Obtaining Progressive Protocols for a Simple Multiversion Database Model[1]

Gael N. Buckley
A. Silberschatz

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

Most database systems ensure the consistency of the data by means of a concurrency control scheme that uses a polynomial time on-line scheduler. Papadimitriou and Kanellakis have shown that for the most general multiversion database model no such effective scheduler exists. In this paper we focus our attention on an efficient multiversion database model and derive necessary and sufficient conditions for ensuring serializability and serializability without the use of transaction rollback for this model. It is shown that both these classes yield additional concurrency through the use of multiple versions. This characterization is used to derive the first general multiversion protocol which does not use transaction rollback as a means for ensuring serializability.

1. Introduction

User response time in database systems can be improved by concurrent execution of user transactions. If each user transaction maintains the consistency of the database when executed alone, the database system must guarantee that any allowed concurrent execution of a set of transactions also maintains the consistency of the database. A system which guarantees this for any set of transactions is said to be serializable [1].

One recent method to increase concurrency is the use of the multiversion data item concept. This concept enhances concurrency by retaining individual updates of a data item as separate versions, and allowing a transaction to read one of several versions of a data item. The concurrency control must ensure that the versions read and written maintain serializability. This was used in the Honeywell FMS system [2], and has been formally developed and extended in the work of Reed [3], Stearns et al [4], Stearns and Rosenkrantz [5], Bayer [6], and Silberschatz [7]. Complexity results and necessary and sufficient conditions for the most general multiversion schemes were recently presented by Bernstein and Goodman [8] and Papadimitriou and Kanellakis [9].

There are some problems associated with the various multiversion database model proposals [10]:

1) The result in [9] states that there is no polynomial time on-line scheduler that maintains serializability and yet exploits maximum concurrency for the most general multiversion database model. This result makes study of the various multiversion models more interesting; for unlike single version database, the several multiversion protocols existing are not even uniformly based on the same database model.

2) In many models reading a data item also requires the updating of information in the data-item header, resulting in potentially two disk I/O rather than one.

3) In many models some of the conflicts between transactions are resolved through rollbacks rather than waits. If it is detected that a transaction may not be serializable upon transaction completion, then consistency is maintained by rolling back (removing) some number of transactions from the system and restarting them at a later time. Gray has observed that transaction restarts are very expensive [11].

It is our aim here to develop a multiversion database model that does not suffer from the above deficiencies. In particular, in section two we present an efficient multiversion database model, and derive

[1]This research was supported in part by the Office of Naval Research under Contract N00014-80-K-0987, and by the National Science Foundation under Grant 81-04017.

necessary and sufficient conditions for ensuring serializability and serializability without the use of transaction rollback for this model. In section three we present a new general protocol which fits this model. This protocol requires that each transaction predeclare its writeset. In this protocol a read does not result in updating of information in the database system. Also, no transaction rollbacks are required in order to assure serializability and deadlock freedom, implying that restarts occur only due to process or hardware failure. In section four we compare our protocol with the concurrency available with other protocols of this model. Finally, in section five we present optimizations for read-only transactions, an algorithm for version discarding, and methods for avoiding the need for a transaction to predeclare its writeset.

2. The Multiversion Database Model

Since there is no efficient method to maximize concurrency for the most general multiversion model, we must develop and characterize models which have effective concurrency controls and yet still make use of the availability of multiple versions of a data item. There are several important considerations when designing a restricted model.

a) It is crucial that a transaction can easily and quickly determine which version of a data item should be read.

b) Transactions should not be restarted often, if at all, and every transaction submitted to the system should eventually complete.

c) It is useful to have a number of readable versions of a data item available to allow interesting variation in scheduling of read-only transactions.

The multiversion database model described below easily meets all these objectives, and the behavior is discussed fully in the section on the new progressive protocol.

We now present the multiversion model, which is a simple generalization of Reed's model [3]. The database system is composed of data items, transactions, and the concurrency control. Each of these entities are defined by the following restrictions:

1. A transaction T_i consists of:
   a. a time ordered sequence of accesses to data items, which may be either read or write accesses. The items written need not be a subset of the items read.
   b. a static timestamp, denoted $TS(T_i)$. This is assigned by the database system before or at the time a transaction accesses its first data item. If T_i and T_j both write data items, then $TS(T_i) \neq TS(T_j)$.
2. A data item d has a sequence of versions $<d_1,\ldots,d_k>$ arranged in ascending order of timestamp, where version d_i has timestamp i. If transaction T_j creates version d_i, then $TS(T_j)=i$.
3. A concurrency control must satisfy the following criteria:
   a. At the first read access of a transaction T_j to data item d, it reads the version of d with timestamp closest to but less than $TS(T_j)$. All future read accesses are to the same version. (As a special case, a transaction that only reads data items can read a version with a timestamp equal to its timestamp. This case is explicitly covered in the proofs.)
   b. A transaction creates at most one version per data item, and this version is added to the sequence only when the transaction which created it will no longer update its contents.
   c. The protocol cannot use a global dependency graph to detect possible nonserializability. It must decide to accept or reject an access to d using any information derivable from the transactions which have accessed or will access d at some time, or derivable from the set of data items accessed by these transactions.

Several concurrency controls [6,12] have the protocol maintain dependency graph of the active transactions in the system. The protocol uses this graph to determine which version of a data item to read, and also to maintain serializability. We believe that this technique is expensive and slow when the number of interacting transactions in the system is large, and becomes prohibitively expensive when attempting to maintain a current global dependency graph in databases distributed over a number of different sites. To eliminate the expense of such a graph, by rule 3c we restrict the model to use information directly related to the access of a data item.

In order to present a complete characterization of the model several definitions and notational conveniences are now introduced that will be used throughout the remainder of the paper.

A protocol is said to be <u>safe</u> if it assures serializability. If a safe protocol does not use transaction rollback as a means for assuring serializability, then it is termed <u>progressive</u>. A version is <u>uncommitted</u> if it may later be removed due to rollback of the transaction that created it; otherwise it is called <u>committed</u>. A transaction which only read accesses data items is termed a <u>read-only</u> transaction; all other transactions are referred to as <u>update</u> transactions. A history H is the trace, in the chronological order, of a concurrent set of transactions $T = \{T_0, T_1, \ldots, T_{n-1}\}$.

We define a precedence relation $\rightarrow$ on a history H by writing $T_i \rightarrow T_j$ if and only if there exists a data item d, accessed (i.e., read or written) by T_i and T_j, such that either one of the following holds:

a. T_j created version d_j and T_i read or created version d_m, $m < j$.

b. T_j read version d_n and T_i created version d_i, $i \leq n$.

We say that T_i and T_j <u>interact</u> in the system if they are related via the $\rightarrow$ relation. If $T_i \rightarrow T_j$, then we say T_i <u>precedes</u> T_j. Since the versions ordered by increasing timestamp will be shown to be the serializable order, the definition of transaction conflict can be made as follows:

a) T_i read-write (or write-write) conflicts with T_j if T_j creates a version, and T_i reads or creates a version with timestamp less than $TS(T_j)$.

b) T_i write-read conflicts with T_j if T_i created a version with timestamp less than $TS(T_j)$.

A transaction can always access the value it has created for a data item, and does not come under the restrictions of reading a data item as defined by our model.

Using the multiversion model above, we now present the necessary and sufficient conditions that any protocol in this model must meet to be either safe or progressive. The conditions are simple and are based only on the respective timestamps of the transactions accessing a single data item. For brevity we have omitted the correctness proofs; they can be found, however, in [13].

We first present the necessary and sufficient condition to ensure serializability for any set of transactions executing in a multiversion database system model as described above.

S1: Let T_i and T_j be two transactions that interact in the system, where $TS(T_i) \leq TS(T_j)$. We shall say that T_i and T_j satisfy condition S1 if and only if the both of the following requirements are met:

a. If $TS(T_i) < TS(T_j)$, then $T_i \rightarrow T_j$.

b. If $TS(T_i) = TS(T_j)$, then without loss of generality, let T_i be the read-only transaction. (Recall that update transactions are required to have unique timestamps.) Then either $T_i \rightarrow T_j$ on all data items accessed by both transactions, or $T_j \rightarrow T_i$ on all data items accessed by both transactions.

Theorem 1: A database system that satisfies our multiversion database model is serializable if and only if every pair of transactions satisfy condition S1.□

Although S1 maintains serializability, it is possible to construct protocols fulfilling S1 which require transaction rollback. Hence, we introduce a new condition S2, which together with condition S1 preserves serializability without the use of transaction rollback. If no transaction reads a data item, it is trivial to show that S1 is sufficient to ensure serializability without rollbacks, since a version can be put in the proper place in the sequence of a data item at any time. If there exists at least one transaction which reads the value of a data item, then we must create a stronger condition than S1. As before, this new condition is necessary for both structured and unstructured databases. To ensure that a protocol will not require transaction rollback, we must enforce the following condition:

S2: Let T_i and T_j be two transactions which interact on data item d, where T_j reads a version of d and T_i at some time creates the version of d with highest timestamp that is readable by T_j, as specified by S1 and rule 3a. We shall say that T_i and T_j satisfy condition S2 if and only if T_i appends its version of d before the first access of T_j to d.

Condition S2 implies that a transaction may need to wait to read the appropriate version of a data item. Therefore we must prove both that S2 ensures deadlock freedom, and that S1 and S2 are necessary and sufficient for serializability without transaction rollback.

Theorem 2: A database system that satisfies our multiversion database model is serializable without transaction rollback if and and only if every pair of transactions satisfy conditions S1 and S2. □

3. The New Progressive Protocol

Any progressive protocol in this database model must meet conditions S1 and S2. This implies that a transaction T_i must delay a read request of data item d until the version d_k with timestamp closest to but less than $TS(T_i)$ has been inserted. Since we do not require write access to be a subset of read access, this may be well before all earlier versions have been inserted. Hence, it is easy to see that this characterization makes use of multiversions by eliminating entirely the need to delay for two of the three types of transaction conflict:

- the write-write conflict, and
- the read-write conflict.

Only a restricted form of the write-read conflict requires delay, when the transaction with higher timestamp wishes read access before the appropriate version has been inserted. These savings are a significant gain in the use of multiple versions.

This exposition allows the development of a new general progressive protocol that fits our model. It is necessary for a transaction to wait only for read access, and only until the transaction with lower but closest timestamp has been inserted. To accomplish this, we associate the following two data structures with each data item:

a. a sequence of versions of the data item, in ascending order of timestamp, and

b. a sequence of timestamps (in ascending order) of active transactions which create a version of this data item, but have not yet inserted the version into the sequence of versions.

When assigning a timestamp to a transaction, the model requires only that timestamps for update transactions must be unique. To obtain the flexibility necessary for an optimal protocol, we maintain an avail list of available timestamps. When an update transaction enters the system, it chooses the smallest unmarked timestamp in the avail list if it issues read requests; otherwise it can select any arbitrary unmarked timestamp. The transaction then marks the timestamp to prevent duplicate issues, inserts its timestamp into the sequence of each data item in its writeset, and then removes its timestamp from the avail list. A read-only transaction can select any timestamp between 1 and one less than the smallest number in the avail list. After this preprocessing is completed, a transaction begins execution. The rules to read or write a data item d are as follows:

1. A transaction T_i updates data item d by performing its final write of d, inserting its version with timestamp $TS(T_i)$ into the correct order in the version sequence, and then deleting $TS(T_i)$ from the timestamp sequence of d.

2. When transaction T_i performs its first read of any data item it must wait until all timestamps less than $TS(T_i)$ have been removed from the avail list. It then performs its first read of data item d by finding the timestamp (denoted by j) in the timestamp sequence of d closest to but less than $TS(T_i)$, if there is one. If none exists, then all previous versions have been inserted, and T_i may select the appropriate version by the rule given in the concurrency control. If some timestamp j exists, then T_i determines if there is some version d_k in the sequence such that $j < k < TS(T_i)$ (or $j < k \leq TS(T_i)$ if T_i is a read-only transaction), and which T_i can immediately read, by the given rule. Otherwise, T_i must wait until transaction T_j with timestamp j creates the intended version.

Theorem 3: The new multiversion protocol is progressive; that is, it ensures serializability and deadlock freedom without the use of transaction rollback.

Proof: We show that any precedence relation between transactions T_i and T_j imply S2. Assume, without loss of generality, that $TS(T_i) \leq TS(T_j)$. We separate the proof into four cases, the first three cases specify $TS(T_i) < TS(T_j)$, and the last case has $TS(T_i) = TS(T_j)$.

1. T_i and T_j both create versions of data item d. By the definition of the precedence relation, $T_i \rightarrow T_j$.

2. T_i reads a version of d, and T_j creates a version of d. By rule 3a, T_i must read a version with a timestamp less than or equal to its own timestamp, and so $T_i \rightarrow T_j$, by definition of the precedence relation.

3. T_i creates a version of d, and T_j reads a version of d. The protocol states that T_j must wait until all smaller timestamps have

been removed from the avail list, and consequently T_i had added its timestamp to the timestamp sequence of d. The protocol delays T_j until some version d_k has inserted its version, where $i \leq k \leq TS(T_j)$. This directly implies condition S2.

4. For the cases where $TS(T_i) = TS(T_j)$, rule 1b stipulates that only one transaction can create versions. Without loss of generality we assume T_i creates a version and T_j reads a version, where T_j only read accesses data items. The protocol specifies that T_j does not read until T_i removed its timestamp from the avail list, which is after it added its timestamp to the timestamp sequence of data item d. Hence, T_j must wait to access d until after T_i created its version, if T_i consistently precedes T_j; or else T_j consistently precedes T_i. □

We present a short example to illustrate the behavior of the new protocol. The database consists of the data items a, b, and c, each with a base version available to read. There are two update transactions, T_1 and T_2, and one read-only transaction, T_3. T_1 will read a and write a version of b, T_2 will read a and b and write a version of c, and T_3 will read a and c. T_1 enters the database, is assigned $TS(T_1)=1$, appends 1 to the timestamp sequence of b, and reads a. T_2 enters, is assigned a timestamp of 2, appends 2 to the timestamp sequence of c, reads a, and waits for T_1 to insert its version of b. T_1 inserts its version and completes, and T_2 reads b. T_3 enters, and can select a timestamp of 1 or 2. T_3 can proceed without delay if it selects $TS(T_3)=1$, or can wait for more current results by selecting $TS(T_3)=2$. It chooses a timestamp of 2, reads a, and waits until T_2 inserts its version of c.

This very simple protocol avoids many of the performance drawbacks of ,other published multiversion protocols. First, there is no wasted execution time or system overhead due to transaction rollback. This absence of rollback guarantees completion of all transactions entering the system, and also decreases the waiting time to read an individual version. This is due to the fact that every execution completes, so once a transaction inserts a version it can be considered committed and read immediately. This differs from most other multiversion protocols, where either a transaction reads an uncommitted version and may be involved in cascading rollback, or waits until the transaction creating a version has finished its execution. In addition, there is no need to maintain a transaction dependency graph, nor use a cycle detection algorithm to determine either serializability or selection of the appropriate version to read. Finally, a read operation can be executed without any updates.

4. Comparison

We now compare the new protocol with the other two existing protocols using the same multiversion database model.

The first protocol is a safe protocol proposed by Reed [3]. The protocol assigns unique timestamps to each transaction in the order they enter the database. When a transaction issues a read request, it reads the most current version less than its own timestamp if the version is committed, otherwise it delays until the version is committed or removed. A read operation may result in the updating of information concerning the latest time that version has been read. A new version with timestamp t is installed after version p, where p is the largest number less than t. Version t is installed only if no transaction with timestamp greater than t has read p, otherwise the transaction creating version t must be rolled back. However there is no cascading rollback, since transactions are restricted to reading only committed versions. Thus, Reed's protocol also suffers from only one case of write-read conflict, which occurs when transactions writing versions with smaller timestamps have not yet completed. If the version has been inserted, the read is delayed until transaction completion; otherwise, the read is processed immediately but causes the abortion of the transaction when it attempts to insert its version.

We now contrast the performance of Reed's protocol using the example given above. T_1 enters the database, is assigned a timestamp of 1, and reads a. T_2 enters, is assigned a timestamp of 2, and reads a and b. Now T_1 attempts to update item b, but is rejected due to the read issued by T_2. T_3 enters, is assigned the next unique timestamp (3), and reads a and c. T_2 now attempts to update c and must also be rolled back.

In general, Reed's protocol allows read requests for transactions to be granted earlier than in our protocol only when it will indeed cause earlier transactions to be rolled back. This is due to the fact that our protocol only delays a read when a version will indeed be inserted. Our protocol also allows a version to be read <u>as soon as it is created</u>, since it will <u>always</u> be committed, while Reed's protocol must wait until the transaction creating the version has completed. Lastly, our protocol does not require the updating of information in the database, while Reed's protocol does. The disadvantages of our scheme are the initial overhead for timestamp assignment, and that a transaction must declare its writeset.

The new protocol can be easily shown to encompass the tree multiversion protocol in [7] as a special case. That protocol operates in a database structured as a tree, where all update transactions starts at the root, and are given timestamps in ascending order when each successfully locks the root of the tree. Read-only transactions may begin anywhere in the tree. Update transactions issue only X locks, and may overtake any read-only transaction's S locks as it traverses down the tree, but read-only transactions may only overtake X locks with higher timestamp. Hence, no transaction with a higher timestamp intending to read some data item can overtake an update transaction which may create a version of lower timestamp, and this behavior thereby maintains the write-read delay required by S2. Consequently, all transactions with higher timestamps will follow the update transactions of lower timestamp down the tree. Using this description of the multiversion tree protocol, one can extend the performance in several small ways. Considered in the light of our new protocol, this progressive protocol is a special case that exchanges the time needed for the mutual exclusion of the update timestamp assignment for the delay time of access to the lower portions of the graph.

5. Discussion

We now discuss three questions related to performance of our scheme. We first discuss optimal choices when selecting a timestamp for a read-only transaction. Since a transaction can select a timestamp from a wide range of values, the transaction can avoid being delayed by selecting a timestamp less than the minimum value over all TM's associated with the data items it accesses. If the timestamp is much less than the minimum, the versions read may be quite old and have been superseded by many more current versions. If the timestamp is the minimum value minus one, this gives the desirable property of reading the most current version of the set of contiguous completed transactions at the time the read transaction enters the database. Finally, if a read-only transaction wishes to read the most current values possible, it would select the largest timestamp assigned to an update transaction. Thus it would read the most current update of any transaction currently active in the system. This may cause delay. However, since no update transaction need be restarted, this delay may be acceptable for many systems.

The second issue involves the question of when old versions of a data item can be discarded. This algorithm was previously presented in [7], and we present a short summary here. If the concurrency control can determine some minimum timestamp n that any active or future transaction is assigned, it can delete all versions d_k of d such that there exists some version d_m with $k < m < n$. The minimum timestamp for an update transaction is the earliest active transaction, since the timestamps are assigned in ascending order. This is easily found by keeping a sequence of the timestamps of all active update transactions. The minimum timestamp assigned to a read-only transaction must be artificially set by the system. The system must determine when the active read-only transactions with smaller timestamp have finished; this can be done using several counters counting the number of read-only transactions within designated ranges that have entered and not yet left the system. When the value of the counters with ranges less than n has gone to zero, n becomes the minimum active read-only timestamp.

The third issue involves the question of what can be done if it is not possible to estimate the writeset of a transaction. There are two possible answers:

a) Use a multilevel granularity of data items solution. A transaction declares its writeset to include the highest needed data item in the hierarchy. A scheme that can be effectively used with our protocol was recently proposed by Carey [14].

b) Redefine our protocol as follows: The system maintains a new list, called Undeclared-Update, which contains the time stamps of all the active update transactions that cannot estimate their writeset. When a transaction T_i obtains its time stamp, $TS(T_i)$, it is checked whether $TS(T_i) < Min[Update]$. If so, then the execution proceeds as before. Otherwise, every read by transaction T_i can proceed only when there exists a version of X with time stamp Y such that Y is the closest time stamp to $TS(T_i)$ in either the version sequence, the time stamp sequence, or the Undeclared-Update list.

6. Conclusion

This paper presents the first complete characterization for an effective multiversion database model. We demonstrated that this model has effective protocols that ensure both safety and progressiveness. Using these characterizations, we developed a new general progressive protocol which has useful behavior characteristics not offered by any other protocol applicable to a general database system. The flexibility of assigning timestamps allows performance tuning and algorithms for discarding old versions based only on the respective timestamps of the transactions.

We reemphasize that the elimination of the use of transaction rollback in the new protocol serves two important purposes. First, it eliminates the bookkeeping overhead, the wasted processing, and the restart delays connected with rollback. Second, it removes the restriction that a read must occur only on a committed version, since all versions a transaction creates or reads will maintain serializability and deadlock freedom. Hence, there is no need to delay access to a version until the execution of a transaction proceeds past a certain point. Most other multiversion protocols in the literature require this in order to avoid cascading rollback. We note, however, that cascading rollbacks can occur in our scheme in case of process or hardware failure. Thus in order to assure atomicity, a commit protocol must be used in conjunction with our protocol.

References

1. Eswaran, K.P., Gray. J.N., Lorie, R.A. and Traiger, I.L. The notions of consistency and predicate locks in a database system. CACM 10, 11 (Nov. 1976), 624-723.

2. Honeywell File Management Supervisor, Order Number DB54, Honeywell Information Systems Inc., 1973.

3. Reed, D.P. Naming and synchronization in a decentralized computer system. Ph.D. Thesis, M.I.T. Dept. of Electrical Engineering and Computer Science, Sept. 1978.

4. Stearns, R.E., Lewis, P.M and Rosenkrantz, D.J. Concurrency control for database systems. Proceedings IEEE Symposium on Foundations of Computer Science (Oct. 1976), 19-32.

5. Stearns, R.E. and Rosenkrantz, D. Distributed database concurrency control using before values. Proceedings ACm-SIGMOD International Conference on Management of Data (April 1981).

6. Bayer, R., Elhardt, E., Heller, H. and Reiser, A. Distributed concurrency controls in database systems. Proceedings Sixth International Conference on Very Large Data Bases (Oct. 1980), 275-284.

7. Silberschatz. A. A Multiversion Concurrency Control Scheme with No Rollbacks, Proceedings ACM SIGACT-SIGOPS Symposium on Distributed Computing (August 1982), 216-223.

8. Bernstein, P. and Goodman, M. Concurrency Control Algorithms for Multiversion Database Systems, Proceedings ACM SIGACT-SIGOPS Symposium on Distributed Computing (August 1982), 209-215.

9. Papadimitriou, C. and Kanellakis, P. On Concurrency Control by Multiple Versions, Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Database Systems (March 1982), 76-82.

10. Gray, J.N. The Transaction Concept: Virtues and Limitations, Proceedings 7th International Conference on Very Large Data Bases, (September 1981), 144-154.

11. Gray, J.N. "A Discussion of Distributed Systems," Invited Lecture at the Congresso Annuale of Associazione Italiana per il Calcolo Automatico, Bari, Italy (August 1979).

12. Muro, S., Kameda, T., and Minoura, T. Multi-version Concurrency Control Scheme for a Database System, Technical Report 82-2, University of Toronto, (February 1982).

13. Buckley, G.N. and Silberschatz, A. A Complete Characterization of a Multiversion Database Model with Effective Schedulers, Technical Report TR-217, The University of Texas at Austin, March 1983.

14. Carey, M. Granularity Hierarchies in Concurrency Control, Technical Report, University of California at Berkeley, January 1983.