ENHANCING THE PERFORMANCE OF PRESUMED COMMIT PROTOCOL*

Yousef J. Al-Houmaily
Dept. of Electrical Engineering
University of Pittsburgh
Pittsburgh, PA 15261
yjast1+@pitt.edu

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
panos@cs.pitt.edu

Steven P. Levitan
Dept. of Electrical Engineering
University of Pittsburgh
Pittsburgh, PA 15261
steve@ee.pitt.edu

Keywords: Atomic Commit Protocols, Read-Only Transactions, Distributed Database Systems.

ABSTRACT

This paper presents a new read-only optimization called the *unsolicited update–vote* that when combined with the presumed commit protocol (PrC), eliminates *all* the logging activities from PrC for read-only transactions and significantly reduces them for partially read-only ones.

1 INTRODUCTION

To ensure consistent termination of distributed transactions despite site and communication failures, all the sites participating in a transaction's execution engage in an *atomic commit protocol* (ACP). The *two-phase commit* (2PC) protocol [1] is the simplest and most used ACP. Since 2PC consumes a substantial amount of a transaction's execution time due to the cost of its coordination messages and forced log writes to stable storage required for recovery, a number of 2PC variants appear in the literature, most notably, *presumed abort* (PrA) and *presumed commit* (PrC) [2]. As opposed to PrA, PrC has been designed to reduce the cost associated with committing transactions rather than aborting ones. However, PrC applicability is curtailed due to its cost to commit read-only transactions which are the majority of transactions in any general database system. Even though the traditional *read-only* optimization reduces, when applied to PrC, the cost of commit processing associated with read-only participants, it fails to eliminate the `initiation` log records required by PrC for read-only transactions.

In this paper, we present a new read-only optimization called the *unsolicited update–vote* (UUV) that enhances performance over the traditional read-only optimization by eliminating *all* the logging activities from PrC for completely read-only transactions. UUV exploits the semantics of the underlying database management mechanisms to achieve this performance enhancement. The underlying assumptions in UUV is that each site employs a *conservative* and *avoiding cascading abort* scheduler, and *write-ahead logging*, for recovery [1].

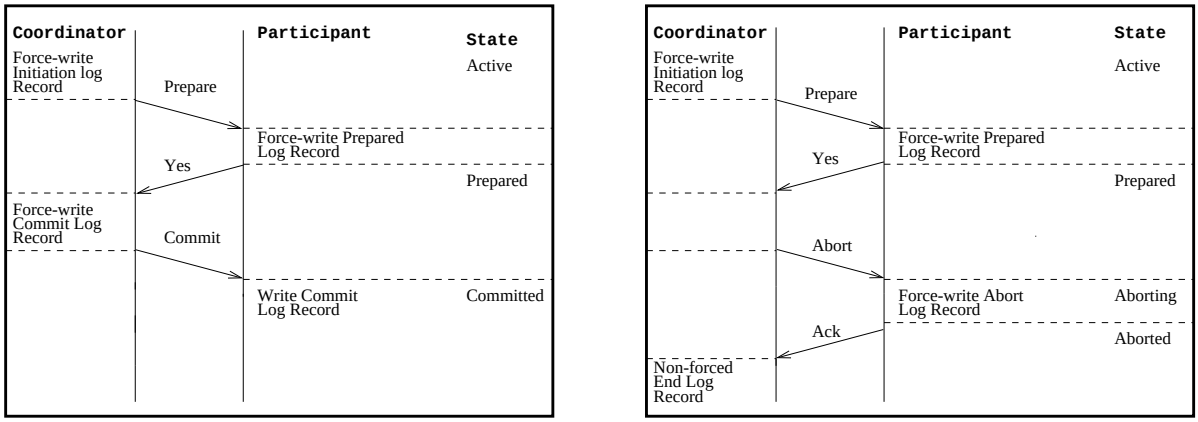
*Supported in part by N.S.F. under grants IRI-9210588 and IRI-95020091 and a Saudi Arabian graduate student scholarship.

2 THE PRESUMED COMMIT PROTOCOL

As in the basic 2PC, PrC consists of a *voting* phase and a *decision* phase as shown in Figure 1. However, PrC reduces the cost of committing a transaction by *not* requiring that the participants to force write a commit log record to a stable storage and to acknowledge a commit decision during the decision phase. PrC achieves this by making an *explicit* commit presumption about the outcome of transactions in the absence of information about the transactions. That is, after recovering from a failure, when a participant in the execution of a distributed transaction inquires the coordinator of the transaction (i.e., the site where the transaction has been initiated) about the status of the transaction, the coordinator responds with a commit decision if it has no recollection about the transaction.

In order to prevent a coordinator from wrongly interpreting missing information as a commitment, a coordinator, in PrC, has to force write an `initiation` log record before sending out *prepare to commit* messages to the participants during the voting phase. When a participant receives a *prepare to commit* message, it force writes a `prepared` log record before replying with a `Yes` vote. During the decision phase, to commit a transaction after *all* the participants have voted `Yes` (Figure 1 (a)), the coordinator first force writes a `commit` record to *logically* eliminate the `initiation` record of the transaction, then sends out the commit decision and finally discards any information about the transaction. When a participant receives the decision, it writes a non-forced `commit` record and commits the transaction without having to acknowledge the decision. To abort a transaction (Figure 1 (b)), on the other hand, the coordinator does not write the abort decision in its log. Instead, the coordinator, sends out the abort decision and waits for acknowledgments from the participants. When a participant receives the decision, it force writes an `abort` record and then acknowledges the decision. The coordinator writes a non-forced `end` record and forgets about the transaction once it receives acknowledgments from all the participants.

In some distributed database environments, distributed transactions are not processed based on the two-level transaction processing model assumed above. Instead, these environments adopt a multi-level transaction processing model, such as *the tree of processes model* in which a participant in the execution of a transaction is a process that can initiate other participant processes at its site or other sites resulting in a transaction execution tree [2]. To commit a distributed transaction in a multi-level transaction processing model using PrC, the participants need to be distinguished into



(a) Commit case **(b) Abort case**

Figure 1: The presumed commit protocol.

the coordinator (i.e., root participant), leaf participants and cascaded coordinators (i.e., non-root and non-leaf participants). The behavior of the coordinator and each leaf participant in the transaction execution tree remains the same as we have discussed above. However, each cascaded coordinator behaves as a leaf participant with respect to its direct ancestor and a coordinator with respect to its direct descendants. Thus, a cascaded coordinator has to force write an `initiation` record before propagating the *prepare to commit* message to its descendants.

2.1 PrC and Read-Only Transactions

When a participant that has executed only read operations on behalf of a transaction receives a *prepare to commit* message, it validates the transaction with respect to *serializability* and *recoverability* [1], as would be the case when the transaction had executed some update operations. If the transaction is validated, the participant responds with a `read-only` vote. Otherwise, the participant votes `No`. In either case, the participant releases all the resources held by the transaction once it votes without writing any log records. A `read-only` vote means that the transaction has read consistent data and the participant does not need to be involved in the second phase of the protocol because it does not matter whether the transaction is finally committed or aborted. This is the traditional *read-only* optimization [2] that allows each read-only participant to release all the resources held by a read-only transaction earlier than its update counterparts, without having to write any log records and be involved in the second phase of the protocol.

When PrC is combined with the read-only optimization, not knowing whether a transaction is read-only or not, a coordinator or a cascaded coordinator of a read-only transaction still has to force write an `initiation` record before sending out *prepare to commit* messages to its direct descendants. However, a coordinator and each cascaded coordinator complete the protocol by writing a non-forced `end` log record (as if the transaction was aborted) since it is cheaper than writing a forced `commit` record. For a partially read-only transaction (i.e., only some of the participants in its execution have executed only read operations), the coordinator and each cascaded coordinator behave, as in the case of an update transaction discussed earlier, considering only update participants in the second phase of the protocol.

From the above discussion, it is clear that the major overhead associated with PrC is the forcing of the *initiation* log records. Thus, in order to utilize PrC's advantages, there is a need to limit the adversary effects of the forcing of `initiation` records. In our approach, we eliminate the need for *initiation* records in the presence of read-only participants and read-only transactions which are the majority of transactions in any general database system.

3 THE UNSOLICITED UPDATE-VOTE OPTIMIZATION

The cost associated with read-only participants can be reduced further if the coordinator of a transaction knows, before the initiation of the commit protocol, which participants are read-only in the execution of the transaction. In this way, if all participants are read-only, the coordinator can avoid writing any log records. This is the essence of the *unsolicited update-vote* optimization (UUV).

Due to simplicity and ease of implementation, most commercial database management systems use the *strict two-phase locking* protocol (S2PL) [1] for concurrency control and *physical write-ahead logging* (WAL) for recovery [1]. Now, consider a distributed system in which all the sites employ S2PL. In such a distributed system, a transaction is guaranteed to be serializable and recoverable if all its operations have been executed and acknowledged (see [1] for proof).

To determine which participants are read-only without having to (explicitly) poll their votes (which would be the case in the traditional read-only optimization), UUV looks at the participants from the other perspective. That is, which participants are *update participants*. To determine which participants are update participants at run time, UUV utilizes piggybacking in the acknowledgment messages of the operations. Specifically, each transaction, when it starts, is marked as a read-only transaction. Once a participant executes the *first* update operation (which generates an undo/redo log record) on behalf of a transaction, it sends an *unsolicited update-vote*, as part of the operation's acknowledgment to the coordinator.

When a transaction finishes its execution and submits its final commit primitive, its coordinator determines which participants have sent *unsolicited update-votes* as part of their operations' acknowledgments. For each participant that has sent an *unsolicited update-vote*, the

	Coordinator			Participant		
	m	n	p	m	n	q
Commit	2	2	2	2	1	1
Abort	2	1	2	2	2	2
RO	2	1	1	0	0	1
UUV	0	0	1	0	0	0

Table 1: The cost associated with PrC.

coordinator knows that the participant is an update participant and declares it as such. Otherwise, the participant is declared read-only. At this point, the coordinator knows all read-only participants. Furthermore, it knows that the transaction is serializable and recoverable at each one of them. For a completely read-only transaction, the coordinator does not write an `initiation` log record (as it would have been the case if the traditional read-only optimization were used). Instead, the coordinator sends a `read-only` final message to each participant and forgets about the transaction. Once a participant receives a *read-only* message, it releases the resources held by the transaction and forgets the transaction.

For a partially read-only transaction, the coordinator sends a *read-only* message to each read-only participant without waiting for the `initiation` record to be in the stable storage. Thus, a read-only participant does not have to suffer from the cost associated with forcing the `initiation` record before it can release the resources held by the transaction.

In multi-level transaction trees and using UUV, neither the coordinator nor any cascaded coordinator in a read-only transaction tree writes any log records. Furthermore, each participant receives only a single message from its direct ancestor without sending back a reply. For a partially read-only transaction, only cascaded coordinators with update descendants (i.e., descendants that have sent an unsolicited update-vote) need to force write an `initiation` record.

Table 1 summarizes the cost associated with PrC for update as well as read-only, two-level transactions assuming a `Yes` vote from each update participant: m is the number of log records, n is the number of forced log writes, p is the number of messages sent from the coordinator to each participant and q is the number of messages sent back to the coordinator. The first row in the table captures the cost associated with a committing transaction while the second one captures the cost associated with an aborting transaction. The third row in the table summarizes the cost associated with a read-only transaction using the traditional read-only optimization. Recall that in a two-level transaction, there is a single forced `initiation` record for each read-only transaction and a single round of messages required by the voting phase. The traditional read-only optimization eliminates the decision phase. The last row in the table shows the cost associated with read-only transaction using UUV. Using UUV, all log records are eliminated as well as the voting phase. The only cost associated with UUV is the decision message sent to each participant by the coordinator.

3.1 Discussion

For completeness, we discuss in this section three other methods that can be used to determine read-only participants and point at their limitations.

1. By predeclaration in which each transaction indicates that it will perform only read operations [2].
2. By analyzing each submitted (high level) operation of each transaction.
3. By assuming that each participant knows when it has executed the last operation on behalf of a transaction, as it is the case in the *unsolicited vote* optimization [5]. In this case, a participant does not have to wait for the *prepare to commit* message. Instead, it sends its vote proclaiming itself as read-only in its own initiative once it recognizes that the transaction has no more operations to process.

The first method is very restrictive because transactions are written in an ad hoc fashion and their behavior cannot be determined a priori except in very special cases. The second method assumes that a coordinator is able to process and analyze high level operations as well as the return results from the participants. To realize this method requires expansion in the functionality of coordinators in current database management systems as opposed to UUV which requires the interpretation of a bit in an acknowledgment message. The third method assumes that the coordinator either submits to a participant all the operations at the same time, which is again a form of predeclaration, or indicates to the participant the last operation at the time the operation is submitted. The latter is possible if each transaction has knowledge about data distribution and indicates to the coordinator the last operation to be executed at a participant.

4 SUMMARY

In conclusion, UUV significantly enhances the performance of PrC making it an attractive alternative to PrA which is the choice of current commercial database standards, especially in the future highly reliable distributed database systems with transactions having high probability of being committed rather than being aborted. UUV can be also applied to a number of two-phase commit variants including *presumed abort* [2] and *linear* [1]. Furthermore, UUV facilitates the applicability of another widely advocated optimization, namely, the *last agent* [4]. Specifically, UUV provides a solution to the problem of selecting the last agent which requires knowledge that the selected node is an update participant and not a read-only one [3].

References

[1] Bernstein, P. A., V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[2] Mohan, C., B. Lindsay and R. Obermarck. Transaction Management in the R^* Distributed Data Base Management System. *ACM Transactions on Database Systems*, 11(4), Dec. 1986.

[3] Samaras, G. *Personal communications*, Pittsburgh, Aug. 1996.

[4] Samaras, G., K. Britton, A. Citron and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment. *Distributed and Parallel Databases*, 3(4):325–360, Oct. 1995.

[5] Stonebraker, M. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, May 1979.