Indexing Alternatives for Multiversion Locking

Paul M. Bober
Michael J. Carey

Technical Report #1184

November 1993

Indexing Alternatives for Multiversion Locking*

Paul M. Bober** and Michael J. Carey

Computer Sciences Department, University of Wisconsin, Madison Wisconsin 53706, USA

Abstract. Multiversion two-phase locking (MV2PL) provides on-line serializable queries without introducing the long blocking delays that can occur with conventional two-phase locking (2PL). MV2PL requires indexing structures, however, that are capable of supporting multiple versions of data. In this paper, we present several options for extending single-version indexing schemes for use with MV2PL. These basic approaches are largely orthogonal to the underlying indexing structure (e.g., hashing or B+ trees). The options considered differ in where they place version selection information (i.e., references to individual versions); this information is placed either with the data or with the index entries of one or more of the indices. We also present the results from a performance study that show that placing the version selection information with the data is usually the best option, since it keeps the indices smaller and thus enables a larger fraction of the index pages to remain cached in the buffer pool.

1 INTRODUCTION

Due to the adoption of relational database technology and the increasing ability of database systems to efficiently execute ad-hoc queries, query processing is becoming an increasingly important function of transaction processing systems. The concurrency control algorithm found in most commercial database systems, two-phase locking (2PL) [14], however, does not efficiently support on-line query processing. This is because 2PL causes queries to lock large regions of data for long periods of time, thus causing update transactions to suffer long delays. As a result, many applications run queries without obtaining locks or using only short-term locks, allowing the queries to see transaction-inconsistent answers. These approaches are referred to as GO processing and cursor stability locking, respectively.

To solve this data contention problem while providing consistent answers to queries, a multiversion extension to two-phase locking was proposed and implemented in a system developed at Prime in the early 1980s [12]. This extension was also used in a system developed at Computer Corporation of America (CCA) [10, 11], and it has subsequently been incorporated in DEC's Rdb product [23]. In multiversion two-phase

* This research was partially supported by an IBM Research Initiation Grant
An abridged version of this paper will appear in Proceedings of the Fourth International Conference on Extending Database Technology, Cambridge, U.K., March 1994

** Author's current address: Transarc Corporation, The Gulf Tower, 707 Grant Street, Pittsburgh, PA 15219

locking (MV2PL), a timestamp mechanism is used in conjunction with the temporary retention of prior versions of data so that a read-only query can serialize before all update transactions that were active during any portion of its lifetime. In MV2PL, read-only queries do not contribute to data contention since they do not have to set or wait for locks. This form of versioning, where old copies of data are retained temporarily for concurrency control purposes (as opposed to long-term retention for historical queries), has been referred to as *transient versioning* [22].

Since indexes are important for good performance in database systems, it is important to to determine how they may coexist with MV2PL. Conventional single-version indexing structures such as B+ trees and hashing are not entirely compatible with MV2PL in their current forms, as they support searches on key value alone (not on both key value and timestamp together). Without timestamp information encoded in the index, a given query will have no way of knowing if an entry with a matching key references a version that it should see without first retrieving the version and examining its timestamp information. Thus, frequent *false drops* may occur since not all retrieved tuples are actually needed. Furthermore since false drops are possible, the use of *index-only plans*, a common relational query processing optimization that avoids retrieving actual tuples when only indexed attribute values are needed, is ruled out.[3]

To support efficient query processing, it is clear that an MV2PL system must utilize an indexing scheme specifically designed for multiversion data. One approach, taken in DEC's Rdb system, is to treat index nodes like data records at the storage level, including having MV2PL applied to them [17]. While this approach supports index-only plans, it is not compatible with the use of high performance non-2PL B+ tree concurrency control algorithms such as those proposed in [3, 21, 22]. Because (non-2PL) B-tree concurrency control algorithms are widely viewed as being important to achieving acceptable performance, we do not consider the Rdb approach further.

A number of other multiversion indexing approaches have been proposed in the literature; examples include [13, 28, 18, 20, 22]. With the exception of [22], however, all of these proposed indexing schemes are designed to support historical databases, where out-of-date versions are retained for an arbitrary length of time. In contrast to transient versioning databases, historical databases may have a large number of versions of each tuple (some of which may have been migrated to tertiary storage, e.g., optical disk). Because of this, the basic indexing design tradeoffs are different for the two types of versioning. For example, while it

[3] Using an index-only plan, a query computing the average salary of a group of employees, for example, does not have to retrieve the employee tuples if an index on employee salary exists; instead it can compute the average by simply scanning the leaves of the index.

might be reasonable in a transient versioning system to require a query to traverse the entire length of a (short) linked list of the existing versions of a tuple, this would not be reasonable in a historical versioning system. Furthermore, it is likely that a historical versioning system will be required to store pieces of its indexes on tertiary store, as the indexes are apt to grow very large. Lastly, efficient garbage collection is very important in a transient versioning system, as versions are not needed for very long once they have been replaced by a more current version.

In this paper, we compare a range of possible multiversion indexing approaches that are designed specifically for use with MV2PL. This paper is an extension of our previous work, where we proposed and studied the *on-page caching* scheme for placing transient multiversion data on secondary storage [7]; that scheme in turn is a refinement of CCA's version pool scheme [10]. Each of the multiversion indexing approaches that we study in this paper are integrated with on-page caching to present a complete version placement and indexing solution for MV2PL.

The remainder of the paper is organized as follows: In Section 2 we review the MV2PL algorithm, the CCA version pool scheme, and our on-page caching refinement. In Section 3 we describe four multiversion indexing schemes, and in Section 4, we describe the simulation model that we will use to compare them. In Section 5, we present the results of simulation experiments that compare the indexing schemes in terms of their I/O costs for queries and update transactions. Lastly, we present our conclusions in Section 6.

2 MULTIVERSION BACKGROUND

In this section we set the stage for the discussion of multiversion indexing approaches by reviewing the MV2PL algorithm, the CCA version pool scheme for managing storage for multiple versions of data, and our on-page caching refinement to the CCA version pool scheme.

MV2PL is only one of a number of multiversion concurrency control algorithms that have been published in the literature. Because it is a direct extension of the de facto industry standard, 2PL, we are primarily concerned with indexing in the context of MV2PL in this work. For completeness, however, we wish to identify some of the other proposals here. To the best of our knowledge, Reed's distributed timestamp ordering scheme [Reed83] was actually the first multiversion concurrency control algorithm proposal. Several 2PL-based algorithms that retain at most two versions of data in order to reduce blocking due to read/write conflicts have also been proposed [5, 27]. Other multiversion extensions of single-version concurrency control algorithms include: multiversion optimistic validation [25, 9, 19] multiversion timestamp

ordering [2], and the multiversion tree protocol [26]. Finally, in [7], we presented a generalization of MV2PL that provides queries with a tradeoff between consistency and performance.

2.1 Multiversion Two-Phase Locking (MV2PL)

In MV2PL, each transaction T is assigned a startup timestamp, $T_S(T)$, when it begins to run, and a commit timestamp, $T_C(T)$, when it reaches its commit point. Transactions are classified at startup time as being either *read-only* or *update* transactions. When an update transaction reads or writes a page[4] it locks the page, as in traditional 2PL, and then accesses the current version. Update transactions must block when lock conflicts occur. When a page is written, a new version is created and stamped with the commit timestamp of its creator; this timestamp is referred to as the version's create timestamp (CTS).[5] When a read-only query Q wishes to access a page, on the other hand, it simply reads the most recent version of the page with a timestamp less than or equal to $T_S(Q)$. Since each version is stamped with the commit timestamp of its creator, Q will only read versions written by transactions that committed before Q began running. Thus, Q will be serialized after all transactions that committed prior to its startup, but before all transactions that are active during any portion of its lifetime – as though it ran instantaneously at its starting time. As a result, read-only transactions never have to set or wait for locks in MV2PL.

When an update transaction deletes a page in MV2PL, the prior versions of the page must remain in the database until all queries which may require them have completed. Deletes may thus be handled by assigning a *delete timestamp* (DTS) to the last version of each page. Initially, the DTS of the most recent version is infinite, signifying that it is in the current database. When a page is deleted, the commit timestamp of the deleter is assigned to the DTS of the current version (denoting that it is no longer part of the current database). Update transactions should access a page only if it has a current version. Likewise, a query may access a page only if the query's startup timestamp is less than the DTS of the most recent version (i.e., only if the page was not deleted as of the query's arrival time).

2.2 The CCA Version Pool Organization

To maintain the set of versions needed by ongoing queries, the CCA scheme divides the stored database into two parts: the main segment and the version pool. The main segment contains the current version

[4] MV2PL utilized page-level locking in its original form.

[5] Actually, to reduce commit-time processing in the absence of a no-steal buffer management policy, the page is stamped with the creator's transaction id and a separately maintained list is used to map from transaction ids to commit timestamps [10].

of every page in the database, and the version pool contains prior versions of pages. The version pool is organized as a circular buffer, much like the log in a traditional recovery manager [15]. The CCA design chains the versions in reverse chronological order; in Section 3, we discuss other ways of organizing sets of versions. Three attractive properties of the version pool are that (i) updates are performed in-place, allowing clustering of current versions to be maintained, (ii) version pool writes are sequential (i.e., similar to log writes)[6]. and (iii) storage reclamation is relatively straightforward. Figure 1 depicts the main segment of the database, the version pool, the pointers used to manage version pool space, and the version chain

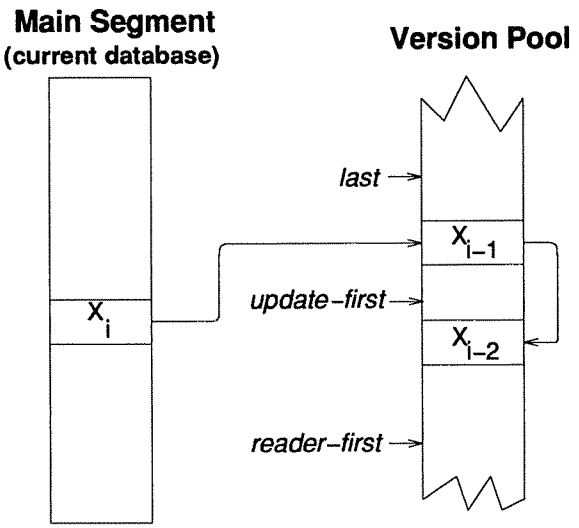

Fig. 1. CCA Version Pool Organization

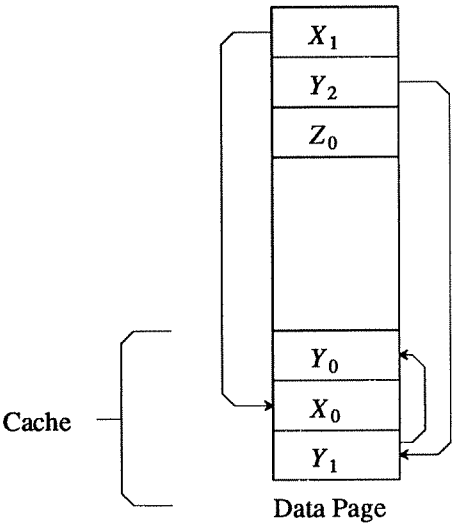

Fig. 2. A Data Page with a Cache

for a page X. Version pool entries between *reader-first* and *last* in the figure contain versions that may be needed to satisfy read requests for ongoing queries. Entries between *update-first* and *last* contain page versions recorded by ongoing (or recently committed) update transactions.

Garbage collection in the version pool is done when the oldest query finishes, thus allowing the *reader-first* pointer to move. Garbage collection is simple due to the sequential nature of this deallocation process; however, a problem with the CCA scheme is that a very long running query may hold up the reclamation of version pool space. Another problem is that the ordinary sequential I/O patterns of queries may become disrupted by random I/O operations to the version pool. Moreover, because a query must read successively older versions (relative to the current database) as it ages, the number of version pool I/O operations that

[6] In contrast, DEC's Rdb system stores old versions of records on "shadow" pages which must be first read and then written whenever an update occurs [17].

it must make to read a given page increases with time. As a result, queries may begin to thrash if they are sufficiently large [6]. The on-page version caching refinement discussed next was designed to alleviate these problems.

2.3 On-Page Version Caching

In [6], we presented and studied a record-level refinement to the CCA version pool scheme in which versions are maintained on records (as opposed to pages) and a small portion of each main segment (i.e., data) page is reserved for caching prior versions. Such an on-page cache reduces the number of read operations required for accessing prior versions. Another benefit is that versions may "die" (i.e., become unnecessary for maintaining the view of a current query) while still in an on-page cache and thus not have to be appended to the version pool at all. We review the concepts of the on-page cache here so that we may show how it can be integrated with the various indexing approaches. Figure 2 shows a data page with records *X*, *Y*, and *Z*, and a cache size of 3. Prior versions of these records are resident in the on-page cache in the figure.

With on-page caching, updates to records are handled in the following manner: When a tuple is updated, the current version is copied [7] into the cache before it is replaced by the new version. Likewise, when a tuple is deleted, the current version is also copied into the cache. If the cache is already full, *garbage collection* is attempted on the page. Garbage collection examines each entry in the cache to determine whether or not it is needed to construct the view of any current query. If garbage collection is unsuccessful in freeing a cache slot, then some prior version is chosen for replacement. The replacement algorithm chooses the least recently updated entry for replacement (i.e., the entry which has resided in the cache the longest is moved to the version pool).

In addition to the cache replacement policy, there is also a write policy that determines when a cached prior version should be appended to the version pool. The *write-one* policy appends a version only when it is chosen to be replaced in the cache. This policy attempts to minimize the size of the version pool by 1) keeping only one copy of each prior version and 2) allowing a prior version the maximum chance of being garbage-collected before being written to the version pool. In contrast, the *write-all* policy appends *all* of the prior versions in a page's cache to the version pool at once; this is done when a cache overflow occurs

[7] In practice, copying a version into the cache simply means moving its entry in the page's slot table. The cache does not have to be a physically contiguous region of the page; cache overflows can be detected by maintaining a count of bytes occupied by prior versions.

and the least recently updated entry has not yet been appended to the version pool. In this scheme, a version chain will actually contain two copies of those versions that have been written to the version pool but not yet replaced from the cache. The write-all policy introduces a degree of positional clustering in the version pool by attempting to cluster versions from the same main segment page; this benefits queries that sequentially scan the data by increasing their locality of reference to the version pool. Because of the added positional clustering, the write-all policy was shown to be able to complete queries at a faster rate than write-one [6]. Despite having to replicate some prior versions under write-all, the more rapid completion of queries under this policy was also shown in many cases to lead to a significantly lower overall storage cost than under write-one; in other cases, the storage cost was not appreciably higher.

As we pointed out earlier, a very long running query may hold up the reclamation of version pool space. In contrast, versions that reside in an on-page cache may be garbage-collected soon after they become unnecessary. Also, on-page garbage collection is done whenever an update occurs on a page whose cache is full, at which time each prior version in the cache is examined to determine whether it is still needed. Since such update operations dirty the data page anyway, on-page garbage collection is essentially free. For further details about version garbage collection and other aspects of on-page caching, the reader is referred to [6].

3 MULTIVERSION INDEXING APPROACHES

In this section, we discuss options for extending single-version indexing schemes to handle multiversion data. We outline four different approaches here and discuss their performance tradeoffs. The approaches include: Chaining (CH), Data Page Version Selection (DP), Primary Index Version Selection (PI), and All Index Version Selection (AI). These basic approaches are largely orthogonal to both the version placement scheme employed and to the underlying indexing structure (e.g., hashing or B+ trees). We describe the approaches here as they would work with the on-page caching method for storing prior versions (as described in Section 2) and the B+ tree indexing method [4].

We used several criterion to select the schemes that we will be considering here. First, to be practical, we decided that the schemes should involve only relatively simple changes to proven indexing methods (i.e., we didn't want to consider something so foreign that nobody would want to implement it). Furthermore, because versions come and go rapidly in transient versioning, garbage collection should be relatively inexpensive. Lastly, we decided that index-only plans should be supported since they are an important

optimization in many existing systems.

The multiversion indexing schemes that we consider differ in how they accomplish *version selection*, the mechanism by which the appropriate version of a tuple is located in the collection of existing versions. Version selection information is either placed with the data or with the index entries of one or more of the indices. In all of the schemes, we assume that relations have a single primary key index and zero or more secondary key indices, and that tuples are stored separately from the indices. For purposes of presentation, we further assume that primary key values cannot be updated.

3.1 Chaining (CH)

In the Chaining (CH) versioning selection scheme, each index leaf entry simply references the most recent version of a tuple; the remainder of the versions are chained behind the current version in reverse chronological order, as in the CCA scheme [10]. The organization of data pages (with on-page caching) and the version pool was discussed in the previous section. As described earlier, each version of a tuple has a *create timestamp* (CTS) which is the commit timestamp of the transaction that wrote the version. The most recent version also has a *delete timestamp* (DTS) which is the commit timestamp of the transaction that deleted the tuple; the value of the field is infinite if the tuple exists in the current database.

Figure 3 illustrates this scheme by showing an example of how a single tuple is stored and indexed both on the primary key and on a secondary key. Interior nodes of the index are not shown since they are identical to those in a single-version B+ tree, as in all of the schemes that will be considered here. The tuple in Figure 3 has four versions: (a1, b1, c1) with CTS 25, (a1, b1, c2) with CTS 35, (a1, b2, c2) with CTS 50, and (a1, b2, c3) with CTS 60. The primary key index is built on the first attribute, with the secondary key index on the second. Currently, there are three queries running in the system: Q_1 with a startup timestamp of 25, Q_2 with startup timestamp 40, and Q_3 with startup timestamp 55. The existence of these queries necessitates the retention of all of the prior versions shown in the figure.

As shown in the figure, index leaf page entries in the CH scheme consist of a key, a tuple pointer, a create timestamp (CTS), and a delete timestamp (DTS). The CTS field contains the timestamp of the transaction which inserted the key into the index, and the DTS field contains the timestamp of the transaction that (logically) deleted the key from the index. Together, the CTS and DTS fields represent the range of time over which an index entry's key value matches some version of the referenced tuple. For example, in Figure 3, the CTS and DTS fields in the secondary index entry with key b denote that all versions of the illustrated tuple with timestamps greater than or equal to 25 and less than 50 have b1 as

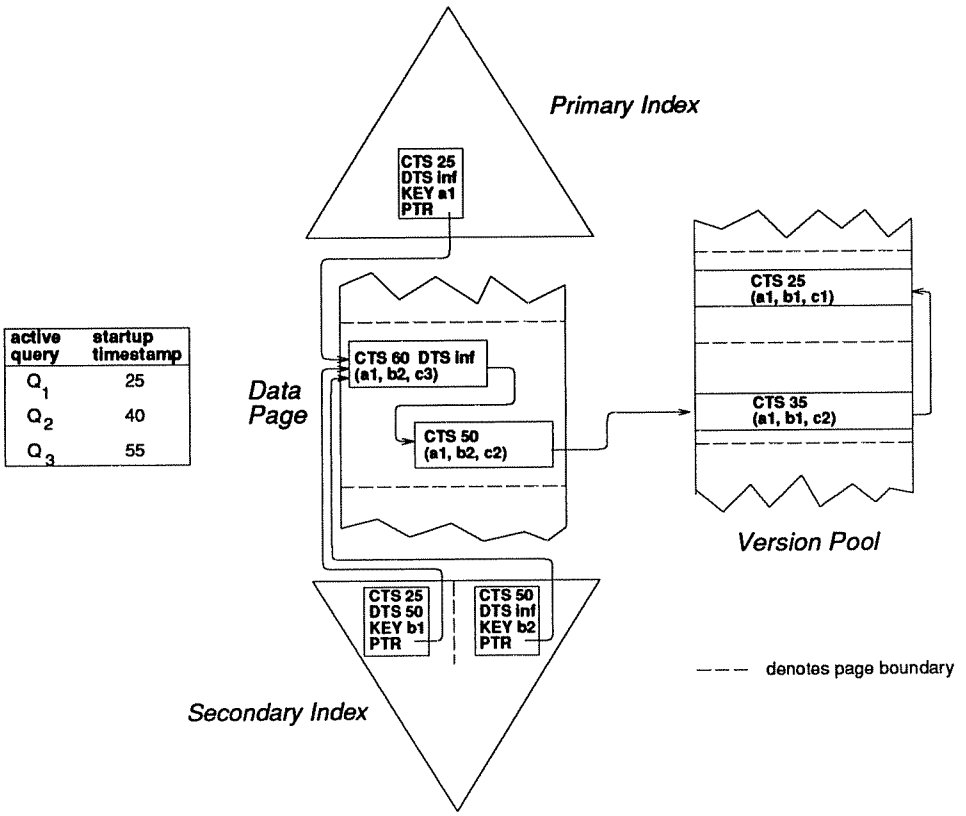

Fig. 3. Chaining

the value of their second attribute; likewise the CTS and DTS fields in the entry with key b2 denote that all versions with timestamps greater than or equal to an 50 have an indexed attribute value of b2. Note that the entries that reference a given tuple (from within the same index) have non-overlapping timestamp ranges since each version may have only one key value at a time. Delete operations do not physically remove leaf entries because they may be needed by queries to provide access paths to prior versions of tuples. We will discuss shortly how the index is searched and how leaf entries are eventually garbage-collected when they are no longer needed.

With the exception of having logical deletes (i.e., setting the DTS field instead of immediately removing an entry), operations on the multiversion B+ tree parallel those on an ordinary B+ tree. A single insertion is made into each index when a tuple is inserted into a relation; a single logical deletion is made in each index when a tuple is deleted; both an insertion and a logical deletion are made in each affected index when a tuple is modified (i.e., for each changed key value, the new value is inserted and the old value is deleted). Later we will see that additional index operations are required in some of the other multiversion

indexing schemes.

The multiversion index is searched just like a B+ tree, except that transactions filter out entries which do not pertain to the database state that they are viewing. An update transaction, which views the current database state, pays attention only to index entries whose DTS is infinity (*inf* in the figure). A query Q, which views the state of the database as of its arrival, pays attention only to index entries whose CTS and DTS values satisfy the inequality $CTS \leq T_S(Q) < DTS$. Such entries were inserted, but not yet deleted, as of Q's arrival in the system. By following these rules, false drops do not occur, and therefore index-only plans may be utilized when applicable. In the example shown in Figure 3, queries Q_1 and Q_2 must follow the secondary index entry with key b1, while Q_3 must follow the entry with key b2.

As in all of the schemes that we will be discussing, *garbage collection* within an index leaf page is invoked when the page overflows. Since the page is already dirty and pinned in the buffer pool at such times, index garbage collection does not require any additional I/O operations. The garbage collection process examines each logically deleted entry (i.e., each one with a finite DTS) to determine whether or not it is still needed for some active query. Specifically, an entry is still needed if there exists a query Q $\in$ active queries such that $CTS \leq T_S(Q) < DTS$ (as described above). A logically deleted entry is physically removed if it is not needed for any active query; such an entry will never be needed later for a subsequently arriving query, as such queries will be assigned startup timestamps that are greater than or equal to the entry's DTS.

To minimize the additional storage overhead due to versioning, *compression* of the timestamp information (CTS and DTS) is possible. This will be especially important for indices with small keys. To this end, a single bit may be used to encode a DTS value of infinity. Likewise, a single bit may also be used to encode any CTS value that is less than the startup timestamp of all active queries, as all that matters is the fact that the entry preceded their arrival. (In practice, these two bits together will require extending index entries by a whole byte.) If a tuple requires only one leaf entry in some index, the entry may have both fields compressed. This occurs when the index key value in the tuple has remained constant since the arrival of the oldest active query, which is likely to be a common case. In the example in Figure 3, the CTS of the secondary leaf entry having key b1 may be compressed, and likewise for the DTS of the entry having key b2. Thus, an index on an attribute that is rarely changed will remain close in size to a single-version B+ tree. Furthermore, during periods when queries are not run, the indices may be gradually compressed down towards the size of ordinary B+ trees (with the exception of the additional byte per entry) by merging pages during garbage collection; when queries reenter the system, the indices will gradually expand as needed. The main disadvantage of compressing the timestamps is the added overhead

of maintaining growing and shrinking index entries, but code to handle this should already be part of any B+ tree implementation that supports variable-length keys.

3.2 Data Page Version Selection (DP)

A drawback of the chaining approach used in CH is that a long-running query may have to read a large number of pages to reach the version of a tuple that it needs. The data page (DP) version selection scheme is a modification of CH that limits the number of pages that a query must read to two (exclusive of index pages). It accomplishes this by recording the addresses and timestamps of each version of a tuple in a small table known as a *version selection table* (VST).[8]

The VST is located on the data page that contains the current version of the tuple (or in the case of a deleted tuple, on the page that contained the final version). Rather than referencing the current version of a tuple, an index leaf entry references the tuple's VST. Figure 4 illustrates the DP scheme by modifying the example used to illustrate the CH scheme. From the figure, it can be seen that a query must now read at most two pages (a data page and a version pool page) to locate any tuple. In contrast, to locate the version with CTS 25 in Figure 3 under the DP scheme, a query would have to read three pages.

A disadvantage of DP over CH is the additional room on data pages consumed by the VST entries of versions that have migrated to the version pool. However, since VST entries are small, this is not likely to have a significant impact unless the tuples themselves are small in size.

3.3 Primary Index Version Selection (PI)

The Primary Index (PI) version selection scheme is a modification of DP that stores the version selection table together with the tuple's primary index leaf page entry (instead of on a data page). It is similar[9] to the scheme presented in [22], which is the only previously published indexing scheme for multiversion locking that we are aware of.

Figure 5 illustrates the PI scheme by adapting the running example. Note that a versioned tuple has only one entry in the primary index because primary index keys cannot be changed.

[8] For versions that are replicated under the write-all cache write policy, this scheme would list replicated versions in the VST twice (i.e., once for their on-page cache copy and once for their version pool copy).

[9] The overall versioning scheme in [22] differs somewhat in that it bounds the number of versions of each tuple by essentially restricting the number of query startup timestamps in use simultaneously. This difference is orthogonal to the indexing issues that we are discussing here, however.

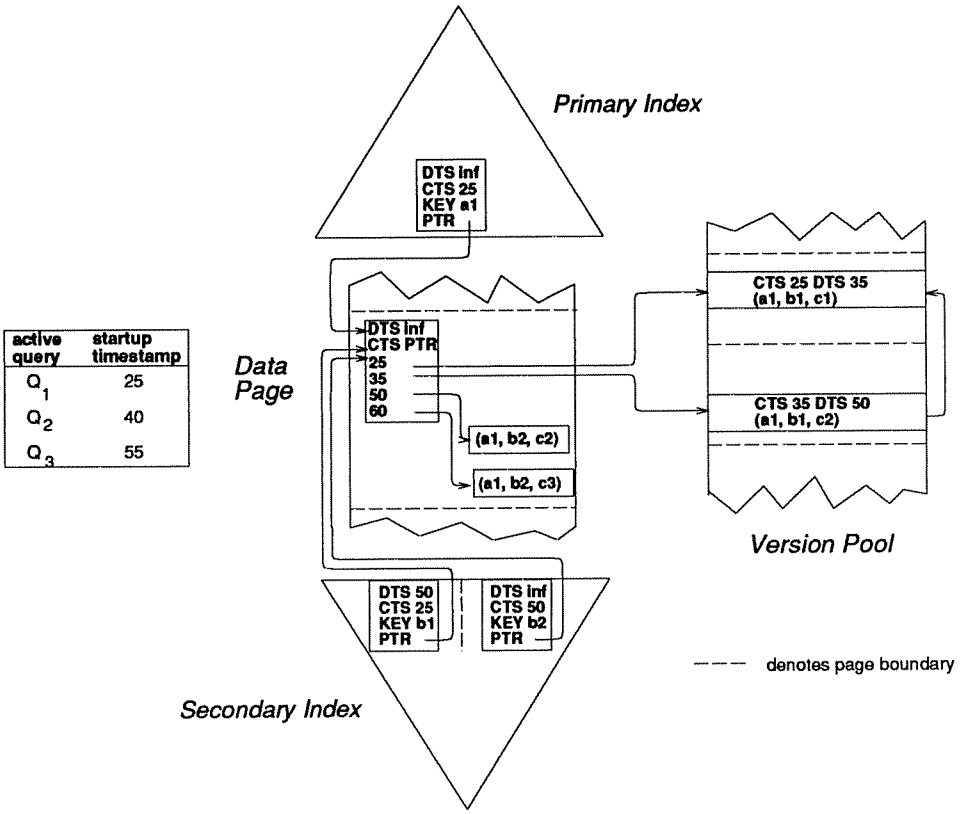

Fig. 4. Data Page Version Selection

The motivation for placing VSTs in the primary index is that it enables queries to retrieve versions through the primary index by reading only a single data page or version pool page. There are several drawbacks to this approach, however. One drawback is that the pointer to a version from its VST must also be updated when the version is migrated to the version pool. If on-page caching is used, this increases the path length of update transactions that need to free cache space in order to modify or delete a tuple. Another drawback is that the presence of the VSTs on primary index leaf pages will lead to a larger primary index.

The largest drawback with placing the VSTs in the primary index is that secondary indices no longer provide queries with direct access to the data. Instead, secondary indices provide a mapping from secondary key to primary key, with the data being retrieved through the primary index. As a potentially important optimization for update transactions, however, a secondary index entry for the *current* version of a tuple can store the address of the current version. This shortcut is illustrated in Figure 5 by the presence of the CURR field in each secondary index entry. However, this optimization can be used only if the current version

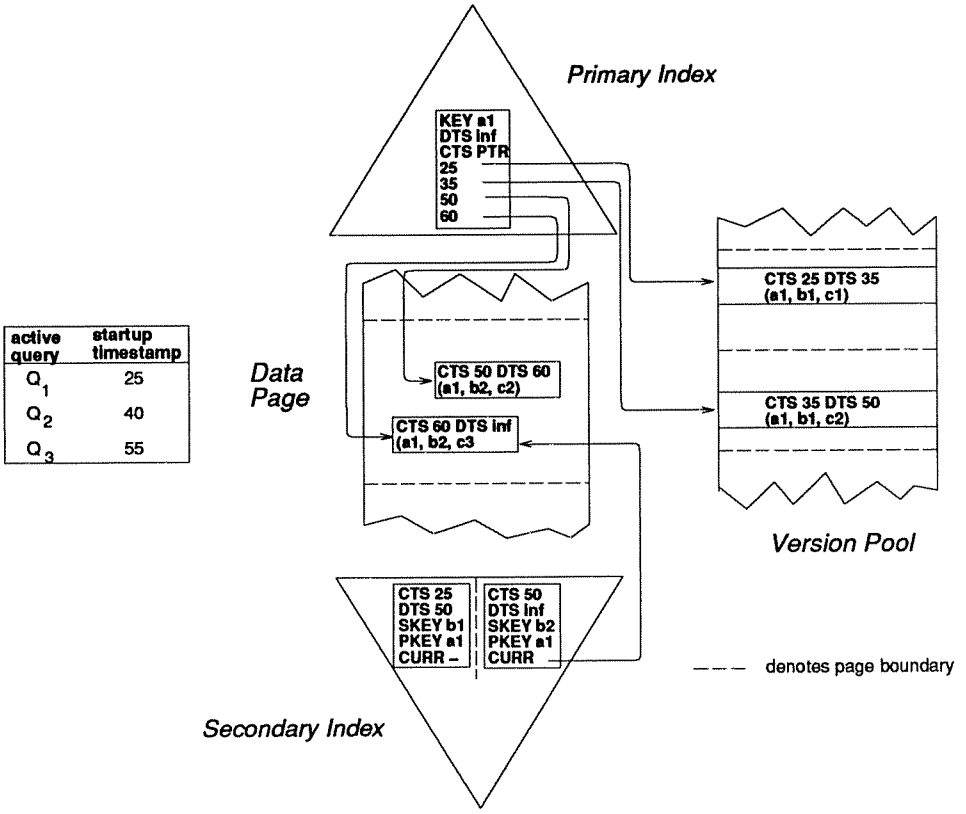

Fig. 5. Primary Index Version Selection

of a given tuple is always stored in a fixed location (determined when the tuple is created). Furthermore, read-only queries cannot use this optimization because they cannot tell which version of a tuple to retrieve without first examining the tuple's VST in the primary index.

Finally, in terms of performance, requiring all read-only queries to access data through the primary index is likely to be problematic unless most queries are primary index scans anyway or a large fraction of the primary index remains resident in the buffer pool. Otherwise, if the buffer pool is not sufficiently large, a secondary index scan will generate a random I/O pattern in the primary index. Thus, even if the data were ordered (clustered) on the relevant secondary index key, the query's I/O pattern would be partially random. As a result, it appears unlikely that this scheme can perform well for queries using a secondary index.

3.4 All Index Version Selection (AI)

In the PI scheme, the primary index is an efficient access method for primary key queries because its leaves contain the addresses of all tuple versions; secondary indices are inefficient for queries, however, because accesses must additionally go through the primary index. The all index (AI) version selection scheme is a modification of PI that places VSTs in the leaf entries of *all* indices (secondary as well as primary). This allows direct access to the data from each index, thus removing the aforementioned inefficiency.

Figure 6 illustrates the AI scheme by adapting the running example one last time. The figure shows the addition of a VST in each secondary index leaf entry, providing a direct access path for queries from the secondary indices to the data. However, a drawback of this modification is that placing additional information in the secondary indices increases the size of all indices. Another serious drawback of the AI scheme is the additional path length that it imposes on update transactions when versions are created or when they are migrated to the version pool. In the AI scheme, when a new version is created as a

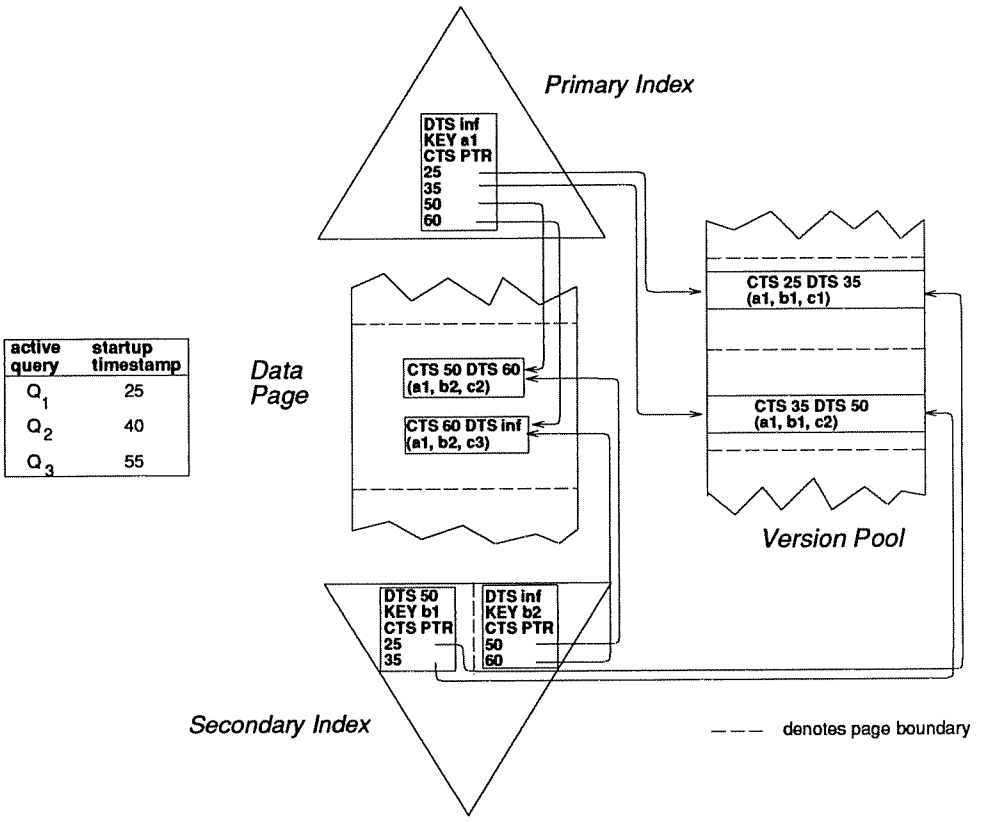

Fig. 6. All Index Version Selection

result of a tuple modification, *every* secondary index must be updated–even if the associated key value was unaffected by the modification. In contrast, in the other multiversion indexing schemes, a secondary index does not have to be updated if the modification does not change the indexed attribute value. For example, in Figure 6, the creation of the versions (a1, b1, c2), and (a1, b2, c3) required placing their addresses in the secondary index VSTs; in the other multiversion indexing schemes, the creation of these versions did not require updating the secondary index at all. Likewise, when a version is migrated to the version pool, all of the references to the version must be updated.

3.5 Summary of Multiversion Indexing Approaches

In this section we have outlined a range of B+ tree based multiversion indexing schemes, with each scheme differing in how it supports version selection. In CH, version selection is supported by chaining versions in reverse chronological order, while in DP, version selection is accomplished via the use of a VST that is stored together with the current version of a tuple. In PI, VSTs are stored in the primary index instead, and in AI, VSTs are stored in all of the indices. The advantage of placing version selection information in an index is that it allows queries to directly access the versions that they need from the index without having to go through one or more intermediate data pages for each version that has been migrated to the version pool. A drawback to placing version selection information in an index is that when a tuple is modified (i.e., a new version of the tuple is created), version selection information for the tuple must be updated in the index–even if the key value for that index was not affected by the change. In the next section we describe a simulation model that we will use to compare these alternative schemes, and in the following section we present the results.

4 SIMULATION MODEL

In this section, we describe the model that we will be using to compare the performance of the different approaches to adding versioning to B+ tree indexes. The model is designed to predict the I/O cost of update and search operations issued by short update transactions and long-running queries, respectively. We consider update operations including record insertions, deletions, and modifications, and queries executing index scans.

We do not model the details associated with locking of the indexes and data in the simulation model; we do this to avoid the significant overhead that they would introduce in terms of code complexity and

simulation execution time. To insure that simulated transactions see a consistent view of the meta-data (indexes, VSTs, etc.) in the absence of concurrency control, we execute record updates atomically, and we handle each query's initial index descent similarly (i.e., before it reaches the leaf level). It should be noted that this modeling approach requires that we have only a single resource in our model (i.e., the resource that we believe to be the bottleneck in a real system), as multiple resources would not be utilized properly; we thus model a DBMS with a single disk.

To explain the details of the model, we will break it down into two major components, the application model and the system model. Both of these have several subcomponents that will be described in this section. Table 1 summarizes the parameters of the application model, and Table 2 summarizes the parameters of the system model.

4.1 The Application Model

The first component of the application model is the database, which for simplicity is modeled as a collection of records (i.e., as one relation). The database initially contains $NumRecs$ records, and each record occupies $RecSize$ bytes. An unclustered primary key index (P), clustered secondary index (CS), and unclustered secondary index (US) exist on the data.[10] We assume that the cardinality of the domain of the key for index i is $DomainCard_i$, and that the index keys have a fixed size of $KeySize_i$ bytes. The actual key values are chosen from a uniform distribution, with duplicate keys allowed in all but the primary index.

The second component of the application model, the source module, models the external workload of the DBMS. Update transactions arrive in the system with rate $ArrivalRate_{update}$, while queries originate from a fixed set of MPL_{query} terminals. Each of the query terminals submits only one job at a time, and there is no delay between the completion of a query and the submission of the next query from the same terminal. For simplicity, queries execute a single relational select operation, while update transactions execute a single tuple insert, delete, or modify (i.e., select-update) operation.

The additional parameters which describe a read-only query include the access path, $AccessPath_{query}$ (i.e., the index to scan), and the average query selectivity, $AvgSel_{query}$ (i.e., the fraction of tuples that are selected by the scan). The actual selectivity for a query is chosen uniformly from 0.75 to 1.25 times $AvgSel_{query}$.

[10] In our view, primary keys do not usually have an interesting order (e.g., social security numbers, confirmation numbers, etc.), and are unlikely to be used to specify a range scan. Thus, we model a database that is clustered on a secondary key rather than the primary key.

Parameter	Meaning
$NumRecs$	number of records in the database
$RecSize$	number of bytes occupied by a record
$DomainCard_i$	cardinality of the domain of index i's key
$KeySize_i$	number of bytes occupied by index i's keys
$ArrivalRate_{update}$	update transaction arrival rate
MPL_{query}	number of query terminals
$AccessPath_{query}$	access path to be used by queries
$AvgSel_{query}$	average query selectivity
P_{modify}	probability that an update transaction will modify a tuple
P_{insert}	probability that an update transaction will insert a tuple
P_{delete}	$1 - P_{modify} - P_{insert}$
$ModifyProb_{indexed}$	probability that a tuple modification will affect the unclustered secondary index key

Table 1. Application Model Parameters

Parameter	Meaning
$IOCost$	service time of an I/O request
$PageSize$	number of bytes in a disk page
$CacheFrac$	fraction of each data page devoted to its on-page cache
$FillFactor$	initial data page fill factor
$CachePolicy$	on-page cache policy (WRITE-ALL or WRITE-ONE)
$PtrSize$	number of bytes to hold a disk address
$TIDSize$	number of bytes to hold a transaction identifier
$SlotOverhead$	number of bytes used for an object's slot table entry
$NumBuffers$	number of pages in the buffer pool

Table 2. System Model Parameters

For update transactions, the parameters P_{modify}, P_{insert}, and P_{delete} specify the distribution of update transactions among the three operation types. Each tuple in the database has an equal probability of being chosen for modification (or deletion) by a given modify (or delete) operation. The tuple to be modified or deleted is always indexed by its primary key value and located through the primary index. Modify operations change one attribute value at a time, either changing a secondary index key value (with probability $ModifyProb_{indexed}$) or a non-indexed attribute value (with probability $1 - ModifyProb_{indexed}$).

4.2 The System Model

The system model is designed to predict the I/O service time requirements of the various index operations in a workload specified by the application model parameters. The system model is broken down into these subcomponents: the transaction scheduler, the data manager, and the buffer manager. We describe these

components in the remainder of this subsection.

The transaction scheduler controls the timing of transaction execution. Because update transactions typically have stringent response time constraints, the scheduler gives them priority over queries. As discussed in the beginning of this section, update transactions are executed atomically, as are queries when they are initially descending an index for a scan. Upon arriving in the system, an update transaction is queued if it cannot be processed immediately. When the scheduler is invoked it examines the update transaction queue; if the queue is non-empty it removes the first update transaction and executes its index operation to completion. If the update transaction queue is empty the scheduler chooses to execute a portion of a query instead. For fairness, the query that was submitted from the query terminal that has consumed the least amount of disk service time thus far is chosen for execution; the scheduling policy thus allocates an equal fraction of the disk bandwidth among the query terminals. When a query is chosen for execution, it is executed until it has made at least one disk request and is beyond the initial descent phase of an index scan. Lastly, instead of modeling a disk arm in detail, we assume that each disk request requires *IOCost* milliseconds.

The model component known as the data manager encapsulates the implementation details of the indexes, data pages, and version pool. The following are the relevant parameters of the data manager. A fraction of each data page $CacheFrac$ is reserved for use as an on-page cache; the remaining portion is used for storing current versions, and is initially filled to a fraction $FillFactor$ of its total capacity. The cache replacement policy employed is *CachePolicy*. $PtrSize$ is the size of a disk pointer in bytes, and $TIDSize$ is the size of a transaction identifier in bytes. We assume that the usual slotted page organization [11] is used in the implementation, and *SlotOverhead* is the overhead in bytes for each slot table entry (i.e., for versions, VSTs, etc.). We further assume that the optional compression of TID fields described at the end of Section 3.1 is carried out whenever an index page overflows.

The last component of the system model, the buffer manager, encapsulates the details of an LRU buffer manager with "LOVE/HATE" hints (a la Starburst[16]). The number of page frames in the buffer pool is specified as $NumBuffers$, and the frames are shared among data, index, and version pool pages. Two LRU chains are maintained for unpinned pages in the buffer pool, one for pages that were last unpinned with a HATE hint and another for pages last unpinned with a LOVE hint. Because index pages have a higher rate of access than other pages, the index pages are unpinned with a LOVE hint, while data and

[11] Objects on a slotted page are always referenced through a small vector on the page (referred to as a slot table). In this way, objects can be easily moved within a page without having to update external references to the object.

version pool pages are unpinned with a HATE hint. The page replacement algorithm selects a page from the LOVE chain only if the HATE chain is empty. Dirty pages are cleaned when they are being replaced from the buffer pool.

4.3 Discussion of Model Assumptions

The model contains several simplifications that warrant further discussion. These include the absence of locking, the modeling of a single disk, and the lack of modeling of a CPU. The first two simplifications are reasonable since techniques for concurrency control and data placement across multiple disks are orthogonal to the indexing tradeoffs that we are studying here (i.e., those details are unlikely to change the relative ordering of the indexing schemes). The third simplification would be potentially problematic for predicting the performance of a CPU-bound configuration; however, we believe that the qualitative results would be similar since the CPU requirements of a given operation under each indexing scheme is roughly proportional to the number of disk pages accessed. Also, given the relative trends in CPU speeds and disk speeds, we expect I/O to be the bottleneck resource in future OLTP/decision support environments.

5 EXPERIMENTS AND RESULTS

In this section, we present the results of a series of experiments designed to compare the performance of MV2PL under the various indexing approaches described in Section 3. As a baseline for comparison, we also include the results obtained from GO processing using a single-version B+ tree index–this scheme is referred to as SV (for single version). Recall that with GO processing, queries are subject to inconsistent answers since they are run in a single-version database without obtaining locks. The primary performance metric employed in this study is the amount of I/O time required to execute update and query transactions. Update transactions issue either record insertion, deletion, and modification operations, while queries issue clustered or unclustered index scan operations.

The multiversion indexing schemes that we compare in this study differ primarily in where they place version selection information. As we described in Section 3, the indexing schemes place this information either with the data or with the index entries of one or more of the indices. The advantage of placing the version selection information in the indices is that it allows queries to directly access the versions that they need from a given index without having to go through one or more intermediate data pages for each version that has been migrated to the version pool. There are two potential drawbacks of this approach, however.

First, any attribute of a tuple is modified, even an unindexed one, each index that contains version selection information for the tuple must be updated. Second, the inclusion of version selection information in one or more of the indices will lead to larger indices. This in turn may cause the buffer hit rate to drop, as the buffer pool will be able to hold a smaller fraction of these pages.

In this study, we are interested in quantifying the I/O cost impact of the different approaches to the placement of version selection information under a range of operating conditions. In particular, we would like to determine the degree to which including version selection information in the indices reduces query I/O cost, and the degree to which it increases the I/O cost of modifying tuples. In addition, we would also like to determine the impact of having version selection information in the indices on the buffer hit rate since this impacts the I/O cost of all operations; thus, even though the basic page reads and writes involved in inserting or deleting a tuple do not differ from a single version B+ tree in any of the indexing schemes, the costs of these operations are relevant in this study since they will differ from scheme to scheme. Finally, we are interested in comparing the different approaches used in DP and CH to placing version selection information with the data. Tables 3 and 4 list the settings that we use for the application model and

Parameter	Value(s)
$NumRecs$	50,000
$RecSize$	208 bytes
$DomainCard_i$	2^{32} for primary key, 50,000 for secondary keys
$KeySize_i$	8 bytes
MPL_{query}	4
$AccessPath_{query}$	clustered or unclustered secondary index
$AvgSel_{query}$	varies (1% to 50%)
$P_{modify}/P_{insert}/P_{delete}$	60%/20%/20%
$ModifyProb_{indexed}$	20%
$ArrivalRate_{update}$	varies (4 per second to 26 per second)

Table 3. Values of Application Model Parameters

system model parameters, respectively. Some of the parameters remain fixed throughout the study, and others are varied from experiment to experiment. In our experiments, we vary the update arrival rate over a wide range to show how versioning influences I/O cost as the level of update intensity increases. As update intensity is increased, the current database state diverges more rapidly from the transaction-consistent prior states that must be maintained for the active queries. Furthermore, since update transactions compete for resources with queries, queries are left with a smaller share of the disk resources as the update intensity is

Parameter	Value(s)
$IOCost$	20 milliseconds
$NumBuffers$	400-1000 pages
$PageSize$	8192 bytes
$CacheFrac$	0% or 10%
$CachePolicy$	WRITE-ONE
$FillFactor$	80%

Table 4. Values of System Model Parameters

increased. Ultimately, the system will become unstable when it can no longer handle the increased update load.

We now turn to the results of our experiments. In the first experiment, we look at how the alternative schemes perform under a base set of parameter settings. In the subsequent experiments, we will vary some of the key parameters to explore their individual effects on performance.

5.1 Experiment 1: Basic Indexing Tradeoffs

Our base parameter settings include: no on-page caches, a buffer pool size of 500 pages, a query workload consisting of clustered index scans with 50and an update mix consisting of 60deletes. Most of the index pages can remain resident in the buffer pool with its size of 500 pages here; in subsequent experiments we will examine the impact of changing the buffer pool size. Figures 7 through 9 illustrate the results of this experiment. Figure 7 shows the average I/O cost of a clustered index scan, Figure 8 shows the I/O cost to insert a tuple into the database, and Figure 9 shows the I/O cost to modify a non-indexed attribute of a tuple (i.e., create a new version of a tuple with indexed attribute values that are the same as those of the previous version). We vary the update arrival rate between 4 per second and 26 per second along the x-axis in the graphs; since the multiversion indexing schemes are not able to handle the update load throughout this whole range, we truncate each scheme's curve at its last stable point. We do not show the cost of deleting a tuple from the database, nor do we show the cost of modifying an indexed attribute, as in both cases the relative cost results were similar to those for insertions.

In Figure 7, we see that the query I/O cost rises gradually with an increase in the update arrival rate under SV, while it rises much more quickly under the four multiversion indexing schemes. The gradual rise in the case of SV is caused by an increase in the fraction of dirty pages in the buffer pool as the update

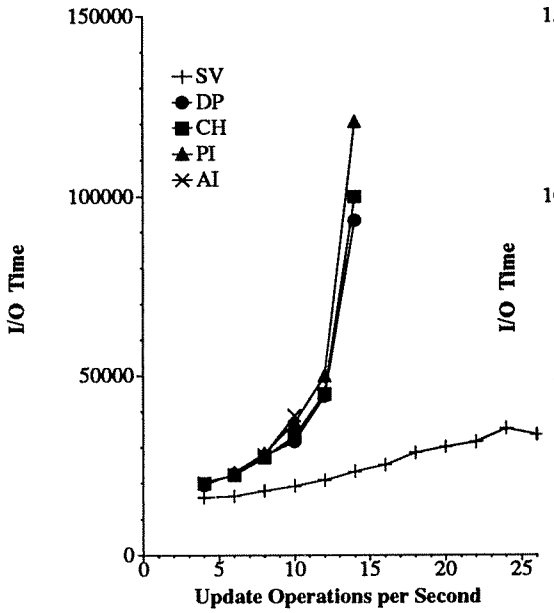

Fig. 7. Clustered Index Scan

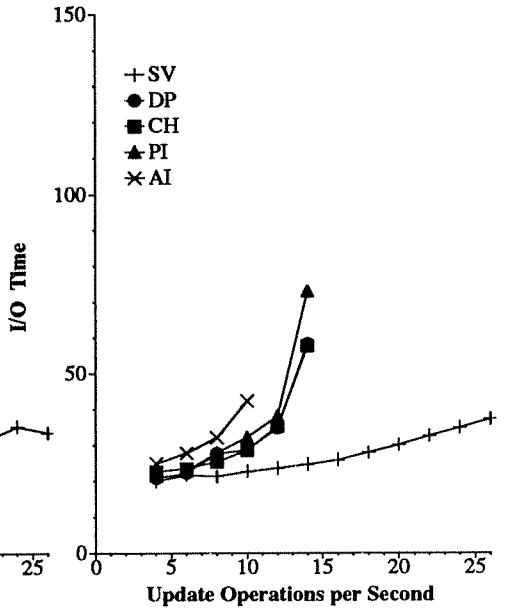

Fig. 8. Tuple Insert

$NumBuffers = 500, CacheFrac = 0\%$

activity is increased (and as query activity is correspondingly decreased). [12] Although the multiversion indexing schemes are also influenced by this factor, the rapid rise in their query I/O costs is primarily due to a corresponding rise in the number of version pool accesses. These version pool accesses quickly dominate the clustered index scan I/O cost since versions are accessed randomly in the version pool (versus sequentially in the main segment). To select 25,000 records clustered on about 850 data pages under the DP scheme, for example, the average number of version pool I/Os per query was approximately 54 at 4 UPS, rising to 280 at 10 UPS, and 1878 at 14 UPS.

The query I/O costs for the different multiversion indexing schemes did not differ much under this workload; however, AI was not able to operate above 10 UPS due to its higher update cost (which we will discuss shortly). One notable difference between the remaining schemes is that PI rises in cost relative to DP and CH at an update arrival rate of about 14 UPS. This increase is primarily a result of having to retrieve each individual tuple through the primary index (rather than being able to go directly to the data from the clustered secondary index). This extra step does not add additional I/O cost at lower arrival rates because the entire primary index is resident in the buffer pool; at higher update rates, however, the

[12] At an arrival rate of 4 UPS (update operations per second), only about 8of the pages that were replaced by query-requested pages in the buffer pool had to be cleaned before the query could be given the page frame, while at 26 UPS nearly 50

primary index no longer fits in the buffer pool, as it contains entries for the larger number of transient versions resulting from the higher update rate.

In this experiment there is not a significant difference between DP and CH in terms of the query I/O cost. This is because CH rarely required more than one version pool access to retrieve a particular version (i.e., it very rarely had to retrieve a version that was not either the current version or the next most recent version of a tuple). For example, at 14 UPS, where a slight difference is visible between the two schemes in Figure 7, CH required a second version pool access to retrieve a given tuple less than 1Likewise, there is not much of a difference between AI and the other multiversion indexing schemes over the range of update arrival rates where AI is able to operate. On the surface this struck us as somewhat surprising, as the AI scheme is supposed to help query performance by providing version addresses directly in the leaf pages of the indices; this allows a query to read a version in the version pool without having to first read the data page that it originated from. However, since the data pages of the relation are accessed sequentially in the case of a clustered index scan, eliminating a few of what would have been repeated accesses to each data page will not reduce the number of I/O operations. Thus, the AI scheme does not really help in the case of clustered index scans. For essentially the same reason, having version addresses directly in the leaf pages of the primary index in the PI scheme does not help here either.

We now turn our attention to the I/O cost for updates, beginning with the cost of insert operations shown in Figure 8. The insert cost differences in this figure are largely due to variations in the index sizes from scheme to scheme, which arise from their different polices on where version selection information is located. The connection between I/O cost and index size is that larger indices permit a small fraction of the index pages to be cached in the buffer pool; thus, the buffer hit rate decreases as index size is increased. AI, which has the highest insert cost in Figure 8, includes VSTs in all of its indices; PI, which has the next highest cost, includes them only in the primary index; DP and CH, which have the lowest insert cost of the multiversion indexing schemes, do not include VSTs in any of the indices. Lastly, SV has the overall lowest cost since its indices only store current versions (and thus have no timestamps either). Now we turn to the cost of modifying a non-indexed attribute, shown in Figure 9. AI has the highest cost for this operation since it has to install the address of a new tuple version in all of the indices. Moreover, AI's cost for this operation will increase relative to the other indexing schemes as the number of indices on each relation grows. PI must install the address of a new version in the primary index; however, the relevant leaf page will already be pinned in the buffer pool since the primary index is used to locate the tuple being modified.

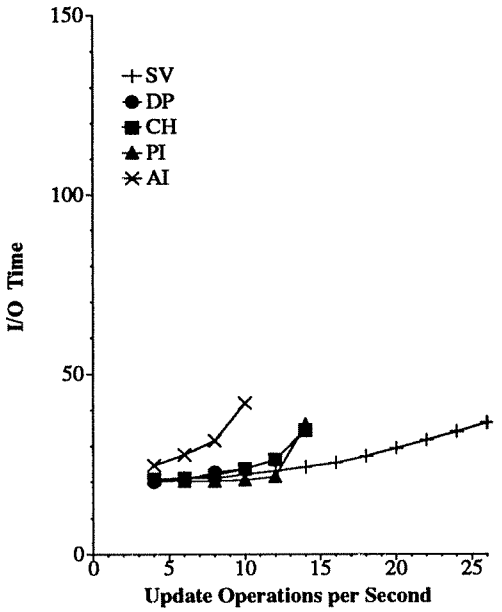

Fig. 9. Modify Non-Index Attr
$CacheFrac = 0\%$

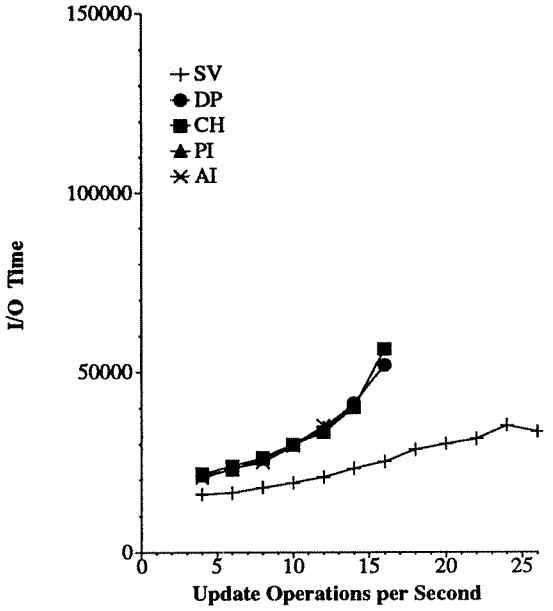

Fig. 10. Clustered Index Scan
$CacheFrac = 10\%$
$NumBuffers = 500$

[13] DP, CH, and SV do not need to update any indices for this operation.

DP, CH, and PI have I/O costs which are closer to SV in Figure 9 than they were in the insert case; this is because the costs of all four of these schemes are now dominated by the cost of accessing the target data page. The hit rate for data pages does not vary much between the indexing schemes since the overall number of data pages is large; however, the costs of the multiversion indexing schemes in Figure 9 do rise somewhat at an update arrival rate of 14 UPS due to a decreased buffer hit rate on the primary index (which is used to locate the tuples that are updated). As we mentioned previously, an increase in the update rate leads to a decrease in buffer hits for index pages because the indices become larger.

5.2 Experiment 2: Effect of On-Page Caching

In this experiment we examine the effect on the results presented so far of introducing on-page version caches that comprise 10scan queries with on-page caching. By comparing this figure to Figure 7, we can see the benefit of the on-page caches on query performance. The presence of the caches reduces the number of read operations in the version pool, and thus forestalls thrashing behavior.

[13] PI's cost dips slightly below that of SV in Figure 9 because the query retrievals though the primary index in PI cause a larger fraction of its primary index pages to remain resident in the buffer pool.

Due to space limitations, we do not show the corresponding update costs here, but they can be summarized as follows: The update costs for all of the multiversion indexing schemes are slightly lower with on-page caches, as queries had lower I/O costs, and thus complete faster. When queries complete faster, fewer transient versions must be indexed, resulting in turn in a higher buffer hit rate on index pages. A drawback of on-page caching with the PI and AI schemes is the potential for increase in the cost of migrat-

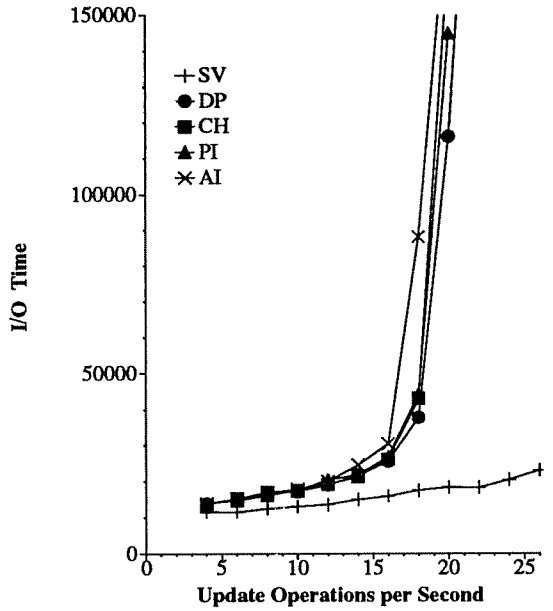

Fig. 11. Clustered Index Scan

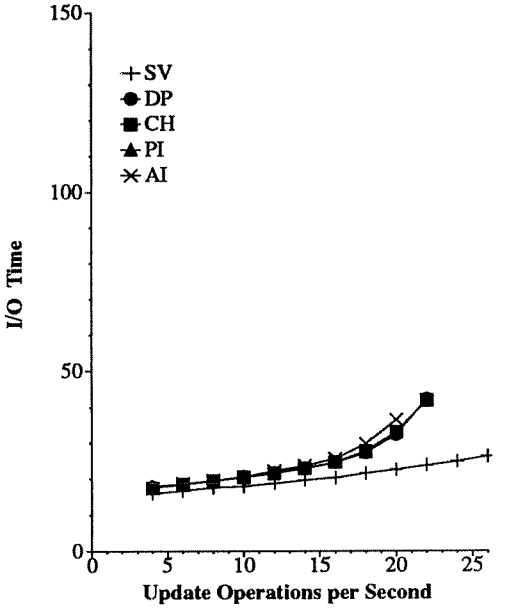

Fig. 12. Tuple Insert

$CacheFrac = 0\%$, $NumBuffers = 800$

ing a version to the version pool, as any index references to a version being migrated must be updated. In this experiment, however, the benefits of on-page caching outweighed this additional cost.

5.3 Experiment 3: Effect of Buffer Pool Size

In this experiment, we increase $NumBuffers$ to 800 so that we may examine the effect of a larger buffer pool size on the results obtained in Experiment 1. In Figure 11, we show the average I/O cost of clustered index scan queries, and in Figure 12, we show the average I/O cost to insert a tuple into the database. We omit the costs of the other update operations since they are very similar to the insert cost here. Comparing Figure 12 to Figure 8, we see that the additional buffers significantly reduce the cost of insertions. They also reduce the I/O cost differences between the various indexing schemes since the indices in all of the

indexing schemes fit entirely, or almost entirely, in the buffer pool. When the indices fit entirely in the buffer pool, the differences in index size between the schemes affect only the main segment and version pool hit rates; this is a secondary factor since the overall number of data pages and version pool pages is relatively large.

Returning to Figure 11, we see that the increase in buffer pool size reduces the query I/O cost as well. To some degree this is a result of increased buffer hits; however, it is primarily due to an increase in the fraction of disk resources available to queries as a result of the lower update cost. With additional disk resources, queries are able to make more progress in between the arrival of consecutive updates, and fewer (expensive) version pool accesses are ultimately necessary for each query. Beyond approximately 16 UPS, however, all of the multiversion schemes still begin to thrash as a result of excessive version pool accesses. It is only at this thrashing stage that we see significant differences in the I/O costs of the different multiversion indexing schemes. (It is unlikely, however, that this would be an acceptable operating region since the query costs are very high in all of the schemes; thus the differences there have limited significance).

To determine the robustness of these results, we also ran a set of simulations with only 400 pages allocated to the buffer pool. Due to space limitations, we do not present those additional results here. Briefly, those results showed that, as anticipated, the effects of reducing the buffer pool size mirror the effects of increasing its size. In particular, doing so increased the cost of both the update and query scan operations in all of the indexing schemes, and it accentuated the cost differences between the schemes. Among the four alternatives, DP and CH delivered the best performance, while PI and AI delivered lower performance.

5.4 Experiment 4: Unclustered Index Scan Queries

In the previous experiments, we explored tradeoffs between the indexing schemes under a query workload consisting of clustered index scans. As we pointed out, the benefits of storing version selection information with the indices (rather than with the data) are not significant for clustered index scans. To determine the regions where AI and PI might excel, we now consider a query workload consisting of unclustered index scans.

In designing this experiment, we explored a wide range of parameter values. We found that PI and AI outperform DP and CH when three conditions are simultaneously satisfied: queries are relatively long-running, the update rate is sufficiently high, and the buffer pool is sufficiently large. Having a high update rate and long-running queries is necessary since version selection information in the indices helps

only if queries end up accessing a significant number of versions that have migrated to the version pool. Furthermore, the buffer pool must be large enough so that the version selection information added to the

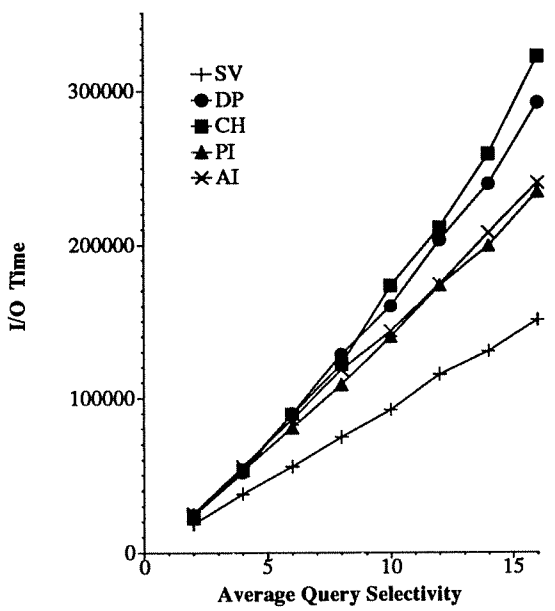

Fig. 13. Unclustered Index Scan

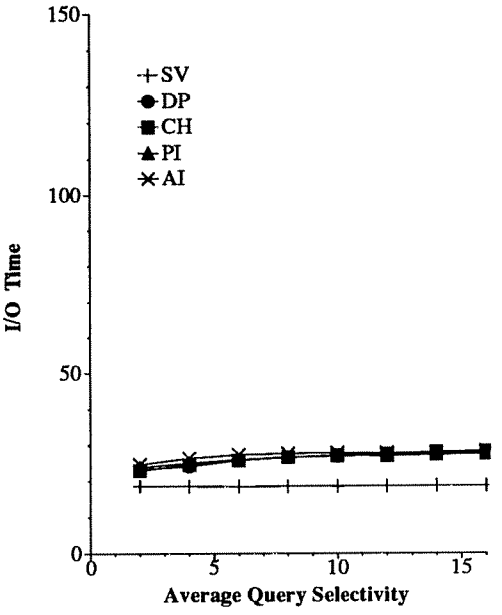

Fig. 14. Tuple Insert

$ArrivalRate = 20/sec.$ $CacheFrac = 0\%$, $NumBuffers = 1000$

indices in PI and AI does not cause the index page buffer hit rate to degrade; otherwise, query performance is seriously impaired because the update transactions require a larger share of the disk resources.

Figures 13 and 14 illustrate a situation where PI and AI indeed outperform the other multiversion indexing schemes. In contrast to the previous figures, we fix the update arrival rate at 20 per second here, and we show I/O cost as a function of the average query selectivity. We vary the selectivity up to 16pays to switch to a full file scan for selectivities above a few percent. We do this to model a situation where a long-running query accesses data through a secondary index (e.g., this might arise in a real application if a long-running query issues multiple SQL statements, the last of which generates an unclustered index scan). Lastly, we use a buffer pool size of 1000 pages here. Figure 13 shows the average I/O cost of an unclustered index scans, and Figure 14 shows the average I/O cost to insert a tuple in the database.

In Figure 13, the unclustered index scan cost rises as expected as the query selectivity is increased. Among the multiversion indexing schemes, DP and CH now have the highest scan costs since they must first access a data page before being able to retrieve a tuple version that has been migrated to the version

pool.

In Figure 14, we show the corresponding insert cost. We do not show the costs of the other update operations since they were again similar to the insert case. The curves in Figure 14 are flat and close together since the large buffer pool is able to keep the index pages cached. As we pointed out in Experiment 3, when the buffer pool is made sufficiently large, the I/O cost of updates is limited mainly to the cost of accessing data pages, and the data page hit rates do not vary much between the schemes.

5.5 Discussion

In this section, we have presented the results of four experiments comparing the query and update transaction I/O costs of the alternative indexing schemes. One goal of these experiments was to determine the conditions under which placing version selection information in the indices reduces query I/O cost. On the positive side, such information can be used by a query to directly access a tuple version from its leaf index entry without having to go through one or more intermediate pages. However, placing the version selection information in the indices causes them to grow larger. Moreover, it became apparent from our experiments that increasing the size of the indices can have a large negative impact on update transaction I/O cost since it reduces the buffer hit rate on index pages.

In the first three experiments, queries executed clustered index scans. As we explained in Section 5.1, having version selection information in the indices cannot benefit clustered index scans, so DP and CH were superior to AI and PI in these experiments. DP showed a somewhat lower query I/O cost than CH; however, this was seen only when queries began to thrash. In the fourth experiment, queries executed unclustered index scans. Our results showed that unless the buffer pool is large enough to hold all of the index pages, DP and CH still outperform PI and AI. Only when the buffer pool is large enough to absorb the additional version selection information in the indices, and when queries are sufficiently long-running to benefit from this information (i.e., they require many prior versions), do PI and AI exhibit a lower unclustered scan I/O cost than DP and CH.

In the experiments that were covered in this section, we did not vary parameters such as index key size or the relative mix of update operations (i.e., P_{insert}, P_{delete}, and P_{modify}). We have run some additional experiments that varied these parameters, but they did not reveal any significant changes in the qualitative results.

6 CONCLUSIONS

In this paper, we have compared four basic schemes for extending single-version indexing structures to handle multiversion data. Although B+ trees were used to illustrate the schemes, they can all be combined with any existing database index structure. The resulting multiversion indexing schemes differ in where version selection information is located. In the AI scheme, version selection information is placed in all of the indices, whereas in the PI approach, the information is placed only in the primary index. In contrast, the DP and CH approaches place version selection information with the data instead. DP and CH differ in that DP maintains a table to locate all of the versions of a tuple, while CH chains the versions in reverse chronological order.

We conducted a simulation study of the alternative multiversion indexing schemes, and we analyzed the results of this study. Despite having the advantage of direct references from index entries to individual versions, we found that the PI and AI schemes have the same or higher I/O costs for queries when the buffer pool is not large enough to hold all of the index pages. This is because the version selection information in the index entries consumes critical buffer pool space, thus lowering the buffer pool hit rate. As a result, the I/O cost for update transactions under PI and AI is higher than DP and CH under these conditions as well. Only when the buffer pool is large enough to hold all of the index pages do the benefits of placing version selection information in the indices begin to appear in terms of lower query I/O cost; however, these benefits apply only to unclustered index scan queries. Lastly, we saw that the I/O cost for queries under DP was somewhat lower than under CH, but only when queries began to thrash. These results indicate that DP is the version indexing approach of choice, with CH being a close second; PI and AI are not recommended due to their relatively poor performance under most conditions.

References

1. Agrawal, D., A. Bernstein, P. Gupta and S. Sengupta, "Distributed Multiversion Optimistic Concurrency Control with Reduced Rollback," *Journal of Distributed Computing*, Springer-Verlag, 2(1), January 1987.
2. Agrawal, D. and S. Sengupta, "Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control," *Proc. 1989 SIGMOD Conference*, 1989.
3. Bayer, R. and M. Schkolnick, "Concurrency of Operations on B-trees," *Acta Informatica*, September 1977.
4. Bayer, R. and E. M. McCreight, "Organization and Maintenance of Large Ordered Indicies," *Acta Informatica*, Volume 1, Number 3, 1972.
5. Bayer, et al., "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Sys.*, 5(2), June 1980.
6. Bober, P. and M. Carey, "On Mixing Queries and Transactions via Multiversion Locking," *Proc. of the Eighth IEEE Data Engineering Conf.*, 1992.

7. Bober, P. and M. Carey, "Multiversion Query Locking," *Proc. of the Eighteenth International Conference on Very Large Databases*, 1992.
8. Bober, P. and M. Carey, *Indexing Alternatives for Multiversion Locking,* Tech. Report #1184, University of Wisconsin—Madison, November 1993.
9. Carey, M. J., *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. Thesis, Comp. Sci., U. of California, Berkeley, 1983.
10. Chan, A., S. Fox, W. Lin, A. Nori, and Ries, D., "The Implementation of an Integrated Concurrency Control and Recovery Scheme," *Proc. 1982 ACM SIGMOD Conf.*, 1982.
11. Chan, A., and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Trans. on Software Eng.*, SE-11(2), Feb 1985.
12. DuBourdieu, D., "Implementation of Distributed Transactions," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1982.
13. Easton, M., "Key-Sequence Data Sets on Indelible Storage," *IBM Journal of Research and Development*, May 1986.
14. Eswaran, K., J. Gray, R. Lorie, I. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *CACM* 19(11), 1976.
15. Gray, J., "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979.
16. Haas, L., "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.
17. Joshi, Ashok, *Personal Communication.*
18. Kolovson, C. and M. Stonebraker, "Indexing Techniques for Multiversion Databases," *Proc. of the Fifth IEEE Int'l Conf. on Data Engineering*, 1989.
19. Lai, M. and K. Wilkinson, "Distributed Transaction Management in Jasmin," *Proc. of 10th International Conference on Very Large Database Systems*, 1984.
20. Lomet, D. and B. Salzberg, "Access Methods for Multiversion Data," *Proc. 1989 ACM SIGMOD Conf.*, 1989.
21. Lehman, P. and S. Yao, "Efficient Locking for Concurrent Operations on B-trees," *ACM Transactions on Database Systems*, 6(4), December 1981.
22. Mohan, C., H. Pirahesh, and R. Lorie, "Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions," *Proc. 1992 ACM SIGMOD Conf.*, 1992.
23. Raghavan, A., and T.K. Rengarajan, "Database Availability for Transaction Processing," *Digital Technical Journal* 3(1), Winter 1991.
24. Reed, D., "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems*, 1(1), February 1983.
25. Robinson, J., *Design of Concurrency Controls for Transaction Processing Systems,* Ph.D. Thesis, Comp. Sci. Tech. Rep. No. CMU-CS-82-114, 1982.
26. Silberschatz, A. "A Multi-Version Concurrency Control Scheme With No Rollbacks," *ACM-SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1982.
27. Stearns, R. and D. Rosenkrantz, "Distributed Database Concurrency Control Using Before-Values," *Proc. of the 1981 ACM SIGMOD Conf.*, 1981.
28. Stonebraker, M., "The Design of the Postgres Storage System," *Proc. Thirteenth International Conference on Very Large Database Systems*, 1987.