ON MIXING QUERIES AND TRANSACTIONS VIA MULTIVERSION LOCKING

by

Paul M. Bober and Michael J. Carey

Computer Sciences Technical Report #1045

November 1991

On Mixing Queries and Transactions via Multiversion Locking

Paul M. Bober
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by an IBM Research Initiation Grant and by the National Science Foundation under grant IRI-8657323.

A slightly abridged version of this paper will appear in the proceedings of the *IEEE Eighth International Conference on Data Engineering*, February 1992.

On Mixing Queries and Transactions via Multiversion Locking

Paul M. Bober
Michael J. Carey
Computer Sciences Department
University of Wisconsin

ABSTRACT

In this paper, we discuss a new approach to multiversion concurrency control that allows high-performance transaction systems to support the on-line execution of long-running queries (e.g., for decision support purposes). Long-running queries are typically run at level 1 or level 2 consistency because they introduce a high level of data contention with two-phase locking. Multiversion algorithms have been discussed as a way to reduce the level of data contention and at the same time support the serializable execution of queries. Our approach extends the multiversion locking algorithm developed by Computer Corporation of America by using record-level versioning and reserving a portion of each data page for *caching* prior versions that are potentially needed for the serializable execution of queries; on-page caching also enables an efficient approach to *garbage collection* of old versions. In addition, we introduce *view sharing*, which has the potential for reducing the cost of versioning by grouping together queries to run against the same transaction-consistent view of the database. Finally, we also present results from a simulation study that compares our approach versus that of providing level 1 or level 2 consistency for queries. The results indicate that our approach is a viable alternative to reduced-consistency locking when the portion of each data reserved for prior versions is chosen appropriately.

1. INTRODUCTION

On-line transaction processing (OLTP) systems are required to deliver high transaction throughputs to support business-critical applications. At the same time, business needs often demand timely ad-hoc query access to data in these systems. Unfortunately, the introduction of ad-hoc queries can cause interference with OLTP transactions in the form of high data contention. In order to reduce the level of data contention and thus maintain reasonable OLTP throughput, users are often forced to make compromises in either the consistency or timeliness of queried data. One compromise is to violate serializability and run ad-hoc queries at level 2 or level 1 consistency [Gray76]. With level 2 or level 1 consistency, a query will be subject to non-repeatable reads, and with level 1 consistency, a query will be permitted to see dirty data as well. A different compromise is to maintain two separate databases, one for OLTP transactions and another for ad-hoc queries [Pira90]. In this approach, the OLTP database is periodically extracted and copied to the ad-hoc query system. Disadvantages of this approach are that the disk storage requirements are doubled, the additional cost of copying the database periodically is incurred, and queries must run against a database that can be many hours or even days old.

A better solution is necessary for users that require an ad-hoc query facility capable of providing timely access to data, serializability, and minimal interference with OLTP transactions. It has been proposed that *transient versioning* concurrency control algorithms [Pira90] might satisfy these requirements. Examples of such algorithms include those discussed in [Robi82, Chan82, Bern83, Reed83, Care86, Weih87, Agra87, Agra89, Son90]. Transient versioning algorithms are also used in a few commercial database systems, such as DEC's Rdb/VMS product [Ragh91]. In transient versioning systems, prior versions of data are retained to allow queries to see slightly outdated but transaction-consistent database snapshots. Because queries view a prior snapshot of the database, they serialize before all concurrent update transactions and (in some sense) do not have to follow any concurrency control protocol in most such proposals. Several related algorithms that retain at most two versions of data items in order to reduce blocking due to read/write conflicts have also been proposed [Baye80, Stea81]. Because queries can still block OLTP transactions, however, such *two-version* algorithms do not meet our goal of minimal interference; we will therefore not consider two-version algorithms further here.

The logical snapshots taken by transient versioning algorithms are generated using a mechanism that is analogous to copy-on-write, allowing many logical snapshots to be maintained simultaneously with a storage and management cost that is potentially lower than that of extracting entire database snapshots. Also, because many logical snapshots can be maintained simultaneously, each query can be given a view of the database that is seconds or minutes old rather than hours or days old. (It should be pointed out that transient versioning systems maintain prior versions only for concurrency control purposes, not for answering temporal queries.)

1.1. Multiversion Two-Phase Locking (MV2PL)

Computer Corporation of America (CCA) designed and implemented a transient versioning variant of two-phase locking (MV2PL) in the context of their LDM and DDM data manager prototypes [Chan82, Chan85]. 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 data item, it locks the item, as in traditional 2PL, and then accesses the most recent version. Update transactions must block when lock conflicts occur. When an item is written, a new version is

created and stamped with the commit timestamp of its creator[1]. When a read-only query Q wishes to access an item, on the other hand, it simply reads the most recent version of the item 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.

To maintain the set of versions needed by ongoing read-only transactions, MV2PL divides the stored database into two parts: the main segment and the version pool. The main segment contains the current version of every object in the database, and the version pool contains prior versions of objects. The version pool is organized as a circular buffer, much like the log in a traditional recovery manager [Gray79, Gray81]. Object versions are chained in reverse chronological order, and version pool space is allocated sequentially. Two attractive properties of this approach to version management, as compared to other proposals, are that (i) updates are performed in-place, allowing clustering to be maintained, and (ii) version pool writes are sequential (i.e., similar to log writes). Figure 1.1 depicts the main segment of the database, the version pool, the pointers used to manage version pool space, and a version chain for an object X. Version pool entries between *reader-first* and *last* in the figure contain versions that may be needed to satisfy read requests for ongoing read-only transactions. Entries between *update-first* and *last* contain object versions recorded by ongoing (or recently committed) update transactions; this range can also serve as a transaction UNDO log [Chan82].

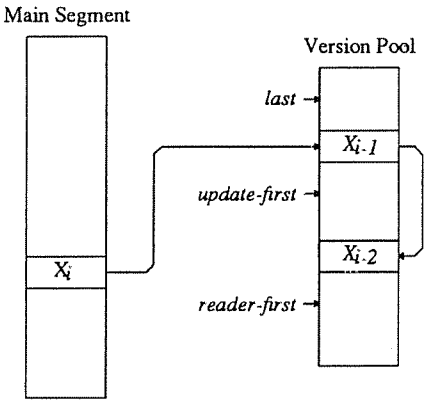

Figure 1.1: MV2PL Storage Organization

[1]Actually, to reduce commit-time processing in the absence of a no-steal buffer management policy, the item is stamped with the creator's transaction id and a separately maintained list is used to map from transaction ids to commit timestamps [Chan82].

2. A NEW MV2PL DESIGN

The CCA MV2PL design [Chan82] employs page-level versioning and locking, requires a page's before-image to be copied from the main segment into the version pool before it can be updated, and requires updated pages to be forced to disk at commit time — three strikes as compared to the high-performance, log-based update schemes used in competitive OLTP systems. In order to make MV2PL useful for transaction processing we make several refinements here. First, to remove the force requirement, we separate recovery from versioning by using the traditional write-ahead logging (WAL) approach to crash recovery [Gray79, Moha89]. Second, we make the (straightforward) conversion from page-level to record-level versioning and locking. These two refinements are used in DEC's Rdb/VMS system as well [Ragh91]. Third, we use on-page *caches* for prior versions in order to reduce I/O activity to the version pool. Last, we introduce the technique of *view sharing* which is used to reduce the number of snapshots that must be maintained by the system. The remainder of this section expands on the last two of these refinements.

2.1. On-Page Caching

One of the advantages of moving to a record-level versioning scheme is that it allows us to allocate a portion of each data page for caching prior versions. Such an on-page cache reduces the number of read operations required for accessing prior versions. In addition, versions may "die" (i.e., no longer be needed for maintaining the view of a current query) while still in a page cache and thus not have to be appended to the version pool at all. Figure 2.1 shows a data page with records *X*, *Y* and *Z* and a cache size of 3. All 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 record is updated, the current version is copied into the cache before it is replaced by the new version. If the cache is already full, *garbage collection* is attempted on the page. Garbage collection attempts to find prior versions in the cache that are no longer needed to satisfy the request of any current query (i.e., that are not 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 is appended to the version pool. The *write-one* policy appends a version when it is chosen to be replaced in the cache.

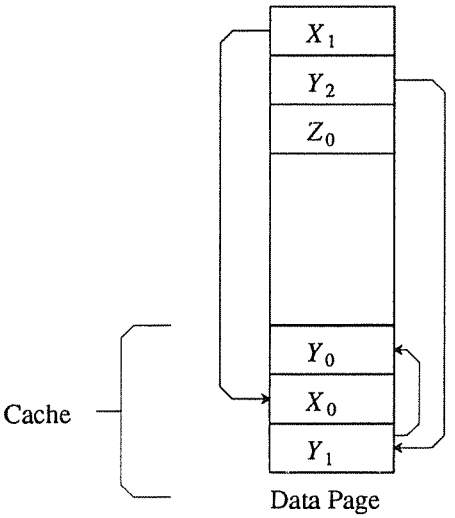

Figure 2.1: A data page with a cache size of 3

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 and the least recently updated entry has not yet been appended to the version pool. The write-all policy thus attempts to cluster entries from the same data page together in the version pool.

Figure 2.2 shows the write-one policy being used when an update to record Z on the page from Figure 2.1 causes a cache overflow. Record version Y_0 is found to be the least recently updated cache entry and is appended to the version pool. Figure 2.3 shows how the same situation is handled with the write-all policy. In this case, the entire cache is written to the version pool, but again only Y_0 is actually replaced. Notice that the next two entries to be replaced from the cache, X_0 and Y_1, have special pointers (represented by dashed arrows) into the version pool; these pointers are used to locate version pool copies of the cached entries. When X_0 is later replaced in the cache, the pointer stored in its cache slot, represented by the dashed arrow, will be used to simply redirect X_1's next pointer to the appropriate position in the version pool; thus, no version pool write will be needed for the later replacement of X.

2.2. Garbage Collection

Prior versions are no longer necessary once they become inaccessible by any currently executing query. In the version pool, space is reclaimed sequentially when the oldest query finishes, allowing the *reader-first* pointer to move

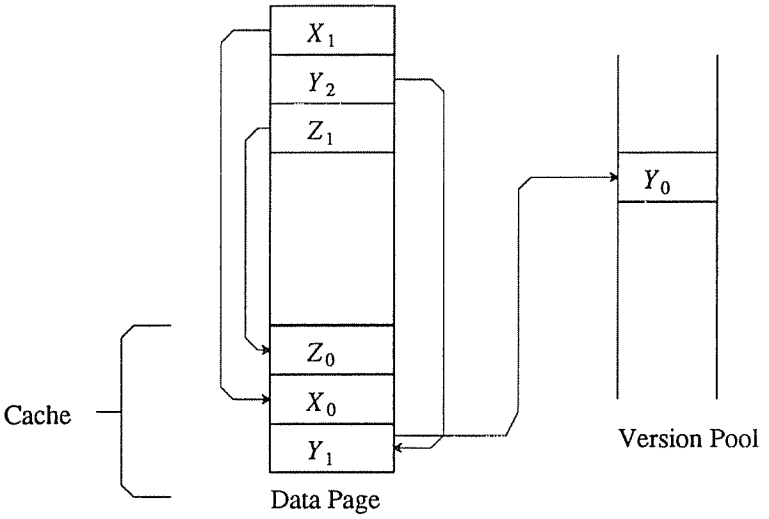

Figure 2.2: Write-one policy

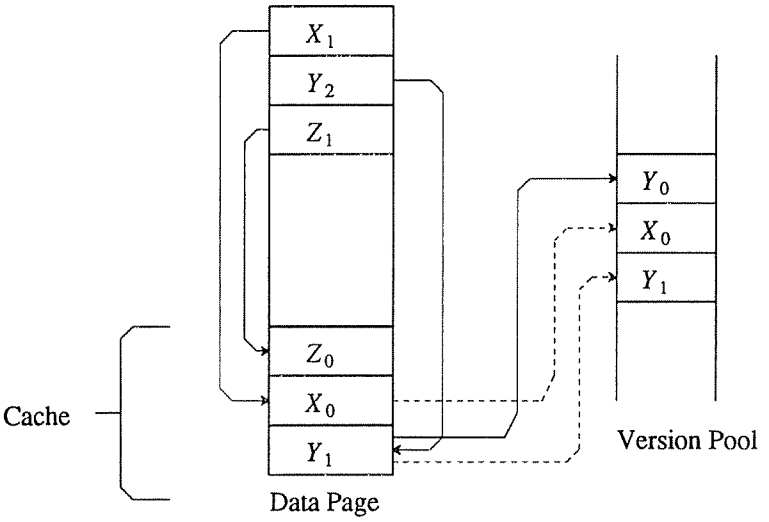

Figure 2.3: Write-all policy

[Chan82]. Because of the nature of this sequential deallocation process, versions may become unnecessary before they can be garbage-collected. One resulting problem is that a very long-running query may hold up the reclamation of version pool space that is occupied by versions other than those that are in its view.

In contrast to versions in the version pool, versions that reside in an on-page cache may be garbage-collected soon after they become unnecessary. 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 or not it is still needed. Note that this

method of garbage collection is quite inexpensive, as the page must already be resident in the buffer pool and will be dirtied anyway by the update that initiated it. Figure 2.4 shows an example of a page that is about to have garbage collection run on it. The numbers in parentheses are the commit timestamps of the transactions that wrote each version. There are two queries executing in the system, Q1 (with startup timestamp 100) and Q2 (with startup timestamp 200). Y_0 is needed to satisfy a potential request by Q1 and must remain in the cache. Y_1 and X_0 are not needed for either Q1 or Q2, however, so their space may be reclaimed.

Note that garbage collection does not have to be done in chronological order to slots in the on-page version cache. For example, Y_1 can be deleted even though the page contains older versions (of either the same or a different record) that are still needed (e.g., Y_0). In contrast, if Y_1 resided in the version pool instead of in an on-page cache, its space could not be reclaimed until every version that was previously written to the version pool also became unnecessary. The use of an on-page cache can therefore mitigate to some extent the large transient storage requirements caused by long queries. In particular, while versions that are needed for maintaining the view of a very long query will still migrate to the version pool, versions needed only for shorter queries will (hopefully) remain in an on-page cache and be garbage-collected there when no longer needed. In comparison, in CCA's original version pool-only architecture, a very long query will hold up all garbage collection from the time when it becomes the oldest running query until the time when it finishes.

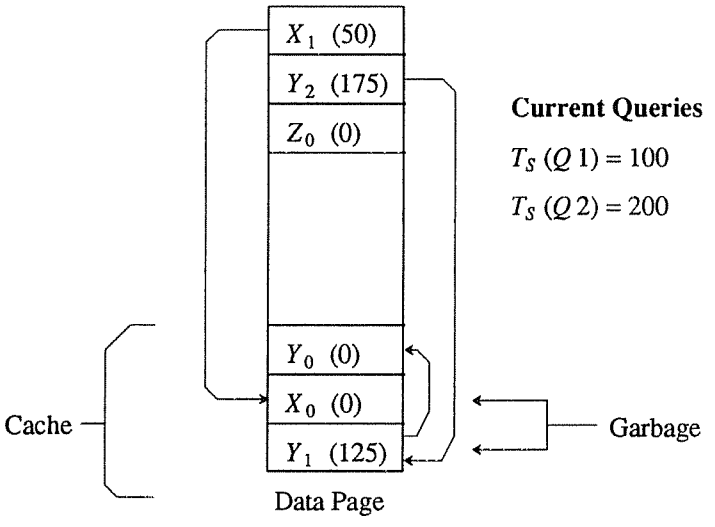

Figure 2.4: On-page garbage collection

2.3. View Sharing

If several queries are grouped together and executed with the same startup timestamp, fewer logical transaction-consistent views of the database must be maintained and thus potentially fewer versions must be retained. To understand why, observe that when an update to a record occurs, the prior version of the record is unnecessary if no currently executing query has a startup timestamp in between the timestamps of the prior and current versions of the record. Once the transaction generating the new version commits, and thus its timestamp is known, the prior version may be garbage collected if it has remained in the cache.[2] We can increase the likelihood that this will occur by grouping queries together and allowing them to share the same startup timestamp. In this way, a query may elect to run "back in time" by reusing the startup timestamp of the youngest currently executing query and thus share its view of the database.[3]

Figure 2.5 illustrates how view sharing works. In the figure, a query Q_2 enters the system when the commit timestamp counter is equal to t_d. This query may either elect to share the logical view of query Q_1 and use the same startup timestamp, t_b, or it may decide to generate a new logical view and run with a startup timestamp of t_d. After the transaction begins, a transaction that has updated X (generating X_2) subsequently commits with a timestamp of t_e. If Q_2 decides to share Q_1's startup timestamp, then the previous version of X (X_1) will become unnecessary at this point and may be garbage-collected the next time that its associated on-page cache overflows. On the other hand, the version that would have otherwise become unnecessary when Q_1 finished (X_0) will now still be necessary until both Q_1 and Q_2 are finished. Hence, the tradeoff of view sharing is that fewer versions are likely to be necessary beyond the commit of the transaction that overwrites them, but versions that are retained are likely to be retained for a longer period of time.

3. THE SIMULATION MODEL

In this section, we describe the model that we used to evaluate the performance of our record-level MV2PL design. The model was implemented as a collection of modules in the DeNet simulation language [Livn89]. The simulator was derived from a single-site configuration of a simulator constructed for the Gamma parallel database system [DeWi90] and used in studies of replication strategies [Hsai90] and complex query processing [Schn90]. We used this simulator as a starting point primarily to facilitate subsequent research on MV2PL extended for use in a parallel DBMS environment.

[2] We could discard the prior version when the update occurs (i.e., without waiting for the updating transaction to commit) if we knew that no query will enter the system with a startup timestamp that is less than the commit timestamp of the updating transaction. This could be accomplished by requiring that a new query register itself and wait for all currently executing updaters to finish before entering the system. While the query is waiting, subsequently arriving update transactions would not be allowed to discard their prior versions before commit.

[3] A similar idea, developed independently, is presented in [Wu91].

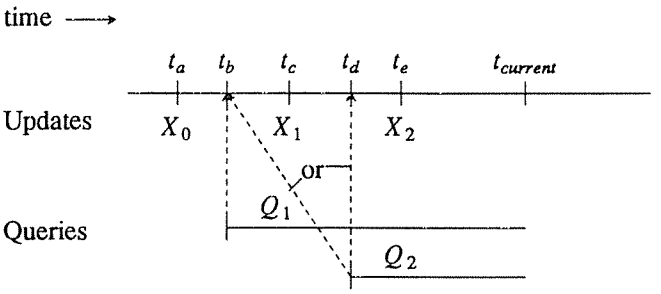

Figure 2.5: View Sharing

In addition, the basic Gamma simulator was validated against the actual Gamma implementation [Schn90, Hsai90], so we knew that we were at least starting from something that modeled reality fairly accurately.

In order to explain the model, we will break it down into two major components, the application model and the system model. Each of these have several subcomponents that we will describe in this section.

3.1. The Application Model

The first component of the application model is the database, which is modeled as a collection of *relations*. Each relation, in turn, is modeled as a collection of records. One clustered and one unclustered index exist on each relation. We assume that there are *NumKeys* keys per index page and (for simplicity) that there is a one-to-one relationship between key values and records. Index entries reference the head of each record version chain; the key value matches *some* version in the chain. We assume that each relation has *RelSize* records and that each record occupies *RecSize* bytes. The data is physically organized as a series of <relation, clustered index, unclustered index> triples that are laid out on the disk in cylinder order. The version pool is placed on the disk following all of the primary data. The parameters for this portion of the overall model are summarized in Table 3.1.

Parameter	Meaning
NumFiles	Number of files in database
$RelSize_i$	Number of records in file *i*
$RecSize_i$	Size of records in file *i*
$NumKeys_i$	Number of keys per index page in file *i*

Table 3.1: Database Model Parameters

The second component of the application model, the source module, is responsible for modeling the external workload of the DBMS. Table 3.2 summarizes the key parameters of the workload model. The system is modeled as a closed queueing system with the transaction workload originating from a fixed set of terminals. Each terminal submits only one job at a time and is dedicated to either the *Update* transaction class or the *Query* transaction class. *Query* transactions execute relational select operations, while *Update* transactions execute select-update operations. In each case, selections can be performed via sequential scans, clustered index scans, or non-clustered index scans.

For each transaction type (*Query* or *Update*), an execution plan is provided in the form of a set of parameters. In particular, both the access path and the mean selectivity for each relation are provided as execution plan parameters. The actual selectivity is varied uniformly over the range from $\frac{1}{2}$ $Selectivity_{class}$ to $\frac{3}{2}$ $Selectivity_{class}$. The probability that a selected tuple is updated is *UpdateFrac*. It is assumed that indexed attributes are not updated by the *Update* transactions.[4] The number of scans done by a *Query* is given by *NumScans*. For each scan, a new file is chosen (with replacement), but the mean selectivity and access path are kept the same. In order to model skewed data access, each file also has a hot region that consists of a percentage, *HotSize %*, of the file size. The location of the hot region in the file is denoted by *HotLoc*, which may be either *beginning*, *middle*, or *end*. The percentage of updates going to the hot region of a file is *HotUpdate %*.

We chose this workload in order to model a situation where there is a relatively large number of updates per query and where queries do a significant amount of work. We felt that this workload model, while it is simple, would be place sufficient demands on the version management system to highlight the important performance issues and tradeoffs.

Parameter	Meaning
MPL_{class}	Number of terminals (*class* is *Query* or *Updater*)
$ThinkTime_{class}$	Mean of exponential external think time for each class
NumScans	Number of scans that a query does (constant)
$AccessMeth_{class}$	Access method used by class
$Selectivity_{class}$	Average selectivity (uniform)
UpdateFrac	Fraction of selected tuples actually updated
HotSize	Size of the hot region of each file as a percentage of the file size
HotUpdate	Percentage of updates that go to the hot region of a file
HotLoc	Location of hot region within the file (*beginning*, *middle*, or *end*)

Table 3.2: Workload Model Parameters

[4] This assumption was made so that we could use single-version indexes in this study, leaving further exploration of indexing for future work. A more detailed discussion of indexing issues can be found in section 5.

3.2. The System Model

The system model encapsulates the behavior of the various DBMS (and operating system) components that control the logical and physical resources of the DBMS. The relevant modules are described in the remainder of this subsection. They include the CPU module, the disk manager module, the buffer manager module, the lock manager module, the version manager module, and the operator manager module. Table 3.3 summarizes the key parameters of the system model.

The CPU module encapsulates the behavior of the CPU scheduler. Except for disk transfer requests from the disk manager, which preempt other requests, the FCFS policy is used for CPU scheduling. Unless preempted then, a transaction is granted the CPU until it requests a new page from the buffer manager. The disk manager module encapsulates the behavior of the disk driver and controller, scheduling disk requests according to the elevator algorithm [Teor72]. The total service time is computed as the sum of the seek time, latency, settle time, and transfer time. The seek time of a disk request is computed by multiplying the parameter *DiskSeekFactor* by the square root of the number of tracks to seek [Bitt88]. The actual rotational latency is chosen uniformly over the range from 0 to *DiskLatency*. Settle time is a constant and is given by the parameter *DiskSettle*. The last component of the disk service time, transfer time, is computed from the given transfer rate, *DiskTransfer*. The CPU cost of transferring pages between the disk controller and buffer memory is modeled by charging *DiskXferCPU* instructions per page per transfer.

Parameter	Meaning
CPURate	Instruction rate of CPU
NumDisks	Number of disks
DiskSeekFactor	Factor relating seek time to seek distance
DiskLatency	Maximum rotational delay
DiskSettle	Disk settle time
DiskTransfer	Disk transfer rate
DiskPageSize	Disk block size
DiskTrackSize	Disk track size
NumBuffers	Number of buffer frames in the buffer pool
DiskXferCPU	Cost to transfer a page between memory and disk controller
BufCPU	Cost for a buffer pool hash table lookup
LockCPU	Cost for a lock manager request
VersionCPU	Cost to traverse a version-chain link
SelectCPU	Cost to select a tuple
CompareCPU	Cost to compare index keys
StartupCPU	Cost to start a select or select-update operator
TerminateCPU	Cost to terminate an operator

Table 3.3: System Model Parameters

The buffer manager module encapsulates the details of an LRU buffer manager. The number of page frames in the buffer pool is specified as *NumBuffers*, and they are shared among the main segment, index, and version pool pages of the database. To improve sequential access performance, the buffer manager also supports requests to read an entire track at a time. The CPU cost of searching for a page in the buffer pool hash table is modeled by charging *BufCPU* instructions. If the page is not resident, an additional *BufCPU* instructions are charged to insert the newly requested page in the buffer table. The lock manager module models a typical lock manager with hierarchical locking, lock escalation, and deadlock detection. Locking is done at the page-level (except for a few experiments where we also include results obtained using file-level two-phase locking). We chose page-level over record-level locking in order to decrease the required simulation time; we felt that this decision would not change the basic results because the update transactions are extremely short. The CPU cost associated with lock management is modeled by charging *LockCPU* instructions for each lock manager request.

The version manager module encapsulates the operations of record-level versioning with on-page caching and an overflow version pool, as was described in Section 2. The CPU costs of version management are modeled by charging *VersionCPU* instructions each time the version manager must traverse a version-chain link.

The operator manager encapsulates the operations necessary to execute the transaction types in the workload (i.e., select and select with update). As was previously described, the access methods supported are sequential, clustered index, and non-clustered index scans. The CPU costs of the operators are modeled by charging *SelectCPU* instructions to extract a single tuple from a disk page and *CompareCPU* instructions to compare two index keys. The model employs the following execution strategy for non-clustered index scans: A list of record IDs is generated using a B+ tree index, the list is sorted, and the records are then retrieved in physical order.

4. EXPERIMENTS AND RESULTS

In this section, we present the results of three experiments designed to examine the performance and storage characteristics of the new record-level MV2PL algorithm, presented in Section 2. In order to get a basic understanding of the algorithm's behavior, in Experiment 1 we look at the effects of varying the average query size under several cache sizes. To explore the impact of versioning on record clustering, this experiment is divided into two parts — one where the queries in the workload use clustered index scans and one where the queries use unclustered index scans. In Experiment 2 we compare the the tradeoffs of using the write-all cache policy versus the write-one policy. Finally, in

Experiment 3 we look at the performance impact of introducing access skew in the update transaction workload.

As a yardstick for comparison, we include performance results for level-one consistency locking in all of the relevant graphs.[5] In addition, we present results for standard two-phase locking (level-three consistency) to show why it is not suitable for use in a transaction processing environment with long-running queries. Our primary performance metrics are updater transaction throughput and query throughput. In addition, we are also interested in the storage cost necessary to retain prior versions. To ensure the statistical validity of our results, we verified that the 90% confidence intervals for response times (computed using batch means [Sarg76]) were sufficiently tight. The size of these confidence intervals were within approximately 1% of the mean for update transaction response time and within approximately 5% of the mean for query response time. An exception to the latter is when queries exhibit thrashing behavior (which we explain later in the text); however, we discuss only performance differences that were found to be statistically significant.

Table 4.1 contains the settings for the parameters that are used in the experiments. The system has a CPU that executes 12 million instructions per second and a single disk with a page size of 8K bytes, a track size of 2 pages and a buffer that holds an entire track. With this configuration, typical disk access times were on the order of 15 milliseconds and the system was I/O-bound for all of our experiments.

The database is composed of 4 files, each containing 5000 Wisconsin benchmark-sized records [DeWi90]. Each record contains 208 bytes of data and 19 bytes of overhead, for a total of 227 bytes. For MV2PL, records contain an additional 8 bytes to store a transaction identifier and a version chain pointer. With this record size, 36 records fit on a page (or 34 for MV2PL). Each file contains both a clustered and an unclustered B+ tree index, each with a node fanout of 450.

The workload consists of 12 update terminals and 4 query terminals. The query terminals wait an average of one second between the termination of one query and the submission of the next query (the actual time is chosen from an exponential distribution). To keep the load on the system high, update terminals do not have an external think time delay. Queries complete four scans using either a clustered index or an unclustered index (depending on the experiment). The query selectivity is chosen uniformly from $\frac{1}{2}$ to $\frac{3}{2}$ times the average selectivity, which is varied from 2% to 50%. Update transactions use a non-clustered index to select and then update 1-3 records in a single file. Both the updated file

[5] We also ran experiments with level-two consistency (i.e., where queries obtain short read locks), but the results were almost identical to level-one; therefore we do not include them in this paper.

Parameter	Setting
CPURate	12 MIPS
NumDisks	1
DiskSeekFactor	0.78 msec
DiskLatency	0-16.67 msec (uniform)
DiskSettle	2.0 msec
DiskTransfer	2,0 MBytes/sec
DiskPageSize	8K
DiskTrackSize	2 pages
NumBuffers	150 pages
NumFiles	4
FileSize	5000 records
RecSize	227 bytes, including overhead (235 for MV2PL)
NumKeys	450
MPL_{Query}	4
$MPL_{Updater}$	12
$ThinkTime_{Query}$	1 sec. (exponential)
$ThinkTime_{Updater}$	0 sec.
NumScans	4
$Selectivity_{Query}$	ranges from 2% to 50%
$Selectivity_{Updater}$	0.01% (1-3 records)
$AccessMethod_{Query}$	clustered or unclustered index
$AccessMethod_{Updater}$	unclustered index
UpdateFrac	100%
HotUpdate	0%
HotSize	0%
HotLoc	middle
DiskXferCPU	3600 instructions
BufCPU	150 instructions
LockCPU	150 instructions
VersionCPU	100 instructions
SelectCPU	400 instructions
CompareCPU	50 instructions
StartupCPU	10000 instructions
TerminateCPU	2000 instructions

Table 4.1: Parameter Settings

and the records accessed within the file are chosen uniformly (i.e., *HotUpdate* and *HotSize* are 0%), except in Experiment 3 where the access pattern is skewed.

In our experiments, we vary the query selectivity over a wide range to show how the various concurrency control and version management alternatives perform as queries increase in size. Size is a key factor here because as the queries become larger, the version management system must maintain transaction-consistent views of the database that are increasingly different than the current view. As an alternative, we could have achieved a similar effect by varying the database update rate (e.g., by changing the number of update transaction terminals). We should point out that even though we use a relatively small database in our experiments (to make simulations with large queries feasible), the algorithms should scale rougly linearly along with the database size. This is because the update rate to individual pages (and

tuples) decreases proportionally as the database size is increased. For example, if the database size is doubled, a selection query with a given selectivity will have to read twice as many data pages (and tuples); however, the update rate to each individual page (and tuple) will be halved.

4.1. Experiment 1a: Base Experiment — Clustered Index Scans

In this experiment, we study the performance impact of using record-level MV2PL with caching for our clustered index scan query workload. Figure 4.1 shows query throughput over a range of average query selectivities and Figure 4.2 shows the corresponding updater throughput. The first thing to notice in the graphs is that 2PL exhibits significantly better query throughput than the other algorithms at the expense of significantly worse updater throughput. This is because the update transactions in the workload are delayed due to long query lock-holding times; the system resources are therefore largely devoted to the execution of queries. This confirms our premise that 2PL is undesirable for running long queries in a OLTP system. For this reason we will not consider 2PL further in this paper.

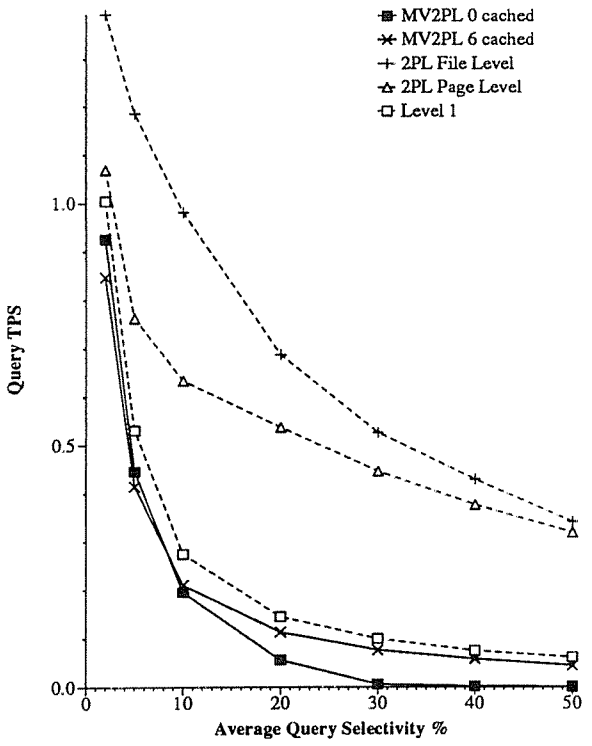

Figure 4.1: Query Throughput

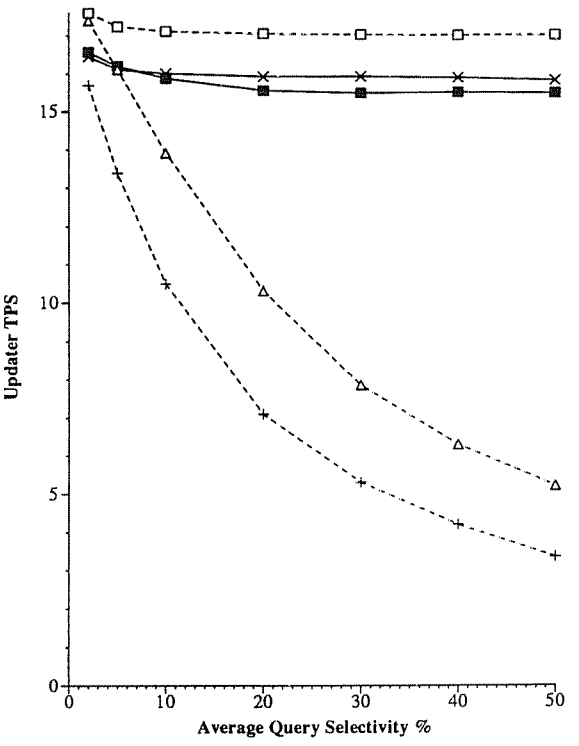

Figure 4.2: Updater Throughput

Clustered Query Workload with Write-One Policy

Turning to the other curves in Figures 4.1 and 4.2, we see that level-one consistency locking provides better query and update throughput than MV2PL. This is to be expected, as there must be some cost for providing serializability through versioning. It is important that this cost be reasonable, however, and that the multiversion locking algorithm satisfies our goal of minimal OLTP interference of queries with update transactions. Figure 4.2 shows evidence that MV2PL indeed satisfies this goal, as the update transaction throughput remains constant as query size is increased. In addition, the throughput level is reasonable, being less than 10% below the updater throughput for level-one consistency locking. This throughput difference for MV2PL can be attributed to the cost of writing records to the version pool and also to a lower buffer pool hit ratio for update transactions. The latter effect occurs because the database occupies more pages in MV2PL (due to the version chain pointers and on-page caches) and because of competition for database buffer frames from pages of the version pool.

The cost of versioning in terms of query throughput has two components — the additional cost to scan a larger database and the cost of performing disk accesses into the version pool. Figure 4.3 shows this clearly by presenting the

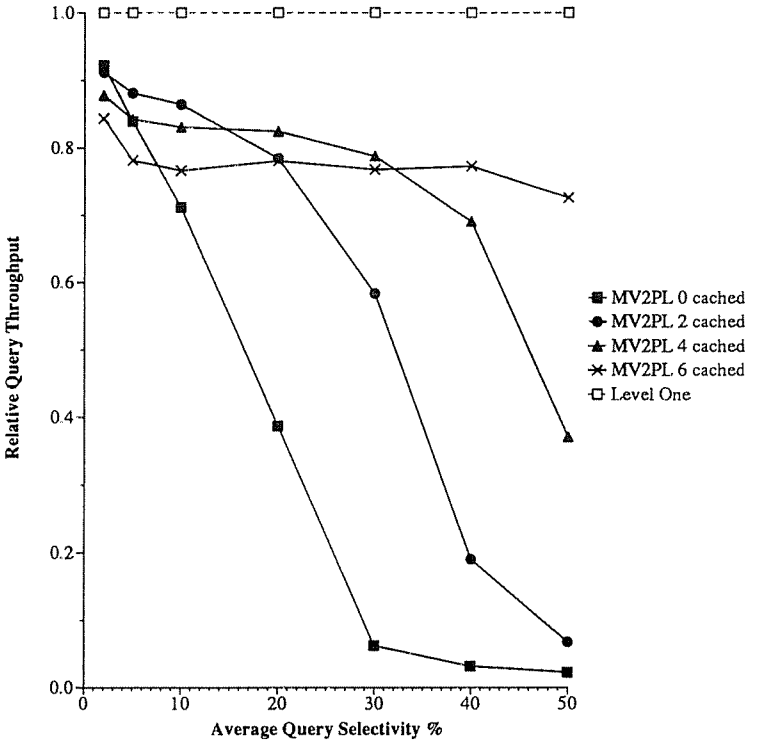

Figure 4.3: Relative Query Throughput
Clustered Query Workload with Write-One Policy

throughput of MV2PL with several different cache sizes *relative* to the throughput of level-one locking. Examining the graph, we see that at 2% selectivity the throughput values line up in cache-size order — cache size 0 has the highest throughput, size 2 has the next highest, 4 has the next highest, and 6 has the lowest. At this low level of selectivity, the first cost component, database size, is dominant because virtually all prior versions are garbage-collected while they are still in an on-page cache (i.e., there is very little version pool activity). As queries become larger, however, prior versions must be retained longer and there are more accesses into the version pool. When this occurs, the second cost component, version pool access, becomes dominant and the throughput relative to level-one locking drops significantly. In Figure 4.3, we see that the 0 cache-size curve drops before the 5% query selectivity point, the 2 cache-size curve drops at about 10% selectivity, and the 4 cache-size curve drops off at about 30% selectivity. The 6 cache-size curve is almost flat up to 50% selectivity[6]; virtually all of the prior versions accessed here are still in the on-page caches, so there are few, if any, version pool accesses.

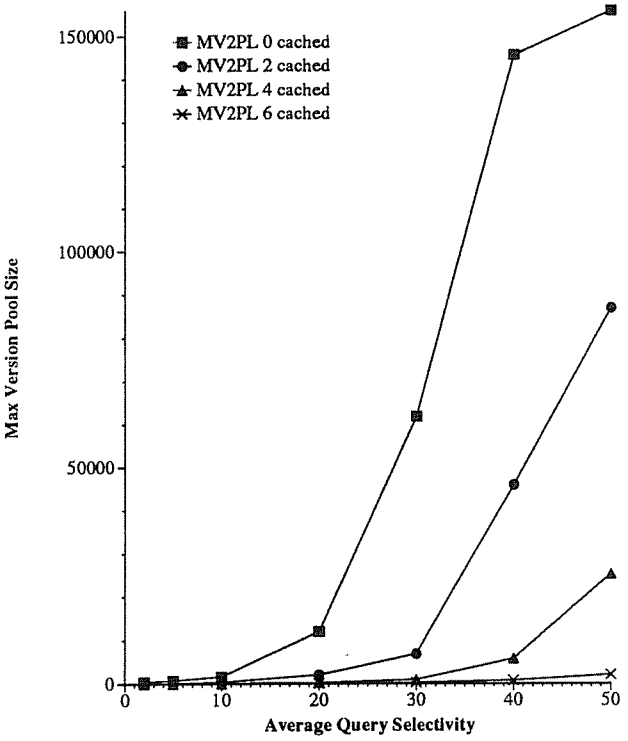

Figure 4.4: Maximum Version Pool Size

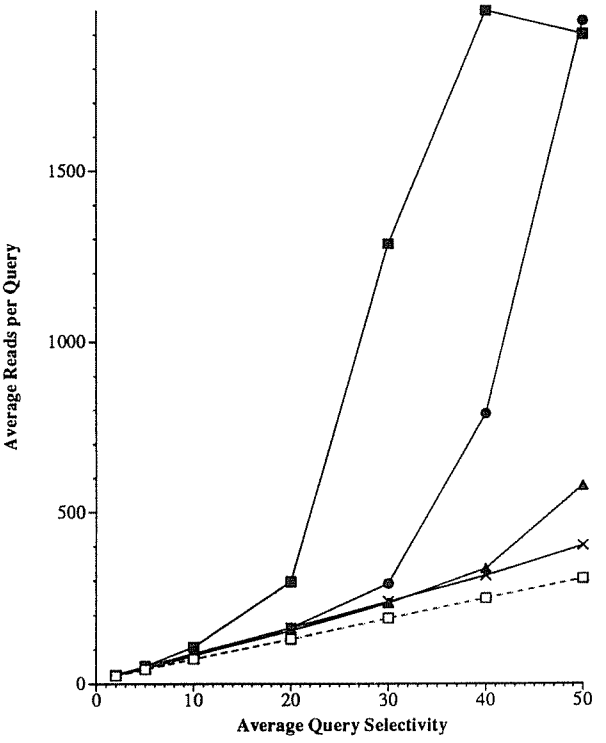

Figure 4.5: Pages Read per Query

Clustered Query Workload with Write-One Policy

[6] Recall that the actual selectivities of individual queries range between 25 and 75% at this point.

The preceding explanations of performance are supported by Figures 4.4 and 4.5, which show the maximum version pool size during the simulation and the average number of (logical) pages read per query.[7] The maximum version pool size is a recording of the maximum distance between *reader-first* and *last*, in terms of records, during the simulation execution. By comparing Figures 4.4 and 4.5 to the graph in Figure 4.3, we can see the relationship between the drop off in query throughput and the increase of activity in the version pool as the average query size is increased. Initially, the larger number of prior versions that must be retained for queries are absorbed by the on-page caches; after some point, however, a significant level of version pool activity begins. Version pool accesses are quite expensive, as one or more I/Os may be required to read a single record, which explains why the curves in Figure 4.3 fall sharply once version pool activity begins (i.e., where the curves in Figures 4.4 and 4.5 rise sharply). In addition, there is a non-linear relationship between query size and response time that leads to thrashing for queries: Increasing the query size requires that it must read more records *and* that it traverse version chains further because it is running against an older transaction-consistent snapshot of the database. Moreover, the additional I/Os that are necessary for traversing the version chains will make the query execute even longer, thus making its view even older. Therefore, once version pool accesses begin to slow down the queries in the workload, their response times and throughput degrade quickly.

Once queries begin to slow down due to significant version pool activity, the version pool can grow unreasonably large. Without an on-page cache, the version pool will retain the prior versions from all updates that have occurred during the lifetime of the longest-running query (since version pool space is deallocated sequentially). Indeed, we see that the version pool for the 0 caching case becomes extremely large in Figure 4.4, illustrating this point. As the cache size is increased, the version pool becomes less large at high selectivities. With a cache size of 4, the version pool becomes about 25% larger than the database itself, and with a cache size of 6, the version pool does not grow beyond approximately 10% of the database size. There are several reasons for this. Obviously, with a larger cache size more prior versions are stored on data pages and not in the version pool. More importantly, however, a larger cache allows queries to complete faster. Another important factor is that a larger cache provides more opportunities for prior versions to be garbage-collected soon after becoming unnecessary; in contrast, if a prior version spills into the the version pool, its storage cannot be reclaimed until it (and all previously written versions) becomes unnecessary due to the sequential allocation and deallocation of version pool space.

[7]The 0 cache size curve dips slightly from 40% to 50% query selectivity in Figure 4.5. This is because so few queries finished with a cache size of 0 that the actual number is not statistically significant.

The results of this experiment clearly show the advantages of keeping prior versions clustered in the main segment of the database (using caching) for a clustered index scan workload. Clearly, similar results would be obtained for a workload containing full sequential scan queries, as clustered index scans are just partial sequential scans. This experiment also showed that even when long queries begin to thrash due to accesses in the version pool, they do not affect the throughput of update transactions. Next, we turn our attention to a workload where queries access data through a non-clustered index rather than a clustered index.

4.2. Experiment 1b: Base Experiment — Unclustered Index Scans

In this experiment, we duplicate the previous experiment with the query workload changed from clustered- to unclustered index scans. Figure 4.6 shows query throughput over a range of average query selectivities and Figure 4.7 shows the corresponding update transaction throughput. The results are similar to those seen for the clustered index scan workload. The update transaction throughput curves remain flat for all of the cache sizes, which again is evidence that our goal of minimal interference is satisfied. Turning to the performance of queries, an examination of Figure 4.6 reveals that the throughput curves for level-one consistency locking and MV2PL with a cache size of 6 drop off initially and then level off at about 10% query selectivity. This is because the unclustered index scan access method retrieves records in physical order after obtaining a record-ID list and sorting it; by the 10% mean query selectivity point, all of the pages in the database must usually be accessed in order to retrieve the desired records. When the query selectivity is increased beyond this point, additional CPU processing is required, but with the level-one consistency locking algorithm, no additional I/Os are necessary; Since the system is I/O-bound with our parameter settings, this additional CPU processing (to select additional records from each page) does not affect throughput. On the other hand, the MV2PL algorithm has to do additional I/Os if the desired versions of newly selected records are not resident on the page. The cache size 6 curve in Figure 4.6 is relatively level across the selectivity range because the caches are large enough to hold the necessary prior versions; its performance is worse for queries than level-one consistency locking because of the lower buffer hit ratio and the additional pages that must be scanned due to the on-page caches. With smaller cache sizes, however, version pool accesses interfere with query performance as the query selectivity is increased, as we saw in Experiment 1a.

4.3. Experiment 2: Cache Write Policy Tradeoffs

In this experiment, we study the effects of changing the cache write policy from write-one to write-all with the clustered index scan query workload. As described in Section 2.1, the write-one policy appends a version to the version

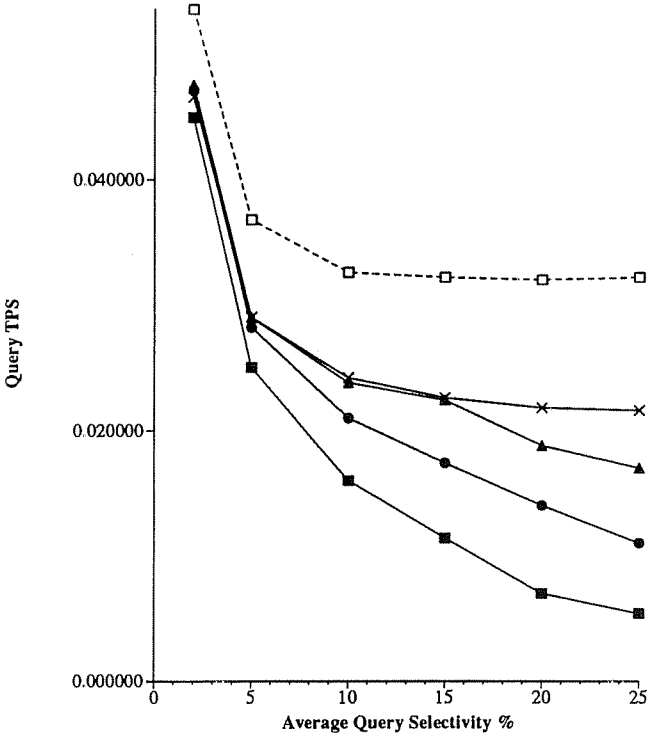

Figure 4.6: Query Throughput

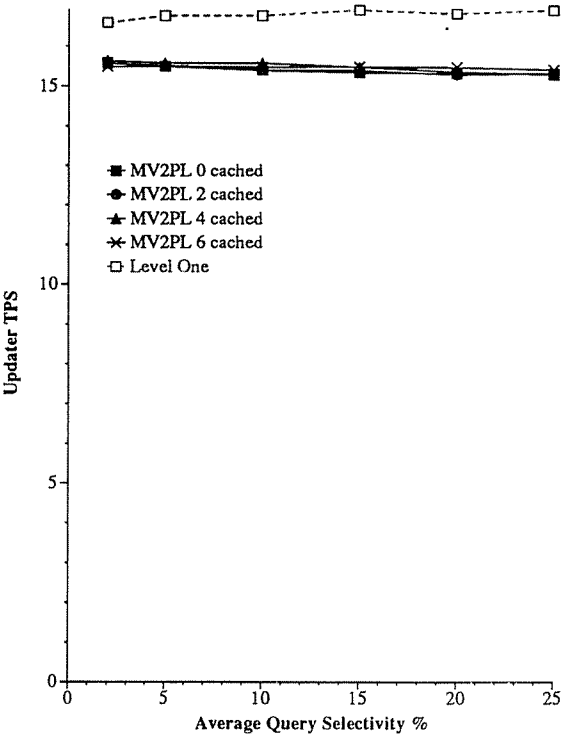

Figure 4.7: Updater Throughput

Unclustered Query Workload with Write-One Policy

pool only when it is chosen to be replaced in the cache. 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 and the least recently updated entry has not yet been appended to the version pool.

Figure 4.8 shows query throughput of both the write-one and write-all policies for the clustered-index scan workload (write-one has solid lines, write-all has dashed). Figure 4.9 shows the throughput the of write-all policy, relative to write-one, for each cache size. With a cache size of 6, the write policy does not affect query throughput over the range of selectivities that we have examined. This is to be expected since we determined in Experiment 1a that accesses to the version pool were rare with a cache size of 6. For the smaller cache sizes, the write-all policy improves the throughput of queries significantly for the higher query selectivities. For example, write-all with a cache size of 4 provides nearly as good query throughput as a cache size of 6. This is because the write-all policy writes a page's entire cache to the version pool at once, causing the version pool to be better clustered for queries that need to access it; this results in a higher query throughput. As in the previous experiments, update transaction throughput was not affected by the choice of cache write

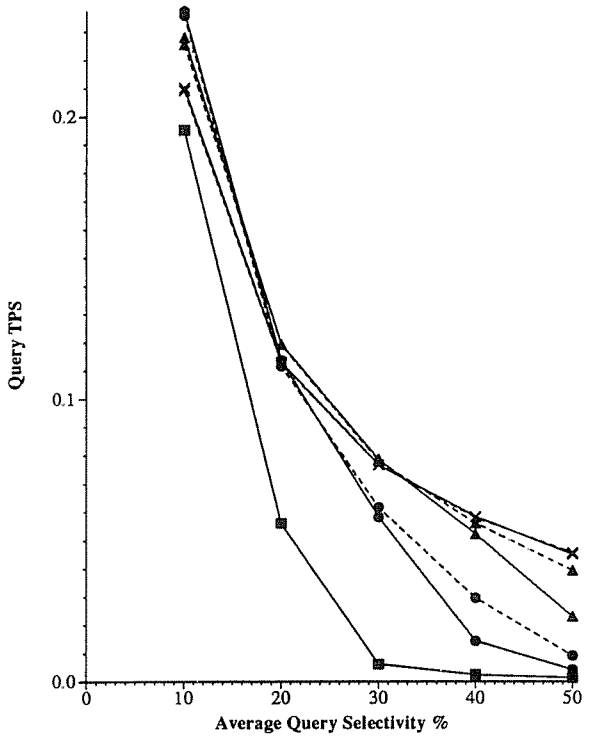

Figure 4.8: Query Throughput
(Write-one: solid lines, Write-all: dashed)

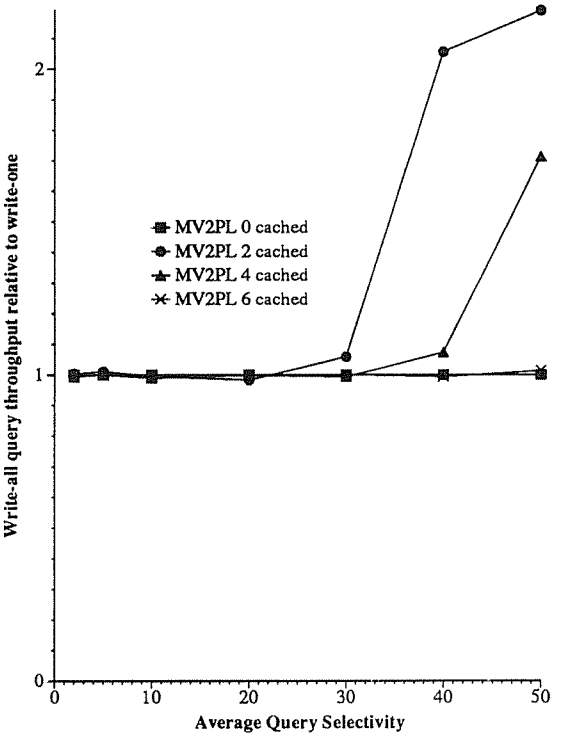

Figure 4.9: Throughput Improvement of Write-All
(Relative to Write-One)

Clustered-Scan Query Workload

policy, so the graph is not shown.

It is to be expected that the write-all policy will require a larger version pool size. There are two reasons for this: First, there will be multiple copies of some versions, and second, prior versions will not be given the most opportunity to be garbage-collected before being written to the version pool. Figure 4.10 shows the maximum version pool size for both the write-one and write-all policies for the clustered index scan workload (write-one has solid lines, write-all has dashed). Figure 4.11 shows the maximum size of the version pool for the write-all policy, relative to that of the write-one policy. For the region where the query throughputs of the write-one and write-all policies were the same, the write-all policy generated a larger version pool, as expected. However, the relative version pool size does not matter all that much in this region because the absolute sizes are quite small.[8] Interestingly, in the regions where write-all had a higher throughput than write-one, its version pool was actually smaller, as well. This is because the write-all policy completes queries at a

[8] This explains also why the cache size 6 curve is somewhat erratic.

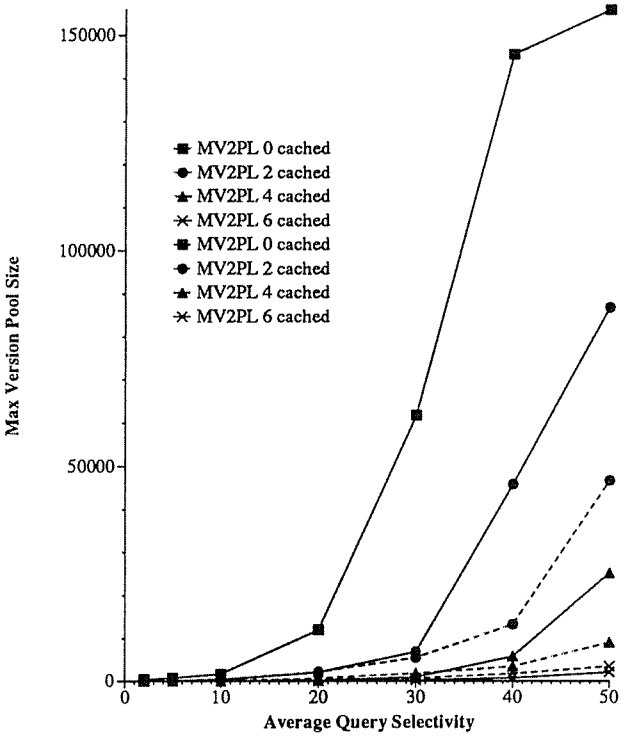

Figure 4.10: Maximum Version Pool Size
Write-one: solid lines Write-all: dashed

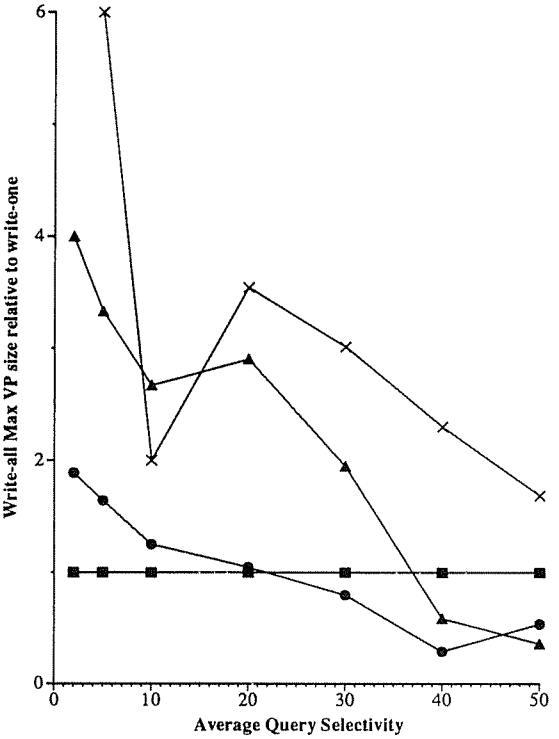

Figure 4.11: Max Version Pool Size of Write-All
(Relative to Write-One)

Clustered-Scan Query Workload

faster rate, and thus needs to keep around fewer prior versions.

This experiment has shown that the write-all policy is superior to the write-one policy in the presence of very long queries for the clustered-index scan workload. This is a direct consequence of its improved clustering of versions for queries. For short-running queries, the write policy did not affect the query throughput, nor was there an appreciable difference in the absolute sizes of the version pool. Therefore, write-all is a better policy to use in general for a clustered-index scan query workload.

To explore the robustness of these results, we repeated this experiment for the unclustered index scan query workload. We omit these results due to space limitations, but we summarize them here. For this workload, as should be expected, we found a smaller improvement for write-all when the level of version pool activity was significant. This is because the clustering behavior of write-all matters less here, as only a fraction of the records on each page are accessed (up to an average of 25% in this experiment).

4.4. Experiment 3: Skewed Updates

In this experiment, we study the effects of a skewed update pattern. In order to do so we vary the *HotSize* parameter from 1% (highly skewed) to 50% (no skew) while keeping *HotUpdate* fixed at 50%; *HotLoc* is set to *middle*. Queries in this experiment use the clustered-index scan access method. Figure 4.12 shows the updater throughput for a 25% average query selectivity across a range of hot region sizes, and Figure 4.13 shows the corresponding query throughput. Starting from the right-hand side of the update throughput graph, we see that updater throughput is fairly level for the largest hot region sizes (i.e., least skew) and then it increases as the hot region size is made smaller. This is because the buffer pool hit ratio for the update transactions is highest when the hot region size is smallest; for level-one consistency locking it varies from 52% to 72% across the range of hot region sizes in the figure.

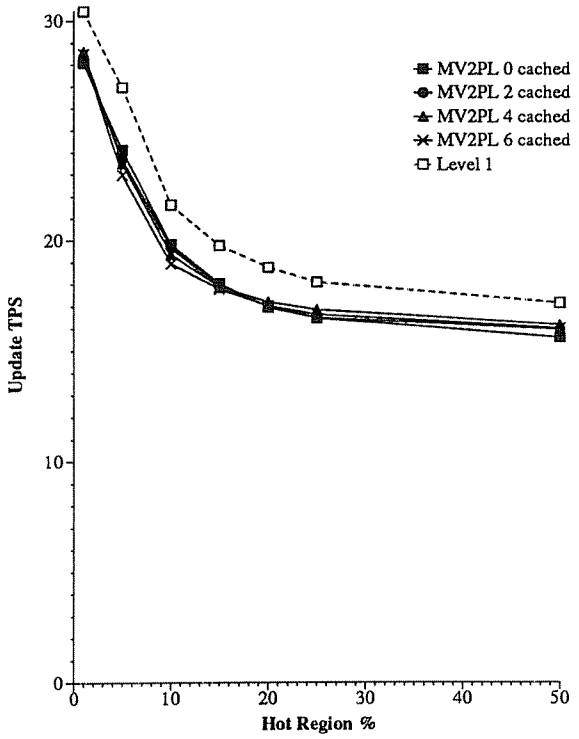

Figure 4.12: Updater Throughput

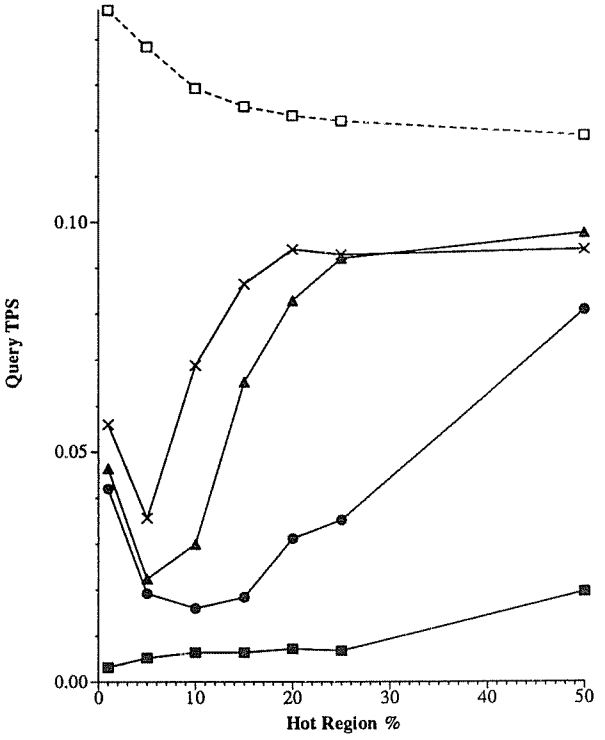

Figure 4.13: Query Throughput

Write-One Cache Write Policy
25% Average Query Selectivity
Skewed Update Pattern (50% of updates to X% of each file)

In the query throughput graph (Figure 4.13), we see that for level-one consistency locking, query throughput is also level at the larger hot region sizes; it then increases as the hot region becomes very small (although it does so less sharply than the corresponding update throughput curve). This trend is directly related to the updaters' buffer pool hit ratio. When this hit ratio is high, the updaters utilize the disk less frequently, thus leaving more of the disk resources for use by the queries. For the MV2PL curves, however, we see a somewhat different trend. As the hot region is made smaller, moving from right to left in Figure 4.13, each throughput curve drops to a certain point (between 5 and 10% selectivity) and then increases, again (an exception is the cache size 0 curve, for reasons which will be identified shortly.) These drops in query throughput are due to the overloading of the on-page caches as the hot region becomes smaller and thus more concentrated. This is made more clear by Figure 4.14, which shows the ratio of records that are garbage-collected while in an on-page cache to the total number of updates. The MV2PL curves in this figure follow the same general pattern as the query throughput curves in Figure 4.13. As the hot region becomes smaller and more highly concentrated, the on-page caches become overloaded and the garbage collection ratio falls off due to versions spilling into the version pool. This does not affect the cache size 0 curve, for obvious reasons, which explains why the 0 cache size throughput curve does not follow the other MV2PL curves. Turning to the left hand portion of Figure 4.14, we see that the garbage collection ratio begins to increase again as the hot region becomes small and highly concentrated. This is because the high frequency of updates to records in the small hot region results in the creation of more versions of many records than are needed to satisfy the view of each query in the system. To understand why, recall what happens when a particular record is updated twice between the entry of one query into the system and the entry of the next query; the first update may be discarded because it is no longer necessary. The resulting increase in the garbage collection ratio here leads to a corresponding increase in query throughput.

This experiment has shown that there is a loss of query throughput due to cache overflows when a uniform cache size is used in in the presence of non-uniform updates to pages. This suggests that the cache size on each page should be perhaps determined based upon the update frequency of the page; how to accomplish this is an interesting question for future work. This experiment has also shown that when particular records are updated frequently, fewer overall versions must often be maintained, resulting in shorter version chains and higher query throughput.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced a new design for record-level, multiversion, two-phase locking (MV2PL) and provided some insights into its performance. Our design utilizes on-page caching and garbage collection of prior

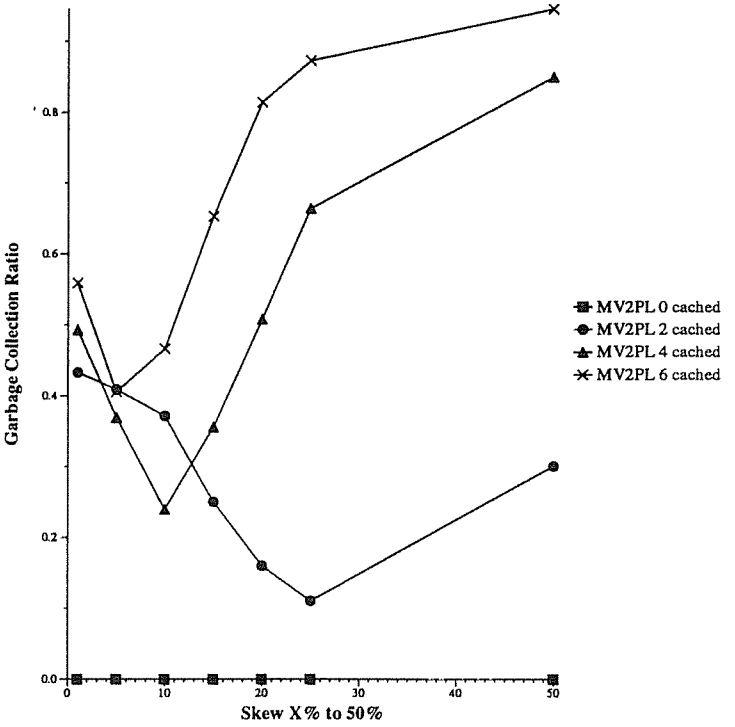

Figure 4.14: Garbage Collection Ratio
Write-One Cache Write Policy
25% Average Query Selectivity
Skewed Update Pattern (50% of updates to X% of each file)

versions in order to reduce the number of accesses into the version pool. Our performance results indicate that the design provides reasonable throughput, as compared to level-one and level-two consistency locking, when the size of the on-page caches are large enough to prevent significant version pool activity. In addition, we have described the write-one and write-all policies for writing entries from the cache to the version pool. We have found the write-all policy to be superior to the write-one policy in the presence of very long queries. In situations where queries are relatively short, both policies have similar throughput, but the write-one policy is more space-efficient in the version pool; the absolute size of the version pool in such situations is minimal, however. Therefore, write-all is a better policy to use in general.

Our results are significant because they indicate that versioning can be used to provide serializability for queries in an on-line transaction processing setting, and this can be achieved at a reasonable cost. However, further work is required before these ideas can be applied directly in practice. Towards this goal, we plan to continue our work in several directions. First, we plan to study the utility of the *view sharing* idea that was presented in section 3, as its performance

was not considered in this initial study. Second, we plan to extend our work to include multiversion indexes. Work has been done in the area of multiversion indexes for historical databases [Kolov89, Lome90], but we are interested in indexing in transient versioning systems where prior versions relatively short-lived. One important design decision here is whether index entries should point to the head of a version chain (as we have assumed in this paper) or to each individual version (i.e., with one index entry per version). The latter option has the advantage that the appropriate version to read can be identified directly from the index, but updates will be significantly more costly since any update would then always require updates to each index on the file. Finally, we plan to study how our MV2PL ideas can be extended to parallel database systems. In particular, we are interested in exploring how versioning can best be made to coexist with data replication schemes such as the one introduced in [Hsia90].

REFERENCES

[Agra87] Agrawal, D., A. Bernstein, P. Gupta and S. Sengupta, "Distributed Optimistic Concurrency Control with Reduced Rollback," *Journal of Distributed Computing*, Springer-Verlag, 2(1), January 1987.

[Agra89] Agrawal, D. and S. Sengupta, "Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control," *Proc. ACM 1989 SIGMOD Conf.*, Portland, OR, June 1989.

[Baye80] Bayer, et al., "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Sys.*, 5(2), June 1980.

[Bern83] Bernstein, P., and N. Goodman, "Multiversion Concurrency Control: Theory and Algorithms," *ACM Trans. on Database Sys.*, 8(4), December 1983

[Care86] Carey, M., W. Muhanna, "The Performance of Multiversion Concurrency Control Algorithms," *ACM Trans. on Computer Sys.*, 4(4), November 1986.

[Chan85] Chan, A., and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Trans. on Software Eng.*, SE-11(2), Feb 1985.

[Chan82] 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

[DeWi90] DeWitt, D., et al., "The Gamma Database Machine Project," *IEEE Trans. on Knowledge and Data Eng.*, 2(1), March 1990.

[Gray76] Gray, J., R. Lorie, F. Putzolu, and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in *Modeling in Data Base Systems*, North Holland Publishing (1976).

[Gray79] Gray, J., "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979

[Gray81] Gray, J., et al., "The Recovery Manager of the System R Database Manager," *ACM Comp. Surveys*, 13(2), June 1981.

[Hsia90] Hsiao, H., *Performance and Availability in Database Machines with Replicated Data*, Ph.D. Thesis, Comp. Sci. Tech. Rep. No. 963, Univ. of Wisconsin-Madison, Sept 1990.

[Kolov89] Kolovson, C., and M. Stonebraker, "Indexing Techniques for Historical Databases," *IEEE Conf. on Data Enginering*, Los Angeles, CA, February 1989.

[Livn89] Livny, M., *DeNet User's Guide*, Version 1.5, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1989.

[Lome90] Lomet, D., and B. Salzberg, "The Performance of a Multiversion Access Method," *Proc. ACM 1990 SIGMOD Conf.*, Atlantic City, NJ, May 1990.

[Moha89] Mohan, C., et al., *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, DBTI Research Report RJ7341, IBM Almaden Research

Center, 1989.

[Pira90] Pirahesh, H., et al, "Parallelism in Relational Database Systems: Architectural Issues and Design Approaches," *IEEE 2nd International Symposium on Databases in Parallel and Distributed Systems,* Dublin, Ireland, July 1990.

[Ragh91] Raghavan, A., and Rengarajan, T.K., "Database Availability for Transaction Processing," *Digital Technical Journal* 3(1), Winter 1991.

[Reed83] Reed, D., "Implementing Atomic Actions on Decentralized Data," *ACM Trans. on Computer Sys.* 1(1), February 1983.

[Robi82] Robinson, J., *Design of Concurrency Controls for Transaction Processing Systems,* Ph.D. Thesis, Comp. Sci. Tech. Rep. No. CMU-CS-82-114, 1982.

[Sarg76] Sargent, R., "Statistical Analysis of Simulation Output Data," *Proc. 4th Annual Symp. on the Simulation of Computer Systems,* 1976.

[Schn90] Schneider, D., *Complex Query Processing in Multiprocessor Database Machines,* Ph.D. Thesis, Comp. Sci. Tech. Rep. No. 965, Univ. of Wisconsin-Madison, Sept. 1990.

[Son90] Son, S., and N. Haghighi, "Performance Evaluation of Multiversion Database Systems," *IEEE Conf. on Data Enginering,* Los Angeles, CA, February 1990.

[Stea81] Stearns, R. and D. Rosenkrantz, "Distributed Database Concurrency Control Using Before-Values," *Proc. 1981 ACM SIGMOD Conf.,*, 1981

[Weih87] Weihl, W., "Distributed Version Management for Read-Only Actions," *IEEE Trans. on Software Eng.*, 13(2), January 1987.

[Wu91] Wu, K.-L., P.S. Yu and M.-S. Chen, *Dynamic Finite Versioning for Concurrent Transaction and Query Processing,* Technical Report RC 16633, IBM T.J. Watson Research Center, Yorktown Heights, NY, March 1991.