Performance Evaluation of Multiversion Database Systems

Sang H. Son and Navid Haghighi
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

This paper describes a detailed simulation study of the performance of multiversion database systems, investigating the characteristics and the extent to which they provide performance benefits over their single-version counterparts. First, the paper presents the structure of a software prototyping environment for the evaluation of distributed database systems. Using this environment, it is shown that for specific workload, multiversion database systems offer performance improvements despite of additional CPU and I/O costs involved in accessing old versions of data. It is also shown that transaction size is one of the most critical parameters that affect the system performance.

1. Introduction

Recently, a number of multiversion concurrency control algorithms to increase the level of concurrency have appeared in the literature [2, 3, 5, 6, 7]. The basic idea in those algorithms is to maintain one or more old versions of data objects in order to allow transactions to proceed using both the current version and older versions. The goal is to permit read-only transactions to read old versions of data while allowing update transactions to create newer versions concurrently.

In addition to the development of new algorithms, multiversion concurrency control has been the subject of theoretical studies [7, 8]. Serializability theory has been extended to include multiversion concurrency control algorithms, and it has shown that multiversion algorithms are able to provide serializability and more concurrency than single version algorithms. However, the performance of multiversion database systems in a realistic environment has not been received very much attention. The focus of this paper is to investigate the characteristics and possible performance improvement of a multiversion database system with a timestamp-based concurrency control algorithm.

Few previous studies have addressed the performance of multiversion database systems [2, 3, 5, 9]. Although these studies provide a valuable insight in regards to the particular approach adopted in undertaking the evaluations, it is almost impossible to compare or integrate their results. This is due to the fact that they each have made different assumptions about the environment and the system, and they have often used widely varying performance metrics. As a consequence, performance results in some of the studies are inconclusive and sometimes even contradictory [10]. We feel that an important reason for this situation is that many interrelated factors affecting performance (concurrency control, buffering schemes, data distribution, etc.) have been studied as a whole, without completely understanding the overhead imposed by each.

A prototyping technique can be applied effectively to the evaluation of concurrency control algorithms for distributed database systems. A database prototyping environment is a software package that supports the investigation of the properties of a database control techniques in an environment other than that of the target database system. The advantages of an environment that provides prototyping capability are obvious. First, it is cost effective. If experiments for a twenty-node distributed database system can be executed in a software environment, it is not necessary to purchase a twenty-node distributed system, reducing the cost of evaluating design alternatives. Second, design alternatives can be evaluated in a uniform environment with the same system parameters, making a fair comparison. Finally, as technology changes, the environment need only be updated to provide researchers with the ability to perform new experiments.

A prototyping environment can reduce the time of evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation.

This paper describes a detailed simulation study of the performance of multiversion database systems, investigating the characteristics and the extent to which they provide performance benefits over their single-version counterparts. In our study, by using the prototyping environment developed at the University of Virginia [1], the assumptions which significantly influence the system behavior have been specified as parameters during the evaluation process. Therefore we not only have the capability of investigating the influence of these assumptions on system performance, we can also study the other major performance factors while retaining the assumptions at a constant.

2. Structure of the Prototyping Environment

For a prototyping tool for distributed database systems to be effective, appropriate operating system support is mandatory. Database control mechanisms need to be integrated with the operating system, because the correct functioning of control algorithms depends on the services of the underlying operating system; therefore, an integrated design reduces the significant
Although an integrated approach is desirable, the system needs to support flexibility which may not be possible in an integrated approach. In this regard, the concept of developing a library of modules with different performance and reliability characteristics for an operating system as well as database control functions seems promising. Our prototyping environment follows this approach [1, 11]. It is designed as a modular, message-passing system to support easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It efficiently supports a spectrum of distributed database functions at the operating system level, and facilitates the construction of multiple "views" with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup.

The prototyping environment provides support for transaction processing, including transparency to concurrent access, data distribution, and atomicity. An instance of the prototyping environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms and facilities into the prototyping environment, or testing alternate implementations of algorithms. To permit concurrent transactions on a single site, there is a separate process for each transaction that coordinates with other server processes.

The prototyping environment is based on a concurrent programming kernel, called the StarLite kernel, written in Modula-2. The StarLite kernel supports process control to create, ready, block, and terminate processes. Based on the StarLite kernel, the environment consists of the modules for user interface, configuration management and transaction generation, transaction manager, message server, resource manager, and performance monitor.

User Interface (UI) is a front-end invoked when the prototyping environment begins. UI is menu-driven, and designed to be flexible in allowing users to experiment with various configurations with different system parameters. A user can specify the following:

- system configuration: number of sites and the number of server processes at each site.
- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.
- load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only or update) and their priorities, and the mean interarrival time of transactions.
- concurrency control: locking, timestamp ordering, and priority-based.

UI initiates the Configuration Manager (CM) which initializes necessary data structures for transaction processing based on user specification. CM invokes the Transaction Generator at an appropriate time interval to generate the next transaction to form a Poisson process of transaction arrival.

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. Each transaction is assigned to the Transaction Manager (TM). TM issues service requests on behalf of the transaction and reacts appropriately to the request replies. TM executes the two-phase commit protocol to ensure that a transaction commits or aborts globally.

The Message Server (MS) is a process listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site and the sender blocks itself on a private semaphore until the message is retrieved by MS. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When MS retrieves a message, it wakens the sender process and forwards the message to the proper servers or TM. The prototyping environment implements Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and receive messages directly through their associated ports.

The Performance Monitor interacts with the transaction managers to record, priority/descriptor and read/write data set for each transaction, time when each event occurred, statistics for each transaction and cpu hold interval in each node. The statistics for a transaction includes arrival time, start time, total processing time, blocked interval, whether deadline was missed or not, and number of aborts.

3. Multiversion Control

In a multiversion database system, each write operation creates a new version of data objects instead of overwriting it. Hence, for each read operation, the system selects an appropriate version to read from a collection of available versions.

The multiversion database system we have implemented is based on timestamp ordering. Each data object is represented as a list of versions connected by a doubly linked pointer, and each version is associated with timestamps for its creation (write timestamp) and the latest read (read timestamp), and a validity bit to specify whether the version is certified (i.e., the transaction that created it is committed).

A read operation does not necessarily read the latest committed version of a data object. A read request is transformed to a version-read operation by selecting an appropriate version to read. When a read request with timestamp T is sent to the Resource Manager, the version of a data object with the largest timestamp less than T is selected as the value to be returned.

The timestamp of a write request is compared with the read timestamp of the highest version of the data object. The highest read timestamp of the data object must be less than the timestamp of the write request. A new version with the write timestamp greater than the read timestamp of the highest certified version is built on the upper level, with a "0" tag bit to indicate that the new version is not valid yet. If the timestamp of a transaction is less than the read timestamps of all the existing versions, the transaction will be aborted.

In order to simplify the concurrency control mechanism, we allow only one temporary version for each data object. Inserting a new version in the middle of existing valid versions
is not allowed. In Section 4 we discuss the consequences of removing these constraints.

3.1. Version Retention

Read requests are never rejected in a multiversion database system if all the versions are retained. However, all versions cannot be retained forever in any multiversion database systems due to the storage and processing overhead. We now discuss how versions can be discarded when they are not needed by read-only transactions. Recall that each data object keeps track of the read requirements that have accessed the data object. By using timestamp assignment method for read-only transactions different from that for update transactions, version management can be simplified.

When a read-only transaction begins, the coordinator sends messages to the participants telling them the data objects the transaction needs to read. When a participant receives such a request, it checks the latest write timestamp of each data object at the site read requested by the transaction, and sends the maximum timestamps among them to the coordinator. Each data object accessed by a read-only transaction in this way records the pair of the identifier of that transaction and the current timestamp it reported. After receiving responses from all participants, the coordinator chooses a unique timestamp greater than all the responses. The timestamp recorded for the read-only transaction at each object is thus a lower bound on the timestamp of the transaction, and it will be used in making a decision to discard or retain versions of the data object.

When a read-only transaction with timestamp TS invokes a read operation on a data object, the participant chooses the version of the data object with the largest write timestamp less than TS. This invocation of read operation is nothing but sending the timestamp TS to the participants, so each participant already knows which data object to read. If TS is larger than the read timestamp of the data object, it will be updated as TS. This will force update transactions that commit later to choose timestamps larger than TS, ensuring that the version selected for the read-only transaction does not change. Resource Manager can use the following rule to decide which versions to keep and which to discard.

Rule for retention:

A version with the write timestamp TS must be retained if

1. there is no version with timestamp greater than TS (i.e., most up-to-date version), or
2. there is a version with timestamp TS’ > TS, and there is an active read-only transaction whose timestamp might be between TS and TS’.

By having a read-only transaction inform data objects when it completes, versions of data objects that are no longer needed can be discarded. This process of informing data objects that a read-only transaction has completed need not be performed synchronously with the commit of the transaction. It imposes some overhead on the system, but the overhead can be reduced by piggybacking information on existing messages, or by sending messages when the system load is low.

When a read-only transaction sends a read request to an object, Resource Manager effectively agrees to retain the selected version and any later versions, until it knows which of those versions is needed by the read-only transaction. When Resource Manager finds out the timestamp chosen by the transaction, it can tell exactly which version the transaction needs to read. At that point any versions that were retained only because the read-only transaction might have needed them can be discarded. By minimizing the time during which only a lower bound on the transaction’s timestamp is known, the system can reduce the storage needed for maintaining versions. One simple way of doing this is to have each read-only transaction broadcast its timestamp to all Resource Manager when it chooses the timestamp.

The version management described above is effective at minimizing the amount of storage needed for multiple versions. For example, unlike the "version pool" scheme in [6], it is not necessary to discard a version that is needed by an active read-only transaction because the buffer space is being used by a version that no transaction wants to read. However, ensuring that each Resource Manager knows which versions are needed at any point in time has an associated cost; a read-only transaction cannot begin execution until it has chosen a timestamp, a process that requires communicating with all data objects it needs to access.

3.2. Performance Evaluation

Various statistics have been collected during the experiments for comparing the performance of a multiversion database system and its corresponding single-version system, both implemented using the prototyping environment. The average response time and the number of aborts for a group of transactions running on both systems are the most important performance measures investigated in the experiments.

Transaction are generated with exponentially distributed interarrival times, and the data objects updated by a transaction are chosen uniformly from the database. Two groups of transactions with different characteristics (e.g., type and number of data objects) are executed concurrently. A transaction has an execution profile which alternates data access requests with equal computation requests and an execution requirement for termination (either commit or abort). Thus the total processing time of a transaction is directly related to the number of data objects accessed. Due to space considerations, we cannot present all our results but have selected the graphs which best illustrate the difference and performance of two database systems. For example, we have omitted the results of an experiment that varied the size of the database, and thus the number of conflicts, because they only confirm and not increase the knowledge yielded by other experiments.

In each of the four experiments to be presented in this section the performance of the multiversion system was compared to that of a corresponding single-version database system. For each experiment, we collected performance statistics and averaged over the 10 runs. We chose the arrival rate so that database systems can be tested in a heavily loaded rather than lightly loaded situation. Even though they may not arise frequently, one would like to have a system that performs reasonably well when such peaks occur. In other words, when a crisis occurs and the database system is under pressure is precisely when a multiversion database should show its advantage over single-version database systems.
3.2.1. Experiment 1: Parameters in Favor of Multiversion

The purpose of this experiment is to examine the behavior of the multiversion database system under a workload for which multiple versions would be beneficial. Intuitively, multiple versions of data objects should help the scheduler to avoid aborts of read-only transactions that arrive too late. Since multiple versions are maintained, only read-only transactions can access earlier versions of the data.

The transaction mix emphasizes conflicts involving read-only transactions. In each case, 80% of the transactions are update transactions and the remaining 20% are read-only transactions. Update transactions are very small; they access 2 data objects. On the other hand, read-only transactions are large, ranging in size from 8 to 40. Conflicts between update transactions are possible but not very likely. Conflicts between read-only and update transactions occur quite frequently. The multi-version system should avoid these conflicts by allowing read-only transactions to read earlier versions of the data objects [3].

The average response times and the number of aborts of both systems are shown in the Figures 1 and 2. Clearly the multiversion system provides a significant improvement in terms of the number of aborts. Every time a transaction is aborted, we effectively lose the work that transaction has already accomplished, and in addition, that transaction must be restarted later. Since the read-only transactions are large, this could amount to considerable wasted processing time. As discussed earlier, multiversion system is expected to reduce the average response time because of the presence of read-only transactions. Therefore, it would be of interest to examine the effect of the multiversion system on the performance of the read-only transactions. In order to accomplish this, we isolated the read-only transactions from the update transactions. This way, the behavior of the read-only transactions can be represented in the graph correctly. The graphs in Figures 3 and 4 show the average response times and the number of aborts of the read-only transactions in isolation.

Timestamp ordering using single versions of data objects is biased against large read-only transactions [3]. The algorithm aborts a transaction every time it tries to read a data object whose write timestamp is greater than the transaction’s timestamp. As the read-only transactions get larger, the probability that the transactions get aborted increases. In this experiment, these sorts of conflicts are very likely. In the single-version system the large read-only transactions often get starved out (restarted over and over) as the small update transactions proceed with their computations. These aborts result in high average response times. The multi-version system avoids this problem by satisfying read requests with older versions. Therefore the average response times are lower. The multiversion concurrency control algorithm results in considerable performance improvement when the transaction mix consists of large read-only transactions and small update transactions. In the next experiment, we discuss the effects of transaction mix on system performance in detail.

3.2.2. Experiment 2: Effects of Transaction Mix

The purpose of this experiment is to investigate the effect of the transaction mix (update/read-only ratio) on the performance of the single and multiversion systems. In experiment 1, we evaluated the performance of two systems under a transaction load mix of 80% update and 20% read-only transactions. This experiment analyzes the following update/read-only ratios:

- 20% update, 80% read-only
- 50% update, 50% read-only
- 100% update

Throughout this experiment, the transaction size is held constant at 5. This means that all transactions read or write 5 data objects.

20% Update, 80% Read-Only Transactions

Figure 5 illustrates the results for this transaction mix. Since read-only transactions dominate the transaction mix, conflict probability is low. This probability is further reduced due to the small number of data objects that are accessed by transactions. Since read-only transactions rarely get aborted in the single-version system, there is little hope that the existence of multiple versions would help in preventing aborts. The graph shows this, and we may conclude that when transactions are uniformly small and read-only transactions dominate, multiple versions do not provide a performance improvement.

50% Update, 50% Read-Only Transactions

Again, due to the small size of transactions, there is not a significant number of read-only transaction aborts that the multiple versions can help to avoid. Therefore, the performances of the single and multiversion systems are almost identical.

100% Update Transactions

The graph is shown in Figure 6. Since there are no read-only transactions, and therefore no aborts involving them, we would not expect the multiversion system to outperform its single version counterpart. In fact, as shown in the graph, the single version system performs better. This difference in the performances becomes more noticeable as we increase the number of transactions. The difference is due to the fact that the implementation of the multiversion system is prejudiced against update transactions. This is partly caused by the rules for version creation. As a result, more transactions are aborted due to write-write conflicts. The consequences of changing these rules are discussed in the next section.

From the results of these experiments, we conclude that regardless of the transaction mix, multiple versions do not provide much performance gain (if anything at all) when the transaction sizes are uniformly small.

3.2.3. Experiment 3: Effects of Transaction Size

This experiment investigates the behavior of the single and multiversion systems under varying transaction sizes. The purpose is to examine whether transaction size has a noticeable affect on the performance of the single and multiversion systems. We have investigated the performance of both systems while varying transaction sizes under the following three different transaction mixes:

- 80% update, 20% read-only
- 20% update, 80% read-only
- 50% update, 50% read-only
In all cases, 40 transactions were randomly generated. The graphs displaying the relative performances appear in Figures 7, 8, and 9.

As shown in the graphs, multiple versions do not provide a performance improvement in any of the three cases. In some cases the single-version system performed better. The multiple versions do not enhance the performance of the system when the sizes of the read-only and update transactions do not differ widely. If the read-only transactions are small or generally the sizes of the read-only and update transactions are close or equal, both systems yield almost the same performances.

3.2.4. Experiment 4: Effects of Data Distribution

The purpose of this experiment is to examine the performance of the single and multi-version systems in distributed environments. In our model of a distributed environment, each site is equipped with its own processor. In addition, each site has its own stable storage which contains a portion of the database. Data objects are not replicated; each data object is stored at only one site. Transactions make data access requests that are either local or remote. Local requests are satisfied by accessing data objects stored at the site itself. Remote requests are handled by sending a request message to the data manager of the site that stores the data object.

In a distributed environment, one of the key factors that affects performance is the communication delay. This delay reflects in terms of response time for the communication that must take place between two sites. In our model the system is fully connected and the inter-site communication delay is five times longer than the delay involved in inter-process communication within the same site.

The experiments investigate the behavior of the single and multiversion systems in distributed environments having 2 or 5 sites. In each case transactions access 5 data objects. The transaction mix consisted of 80% update and 20% read-only transactions. Figures 10 and 11 show the average response times of the systems in a distributed environment with 2 and 5 sites respectively.

Our first observation is that the average response time in the distributed systems is generally much lower than that in the centralized cases observed in experiments of previous sections. This is due to the fact that in the distributed cases we have increased our computational power by providing additional processors. The communication delay introduced as a result of distributing the data objects across network is offset by the increased computational power. With only one site, as shown in the slope of the curves of previous experiments, there is a direct relationship between the number of transactions and the average response time. The slope of Figure 10 is not as sharp, indicating that the effect of increasing the number of transactions on the average response time is more subtle. This is more clear in Figure 11 in which the number of transactions does not seem to affect the average response time.

In our experiments data objects accessed by transactions are uniformly distributed across the entire database. This strategy causes the workload to be divided and shared equally by processors at different sites. Therefore, a significant increase in the workload was not experienced by each processor as we increased the number of transactions. If however, the access pattern is biased toward a certain subset of the database, increasing the number of transactions does not uniformly increase the workload of all processors. In such cases, a small number of processors which are located at the sites where the "hot" areas of the database reside perform the majority of the work. Therefore, a large subset of the processors wait idlely, while a selected few process the remote requests of idle transactions. Consequently, increasing the number of transactions would increase the workload of only a small set of the processors, and therefore we can expect to see a more direct relationship between the number of transactions and the average response times.

3.3. Analysis of Results

The results of the experiments indicate that multiple versions only provide a significant performance enhancement under a mix of transactions for which they were intuitively thought of as being beneficial. In general, a database system adapting a multiversion concurrency control algorithm performs better while processing read requests. Read requests that would be aborted in a single-version database system due to conflicts could be successfully processed in a multiversion system using older versions of data. Therefore, when the read requests dominate the transaction load, and there is a high probability for read-only transactions to be aborted due to conflicts with update transactions, a multiversion system performs better than its corresponding single-version system. The relative sizes of the read and write sets of transactions is an important factor affecting the performance.

If multiple versions are not used in preventing aborts of read-only transactions, they should not degrade the performance of the system either. However in practice, as observed in some of the graphs of the experiments, the multiversion system performed even more poorly than its single-version counterpart. While examining the operation of the multiversion system we found that the rules for version creation works against update transactions. It turns out that allowing only one temporary version for each data object and not allowing a new version in the middle of existing valid versions is too costly. It causes aborts due to write-write conflicts which do not occur in the single-version system. In the next section, we discuss the impact of changing those rules for version creation on the performance of the multiversion system.

4. Rules for Version Creation

In this section we will compare the performance of a new implementation of the multiversion system with that of the system studied in the previous section. The reason for implementing the new multiversion system was to remove the constraints in the creation and maintenance of new versions of data objects which were found to be expensive in terms of aborts of update transactions. In the new implementation, new versions of data objects are allowed to be inserted in between two existing valid versions and an unlimited number of temporary versions are allowed.

We examine the behavior of the new implementation under a transaction mix of 80% update and 20% read-only transactions. The graph of Figure 12 displays the relative performances of the new multiversion system and its single version counterpart. This graph should be compared with Figure 7. As shown in Figure 7, the original implementation of the multiversion system performed marginally worse than the single version.
system. This is due to the fact that even though the original implementation avoided aborts of read-only transactions, there was a high probability of aborts of update transactions in that transaction mix. Hence the performance improvement by avoiding aborts of read-only transactions was offset by aborts of update transactions. In the new implementation, since the probability of abort of update transactions is reduced, the performance of the multiversion system gets better than that of the single version system as the size of the transaction gets larger. Similar results are observed when the transaction load consists of 50% update and 50% read-only transactions.

The results presented in this section indicate that in implementing multiversion concurrency control algorithms, careful consideration must be given to the constraints imposed by the manner data versions are created and maintained. We have found that performance anomalies can be avoided by changing the rules for version creation.

5. Discussion

In this section we will compare our approach to prototyping and performance evaluation with other work in the field and to discuss how our results are related to those of other researchers. The nature of most previous studies and their results were highly affected by the implementation assumptions. For example, in [9] the results are highly influenced by the fact that only two versions of each data item were allowed. In order to examine the approaches to performance evaluation studies, we need to explore the model assumptions that were involved. In addition, only few studies have examined the response time of the multiversion concurrency control algorithms.

Model assumptions characterize the environment in which the performance evaluation takes place. One of the problems associated with the majority of previous studies is that in most cases the performance evaluation process is completely dependent on the underlying hardware assumptions. In these cases, it is extremely difficult to examine the behavior of concurrency control algorithms independently of the influence of the underlying architecture. For instance, in [4] the authors present a performance analysis for two concurrency control algorithms in a distributed environment with two sites. Each site has its own processor (VAX 11/780) and main memory. The database itself resides on two different file servers with different performance characteristics. It is extremely difficult, if not impossible, to consider the performance of the concurrency control algorithms in isolation from the effects of the hardware factors. The test environment dependencies make it hard for the results given in one study to be applied to other environments.

Our approach minimizes the limitations imposed upon the performance evaluation process by rigid hardware configurations. For example, one can easily examine the effect of decentralizing the database by increasing the number of sites while keeping the other parameters constant. Different concurrency control algorithms can be selected to process the same set of transactions in identical environments. The CPU and I/O rates, as well as communication costs, are not dependent on the underlying architecture, but are rather simulated by the model and can be altered by the researcher. This flexibility allows the performance evaluation process to be liberated from the limitations imposed by hardware configurations of the evaluating environment.

The nature of creation and maintenance of new versions of data in a multiversion system has not been considered carefully. The model in [3] allows only one temporary version while creating a new version. This is an implementation assumption which, as we have discussed earlier, has a significant effect on performance results.

The results of our experiments confirm those observed in [2]. The multiversion system performs only marginally better in some specific cases. As reported in [3], our multiversion system performed better under a mix of transactions for which the multiple versions were thought to be beneficial. Under a workload of large read transactions and a large number of update transactions, the multiversion system outperforms its corresponding single-version system. Our results confirm those observed in [2]: multiple versions do not provide a significant performance improvement when the read-only transactions are small.

The performance evaluation results in [3] indicate that multiversion algorithms offer a significant performance improvement in most cases. This seems in conflict with our observations. However, the level of multiprogramming (MPL) in their study is 10 in most cases, while it becomes very large in our experiments. In fact, one of their results shows that for a high MPL, performance improvement of the multiversion system is reduced. When the transactions are small, the probability of aborting the read-only transactions is small anyway, so multiple versions are not of much help in preventing aborts. Unless the transaction mix is such that there is a high probability for abort of read-only transactions due to conflicts with update transactions, the multiple versions will not provide a significant performance improvement.

6. Conclusions

Prototyping large software systems is not a new approach. However, methodologies for developing a prototyping environment for distributed database systems have not been investigated in depth in spite of its potential benefits. In this paper, we have presented a prototyping environment that has been developed based on the StarLite concurrent programming kernel and message-based approach with modular building blocks. Although the complexity of a distributed database system makes prototyping difficult, the implementation has proven satisfactory for experimentation of design choices, different database control techniques, and even an integrated evaluation of database systems. It supports a very flexible user interface to allow a wide range of system configurations and workload characteristics.

Using the prototyping environment, we have shown that in general, a database system with a multiversion concurrency control algorithm performs better while processing read requests. Read requests that would be aborted in a single-version database system due to conflicts may be successfully processed in a multiversion system using older versions. Therefore, when the read requests dominate the transaction load, and there is a high probability for abort of read-only transactions due to conflicts, a multiversion system outperforms its corresponding single-version system. The relative sizes of the read and write sets of transactions is an important factor affecting the performance. Although the actual performance figures will vary depending on workload and implementation details, we believe that our results provide a good picture of the costs.
and benefits associated with the multiversion approach to concurrency control.

We also have shown that careful consideration must be given to the implementation constraints regarding the creation and maintenance of the multiple data versions. These constraints can degrade the system performance to the point that they offset any performance improvements that the multiple versions may provide. By changing the rules for version creation and maintenance, it is possible to implement a multiversion database system which never performs more poorly than its single version counterpart and provides a performance improvement in other cases.

References


