed by the parent of v is less than $2T_L$. Let the number of records contributed by the sibling block involved in the merge (call it μ) be x. These $(2T_L + x)$ records are split evenly into two blocks. Hence, $N < (2T_L + x)/2 + 1 = (T_L + x/2 + 1)$ records. From Lemma 5.4, when v dies it will have accumulated a total charge of at least $5 - (T_L + x/2 - 5)/T_L = 4 - (x - 10)/2T_L$ blocks.

If $x \leq T_H = 4T_L + 4$, then the accumulated charge is at least $4 - (4T_L - 6)/2T_L = 2 + 3/T_L$ blocks. If $x > T_H$, let $x = T_H + y$, $y > 0$. Then, the charge accumulated by v will be $2 + 3/T_L - y/(2T_L)$ blocks. Since no node is created with more than T_H total entries, at least y insert or update operations (or record dumps) must have been done to μ before it is merged. Hence, the charge equivalent to $y/2$ version operations can be *inherited* by each of the two nodes created by the merge. Therefore, when v is created it inherits a charge of at least $y/(2T_L)$ from the sibling block that is killed. The total charge brought in by version operations to v plus the inherited charge is therefore $2 + 3/T_L - y/(2T_L) + y/(2T_L) > 2$ blocks. □

LEMMA 5.7. *Let v be a one-node that is created by* `Case 1` *of* `delete` *and dies due an* `insert`. *When v dies, it must have accumulated a charge of at least two blocks.*

PROOF. Since v is created by `Case 1` of a `delete` operation, v is created with exactly $2T_L$ records. From Lemma 5.4, since $N = 2T_L$, when v dies, it must have accumulated a total charge of $5 - (2T_L - 6)/T_L = 3 + 6/T_L > 2$ blocks. □

LEMMA 5.8. *Let v be a one-node that is created by* `Case 2` *of procedure* `delete`, *and dies due to an* `insert`. *When v dies, it must have accumulated a charge of at least two blocks.*

PROOF. From Corollary 5.5, v will accumulate a charge of one block between its creation and death. We show below that v will inherit one block of charge from the sibling block that was killed when v was created, for a total of two blocks. Let β be the parent of v, and let α be the sibling block that was killed. We show that α can transfer one block of charge to v. The proof is by induction on the number of sibling blocks that are killed on deletes (i.e., by `Case 2`).

`Basis`: Let α be the *first* sibling block to be killed by `Case 2` of a delete operation. By the definition of `Case 2`, the total number of records in α exceeds $(T_H + 1)$. There are two cases to consider, depending on whether α is a two-node or a one-node.

Case 1: α is a two-node. Since α is the first sibling block to be killed by `Case 2` of `delete`, α itself must have been created by an `insert` or by `Case 1` of `delete`. Therefore, when α is created it has no more than $B/2 + 1$ records (Invariant 1 in Section 3.1). Since, when it is killed it has more than $T_H + 1$ records, at least $T_H + 1 - (B/2 + 1) + 1 = (T_H - B/2 + 1)$ records must be added to it. These records will be made up of either version operations (inserts and updates) or result from some other node *dumping* its records to α. In either case, each record brings in a charge of at least $1/T_L$ blocks to α. Hence, when α dies, it must have accumulated a total charge of

$$\frac{T_H - B/2 + 1}{T_L} = \frac{4T_L - (5T_L + 5)/2 + 5}{T_L} = \frac{3T_L/2 + 5/2}{T_L} > 1 \text{ block}.$$

Case 2: α is a one-node. In this case, consider the path of one-nodes from α up the tree to the first two-node, δ, in the path.[2] If δ is created and dies due to an `insert`, then by Lemma 5.6 (and the fact that δ died into a one-node), it has an *excess capacity* of one block, which v can inherit from α. Since the death of α terminates the chain of nodes starting from δ, there will never be any further requests for charge from any nodes on this path.

If δ is created on an `insert` and dies on a `delete`, then either all nodes in the path from δ to α are created by deletes, or at least one node in this path is created on an insert. In the first case, note that all of these nodes (including α) must be created by `Case 1` of `delete` (v is the first node created by `Case 2` of `delete`), and hence are created with $2T_L$ records. When α is killed by v due to Case 2 of `delete`, it

2. If a node corresponds to a sibling block, then it must have an ancestor in the tree that is a two-node.

must have more than $T_H + 1$ total records. Hence, at least $T_H + 1 - 2T_L + 1$ records must be added to it, incurring a charge of

$$\frac{T_H - 2T_L + 2}{T_L} = \frac{2T_L + 6}{T_L} > 2 \text{ blocks.}$$

In the second case, there will be a node that was created on a `delete` and that dies on an `insert`, in the path from δ to α. Consider the first such node in the path. This node must also have been created by `Case 1` of `delete`. By Lemma 5.7, this node will have an *excess capacity* of one block (since it died into a one-node), which can be inherited from α by v.

In either case, one block of accumulated charge can be inherited by v when α is killed.

`Inductive Step`: Assume the lemma is true for first n sibling blocks that are killed on a delete. We will show it is true for the $(n + 1)$*th* such sibling block, say α. Once again, let δ denote the first two-node encountered in the path from α up the tree.

If α is a two-node or if α is a one-node such that none of the nodes in the chain from δ to α was created by `Case 2` of `delete`, then the arguments of the base case can be used to show that the lemma holds for α.

If not, find the lowest node in this path (closest to α) between δ and α that was created by `Case 2` of the `delete` operation. From the induction hypothesis, this node has an excess charge of one block (inherited by it from the sibling block that was killed when it was created). This charge would not have been used for anything until now (by Lemma 5.2 and Corollary 5.5, every node along a chain of one-nodes generates enough charge to always pay for its one child, without using any inherited charges) and can be inherited from α by v. □

LEMMA 5.9. *Let v be a one-node created by an* `insert` *and dies due to an* `insert`. *When v dies, it must have accumulated a charge of at least two blocks.*

PROOF. Let v be the one-node created by an `insert` that must create two new blocks on its death. By Corollary 5.5, v can pay for one child using the charge brought in by version operations done (or records copied) to it. For the second child, v will utilize unused charge accumulated by one of its ancestor nodes. The proof is similar in spirit to that of Lemma 5.8.

Consider the path of one-nodes from v up the tree until the first two-node, δ. If δ was created and died due to an `insert`, it has an excess unused charge of one block (see Lemma 5.6), which can be used by v. (The only time a two-node will not be found is in the degenerate case where all splits from the start have created only one-nodes; in this case, the excess reserve of the root node can be used; note that this can happen only once, since this operation is creating a two-node.) If not, then δ must have died due to a `delete`, and there must be at least one node on the path between δ and v that was created on a `delete` and died on an `insert`. By Lemmas 5.7 and 5.8, such a node has a reserve charge of one block, sufficient to cover v's deficiency. □

THEOREM 5.10. *The number of blocks required by the index nodes of $\mathcal{T}$ in the* worst case *is less than* $50\,N/(B-5)(B-15)$, *after any sequence of N version operations.*

PROOF. The space analysis for the index levels is similar. Let $B' = B - 5$. Consider the first level of index blocks, immediately above the data blocks. For every data block created, one new index record is created (call it a type 1 record). Furthermore, an index record is created to point to an already existing block (call it a type 2 record), whenever the smallest key of a data block is copied in `Case 3B` of `insert`, and `Cases 1` and `3` of `delete`. The number of type 1 index records created is $5N/B'$. To bound the number of type 2 index records, one new record can be sent up on every block underflow, and on every block overflow that satisfies the conditions of `Case 3B` of `insert`. Now, assume that N_1 of the N version operations were to blocks that underflowed, and the remaining $N - N_1$ to blocks that overflowed by `Case 3B` of `insert`. Since underflows can occur only after T_L delete operations, at most $N_1 / (T_L)$ type 2 index records can be created by underflows. For a type 2 index record to be created by `Case 3B` of `insert`, there must have been at least $B - T_H + 1$ inserts, i.e., again at least T_L version operations. Hence, at most $(N - N_1)/T_L$ such type 2 records are created. The total number of type 2 records created is $N_1/T_L + (N - N_1)/T_L = N/T_L$. Hence, the total number of type 1 and type 2 index records created at the first index level is at most $5N/B' + 5N/B' = 10N/B'$. The analysis for data blocks above then shows that the index blocks at this level will be at most $5/B'(10N/B')$.

A similar situation holds at every index level. At level $l \geq 1$, the number of type 1 index records equals the number of blocks at the next lower index level, $l - 1$. The number of type 2 index records will, by an argument similar to that above, be no more than this quantity. Thus, the number of blocks at index level l equals $5/B' * 2 *$ [blocks required at level $l - 1$], which is [blocks required at level $l - 1$] $*\ (10/B')$ (level 0 corresponds to data blocks). Therefore, the total index space in blocks is at most

$$(5N / B') \sum_{i=1}^{\log_w N} (10 / B')^i = 5N / B' * (10 / B') *$$

$$[(1 - (10 / B')^{\log_w N}) / (1 - (10 / B'))].$$

Assuming $B' > 10$, we get a bound of $50N/B'(B' - 10)$. □

6 SIMULATION RESULTS

A simulation program was developed to compare the storage utilization in the average case against the worst-case analysis presented in Section 5 and to determine the performance of typical queries. As is usual in such studies, we make the assumption that the keys are uniformly distributed, and that on an update or a delete operation each record with a unique key is equally likely to be chosen [14].

The following parameters were used in the simulation: p, q, and r, which are the probabilities of an insert, an update,

and a delete operation, respectively, and the block size *B*. The block size was fixed at 25 records for the results reported here. We varied the block size and noticed that in general smaller blocks performed better than larger ones. Beyond a block size of 25, the performance did not change significantly. Hence, 25 was chosen since we were able to obtain stable results with reasonable running times for the simulations. Insert, delete and update probabilities were varied between 0 and 1 subject to the constraints $p + q + r = 1$ and $p \geq r$. Note that p must be greater than r for the system to reach a steady state; otherwise, the number of live blocks will ultimately fall to one. The database was initialized with 1,000 records; this was followed by 50,000 version operations, i.e., inserts, deletes, or updates with probabilities p, q, and r, respectively, as mentioned above. Note that the initial records will occupy about 55 blocks.

We report the utilizations of the data blocks of $\mathcal{T}$, which require the most storage in the MVAS. In particular, we report the Single Version Current Utilization (SVU) and the Total Version Utilization (TVU) defined below (SVU measures the average number of live records in a data block, i.e., the occupancy of the current database; TVU measures the average space cost incurred by each version operation):

- SVU = (Number of Inserts – Number of Deletes)/(B ∗ Number of live blocks)
- TVU = Version Operation Number/(B ∗ Total number of blocks)

Fig. 5 is a plot of the Single Version Utilization for different values of p, q, and r. For $r = 0$ (i.e., no deletes), the SVU is the same as that of a Write Once B-Tree [5]. The rising trend of SVU with decreasing numbers of deletes is to be expected, since a delete reduces the number of live records in a data node, without freeing up any storage. Updates do not change the number of live records but do increase the number of live blocks, thereby explaining the rising trend of SVU with decreasing numbers of updates. In fact, using the analysis of [14] for 100 percent updates the utilization should be $4/5 * ln2 = 0.56$, which agrees with the experimental value. At 100 percent inserts, the *live nodes* form a B+-tree whose average utilization is $ln2 = 0.693$, which matches our experimental result.

Fig. 6 shows TVU for different ranges of p, q, and r. Note that for $r = 0$, the utilization is roughly 0.4 and increases to 0.75 as the percentage of deletes increases. This is to be compared with the worst-case utilization of 0.2 derived in Section 5 showing that the worst-case bound is very pessimistic and supports the practicality of this method.

Direct comparison with related multiversion access structures proposed in [2], [5], [14] for the average case is difficult, since the methods in [5], [14] do not handle deletes and no experimental results are presented in [2].

For query performance, we make the same uniformity assumptions that we made when measuring storage utilization, and the same initialization procedure. We evaluated the query performance at two extreme situations: (i) insert probability 0.1 and update probability 0.8 (high updates), and (ii) insert probability 0.8 and update probability 0.1 (high inserts). From Fig. 6, the first situation has better storage utilization.

The experimental procedure was as follows. For the snapshot query a time t is chosen uniformly between 1 and the version number and a snapshot at time t is found. This process is repeated 100 times for each version number, and the averages are reported. For the time-range query, the lower time t_1 is chosen as in the snapshot query and the higher time $t_2 = t_1 + \text{range}$. Note that t_2 can exceed the current version number but it does so rarely (especially for the lower ranges).[3] As a consequence, the true time range of the query is smaller than the nominal value $(t_2 - t_1)$, and the number of blocks actually accessed will be less than that suggested by the nominal time range. This process is also repeated 100 times and the averages are reported. For the key-range query, the lower key is chosen uniformly between 1 and 10^6 (this is the entire range of keys), and a time is chosen uniformly between 1 and the version number. The higher key equals the lower key plus range. Again the higher key can exceed the highest key in the database and does so fairly often at the higher ranges. As in the time-range query, the actual key range (and number of accessed blocks) will be less than the nominal key range specified in the query. The keys are chosen 1,000 times for each version number and, for each choice of keys, the time is chosen 10 times, and the averages are reported.

Figs. 7 to 11 show the simulation results of different queries. For the key-range query, the number of blocks accessed depends on the range, reaching a maximum of about 50 blocks for the low insert situation when the query range encompasses the entire key range. For the high insert situation (Fig. 8), the number of blocks increases monotonically with the version number as expected, and the slope increases with increasing key range. For the time-range query when the size of the database is steady (i.e., probability of insertion equals probability of deletion), the query time is relatively insensitive to the version number as expected. Note that for the time range of 10,000 at 10,000 version operations, the true range will be smaller than the nominal range almost all of the time. This explains the low number of block access for this time range in Fig. 9. This effect decreases with increasing version number.

The apparent large difference between the performance of the key range and time range queries for a given number of version operations is a direct consequence of the fact that the key space (between 1 and 10^6) is much larger than the maximum time range (1 ... 50,000). Thus, a time range query of a given interval (say 5,000) will translate into retrieving far more records than a key range query with the same interval. When ranges are adjusted so that both types of queries have the same output size, then both queries are comparable. Note that the database is initialized with 1,000 records before version simulation begins. Hence, at least $1{,}000/(25 * 0.5) = 80$ block accesses are required just for these records alone. Moreover, the implementation of the time range query in our simulation studies can be optimized further. We chose a simple implementation in which the time range query for the interval $[t_1, t_2]$ is implemented by (i) a snapshot at t_1, and (ii) then a search for the lowest key at t_1 is done followed by

3. We allow this to happen because, in practice, one does not generally know how many operations have been done on the database or even the range of keys that are present.

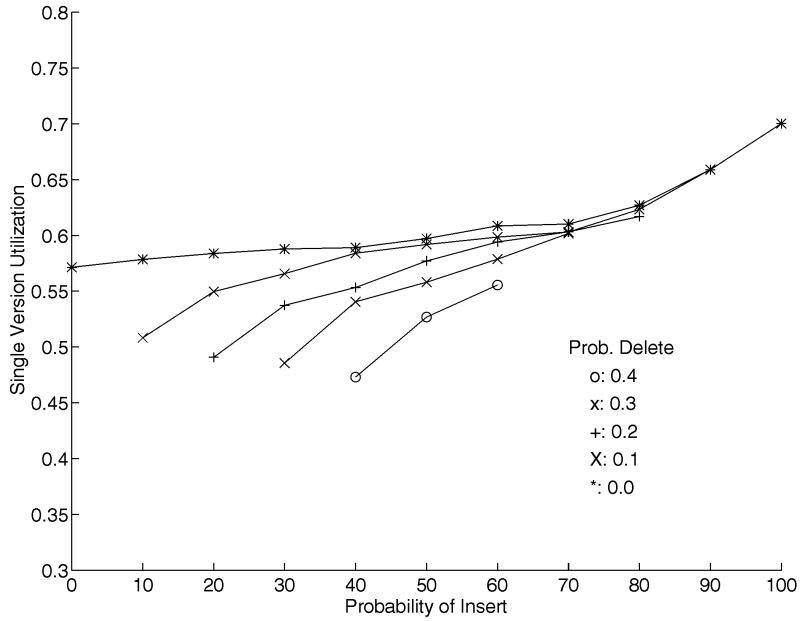


Fig. 5. Single version current utilization.

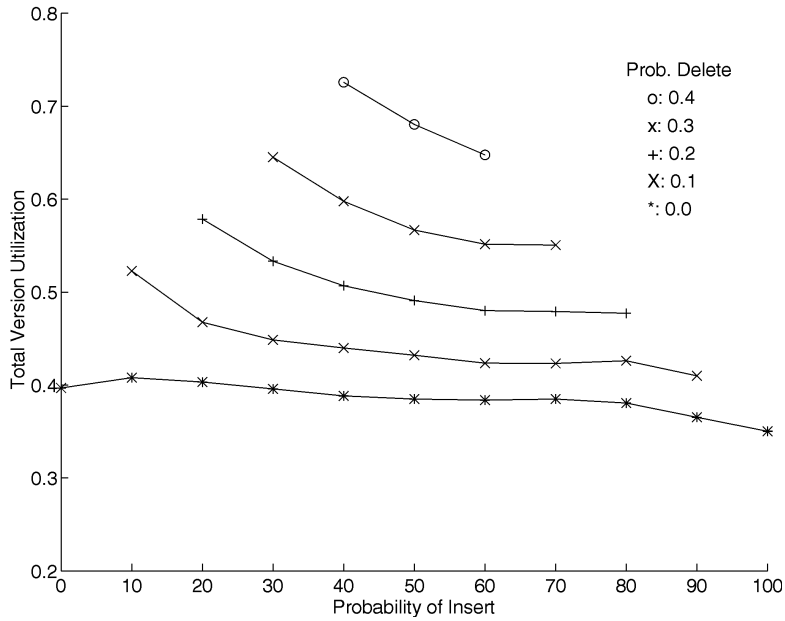


Fig. 6. Total version utilization.

accessing the chain of data buckets until a bucket with timestamp higher than t_2 is reached. There is a significant overlap between the bucket accesses of (i) and (ii), which can be eliminated by accessing the chain of data buckets starting from the highest timestamp data bucket accessed by the snapshot query at t_1.

Finally, the performance of the snapshot query is relatively stable for a steady-state database, and increases with time for a growing database (insert probability greater than delete probability), as shown in Fig. 11. Theoretically, the block accesses for the snapshot query (which is just the key range query with the entire range of keys at a certain time) and the key range query for the largest range should be almost the same. The reason for the difference between Figs. 8 and 11 is that the true key range for a key-range query is less than the nominal range, due to the random choice of the initial key value—as described earlier. Hence, the accesses are higher for the snapshot query and lower for the key-range query. The snapshot query performance is as expected in the case of a growing database, since the number of live records at time t increases linearly with time.

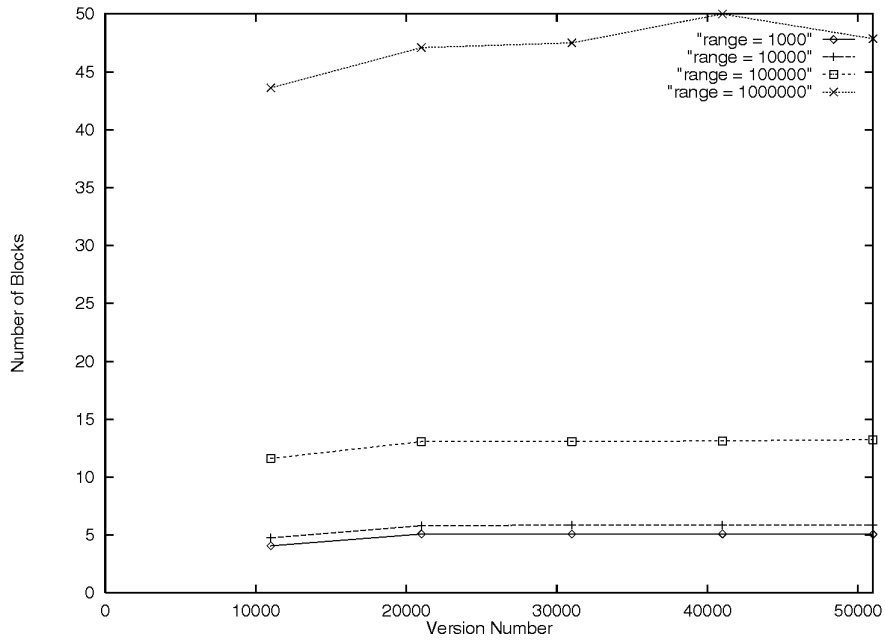


Fig. 7. Results for the Key Range Query (insert prob. 0.1, update 0.8).

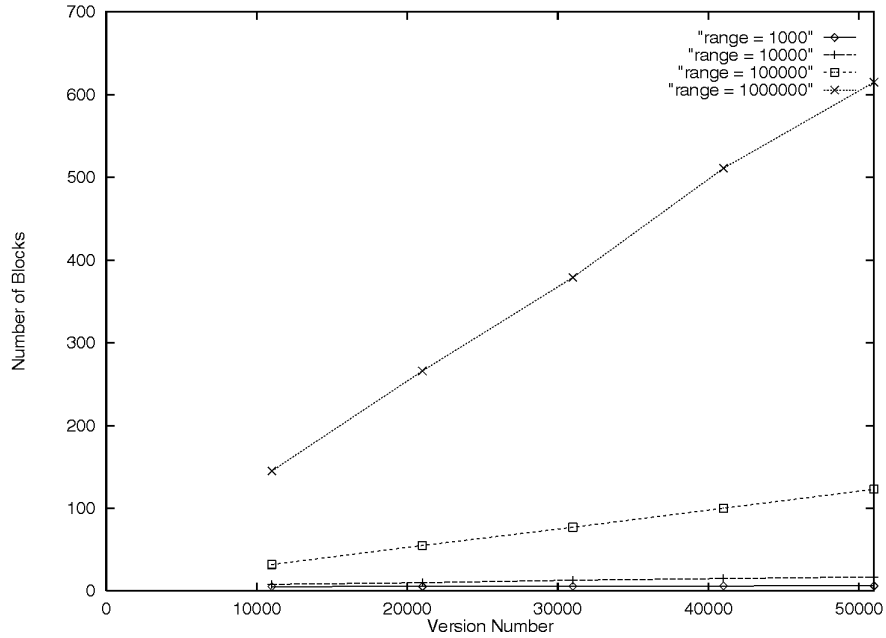


Fig. 8. Results for the Key Range Query (insert prob. 0.8, update 0.1).

7 Integrating Access Lists into MVAS

In this section, we describe the augmentation of the MVAS with access lists to improve the performance of key-history queries. The trade-off is a small increase in the amortized time for version operations necessary to maintain the associated structures, although it is still logarithmic in the size of the current database.

The access list (denoted by *C*-list) is made up of a collection of blocks that contain data records clustered by the key attribute. The data nodes of the MVAS and the blocks of *C*-list are linked together by two-way pointers as discussed below. The pointers from the MVAS are used as entry points into *C*-list; the backpointers from *C*-list are used to locate records in the data nodes of MVAS that must be updated when the address of a *C*-list block changes. Fig. 12 shows the organization of the data nodes of MVAS and *C*-list. In the example, there are three MVAS data nodes (numbered 1, 2, and 3) and two *C*-list blocks. Each data node and *C*-list block can hold five records. Each record in Fig. 12 is represented by three fields: a key, the start time, and a pointer. The record with key A and start time 2, for instance, is shown as (A, 2).

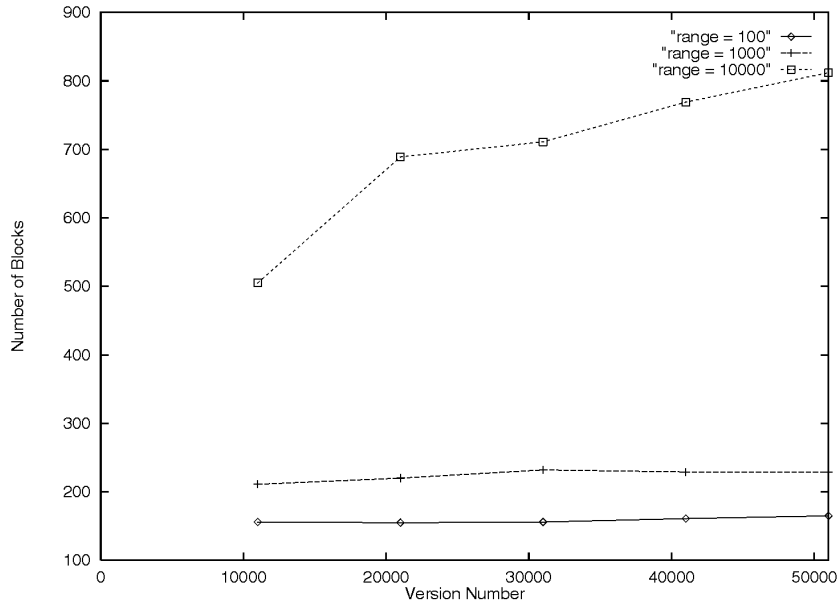


Fig. 9. Results for the Time Range Query (insert prob. 0.1, update 0.8).

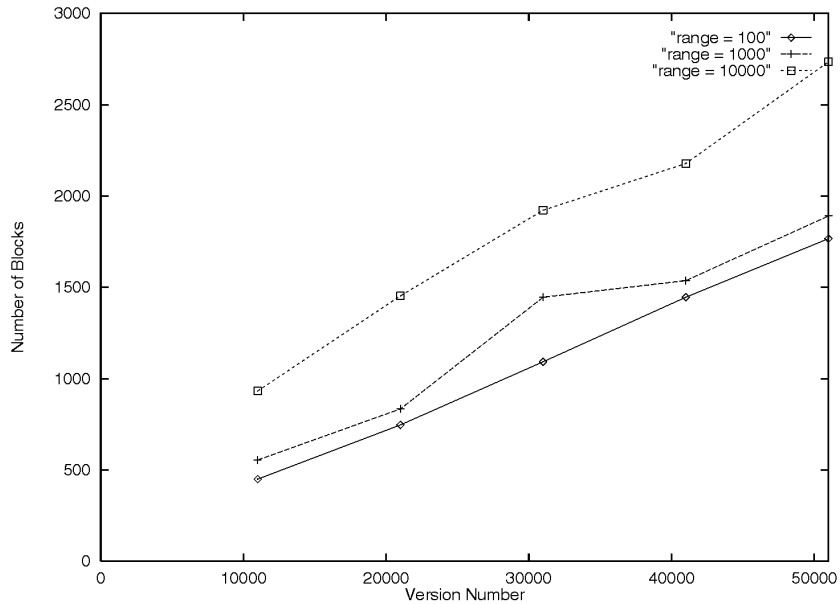


Fig. 10. Results for the Time Range Query (Insert prob. 0.8, update 0.1).

The blocks of the *C*-list are chained together in reverse order (i.e, later versions are toward the head of the list) as a singly linked list, so that the list is ordered by key and successive versions of a record with a given key are clustered together. In Fig. 12, for instance, the first block contains the various versions of the records with key *A* ((A, 2) and (A, 7)), and key B ((B, 5)). The second block contains the different versions of keys *C* and *D*. This is similar to access lists proposed in [1]. Two major differences are that here the access points into *C*-list are from the data nodes of MVAS, and secondly, the maintenance of the *C*-list is complicated, since nodes of the MVAS will split and merge as they overflow and underflow.

A record in a data node of MVAS points to either a block in *C*-list directly, or indirectly through another MVAS data node. For instance, record (A, 2) in node 1 points to the block in *C*-list that contains a copy of the record (A, 2). However, record (A, 2) in node 2 (which is a copy of the record in node 1) does not point directly to the *C*-list, but rather to node 1 of MVAS whose (A, 2) record, in turn, points to the *C*-list block containing (A, 2). This level of indirection means that if a record in *C*-list is moved to another block, only one entry in MVAS needs to be updated with the new location. To locate the entries in MVAS that need updating, each record in *C*-list has a backpointer to the data node in MVAS that

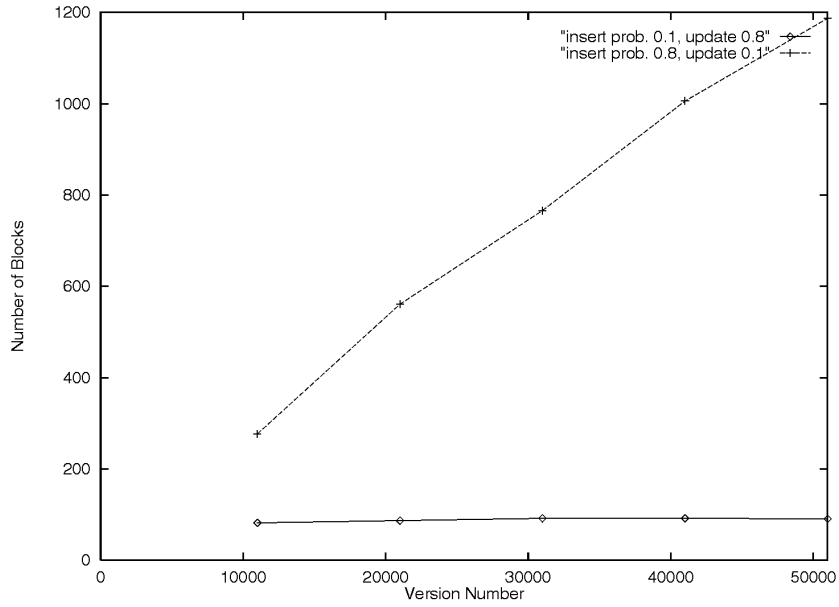


Fig. 11. Results for the Snapshot Query.

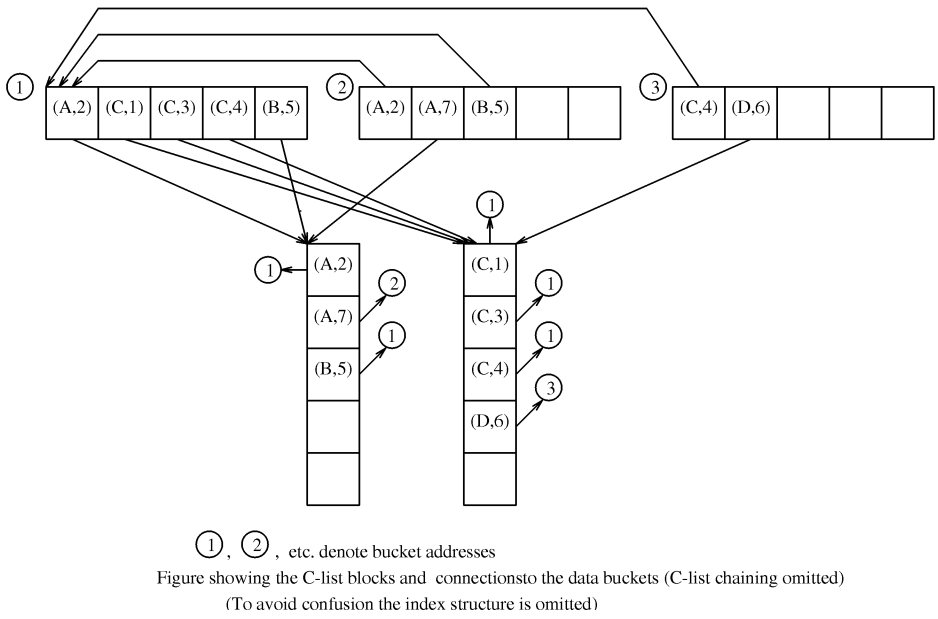


Fig. 12. The C-list (end times are not shown).

contains the record pointing to it. For instance, (A, 2) in *C*-list points to block 1 of MVAS, (A, 7) to block 2, and (B, 5) to block 1.

When a record with key *K* is created at time *i*, the tuple $\langle K, i \rangle$ must be inserted into the appropriate block in *C*-list, besides being inserted into the appropriate MVAS data node. A `cptr` field is introduced in the record that is inserted in MVAS and this field is updated with the *C*-list block address (a marker indicating that it points directly to the *C*-list is also made). Similarly, the entry in the *C*-list has its `cptr` set to point back to the MVAS node where $\langle K, i \rangle$ was inserted. As the MVAS nodes overflow and underflow, the record $\langle K, i \rangle$ may be copied several times. Rather than copying the `cptr` value that is stored in the first copy of $\langle K, i \rangle$, a pointer to the data block containing the first $\langle K, i \rangle$ is stored in the `cptr` field for the copies. When a block in the *C*-list becomes full, it is split in a manner similar to the split of data blocks in regular B+-trees; i.e., a new block is created and half the keys with the lower key values are moved to it (note that half the space in the old block is now available). The new block containing the lower subrange of keys is linked into the *C*-list immediately following the overflowing block. The images in MVAS of these records that are moved must have their `cptr` fields updated to point to

the new block; these records in MVAS can be identified using the backpointers stored with each key in *C*-list. Note that although this update using the backpointers may require up to $B/2$ node accesses in the worst case, this can happen only after every $B/2$ updates or inserts, for an amortized cost of one block access per update or insertion. Since the minimum occupancy of each block in *C*-list is at least $B/2$, so the *C*-list will require at most $2N/B$ blocks to store N versions.

With the use of the *C*-list the `key-history` search procedure is modified as follows. To `search` for all versions of a key K between times i and j, we first perform a key search for K at time j in MVAS. If K is found in the MVAS data block, we use the `cptr` field to access that record in *C*-list, either directly or indirectly, with an additional access of the MVAS data node that points to the *C*-list block. Scan records in *C*-list sequentially, in reverse time order from this point until the version of key K with end time less than i is found. The records obtained in this scan is the history of K between times i and j.

The procedure above is sufficient to find all the versions of a key K between times i and j in the absence of deletes. If K is not found in the data block of the MVAS at j, we can conclude that it was not created before j, and hence has no existence in the interval i to j. However, in the presence of deletes, the record may have been deleted before j, but may have had several versions between i and j.

To handle this situation a *sparse* index similar to a B+-tree index is built on top of the data blocks that make up the *C*-list. There is an index entry for every distinct key that is also the lowest key of a block of *C*-list. If a key has several versions then the index record (if any) for that key points to the latest version of that key. After a record has been deleted its last version can be found by searching this sparse B+-tree index. This index will be used for a key-history search in the interval $[i, j]$ *only* in the case that no record with that key is live at time j; i.e., no matching record is found in the MVAS data node. Otherwise, as noted earlier, the `cptr` field will be used to access the appropriate *C*-list block. The search of the index on *C*-list will result in either finding no record with that key in the *C*-list, or will lead to the last version of the key before it was deleted. If the search fails or if the last-version found has an end time less than i, then the query output is null; otherwise, the versions with end time less than i are output by scanning the *C*-list blocks.

We now derive the number of block accesses required for the key-history query. For a key which was not current at any time between i and j, the number of block accesses is bounded by $\lceil \log_w N \rceil + \lceil \log_{B/2} N \rceil$. The first term is for searching the MVAS at time j, and the second is for searching the sparse B+-tree index on *C*-list. Note that the number of distinct keys in *C*-list can be anywhere between $\Theta(1)$ and $\Theta(N)$, depending on the relative frequencies of inserts and updates. For a key which was current at time j, the number of accesses is bounded by $\lceil \log_w N \rceil + 2 + O(R/B)$, where R is the number of records in the output; and for a key, which was deleted at some time between i and j, the number of accesses is bounded by $\lceil \log_w N \rceil + \lceil \log_{B/2} N \rceil + O(R/B)$.

The index on the *C*-list needs to be carefully designed to keep the costs of version operations small. The index must allow access to all distinct keys created from the start up to the present time; in general, there may be $\Theta(N)$ such keys. If the index is traversed every time that a *C*-list block is accessed, then the cost for version operations will be $\Theta(\log N)$. For simultaneous optimality, the cost of these operations must also be kept proportional to $\log N_t$ (the number of live records at time t). This is achieved by using two-way pointers to link the parent and child levels of the B+-tree index. The index can be ascended from leaf to root by following these back pointers. Back pointers are useful here due to the differences in the insertion procedures into a usual B+-tree and into this index. In a usual B+-tree, the location of a new key is determined solely by following the pointers from the root to the leaf of the tree. In contrast, the location of a record that is being inserted into *C*-list (the leaf level of our index) is determined directly by following pointers from the data blocks of MVAS. There is no need to traverse the B+-tree to find the location for the new record in *C*-list.

We now describe how `insert`, `update`, and `delete` operations affect the *C*-list and its associated index. For all three operations, the appropriate data node in MVAS is first accessed, and the `cptr` field followed to the appropriate block of *C*-list. A `delete` merely changes the end time of the record in *C*-list to the present time. An `insert` and `update` require a new record to be inserted into the *C*-list. If there is space in the block for the new record, no further action is necessary. If not, the block overflows: A new block is created, half the records with lower key subrange are copied into it, the backpointer of the new block is set to point to the parent of the overflowing block, and the new block is linked into the *C*-list after the overflowing block. Next, the nodes of the index on the *C*-list are updated as follows. The parent node of the overflowing block is accessed using the backpointer. An index entry for the new block is added if necessary, and the index entry for the overflowing block is changed (if necessary) to reflect the (possibly) new smallest key in that block. Overflow of the index block is handled in a similar manner, using its backpointers to access its parent. When an index node overflows and splits, the backpointers in its children must also be updated.

Note that there must be at least $B/2$ `insert` or `update` operations for a block of *C*-list to overflow. An index block at level i, $i \geq 1$, overflows only after $(B/2)^{i+1}$ operations. Each overflow of a block of *C*-list causes two additional block accesses. An overflow of an index block causes at most $B/2 + 2$ additional block accesses (one for the new index block, one for the parent, and up to $B/2$ for the the children whose parent has changed). The amortized cost of these additional accesses is at most a constant, say $c' < 1$, as in the analysis in Section 3.2. Note that c' is smaller than the constant c of Section 3.2.

Finally, we determine the number of accesses required for version operations. The cost will consist of several components: to search the MVAS ($(\log_w N_t)$), for structural changes to the MVAS (a constant $c \leq 2$ amortized cost per version operation as shown in Section 3.2), to access the

C-list (2), for making structural changes to the *C*-list (a constant $c' \leq c$ amortized cost per version operation as argued above), and for updating the `cptr` fields in MVAS when a block in *C*-list overflows (amortized cost one per version operation).

An insert or update may trigger an overflow in the *C*-list. In this case, the total amortized cost is bounded by $\lceil log_w N_t \rceil + c + 2 + c' + 1 \leq 6 + \lceil log_w N_t \rceil$. In practice, the cost will be even smaller because every block accessed for updating `cptr` field of records is likely to contain many records that need to be updated. Finally, since deletes do not alter the B+-tree index, the amortized cost for deletes is bounded by $c + \lceil log_w N_t \rceil + 2 \leq 4 + \lceil \log_w N_t \rceil$.

8 Previous Work

Many previous proposals in recent years have considered the design of multiversion access structures [1], [2], [5], [6], [7], [10], [11], [12], [13], [14], [15], [24]. We review them briefly below. For a comprehensive comparison between the different methods proposed for access methods for time-evolving data, readers are referred to [18].

Ahn and Snodgrass [1] proposed the use of access lists that permit efficient retrieval of the history of any key. They also discuss various alternative clustering methods associated with this approach. The method provides good storage efficiency and good time complexity for key-history queries. Improved analysis of the time for AND queries on such systems by traversing multiple chains was provided by Manolopoulos [16]. However, snapshot queries that ask for the state of the database at some time in the past, and key range queries that ask for information about a range of keys at a particular time, could require a large number of block accesses.

The Write-Once B-Tree (WOBT) of Easton [5] has been the basis of several multiversion access structures [2], [13]. The main emphasis of the WOBT is to implement an access structure on write-once storage such as optical disks. The Time-Split B-Tree of Lomet and Salzberg [13], [14] is a clever adaptation of WOBT that uses an additional type of block split based on time, with a view to clustering records having the same version. This results in good space efficiency as well as good time efficiency for certain queries. The only version operations explicitly supported are `insert` and `update`, i.e., deletes are treated as a special case of update. This can result in the present-version records being fragmented over several blocks leading to degraded query performance for range queries. As noted by [2], even in the absence of delete operations, key-range search queries may have poor performance, since records close in key space at some version may not be clustered in blocks. It may require up to $\Theta(log_B N + R)$ block accesses to retrieve R records, instead of the optimal $\Theta(log_w N + R/B)$ achieved by our method and that of [2]. For key-history queries, [13] proposed using backpointers to chain together different versions of a key. This can result in the same performance difference ($\Theta(R)$ versus $\Theta(R/B)$) in comparison with our key-history query performance.

Recently, Becker et al. [2] proposed a data structure for multiversion data that can handle deletes as well. This was the first multiversion access structure that handled deletes in a quantifiable manner, and achieved simultaneously optimal performance in storage, time for version operations and times for key-range and time-range view queries. We significantly improve the worst-case storage bound for the data level nodes of about $10N/B$ reported in [2] to $5N/(B-5)$. This is achieved by a combination of using more storage-conservative overflow and underflow policies, and coming up with a tighter (and nontrivial) analysis. These policies attempt to reduce storage, when possible, in two ways: by partial copying of (copying only some) records from a block, and by using available space in an existing block rather than creating a new one. These new policies result in the MVAS having the structure of a directed, acyclic multigraph (parallel edges between nodes) rather than a DAG as in the designs in [2], [13]. Note that [2] did not consider the key-history query explicitly.

Our method is simultaneously optimal for the key-history query as well. The time for key-history search queries is $\Theta(\log_w N + R/B)$; in contrast without the *C*-list the time required would be $\Theta(\log_w N + R)$, using the backpointer method of [13]. Furthermore, the time for version operations required to maintain the additional structures is only $O(\log_w(N_t/B))$. The designs in [1], [7] obtain optimal query times for the key history query, but require $O(N)$ rather than $O(N/B)$ storage. Our design for key-history queries can also be incorporated into other designs, like the Time-Split B-tree of Lomet and Salzberg [13], or the Multiversion B-Tree of Becker et al. [2], to improve the time complexity of key-history queries. However, neither of these papers had suggested any approach to obtain optimal performance for key-history queries.

Note that in our design, the time for version operations at time t is $O(log_w(N_t/B))$, where N_t is the number of live records at time t. The time for query operations depends on N, the total number of version operations. Different assumptions have been made in the literature (see [18]) in this context.

A common proposal has been to use an additional structure, root*, which is an index on the roots of the multiversion access structure at different times. If timestamps are restricted to be consecutive integers then root* can be implemented as an array accessible in $O(1)$ time, and both version and query operations can be made to depend on N_t rather than N. However, in this work, timestamps can be arbitrary monotonically increasing numbers, not necessarily consecutive integers. Alternatively, root* may be implemented as a B-tree index on the roots at different times. In [2], it is assumed that this index is stored in main memory, and no I/O is needed to access it. With this assumption, the times for both query and version operations are made dependent on N_t rather than N. We do not make this strong assumption that part of the structure is stored in memory in our paper.

A more general model allowing updates in past versions of objects (full persistence rather than partial persistence in the terminology of [4]) was proposed in [12]. As has been observed previously, the structure requires $\Theta(N)$ storage (rather than optimal $\Theta(N/B)$) even when specialized for partial persistence, and query performance is no better than our method.

The Monotonic B+-Tree [6] and the methods (AP, ST, and AT trees) in [7] are an alternative approach to designing multiversion access structures, based on keeping the time and key dimensions separate. An advantage of this approach is that insertion and update may require only a constant number of block accesses, rather than logarithmic required by methods which also need to traverse the key space. In these methods, key-range or key-history or time-range view queries are inefficient due to lack of clustering along both dimensions. Two recent methods in this class (albeit without the constant update time) have been proposed in [20], [27]. Both improve upon the storage requirements of [6], and [27] also improves the query times.

Multiversion access structures for main memory databases have been discussed by Tsotras and Gopinath [26]. Since these models do not consider the pagination problems associated with secondary storage, the results are not comparable with our method.

Multiversion access structures in the innovative Postgres system [24] and the schemes in [10], [11] are based on two-dimensional R-trees [8]. Kolovson and Stonebraker [10] proposed combined-media R-tree indexes (improving the cost/performance characteristics of single-media designs [5], [28]). These designs perform best for problems exhibiting clustering in multiple dimensions (spatial structures), in contrast to a temporal application where key value and time are usually unrelated. The SR-tree [11] addresses this problem at the expense of an increase in space from linear to $O(N\log N)$. While this structure also handles deletes using a search-delete-reinsert operation, it can have poor worst-case performance due to the search. The overlapping B+-tree of Manolopoulos and Kapetanakis (see the description in [18]) performs the key and time range queries optimally, but requires $O(N\log N)$ space. The worst-case performance of key history searches can be degraded as blocks that do not contain the key may be accessed. In our design, we achieve asymptotically optimal (with small associated constants) query time for all these queries, and use only $O(N/B)$ storage. Note that combined key-range/time-range queries have been explicitly addressed in [11], though not in [2] or in this paper.

9 Summary

We have presented an efficient multiversion access structure for a transaction-time database, for several important queries. Our method requires optimal storage and query times and logarithmic update times. We permit version operations insert, update and delete on the current database. Queries to both past and present versions are permitted. The following queries can be answered in optimal query time: key range search, key history search and time range view. The key-range query retrieves all records having keys in a specified key range at a specified time, the key history query retrieves all records with a given key in a specified time range, and the time range view query retrieves all records that were current during a specified time interval. Special cases of these queries include the key search query which retrieves a particular version of a record, and the snapshot query which reconstructs the database at some past time.

The design consists of two components: an MVAS, and a *C*-list with a sparse B-tree index. MVAS is a structure that clusters together keys that are current at a given time by their key value. *C*-list is a structure that clusters all versions of a key. The two-way-linked sparse B-tree index on *C*-list enables efficient processing of key-history queries. The MVAS can be used to answer key range and time-range view queries in optimal query time, while the MVAS in conjunction with *C*-list and the sparse index are used to obtain optimal query performance for the key-history query. The time required to update all the components for all three version operations is logarithmic (amortized) in the size of the current database, matching the lower bound for the key-range query.

The closest design to ours previously was the multiversion B-tree of Becker et al. [2]. The MVAS of this paper improves significantly upon the storage requirements of their design. This is obtained by using new storage-conserving policies for block overflows and underflows. The new policies attempt partial copying of a block (saving on the number of new copies created) and using space in an existing block (rather that creating a new block). We provide a nontrivial analysis of the storage required using these policies, and are able to obtain a worst-case storage bound of $5N/(B-5)$ data blocks to hold N versions. For comparison, the earliest design of this category due to Easton [5] had a worst-case storage of about $4N/B$, but did not handle deletes. The bound in [2] is about $10N/B$.

The second contribution of this paper is to design a structure to handle the key-history query in optimal query time, and logarithmic update. Previous designs [1], [7] achieved optimal query times but required significantly more storage ($O(N)$ versus $O(N/B)$). Our design can also be incorporated in the Time-Split-B-Tree [13] as well as Multiversion B-tree [2] to improve the performance of key-history queries. Finally, simulation results indicate that the MVAS design achieves good performance in practice.

Acknowledgments

This paper was partially supported by the National Science Foundation, U.S. Department of Defense Advanced Research Projects Agency Grant CCR 9006300, and NSF Grants CCR 9010366 and CCR 9303011.

The simulation program was developed by H. Jiang under the guidance of Rakesh Verma. We thank her for finding a missing case in the specifications. The program was further debugged and optimized by Verma. We thank the anonymous reviewers of the paper for their time, constructive comments, and excellent suggestions. The paper is much improved as a result. The idea of using a sparse index on top of the *C*-list was inspired by a suggestion from reviewer no. 1. For their generous hospitality, Verma thanks Michael Rusinowitch, Claude Kirchner, and CRIN in Nancy, France, where the revision was completed.

REFERENCES

[1] I. Ahn and R. Snodgrass, "Partitioned Storage for Temporal Databases," *Information Systems*, vol. 13, pp. 369–391, 1988.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "On Optimal Multiversion Access Structures," *Proc. Workshop Advances in Spatial Databases*, pp. 123–141, 1993.

[3] J. Clifford and A. Tansel, "On an Algebra for Historical Relational Databases: Two Views," *Proc. ACM SIGMOD Conf.*, 1985.

[4] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan, "Making Data Structures Persistent," *J. Computer and System Sciences*, vol. 38, pp. 86–124, 1989.

[5] M. Easton, "Key-Sequence Data Sets on Indelible Storage," *IBM J. of Research and Development*, vol. 30, pp. 230–241, 1986.

[6] R. Elmasri, G.T.J. Wuu, and Y.-J. Kim, "The Time Index: An Access Structure for Temporal Data," *Proc. VLDB Conf.*, pp. 1–12, 1990.

[7] H. Gunadhi and A. Segev, "Efficient Indexing Methods for Temporal Relations," *IEEE Trans. Knowledge and Data Eng.*, vol. 5, pp. 496–509, 1993.

[8] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Conf.*, 1984.

[9] C.S. Jensen et al., "A Consensus Glossary of Temporal Database Concepts," *ACM SIGMOD Record*, vol. 23, no. 1, pp. 52–64, 1994.

[10] C. Kolovson and M. Stonebraker, "Indexing Techniques for Historical Databases," *Proc. IEEE Conf. Data Eng.*, pp. 127–137, 1989.

[11] C. Kolovson and M. Stonebraker, "Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data," *Proc. ACM SIGMOD Conf.*, pp. 138–147, 1991.

[12] S. Lanka and E. Mays, "Fully Persistent B+ Trees," *Proc. ACM SIGMOD Conf.*, pp. 426–435, 1991.

[13] D. Lomet and B. Salzberg, "Access Methods for Multiversion Data," *Proc. ACM SIGMOD Conf.*, pp. 315–324, 1989.

[14] D. Lomet and B. Salzberg, "The Performance of a Multiversion Access Method," *Proc. ACM SIGMOD Conf.*, pp. 353–363, 1990.

[15] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, "Designing DBMS Support for the Temporal Dimension," *Proc. ACM SIGMOD Conf.*, pp. 115–130, 1984.

[16] Y. Manolopoulos, "Reverse Chaining for Answering Temporal Logical Queries," *The Computer J.*, vol. 35, no. 6, pp. 666–668, 1992.

[17] S.B. Navathe and R. Ahmed, "A Temporal Relational Model and a Query Language," *Information Sciences*, vol. 49, nos. 1, 2, and 3, 1989.

[18] B. Salzberg and V.J. Tsotras, "A Comparison of Access Methods for Time Evolving Data," Technical Report No. CATT-TR-94-81, Polytechnic Univ., 1994.

[19] A. Segev and A. Shoshani, "Logical Modeling of Temporal Data," *Proc. ACM SIGMOD Conf.*, 1987.

[20] H. Shen, B.C. Ooi, and H.J. Lu, "The tp-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases," *Proc. IEEE Conf. Data Eng.*, pp. 274–281, 1994.

[21] R. Snodgrass, "The Temporal Query Language Tquel," *ACM Trans. Database Systems*, vol. 12, no. 2, 1987.

[22] R. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," *Proc. ACM SIGMOD Conf.*, 1985.

[23] R. Snodgrass and I. Ahn, "Temporal Databases," *Computer*, vol. 10, no. 9, 1986.

[24] M. Stonebraker, "The Design of the Postgres Storage System," *Proc. VLDB Conf.*, pp. 289–300, 1987.

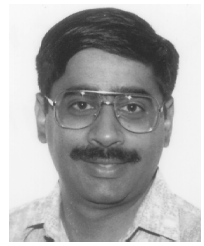
[25] V.J. Tsotras and N. Kangelaris, "The Snapshot Index: An I/O-Optimal Access Method for Timeslice Queries," *Information Systems*, vol. 20, no. 3, pp. 237–260, 1995.

[26] V.J. Tsotras and B. Gopinath, "Efficient Algorithms for Managing the History of Evolving Databases," *Proc. Int'l Conf. Database Theory*, 1990.

[27] R.M. Verma and P.J. Varman, "Efficient Archivable Time Index: A Dynamic Indexing Scheme for Temporal Data," *Proc. Int'l Conf. Computer Systems*. Tata McGraw-Hill, 1994.

[28] J.S. Vitter, "An Efficient I/O Interface for Optical Disks," *ACM Trans. Database Systems*, vol. 10, no. 2, pp. 129–162, 1985.

Peter J. Varman received his BTech degree from the Indian Institute of Technology, Kanpur, in 1978; and his MS and PhD degrees from the University of Texas at Austin in 1980 and 1983, respectively—all in electrical engineering. He is currently an associate professor in the ECE Department at Rice University. He has held visiting positions at the IBM T.J. Watson and Almaden Research Centers and at Nanyang Technological University in Singapore. His research interests are in the areas of parallel computation, parallel I/O, database implementation and performance, and temporal and spatial databases. He has published extensively in these areas and is a co-inventor of a U.S. patent awarded for multiprocessor sorting. Dr. Varman is a member of the ACM and the New York Academy of Sciences.



Rakesh M. Verma received his BTech degree (with honors and a Gold Medal) in electronics engineering in 1984 from the Institute of Technology, Banaras Hindu University, Varanasi, India. He received the MS degree in 1985 and the PhD degree in 1989, both in computer science, form the State University of New York at Stony Brook. He was a Catacosinos fellow at SUNY, Stony Brook, in 1988 and 1989. He is currently an associate professor in the Computer Science Department at the University of Houston. He served as a visiting professor during the summer of 1995 at the Center for Informatics Research (CRIN) in Nancy, France, at a joint center of INRIA Lorraine, and at the French National Research Agency (CNRS). His main research interests are in the areas of term rewriting and symbolic computation with applications to automated reasoning/verification, functional and equational programming, and computer algebra; and storage and access structures for temporal and spatial databases with applications to scientific databases and geographic information systems. Dr. Verma is a member of the ACM.