Multiversion Query Locking

Paul M. Bober
Michael J. Carey

Technical Report #1085a

June 1992

Multiversion Query Locking

Paul M. Bober
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

Multiversion two-phase locking (MV2PL) has been incorporated in some commercial transaction processing systems to support the serializable execution of queries. A drawback to this algorithm is the potentially high cost that it adds to maintain and access prior versions of data. In this paper, we present a new multiversion locking algorithm, *multiversion query locking* (MVQL), that reduces the cost of versioning by accepting weaker forms of consistency for queries than MV2PL. Nevertheless, queries are guaranteed to see transaction-consistent data. We present results from a detailed performance study that show that, under a wide range of conditions, MVQL provides higher throughput to queries and update transactions with a lower storage cost than MV2PL. In the worst case, the performance of MVQL approaches that of MV2PL.

1. INTRODUCTION

Due to the adoption of relational database technology and the increasing ability of database systems to efficiently execute ad-hoc queries, query processing is becoming an increasingly important function of transaction processing systems. Concurrency control techniques for on-line query processing, however, are still lacking. The concurrency control algorithm found in most commercial database systems, two-phase locking (2PL) [Eswa76], does not efficiently support on-line query processing. This is because 2PL causes queries to lock large regions of data for long periods of time, thus causing update transactions to suffer long delays.

A solution that avoids the data contention problem of 2PL is to extend it with versioning. Under multiversion two-phase locking (MV2PL) [DuBo82, Chan82, Chan85], prior versions of data are retained to allow queries to run against past transaction-consistent database states. The presence of versions allows queries to serialize before all concurrent update transactions, and thus queries and update transactions do not conflict. Commercial systems that employ

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

An abridged version of this paper will appear in the proceedings of the *Eighteenth International Conference on Very Large Data Bases*, Vancouver, Canada, August 1992.

MV2PL as an option include Prime's DBMS [DuBo82], DEC's Rdb/VMS product [Ragh91], Interbase, and Object Design. 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 (if in-place updates are employed).

Towards the goal of making versioning more affordable, this paper introduces a new multiversion two-phase locking algorithm, *multiversion query locking* (MVQL), that supports weaker forms of consistency for queries than that provided by MV2PL. We review these forms of consistency in Section 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. Another example is the class of 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).

The remainder of this paper is organized as follows: Section 2 reviews the various forms of consistency that are provided by MV2PL and MVQL. Section 3 describes the existing MV2PL algorithm and then presents the new MVQL algorithm as a generalization of MV2PL. Section 4 describes the simulation model used to study the performance of MVQL. Section 5 presents the results of experiments that compare MVQL to MV2PL in terms of update transaction performance, query performance, and storage cost. Lastly, Section 6 presents our conclusions.

2. FORMS OF QUERY CONSISTENCY

In the introduction, 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. In the next section, we review the MV2PL algorithm, which provides only strict consistency, and we then present the MVQL algorithm as a generalization of MV2PL that can provide any of these levels of consistency.

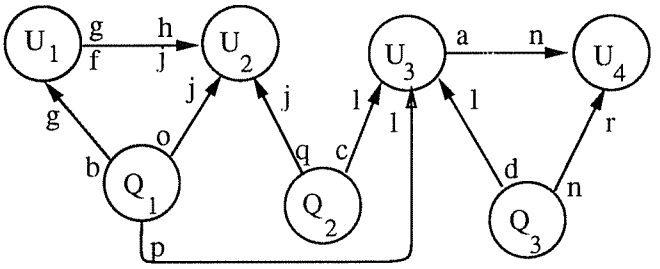
An underlying requirement that we impose is that the execution of update transactions alone must be serializable; this is guaranteed in MV2PL and MVQL by having update transactions run using dynamic 2PL. This prohibits serialization graph cycles that contain only update transactions (*update transaction cycles*).[1] Throughout the paper 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 2.1 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

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 2.1: Forms of Consistency

[1] A serialization graph consists of nodes, which represent transactions, and edges, which represent constraints on equivalent serial orderings. A path (T_i,..., 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



	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 2.2: A Schedule and Serialization Graph Illustrating Strict Consistency for Queries

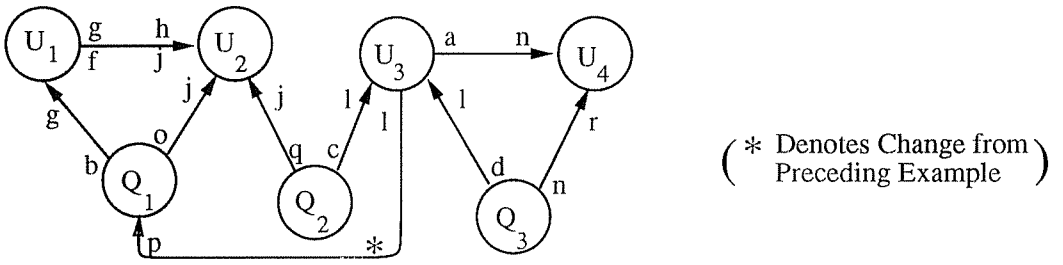
characterized by an acyclic serialization graph where the order of update transactions is consistent with their commit order. Having the update transactions observe 2PL guarantees that, in the subgraph consisting of only the actions of update transactions (and not those of queries), the partial order 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 2.2 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 2.2 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 2.3 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 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



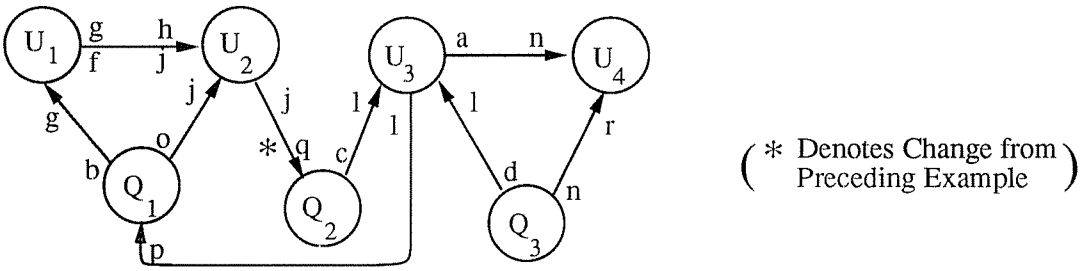
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
U_1					$R(X_0)R(Y_0)W(X_1)$													
U_2								$R(X_1)R(Y_0)W(Y_1)$										
U_3	$R(W_0)$										$R(Z_0)W(Z_1)$							
U_4													$R(W_0)W(W_1)$					
Q_1		$R(X_0)$														$R(Y_0)R(Z_1)$		
Q_2			$R(Z_0)$														$R(Y_0)$	
Q_3				$R(Z_0)$			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

Figure 2.3: A Schedule and Serialization Graph Illustrating Strong Consistency for Queries

[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 3.2.2.

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 corresponding serialization graph in Figure 2.4 illustrate this form of consistency. The schedule differs from that of Figure 2.3 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



	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 2.4: A Schedule and Serialization Graph Illustrating Weak Consistency for Queries

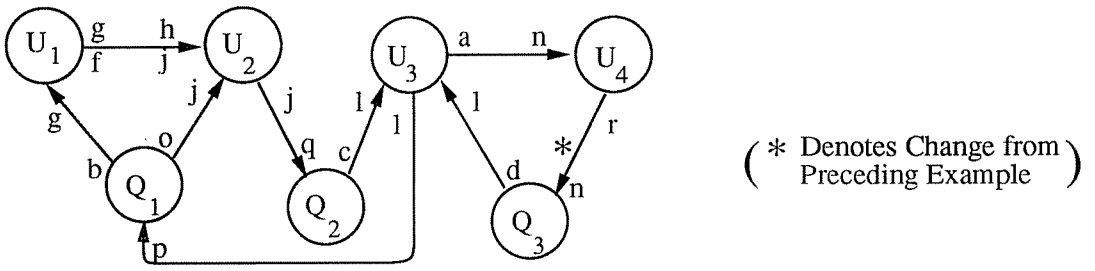
ordering (U_3, Q_1, U_1, U_2), while Q_2 has seen the ordering (U_1, U_2, Q_2, U_3). Thus, U_1 and U_3 (as well as U_2 and U_3) are ordered differently in the two queries' observed schedules.

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. In the appendix, we prove that a query has seen update consistency if i) the serialization graph contains no update transaction cycles, and ii) each single-query cycle involving the query can be broken by removing a read-write edge between update transactions.

The schedule and corresponding serialization graph in Figure 2.5 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 2.5, Q_3 sees a state of the database that would have existed if U_3 had never executed (or if it had aborted). Because 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



	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 2.5: A Schedule and Serialization Graph Illustrating Update Consistency for Queries

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).

3. MULTIVERSION ALGORITHMS

In this section, we describe the MV2PL algorithm which is currently used in several commercial DBMSs. We then present our new algorithm, MVQL, as a generalization of MV2PL. Both MV2PL and MVQL use two-phase locking for serializing update transactions. MV2PL provides only strict consistency for queries, while MVQL relaxes this by permitting a choice between update, weak, strong, and strict consistency. In the last case, MVQL is equivalent to MV2PL.

3.1. Multiversion Two-Phase Locking (MV2PL)

In MV2PL [DuBo82, Chan82], a transaction is classified at startup time as being either a *query* transaction (read-only) or an *update* transaction. When an update transaction reads or writes an object (e.g., a page or a tuple), it locks the object, as in traditional 2PL, and then accesses the most recent version. Update transactions must block when conflicts occur. When an object is written, a new version is created and stamped with the identifier of its creator. When an update transaction completes, it is assigned a commit timestamp from the incremented value of a commit timestamp counter.

When a query begins, it is assigned a startup timestamp that is equal to the current value of the commit timestamp counter. When the query wishes to read an object, it simply reads the most recent version of the object written by a transaction that was assigned a commit timestamp less than or equal to the query's startup timestamp[3]. A table is maintained to map transaction identifiers (which are stamped on object versions) to commit timestamps. Thus, the query 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 as of its starting time. As a result, queries never have to set or wait for locks in MV2PL. MV2PL provides strict consistency because all queries see a serialization order of update transactions that is consistent with their commit (timestamp) order.

3.2. Multiversion Query Locking (MVQL)

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

[3]The garbage collection mechanism must guarantee that this version is available to the query. Thus, a prior version can only be garbage-collected if there are no active queries running with a startup timestamp greater than the commit timestamp of the version's creator and less than the commit timestamp of the transaction which overwrote this version. In Section 3.2.4, we discuss two algorithms for garbage collection.

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.) We will discuss the details of set representation shortly.

As discussed in the introduction, 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, we present the rules for placing update transactions in the AFTER sets of queries for each form of consistency in the order: strict, update, weak, and strong. Recall that Table 2.1 summarizes these forms of consistency. In the next subsection we describe how these rules may be efficiently implemented.

3.2.1. 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):

(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

[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.

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):

(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]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 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)

3.2.2. 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.

3.2.3. Implementation of 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).

3.2.4. 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 is relatively inexpensive.

3.3. 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.

3.3.1. 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).

3.3.2. 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 3.2.1 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,

[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.

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 3.2.1. 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).

3.3.3. 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

[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.

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 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 3.2.1 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.

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. 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 has several subcomponents that will be described in this section. The model was implemented in the DeNet simulation language [Livn89].

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 4.1.

The second component of the application model, the source module, is responsible for modeling the external workload of the DBMS. Table 4.2 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 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 $Selectivity_{class,file}$, respectively). The actual selectivity for a given run is chosen from a distribution, $SelectivityDistr_{class}$. For update transactions, $UpdateFrac_{file}$ 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.

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 4.1: 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 4.2: The Workload Model Parameters

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.

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 4.3 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.

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 4.3: The System Model Parameters

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 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 3.2.4 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 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

[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.

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. 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 paper we discuss only performance differences that were found to be statistically significant.

Table 5.1 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.

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 5.1: System and Application Model Parameter Settings

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 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 5.1

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 5.2: Workload 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.

The parameter settings for the workload model were varied from experiment to experiment. These settings are listed in Table 5.2, and are described with each experiment.

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 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 5.1 through 5.4 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 5.1, 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 5.2, which shows the fraction of query accesses

[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.

[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.

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 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 3.2.1. 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 5.1, 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 5.3, which shows the average storage costs observed for MV2PL and MVQL during each simulation run; note that the curves show storage cost relative to the total database size. 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

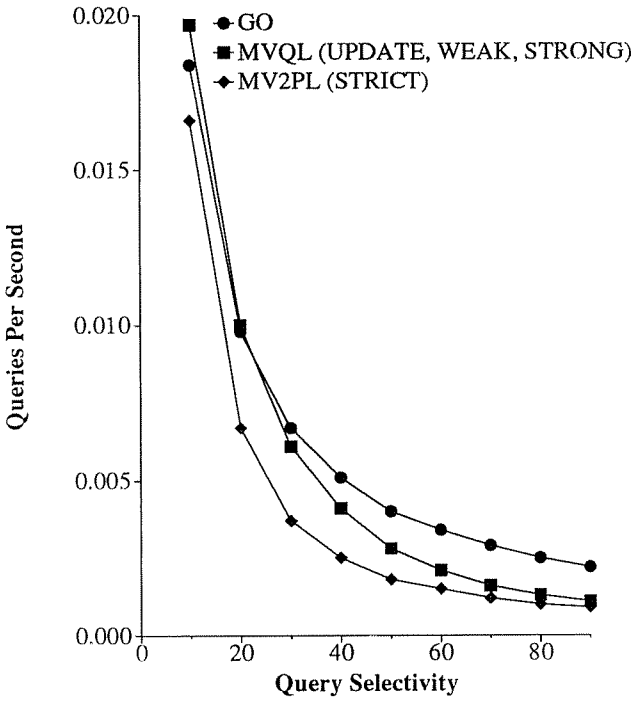


Figure 5.1: Query Throughput

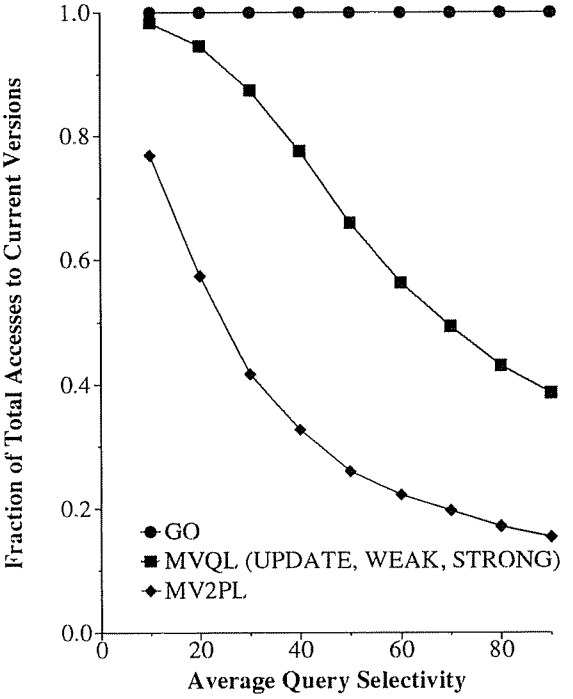


Figure 5.2: Current Version Access

(MPL_{query} = 1, MPL_{update} = 12, update transaction size = 2)

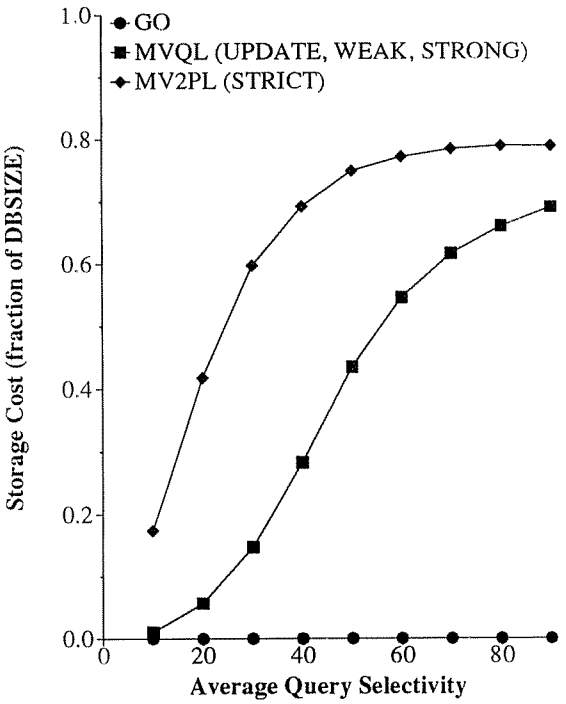


Figure 5.3: Storage Cost

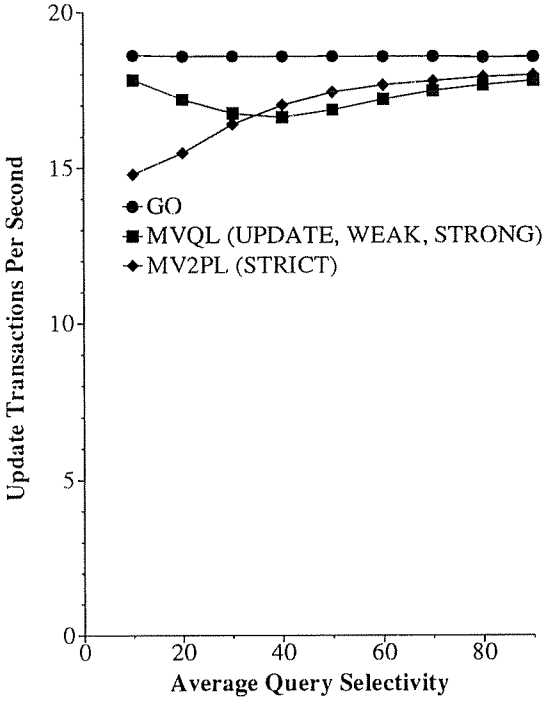


Figure 5.4: Update Transaction Thoughput

(MPL_{query} = 1, MPL_{update} = 12, update transaction size = 2)

all update transactions that run during a query's lifetime into its AFTER set, while MVQL does not.

Finally, we turn our attention to the update transaction throughput, shown in Figure 5.4. 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.

5.2. 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 3.2.1. 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 5.5. 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 3.2.1, 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]. Due to space limitations, 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 5.6 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 5.6 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.

Due to space limitations, we do not show the update transaction throughput for this experiment, but we summarize the results here. MVQL achieved an update transaction throughput that ranged between 98% and 92% of that achieved by GO Processing (as the update transaction size was increased along the range from 1 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

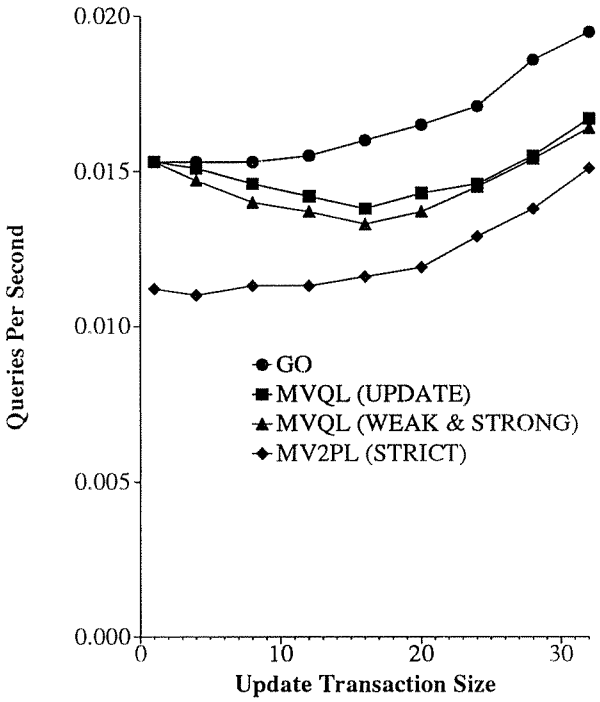


Figure 5.5: Query Throughput

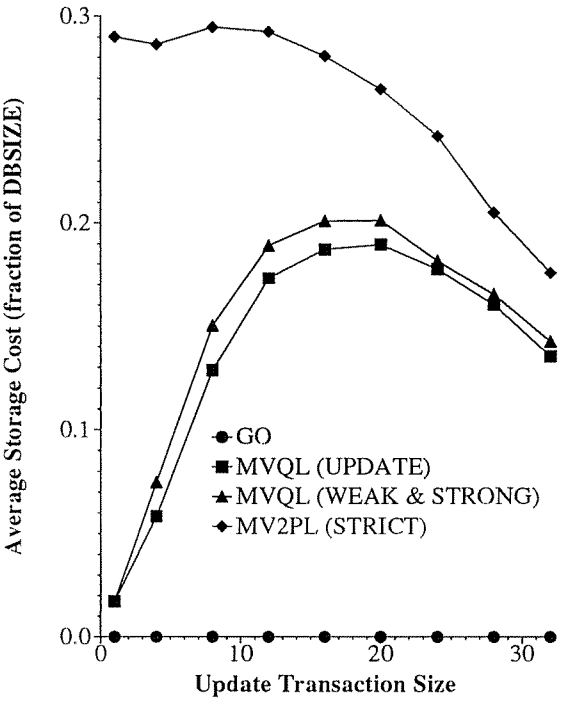


Figure 5.6: Storage Cost

($MPL_{query} = 1$, $MPL_{update} = 12$, $Selectivity_{Query} = 25\%$)

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.

5.3. 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 5.7, we see that the query throughput for all algorithms rises as the number of query terminals is increased, while in Figure 5.8, there is a corresponding decline in update transaction throughput. The rise in query

throughput is linear in the number of query terminals, and not in their fraction of the overall number of terminals. This result is due to the system shifting its effort from update transaction processing to query processing, which increases the fraction of clean pages in the buffer pool and reduces the buffer cleaning work that must be done by queries.

In Figure 5.7, 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 3.2.1) 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 5.9, diverges only very slightly from that of weak (and update) MVQL as the number of query terminals is increased.

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

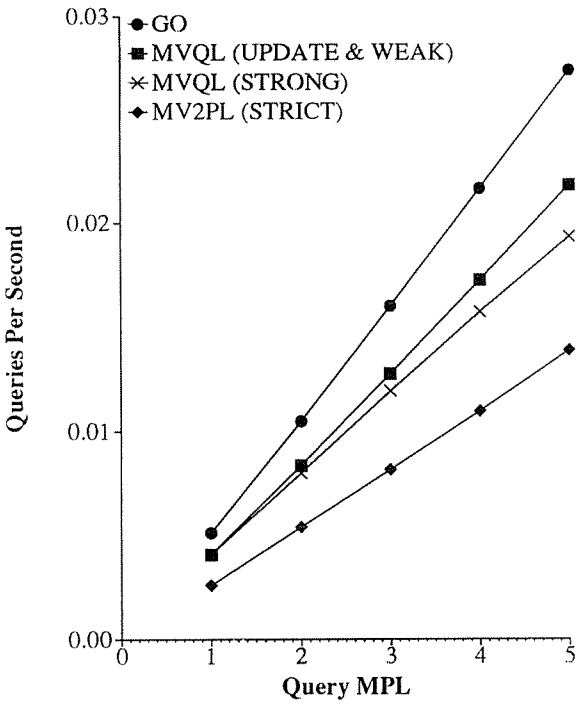


Figure 5.7: Query Throughput

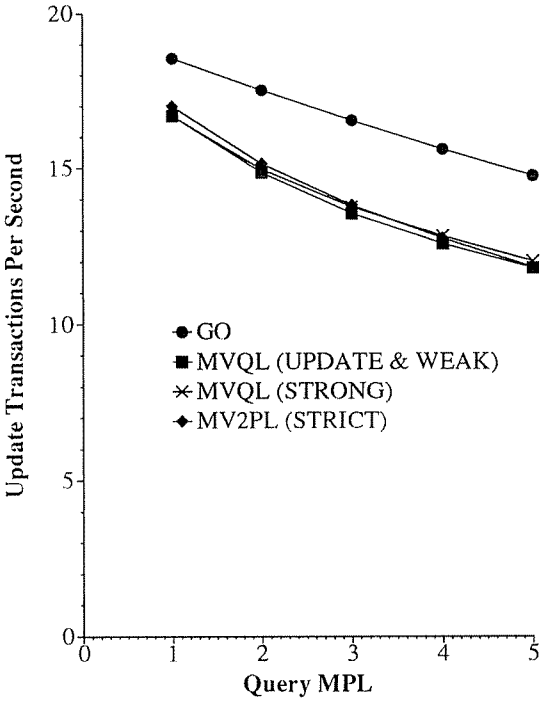


Figure 5.8: Update Transaction Throughput

(MPL_{update} = 12, $Selectivity_{Query}$ = 40%, update transaction size = 2)

initially expect.

5.4. Discussion

In this section, we have presented the results of a preliminary 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 alternative to MV2PL for this sort 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).

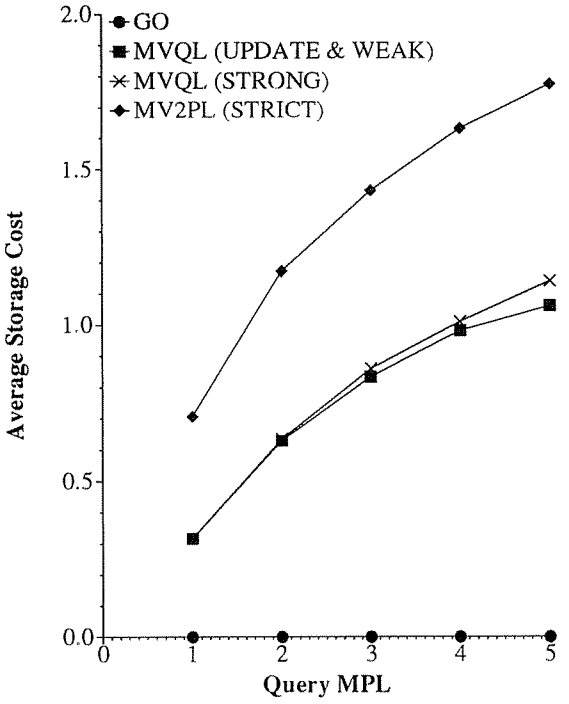


Figure 5.9: Storage Cost
(MPL_{update} = 12, $Selectivity_{Query}$ = 40%, update transaction size = 2)

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.

6. CONCLUSIONS

In this paper, 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 preliminary 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. In the future, we plan to extend our preliminary analysis of MVQL to a more comprehensive set of workloads and version management schemes. In particular, we did not look at workloads with skewed access patterns, nor did we consider a reverse-chaining version management scheme, though we fully expect MVQL to perform at least as well under such conditions.

ACKNOWLEDGEMENTS

The authors would like to thank one of the anonymous VLDB-92 referees whose expert and careful review of the paper revealed two errors in the original MVQL algorithm and presentation. We would also like to thank V. Srinivasan for making helpful comments on a draft of this paper.

REFERENCES

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

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

[Bobe92] Bober, P. and M. Carey, "On Mixing Queries and Transactions via Multiversion Locking," *Proc. of the 1992 IEEE Data Engineering Conf.*, 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 Trans. on Knowledge and Data Eng.*, 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.

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

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

[Gray81] Gray, J., P. Homan, H. Korth, and Obermarck, R., *A Strawman Analysis of the Probability of Waiting and Deadlock in a Database System* Research Report RJ3066, IBM San Jose, Feb. 1981.

[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.

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

[Moha90] Mohan, C., et al. "Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques," *Proc. International Conference on Extending Database Technology*, 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 Trans. 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.

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

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

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

[Wu92] Wu, K.-L., P.S. Yu, and Pu, C., "Divergence Control for Epsilon Serializability," *Proc. of the 1992 IEEE Data Engineering Conf.*, 1992.

APPENDIX

Theorem 1: *A query Q sees update consistency (i.e., it serializes with all update transactions that it has seen) if i) the serialization graph contains no update transaction cycles, and ii) each single query cycle involving Q can be broken by removing a read-write edge between update transactions.*

Proof: Let G be the serialization graph consisting of Q and all update transactions. Assume that any and all single-query cycles can be broken as described and that the serialization graph contains no update transaction cycles. If G is acyclic, then Q serializes with the entire set of update transactions (and therefore it must also serialize with the subset that it has actually seen). Otherwise, let E be a minimal set of read-write edges that can be removed to break all single-query cycles involving Q in G. Let G' be the graph that results from removing all update transactions from G that exist as the origin of some edge in E. G' must be acyclic since i) all single-query cycles involving Q were broken by removing the edges in E, ii) Q is the only query in the graph, and iii) no update transaction cycles are permitted. We must now show that in deriving G' from G, no update transaction seen by Q is removed.

Let e be an arbitrary edge in E of the form (U_i, U_j). Since E is a minimal set, there must be a path in both G and G' from Q to U_i and one from U_j to Q (otherwise, removing placing e in E would be redundant). Thus, $U_i < Q < U_j$ is implied by both graphs. Furthermore, since G' is acyclic, all paths between U_i and Q in G must be dependent on at least one (read-write) edge in E. Since there are no paths in G from U_i to Q that traverse only write-read edges, Q has seen none of the updates of U_i. Because the edge e was chosen from the set E arbitrarily, we know that Q has not seen any of the updates issued by the transactions that were removed from G to form G'. Furthermore, since we also know that G' is acyclic, Q must then serialize with the update transactions in G' (which includes all of the update transactions that it has seen). □