Towards Practical Multiversion Locking Techniques for On-Line Query Processing

Paul M. Bober

Technical Report #1160

November 1993

TOWARDS PRACTICAL MULTIVERSION LOCKING TECHNIQUES FOR ON-LINE QUERY PROCESSING

by

PAUL M. BOBER

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1993

ABSTRACT

Query processing is becoming an increasingly important function of on-line transaction processing (OLTP) systems. The concurrency control algorithm found in most commercial database systems, two-phase locking (2PL), however, does not efficiently support on-line query processing. This is because 2PL causes queries to lock large regions of the database for long periods of time, thus causing update transactions to block. To solve the query/update data contention problem, multiversion extensions to two-phase locking have been proposed. In multiversion two-phase locking (MV2PL), prior versions of data are retained temporarily to allow each read-only query to serialize before all update transactions that were active during any portion of its lifetime. Queries do not contribute to data contention in MV2PL since they do not set or wait for locks. This form of versioning, where old copies of data are retained temporarily for concurrency control purposes, is sometimes referred to as transient versioning.

In the thesis, techniques for the efficient implementation of multiversion locking are proposed and evaluated. In the first part of the thesis, an efficient scheme for organizing multiple versions of tuples on secondary storage is proposed. The results of a simulation study are then presented that clearly demonstrate the benefits of this scheme. In the second part of the thesis, a number of options for extending single-version index structures to handle transient multiversion data are outlined, and the results of a performance study comparing them are presented. Finally, in the last part of the thesis, a new multiversion locking algorithm, multiversion query locking (MVQL), is proposed and evaluated. MVQL generalizes MV2PL by providing queries with a range of consistency forms (the strictest being that provided by MV2PL). The thesis describes how each successively weaker form of consistency can reduce query execution cost as well as version storage cost, and the results of simulation experiments are presented that quantify these cost reductions.

[illegible]

ACKNOWLEDGMENTS

It has been my distinct privilege to work under the direction of Michael Carey. Mike has given me superb technical guidance and a great deal of encouragement and support. I have the deepest respect for him both as a professor and as an individual.

I am grateful to David DeWitt for the encouragement he has given me during my years in the department, and for the interest he has shown in me.

I would also like to thank the other members of my committee: Parmesh Ramanathan, Jeff Naughton, and especially Miron Livny. I had many conversations with Miron, and I both enjoyed and benefited from them.

I also benefited from my interaction with Dan Dias with whom I worked for a summer at IBM Research. I'd like to thank Dan for taking the time to work with me, and for his encouragement.

Mike Litzkow deserves special thanks for keeping condor up and running, and for his prompt handling of my bug reports and requests. I would also like to thank the lab staff for their excellent assistance.

Thanks to Sheryl Pomraning, Lorene Webber, Susan Dinan, Lynn White, and Ralph Rodriguez for their friendly help with administrative matters, and for making the department a more interesting place to work.

I'd also like to acknowledge Bruce Hillyer for having introduced me to computer science. Bruce devoted a lot of time to me when I was in high school, and provided a role model for me by subsequently pursuing a Ph.D. Throughout the years I've been glad to keep in touch with him, and he never ceases to impress me.

At Wisconsin, I was fortunate to have made a number of good friends in the department who helped make graduate school more enjoyable: Mike Franklin (with whom I drank countless cups of coffee), Beau Shekita, Manish Mehta, Shaul Dar, Dan Lieuwen, Kurt Brown, Jonathan and Lisa Yackel, Trip and Lexie Lazarus, Scott Vandenberg, Kristin Bennett, V. Srinivasan (Srini), Sarita and Vikram Adve, Scott and Lisa Leutenegger, Mike Zwilling, Molly Peebles, and others. I'd like to give special thanks to Vikram Adve for answering my many questions on analytical performance modeling.

My daughter, Katie, has brought me so much joy and happiness during the past 16 months. She deserves a special acknowledgement for delaying her arrival into the world until my VLDB paper was completed (but just barely!).

Finally, I would like to thank my wife, Barbara. She has given me an incredible amount of love and support, and I sincerely doubt that I would have been able to finish this thesis without her.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

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 on-line transaction processing (OLTP) systems. The concurrency control algorithm found in most commercial database systems, two-phase locking (2PL) [Eswa76], however, does not efficiently support on-line query processing. This is because 2PL causes queries[1] to lock large regions of data for long periods of time, thus causing update transactions to suffer long delays due to data contention with queries. As a result, users are often forced to make compromises in either the consistency or timeliness of queried data. One such compromise is to run queries without obtaining locks or using only short-term locks, allowing the queries to see transaction-inconsistent answers [Gray79]. These approaches are commonly referred to as GO processing and cursor stability locking [Pira90], respectively. A different compromise is to maintain two separate databases, one for OLTP transactions and another for ad-hoc queries [Pira90]. In this latter 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.

To solve the query/update 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 Computer Corporation in the early 1980s [DuBo82]. This extension was also used in a system developed at Computer Corporation of America (CCA) in the same time frame [Chan82, Chan85], and it has been incorporated in DEC's

[1] Unless otherwise stated, the term *query* in this thesis refers to a long-running read-only transaction (possibly containing many SQL statements), rather than a single SQL statement.

Rdb relational DBMS product as well [Ragh91]. In multiversion two-phase 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), is sometimes referred to as *transient versioning* [Pira90].

1.2. Thesis Preview

In this thesis we propose and investigate techniques for the efficient implementation of multiversion locking to support on-line query execution in high performance transaction processing systems. The thesis is subdivided into three major parts, each of which is now previewed briefly.

1.2.1. Efficient Version Management Techniques

A decision that must be made before building a multiversion locking system is how to arrange the transient multiversion data on secondary storage. In the CCA prototype, current versions of data pages were stored as they would be in a single-version database, and prior versions of data were appended to a sequential log-like version pool [Chan82, Chan85]. This scheme supports the efficient creation of new versions as well as inexpensive garbage collection. Since accesses to the version pool are random, however, the scheme disrupts the inherent sequentiality of query access. To improve query sequentiality, we propose and evaluate a tuple-level refinement, called on-page version caching, that reserves a small portion of each data page to cache prior tuple versions; the version pool is thus used only for versions that overflow from page caches. Lastly, we introduce a *view sharing* mechanism that has the potential for reducing the storage cost of versioning by running several queries together against the same transaction-consistent view of the database.

1.2.2. Multiversion Indexing

Since indexes are important for good performance in database systems, it is important to determine how they may coexist with multiversion locking. Indexing in a multiversion database is more complicated than indexing in a single version database because tuples can have potentially many versions, each having different indexed attribute values and physical locations. In this thesis, we outline and evaluate four different options for extending single-version indexing schemes to handle multiversion data. The schemes are all capable of supporting certain important query optimizations, and they differ in where they place version selection information (i.e., references to individual tuple versions).

1.2.3. Multiversion Query Locking

Because update transactions typically have stringent response time constraints, storage systems for transient versioning usually optimize access to current versions of data at the expense of making access to older versions more expensive. Thus, a query's execution cost is related to the age of the versions that it reads. In this thesis, we outline several query consistency forms, and we show that each successively weaker form can reduce query execution cost as well as version storage cost by allowing queries to read younger versions. We then present a generalization of MV2PL, called multiversion query locking (MVQL), that can be tailored to provide each of these forms (with standard MV2PL providing the strictest form). Even the weakest of the consistency forms guarantees that queries at least see transaction-consistent data; this is in contrast to approaches that avoid versioning altogether, instead allowing queries to see inconsistent data (e.g., GO processing, cursor stability locking, and epsilon-serializability [Wu92]).

1.3. Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 covers related work. In this chapter, we briefly survey the relevant multiversion concurrency control literature, and we describe both the basic MV2PL algorithm and the CCA multiversion storage organization scheme in detail.

The first research chapter, Chapter 3, is devoted to efficient storage management techniques for versions. In this chapter, we identify some of the performance problems of the initial CCA storage management scheme, and we then present the on-page caching refinement to this scheme. We also introduce view sharing, and we show how it may be used with on-page caching to reduce storage cost. Finally we present the results of a set of experiments that clearly demonstrate the benefits of on-page caching.

The next chapter, Chapter 4, is devoted to multiversion indexing. In this chapter, we outline four options for extending single-version index structures to handle transient multiversion data. We then explore the tradeoffs between the four approaches via simulation.

The final research chapter, Chapter 5, is devoted to MVQL. In this chapter, we first explain the various forms of consistency that are provided by MV2PL and MVQL, and we then derive the new MVQL algorithm as a generalization of MV2PL. Finally, we present the results of a series of experiments that compare the performance and storage overhead of MVQL to that of MV2PL.

Finally, we conclude with a summary of the main lessons of the thesis in Chapter 6, and we close by discussing some possible directions for future work.

CHAPTER 2

RELATED WORK

In this chapter, we set the stage by reviewing the work that served as a takeoff point for the results presented in this thesis. We begin by surveying the relevant literature on multiversion concurrency control. We then review the basic MV2PL algorithm and the CCA version pool scheme for managing storage for multiple versions of data.

2.1. Multiversion Concurrency Control Algorithms

As mentioned earlier, MV2PL is the starting point for the work reported here. MV2PL is only one of a number of multiversion concurrency control algorithms that have been published in the literature. We are primarily interested in MV2PL in this thesis because it is a direct extension of the de facto industry standard, 2PL. 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 [Reed78] was actually the first multiversion concurrency control algorithm proposal; it can be viewed as a multiversion extension of basic timestamp ordering (BTO) [Bern81].[1] Reed's scheme is not ideally-suited for use with long-running read-only queries, however, as read-only queries can sometimes cause update transactions to abort. Furthermore, operations issued by read-only queries must update timestamp information that is physically associated with each object (possibly turning reads into writes).

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 [Baye80, Stea81]. Again, however, these schemes are not ideally-suited for use with long-running read-only queries. They either allow update transactions to abort queries, or they allow queries to block or abort update transactions, neither of which is particularly desirable in an OLTP setting.

[1] The development of Reed's scheme actually preceded that of basic timestamp ordering, however.

A number of multiversion concurrency control algorithms have been proposed that employ an existing single-version concurrency control algorithm to serialize the execution of update transactions, while using a combination of versioning and timestamps to provide queries with a prior transaction-consistent state of the database. In fact, Agrawal and Sengupta [Agra89] describe how this may be done in a modular fashion (i.e., separating concurrency control and version management) using *any* existing single-version conflict-based concurrency control algorithm. Queries and update transactions cannot cause each other to block or abort in these algorithms, as each query serializes before all update transactions that arrived prior to its start. As a result, these algorithms are potentially well-suited for on-line query processing. In addition to MV2PL, which we have already mentioned, examples of such algorithms include multiversion optimistic algorithms [Robi82, Care83, Lai84, Agra87], multiversion timestamp ordering [Agra89], and the multiversion tree protocol [Silb82].

A *compensation-based* query processing technique has recently been proposed as an alternative to multiversion concurrency control for relational queries [Srin92]. It can be viewed as a form of *semantic versioning* since a query essentially creates its own consistent version of the relevant underlying data while executing [Srin92]. A high-level description of the approach is as follows: In the first phase of this two-phase approach, queries do a "fuzzy" scan of the base relations using cursor-stability locking, and a set of temporary relations is created. In addition, a special compensating record is generated for each update during this phase that could be necessary to bring the temporary relations to a transaction-consistent state. During the second phase of the approach, query execution is completed using the temporary relations, with compensating records being applied on-the-fly as needed to provide the query with a transaction-consistent view of the relevant data. A limitation of the approach, however, is that it is specific to individual queries that use relational style set-oriented operators. Thus, it is not directly applicable to the execution of long-running application programs that may involve several queries, the use of cursors, and so on.

2.2. 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, 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).[2] 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 that might 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. Later, when a page is deleted, the commit timestamp of the deleter becomes 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, i.e., a version with an infinite value. 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).

2.3. 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 of every page in the database, and the version pool contains prior versions of pages. Before a main segment page may be updated, the entire page must be copied to the version pool. The version pool is organized as a circular buffer,

[2]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. A separately maintained list is then used to map from transaction ids to commit timestamps [Chan82].

much like a *write-ahead log* in a traditional recovery manager [Gray79]. In fact, the version pool was designed to be used as a UNDO log for recovery, as well as a version storage structure for MV2PL. In the absence of a separate write-ahead log for recovery (containing both UNDO *and* REDO log records), any main segment pages that are updated by a transaction must be forced to disk at commit time.

The CCA design chains all of the versions of a page in reverse chronological order; thus, the desired version of a given page is located by first starting at the current version of the page (in the main segment), and then following the version chain until the desired page version is reached. Three attractive properties of the CCA version pool approach 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[3]), and (iii) storage reclamation is relatively straightforward. Figure 2.1 depicts the main segment of the database, the version pool, the pointers used to manage version pool space, and the version chain 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.

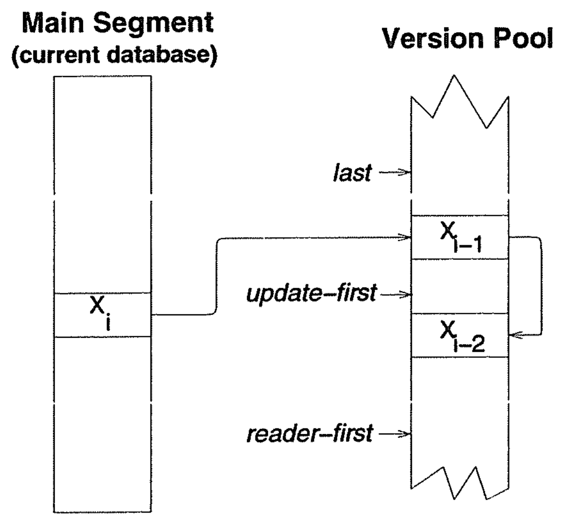


Figure 2.1: CCA Version Pool Organization

[3] In contrast, DEC's Rdb system stores old versions of data on "shadow" pages which must be first read and then written whenever an update occurs [Josh93].

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 it must make to read a given page increases with time. As a result, queries may begin to thrash if they are sufficiently large [Bobe92a]. The on-page version caching refinement discussed in the next chapter was designed to alleviate these problems.

CHAPTER 3

EFFICIENT VERSION STORAGE MANAGEMENT

3.1. Introduction

As described in the previous chapter, the CCA MV2PL design 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 present several refinements in this chapter. 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, to reduce storage overhead and increase potential concurrency, 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 following sections describe the last two of these refinements in detail.

3.2. On-Page Version 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 3.1 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.

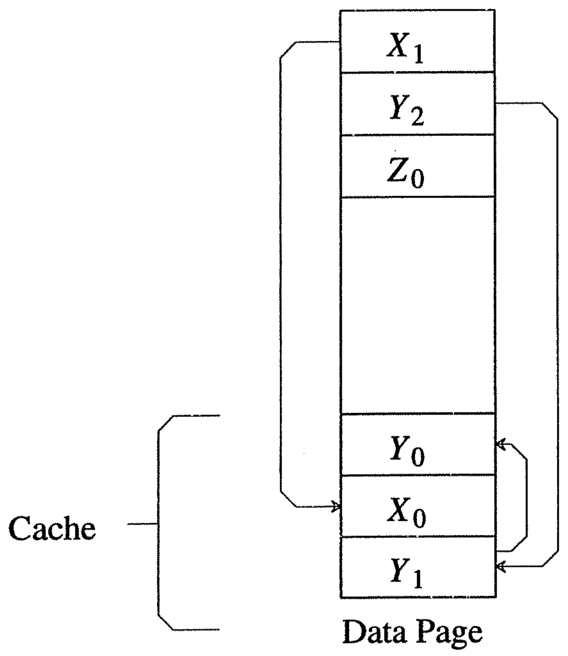


Figure 3.1: A data page with a cache size of 3

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 version located in a on-page cache is appended to the version pool. The *write-one* policy appends a tuple version 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 and the least

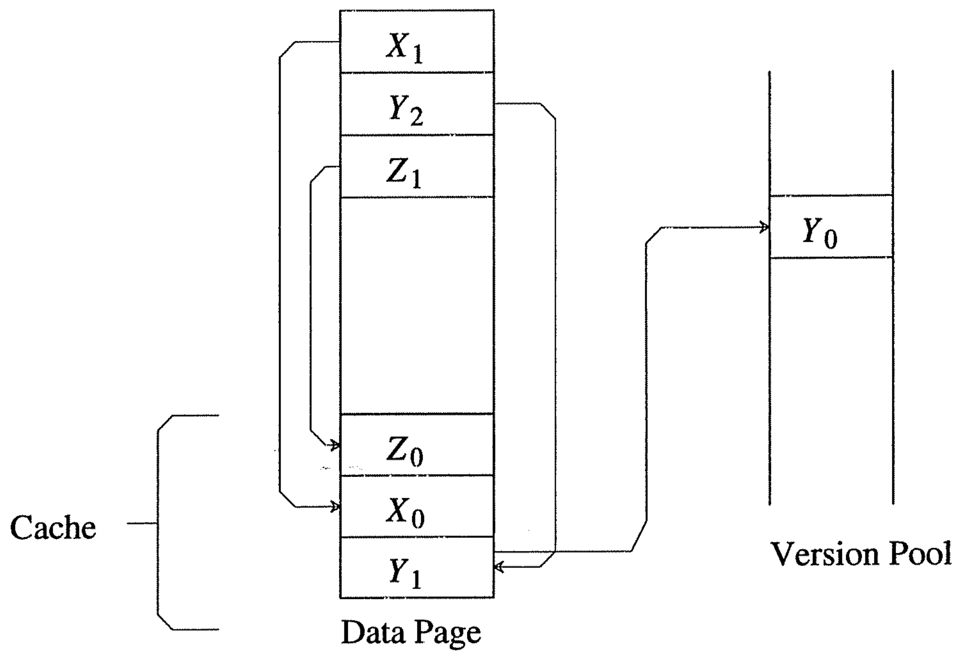


Figure 3.2: Write-one policy

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 3.2 shows the write-one policy being used when an update to record Z on the page from Figure 3.1 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 3.3 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.

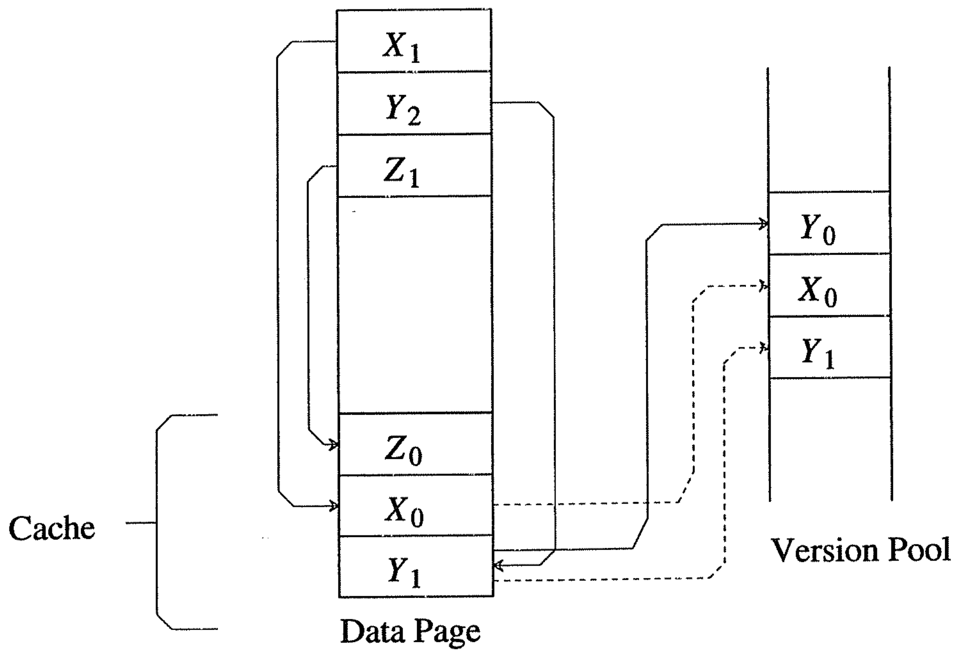


Figure 3.3: Write-all policy

3.2.1. Garbage Collection with On-Page Version Caching

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 [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 that have migrated to the version pool, versions that still 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 3.4 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

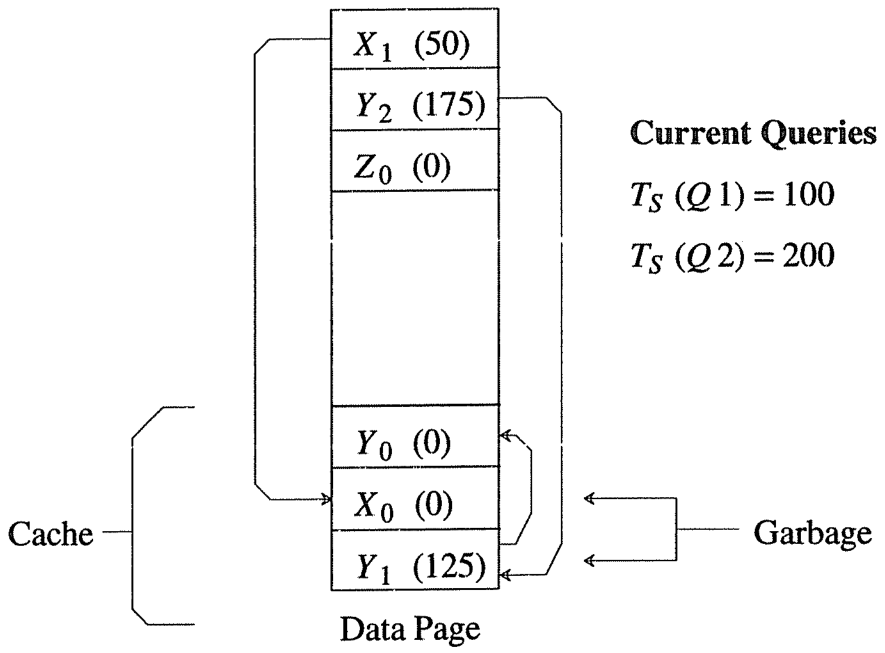


Figure 3.4: On-page garbage collection

time when it becomes the oldest running query until the time when it finishes.

3.3. 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.[1] 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.[2]

Figure 3.5 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

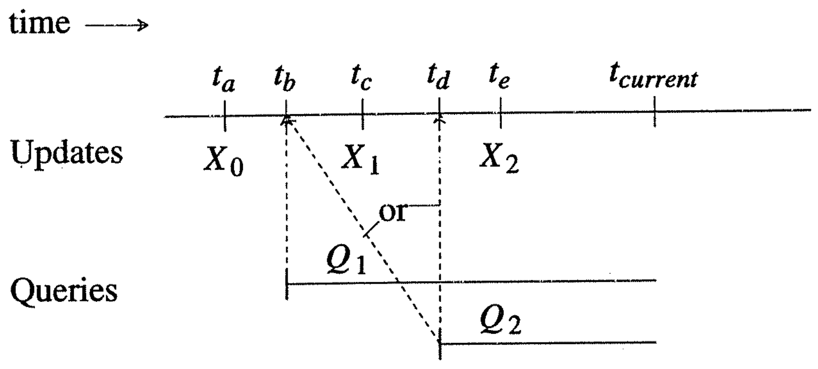


Figure 3.5: View Sharing

[1] 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.

[2] Similar ideas, developed independently, are presented in [Wu91] and [Moha92].

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. [Bobe92c] explores this tradeoff further using an analytical storage cost model.

3.4. 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. 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.

3.4.1. 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.

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.

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

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

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 1/2 $Selectivity_{class}$ to 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.[3] 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.

3.4.2. 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

[3] This assumption was made so that we could use single-version indexes in this study, leaving exploration of multiversion indexing to the next chapter.

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

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.

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 3.2. 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.

3.5. 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 this chapter. 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 GO processing in all of the relevant graphs.[4] In addition, we present results for standard two-phase locking 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 3.4 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 1/2 to 3/2 times the average selectivity,

[4] We also ran experiments with cursor-stability locking (i.e., where queries obtain short read locks), but the results were almost identical to GO processing; 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 3.4: Parameter Settings

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 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.

3.5.1. 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 3.6 shows query throughput over a range of average query selectivities and Figure 3.7 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

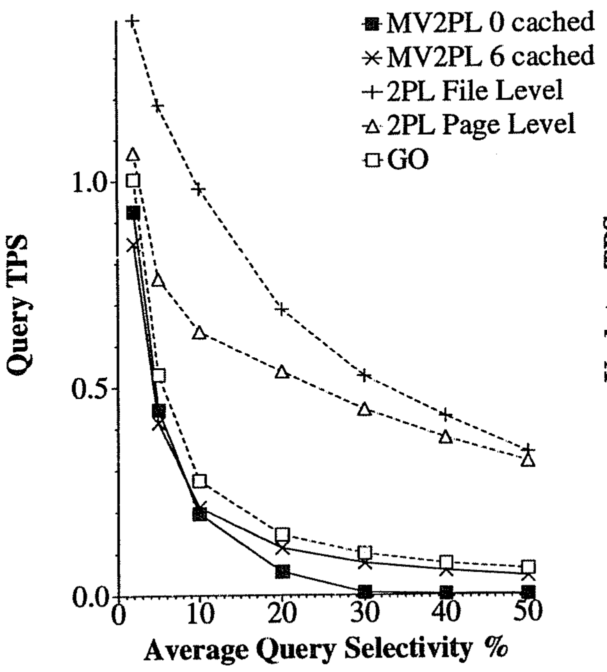


Figure 3.6: Query Throughput

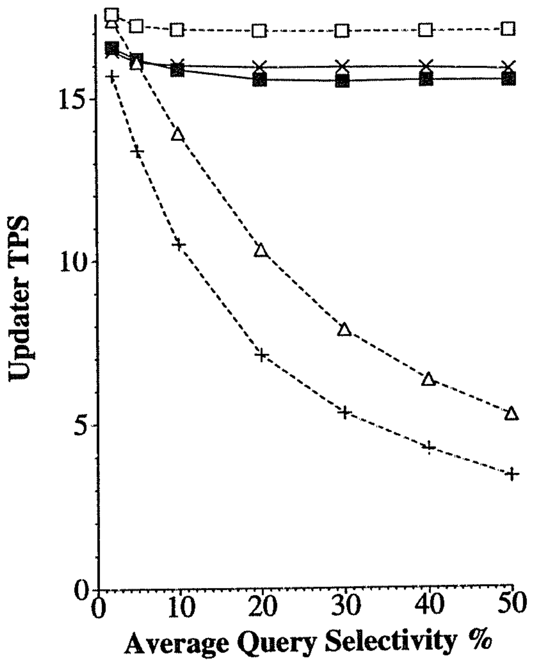


Figure 3.7: Updater Throughput

Clustered Query Workload with Write-One Policy 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.

Turning to the other curves in Figures 3.6 and 3.7 we see that GO processing 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 3.7 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 GO processing. 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 3.8 shows this clearly by presenting the throughput of MV2PL with several different cache sizes *relative* to the throughput of GO processing. 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 GO processing drops significantly. In Figure 3.8, 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

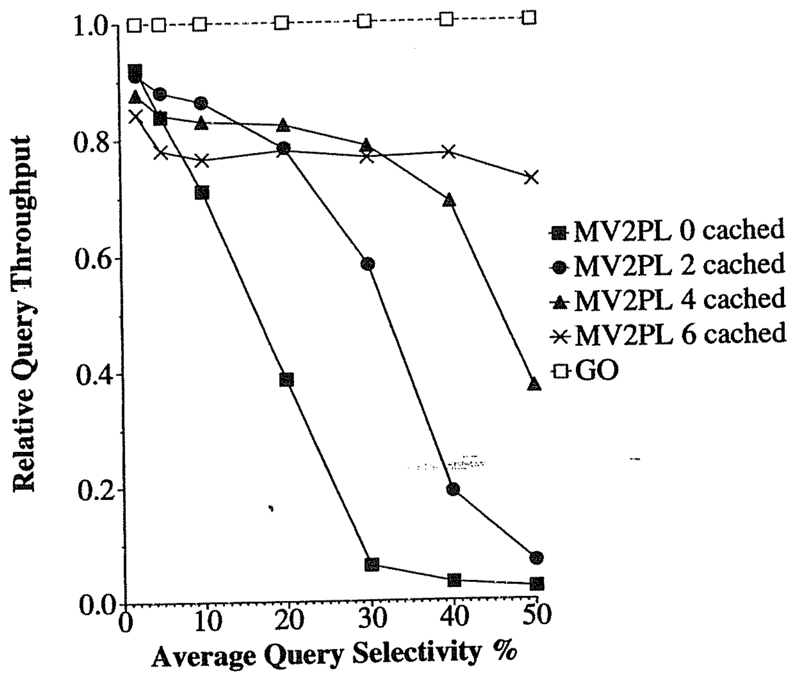


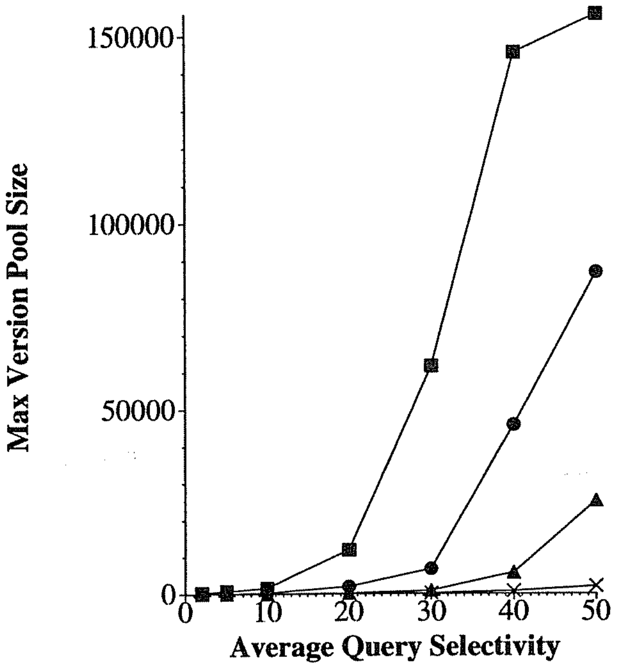
Figure 3.8: Relative Query Throughput
Clustered Query Workload with Write-One Policy

curve drops off at about 30% selectivity. The 6 cache-size curve is almost flat up to 50% selectivity[5]; virtually all of the prior versions accessed here are still in the on-page caches, so there are few, if any, version pool accesses.

The preceding explanations of performance are supported by Figures 3.9 and 3.10, which show the maximum version pool size during the simulation and the average number of (logical) pages read per query.[6] 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 3.9 and 3.10 to the graph in Figure 3.8, 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

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

[6]The 0 cache size curve dips slightly from 40% to 50% query selectivity in Figure 3.10. This is because so few queries finished with a cache size of 0 that the actual number is not statistically significant.



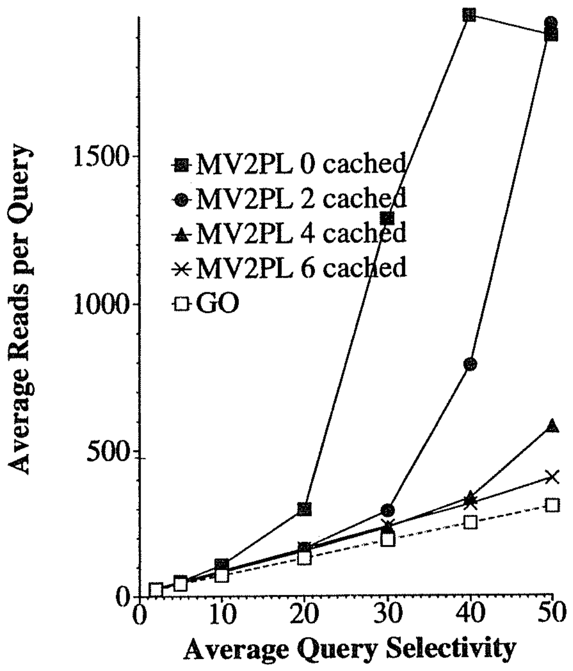


Figure 3.9: Maximum Version Pool Size **Figure 3.10: Pages Read per Query**

Clustered Query Workload with Write-One Policy

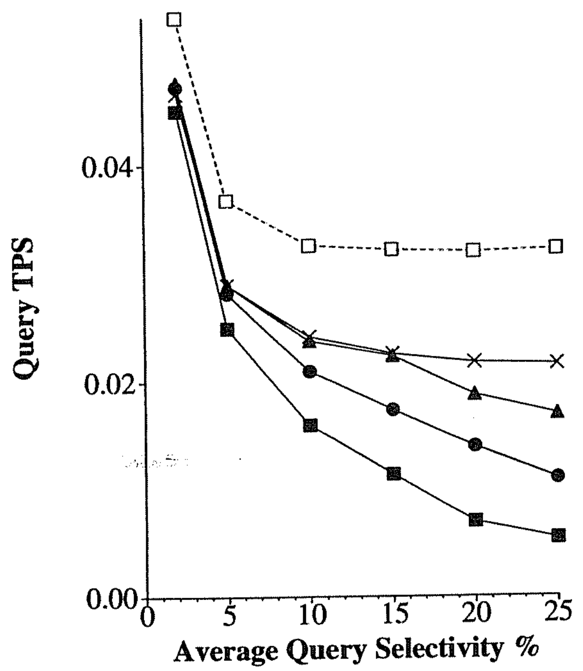
curves in Figure 3.8 fall sharply once version pool activity begins (i.e., where the curves in Figures 3.9 and 3.10 rise sharply). In addition, there is a non-linear relationship between query size and response time that leads to thrashing for queries: Increasing the query size requires that it must 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 3.9, 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.

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.

3.5.2. 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 3.11 shows query throughput over a range of average query selectivities and Figure 3.12 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 3.11 reveals that the throughput curves for GO processing 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 GO processing locking algorithm, no additional I/Os are necessary; Since the system is



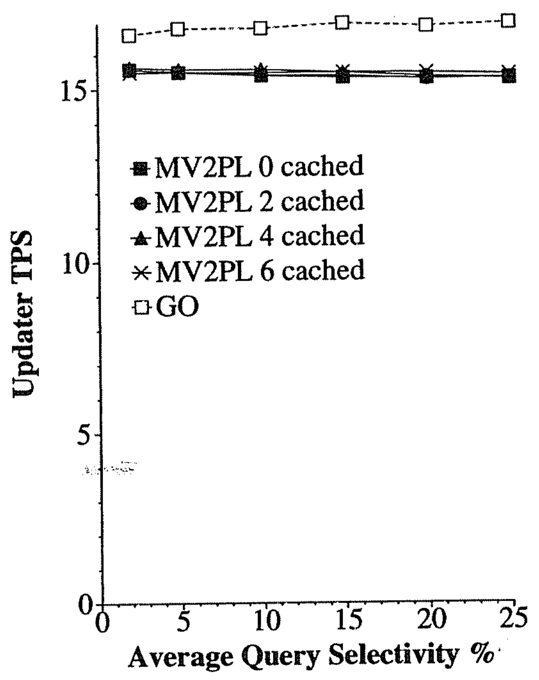


Figure 3.11: Query Throughput **Figure 3.12: Updater Throughput**

Unclustered Query Workload with Write-One Policy

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 3.11 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 GO processing 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.

3.5.3. 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 3.2, the write-one policy appends a version to the version 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

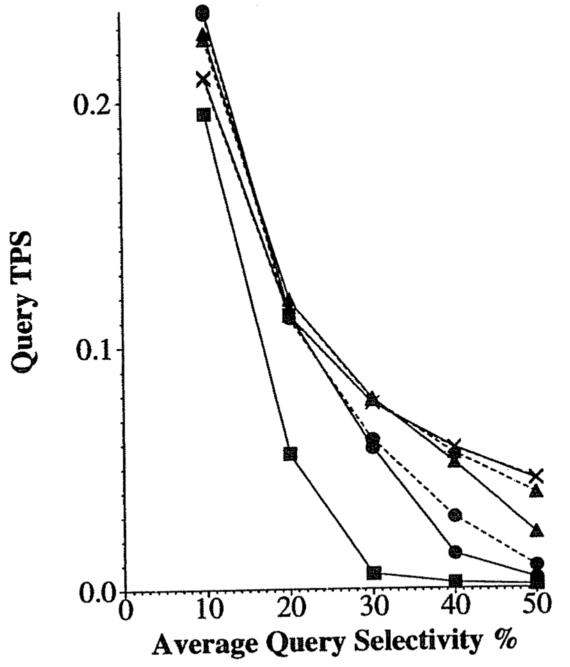


Figure 3.13: Query Throughput
(Write-one: solid lines, Write-all: dashed)

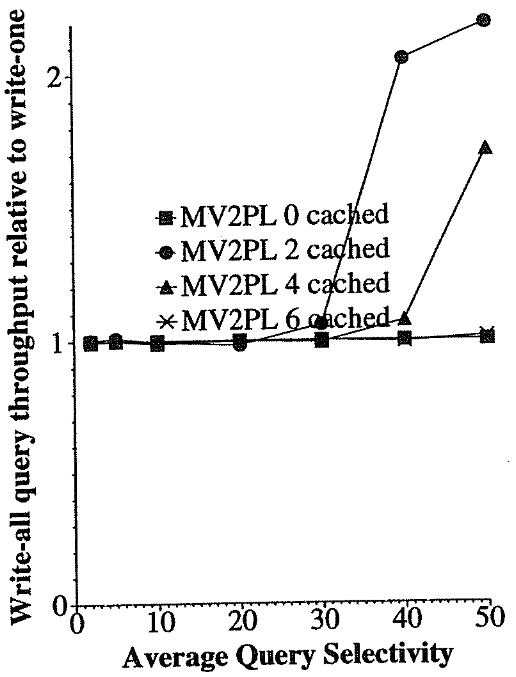
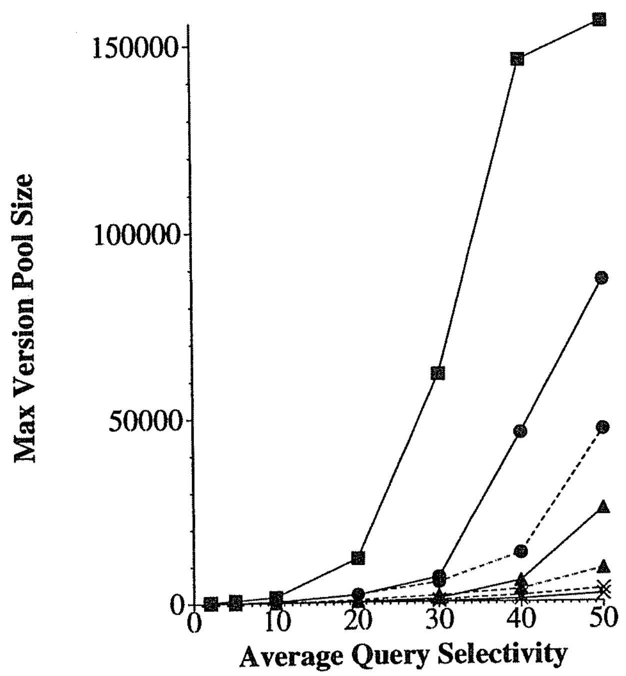


Figure 3.14: Throughput Improvement of Write-All
(Relative to Write-One)

Clustered-Scan Query Workload

least recently updated entry has not yet been appended to the version pool.

Figure 3.13 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 3.14 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 policy, so the graph is not shown.



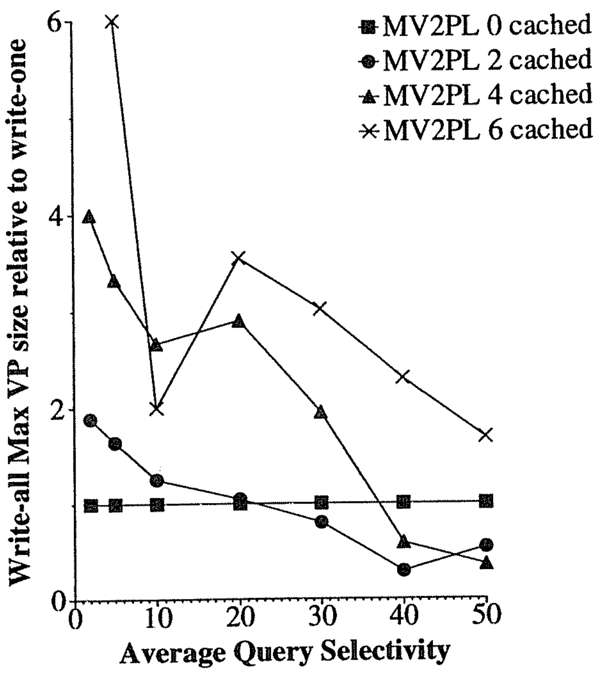


Figure 3.15: Maximum Version Pool Size
Write-one: solid lines Write-all: dashed

Figure 3.16: Max Version Pool Size of Write-All
(Relative to Write-One)

Clustered-Scan Query Workload

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 3.15 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 3.16 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.[7] 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 faster rate, and thus needs to keep around fewer prior versions.

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

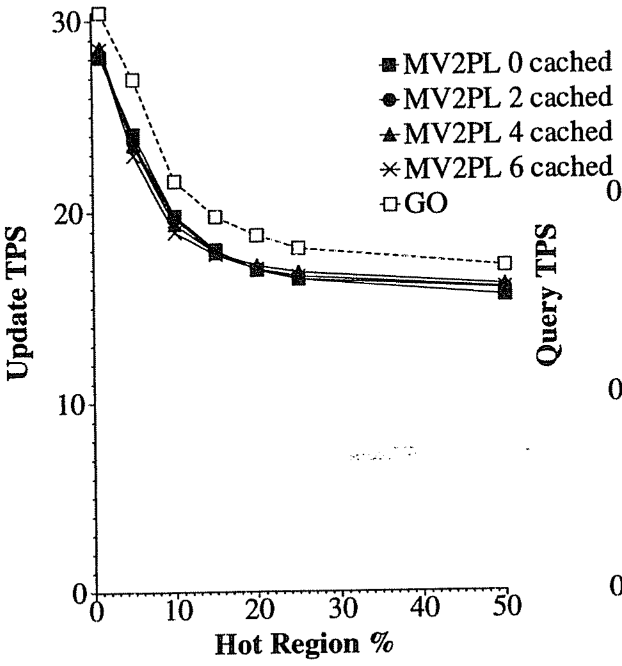
This experiment has shown that the write-all policy is superior to the write-one policy in the presence of very long queries for the clustered-index scan workload. This is a direct consequence of its improved clustering of versions for queries. For short-running queries, the write policy did not affect 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 for the sake of brevity, 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).

3.5.4. 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 3.17 shows the updater throughput for a 25% average query selectivity across a range of hot region sizes, and Figure 3.18 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 GO processing it varies from 52% to 72% across the range of hot region sizes in the figure.

In the query throughput graph (Figure 3.18), we see that for GO processing, 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



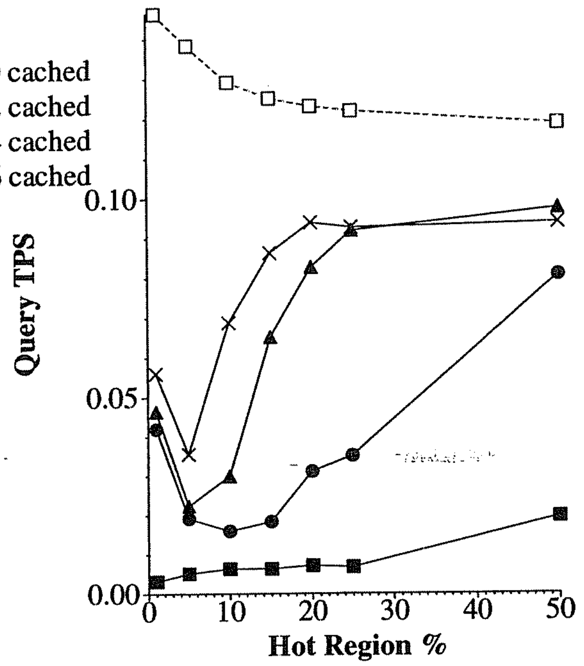


Figure 3.17: Updater Throughput **Figure 3.18: Query Throughput**

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

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 3.18, 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 3.19, 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 3.18. 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 3.19, 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

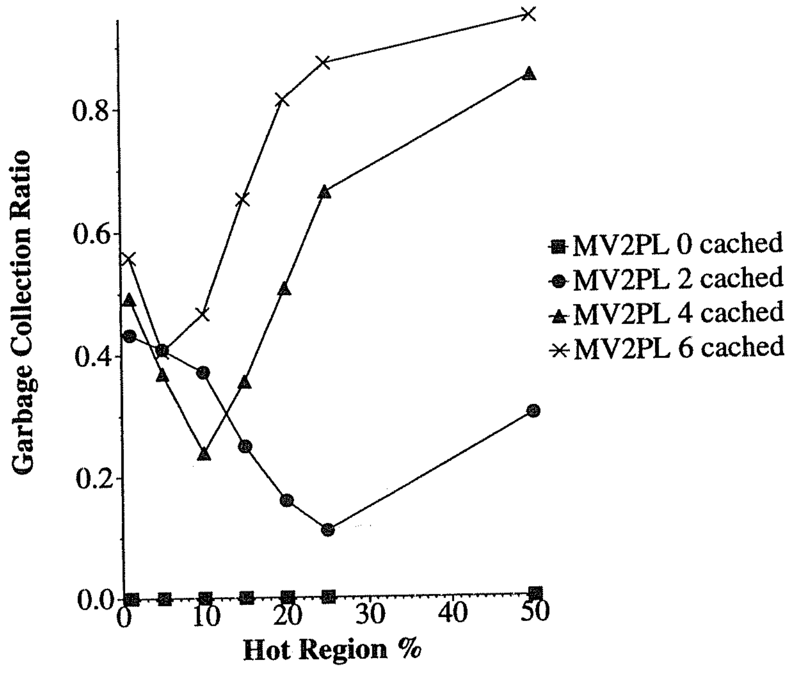


Figure 3.19: Garbage Collection Ratio
Write-One Cache Write Policy
25% Average Query Selectivity
Skewed Update Pattern (50% of updates to X% of each file)

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.

3.6. Conclusions

In this chapter, we have introduced a new storage management 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 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 GO processing and cursor-stability 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. Finally, we have also introduced the concept of view sharing as a way to further reduce storage overhead for versioning.

This chapter has proposed and studied a scheme for organizing multiple versions of data on secondary storage, without considering how the versions can be indexed. In the next chapter, we consider several options for extending single-version indexing structures to handle multiversion data.

CHAPTER 4

MULTIVERSION INDEXING ALTERNATIVES

4.1. Introduction

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.[1]

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 [Josh93]. While this approach supports index-only plans, is not compatible with the use of high performance non-2PL B+ tree concurrency control algorithms such as those proposed in [Baye77, Lehm81, Moha90]. 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 [East86, Ston87, Kolo89, Lome89, Lome90, Moha92]. With the exception of [Moha92], however, all of these

[1] 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.

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 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 chapter, we compare a range of possible multiversion indexing approaches that are designed specifically for use with MV2PL. Each of the multiversion indexing approaches that we study in this chapter are integrated with on-page caching to present a complete version placement and indexing solution for MV2PL.

The remainder of the chapter is organized as follows: In Section 4.2 we describe four multiversion indexing schemes, and in Section 4.3, we describe the simulation model that we will use to compare them. In Section 4.4, 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 ou conclusions in Section 4.5.

4.2. 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: 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 Chapter 3) and the B+ tree indexing method [Baye72].

We used several criterion to select the schemes that we will be considering in this chapter. First, to be practical, we decided that the schemes should involve only relatively simple changes to proven indexing methods (i.e., we did not 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.

4.2.1. 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 [Chan82]. 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 which wrote the version. The most recent version also has a *delete timestamp* (DTS) which is the commit timestamp of the transaction which deleted the tuple; the value of the field is infinite if the tuple exists in the current database.

Figure 4.1 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 in this chapter. The tuple in Figure 4.1 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

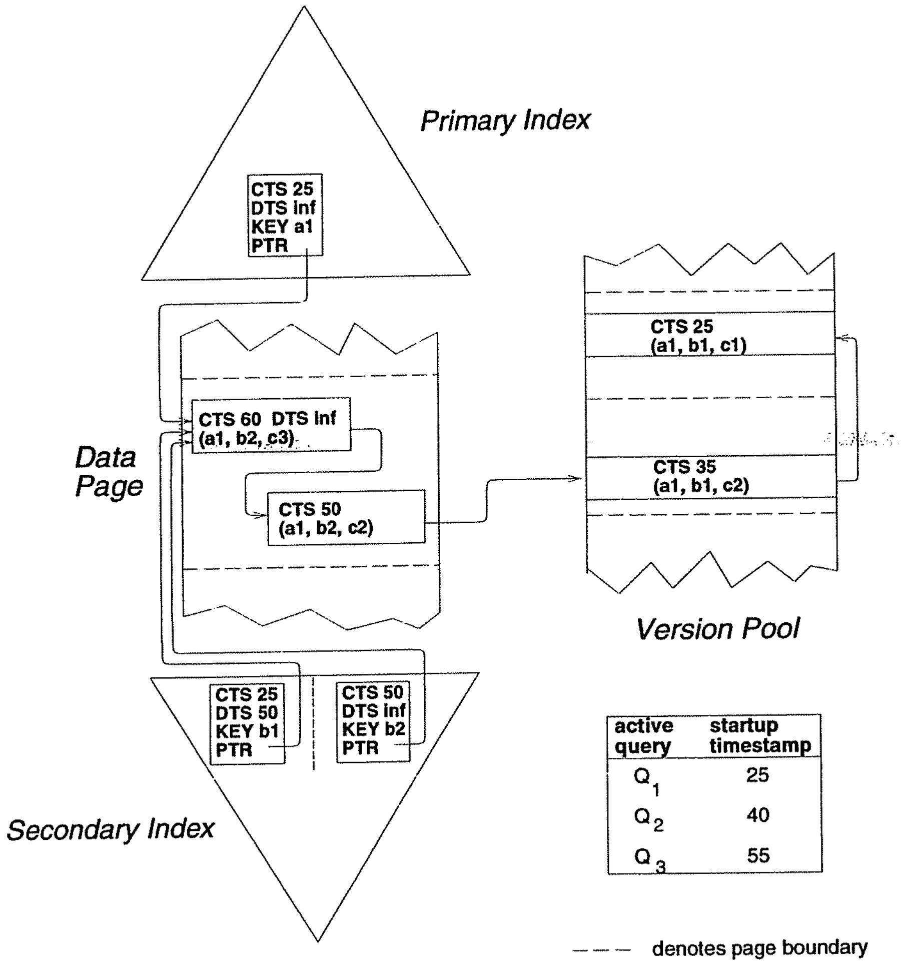


Figure 4.1: Chaining (CH)

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 which (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 4.1, the CTS and DTS fields in the secondary index entry with key b1 denote that all versions of the illustrated tuple with timestamps greater than or equal to 25 and less than 50 have b1 as 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 4.1, 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 add 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 needed if there exists a query Q ∈ {active queries} such that CTS ≤ 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 can 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 4.1, 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 to 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.

4.2.2. 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).[2] This table 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 4.2 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 4.1 under the DP scheme, a query would have to read three pages.

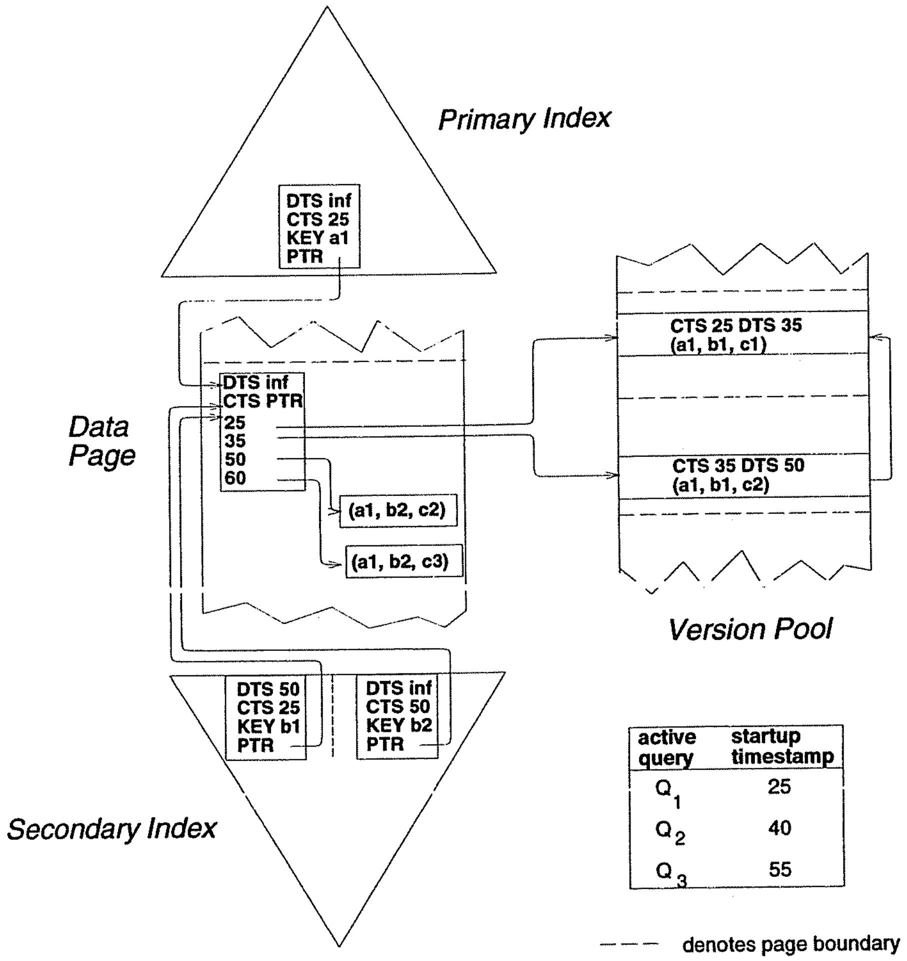


Figure 4.2: Data Page Version Selection (DP)

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

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.

4.2.3. 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 to the scheme presented in [Moha92], which is the only previously published indexing scheme for multiversion locking that we are aware of.[3] Figure 4.3 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.

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 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 4.3 by the presence of the CURR field in each secondary index entry. However, this optimization can be used only if the current version of a given tuple is always stored in a fixed location (determined when the tuple is created). Furthermore, read-only queries cannot use this optimiza-

[3]The overall versioning scheme in [Moha92] 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 schemes that we are discussing here, however.

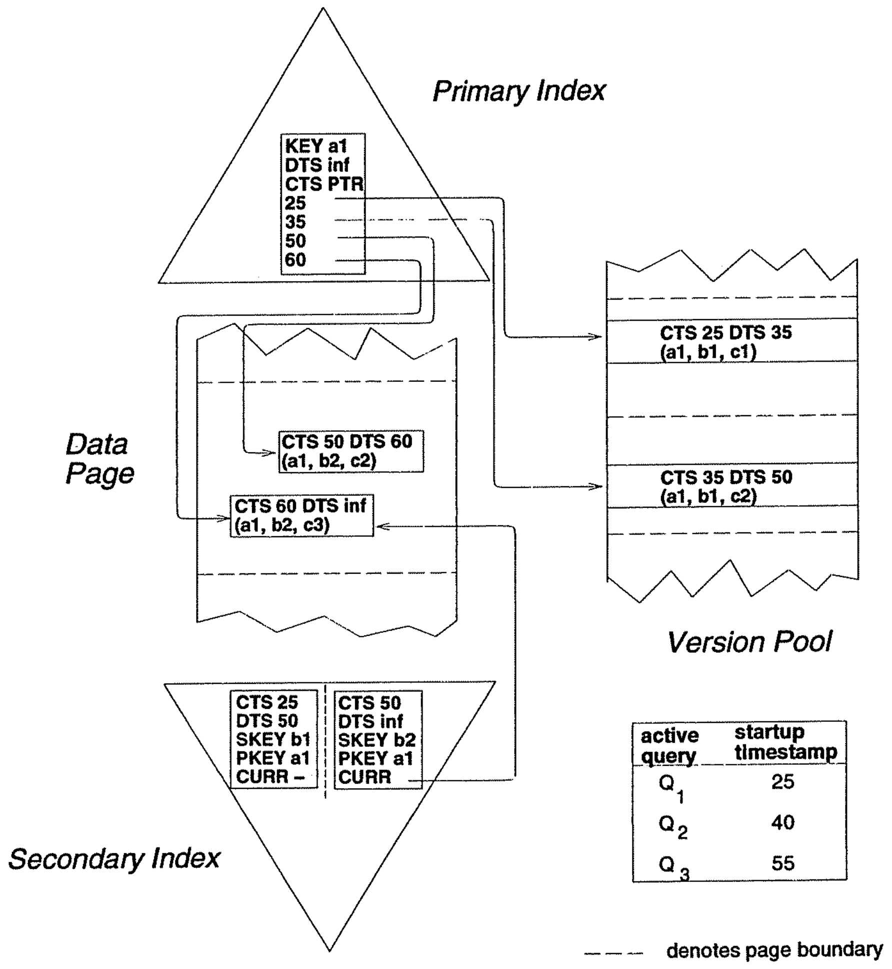


Figure 4.3: Primary Index Version Selection (PI)

tion 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.

4.2.4. 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 4.4 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 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 result of a tuple modification, each 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 4.4, 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.

4.2.5. 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

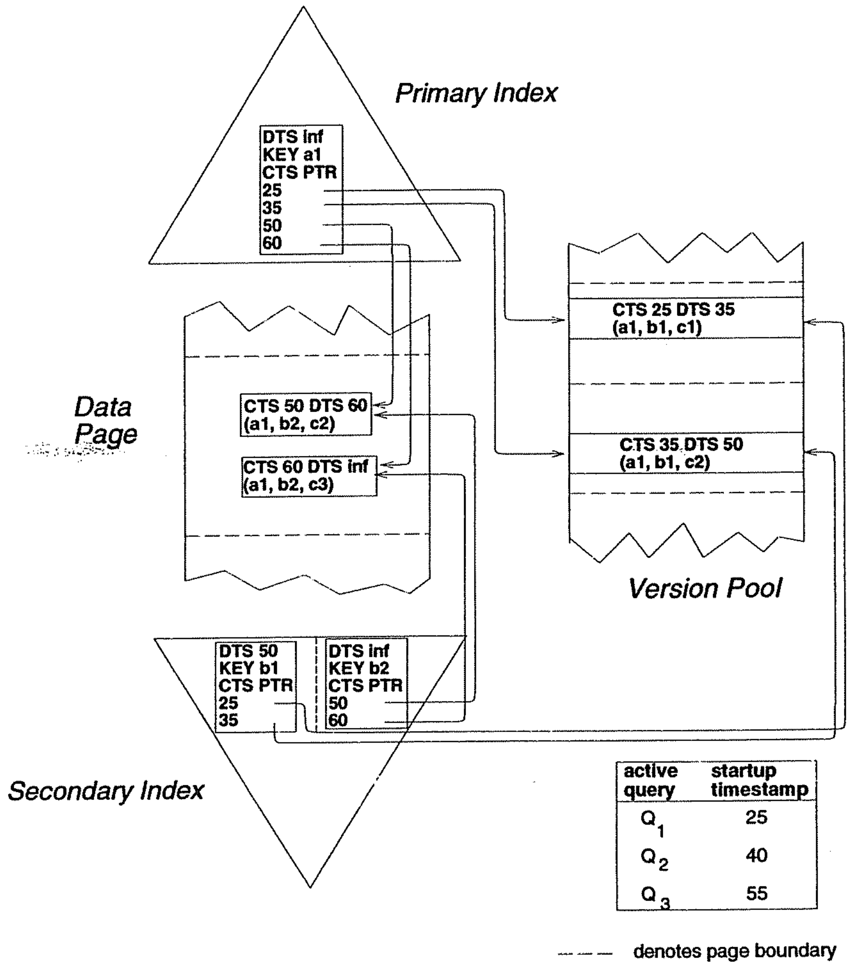


Figure 4.4: All Index Version Selection (AI)

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.

4.3. The 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 execute index scans.

In contrast to the model in Chapter 3, the model in this chapter does not capture the details associated with locking of the indexes and data; this is done 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.

As in Chapter 3, we break down the model 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 4.1 summarizes the parameters of the application model, and Table 4.2 summarizes the parameters of the system model.

4.3.1. 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.[4] 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

[4] 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 4.1: Application Model Parameters

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 $AvgSel_{Query}$.

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

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 4.2: System Model Parameters

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 $1 - ModifyProb_{Indexed}$).

4.3.2. 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[5] is used in the implementation, and *Slot-Overhead* 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 [Haas90]). 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 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.

4.3.3. 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]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.

4.4. 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 4.2. 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 4.2, 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, when 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 the index 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 4.3 and 4.4 list the settings that we use for the application model and 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 increased. Ultimately, the system will become unstable when it can no longer handle the increased update load.

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 4.3: Values of Application Model Parameters

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

Table 4.4: Values of System Model Parameters

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.

4.4.1. 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 50% average selectivity, and an update mix consisting of 60% tuple modifications, 20% inserts, and 20% deletes. 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 4.5 through 4.7 illustrate the results of this experiment. Figure 4.5 shows the average I/O cost of a clustered index scan, Figure 4.6 shows the I/O cost to insert a tuple into the database, and Figure 4.7 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 4.5, 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 activity is increased (and as query activity is correspondingly decreased).[6] 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

[6] At an arrival rate of 4 UPS (update operations per second), only about 8% of 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% of the replaced pages had to be cleaned.

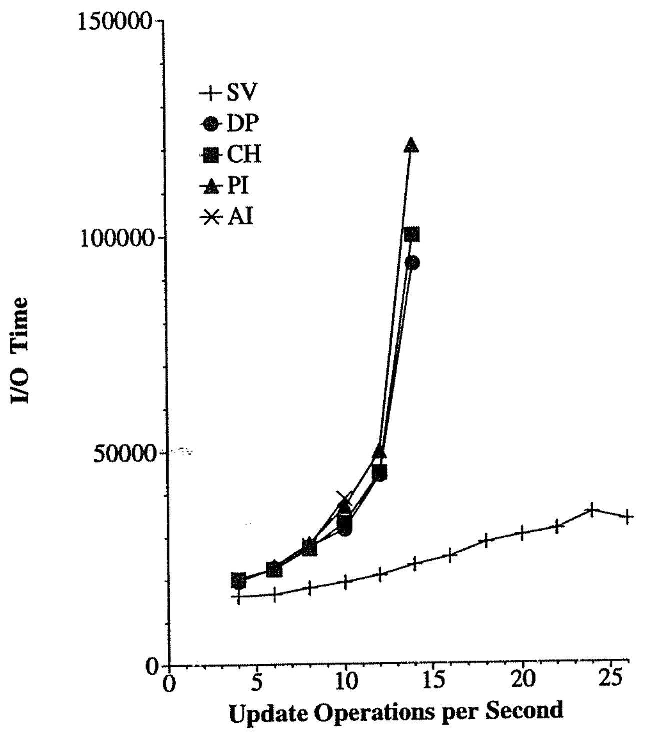


Figure 4.5: Clustered Index Scan
(*NumBuffers* =500, *CacheFrac* = 0%)

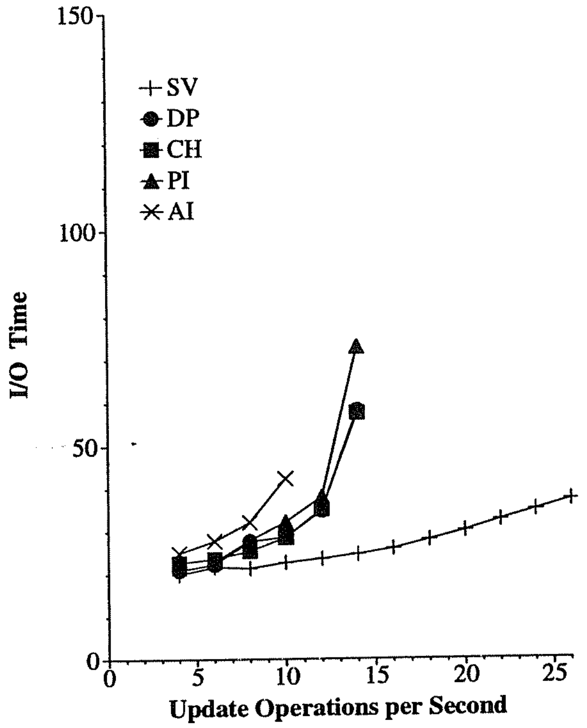


Figure 4.6: Tuple Insert
(*NumBuffers* =500, *CacheFrac* = 0%)

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 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 4.5, CH required a second version pool access to retrieve a given tuple less than 1% of the time. Likewise, we 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 pags 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 4.6. 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 smaller 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 4.6, 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 4.7. 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 rela-

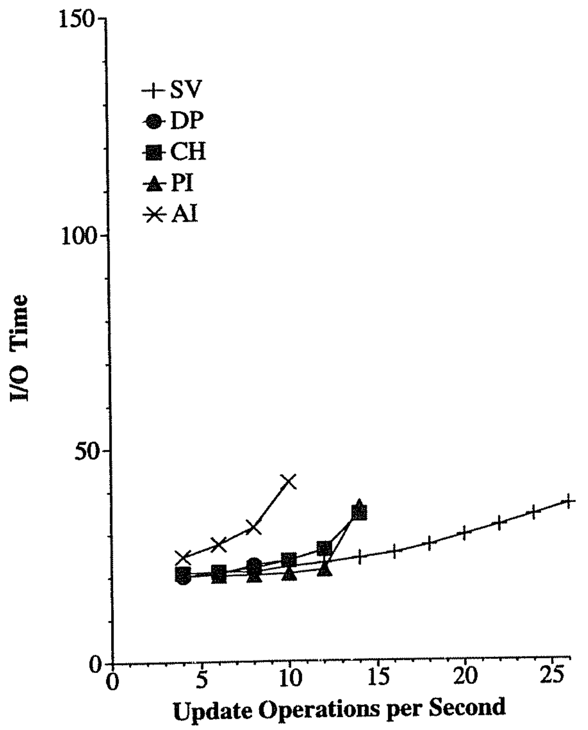


Figure 4.7: Modify Non-Indexed Attribute
(*CacheFrac* = 0%, *NumBuffers*=500)

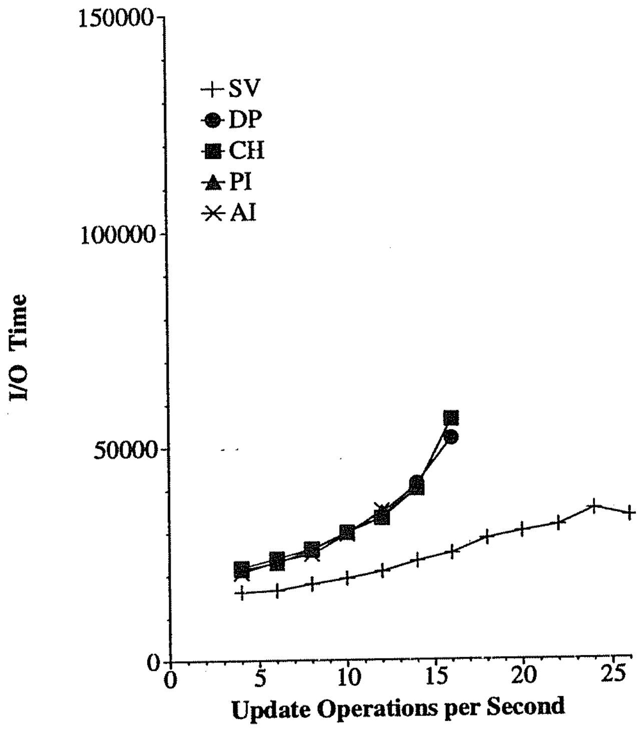


Figure 4.8: Clustered Index Scan
(*CacheFrac* = 10%, *NumBuffers*=500)

tion 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.[7] 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 4.7 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 4.7 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.

[7]PI's cost dips slightly below that of SV in Figure 4.7 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.

4.4.2. 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 10% of each data page. Figure 4.8 shows the I/O cost of clustered index scan queries with on-page caching. By comparing this figure to Figure 4.5, 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.

In order to highlight the most important results, 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 migrating a version to

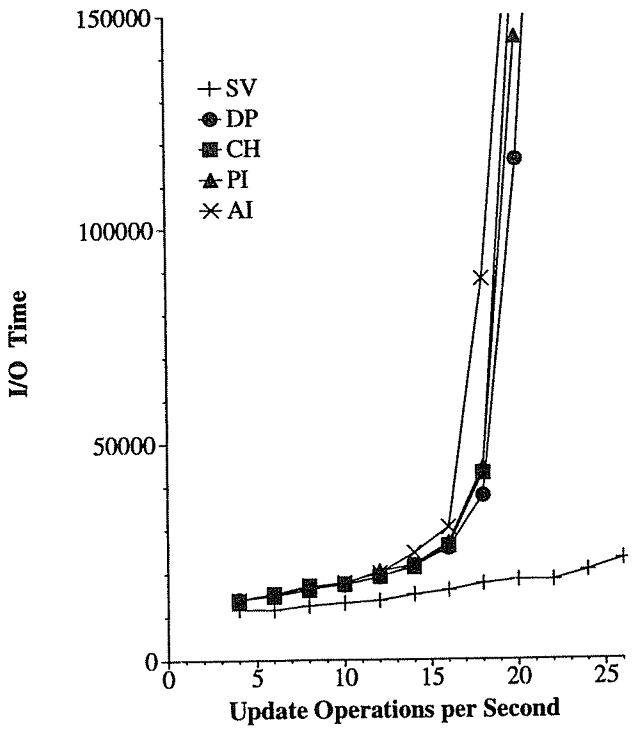


Figure 4.9: Clustered Index Scan
(*CacheFrac* = 0%, *NumBuffers*=800)

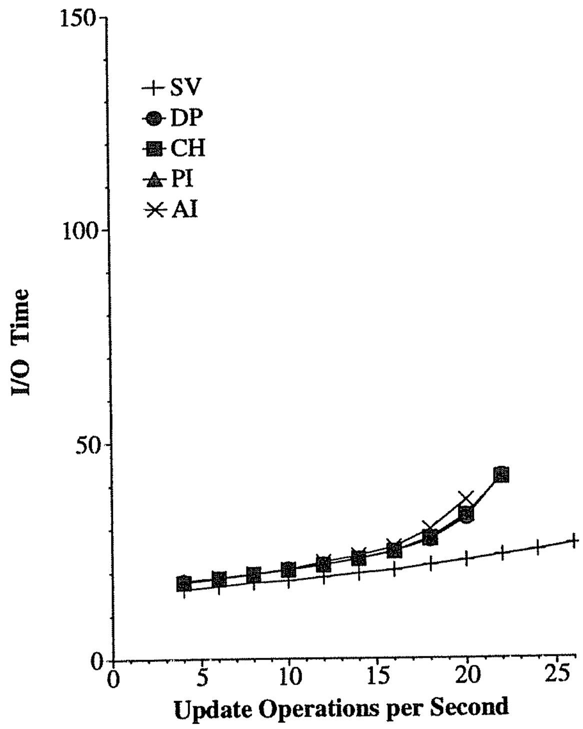


Figure 4.10: Tuple Insert
(*CacheFrac*=0%, *NumBuffers*=800)

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.

4.4.3. 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 4.9, we show the average I/O cost of clustered index scan queries, and in Figure 4.10, 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 4.10 to Figure 4.6, 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. At an update arrival rate of 10 UPS, the total index size ranged from 389 pages for SV to 526 pages for AI, while at a rate of 20 UPS, the total index size ranged from 389 pages for SV to 836 pages for AI. 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 4.9, 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. For the sake of brevity, 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.

4.4.4. 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 large enough to hold all, or nearly all, of the index pages. 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 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 4.11 and 4.12 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 16% in the figures, even though it typically pays 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 4.11 shows the average I/O cost of an

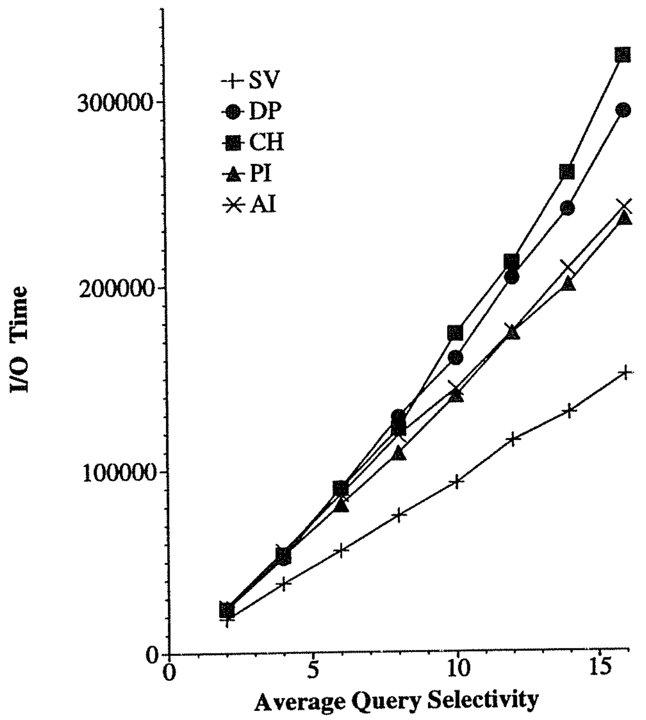


Figure 4.11: Unclustered Index Scan
(*ArrivalRate*=20/*sec. CacheFrac* = 0%, *NumBuffers*=1000)

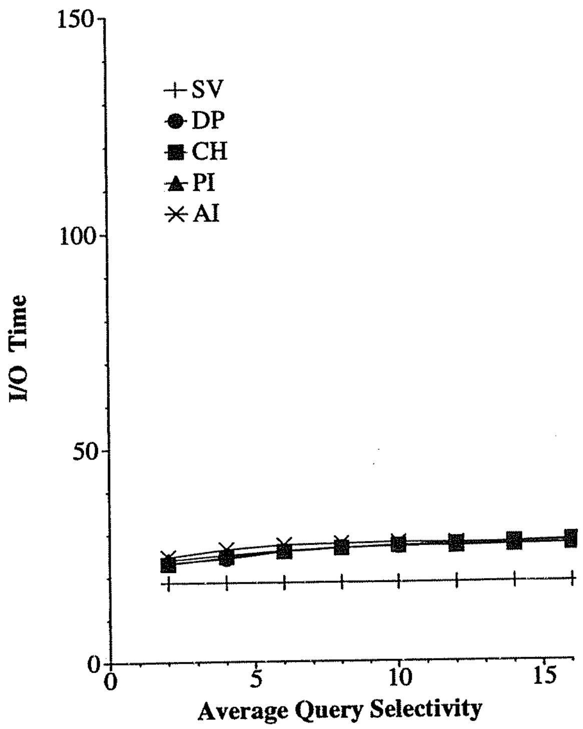


Figure 4.12: Tuple Insert
(*ArrivalRate*=20/*sec., CacheFrac* = 0%, *NumBuffers*=100C

unclustered index scans, and Figure 4.12 shows the average I/O cost to insert a tuple in the database.

In Figure 4.11, 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 4.12, 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 4.12 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.

4.4.5. 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 4.2, 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., when 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.

4.5. Conclusions

In this chapter, 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.

CHAPTER 5

MULTIVERSION QUERY LOCKING

5.1. Why Another Algorithm?

A drawback to MV2PL is the storage cost that it imposes, as well as the additional costs for accessing prior versions and for copying objects before they are updated (assuming that in-place updates are employed). These costs were quantified for MV2PL in the preceding chapters. Towards the goal of making versioning more affordable, a new multiversion two-phase locking algorithm, *multiversion query locking* (MVQL), is introduced in this chapter. MVQL enables a tradeoff to be made between query consistency and performance; it supports several weaker forms of consistency for queries than that provided by MV2PL. We review these forms of consistency in Section 5.2, but we wish to emphasize here that they still guarantee that queries see transaction-consistent data. This is in contrast to approaches that avoid versioning altogether, instead allowing queries to see transaction-inconsistent data. For example, under cursor-stability locking, queries may release locks before acquiring new ones (violating the two-phase rule), and under GO processing [Pira90], queries do not obtain any locks at all (except latches to guarantee page consistency). In the terminology of [Gray79], the former provides degree 2 consistency, and the latter, degree 1. Other examples include epsilon-serializability algorithms, which accept inconsistent schedules as long as they are within some number of inversions from a serializable schedule [Wu92].

Because MVQL provides weaker consistency than MV2PL, it can allow queries to read more recent versions of objects. Performance savings are gained because it is typically less efficient for queries to read older versions of objects rather than younger (or current) ones. Depending on the particular storage organization employed, this may be true for any of the following reasons:

(1) If the current versions of objects are clustered together, accessing an older version of an object will degrade sequential scan performance that would otherwise be available using prefetch.

(2) If the versions of an object are chained in reverse chronological order (as in [Chan82]), accessing an older version will require additional I/O operations.

(3) Using older versions to construct a query's view will require that additional prior versions be retained for the query (thus delaying their garbage collection and increasing storage cost).

All of these effects were evident in the simulation results presented so far in this thesis.

The remainder of this chapter is organized as follows: Section 5.2 reviews the various forms of consistency that are provided by MV2PL and MVQL. Section 5.3 presents the new MVQL algorithm as a generalization of MV2PL. Section 5.4 describes the simulation model used to study the performance of MVQL. Section 5.5 presents the results of experiments that compare MVQL to MV2PL in terms of update transaction performance, query performance, and storage cost. Lastly, Section 5.6 presents our conclusions for this chapter.

5.2. Forms of Query Consistency

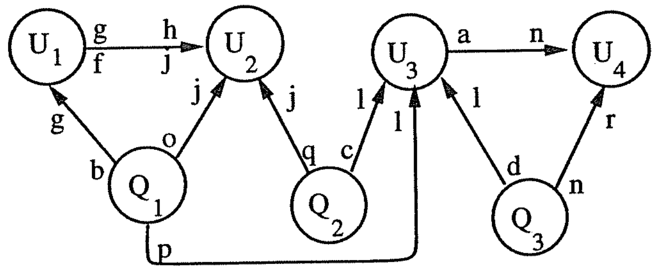
In the introduction of this chapter, we argued that performance advantages may be gained by relaxing the level of consistency provided to queries. In this section, we review four forms of consistency which all guarantee that queries see a transaction-consistent database: strict consistency, strong consistency [Garc82], weak consistency [Garc82], and update consistency. An underlying requirement that we impose is that the execution of update transactions alone must be serializable; this is guaranteed in MV2PL and MVQL by having update transactions run using dynamic 2PL. This prohibits serialization graph cycles that contain only update transactions (*update transaction cycles*).[1] Throughout the chapter we assume that update transactions read all objects before modifying them. We now give definitions for the various forms of consistency of interest here, proceeding in decreasing order of strictness. Table 5.1 summarizes the consistency forms that are being defined.

A query is said to see *strict consistency* if it is serializable with respect to all transactions, and it observes a serial order of update transactions which agrees with the order in which they committed. This form of consistency is characterized by an acyclic serialization graph where the order of update transactions is consistent with their commit order. Having the update transactions observe 2PL guarantees that, in the subgraph consisting of

[1]A serialization graph consists of nodes, which represent transactions, and edges, which represent constraints on equivalent serial orderings. A path $(T_i, \ldots, T_j)$ in the graph means that transaction T_i must come before transaction T_j in any equivalent serial order. A directed edge of the form (T_1, T_2) is placed in the graph if either T_2 attempts to read a version written by T_1, T_2 attempts to create a version of an object that will replace one read by T_1, or T_1 reads a version that was already replaced by T_2.

Forms of Consistency		
Name	*Description*	*Serialization Graph Constraints*
strict consistency	each query sees a serial order of update transactions that is consistent with their commit order	update transaction (partial) order is consistent with their commit order; cycles are prohibited
strong consistency	each query sees a common serial order of update transactions (not necessarily consistent with their commit order)	cycles are prohibited
weak consistency	each individual query sees a serial order of update transactions (not necessarily the same as other concurrent queries)	single query cycles are prohibited; update transaction cycles are prohibited
update consistency	each query sees a transaction-consistent database (i.e., it serializes with all update transactions that it sees)	single query cycles are prohibited if they cannot be broken by removing a single read-write edge between update transactions; update transaction cycles are prohibited

Table 5.1: Forms of Consistency



	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
U_1					R(X_0)	R(Y_0)	W(X_1)											
U_2								R(X_1)	R(Y_0)	W(Y_1)								
U_3	R(W_0)										R(Z_0)	W(Z_1)						
U_4													R(W_0)	W(W_1)				
Q_1		R(X_0)													R(Y_0)	R(Z_0)		
Q_2			R(Z_0)														R(Y_0)	
Q_3				R(Z_0)														R(W_0)

Figure 5.1: A Schedule and Serialization Graph Illustrating Strict Consistency for Queries

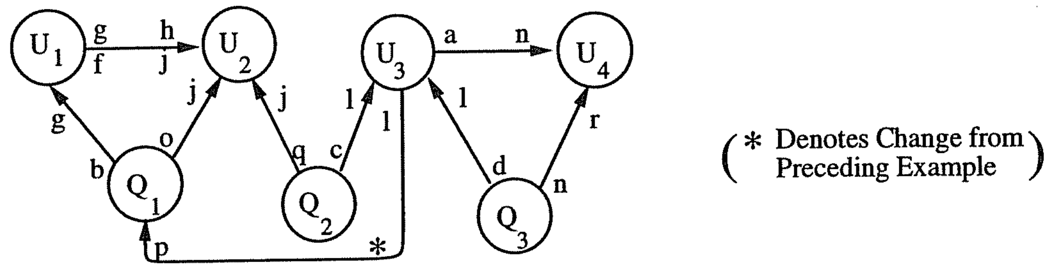
only the actions of update transactions (and not those of queries), the partial order of update transactions will agree with their commit order. The addition of query read actions further constrains the partial order so that it may no longer agree with the commit order; strict consistency algorithms prevent such a disagreement by assigning appropriate versions to query read steps.

The schedule and corresponding serialization graph in Figure 5.1 provide an example of strict consistency. The schedule shows the operations of four update transactions and three queries, with time progressing from left to right. We assume that the last operation of each transaction in the schedule also marks its commit point. To identify each operation in the schedule, we label it with a lower case letter; these letters are then used to label each edge in the serialization graph with the operations that generated the edge. For example, the edge between Q_1 and U_1 was generated because Q_1 read X_0 (step b) and U_1 wrote X_1 (step g). In the serialization graph, we observe that the partial order of update transactions induced by the serialization graph is indeed consistent with their actual commit order (e.g., $U_1 < U_2$ and $U_3 < U_4$). The cost of providing strict consistency is the cost of accessing the prior versions of the data items W, X, Y, and Z (instead of the current versions) and the cost of retaining these versions until all of the queries complete.

We illustrate the remaining forms of consistency by incrementally modifying the example in Figure 5.1 by substituting current versions for prior ones in one or more query read operations. This will have the effect of reversing the direction of certain edges in the serialization graph. To highlight these changes, we place an asterisk next to each such reversed edge. As the constraints on consistency are relaxed, we will see that queries are allowed to access more recent data.

The next form of consistency relaxes strict consistency by eliminating the requirement that the serial order of update transactions be consistent with their commit order. A query is said to see *strong consistency* [Garc82] if it is serializable with respect to all transactions.[2] This form of consistency is characterized by an acyclic serialization graph, and is provided by algorithms that guarantee multiversion serializability [Bern83, Papa84, Hadz85]. Since the previous restriction on the commit ordering of update transactions is relaxed here, strong consistency may produce apparent anomalies in query results if users are somehow cognizant of the commit order of update transactions. The schedule and corresponding serialization graph in Figure 5.2 provide an example. The schedule differs from the one in the strict consistency example in that Q_1 reads Z_1 instead of Z_0. This reduces the cost of

[2]This definition of strong consistency is slightly different than the one presented in [Garc82]. In their definition, a strong consistency query is required to serialize only with the update transactions and other strong consistency queries. We discuss how MVQL can support this definition of strong consistency in Section 5.3.1.

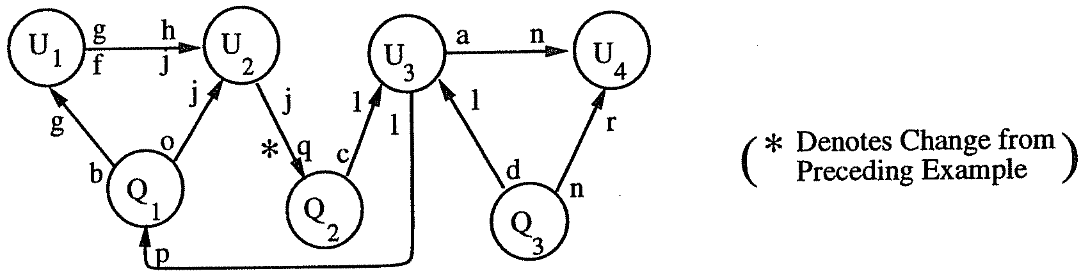


	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
U_1					$R(X_0)$	$R(Y_0)$	$W(X_1)$											
U_2								$R(X_1)$	$R(Y_0)$	$W(Y_1)$								
U_3	$R(W_0)$										$R(Z_0)$	$W(Z_1)$						
U_4													$R(W_0)$	$W(W_1)$				
Q_1		$R(X_0)$													$R(Y_0)$	$R(Z_1)$		
Q_2			$R(Z_0)$														$R(Y_0)$	
Q_3				$R(Z_0)$														$R(W_0)$

Figure 5.2: A Schedule and Serialization Graph Illustrating Strong Consistency for Queries

executing Q_1, and it potentially allows Z_0 to be garbage-collected before Q_1 completes (i.e., because Q_1 will never need it). As a result of this change, Q_1 serializes after U_3, but before U_1 and U_2; this is despite the fact that U_1 and U_2 actually committed before U_3. Q_1 is allowed to see this order since neither U_1 nor U_2 execute any conflicting operations with U_3, and no other query has seen an order that is contradictory.

The next form of consistency relaxes strong consistency by allowing each query to serialize *individually* with the set of update transactions. A query is said to see *weak consistency* if it is serializable with respect to update transactions, but possibly not with respect to other queries. This form of consistency, which was first introduced in [Garc82] for use in replicated databases, still ensures that queries see transaction-consistent data. However, it permits cycles in the serialization graph that contain multiple queries plus one or more update transactions (*multiple-query cycles*). Cycles involving a single query and one or more update transactions (*single-query cycles*), and cycles involving only update transactions (*update transaction cycles*), are both still prohibited. The queries in a multiple-query cycle see mutually inconsistent orderings of the update transactions in the cycle (i.e., one query will perceive a different serial ordering of update transactions than another query); the relative order of two update transactions may be transposed if they have not issued any conflicting operations. The schedule and



	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
U_1					$R(X_0)$	$R(Y_0)$	$W(X_1)$											
U_2								$R(X_1)$	$R(Y_0)$	$W(Y_1)$								
U_3	$R(W_0)$										$R(Z_0)$	$W(Z_1)$						
U_4													$R(W_0)$	$W(W_1)$				
Q_1		$R(X_0)$													$R(Y_0)$	$R(Z_1)$		
Q_2			$R(Z_0)$														$R(Y_1)$	
Q_3				$R(Z_0)$														$R(W_0)$

Figure 5.3: A Schedule and Serialization Graph Illustrating Weak Consistency for Queries

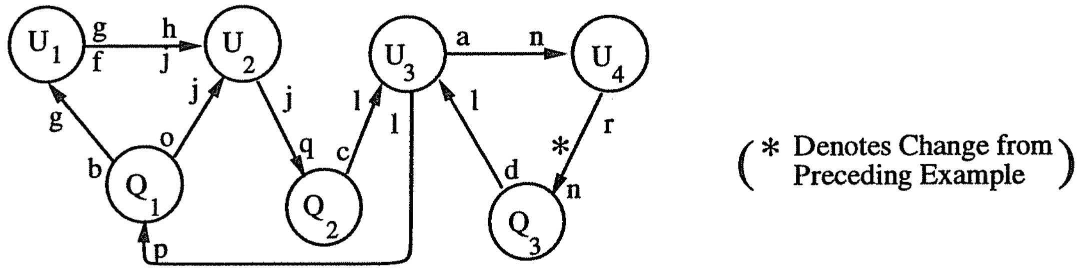
corresponding serialization graph in Figure 5.3 illustrate this form of consistency. The schedule differs from that of Figure 5.2 in that Q_2 reads Y_1 rather than Y_0, thus reducing Q_2's cost and potentially allowing Y_0 to be garbage-collected earlier than it would have been under strong consistency. This execution introduces a multiple-query cycle involving Q_1, Q_2, U_1, U_2, and U_3. This cycle indicates that Q_1 has seen the serial ordering (U_3, Q_1, U_1, U_2), while Q_2 has seen the ordering (U_1, U_2, Q_2, U_3). Thus, U_1 and U_3 (as well as U_2 and U_3) are ordered differently in the two queries' observed schedules.

The last form of consistency considered here relaxes weak consistency by also allowing certain single-query cycles. The effect of this relaxation will be described shortly. A query is said to see *update consistency* if it serializes with the set of update transactions that produced values that are seen (either directly or indirectly) by the query. Even though an update consistency query may not serialize with the complete set of update transactions, it is guaranteed to see a transaction-consistent database state. Recall that a transaction-consistent database is assumed to satisfy a set of static integrity constraints, and each update transaction is assumed to take the database from one transaction-consistent state to another (possibly through one or more inconsistent intermediate states) [Gray76]. In order to observe a transaction-consistent database, a query must not see the partial effects of any update transactions; for each update transaction, it must see either all of its effects or none of its effects.

Update consistency permits multiple-query cycles in the serialization graph, as well as permitting single-query cycles if they can be broken by removing a read-write edge between two update transactions. An edge (T_1, T_2) is a read-write edge if it was formed due to a read operation by T_1 followed by a conflicting write operation by T_2. Of course, it is important to remember that, as for all forms of consistency here, the full serialization graph is not permitted to contain update transaction cycles.

The schedule and corresponding serialization graph in Figure 5.4 illustrate update consistency. A single-query cycle is introduced between U_3, U_4, and Q_3 because Q_3 now reads W_1 rather than W_0. This cycle is allowed under update consistency (but not under higher forms of consistency) because it does not persist when the read-write edge from U_3 to U_4 is removed. In the execution, Q_3 sees all of the effects of U_4, but none of the effects of U_3. The query nevertheless sees a transaction-consistent database since the output of U_3 has no bearing on the the execution of U_4.

While update consistency guarantees that queries see a transaction-consistent database state, it allows them to see a state that might not be logically consistent with the current state of the database. For example, in Figure 5.4, Q_3 sees a state of the database that would have existed if U_3 had never executed (or if it had aborted). Because



	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
U_1					R(X_0)	R(Y_0)	W(X_1)											
U_2								R(X_1)	R(Y_0)	W(Y_1)								
U_3	R(W_0)										R(Z_0)	W(Z_1)						
U_4													R(W_0)	W(W_1)				
Q_1		R(X_0)													R(Y_0)	R(Z_1)		
Q_2			R(Z_0)														R(Y_1)	
Q_3				R(Z_0)														R(W_1)

Figure 5.4: A Schedule and Serialization Graph Illustrating Update Consistency for Queries

of the read-write conflict between U_3 and U_4, it is not possible to assume that U_3 was executed logically after U_4; U_3 might have an entirely different effect if it were executed after U_4. For example, suppose that U_3 is a transaction that adds a passenger (Mr. Smith) to a flight manifest (Z), and U_4 is a transaction that registers the flight's departure in the relevant flight record (W). Furthermore, assume that a transaction will not add a passenger to a flight manifest if the flight has been registered as departed (checking this requirement is the source of the read-write conflict). In this scenario, it will appear to query Q_3 that the flight has departed and that there is no Mr. Smith registered as a passenger (which is indeed a potential transaction-consistent database state). Also, the fact that the flight has departed would seem to imply that Mr. Smith could not later be added to the passenger list; however, later queries will reveal that Mr. Smith was indeed a passenger on the flight. Despite the presence of this type of anomaly, update consistency may be useful in situations where degree 1 or 2 consistency is insufficient (e.g., checking integrity constraints).

To the best of our knowledge, almost all previously proposed multiversion concurrency control algorithms provide only strict consistency for queries. The only exception that we are aware of is distributed MV2PL, where weak consistency arises among queries at different sites due to inconsistent global state information [Chan85]. In contrast, MVQL deliberately introduces weaker forms of consistency among queries by allowing them to read newer versions of data for performance reasons. In the next section, we describe the MVQL algorithm and show how it can be used to provide queries with either update, weak, strong, or strict consistency (as desired by a given application).

5.3. Multiversion Query Locking (MVQL)

In this section, we describe the basic MVQL algorithm, and in the next section, we describe some refinements to the basic algorithm. MVQL generalizes the startup timestamp in MV2PL to provide queries with the range of consistency forms discussed in the previous section. The startup timestamp assigned to a query in MV2PL is used to define the transaction-consistent state that it sees. More specifically, it serves to concisely divide the set of update transactions into two subsets, the set of update transactions which come *before* the query in the serial order, and the set of update transactions which come *after* the query; we will refer to these sets of update transactions as the query's BEFORE set and AFTER set, respectively. The query sees the correct state by always reading the most recent version of an object written by a transaction that belongs in its BEFORE set (i.e., by one whose commit timestamp is less than or equal to the query's startup timestamp).

Under weaker forms of consistency, each query defines its own interpretation of the serial order. As a result, a single-valued timestamp is insufficient to represent the AFTER and BEFORE sets of a query in MVQL. Rather, one of the two sets must be represented explicitly. (It is not necessary to represent both explicitly since they are complements of each other.) The details of set representation will be discussed shortly. To locate the correct version to read of a given tuple, a query scans the tuple's VST (or traverses its version chain) until it reaches the most recent version that was written by a committed update transaction not in its AFTER set. Because update transactions follow 2PL, there is a total ordering of versions; thus, there is no ambiguity as to which version should be chosen.

As discussed in the beginning of this chapter, the goal of adopting weaker forms of consistency is to allow queries to read more recent data, thus reducing the cost of versioning. MVQL will therefore place an update transaction in a query's AFTER set only when necessary to prevent a violation of the desired form of consistency. A query always begins with an empty AFTER set. For purposes of explanation, the rules for placing update transactions in the AFTER sets of queries for each form of consistency will be presented in the order: strict, update, weak, and strong. Recall that Table 5.1 summarizes these forms of consistency. In the next subsection we describe how these rules may be efficiently implemented.

5.3.1. Varying Consistency Levels

For *strict consistency*, which is the most restrictive form, all update transactions running during any portion of a query's lifetime are placed into the query's AFTER set. This makes the algorithm identical to MV2PL.

For *update consistency*, which is the least restrictive form, an update transaction U is placed into the AFTER set of a query Q under any of the following conditions (which comprise Rules 1-3):

[3] MVQL is therefore directly compatible with all of the indexing schemes considered in the preceding chapter.

(1) U attempts to write lock an object that has been already read by Q.

(2) Q attempts to read an object that is currently write locked by U.[4]

(3) U reads an object version (always the current one) that was written by another update transaction U', and U' is currently a member of Q's AFTER set.

Recall that update consistency guarantees that queries do not see the intermediate effects of an update transaction; the query sees either all or none of its effects. Rules 1 and 2 guarantee that a query will not see the partial effects of an update transaction directly, and Rule 3 guarantees that it will not see them indirectly. It should be noted that Rule 3 is sufficient to recognize indirect partial effects passed through any number of update transactions. This is true for two reasons. First, the algorithm always recognizes that an update transaction should come after a query in the serial order while the update transaction is still uncommitted; thus, if a committed update transaction is at some point not in a query's AFTER set, it never will be. Second, an update transaction reads only committed data. Thus, when an update transaction reads a version, it knows immediately for each active query whether the version's creator came before or after the query.

For *weak consistency*, which is the next more restrictive form, the following rule is added to the update consistency rules (Rules 1-3):

(4) An update transaction U is placed into the AFTER set of a query Q if U overwrites an object version (always the current one) that was read by another update transaction U', and U' is currently a member of Q's AFTER set.

This rule is necessary to recognize read-write dependencies between update transactions, and along with the first three rules it prevents single query cycles in the serialization graph.[5] The additional consistency comes at the expense of making query AFTER sets larger (thus requiring that queries read older data).

For *strong consistency*, which is even more restrictive, the following rule is added to the weak consistency rules (Rules 1-4):

[4]Alternatively, it is possible instead to block Q behind U. This may make the implementation easier, and will probably not have a significant performance impact if update transactions are short. On the other hand, doing so could cause significant delays for queries if update transactions are long.

[5]Note that this rule would be unnecessary in a closed transaction workload if a read-write conflict between two update transactions is known to exist only if there is also a write-read conflict between the transactions. Also, a workload that does not satisfy this property initially might easily be modified so that it does. For example, in flights example in Section 5.2, this property would be satisfied if the transaction registering the flight's departure (U_4) was required to read the seat count (located in the flight manifest Z).

(5) A query is considered to have read an object (for the purposes of Rules 1 and 2) if either it has read the object explicitly, or if some younger query has read the object explicitly.

In other words, a read by one query is treated as though it were made by all older queries as well; in the next subsection, we describe how this may be done efficiently. With the addition of this rule, multiple query cycles are eliminated since the AFTER set of an older query will always subsume the AFTER sets of all younger queries. This prevents a path in the serialization graph from a younger query to an older query; any multiple query cycle would have to contain such a path. The avoidance of multiple query cycles means that all queries will see a consistent serial ordering of update transactions; however, this additional consistency comes at the expense of making query AFTER sets still larger (thus requiring that queries read even older data).

5.3.2. Implementing the AFTER Set Insertion Rules

Determining when an update transaction should be inserted into the AFTER set of a query under strict consistency is straightforward: When a query enters the system, all currently executing update transactions are placed into its AFTER set. All subsequently arriving update transactions are also placed into this set. Determining when the rules apply under the other forms of consistency is less straightforward, however. In order to determine when Rule 1 applies, we need a mechanism for determining whether or not an active query has read an object that an update transaction now wishes to write lock. This can be handled by adding a new, non-conflicting lock mode to the 2PL lock manager called a *read-only* lock. A query must obtain a read-only lock on each object that it reads; note that it obtains locks on objects, not on object versions. As with traditional locks, a query releases all of its read-only locks when it finishes. When granting a write lock on an object to an update transaction, the lock manager will respond with a list of the object's current read-only lock holders. The applicability of Rule 2 can be easily detected when a query obtains a read-only lock; the lock manager will respond to a read-only lock request by returning the identifier of the current write lock holder (if there is one). Furthermore, the applicability of Rule 3 may be easily checked by an update transaction since each version is stamped with the identifier of its creator. Specifically, when an update transaction reads an object, it can check to see if the creator of the current version is a member of the AFTER set of any active queries.

Rule 4, added for weak consistency, may be enforced by requiring that each query inherit the read locks of all committing update transactions in its AFTER set (converting them to read-only locks in the process). This lock inheritance will cause a subsequent update transaction to be inserted into the query's AFTER set if it later issues a write operation that conflicts with a read operation by an update transaction already in the set. Rule 5, the rule added for strong consistency, may be enforced when a query requests a read-only lock on an object by automatically acquiring the lock for all older active queries as well.

5.3.3. Implementing the AFTER Sets

AFTER sets must be stored in a space-efficient manner, as in the worst case there may be an entry in a query's AFTER set for each update transaction that runs during its lifetime. In addition, the implementation of AFTER sets must support efficient access, as insertions and lookups occur quite frequently. For example, when an update transaction reads an object, as just described, the update transaction must check to see if the current version's creator is a member of the AFTER set of any currently executing query.

The scheme that we propose for representing each query's AFTER set is a bitmap indexed by the sequence numbers (transaction identifiers) that are assigned to update transactions when they enter the system. Operations on an AFTER set will then require only the testing or setting of bits in this bitmap. Since a query's AFTER set contains only update transactions which ran sometime during its lifetime, the first entry of a query's bitmap is assigned an index that is equal to the sequence number of the oldest update transaction running when the query entered the system. Furthermore, the bitmap may have a fixed size, as it is possible to assume that any update transaction whose sequence number falls beyond the end of the bitmap is automatically a member of the query's AFTER set. This assumption will not affect the correctness of the algorithm, merely its ability to exploit lesser forms of consistency. As an extreme, a bitmap of size zero, represented only by the sequence number of the last update transaction to commit prior to the start of the query, would cause the reduced consistency variations of MVQL to degenerate to strict consistency (i.e., to MV2PL).

5.3.4. Garbage Collection

We discuss two alternative approaches for removing unnecessary versions in MVQL (and MV2PL as well). A version is unnecessary if, for every active query, there is a more recent committed version of the object that was created by an update transaction that is not in the query's AFTER set. The first alternative is a *sequential* garbage collection scheme, as proposed in [Chan82], where prior versions are stored in a sequential log-like version pool; before an object is updated, it is appended to the version pool. In this approach, there are three pointers that mark regions in the version pool. *Last* marks the tail of the version pool (i.e., the most recent version), *update-first* marks the version that was appended least recently by an uncommitted update transaction, and *reader-first* marks the head of the version pool. When a query enters the system, it records the current value of *update-first*. When it exits the system, if it is the oldest query, it sets *reader-first* to the position it previously marked. The drawback of this approach is that is possible for a long-running query to hold up the garbage-collection of a potentially large number of prior versions, leading to a high storage overhead [Bobe92].

As an alternative to the sequential scheme, a *sifting* garbage collection scheme can be used in conjunction with a heap-based organization for storing prior versions. In this approach, when a update transaction completes, it assigns each prior version that it replaced to the youngest query that requires the version. When this query completes, it must sequence through its list of assigned versions and reassign them to the next-youngest query that requires the version.[6] If there is no such query, then the version may be garbage-collected. Compared to the query's overall path length, sequencing through a assigned list of versions should be relatively inexpensive.

5.3.5. Further MVQL Refinements

In this section we discuss several refinements to the basic MVQL algorithm. We present techniques for reducing storage cost and providing more control over consistency. In addition, we present a distributed version of MVQL that can be used in a shared-nothing parallel DBMS.

[6]Determining if a query requires a version may be done by simply checking its AFTER set; this adds only a small amount to the path length of update transactions since, as described above, AFTER set operations are inexpensive. Furthermore, maintaining a query's list of assigned versions is also inexpensive since the entries are small and the list may be spooled to secondary storage if necessary.

5.3.5.1. Early Garbage Collection

One way to reduce the storage cost of MVQL is to allow queries to specify that a particular object will not be accessed again; a new type of read-only lock mode may be introduced for this purpose, with read-only locks being downgraded to this new mode when appropriate. If the version of such an object seen by a query is a prior version, it could be garbage-collected either immediately or when other active queries no longer need the version. Likewise, if such a version is a current version, it will not have to be retained for the query when a newer version is created. In principle, a query optimizer could pass the information that would be needed to generate the lock modification calls to the access methods of a DBMS. Of course, this may not be feasible for queries written partly in a general-purpose programming language (e.g., a C program containing several SQL queries).

5.3.5.2. Consistency Groups

A single form of consistency may prove to be too loose for some queries and too strict for others. In order to provide different levels of consistency for different queries, it is possible to extend MVQL with the notion of consistency groups. A set of queries which belong to a *strong consistency group* serialize with both the update transactions and each other, but not with other queries outside the group. This is useful for situations where a set of queries must be run together as part of some sort of complex data analysis which is independent of other concurrent queries. A strong consistency group may be implemented by modifying Rule 5 in Section 5.3.2 to include only the queries that are part of the group. Similarly, a *strict consistency group* requires not only that the queries in the group serialize among themselves, but that the queries see a serial ordering of the update transactions that is consistent with their commit order. This may be implemented by having the queries in the group follow the strict consistency algorithm discussed in Section 5.3.2. Queries that do not belong to either a strong or a strict consistency group may decide independently to see either update or weak consistency by subjecting themselves to the appropriate set of rules (i.e., Rules 1 through 3 for update, or Rules 1 through 4 for weak).

5.3.5.3. Distributed MVQL

In recent years, shared-nothing parallel database systems have begun to replace centralized mainframe database systems [DeWi92]. In this section we discuss a distributed version of MVQL that is applicable to parallel DBMSs. We do not consider at this time more general distributed database systems containing replicated data.

In the distributed MVQL algorithm, we logically replicate a query's AFTER set at each processing node that is executing the query. Thus, we need an efficient way to update all of the copies of a query's AFTER set so that the query will see the same serialization order at each node. Clearly, a correct but inefficient solution is to somehow update all of the copies atomically (so that each copy is always seen to be identical). We choose a more efficient solution, however, which is to piggyback the AFTER set insertions of a given update transaction on the messages exchanged during the transaction's commit processing. We assume that the two-phase commit (2PC) protocol [Gray79] (or some other suitable commit protocol) is used to guarantee the atomic commitment of update transactions.[7] Specifically, the vote messages of 2PC are used to inform the coordinator of any local query AFTER set insertions involving the update transaction being committed, and the vote-reply messages are used to propagate these local insertions to each node participating in the transaction's execution. Upon receiving a vote-reply message, each node applies these AFTER set insertions before releasing the committing update transaction's locks.[8]

We argue that our piggybacked AFTER set update method is correct by showing that, where it differs from the atomic update method, the differences do not affect the correctness of the algorithm. Our update method differs from the atomic update method in the following three ways: (i) it propagates the insertions involving an update transaction only to nodes where the transaction executed, (ii) it delays the insertions until commit time, and (iii) it does not propagate insertions from a node if they occur there after the update transaction's vote

[7]The centralized 2PC protocol (which is one version of 2PC) works as follows: In the first phase, each node participating in a distributed transaction votes to either commit or abort the transaction by sending a message to the coordinator node. In the second phase, the coordinator responds to each participating node with either a commit or abort reply; the transaction is committed if all of the nodes voted to commit, otherwise it is aborted.

[8]In the ordinary 2PC protocol, the vote-reply message is usually not sent to sites where the transaction only read the database since there are no further commit or abort steps to be taken at these sites. In the distributed MVQL algorithm, we must send the vote-reply messages to these sites in order to inform them of any AFTER set insertions.

message has been sent. The first difference does not affect correctness because MVQL will never check query AFTER set membership for a given update transaction at a node where the update transaction did not execute. Rule 3 requires knowledge of the AFTER set membership of update transactions that have modified a record locally, and Rule 4 requires knowledge of the membership of those that have read a record locally; the remainder of the rules do not require any knowledge of AFTER set membership. The second difference does not affect correctness either, as even with delayed propagation a query will not be able to see the effects of an uncommitted update transaction; thus, the arrival at a node of a propagated insertion may occur at any point up until the update transaction releases its locks at the node. The third difference does affect the correctness of the algorithm; however, we can eliminate this difference by adopting a small change in the MVQL algorithm. The required change is that a query must now block behind an update transaction if it attempts to read an object that is write-locked by the transaction while the transaction is in the second phase of 2PC (i.e., after it has sent its vote message). This change is made by modifying Rule 2 in Section 5.3.2 to read as follows:

(2') U is placed into the AFTER set of a query Q if Q attempts to read an object that is currently write locked by U *and* U has not yet sent its vote to the two-phase commit coordinator. Otherwise, if the vote has already been sent, Q must block until U releases its lock on the object.

5.4. The Simulation Model

In this section, we describe the model that we used to compare the performance of MV2PL and MVQL. The model captures the details of page-level versioning implementations of each of these algorithms. Since we are no longer focusing on the storage organization details of individual pages, we chose to model page-level versioning in this chapter to reduce the complexity of the simulation program, and to enable longer-running simulations. As in the models used in Chapters 3 and 4, the model used in this chapter has two major components, the application model and the system model. Each of these has several subcomponents that will be described in this section. The model was implemented in the DeNet simulation language [Livn89].

5.4.1. The Application Model

The first component of the application model is the database, which is modeled as a collection of *files*. Each file, in turn, is modeled as a collection of records. One clustered and one unclustered index exist on each file. We assume that each index has *IndexFanout* keys per index page and (for simplicity) that there is a one-to-one relationship between key values and records. Each file has *FileSize* records, and each record occupies *RecSize* bytes. The overall database is physically organized as a series of <file, clustered index, unclustered index> triples that are laid out on the disk in cylinder order. Prior versions of pages are stored in a version pool following all of the primary data. We discuss the version pool organization in more detail in the next subsection. The parameters for this portion of the overall model are summarized in Table 5.2.

The second component of the application model, the source module, is responsible for modeling the external workload of the DBMS. Table 5.3 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 read-only *query* transaction class. *Query* transactions execute relational select (range-query) 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. The parameters include an access method and a mean selectivity for each file ($AccessMeth_{class,file}$ and

Parameter	Meaning
NumFiles	Number of files in database
$NumKeys_{file}$	Number of keys per index page for file
$FileSize_{file}$	Number of records in file
$RecSize_{file}$	Size of records in file

Table 5.2: The Application Model Parameters

Parameter	Meaning
MPL_{class}	Number of terminals (*class* is *query* or *update*)
$AccessMeth_{class,file}$	Access method used by class for file
$Selectivity_{class,file}$	Mean selectivity for class for file
$SelectivityDistr_{class}$	Distribution of actual selectivities
$UpdateFrac_{file}$	Fraction of selected tuples to update

Table 5.3: The Workload Model Parameters

$\mathit{Selectivity}_{\mathit{class,file}}$, respectively). The actual selectivity for a given run is chosen from a distribution, $\mathit{SelectivityDistr}_{\mathit{class}}$. For update transactions, $\mathit{UpdateFrac}_{\mathit{file}}$ specifies the fraction of selected records to actually update. It is assumed that indexed attributes are not updated by the *update* transactions. This assumption was made so that we could use single-version indexes in this study, leaving further exploration of indexing for future work.

We chose this workload model in order to capture situations where there are a relatively large number of updates per query and where queries do a significant amount of work. This workload model, despite its simplicity, places sufficient demands on the version management system to highlight the important performance issues and tradeoffs.

5.4.2. 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 operator manager module, the concurrency control module, the buffer manager module, the CPU module, and the disk manager module. Table 5.4 summarizes the key parameters of the system model.

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 record from a disk page and *CompareCPU* instructions to compare two index keys. In addition, *StartupCPU* and *TerminateCPU* instructions are charged to start and terminate an operator, respectively.

The concurrency control manager encapsulates the operations of the MVQL and MV2PL algorithms. It consists of two subcomponents: the lock manager and the version manager. The CPU costs of concurrency control are modeled by charging *LockCPU* instructions for each lock request. This includes both the traditional 2PL read

Parameter	Meaning
NumBuffers	Number of page frames in the buffer pool
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
PrefetchNum	Number of pages to prefetch
CacheSize	Size of disk prefetch cache
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
ccCPU	Cost for a lock manager request
BufCPU	Cost for a buffer pool hash table lookup
IO_CPU	Cost to initiate an I/O operation

Table 5.4: The System Model Parameters

and write locks as well as the read-only locks introduced by MVQL. Both locking and versioning are both supported at the page level. We chose page-level versioning to simplify the implementation of the simulator and to reduce the length of simulation runs. The basic MV2PL and MVQL algorithms are compatible with record-level versioning; schemes for record-level versioning are discussed in [Bobe92, Moha92].

The version manager divides the database into two segments: the main segment, containing the current versions of pages, and the version pool, containing prior page versions. This organization is similar to the one described in [Chan82], except that we arrange the version pool as a heap of disk tracks rather than as a circular (log-like) buffer. This change alleviates the problems of sequential garbage collection discussed in Section 5.3.4 while still providing good write performance (as write operations to the version pool are done a track at a time).[9] Access to prior versions is provided through a memory-resident index that maps a current page number and the AFTER set of a query to the location of the appropriate prior version of a page. The memory-resident index is

[9]It should be noted that the decision to use a heap-based version pool rather than a circular buffer is orthogonal to the basic MVQL and MV2PL algorithms; we could have chosen to use the circular buffer organization instead.

another departure from the scheme in [Chan82], where prior versions of a page were located by chaining back from the current version. We chose the directory approach in order to present the performance differences of the algorithms relatively conservatively. With reverse chaining, the MVQL algorithm would appear even more attractive than MV2PL, as queries tend to access younger versions under MVQL than MV2PL.

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 the frames are shared among the main segment, version pool, and index pages. Version pool pages that are inserted into the buffer manager by the version manager are not assigned physical disk addresses until they are written out to disk; this is done to eliminate fragmentation problems on tracks due to versions that can be garbage-collected while still in the buffer pool. When a dirty version pool page reaches the end of the LRU chain, a track's worth of dirty version pool pages are written to a free track on disk. The CPU cost of searching for a requested page in the buffer pool hash table is modeled by charging *BufCPU* instructions. If the page is not resident, an additional *BufCPU* instructions is charged to insert the page in the table; *IO_CPU* instructions are then charged to initiate an I/O operation.

The CPU module encapsulates the behavior of an FCFS CPU scheduler, granting transactions the use of the CPU until they request a new page from the buffer manager. The disk manager module is designed to model the behavior of a disk controller and driver. The controller schedules 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*. We assume that the disk controller has a prefetch option that may be selected on a per-request basis to optimize sequential access performance. In addition to reading the requested page, the prefetch mechanism will load the next *PrefetchNum* pages into a FIFO cache contained within the disk controller; subsequent requests for these pages will then not require physical I/O operations. The controller contains room for a total of *CacheSize* prefetched pages.

5.5. Experiments and Results

In this section, we present the results of three experiments that compare the performance of MV2PL and MVQL. As a yardstick for comparison, we also include the results of GO processing; recall that GO processing allows queries to run without setting locks at all. The primary performance metrics employed in this study are query throughput, update transaction throughput, and storage cost for maintaining older versions of pages. Additional metrics are used in the analysis of the experimental results. 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 in almost all cases. Throughout the chapter we discuss only performance differences that were found to be statistically significant.

Table 5.5 lists the settings for the system model and the application model parameters. The system has a CPU that executes 12 million instructions per second and a single disk with a page size of 8K bytes and a track size of 5 pages. The disk controller can prefetch up to 4 pages following a requested page;[10] the controller contains a 256K byte cache for storing prefetched pages. This model was patterned after the Fujitsu M2266 disk drive [Fuji90], which is as an example of a current generation disk drive. With this configuration, typical disk access

Parameter	Setting	Parameter	Setting
CPURate	12 MIPS	*NumFiles*	4
NumDisks	1	*IndexFanout*$_{file}$	450
DiskSeekFactor	0.617 msec	*FileSize*$_{file}$	25000 records
DiskLatency	0-16.67 msec (uniform)	*RecSize*$_{file}$	227 bytes, including overhead
DiskSettle	2.0 msec	*SelectCPU*	400 instructions
DiskTransfer	3.07 MBytes/sec	*CompareCPU*	50 instructions
DiskPageSize	8K	*StartupCPU*	10000 instructions
DiskTrackSize	5 pages	*TerminateCPU*	2000 instructions
CacheSize	32 pages	*LockCPU*	150 instructions
PrefetchNum	5	*BufCPU*	150 instructions
NumBuffers	600 pages	*IO_CPU*	1000 instructions

Table 5.5: System and Application Model Parameter Settings

[10]The prefetch option is used for main segment read requests by the sequential and clustered index scan access methods. To prevent disk bandwidth from being wasted as a query shifts from a sequential access pattern in the main segment to a random pattern in the version pool, a query stops requesting the prefetch option once it observes a disk cache hit ratio of less than 60% from its prefetch requests.

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 25,000 Wisconsin benchmark-sized records. Each record contains 208 bytes of data and 19 bytes of overhead, for a total of 227 bytes (as is the case in the Gamma system [DeWi90]). With this record size, 36 records fit on a page. Each file contains both a clustered and an unclustered B+ tree index, each with a node fanout of 450. The CPU costs of executing transactions in the workload include various instruction charges that are detailed in Table 5.5

The parameter settings for the workload model were varied from experiment to experiment. These settings are listed in Table 5.6, and are described with each experiment.

5.5.1. Experiment 1: Effect of Query Selectivity

In this experiment, we study the effect of query selectivity on each of the alternative concurrency control algorithms. A transaction workload is initiated from a set of 12 update transaction terminals and 1 query terminal; the terminals do not involve an external think time delay.[11] Update transactions use the non-clustered indexes to select and then update 2 randomly selected records in the database, while queries use clustered indexes to scan a randomly selected region of each of the 4 files in the database. The query selectivity is kept constant within the same simulation run (across both files and queries), but is varied from 10% to 90% between runs. We vary the

Parameter	Experiment 1	Experiment 2	Experiment 3
MPL_{Query}	1 query	1 query	1 to 5 queries
MPL_{Update}	12 updaters	12 updaters	12 updaters
$Selectivity_{updater,file}$	2 records	1 to 32 records	2 records
$SelectDistr_{query}$	constant	constant	uniform over 2/3 to 4/3 of mean
$SelectDistr_{update}$	constant	constant	constant
$UpdateFrac$	100%	25%	100%
$Selectivity_{query,file}$	10% to 90%	25%	40%
$AccessMethod_{Query}$	clustered index	clustered index	clustered index
$AccessMethod_{Updater}$	unclustered index	unclustered index	unclustered index

Table 5.6: Workload Model Parameter Settings

[11]This captures the average behavior of a system with a larger number of terminals that do involve an external think time delay. By abstracting the model in this way, we were able to reduce the variance in the statistics and obtain tight confidence intervals without excessive simulation lengths.

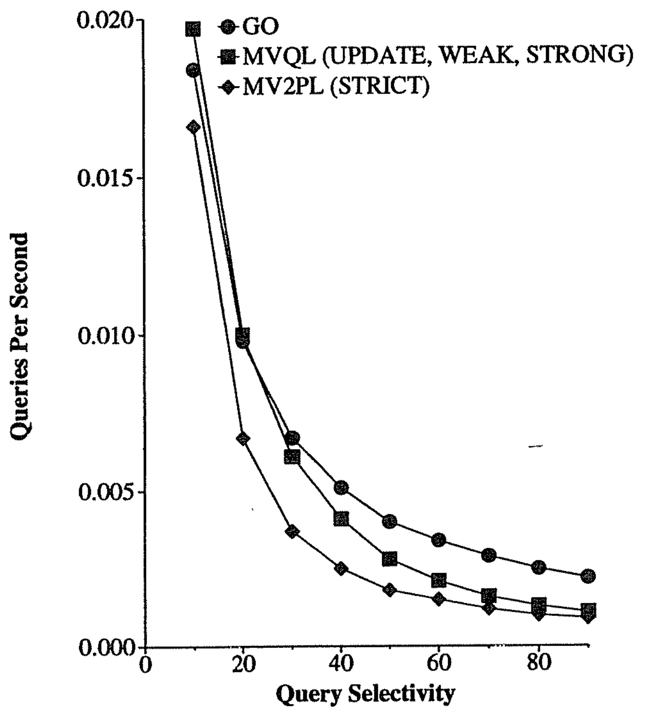
query selectivity over a wide range to show how versioning influences system performance as queries increase in size; size is a key factor here because as the queries become larger, the version management system must maintain transaction-consistent states of the database that are increasingly different than the current state. 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).

Figures 5.5 through 5.8 show query throughput, fraction of query accesses to current versions, average storage cost, and update transaction throughput for each of the algorithms over a range of query selectivites. Note that for a single-query workload, weak and strong consistency are identical. Also, since update transactions modify each record that they read, weak consistency and update consistency are the same here as well. This explains why there is only a single curve in the graphs for MVQL in this experiment. We start by considering query throughput. In Figure 5.5, we see that the highest query throughput is observed with GO processing, while the lowest is observed with MV2PL. An exception to this occurs below approximately 20% query selectivity, where a slightly higher query throughput is achieved with MVQL than with GO processing.[12] At lower query selectivities, MVQL's query throughput is close to that of GO processing, and at higher selectivities it approaches that of MV2PL. The reasons for the differences in query throughput between the algorithms are illustrated by the graph in Figure 5.6, which shows the fraction of query accesses that go to the current version of a page. Recall that when queries access the current versions of pages, sequential access is preserved; thus the prefetch option may be used to read five pages from the disk with a single arm movement and rotational delay. GO processing achieves the highest query throughput since only current versions of pages are accessed. At the opposite side of the spectrum, MV2PL achieves the lowest query throughput since, to maintain strict consistency, queries access the fewest current page versions. As queries become larger with both MVQL and MV2PL, the fraction of accesses to current page versions drops since queries see a state of the database that becomes increasingly older

[12]This exception is caused by the garbage-collection of page versions in the buffer pool. With versioning, an update transaction must copy a page in the buffer pool before updating it. This may require cleaning the buffer frame at the end of the LRU chain. If the prior version is garbage-collected when the update transaction commits, the update transaction will contribute this clean buffer to the next transaction that requests one. Some fraction of the time, a query will be the recipient of a page cleaned by an update transaction in this manner. Thus, when garbage collection in the buffer pool is frequent, a small amount of work will be shifted from the queries to the update transactions. As we will see shortly, garbage collection in the buffer pool is common with MVQL at low query selectivities.

than the current state. This happens more quickly with MV2PL than with MVQL since MVQL does not place all update transactions that arrive during a query's execution into its AFTER set (i.e., some of these transactions serialize before the query). The rate at which concurrent update transactions are placed into a query's AFTER set starts out low and then increases steadily as the query ages and acquires more read-only locks; the reason for this can be seen by reviewing Rules 1 through 3 in Section 5.3.1. Specifically, in this experiment (but not shown in the graphs displayed), an average of about 10% of all update transactions that ran during the lifetime of a 10% select query were placed into its AFTER set in MVQL. This percentage increased to just over 40% at a query selectivity of 30%, to about 80% at a selectivity of 60%, and to nearly 90% at a selectivity of 90%. This explains why, as the query selectivity is increased in Figure 5.5, the query throughput of MVQL approaches that of MV2PL.

We now consider storage cost. Storage cost is also dependent on query selectivity, as the multiversion algorithms must keep transaction-consistent states of the database that with time become increasingly different than the current state. This can be seen by the graph in Figure 5.7, which shows the average number of prior page versions observed for MV2PL and MVQL during each simulation run; note that the curve for GO processing is constant at 0 since GO processing does not retain prior page versions. Recall that a query always accesses the most recent version of a page that was written by a transaction not belonging to its AFTER set. Thus, with a query MPL of 1, the version pool must contain the prior version of the first update to each page by a transaction in the currently executing query's AFTER set, and multiple updates to the same page during the lifetime of a query do not increase the storage cost. This explains why the slopes of both the MV2PL and MVQL curves decrease as query selectivity is increased. If the query runs long enough, each page in the database will have been updated by some transaction in its AFTER set; when this occurs, the version pool will contain an entire copy of the database. Notice that MVQL has a considerably lower storage cost than MV2PL. This is because the version pool grows at a slower rate with MVQL, and, as was discussed previously, queries complete faster under MVQL than MV2PL. In order to see why the version pool grows at a slower rate with MVQL than MV2PL, recall that to maintain strict consistency, MV2PL places all update transactions that run during a query's lifetime into its AFTER set, while



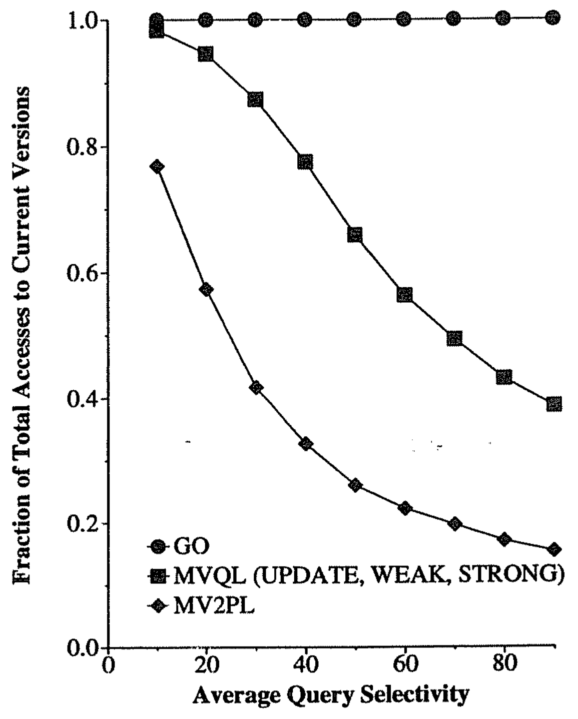
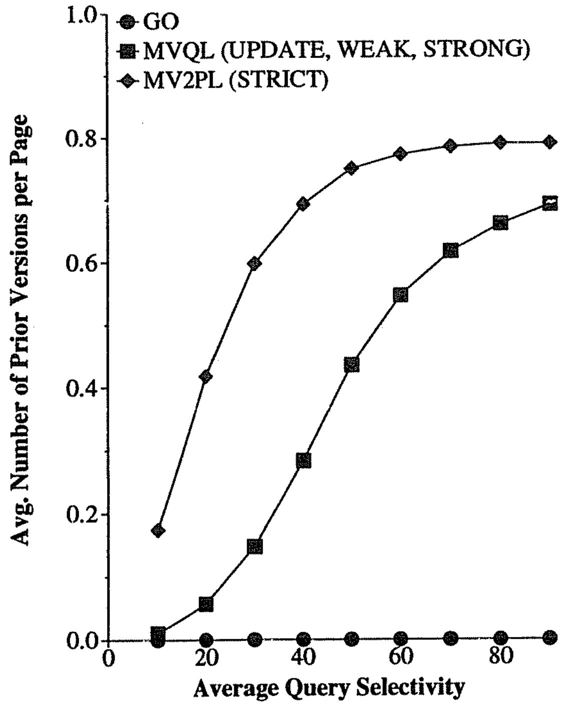


Figure 5.5: Query Throughput **Figure 5.6: Current Version Access**
($MPL_{query} = 1$, $MPL_{update} = 12$, update transaction size = 2)



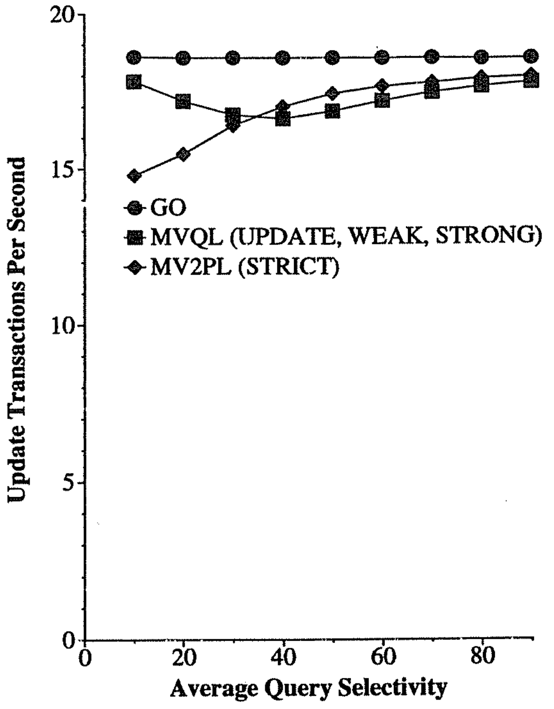


Figure 5.7: Storage Cost **Figure 5.8: Update Transaction Throughput**
($MPL_{query} = 1$, $MPL_{update} = 12$, update transaction size = 2)

MVQL does not.

Finally, we turn our attention to the update transaction throughput, shown in Figure 5.8. The differences between the algorithms here were caused by the number of updates to the version pool. GO processing does not maintain a version pool, so it provides the highest update transaction throughput. Both of the remaining curves vary along with query selectivity. The drop in MVQL's update transaction throughput between 10% and 40% selectivity can be explained by the increase in the rate at which concurrent update transactions are placed in the AFTER set of the active query as it ages. Initially, when the query is young, MVQL places the majority of update transactions that arrive in the system *before* the query in the serial order. As noted previously, an average of only about 10% of update transactions that ran during a 10% selection query's lifetime were serialized after it. When an update transaction that serializes before all active queries commits, the prior versions of its updates can be garbage-collected. Since update transactions were short in this experiment, such prior versions were almost always garbage-collected while still in the buffer pool (and thus were never written to the version pool on disk). Garbage collection of versions in the buffer pool increases the availability of clean buffers, thus helping to increase update transaction throughput relative to MV2PL at low query selectivities.

When a query becomes older, the rate at which concurrent update transactions can be serialized before the query drops. Again, as we noted previously, slightly over 40% of the update transactions that ran during the lifetime of a 25% select query were serialized after the query. This resulted in an increased rate of updates to the version pool, and explains MVQL's drop in update transaction throughput. MV2PL had no such drop in its update transaction throughput since all update transactions that run during the lifetime of a query serialize after it. In fact, update transaction throughput rose as selectivity was increased. This rise, and the rise in the MVQL throughput after 40% selectivity, results from pages being updated multiple times during the lifetime of the currently active query; at most one version of each page must be written to the version pool for this query. As the query selectivity is increased, the MV2PL and MVQL update transaction throughputs both approach that of GO processing. The reason is that the cost of incrementally writing a copy of the database to the version pool for each query is amortized over increasingly longer query executions.

This experiment has shown the clear advantages of the MVQL algorithm over MV2PL in terms of query throughput, storage cost, and to a lesser degree, update transaction throughput. The performance benefits are largest for smaller sized queries, and they decrease as the query size is increased.

5.5.2. Experiment 2: Effect of Update Transaction Size

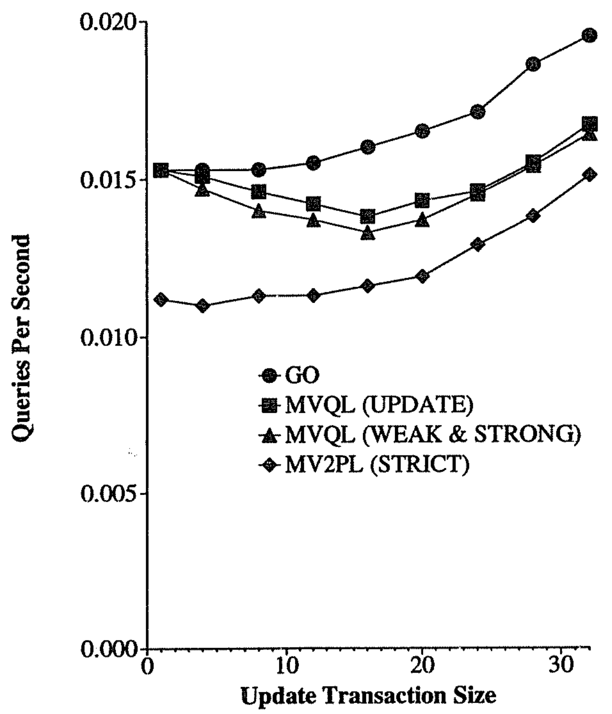
In this experiment, we look at the effect of update transaction size on the algorithms. Update transaction size affects the MVQL algorithm the most, as each additional lock request by an update transaction increases the chance that it will be placed in the AFTER set of an active query. This may be seen by reviewing the rules in Section 5.3.1. To study the effect of update transaction size, we vary the number of record select operations by each update transaction from 1 to 32. Update transactions use non-clustered indexes to select 1 to 32 records (varied across simulation runs), updating an average of 25% of the records selected. Queries, on the other hand, use clustered indexes to scan 25% of each of the four files. Since we again consider a single query workload, weak consistency is identical to strong consistency; weak consistency and update consistency are not identical in this experiment, though, since update transactions modify only a fraction of the records that they read.

We begin by considering query throughput, shown in Figure 5.9. The first thing that we wish to point out is that there is only a small difference between the two MVQL curves (update consistency vs. weak and strong consistency); this was found to be true across the entire range of possible *UpdateFrac* values. This indicates that lock inheritance, introduced due to Rule 4 in Section 5.3.1, has little impact on the size of query AFTER sets. In other words, the rate at which a query receives new read-only locks through inheritance is much lower than the rate at which it requests them explicitly. For simplicity of explanation, we will not distinguish between the MVQL curves for the remainder of this experiment.

Moving to a comparison of MVQL with MV2PL and GO processing, we notice that the query throughput for both GO processing and MV2PL rises as the update transaction size is increased. This rise is caused by a decrease in the system resource demands by update transactions due to increased lock waiting; recall that the probability of lock conflict is proportional to the square of the transaction size [Gray81, Tay85]. So that we may concentrate on the most important results, we do not show the update transaction throughput here. On the other hand, MVQL query throughput drops initially, and then rises. The rise is also caused by reduced resource competition from the update transactions. The initial drop (as the update transaction read size is varied from 1 to 16) is due to an increase in query AFTER set sizes. The AFTER set size increases because each additional lock request by the update transactions increases the chance that the transaction will be placed in the AFTER set of the query. As we explained in the discussion of the previous experiment, increasing the AFTER set size reduces the number of current version accesses. Specifically, for an update transaction size of 1, an average of only 4% of the update transactions that ran during the lifetime of a query were placed into its AFTER set (not shown in the graphs displayed). With an update transaction size of 8, however, this percentage rose to 40%, and at a size of 32, it rose to over 80%. This caused the percentage of accesses to current versions to drop from nearly 100% to 90% for MVQL. Note that for MV2PL, this percentage stayed relatively constant at around 85%. As for query size in the previous experiment, increasing the update transaction size causes the query throughput of MVQL to become closer to that of MV2PL.

We now turn our attention to storage cost. Figure 5.10 shows the average storage cost observed for both MV2PL and MVQL during each simulation run. The difference between update consistency and weak (or strong) consistency for MVQL is again rather small so we do not distinguish between them. In the graph, we see that the storage cost of MV2PL drops from about 30% to 17% of the database size. This corresponds to the drop in update transaction throughput that is caused by increased lock contention as the update transaction size is increased. In contrast, we see in Figure 5.10 that MVQL's storage cost starts out extremely low, rising until an average update transaction size of approximately 20 is reached, and then it decreases again. The initial rise is caused by the increase in the average query AFTER set size; the storage cost starts out low because of the small query AFTER set size with small update transactions. Recall that the connection between the AFTER set size and storage cost is that prior page versions need to be retained only for updates made by transactions in an active query's AFTER set. The AFTER set size also influences storage cost indirectly by influencing the query response time; as discussed in the previous experiment, increasing the AFTER set size degrades the sequentiality of query access, and thus increases query response time (and consequently storage cost). The drop in MVQL storage cost as the average update transaction size increases past 20 is due to the lock contention discussed already.

For the sake of presentation, we do not show the update transaction throughput results for this experiment, but we summarize the results here. MVQL achieved an update transaction throughput that ranged between 98%



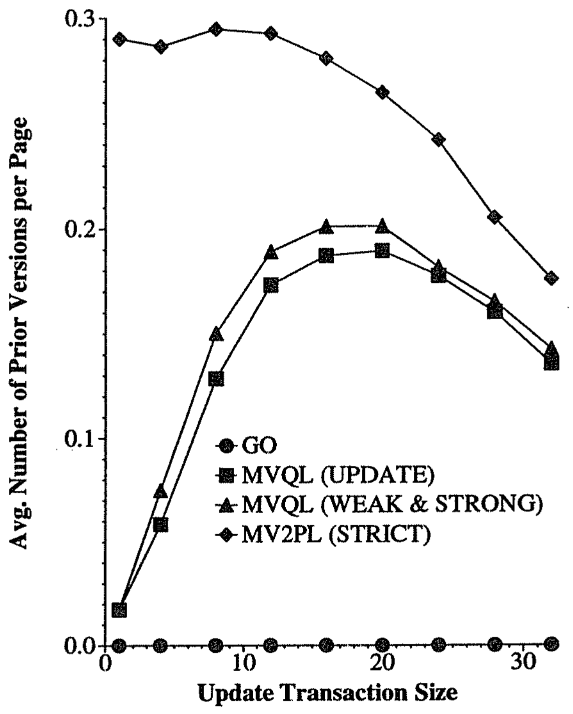


Figure 5.9: Query Throughput **Figure 5.10: Storage Cost**
($MPL_{query} = 1$, $MPL_{update} = 12$, $Selectivity_{Query} = 25\%$)

to 32). In the range of update transaction sizes from 1 to 8, MV2PL had a slightly lower update transaction throughput than MVQL; the largest difference amounted to approximately 8% of MVQL's update transaction throughput at a size of 1.

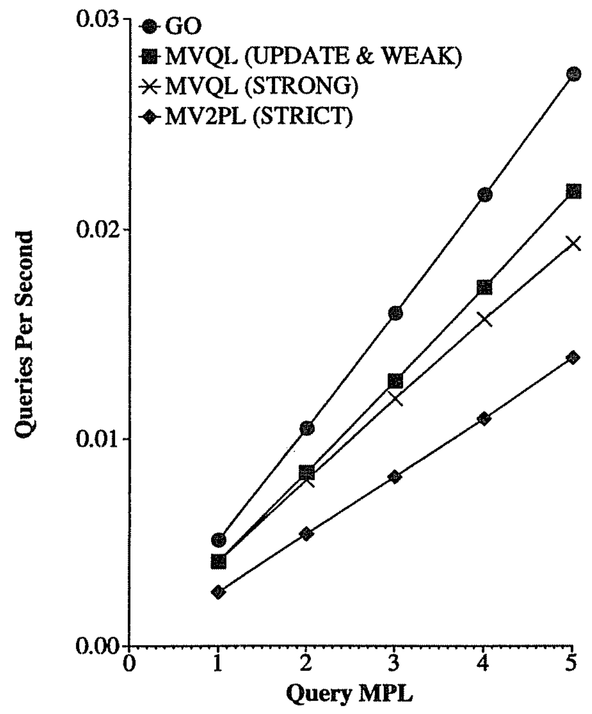
In the first experiment we saw that the performance of MVQL in terms of query throughput, storage cost, and update transaction throughput is close to that of GO processing when queries are small, and it approaches MV2PL as queries become larger. In this experiment we have seen a similar result occur when update transaction size is increased instead. The connection between these results lies in the AFTER set sizes of queries. Increasing either the query size or the update transaction size decreases the opportunities for serializing update transactions before concurrently executing queries. In addition, we have seen that the additional I/O and storage costs for providing weak consistency over update consistency are quite small.

5.5.3. Experiment 3: Effect of Query Multiprogramming Level

In this experiment, we vary the query multiprogramming level from 1 to 5 queries in order to study its impact on the relative performance of the weak and strong consistency variations of MVQL. Update transactions use non-clustered indexes to select and then update 2 records, while queries use the clustered indexes to scan an average of 40% of each of the four files. Since we again consider a workload where update transactions write each record that they read, update consistency is identical to weak consistency here. In order to stagger the start and commit times of queries from different terminals, we vary the actual selectivity across queries uniformly between 2/3 and 4/3 of the average selectivity.

In Figure 5.11, we see that the query throughput for all algorithms rises as the number of query terminals is increased, while in Figure 5.12, there is a corresponding decline in update transaction throughput. The rise in query throughput is due to the system shifting its effort from update transaction processing to query processing; the system is devoting 1 out of 13 terminals to query processing at the left-hand side of the graph, while it is devoting 5 out of 17 terminals at the right-hand side. The bulk of the increase in query throughput is a direct result of the increase in the overall fraction of terminals devoted to query processing. The remaining increase in query throughput is due to a reduction in the number of dirty pages in the buffer pool, which leads to a reduction in the amount of buffer cleaning work that must be done by queries.

In Figure 5.11, we also see that the query throughput of strong MVQL diverges from that of weak (and update) MVQL as the number of query terminals is increased. The separation is caused by the additional rule for strong consistency (Rule 5 of Section 5.3.1) that causes a query's AFTER set to subsume the AFTER sets of all younger queries. The separation is not as large under this workload as one might expect, however. The reason is that the AFTER sets of older queries are already likely to subsume those of younger queries due to the enforcement of Rules 1 through 4; relatively few AFTER set insertions will occur as a result of Rule 5 alone, especially at low MPLs. This reasoning also explains why the storage cost of strong MVQL, shown in Figure 5.13, diverges only very slightly from that of weak (and update) MVQL as the number of query terminals is increased.



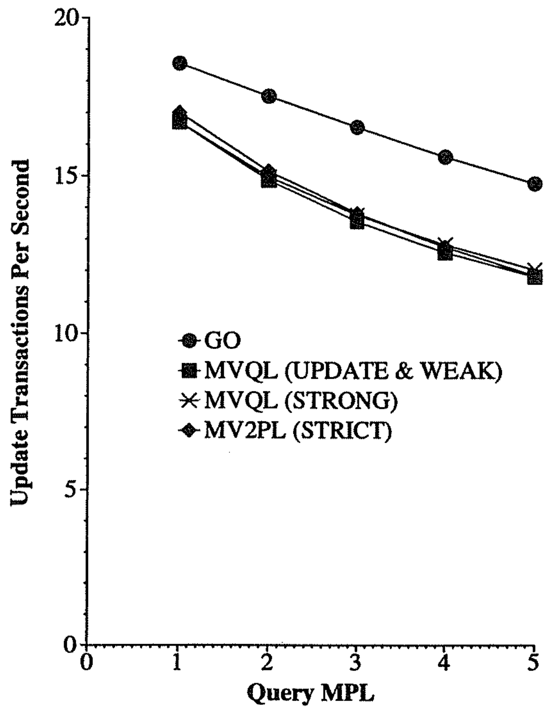


Figure 5.11: Query Throughput **Figure 5.12: Update Transaction Throughput**
(MPL_{update} = 12, $Selectivity_{Query}$ = 40%, update transaction size = 2)

In this experiment, we have seen that the cost of providing strong consistency is not considerably higher than that of providing weak consistency. As we discovered, enforcing Rule 5 is not as detrimental to performance as one might initially expect.

5.5.4. Discussion

In this section, we have presented the results of a performance analysis of the MVQL algorithm. We investigated a workload combining small transactions, each performing record-select/update operations, with large queries executing clustered index scans. We did not consider queries with random file accesses (i.e., through an unclustered index) since we were interested in higher selectivity scans. To avoid re-reading pages, medium and large selectivity scans on an unclustered index attribute can be executed by first obtaining a list of the IDs of matching records from the index, sorting the list according to disk address, and then sequentially scanning the data using the record-ID list [Moha90]. Our results indicate that MVQL should also provide a lower cost alterna-

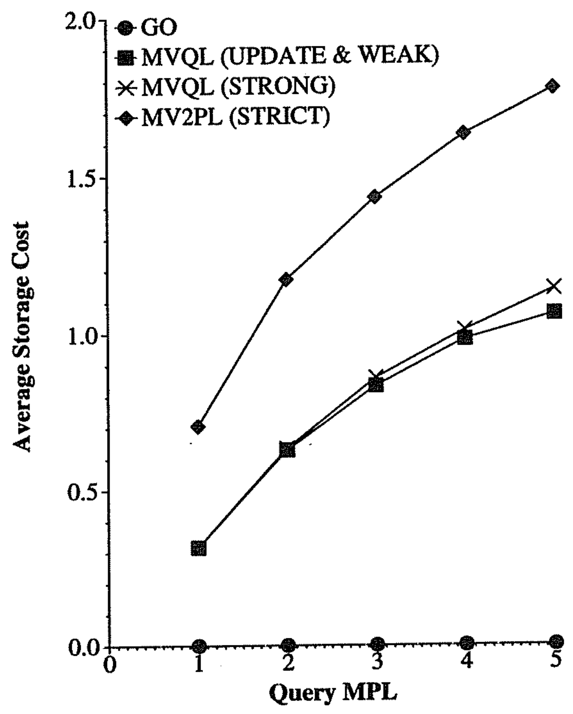


Figure 5.13: Storage Cost
(MPL_{update} = 12, $Selectivity_{Query}$ = 40%, update transaction size = 2)

tive to MV2PL for these sorts of workloads. Finally, the benefits of MVQL should be even more significant when versions are located by reverse chaining rather than through a memory-resident directory (as we assumed in this study).

In order to make simulations with large queries feasible, we used a relatively small database in our experiments; however the update intensity to individual pages was quite high. We feel that the results should scale to a larger database with a proportionally lower update intensity, as the number of updates that fall in the path of a query will remain the same.

5.6. Conclusions

In this chapter, we have presented a new multiversion locking algorithm that has a lower versioning cost than the MV2PL algorithm that several commercial systems use. Our new algorithm, MVQL, reduces the cost of versioning by providing weaker forms of consistency for queries than that provided by MV2PL. To introduce the new algorithm, we reviewed four forms of consistency which all guarantee that queries see transaction-consistent data: update consistency (the least restrictive form), weak consistency, strong consistency, and strict consistency (the most restrictive form). We showed that the increasingly restrictive consistency forms require that queries read older versions of data, and we argued that this will increase the cost of executing queries. We then we presented the MVQL algorithm as a generalization of MV2PL. MV2PL provides only strict consistency, while MVQL can provide either update, weak, strong, or strict consistency; in the case of the latter, it is equivalent to MV2PL.

We also conducted a detailed simulation study of the algorithms, and we analyzed the results of this study. The results show that MVQL can provide performance that is close to that of GO processing at small to medium query selectivities or update transaction sizes; it provides performance closer to that of MV2PL as the query selectivity and update transaction size are increased.

CHAPTER 6

THESIS CONCLUSIONS

This thesis has explored the use of multiversion locking for on-line query processing. Multiversion locking allows queries to be run on-line, against transaction-consistent data, without contributing to data contention. A drawback of multiversion locking is the overhead that it imposes to store, maintain, and access prior versions of data. This thesis has proposed and evaluated techniques that attack each of these sources of overhead.

In Chapter 3, we presented a new record-level storage management design for MV2PL. The design utilizes on-page caching and garbage collection of prior versions in order to minimize query execution cost, reduce the storage overhead for versions, and reduce the costs incurred by update transactions. The results of a simulation study, described in the chapter, indicate that on-page caching indeed provides significant performance gains. Finally, we have also introduced the concept of view sharing as a way to further reduce storage overhead for versioning.

While on-page caching is an efficient scheme for organizing versions on secondary storage, it does not address the question of how to index the versions for efficient associative access. In Chapter 4, we described several options for extending existing single-version indexing structures to handle multiversion data, and we integrated each of the approaches with on-page caching to provide a complete version placement and indexing solution for MV2PL. The alternative indexing approaches differ in where they store version selection information (i.e., the references to individual versions). The results of a simulation study presented in the chapter indicate that placing the version selection information with the data, rather than in the indices, is generally the best approach.

Finally, in Chapter 5, we argued that a query's execution cost will be reduced if it accesses fewer prior versions. This led to the development of a new multiversion locking algorithm, *multiversion query locking* (MVQL), that allows queries to access younger versions of data. In exchange, the queries are subjected to seeing weaker forms of consistency than that provided by MV2PL; however, even the weakest form of consistency ensures that queries will see only transaction-consistent data. The chapter reviewed several interesting forms of consistency and then showed how MVQL can be used to achieve each of those forms. The results of a simulation study presented in this chapter show that both query execution cost and version storage cost can be significantly lower with the weaker consistency forms.

6.1. Directions for Future Work

This thesis has opened up a number of interesting avenues for future work. To begin with, several possibilities remain in the area of storage management for versions. One possibility is to consider *differential versioning* [Moha92], where prior tuple versions are represented by storing only that portion of each version that differs from the next most recent version. When combined with on-page caching, differential versioning would clearly reduce the rate of overflow to the version pool, and could thus further improve query performance. Another possibility in the area of storage management is to examine alternative version pool representations. While the CCA circular buffer pool organization allows versions to be created inexpensively, the clustering of versions in the version pool leads to thrashing for long-running clustered index scan queries. This is because the versions are clustered according to the time they were appended to the version pool, and not according to the clustering of the main segment. Thus, an interesting question is how to organize the version pool to balance the version creation cost against the clustered index scan cost.

Another avenue for future work is resource allocation and load control for queries. As we observed in this thesis, the cost of query execution rises with query age. Thus, from a throughput perspective (and possibly a response time perspective as well), it might be better to restrict the query multiprogramming level. This would allow fewer queries to run simultaneously, but each query would proceed at a faster rate.

A comparison between compensation-based on-line query processing [Srin92] and multiversion locking is yet another avenue for future work. In addition, it would be interesting to see how some of the ideas of compensation-based on-line query processing could be applied to multiversion locking. For example, compensation-based on-line query processing examines the selection predicate on each base relation of a query to determine if a given update is relevant to the query. Combining this idea with multiversion locking might enable a more generally applicable strategy to be developed, one where only the relevant portions of the database would be versioned for use by queries.

Finally, in order to validate the techniques presented in this thesis, an implementation in the context of an actual DBMS (or storage manager) would clearly be useful.

Bibliography

[Agra87] 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.

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

[Baye77] Bayer, R., and Schkolnick, M., "Concurrency of Operations on B-trees," *Acta Informatica*, September 1977.

[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 Transactions on Database Systems," 8(4), December 1983.*

[Bern87] Bernstein, P., V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," *Addison-Wesley Publishing Company*, 1987.

[Care82] Carey, M. J., *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. Thesis, Comp. Sci. Dept., Univ. of California, Berkeley, 1983.

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

[Bobe92a] Bober, P. and M. Carey, "On Mixing Queries and Transactions via Multiversion Locking," *Proc. of the Eighth IEEE Data Engineering Conf.*, 1992.

[Bobe92b] Bober, P. and M. Carey, "Multiversion Query Locking," *Proc. of the Eighteenth International Conference on Very Large Databases*, 1992.

[Bobe92c] Bober. P. and D. Dias. *Storage Cost Tradeoffs for Multiversion Concurrency Control.* Research Report RC 18367, IBM T.J. Watson Research Center, Yorktown Heights, NY, July 1992.

[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.

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

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

[DeWi92] DeWitt, D., and J. Gray, "Parallel Database Systems: The Future of High Performance Database Processing," *Communications of the ACM*, 35(6), June 1992.

[DuBo82] DuBourdieu, D., "Implementation of Distributed Transactions," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1982.

[East86] Easton, M., "Key-Sequence Data Sets on Indelible Storage," *IBM Journal of Research and Development*, May 1986.

[Eswa76] Eswaran, K., J. Gray, R. Lorie, I. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *CACM* 19(11), 1976.

[Fuji90] *M2266S/H Intelligent Disk Drive Technical Handbook*, Publication FS810125-01 rev. B, Fujitsu America, August 1990.

[Garc82] Garcia-Molina, H., and G. Wiederhold, "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems*, 7(2), June 1982.

[Gray76] Gray, J., et al., "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 Computing Surveys*, 13(2), June 1981.

[Haas90] Haas, L., et al, "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[Hadz85] Hadzilacos, T., and C. Papadimitriou, "Algorithmic Aspects of Multiversion Concurrency Control," *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1985.

[Hsia90] Hsiao, H., *Performance and Availability in Database Machines with Replicated Data*, Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, Sept. 1990.

[Josh93] Joshi, Ashok, *Personal Communication.*

[Kolo89] Kolovson, C., and M. Stonebraker, "Indexing Techniques for Multiversion Database," *Proc. of the Fifth IEEE International Conference on Data Engineering*, 1989.

[Lai84] Lai, M., K. Wilkinson, "Distributed Transaction Management in Jasmin," *Proc. of 10th International Conference on Very Large Database Systems,* 1984.

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

[Lome89] Lomet, D., and B. Salzberg, "Access Methods for Multiversion Data," *Proc. 1989 ACM SIGMOD Conf.*, 1989.

[Lome90] Lomet, D., and B. Salzberg, "The Performance of a Multiversion Access Method," *Proceedings ACM SIGMOD Conference*, 1990.

[Lehm81] Lehman, P., and Yao, S., "Efficient Locking for Concurrent Operations on B-trees," *ACM Transactions on Database Systems*, 6(4), December 1981.

[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.

[Moha90a] Mohan, C., et al., "Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques," *Proceedings International Conference on Extending Database Technology*, 1990.

[Moha90b] Mohan, C., "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes," *Proceedings of the Sixteenth International Conferences on Very Large Data Bases,* 1990.

[Moha92] 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.

[Papa84] Papadimitriou, C., and P. Kanellakis, "On Concurrency Control by Multiple Versions," *ACM Transactions on Database Systems, 9(1), March 1984.*

[Papa86] Papadimitriou, C., *The Theory of Database Concurrency Control*, Computer Science Press, Rockville Maryland, 1986.

[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 Transactions on Computer Systems,* 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," *Procedings of the Fourth Annual Symposium on the Simulation of Computer Systems*, 1976.

[Schn90] Schneider, D., *Complex Query Processing in Multiprocessor Database Machines,* Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, September 1990.

[Silb82] Silberschatz, A., "A Multi-Version Concurrency Control Scheme with No Rollbacks," *ACM-SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August, 1982.

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

[Ston87] Stonebraker, M., "The Design of the Postgres Storage System," *Proc. Thirteenth International Conference on Very Large Database Systems*, 1987.

[Tay85] Tay, Y., N. Goodman, and R. Suri, "Locking Performance in Centralized Databases," *ACM Transactions on Database Systems*, 10(4), December 1985.

[Teor72] Teorey, T., and T Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, 15(3), March 1972.

[Wu91] Wu, K.-L., et al., *Dynamic Finite Versioning for Concurrent Transaction and Query Processing*, IBM Research Report RC 16633, IBM T.J. Watson Research Center, March 1991.