Improving Concurrency Control in Distributed Databases with Predeclared Tables

Azzedine Boukerche and Terry Tuck

PARADISE Research Laboratory
Dept. of Computer Sciences, Univ. of North Texas
boukerche@cs.unt.edu
terry.tuck@sw.boeing.com

Abstract. In this paper we present a concurrency control algorithm and recovery protocol for distributed databases that produces a schedule that is equivalent to that of a temporally ordered serial schedule. The algorithm is intended to be somewhat practical from the standpoints of both burden on the application developer and realism of the network environment. Accordingly, it allows transactions to be received out of order within a window of tolerance. Moreover, it doesn't require declaration of readsets, and writesets are declared simply at the table level and without predicates. This paper discusses the algorithm, present its analytical studies, and report on the simulation experiments we carried out to evaluate the performance of ours scheme. Our results clearly indicate that with predeclared tables, performance is greatly improved as compared with that from the conservative MVTO algorithm.

1 Introduction

There appears to be a gap in previous database transaction concurrency control studies on practical concurrency control methods that preserve temporal dependencies only in the presence of data dependence. To fill this gap, concurrency control methods are needed that efficiently produce chronologically ordered execution schedules for data-dependent transactions (for the sake of preserving temporal dependencies) while simultaneously taking advantage of potential performance gains by supporting out-of-order (i.e., concurrent) execution of data-independent transactions. This paper addresses the above-mentioned gap in light of these assumptions. We present a transaction concurrency control algorithm that is designed to execute transactions in a high-performance manner while producing conflict-serializable schedules that preserve temporal dependencies that coexist with data dependencies. The algorithm is intended to be (a) useful in real-world on-line transaction processing (OLTP) applications that are conforming to the above assumptions, (b) practical for the application developer, and (c) applicable to both distributed and centralized databases.

The remainder of this paper is organized as follows. In section 2, we describe the DDBS model we used as a basis for this paper. In section 3, we present our algorithm followed by its analytical performance in Section 4. In section 5, we present the

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 301-309, 2001.

simulation set of experiments we carried out to evaluate the performance of our scheme. Finally, in Section 6 we conclude this paper.

2 DDBS Model

We view a database as a persistent store for a collection of named data items partitioned into disjoint sets that we refer to as tables. (Note: although the term 'table' implies the relational data model, the algorithm doesn't appear to be restricted to this application.) A distributed database system (DDBS) is viewed as a collection of logically interrelated databases distributed among a group of computer nodes that are interconnected via an assumed reliable network. In our view of the DDBS, we consider each data item to be tied to a particular database; that is, the DDBS is restricted to the non-replicated case. In our DDBS model, a single Transaction Manager (TM) manages each transaction in a dedicated/centralized fashion, and multiple TMs operate independently of each other. The TM executes each logical database operation in client/server style by dispatching the corresponding message to the target database within DDBS. A concurrency control algorithm governs the various synchronization mechanisms employed by the DDBS to control the execution order of database operations. This execution order is called a schedule, and it is, in general, correct if it is equivalent to any other schedule in which the transactions are executed serially (i.e., the schedule is serializable). The design requirements discussed earlier directly influenced key characteristics of our algorithm and its corresponding DDBMS model. The requirement of immediate execution of *write* operations requires, in turn, support for multiple versions of each data item. Reducing semantic incorrectness associated with the return of an incorrect data item version suggests a non-aggressive, if not conservative, synchronization technique. The requirement for an execution schedule that is equivalent to a timestamp-ordered serial schedule necessitates the use of timestamp ordering. Finally, the requirement for improved concurrency combined with a non-aggressive synchronization technique requires predeclaration of the data items to be accessed. Combining this with the requirement for ease of use for the application developer requires that predeclaration be done at other than the data-item level.

3 Proposed Concurrency Algorithm

The correctness constraint for all schedules produced by our concurrency control algorithm is that they are conflict serializable and computationally equivalent to a temporally ordered serial schedule for the same set of transactions. It has been shown that the problem of concurrency control based on conflict serializability is decomposable into sub-problems of synchronizing operations involved in the two types of conflict, *read-write* and *write-write* [1,5,6]. These two sub-problems can be solved independently as long as their solutions are combined in a manner that yields an overall transaction ordering. Decomposing the overall problem into *write-write* and *read-write* synchronization subproblems also simplifies the presentation of our algorithm.

In accordance with the design constraints, *write* operations are executed immediately upon receipt by a local database. This requires support for coexisting

versions of each data item; that is, a multiversion DDBS. With a multiversion database *write-write* synchronization is trivial: each version of a data item is uniquely identified with the globally unique timestamp of its creating transaction. This uniqueness effectively eliminates conflict between any two *writes* targeting the same data item, thereby making the order of their execution inconsequential.

Read-write synchronization, on the other hand, is somewhat more complicated in our scheme. Five rules of operation establish a framework within which *read* operations are synchronized with *writes*. In describing these rules, let $W\text{-}ts(x^k)$ represent the creation timestamp of version k of data item x, and $ts(T_i)$ represent the timestamp of transaction i. The rules are as follows:

Write rule: upon receipt of $write_i(x,y)$ from T_i, create a new data-item version x^k with value y and timestamp $W\text{-}ts(x^k)$ set to $ts(T_i)$.

Read rule: upon receipt of $read_i(x)$ from T_i, return the value of data-item version x^j such that $W\text{-}ts(x^j)$ is the largest timestamp such that $W\text{-}ts(x^j) < ts(T_i)$.

The Read rule, when combined with the design constraint of reducing semantic incorrectness associated with the return of a temporally incorrect value for the targeted data item, suggests that each database delays its response to a *read* request until the correct version is both available and committed. This gives rise to the following delay rule.

Delay rule: A $read_i(x)$ from T_i will not be processed while either (a) an older, uncommitted transaction has declared for update the table containing x, or (b) the creating transaction for the accessed data item version (per the Read rule) is still uncommitted.

The Delay rule ensures that each database will respond to a *read* request with the correct and committed data item version as long as the *begin* message from the version's creating transaction is received at the database prior to its receipt of the *read*. However, it is possible for the writer's *begin* message to be delayed by the network to the extent that it is received at the database after the *read*. Then, depending on the commit state of the reader, one of the following two mutually-exclusive rules is appropriate:

Abort rule: A transaction T_t will be aborted if it reads a data item version that is older than a version (of the same data item) created subsequently by a straggling transaction $T_{t\text{-}c}$, where $ts(T_{t\text{-}c}) < ts(T_t)$. In other words, if a database responds to some $read_t(x)$ with a data item version with $W\text{-}ts(x) < ts(T_{t\text{-}c})$, and then a $write_{t\text{-}c}(x)$ is subsequently received at the database, then transaction T_t will be aborted so that it will repeated and returned the correct data-item version.

Reorder rule: The timestamp $ts(T_t)$ of transaction T_t will be reset if it declares for update any table that has been read by a younger committed transaction T_{t+d}. The new timestamp will be set to a value such that the transaction $T_{t'}$ is made to be younger than any committed reader on any of its declared tables.

The Abort and Reorder rules provide the basis for *read-write* synchronization when the *begin* of a writing transaction is late in reaching one of the participating local databases. The Abort rule is similar to that found in other approaches, with the exception that aborted transactions maintain the timestamp with which they started. The Abort rule is applicable in cases where a *begin* is late, but no reader has yet committed having accessed a table included in the *begin*'s table list declaration. The

corrective action in such cases is to abort and restart the reader when a straggling writer creates a data item version with a timestamp that is on the interval of the timestamps of the reader and the erroneously returned version (of the same data item). The Reorder rule, however, is unique to our concurrency scheme.

Table 1: Summary of 2-TM Concurency system parameters

Transaction generation rate, λ	At sites i and j, modeled as independent Poisson processes.
Transaction processing time, S	PDF for time between TM's sending of begin/read and write/commit messages.
Inter-transactional table-level conflict, ϕ_{ji}^{k}	Fraction of writes from site j having table-level conflict at site *k* with reads from site i.
Inter-transactional data-item-level conflict, γ_{ji}^{k}	Fraction of writes from site j having data-item-level conflict at site *k* with reads from site i.
Transmission delays, t and t'	PDF for delays, including pipelining delays.

Whenever a transaction's *begin* message arrives at a local database at which some younger reader has already committed, there is a chance that allowing the transaction to proceed will lead to a non-serializable schedule. This chance exists only when a younger reader has read from one of the tables included in the table list of the delinquent *begin*. It actually occurs whenever there is data-item-level conflict between the committed reader and some *write* to be issued by the transaction associated with the delinquent *begin*.

Unfortunately, it appears that true data-item-level conflict with committed readers cannot be verified without employing costly read-tracking mechanisms at the data-item level. Consequently, a pessimistic approach is taken, and any table-level conflict between a committed reader and delinquent older writer is handled as a data-item-level conflict.

In the presence of younger committed readers with table-level conflict, the Reorder rule is applied to a delinquent *begin* in order to reposition its associated transaction in the temporal order to the earliest point at which the transaction can be safely executed in a serializable fashion at all participating local databases. In summary, each local database hosting a table with a younger committed reader replies to the *begin* message with a safe timestamp, the associated Transaction Manager adopts the maximum of all such replies, and then communicates the new temporal position to the application/entity initiating the transaction for a go/no-go decision.

In Boukerche et. al. [2], we present the high-level pseudo code listings which illustrate how the Transaction Manager and Database components operate in order to provide concurrency control according to the stated rules.

4 Analytical Performance of Our Concurrency Scheme

Our analysis focuses on the probabilities and delays attributed to transaction conflict. The most general case of conflict occurs when transactions issue both *read* and *write* operations to each local database, so this case is chosen for all transactions. A context is established by making the following assumptions. First, transactions are generated at different TM sites of the DDBS as independent Poisson processes. Second, local processing delays are negligible in comparison to communication delays. Third,

transaction generations and communications delays are independent. Finally, in accordance with prior assumption, all clocks within the system are synchronized.

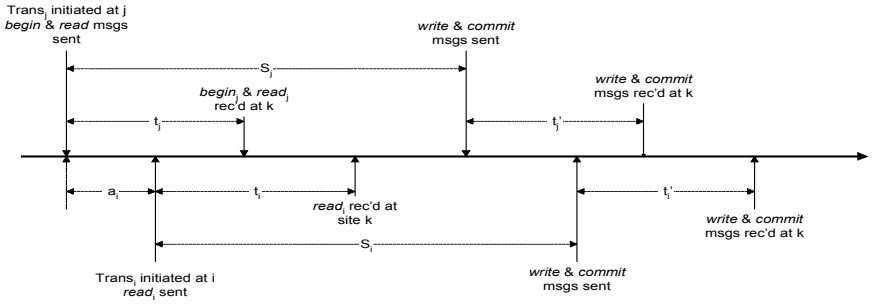

Figure a: Transaction event timeline

The transaction model is modified slightly. Transactions are restricted to be 2-step. In addition, for the purpose of the analysis, the transaction model is such that all *begin* and/or *read* messages are simultaneously dispatched at the exact moment the transaction is started. Similarly, after a period of time for transaction processing, all *write* and *commit* messages are simultaneously dispatched. Finally, a single-phase commit protocol is used. For more details and a complete proof correctness of this scheme, the authors may wish to consult [2].

(i) Conflict Model for a 2-TM MVTC System: Consider two sites hosting TMs, *i* and *j*. Suppose that these two sites generate transactions as independent Poisson processes with rates λ_i and λ_j, respectively. Also suppose that a third site, *k*, exists which hosts a database which will be accessed via a pipelined network by transactions at *i* and *j*.

Suppose that transaction T_j is initiated at time $ts(T_j)$ at site *j*. Further, suppose that this transaction will attempt to write some x_n in table *X* at site *k*. At the instant the transaction initiates, $begin_j(X)$ and $read_j(x_n)$ are simultaneously sent to site *k* in order to register T_j as a writer on table *X* and to start its *read*. The propagation delay for this message is represented by t_j. Assume that T_j performs local processing that will be concluded with simultaneous transmission of both $write_j(x_n)$ and $commit_j$ messages to site *k*. The processing time is represented with S_j (See Figure above)

Similarly, suppose that a transaction at site *i,* T_i, is initiated at time $ts(T_j) + a_i$, which is later than $ts(T_j)$. This transaction will attempt to read some x_m in table *X* at site *k*, and at the time of initiation a $read_i(x_m)$ is simultaneously sent site *k*. The transaction's processing time is represented with S_i.

(ii) Analysis of the 2-TM MVTC System: We now derive the probabilities and expected times associated with the different ways these two transactions can interact. We use ϕ_{ji}^k and γ_{ji}^k to represent the fraction of *writes* from all transactions originating at site *j* that will conflict at the table level and data-item level, respectively, at site *k* with *reads* from transactions originating at site *i*.

Lemma 1: The probability that $read_i$ is blocked at site *k* for an older transaction T_j is given by

$$BR_{ij}^{k} = \phi_{ji}^{k} P\left(t_j < a_i + t_i \wedge S_j + t_j^{'} > a_i + t_i\right)$$

Lemma 2: The probability that $commit_i$ is blocked at site k for older transaction T_j is given by

$$BC_{ij}^{k} = \phi_{ji}^{k} P\left(a_i + t_i < t_j < a_i + S_i + t_i^{'}\right).$$

Lemma 3: The probability that $begin_j$ is rejected at site k for younger committed transaction T_i is given by

$$RB_{ji}^{k} = \phi_{ji}^{k} P\left(t_j > a_i + S_i + t_i^{'}\right).$$

We now derive expressions that approximate the performance measures in a 2-TM MVTC system.

Theorem 1: *Given exponential approximations for* t_i, t_j, *and* S_j *with respective means of* $1/\mu_i$, $1/\mu_j$, *and* $1/s_j$, *the probability that* $read_i$ *is blocked at site* k *for an older transaction* T_j *is approximated by*

$$BR_{ij}^{k} = \phi_{ji}^{k} \frac{\mu_i \mu_j \lambda_i \left(\lambda_i + \mu_i + \mu_j\right)}{\left(\lambda_i + s_j\right)\left(\mu_i + s_j\right)\left(\lambda_i + \mu_j\right)\left(\mu_i + \mu_j\right)}$$

Theorem 2: *Given exponential approximations for* t_i, t_j, *and* S_j *with respective means of* $1/\mu_i$, $1/\mu_j$, *and* $1/s_j$, *the expected time that* $read_i$ *is delayed by a block at site* k *for an older transaction* T_j *is approximated by*

$$E\left(dr_{ji}^{k}\right) = \frac{1}{s_j} \frac{\lambda_i \left(\mu_i + \mu_j\right)}{\lambda_i \left(\mu_i + \mu_j\right) + s_j \left(\mu_i + \lambda_i + s_j\right)} + \frac{1}{\mu_j}$$

Theorem 3: *Given exponential approximations for* t_i, t_j, *and* S_i *with respective means of* $1/\mu_i$, $1/\mu_j$, *and* $1/s_i$, *the probability that* $commit_i$ *is blocked at site* k *for an older transaction* T_j *is approximated by*

$$BC_{ij}^{k} = \phi_{ji}^{k} \frac{\lambda_i \mu_i \mu_j \left(\mu_i + s_i + \mu_j\right)}{\left(\lambda_i + \mu_j\right)\left(s_i + \mu_j\right)\left(\mu_i + \mu_j\right)^2}$$

Corollary 1: *Given exponential approximations for* t_i, t_j, *and* S_i, *with respective means of* $1/\mu_i$, $1/\mu_j$, *and* $1/s_i$, *the probability that transaction* T_i *is restarted because of a missed* $write_j$ *at site* k *is approximated by*

$$RT_{ij}^{k} = \gamma_{ji}^{k} \phi_{ji}^{k} \frac{\lambda_i \mu_i \mu_j \left(\mu_i + s_i + \mu_j\right)}{\left(\lambda_i + \mu_j\right)\left(s_i + \mu_j\right)\left(\mu_i + \mu_j\right)^2}$$

Theorem 4: *Given exponential approximations for* t_i, t_j, S_i, *and* S_j *with respective means of* $1/\mu_i$, $1/\mu_j$, $1/s_i$, *and* $1/s_j$, *the expected time that* $commit_i$ *is delayed by a block at site* k *for an older transaction* T_j *is approximated by*

$$E\left(dc_{ji}^{k}\right)=\frac{1}{s_j}\frac{\left(\lambda_i+\mu_j\right)\left(s_i+\mu_j\right)\left(\mu_i+\mu_j\right)}{\left(\lambda_i+\mu_j\right)\left(s_i+\mu_j\right)\left(\mu_i+\mu_j\right)+\lambda_i\left(s_i+s_j\right)\left(\mu_i+s_j\right)-s_i\mu_i}+\frac{1}{\mu_j}$$

Theorem 5: *Given exponential approximations for* t_i, t_j, *and* S_i, *with respective means of* $1/\mu_i$, $1/\mu_j$, *and* $1/s_i$, *the probability that* $begin_j$ *is rejected at site* k *because of a younger committed transaction* T_j *is approximated by*

$$RB_{ji}^{k}=\frac{s_i\lambda_i}{\left(s_i+\mu_j\right)\left(\lambda_i+\mu_j\right)}\left(\frac{\mu_i}{\mu_i+\mu_j}\right)^2$$

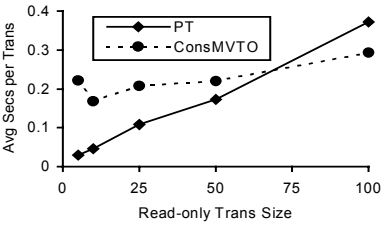

Fig. 1 - Read-only Turnaround -v- Trans Size

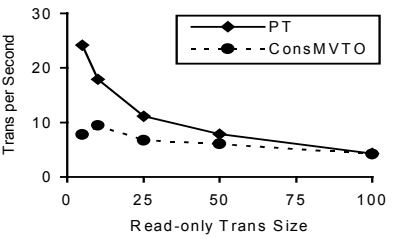

Fig. 2 - Read-only Throughput -v- Trans Size

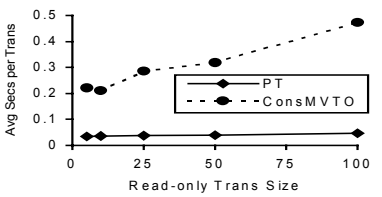

Fig. 3 - Update Turnaround -v- Trans Size

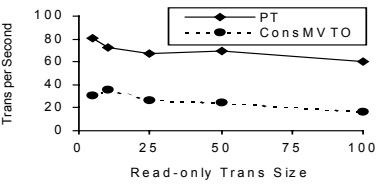

Fig. 4 - Update Throughput -v- Trans Size

5 Simulation Experiments

In order to acquire performance measures, a functional model of a DDBS with our algorithm was implemented in Java. This model includes stand-alone TM and DB components that communicate via Java RMI. To support post-execution analyses, time-stamped operations were recorded within each DB component. Each DB was configured identically with 20 tables of 25 data items ("columns"), for a per-DB total of 500 data items. The experiments were performed on a cluster of Intel Pentium III (350 MHz) PCs running Microsoft Windows NT Workstation 4.0 with Service Pack 5, interconnected with a non-dedicated 10 Mbs LAN, and using the JavaSoft Java™ 2 Runtime Environment, Standard Edition, v 1.2.2_006. Three types of experiments are presented below. These experiments provide insight on the performance of the algorithm when transaction density is high. (At each TM, no delay was induced between the completion of a transaction and the start of the next.) They follow from those in [1,4].

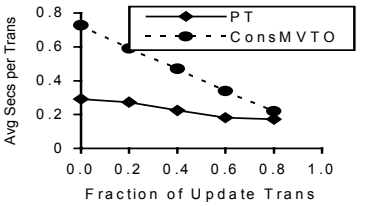

Fig. 5 - Read-only Turnaround -v- Update Fraction

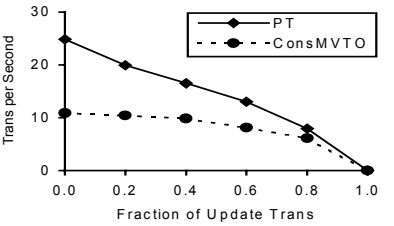

Fig. 6 - Read-only Throughput -v- Update Fraction

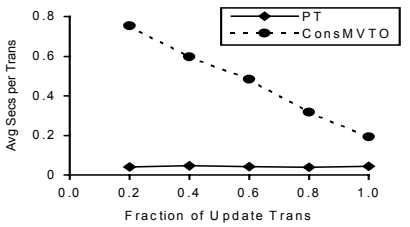

Fig. 7 - Update Turnaround -v- Update Fraction

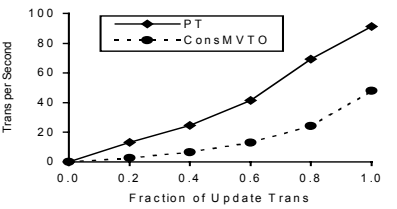

Fig. 8 - Update Throughput -v- Update Fraction

In our first experiment we vary the size of read-only transactions in a mix of small update transactions. The goal of this experiment is to investigate the performance of read-only and update transactions when inter-transaction conflict is almost exclusively between the two types of transactions (i.e., minimal conflict between any two update transactions.) The size of the Consequently, the size of the update transactions was limited to 2 randomly chosen data items. However, unlike the update transactions, each read-only transaction accessed a set of adjacent data items, thereby focusing access to fewer tables.The ratio of update to read-only transactions was fixed at 4 to 1. This was accomplished by restricting each TM to one of the two types of transactions, and allowing each to execute as many transactions as possible within a run. Finally, in each run, 10 TMs executed simultaneously against a single database. Experiment runs were executed with mean sizes of 5, 10, 25, 50, and 100 data items for the read-only transactions. Figs 1-4 show the performance of the read-only and update transactions as measured in turnaround time (i.e., response time) and throughput. The measured values are plotted against the size of the read-only transactions in each run, with “PT” indicating the results with Predeclared Tables, and “ConsMVTO” indicating the performance of the conservative MVTO algorithm. With respect to read-only transactions, the results are expected. For update transactions, the curves for our algorithm, PT, for both throughput and turn-around time would likely be flatter with a more powerful host for the database. The algorithm was designed to eliminate delays on write operations, so the decreased performance with larger read-only transaction sizes appears to result from processor contention at the database host. With the exception of the turnaround time for very large readsets, each of the graphs shows improved performance with the predeclared tables.

In our second set of experiments, we vary the mix of read-only and update transactions to investigate the performance of the two types of transactions. Targeted data items for these types of transactions were selected in the same manner as that in the first experiment. Transaction size for the updates was also the same, and the mean size for the read-only transactions was fixed at 50. Again, 10 TMs were executed

simultaneously, with each dedicated to one of the two transaction types. Experiment runs were executed with [0,..,10] of the 10 TMs dedicated to update transactions. Figs 5-8 show the performance of the read-only and update transactions, respectively, as measured in turnaround time and throughput. These metrics are plotted against the fraction of update transactions in each run. In this scenario, the performance of both types of transactions is greatly improved with the predeclared tables.

6 Conclusion

We have described an efficient algorithm for controlling transaction concurrency in a distributed database system that is suitable for environments free of long-lived update transactions. We have presented an analytical study of our scheme as well as a set of simulation experiments to evaluate the performance of our scheme. When compared with the conservative MVTO scheme, the results obtained show a significant improvement, which is a result of our improved concurrency gained by eliminating needless blocks. Our plan for future work is to further investigate the experimental performance of our concurrency scheme. An optimistic version of this algorithm will also be studied [2,3]. It is also aimed at preserving temporal order, but relaxes the constraint on temporal semantic correctness. The relaxation introduces rollback as the recovery mechanism for transactions that have been returned a temporally incorrect data-item value.

Acknowledgments

This work was supported by UNT Research Grant.

References

[Bouk01] A. Boukerche and Terry Tuck, “T3C: A Temporally Correct Concurrency Control Algorithm for Distributed Databases’’, IEEE/ACM MASCOTS 2000.

[Bouk01] A. Boukerche and Terry Tuck “Concurrency Control Mechanisms in Distributed Database Systems,” UNT Tech. Report – In Preparation

[Bouk99] A. Boukerche, S. K. Das, A. Datta, and T. LeMaster, “Implementation of a Virtual Time Synchronizer for Distributed Databases”, IEEE/ACM IPDPS’99.

[Care86] M. J. Carey and W. A. Muhanna, “The Performance of Multiversion Concurrency Control Algorithms,” ACM Trans. on Comp. Sys., Vo. 4, 1986.

[Özsu99] M. T. Özsu, P. Valduriez, Principles of Distributed Database Systems, Upper Saddle River, NJ: Prentice Hall, 1999.

[Thom96] A. Thomasian, atabase Concurrency Control: Methods, Performance, and Analysis, Boston, MA: Kluwer Academic Publishers, 1996.