Avoiding Contention between Reads and Writes Using Dynamic Versioning

Sreenivas Gukal
Edward Omiecinski
Umakishore Ramachandran

GIT–CC–94/13

February 20, 1994

Abstract

Abstract

In this paper, we discuss a new approach to multi-version concurrency control, called *Dynamic Versioning*, that avoids the data contention due to conflicts between Reads and Writes. A data item is allowed to have several committed versions and at most one uncommitted version. A conflict between a Read and a Write is resolved by imposing an order between the requesting transactions, and allowing the Read to access one of the committed versions. The space overhead is reduced to the minimum possible by making the versions dynamic; a version exists only as long as it may be accessed by an active transaction. *Conditional lock compatibilities* are used for providing serializable access to the multiple versions. The results from simulation studies indicate that the dynamic versioning method, with little space overhead (about 1% the size of the database), significantly reduces blocking (by 60% to 90%) compared to single-version two-phase locking. Lower blocking rates increase transaction throughput and reduce variance in transaction response times by better utilization of resources. This approach also reduces starvation of short transactions and subsumes previous methods proposed for supporting long-running queries. The dynamic versioning method can be easily incorporated into existing DBMS systems. The modifications required for the lock manager and the storage manager modules to implement dynamic versioning are discussed.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

1 Introduction

Two-phase locking is the industry-standard concurrency control mechanism in database management systems. In this method, transactions acquire read (shared) or write (exclusive) locks before accessing a data item, where read and write locks are incompatible. Two-phase locking, being pessimistic, tends to block a transaction at the first instance the transaction requests a lock on a data item already held by another transaction in an incompatible mode. Blocking transactions to resolve data conflicts reduces the concurrency in the system. A lower concurrency level results in lower transaction throughput and higher variance in transaction response times. One of the serious consequences of blocking is that when short and long transactions are executed concurrently, the long transactions may starve the short transactions. Starvation increases the response times of short transactions to almost as much as that of the long transactions.

In this paper, we present a record-level multi-versioning concurrency control method, called *Dynamic Versioning*, to avoid the data contention between Reads and Writes. In a transaction processing environment, the percentage of Writes is lower than the percentage of Reads. Hence, the data contention due to conflicts between Reads and Writes is much more than that caused by Write-Write conflicts. In the dynamic versioning approach, each Write creates a new version of the data item. A conflict between a Read and a Write at a data item is dynamically resolved by creating a dependence relation between the requesting transactions. The transaction requesting the Read becomes a predecessor of the transaction requesting the Write and reads an earlier committed version of the data item. A dependence graph maintains the dependence relations among the transactions. This graph is used to ensure serializability and provide transactions a single version view of the database. Multi-version algorithms incur the overhead of maintaining additional versions. Our approach reduces this overhead to the minimum possible by keeping only the versions necessary for avoiding aborts and providing serializability.

One of the important concurrency control problems [Dewi 92] is to prevent the execution of queries (read-only transactions) from affecting the concurrent update transactions. Multi-versioning algorithms have been suggested as a way to solve this problem [Pira 90]. However, in large databases, a long-running query may require maintaining a large number of versions. In this paper, we propose three dynamic versioning techniques to support queries based on their lengths and access patterns and show that these techniques require 50% to 90% less space than other proposed methods.

The dynamic versioning method is a variant of two-phase locking and hence can easily be incorporated into existing DBMS systems. We explain how the lock manager can be changed to allow multiple read-locks and a write-lock simultaneously on the same data item. We also describe how the storage manager can efficiently maintain the multiple versions in the database and provide indexed access to the versions.

1.1 Related Work

A two version two-phase locking algorithm is proposed by Bayer et al. [Baye 80]. The algorithm is based on the observation that, in a shadowing environment, a copy of

the data item is made and the updates are done to the copy. Hence, to avoid blocking, readers can be allowed to read the before value. A dependence graph is maintained to ensure serializability. However, the algorithm needs to be modified to apply in a system that uses logging, where accessing the before values is inefficient. This algorithm always grants read access immediately, which results in aborts if the transactions are not serializable at commit time. Allowing a transaction to read the before value of a data item may also result in an abort if the transaction later decides to update the data item. Since only two versions of a data item are allowed, a transaction requesting a write lock on a data item will be blocked if none of the two existing versions of the data item can be deleted because of readers. They also do not discuss issues such as version maintenance and access, secondary indices for the versions and space management. A simulation study [Pein 83] of this algorithm concluded that it performs almost the same as single-version two-phase locking for most of the work loads. A similar method [Stea 81], which uses time-stamp based deadlock prevention, has been proposed for distributed databases.

Two-version and multi-version two-phase locking methods are also presented in [Bern 87]. The two-version method is similar to that in [Baye 80]. The multi-version algorithm assumes that for a data item there is a single committed version, used by readers, and several uncommitted versions simultaneously being updated by the writers. This view is unrealistic since usually a writer has to first read a data item before deciding to update [Agar 87a]. Since, in such a case, all the writers would have read the committed version, only one of the concurrent writers to a data item can be allowed to commit to ensure serializability. The dynamic versioning method has a generalized lock compatibility matrix to allow writers to read the data item first. The multiple versions of a data item here consist of several committed versions and one uncommitted version. Maintaining several committed versions allows updating transactions to commit as soon as they complete, instead of waiting for readers of earlier versions to end.

Multi-versioning methods to support queries are proposed in [Bobe 92] and [Moha 92]. Both methods treat all queries in a similar manner, irrespective of the lengths of the queries or amount of data accessed. In Section 5, we show that in some cases the above two methods may result in maintaining a large number of unnecessary versions. We compare these two methods with the dynamic versioning method in that section.

The rest of the paper is organized as follows. Section 2 outlines the dynamic versioning method. The different states of the data items, the corresponding lock modes and the lock compatibility matrix are explained. Section 3 and Section 4 present the results of the simulation studies, which compare the performance of dynamic versioning with that of two-phase locking. Section 3 studies the effect of data contention alone and shows how blocking of transactions can be reduced as much as 90% using dynamic versioning. Section 4 includes resource contention besides data contention, and presents the performance of dynamic versioning in different environments. Section 5 considers running queries and transactions together. There we show that the dynamic versioning method is more flexible and efficient than the other techniques proposed for supporting queries. The last section summarizes our results.

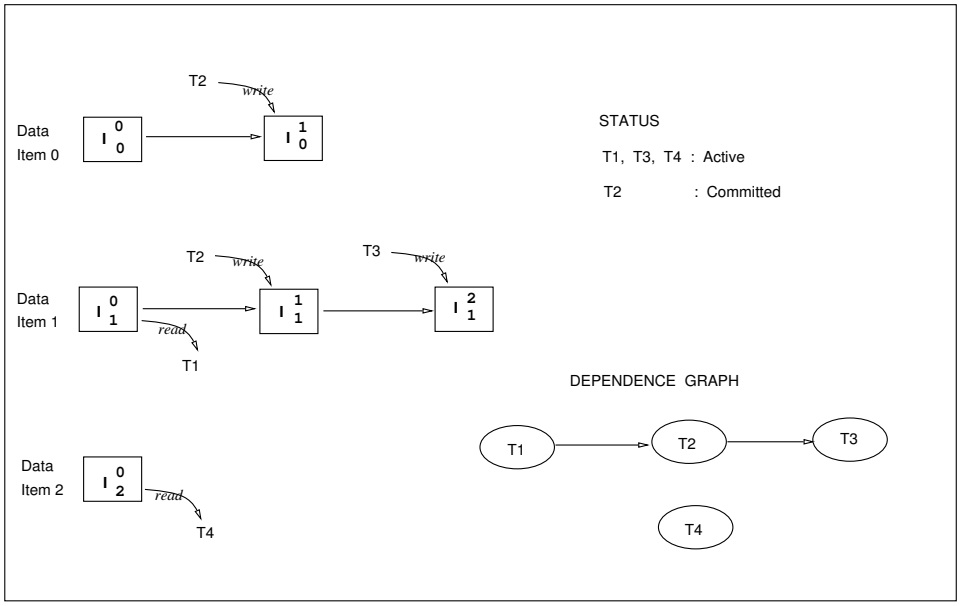

Figure 1: Dynamic versions and the dependence graph

2 Dynamic Versioning

The dynamic versioning method maintains record-level versions to eliminate, where possible, the conflicts between Reads and Writes. Each transaction with a write lock on a data item creates a new version of the item and performs all the modifications on that version. After a transaction commits, all the versions it created become committed. An earlier version of a data item may be replaced by a later, committed version, as soon as there is no possibility of any other transaction accessing the earlier version. Figure 1 shows four transactions (T1, T2, T3 and T4) accessing three data items (I_0, I_1 and I_2). T2 created versions I_0^1 and I_1^1 and committed. The earlier versions (I_0^0 and I_1^0) can be deleted as soon as T1 (a predecessor of T2), which may read the earlier versions, commits.

A data conflict between two transactions is resolved by creating a dependence relation (successor / predecessor) between the two transactions. The dependence relations among the transactions form a partial order, which is used to check serializabilty constraints. A conflict between a Read and a Write may be resolved, without blocking one of the transactions, by making the transaction requesting the Read a predecessor of the transaction requesting the Write. The Read can access one of the earlier committed versions of the data item, while the Write creates and updates a new version. Allowing several read-locks and a write-lock simultaneously on the same data item reduces the data contention. In Figure 1, if T4 requests a read-lock on item I_1, T4 is made a predecessor of the current writer, T3, and allowed to read the earlier version I_1^1.

Based on the above discussion, a data item can be in one of the following states.

1. **Base State** The data item has only one version. A requesting transaction is allowed to read this version (e.g., item I_2).

2. **Versioned State** The data item has more than one version. Two possibilities exist here :

    1. All versions are committed. A requesting transaction is allowed to read one of the versions based on the serializabilty constraints (e.g., item I_0).
    2. The last version is uncommitted. A requesting transaction is allowed to read one of the committed versions if the serializabilty constraints are not violated. Otherwise the transaction is blocked until the last version is committed (e.g., item I_1).

All the versions of a data item are placed in the same page in the database. This reduces the overhead of accessing the earlier versions. To reduce the space requirements, only the incremental difference of a version may be stored. Usually a database page contains some free space to accommodate growing tuples or for inserting new tuples [Gray 93]. This space may be used for holding the dynamic versions. In the worst case, when there is no free space in the page, a page split may be required. Under this scheme, only those pages that contain data items which are updated frequently, end up with additional free space. Data pages that contain items which are usually read but rarely updated do not have any space overhead. When a transaction wants to create a new version in a data page, it first deletes the old versions in the page that are not accessible to any active transaction. This method of garbage collection is inexpensive since creating the new version is anyway going to dirty the page.

Usually a transaction first reads an item (i.e., gets a read-lock) before deciding to modify the item (i.e., upgrading the read-lock to write-lock). This might result in rollbacks, since different transactions may acquire read-locks at the same time on an item and then decide to modify the item. To avoid this situation, the dynamic versioning method contains an additional lock mode called update-lock mode [Gray 93]. A transaction which might modify an item, first acquires an update-lock on the item. If the transaction decides to modify the item, it upgrades the update-lock to a write-lock, else it downgrades the update-lock to a read-lock. To avoid cascading aborts, a request for an update lock on a data item is granted only after the transaction that created the last version of the data item has committed. In Figure 1, suppose T4 requests an update-lock on item I_1. Here T4 is blocked until T3, the current writer, commits. However, if T4 requests an update-lock on item I_0, it can be granted immediately since T2 has committed.

Four lock modes are required to support the different states and the versions in those states, read-lock(r) to read the data item, update-lock(u) to check if the data item satisfies the update conditions, write-lock (w) to create an uncommitted version and commit-lock (c) to denote that a version is committed. The lock compatibility matrix, which allows the required combinations, is given in Table 1. Here, (+) indicates compatibility and (−) indicates that the requesting transaction has to wait due to incompatible lock modes. We also have (±) to denote conditional compatibility, where the requesting transaction is granted read access on the condition that serializabilty constraints are not violated. Note that here, though (+) indicates compatibility, granting the request may sometimes result in a cycle in the dependence graph. In Figure 1,

Requested Mode	Granted Mode			
	r	u	w	c
r	+	±	±	+
u	+	−	−	+
w	+	−	−	−
c	+	−	−	−

Table 1: Lock compatibility matrix

suppose transaction T1 requests for an update-lock on item I_0. The lock modes (for I_0) are compatible. Yet granting the update-lock makes T1 a successor of T2, which violates the order already established between T1 and T2 at item I_1. Hence T1 has to be aborted.

The dynamic versioning method uses a dependence graph to store the dependence relations created while resolving data conflicts among transactions. The dependence graph is a directed acyclic graph, where nodes represent transactions and edges denote the successor/predecessor relationships among transactions. A transaction T1 is a predecessor of another transaction T2, if either T1 reads an earlier version of a data item modified by T2, T2 reads a later version of a data item modified by T1, or T2 is waiting for a lock on a data item currently held by T1 in either u-lock or w-lock mode. When two transactions enter into a data conflict, the dynamic versioning method selects the dependence relation that avoids blocking, in contrast to normal two-phase locking where the order is decided based on a first-come-first-serve policy. A transaction is blocked only if allowing its request creates a cycle in the dependence graph. Figure 1 shows a dependence graph resulting from resolving conflicts among four transactions. A topological sort of the acyclic dependence graph generates a serializable order of the transactions.

Hierarchical locking can be easily supported using the dependence graph. Here, it is possible for transactions to hold locks of different granularities in the same object. If a transaction T requires lock escalation from a number of finer granular locks (e.g, record-locks) to a coarser granular lock (e.g., a table-lock), the lock manager first determines the set of transactions that had accessed the coarser granular object. The transactions in this set that are not already related to T, are made either successors or predecessors of T, provided serializability constraints are not violated. If the request of T conflicts with some finer granular write-locks already held by other transactions, T is blocked until the conflicting transactions complete. For example, in Figure 1, suppose T3 requests for a write-lock on the entire table. Here, T1 and T2 are already predecessors of T3. T4 can also be made a predecessor of T3, and T3 can be granted the write-lock immediately.

Transactions execute in the dynamic versioning system just as they do in a system with single-version two-phase locking. The lock manager module decides which version of a data item a requesting transaction is allowed to read. The storage manager module maintains the dynamic versions and allows transactions to use only the versions

permitted by the lock manager. The storage manager also performs garbage collection tasks. Appendix A discusses how these two modules are efficiently implemented.

3 Measuring Data Contention

The dynamic versioning method is a variant of the two-phase locking algorithm. We compare the performance of the dynamic versioning method with that of the single-version two-phase locking method, which is the industry standard. For ease of reference, we denote the dynamic versioning method as DV and the single-version two-phase locking method as 2PL. The main difference between the two methods is how the data contention is resolved. To study the effect of the reduction in data contention better, we ignore resource contention in the set of simulations presented in this section. The effect of resource contention is examined in Section 4.

3.1 Design of the Simulation Experiments

The database is modelled as having a number of data pages, each page containing a fixed number of records. Transactions contain a series of record references. Each reference is made up of a record identifier, a page identifier, the type of access desired and the amount of operation time the transaction should spend after accessing the record. The operation time is used to model the CPU utilization for accessing the data item, performing any computations using the fetched value and for determining the new value for the data item in case of updates. For each reference, a transaction first makes a request to the lock manager for the corresponding record lock. If the lock is granted, the transaction requests the page latch. After getting the page latch, the transaction accesses the value of the record and releases the page latch. It then waits for the amount of operation time specified in the reference. If the transaction decides to update the record, it requests the lock manager to upgrade the lock, requests again for the page latch, updates the record and releases the page latch. A transaction releases all the locks it acquired at the commit time. This basic operation is the same for both the DV and the 2PL methods.

The two methods can be compared by varying two parameters, the amount of data contention and the percentage of updates. Both these parameters, together, determine the amount of blocking in the system. Data contention depends on the multiprogramming level, the size of the database and the number of references per transaction. Insofar as the observed results, all these aspects that affect the data contention are equivalent. Therefore, we report the results of varying the size of the database, while keeping the multiprogramming level and the number of references per transaction constant[1]. We consider large enough database sizes to demonstrate that DV is applicable to actual databases (250,000 to 1,000,000 records, in increments of 250,000 records). The percentage of updates is important since the main goal of DV is to reduce blocking by allowing a writer and several readers to work on an item simultaneously. Earlier simulation studies (e.g., [Agar 87a], [Agar 87b], [Pein 83]) considered about 25% updates. Here we vary the percentage of updates from 10 to 50.

[1]We also used a uniform distribution of +30% to −30% for the number of references. The results are similar to the case when the number of references is constant. We report the results of having constant number of references, because the variance in the transaction response times makes more sense here.

Simulation Parameter	Value
Data Base Size	250k, 500k, 750k and 1000k records
Records per Data Page	20
Percentage of updates	10%, 25%, 40%, 50%
References per Transaction	100
Multi-programming Level	50
Lock Acquisition (Release) Time	500 μs
Latch Acquisition (Release) Time	50 μs

Table 2: Simulation parameter settings

The other fixed parameters relevant to the simulation are summarized in Table 2. The costs for lock and latch operations are chosen based on our notion of roughly what realistic values might be. The multiprogramming level and the number of references per transaction are selected to provide reasonable level of data contention without resulting in a high number of aborts. We assume an eighty-twenty reference pattern (i.e., 80% of the references go to 20% of the database), since an uniform reference pattern does not model hot-spot pages. The operation time (given by the record reference) is assumed to be uniformly distributed between zero and ten milliseconds.

Each of our simulation runs uses a set of 1000 transactions. The same sets of transactions are used for both 2PL and DV. For each set of input parameters, the system is simulated on a number of transaction sets until the 90% confidence interval for the transaction throughput is within a few percentage points. The simulation is stopped every 50ms to gather statistics to determine the average number of transactions blocked, the average number of record versions, the average number of versions in a data page and the average number of edges in the dependence graph. These statistics allow us to compare the two methods along several metrics.

CSIM [Schw 90], a process-oriented discrete simulation package, is used for all the simulations. Each transaction is modelled as a separate CSIM process. The number of transactions in the system is determined by the degree of multiprogramming. A closed queuing simulation model is followed. When a transaction is completed, a message is sent to the transaction generator to create and release a new transaction into the system.

3.2 Results and Analysis

Figure 2 shows how the average blocking (i.e., average number of transactions blocked) varies with the data contention and the percentage of updates. The figure shows the results for the lowest and highest data contention environments (i.e., 1000k and 250k records). The results for the other two environments are in between. The average blocking in either method increases with both the data contention and the percentage of updates. DV has much lower blocking rates than 2PL. The percentage difference in blocking between the two methods is considerable at low percentage of updates for high

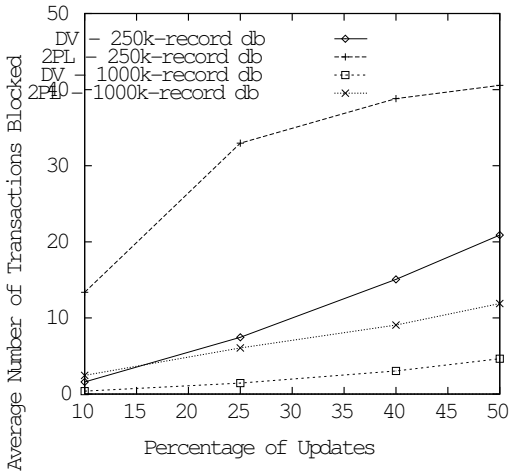

Figure 2: Blocking rates

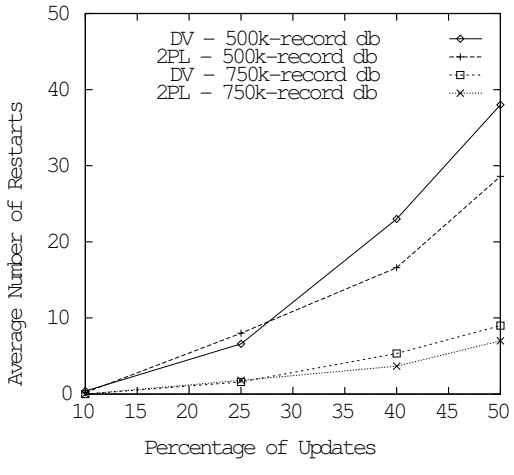

Figure 3: Effect on restarts

contention and at higher percentage of updates for low contention. The difference in blocking between the two methods initially increases with the percentage of updates. As the number of transactions blocked reaches the multiprogramming level, the difference starts reducing. When a high percentage of the transactions are already blocked, further increase in data contention results in only a slight increase in the blocking. Note that at zero percent (and hundred percent) updates, the average blocking for both methods would be the same.

Figure 3 gives the number of restarts (due to deadlocks) for the two methods. Only the results for the 500k-record and 750k-record databases are shown. At low data contention (1000k-record database), the number of restarts for both methods is less than 0.5%. At high contention (250k-record database) the number of aborts ranges from 10% to 20% for both methods, which is rare in an actual database. There are two factors which affect the number of restarts in DV as compared to 2PL. The latter, being pessimistic, blocks a transaction upon the first read-write, write-read or write-write conflict. DV, on the other hand, tries to resolve the read-write and write-read conflicts, if possible, by dynamically imposing an order on the contending transactions. This sometimes results in averting what would be a deadlock in 2PL. For example, suppose transaction T1 has a w-lock on item R1 and tries to acquire an r-lock on item R2. At the same time, transaction T2, which has a w-lock on R2, requests for an r-lock on R1. 2PL would abort one of the transactions to maintain serializability. However, in DV, it is possible create an order on the two transactions (say T2 is a successor of T1). Hence T1 can read the earlier value of R2 and proceed, while T2 is blocked. Such a resolution is beneficial if there is very little chance of T1 and T2 again entering into a conflict for a different item. If T1 and T2 again conflict, one of them may have to be aborted if the earlier order cannot be maintained. Hence at low data contention or low percentage of updates (which means less chance of repeated conflicts), DV has lower number of restarts. At higher data contentions and higher percentage of updates, DV results in more restarts. Note that in an actual system aborts are rare [Gray 93]. In such an environment, DV will have the same or lower number of transaction aborts

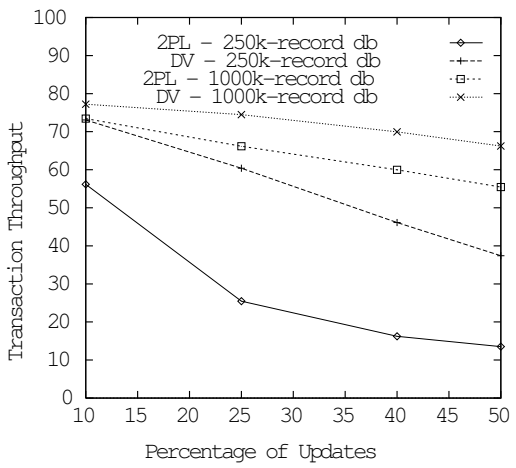

Figure 4: Transaction throughput

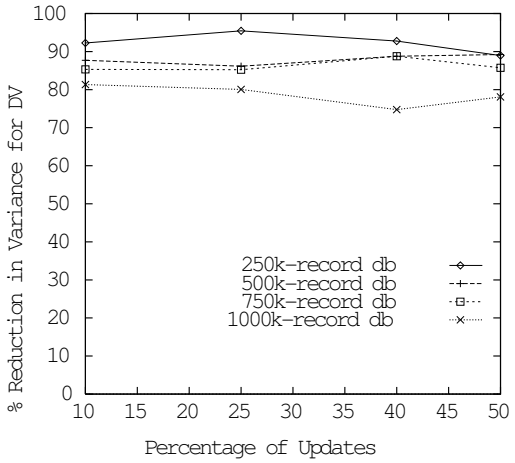

Figure 5: Transaction response time variance

Percentage of Updates	Number of Records with Dynamic Versions			Maximum Versions (for the entire database)
	1 Version	2 Versions	3 Versions	
10	315.85	0.34	0.00	477
25	956.54	4.50	0.01	1495
40	1456.10	10.06	0.05	2522
50	1601.12	10.47	0.07	2669

Table 3: Number of records with dynamic versions (250k-record database)

than 2PL, while reducing the blocking considerably.

Figure 4 presents the transaction throughputs for the two methods. The results for the 250k-record and 1000k-record databases (highest and lowest data contention of the environments considered) are shown. The figures for the other two databases fall in between. Comparing Figure 4 with Figure 2 shows that the improvement in the transaction throughput mirrors the reduction in blocking.

Smaller blocking rates have an important side-effect; they reduce the variance in the transaction response times. Figure 5 contains the percentage reduction in the variance in transaction response times in DV as compared to that in 2PL. The reduction is considerable, as high as 95% in some cases.

Tables 3 and 4 summarize the observed number of records with multiple versions for DV. The numbers for the smallest (250k-record) database are presented since it has the highest data contention. Table 3 lists the average number of records with one, two and three dynamic versions. The number of records with more than one dynamic version is very small. There are never more than three dynamic versions for any record. The last column in Table 3 shows the maximum number of dynamic versions (for all the records combined) that are stored at any time in the database. The maximum space overhead, even at 50% updates, is about 1% of the database size. Table 4 gives the

Percentage of Updates	Number of Pages with Dynamic Versions					
	1 Version	2 Versions	3 Versions	4 Versions	5 Versions	6 Versions
10	284.44	10.77	0.51	0.01	0.00	0.00
25	770.68	95.54	9.83	1.02	0.08	0.00
40	1030.04	181.59	28.16	3.42	0.33	0.06
50	1100.09	212.12	36.10	5.03	0.46	0.06

Table 4: Number of pages with dynamic versions (250k-record database)

number of data pages containing dynamic versions. Given that there are 12,500 data pages (250,000 records and 20 records per page), the fraction of pages with more than two dynamic versions is quite small.

DV requires a dependence graph, while 2PL usually uses a waits-for graph. The overhead for the graphs has two components, the cost of adding and deleting edges from the graph and the cost of checking the graph for cycles. The number of edges in the case of the waits-for graph is fixed (each transaction has as many edges as the number of references). In the dependence graph, each transaction is represented as a node and each order established due to a conflict between two transactions is denoted by an edge. The number of edges in a dependence graph is a measure of the data contention in the system. A high level of data contention is not allowed in an actual database system, since it increases the number of aborts. At reasonable levels of data contention, where aborts are rare (less than 2%), the number of edges in a dependence graph[2] is far lower than the number of edges in a waits-for graph. For the environments considered, there are 50 transactions concurrently executing in the system, with 100 references per transaction. The average number of edges for the dependence graph varies from (3.5) for the 1000k-record database with 10% updates to (57.6) for the 250k-record database with 50% updates. These numbers compare well with the average number of edges ($2500 = 100 * 50 / 2$) for the waits-for graph. The cost of checking for cycles depends both on the number of nodes and the number of edges. Dependence graph, with lower number of nodes and lower number of edges, has 10% to 50% less overhead for checking cycles than waits-for graph.

4 Adding Resource Contention

In an actual database environment, contention for resources affects the metrics studied in Section 3. Considering only data contention, we found in Section 3 that DV has much lower blocking rates than 2PL, which result in higher transaction throughput and lower variance in transaction response times. Here we would like to determine if the lower blocking rates translate into better performance in the presence of resource contention. There are numerous studies in the literature comparing different concurrency control methods. Usually these comparisons are between methods that favor blocking (pessimistic algorithms) and the methods that reduce blocking at the cost of

[2]Note that it is theoretically possible to have (N-1)! edges, with each edge connecting two different nodes, in a graph with N nodes. In a dependence graph, the system will start thrashing far before this limit is reached.

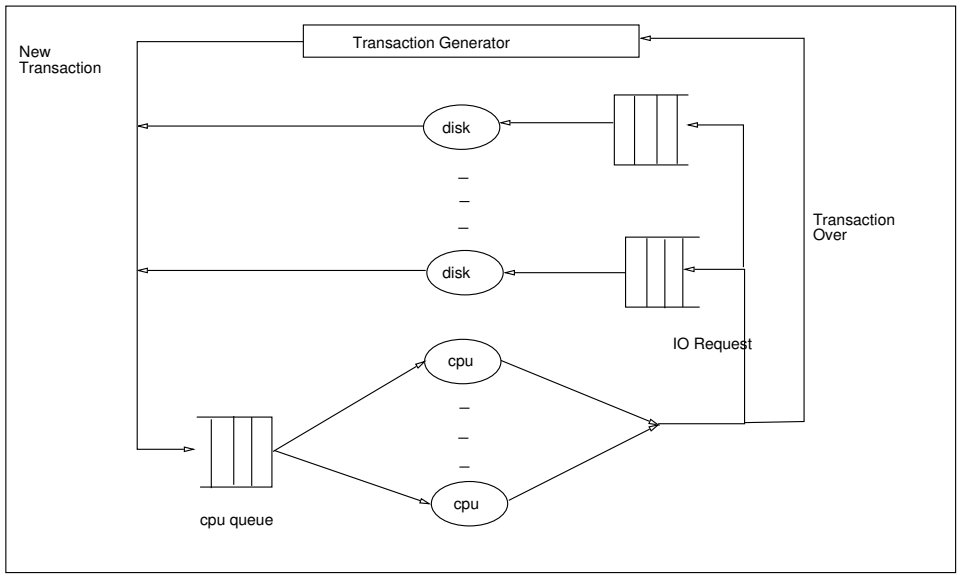

Figure 6: Closed queuing simulation model

more aborts (optimistic). The significant difference here is that we are comparing two similar algorithms (both variants of two-phase locking), with comparable number of aborts but one method with far lower blocking rates than the other.

4.1 Design of the Simulation Experiments

Figure 6 shows the closed queuing simulation model used. Resources are modelled in terms of resource units (similar to [Agar 87a] and [Agar 87b]). Each resource unit contains one CPU and two disks. The number of resource units is varied to change the resource contention. A transaction, on initiation, joins the CPU queue. Transactions from the CPU queue are assigned to an available CPU on a first-come-first-serve basis. For each data reference, a transaction first acquires the appropriate lock on the data item and a latch on the corresponding data page. If either is unavailable, the transaction is blocked. The transaction becomes active again and joins the CPU queue after acquiring the lock (or latch). Data pages are assumed to be uniformly distributed among the disks. A transaction, requiring a data page, checks if the data page already exists in the main memory. If not, the transaction releases the CPU and joins the appropriate I/O queue. Upon I/O completion, the transaction joins the CPU queue again.

Table 5 lists the simulation parameters which have different values than those used in the previous section. The resource contention is varied by letting the number of resource units range from 1 to 8. The data contention is modified by selecting different database sizes (100k, 200k and 300k records). The multiprogramming level and the references per transaction are chosen such that having only one resource unit models high resource contention, and having eight resource units models low resource contention.

Simulation Parameter	Value
Number of Resource Units	1, 2, 4 and 8
Data Base Size	100k, 200k and 300k records
Max. Records per Data Page	25
Page Fill Factor	80%
Percentage of updates	25%
References per Transaction	100
Multi-programming Level	20
Disk Access Time	30ms
Main Memory Size	1000 Pages

Table 5: Simulation parameter settings

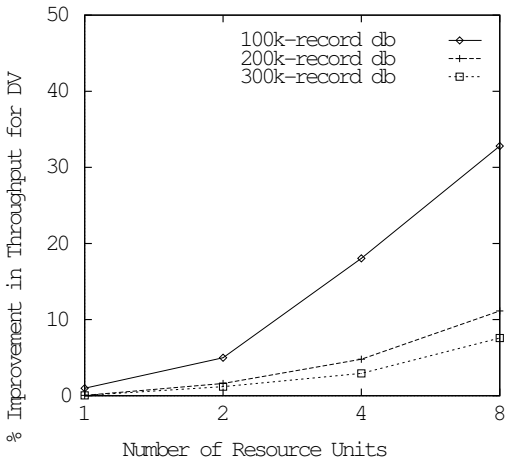

Figure 7: Transaction throughput

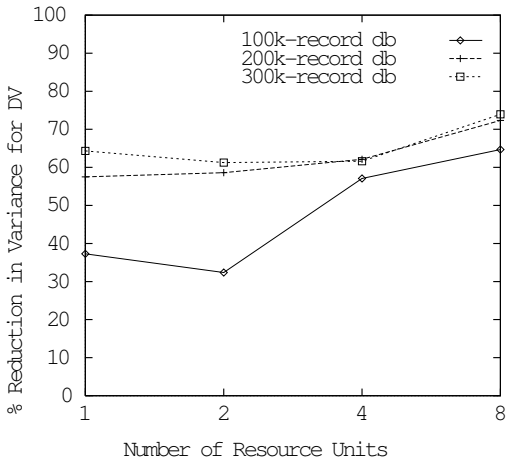

Figure 8: Transaction time variance

4.2 Results and Analysis

Figure 7 shows how the percentage improvement in the transaction throughput for DV (with respect to 2PL) varies with the resource contention. For small number of resource units (i.e., high resource contention) both methods perform similarly. As the number of resource units is increased (i.e., as resource contention is reduced), DV starts performing better. At high resource contention, one of the resources is being utilized nearly 100% by both the methods. Lower blocking rates in DV only result in longer queues for the bottleneck resource. Under such circumstances, transactions end up waiting for resources in DV, as opposed to locks in 2PL. As the resource contention is reduced, data contention becomes the dominant factor. Higher number of active transactions in DV results in better resource utilization and higher transaction throughput.

Figure 8 shows the effect of resource contention on the variance in transaction response times. At high resource contention, in 2PL, the transactions that are blocked due to data contention take longer time to complete compared to the transactions that

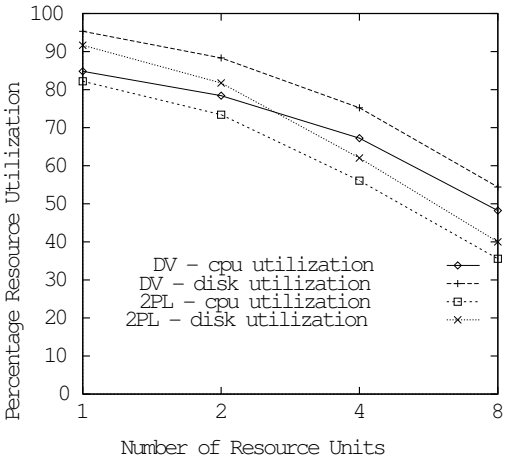

Figure 9: Resource utilizations

Figure 10: Trans. times for trans. mix

are not blocked. Hence, there is a large variance in the average transaction time. In DV, the number of transactions that are blocked due to data contention is quite low. All transactions wait equally for the resources, and hence they take almost equal times to complete. This results in lower variance, though the transaction throughput is the same for both the methods. Note that the reduction in variance at high data contention (100k-record database) is low (30% to 40%) at high resource contentions. This is due to the increased number of aborts at higher data contention. The aborted transactions take longer time because of the high resource contention, and thereby increase the variance. Figure 9 shows the CPU and disk utilizations for the two methods (for the 100k-record database) as a function of resource contention. As would be expected, DV shows higher resource utilization (15% to 20%) with reduced resource contention. Increasing the number of resource units to 16 or 32 shows further improvement in the performance of DV, although the resource utilizations drop to about 20%.

The blocking and restart rates slightly increase with the number of resource units, and show the same variation with data contention as presented in Section 3. The blocking rates for DV are less than one third the rates for 2PL. The number of restarts differs by less than 10%.

The number of dynamic versions increases as the size of the database is decreased. For the 100k-record database, on the average there are about 270 records with one dynamic version, less than 1 record with two dynamic versions and none with more than two dynamic versions. Considering the number of dynamic versions for all the records in a data page, there are, on the average, about 225 pages with one dynamic version, 18 pages with two dynamic versions and less than 1 page with three or more versions. Since there is space for five dynamic versions per page (20% free space per page) the multiple versions do not result in page splits. Also, as pointed in Section 2, only the incremental differences can be maintained in the versions, which further reduces the space overhead. Note that the overhead due to page splits is a one-time cost. A frequently updated page is initially split as many times as the space for dynamic versions is required. Once sufficient free space is available, there will not be any more overhead due to page splits.

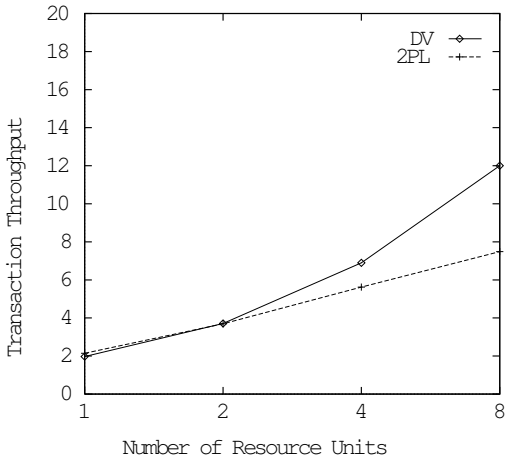

Figure 11: Throughput for trans. mix

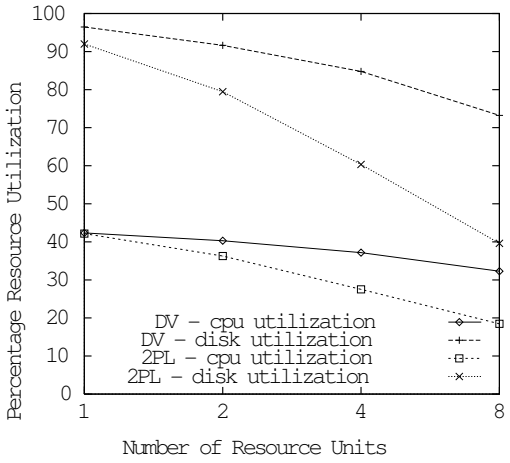

Figure 12: Resource utilizations

The next set of experiments considers a mix of transactions of different lengths. When short transactions are run together with relatively long transactions, the short transactions may starve while waiting for locks held by long transactions. In the experiments, 10% of the transactions are long, each with 200 record references. The other 90% transactions are short, with 10 record references each. Both types of transactions update 25% of the records accessed. The degree of multiprogramming is fixed at 100. The simulations are run on a 500k record database. The other simulation parameters are the same as in the previous case.

Figure 10 shows the effect of resource contention on the transaction response times of the two transaction types. At high resource contention, the long transactions take more time and the short transactions take less time in DV than in 2PL. In 2PL, most of the short transactions are blocked due to data contention. Since there are smaller number of transactions vying for resources, the waiting times are smaller and hence long transactions execute fast. When a short transaction is blocked by a long transaction, the short transaction has to wait until the long transaction completes. This adds significantly to the response time of the short transaction. Note that giving short transactions higher priority for resource access does not solve the problem, since the cause is data contention. On the other hand, in DV, there is very little blocking. Hence both short and long transactions compete for the resources, which increases the queue lengths and waiting times. This results in long transactions taking more time to complete. The short transactions execute faster since now they only have to wait for the resources. As the resource contention is reduced, the performance of DV becomes far better than that of 2PL. Figure 11 shows the throughputs for both the methods. The larger number of active transactions in DV utilizes the resources better. Figure 12 shows the resource utilizations for the two methods. At 8 resource units, the resource utilizations for DV are almost twice that of 2PL.

The average blocking rates for 2PL increase as the resource contention is reduced (from 54 blocked transactions at 1 resource unit to 66 blocked transactions at 8 resource units). At high resource contention, the transactions wait longer in the resource

queues thereby reducing the transactions competing for data. As resource contention is reduced, there are more transactions competing for data and hence higher blocking rates. In DV, the increase in blocking rates is marginal (from 17 blocked transactions at 1 resource unit to 19 blocked transactions at 8 resource units).

5 Supporting Queries

One of the important problems in concurrency control [Dewi 92] is to prevent the execution of queries (read-only transactions) from affecting concurrent update transactions (i.e., transactions which also update some data items). One solution is to violate serializability and run the queries at degree 2 or degree 1 isolation [Gray 76]. The problem becomes interesting if the queries wish to see a transaction consistent database. Here, we consider only the problem of running queries at degree 3 isolation, which ensures repeatable reads.

The single-version, two-phase locking method results in poor performance of update transactions in the presence of long-running queries. Queries hold read-locks to ensure serializability. Update transactions, which conflict with the queries, could be blocked for inordinate periods. The ensuing waiting causes unacceptable transaction response times, especially in an OLTP environment. The dynamic versioning method maintains multiple committed versions. Hence, queries can read the earlier versions without blocking any update transactions. We suggest three different techniques for executing queries efficiently in DV.

First Technique
Queries can be run the same way as update transactions. Queries acquire read-locks before reading any data item. The read-locks ensure degree 3 isolation. The only constraint is that if a query conflicts with an uncommitted update transaction for a data item, the query should be made a predecessor of the update transaction and allowed to read an earlier committed version of the data item. This design precludes a query ever being involved in a cycle, either with update transactions or other queries, which in turn leads to three important properties. Firstly, a request for a read-lock on a data item by a query is never blocked. Secondly, queries are never rolled back. Lastly, an update transaction is never aborted because of a query. These three properties help avoid any interaction between queries and update transactions. The disadvantage of this technique is that the queries incur the overhead of acquiring and releasing locks. However, this overhead is small (less than 3% in our experiments) compared to the total query response time (which includes disk access times and wait times for resources).

Second Technique
Queries can also be executed in DV without incurring the lock overheads. Note that in the first technique, read-locks are used only to create dependence relationships with conflicting update transactions. Alternatively, all uncommitted update transactions can be considered as sources of potential conflicts and a query, before it is started, can be made a predecessor to all such transactions. Hence, the second technique is to create

a node in the dependence graph for the query and add predecessor edges to it from all the nodes representing uncommitted update transactions. This ensures that the committed state of the database at the start of the query is maintained until the query completes. The query does not have to acquire or release any read-locks; the fixed position of the query in the dependence graph determines which versions the query accesses. Queries are never blocked in this technique either, nor do they cause update transactions to abort. This technique is specially suitable for long-running queries which access most of the database. The disadvantage of this technique is that the number of dynamic versions maintained here are much more than that created in the first technique. In the first technique, only those versions created by transactions that are successors of a query are maintained until the query completes. However, in the second technique, since all uncommitted update transactions are made successors of the query, all the dynamic versions created after the query is started should be maintained until the query completes.

Third Technique

The large space overhead in the second technique can be reduced in some cases, without acquiring locks. The reason for the large space overhead in the second technique is that a query is made a predecessor of *all* uncommitted update transactions. If it were possible to identify a subset of the update transactions that are likely to update some records in the part of the database accessed by the query, then only the transactions in that subset need to be made successors of the query. This avoids the necessity of maintaining dynamic versions created by the other update transactions in other parts of the database. For example, consider a database containing a million records. Suppose a long-running query accesses the first 100,000 records. Though the query accesses only 10% of the database, the second technique requires maintaining all the dynamic versions created even in the remaining 90% of the database until the query completes. This unnecessary overhead can be avoided if only the transactions, which are likely to update any of the first 100,000 records, are made successors of the query. An obvious way of implementing this technique would be to consider each table in the database as a distinct part, and order separately the queries and transactions working on each table. However, there are several applications where single tables can exceed 64 GB in size [Moha 93]. In such cases, each table can be further partitioned based on key ranges or general predicates[3]. Carrying the idea further, even predicate locks can be used to determine conflicts between queries and update transactions. The space overhead decreases as the size of the partitions is reduced. This technique has all the advantages of the second technique, but with much lower overheads.

The first technique is applicable for short queries or long running queries which access only a very small part of the database. The second technique is for long-running queries, accessing a significant portion of the database. The third technique is best suited for long-running queries accessing small and well-defined parts of the database.

Several methods have been proposed in the literature to support long-running queries. Due to space limitations, we compare DV with only two recent methods, which also use multi-versioning schemes. Mohan et al., [Moha 92] propose a method

[3]The partitions are not physical. They are used only to identify which parts of the database a query or a transaction accesses.

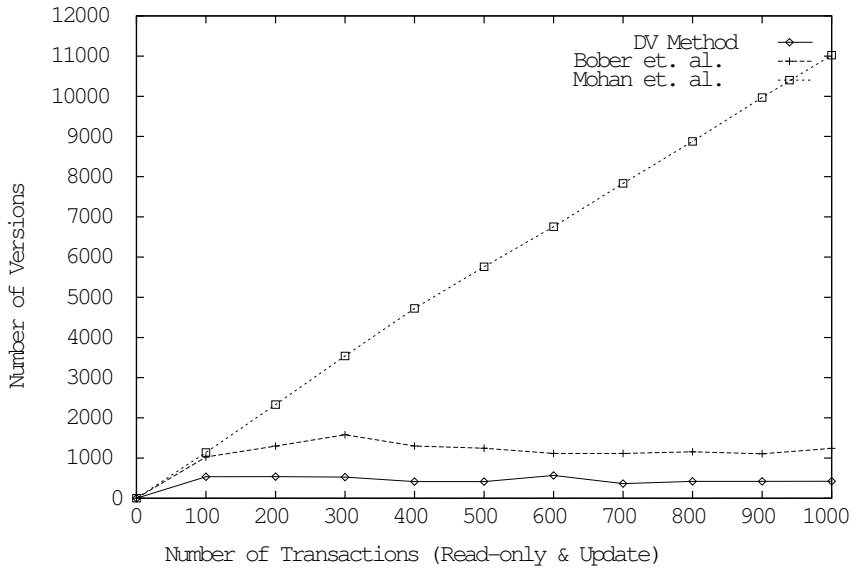

Figure 13: Overhead for versions

which uses version periods. Queries read the stable version of the database of the previous version period, while update transactions execute in the current version period. Queries are never blocked, do not acquire read-locks and do not affect update transactions. However, since queries read the data from the previous version period, they see a slightly old version. Each data item has one stable version in the previous version period and may have at most two additional versions in the current version period[4]. The system periodically switches to a new version period and purges the additional versions. Bober et al., [Bobe 92] use multi-version timestamp ordering to synchronize queries. Each update to a data item creates a new version. Since timestamp ordering is used, the versions created after a query is started have to be maintained (some versions may be deleted, if no query can access them) until the query completes. This method reserves a fixed space on each data page for caching versions. On pages containing data items which are only read, but rarely updated, the space reserved for cache is wasted. If an on-page cache overflows, the versions are moved to a shared version pool, which increases the cost of accessing the versions.

Both the above methods use single-version, strict two-phase locking semantics to synchronize update transactions. Unlike DV, the versions in these two methods are not used to resolve conflicts among update transactions. The two methods provide the same performance for queries as DV. The idea in the second DV technique is similar to multi-version timestamp ordering and hence maintains an equal number of versions as the method by Bober et al.

To compare the overhead due to additional versions in the DV method with that of the other two methods we use the following experimental setup. The environment consists of a mix of queries (25%) with a large number of references (100) and transactions (75%) with small number of references (20). Each transaction updates 75% of the

[4]We consider only the two version period algorithm here, since it has less space overhead than the multiple version period algorithm.

records accessed. The simulations are run on a 100k-record database with the degree of multiprogramming as 50. The other parameters are the same as in the simulations in the previous section. Figure 13 compares the number of additional versions maintained by the first DV technique with that of the methods of Bober et al., (same as second DV technique) and Mohan et al. The first DV technique has an average of 452 versions (with a maximum of 778 versions). The method of Bober et al., (second DV technique) has nearly 100% more overhead, with an average of 1152 versions and a maximum of 1647 versions. The graph for the method by Mohan et al., shows the number of versions created in a version period of different lengths. For example, if the version period is switched after every 500 transactions, this method maintains a maximum of 5760 versions. If the version period is switched after 1000 transactions, the number of versions almost doubles to 11022. These results indicate that the dynamic versioning method is flexible, applicable to a variety of work loads and performs at least as efficiently as any other method in supporting queries.

6 Conclusion

In this paper, we have developed and presented the design of *Dynamic Versioning*, an efficient multi-version, two-phase locking method. Our approach maintains the record-level versions on the data page and uses a dependence graph to reduce the space overhead for the additional versions to the minimum possible. A new lock compatibility matrix and conditional compatibilities were introduced for accessing the multiple versions. Transactions are dynamically ordered based on data conflicts so as to avoid blocking where possible. Using results of simulation studies, the dynamic versioning method was analyzed and compared with the single-version two-phase locking method. The following were the main conclusions assuming reasonable work loads.

- The DV method reduced significantly the number of transactions blocked due to data contention. For the work loads considered, the reduction varied from 60% to 90%.
- Variance in transaction response times was reduced by nearly 90%.
- At resource utilizations (60% to 80%) that are typical in actual databases, the higher concurrency in the DV method resulted in better resource utilization (15% to 20% more).
- Transaction throughput depends on the resource contention besides data contention. When the bottleneck resource was 100% utilized, the resource contention dominated data contention, and both methods had similar throughputs. At reasonable resource utilizations (60% to 80%), the higher concurrency in the DV method resulted in upto 40% higher transaction throughput for the environments considered.
- In a mix of short and long transactions, the DV method reduced significantly the starvation of short transactions by long transactions.

- The above advantages were realized with a minimum space overhead. In all the simulation experiments we performed, the space for the dynamic versions was about 1% or less of the database size.

The dynamic versions can also be used to execute queries without affecting update transactions. The DV method can efficiently support different types of queries in large databases. We show that the space overhead for the versions in the DV method is 50% to 90% less than the other methods proposed to support queries.

Our results are significant because they demonstrate that the same multi-versioning method can be used, with minimal space overhead, to reduce data contention among transactions, while efficiently supporting queries of different types. We also showed that the dynamic versioning method can be easily incorporated into existing database implementations. Currently we are studying how the dynamic versioning ideas can be extended to parallel databases.

References

[Agar 87a] Agrawal, R., Carey, M., Livny, M. *Concurrency Control Performance Modelling: Alternatives and Implications*, ACM Transactions on Database Systems, December 1987.

[Agar 87b] Agrawal, R., Carey, M., McVoy, L. *The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems*, IEEE Transactions on Software Engineering, December 1987.

[Baye 80] Bayer, R., Heller, H., Reiser, A. *Parallelism and Recovery in Database Systems*, ACM Transactions on Database Systems, June 1980.

[Bern 87] Bernstein, P.A., Hadzilacos, V., Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Pub. Co., 1987.

[Bobe 92] Bober, P., Carey, M. *On Mixing Queries and Transactions via Multiversion Locking*, Proc. 8th International Conference on Data Engineering, February 1992.

[Dewi 92] DeWitt, D., Gray, J. *Parallel Database Systems: The Future of High Performance Database Systems*, Communications of the ACM, June 1992.

[Gray 93] Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[Gray 76] Gray, J., Lorie, R., Putzolu, F., Traiger, I. *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, Modelling in Data Base Systems, North Holland Publishing, 1976.

[Guka 93] Gukal, S., Omiecinski, E., Ramachandran, U. *LU Logging: An Efficient Transaction Recovery Method*, Technical Report, Georgia Institute of Technology, GIT-CC 93/21, March 1993.

[Moha 93] Mohan, C. *A Survey of DBMS Research Issues in Supporting Very Large Tables*, Proc. 4th Int'l Conference on Foundations of Data Organization & Algorithms, October 1993.

[Moha 92] Mohan, C., Pirahesh, H., Lorie, R. *Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions*, ACM SIGMOD, June 1992.

[Pein 83] Peinl, P., Reuter, A. *Empirical Comparison of Database Concurrency Control Schemes*, Proc. 9th International Conference on Very Large Data Bases, 1983.

[Pira 90] Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P. *Parallelism in Relational Database Systems: Architectural Issues and Design Approaches*, IEEE 2nd Int'l Symp. on Databases in Parallel and Distributed Systems, July 1990.

[Schw 90] Schwetman, H. CSIM Users Guide, March 1990.

[Stea 81] Stearns, R.E., Rosenkrantz, D.J. *Distributed Database Concurrency Controls Using Before-Values*, ACM SIGMOD, April 1981.

A Implementing Dynamic Versioning

In this appendix, we describe how the dynamic versioning method can be incorporated into an existing database implementation that uses two-phase locking and logging. The lock manager is designed to allow multiple versions as given by the lock compatibility matrix. The storage manager is modified to accommodate the dynamic versions in the database and perform garbage collection. Other database system modules require little modification.

A.1 Lock Manager Design

The lock manager module contains the logic to allow accesses by different transactions to different items. Given the requirements in Section 2, the lock manager should provide the following calls for the transactions to acquire and release the data items in different lock modes.

Get-Read-Lock to get an r-lock.

Get-Update-Lock to get an u-lock, if there is a possibility that the transaction might modify the item.

Upgrade-to-Analyze-Lock to upgrade the u-lock to w-lock, if the transaction decides to modify the item.

Downgrade-to-Read-Lock to downgrade the u-lock to r-lock, if the transaction does not want to modify the item.

Release-Read-Lock to release the r-lock.

Release-Write-Lock to change the w-lock to c-lock.

The lock manager maintains the commit-locks of a committed transaction as long as the transaction has at least one predecessor. The commit lock is released when the transaction has no predecessors.

We provide a way of implementing the required calls by extending the lock manager design given in [Gray 93]. The traditional lock structure conceptually contains two queues of lock requests for a data item, granted requests and waiting requests. Here, for a data item, we may have a number of committed versions and one uncommitted version. We may also have transactions reading any of the committed versions. Hence the following queues of lock requests are needed.

Base-readers transactions that read the original committed value in the database.

Writer-readers-queue queue of transaction sets, with each set containing one writer and the transactions that read the version created by that writer (if committed).

Readers-wait-queue contains transactions requesting an r-lock; the transactions are successors to the current writer, which is in either u-lock or w-lock mode.

Writers-wait-queue contains transactions requesting an u-lock; a current uncommitted writer exists.

Figure 14 shows the data structures maintained by lock manager for a data item, being held in different lock modes by different transactions. The outline of the lock manager routines is given in the appendix B.

A.2 Storage Manager Design

To implement the dynamic versioning method an efficient storage scheme for accessing the multiple versions of the data items is required. The usual practice in logging is to record the old value of the data item in a log record and let the writer modify the item in place. The log record containing the old value may have been forced to disk. This practice will not work well for DV which requires access to the (committed) old values. Hence it is more efficient to keep the multiple values together in the database.

The page directory slot structure, which contains pointers to the items stored on that page, now also supports the dynamic versions. The directory slot for the item contains an extra pointer to the slot for the dynamic version. If this pointer is null, the item has only one value. The dynamic version also contains the identifier of the

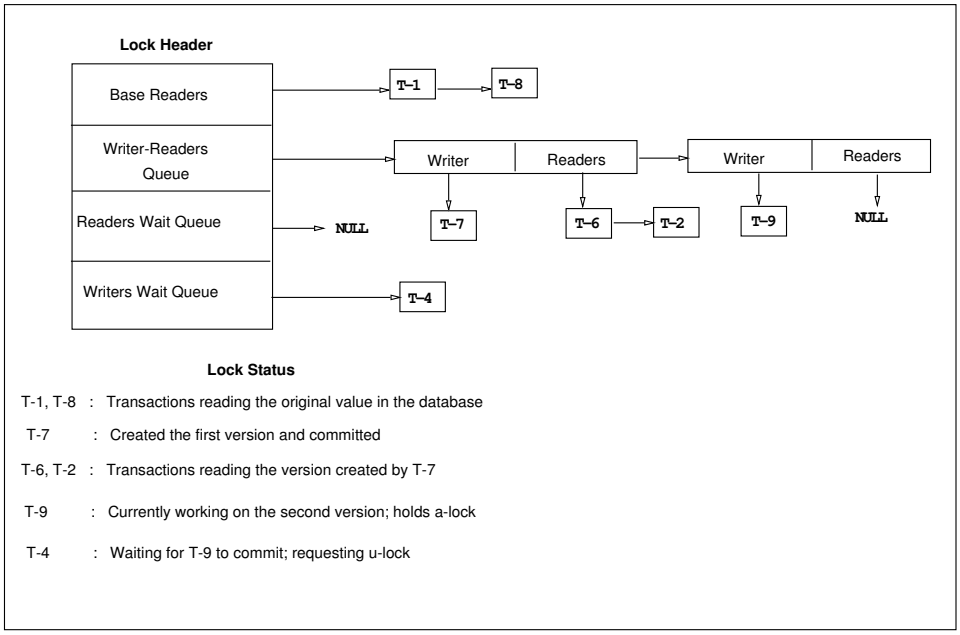

Figure 14: Lock structure

transaction that created it. All the versions of an item are linked in the order of creation through the directory slots.

A new committed version of a data item can replace an old version if no uncommitted transaction may access the old version. If the old version is replaced as soon as the transaction, that created the new version, commits (immediate updating - as in shadowing), we incur additional overhead of reading the data page containing the versions, which may have migrated to the disk. Immediate updating is not necessary here, since both the old and the new versions are on the same page. If the data page is in the main memory, the old version is replaced immediately. If the data page has migrated to the disk, the old version is replaced when some other transaction requires to modify the data page and reads the page to the main memory. We term this as lazy updating [Guka 93]. Hence, during transaction processing, the overhead for committed transactions for the dynamic versioning method is the same as that of single-version two phase locking implementations [Gray 93].

Transaction aborts and partial rollbacks are implemented the same way as that for systems with no versioning. To undo an update, the corresponding page is fetched to main memory, the space occupied by the dynamic version is released and the directory slot pointer to the dynamic version is made null. For a partial rollback, in case the transaction had updated the same item even before the savepoint, the correct value for the version is retrieved from the log records. The same scheme and data organization, with a few modifications, can be used to reduce the crash recovery times by almost an order of magnitude compared to the current log-based recovery methods. [Guka 93] contains the details of the recovery algorithm.

A.3 Secondary Indices

Access to secondary indices can be another source of blocking for transactions. Managing secondary indices for multiple versions under the dynamic versioning method requires some explanation. If an update by a transaction does not modify the key, there is no need to change the index. If an update modifies the key, a new entry in the index with the modified key is made without deleting the old entry. The old entry is marked with the identifier of the transaction that modified the key. When another transaction accesses the old or the new index entry, the action is based on the relative position in the dependence graph of that transaction with respect to the transaction that modified the key. The old index entry is deleted when the transaction that modified it is committed and has no predecessors. If the transaction aborts (or rolls back), the new index entry and the marker on the old index entry are removed.

Suppose transaction (T1) modifies the key of a data item and then transaction (T2) tries to access the same item. The following cases occur.

1. T1 is committed and has no predecessors.

   case A: T2 tries to access the record through the old index entry.
   – The old index entry is deleted and T2 ignores it.

   case B: T2 tries to access the record through the new index entry.
   – Lock manager (using the dependence graph) decides whether T2 should be allowed access, blocked or aborted.

2. T2 tries to r-lock the record through the old index entry.

   case A: T2 is a predecessor of T1.
   – T2 gets the r-lock.

   case B: T2 is a successor of T1.
   – T2 blocks until T1 is committed.

   case C: No relationship exists between T1 and T2.
   – T2 gets the r-lock and becomes a predecessor of T1

3. T2 tries to u-lock the record through the old index entry.

   case A: T2 is a predecessor of T1.
   – T2 aborts.

   case B: T2 is a successor of T1.
   – T2 blocks until T1 is committed.

   case C: No relationship exists between T1 and T2.
   – T2 becomes a successor of T1 and blocks until T1 is committed.

4. T2 tries to r-lock or u-lock the record through the new index entry.

   case A: T2 is a predecessor of T1.
   – T2 ignores the new index entry.

case B: T2 is a successor of T1.
- T2 blocks until T1 is committed.

case C: No relationship exists between T1 and T2.
- T2 becomes a predecessor of T1 and ignores the new index entry.

The index manager need only check cases (1A), (4A) and (4C) above. The lock manager routines resolve the other cases. The dynamic versioning method tries to avoid blocking if possible. Transactions are blocked only if immediate access leads to deadlocks. As pointed out in [Moha 92], keeping the old index entries also has the benefit of avoiding next key locking during key delete operations, which handles the phantom problem.

B Lock Manager Routines

```
1. Get-Read-Lock ( lockname, requester )
        access the lock structure for lockname
        if no writer exists
                add requester to base-readers
        else    { writers exist }
                if one of the committed writers is a successor
                        add requester to the readers in the writer-readers-set
                                                    before the first successor
                else if last writer is committed
                        add requester to the readers in the last
                                                    writer-readers-set
                else if last writer is not a predecessor
                        make the last writer a successor to requester
                        add requester to the readers in the writer-readers-set
                                                    before the last writer
                else
                        add requester to the readers-wait-queue
                        sleep
                endif
        endif

2. Get-Update-Lock ( lockname, requester )
        access the lock structure for lockname
        if the uncommitted writer or one of the committed writers is a successor
                return RESTART
        if no uncommitted writer
                make requester as the last writer
        else
                append requester to writers-wait-queue
```

```
                sleep
        endif

3. Upgrade-to-Analyze-Lock ( lockname, requester )
        change status of requester to analyze

4. Downgrade-to-Read-Lock ( lockname, requester )
        access the lock structure for lockname
        shift the requester to the readers in the previous writer-readers-set
        Call Enable-Waiting-Transactions ( lockname )

5. Release-Read-Lock ( lockname, requester )
        access the lock structure for lockname
        get the readers list which contains the requester
        delete requester from the readers list

6. Release-Write-Lock ( lockname )
        access the lock structure for lockname
        if no predecessors
                Call Delete-Write-Lock ( lockname )
        Call Enable-Waiting-Transactions ( lockname )

7. Delete-Write-Lock ( lockname )
        access the lock structure for lockname
        shift all readers in the first writer-readers set to base-readers
        delete the writer in the first writer-readers set
        delete the first writer-readers set

8. Enable-Waiting-Transactions ( lockname )
        access the lock structure for lockname
        enable the first writer in the writers-wait-queue
        for each reader in the readers-wait-queue
                if the reader is not a successor of the enabled writer
                        move the reader to the readers before the enabled writer
                        wakeup the reader
                endif
        endfor
```