The Case for Independent Updates*

Stefano Ceri† Politecnico di Milano
Maurice A.W. Houtsma‡ University of Twente
Arthur M. Keller§ Stanford University
Pierangela Samarati Universita' di Milano

Abstract

We present the case for allowing independent updates on replicated databases. In autonomous, heterogeneous, or large scale systems, using two-phase commit for updates may be infeasible. Instead, we propose that a site may perform updates independently. Sites that are available can receive these updates immediately. But sites that are unavailable, or otherwise do not participate in the update transaction, receive these updates later through propagation, rather than preventing the execution of the update transaction until sufficient sites can participate. Two or more sites come to agreement using an reconciliation procedure that uses reception vectors to determine how much of the history log should be transferred from one site to another. We also consider what events can initiate a reconciliation procedure.

1 Introduction

Many recent papers have studied the applicability of replicated databases, and many strategies have been developed to deal with updates in such an environment [1, 6, 7]. An overview of such strategies has been given by us in [4]; that analysis shows that read performance may increase enormously by allowing replicated data, but that update performance may also dramatically decrease, especially if two-phase commit is required to synchronize updates on the replicas. Not only is the software complexity of two-phase commit intrinsically high, but failures, such as network partition, will have a negative effect on availability. In the context of autonomous, heterogeneous systems, two-phase commit may not be feasible at all. For these reasons, research is focusing on delayed propagation of updates from one copy to another [5, 8].

To allow such a delayed propagation of updates, we introduce the notion of *independent updates*. We particularly address the problems that arise in an environment where database systems are relatively autonomous, and do not wish to depend on other sites being available while accepting updates. This does, of course, imply that global serializability is lost; but in many applications this seems perfectly acceptable according to several researchers [2]. It also reflects the common practice in applications written for commercial database systems, where the user has to indicate if whether support for global serializability is desired or whether the user can instead accept practical, potentially inconsistent solutions. Examples of applications where independent updates may be acceptable include airline reservation systems, inventory control systems, and banking applications; we definitely do not want our ticket order, or cash retrieval from an automatic teller machine (ATM), to be deferred until global availability of all connected systems is guaranteed.

Of course, when allowing independent updates, global constraints may no longer be fully supported. However, several proposals already exist to rewrite at least some global constraints into local ones; in this way important constraints may still be enforced [3].

2 Independent updates

We study situations where a number of database systems are integrated in a network, be it wide area, local area, or even closely coupled. We suppose that the database systems are relatively autonomous, and update applications cannot afford to wait for all sites to be available. Updates are therefore applied directly on the replicas which are momentarily available, and propagated to the other sites via asynchronous mechanisms to be described shortly (several options exists

*This work was performed as part of the Fauve project and started while some of the authors were visiting Stanford, and was partially supported by NSF grant IRI-9007753.

†Stefano Ceri is partially supported by the LOGIDATA+ project of CNR Italy.

‡The research of Maurice Houtsma has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences.

§Arthur M. Keller is partially supported by the Center for Integrated Systems at Stanford University. Send correspondence to him at Computer Science Dept., Stanford University, Stanford, CA 94305-2140, USA or `ark@db.stanford.edu`

here, to be described in the next section). We assume applications to be action-based. Transactions may commit locally, or, if they are using data from other sites they may use conventional two-phase commit to commit all subtransactions. The actions that are executed at a site are recorded in the so-called history log; this is a small extension to the ordinary log that is already kept for recovery purposes.

Each site keeps a small additional data structure called *reception vector*, which indicates the extent to which a site is informed about actions that have taken place at other sites. When a number of sites decide to reconcile (i.e., to recover to a common state that reflects all the actions that are known to the participating sites), they exchange history logs, rollback their database status as far as necessary, merge in timestamp order the newly known actions with the actions executed at their own site, and apply them to the rolled-back database state. In this way, all sites participating in a reconciliation achieve the same database state and the same history log.

Above we have sketched the reconciliation procedure; indeed, we have investigated several reconciliation procedures, reflecting different degrees of distribution of the reconciliation procedure itself. Several options exist, with different choices of communication and control (e.g., one central coordinator or distributed coordination). We have also given formal proofs that the reconciliation procedure briefly sketched here is indeed correct [5]. Also, protocols have been developed that precisely indicate what messages are required for the reconciliation procedure, what happens if failures appear during the reconciliation process, etc. Detection of partitions is a problem in itself, but can usually be entrusted to the underlying communication system. Alternatively, detection of partitioning can be integrated with the two-phase commit protocol. Finally, we have also investigated a vacuuming process for the history logs, which detects unnecessary parts of the history logs and removes them.

3 Applicability

The concept of independent updates, and the reconciliation mechanism sketched here, can be applied in several ways. Two important characteristics of a system supporting independent updates are degree of independence, and options for application of reconciliation mechanisms. Let us discuss these in some more detail.

The degree of independence that is allowed to the sites may vary along a gliding scale. An obvious option is not to allow independence at all, requiring all updates to execute in two-phase commit. A second option is to have updates executed in two-phase commit as long as no failures (such as network partition) occur, but allow independent updates once failures occur. A third option is to group sites in a number of more or less autonomous groups; within a group all updates execute in two-phase commit, but groups themselves behave in an autonomous way. A fourth option is to give full autonomy to all sites; updates will then, in general, not execute in two-phase commit but be applied locally.

In each of the cases described above, there are several 'levels of asynchrony.' Application of the reconciliation mechanism may vary, according to the application's need. Let us sketch a few options:

Time-based Reconciliation may be done at regular time-intervals. For instance, reconciliation could be done every night, every hour, etc.

Network-driven Reconciliation may be driven by the network; as soon as low-level primitives detect that two or more network partitions are able to communicate, reconciliation may commence. Alternatively, reconciliation may be deferred for some time period.

Operation-based Reconciliation may be done as soon as an operation (application) requires to operate on a consistent state; for instance, a noncommutative operation is requested in the context of systems accepting both commutative and noncommutative operations. An example of this is given by banking applications, where deposits and withdrawals are commutative operations, but posting interest requires a consistent global state.

User demand Reconciliation may be done as soon as a user explicitly demands it, for instance, for some infrequent but important applications that require a global consistent state.

References

[1] D. Agrawal and A. El Abbadi, "The tree quorum protocol: an efficient approach for managing replicated data," in *Proc. 16th Int. Conf. on VLDB*, Brisbane, Aug. 1990, pp. 243–254.

[2] D. Barbara and H. Garcia-Molina, "The case for controlled inconsistency in replicated

data," *Proc. of the Workshop on Management of Replicated Data*, Houston, TX, Nov. 1990.

[3] D. Barbara and H. Garcia-Molina, *The demarcation protocol: a technique for maintaining arithmetic constraints in distributed database systems*, CS-TR-320-91, Princeton University, April 1991.

[4] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "A Classification of Update Methods for Replicated Databases," STAN-CS-91-1932, Stanford University, October 1991.

[5] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "Achieving Incremental Consistency among Autonomous Replicated Databases," in *Proc. DS-5 (Semantics of Interoperable Database Systems)*, IFIP, Lorne, Australia, Nov. 1992.

[6] J.N. Gray and M. Anderton, "Distributed computer systems: four case studies," *Proc. of the IEEE*, Vol. 75, No. 5, May 1987.

[7] A. Kumar and A. Segev, "Optimizing voting-type algorithms for replicated data," in *Advances in Database Technology–EDBT'88*, J.W. Schmidt, S. Ceri, and M. Missikoff (Eds.), LNCS **303**, 1988, pp. 428–442.

[8] C. Pu and A. Leff, "Replica control in distributed systems: an asynchronous approach," *Proc. ACM SIGMOD 91,* Denver, CO, May 1991.