Approaches to Design of Real-Time Database Systems

Sang Hyuk Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
USA

Hyunchul Kang

Department of Computer Science
Chung Ang University
Seoul, Korea

ABSTRACT

Real-time database systems support applications which have severe operational constraints such as transaction deadlines and continued operation in the face of failures. In designing real-time database systems, there can be two approaches to meet those constraints. First approach is to redesign conventional database systems architecture to replace the bottleneck components. Second approach is to trade desired features, such as serializability, or exploit semantic information of transactions and data for high performance and reliability. In this paper, we discuss issues involved in these approaches, and present algorithms to solve problems in real-time database systems.

1. Introduction

As computers are becoming essential part of real-time systems, *real-time computing* is emerging as an important discipline in computer science and engineering [SHI87]. Since any kind of computing needs to access data, methods for designing and implementing database systems that satisfy the requirement of timing constraints in collecting, updating, and retrieving data play an important role in the success of real-time computing. In the recent workshops sponsored by the Office of Naval Research [ONR87, IBM88], developers of "real" real-time systems pointed to the need for basic research in database systems that satisfy timing requirements in processing shared data. Further evidence of its importance is the recent growth of research in this field and the announcements by some vendors of database products that include features achieving high availability and predictability [SON88].

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy not only the database consistency constraints but also the timing constraints associated with transactions. In other words, "time" is one of the key factors to be considered in real-time database systems. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on a tracking data of a missile must be processed within the given deadlines: otherwise, the information provided could be of little value. How to specify or determine deadlines is a relatively unexplored but difficult problem. It is even harder to guarantee that transactions can meet their deadlines due to dynamic nature of conflicts on shared data objects.

State-of-the-art database management systems are typically not used in real-time applications due to two inadequacies: poor performance and lack of predictability. Although conventional database systems provide efficient ways to store and retrieve data through user-friendly interface, they rely on secondary storage to store the database. In conventional database systems, transaction processing requires accessing database stored on the secondary storage; thus transaction response time is limited by disk access delays, which can be in the order of milliseconds. Still these databases are fast enough for traditional applications in which a response time of few seconds is often acceptable to human users. However, those systems may not be able to provide a response which is fast enough for high-performance real-time applications that require responses in the order of micro-seconds. Consequently, requirements and design objectives of real-time database systems widely differ from those of conventional database systems. A natural question is how should we redesign conventional database systems so that its performance and reliability can be acceptable for real-time applications.

One approach to achieve high performance is to replace bottleneck devices (e.g., a disk) by a high speed version. Second alternative is to tinker with features of conventional database systems. For example, by exploiting semantic information associated with transactions and data objects, we can use the notion of correctness different from the serializability of transaction execution. Third approach is the effective use of data replication to improve performance as well as

This work was supported in part by the Office of Naval Research under contract # N00014-88-K-0245, by the Department of Energy under contract # DEFG05-88-ER25063, and by the Federal Systems Division of IBM Corporation.

reliability. In this paper, we discuss issues and approaches to the design of high-performance real-time database systems, and present algorithms to solve problems in those approaches.

2. Main Memory Database Systems

2.1. Recovery Problem

Due to advancements in integrated circuits technology, the chip densities of semiconductor memories have been increased dramatically throughout the past decade, and the cost of main memory has been sharply decreased. The availability of large, relatively inexpensive main memories coupled with the demand for faster response time for real-time systems has brought a new perspective to database system designers: main memory databases in which the primary copies of all data reside permanently in main memory.

Elimination of disk access can contribute to substantial improvement in transaction response time. However, the migration of data from secondary storage to main memory requires a careful investigation of the components of traditional database systems, since they introduce some potential problems of their own. The most critical problem is associated with the recovery mechanism of the system, which must guarantee transaction atomicity and durability [SON86]. Since the least expensive form of main memory is volatile, any loss of electric power destroys all stored data. Consequently, a main memory database system still requires a disk to act as stable storage to provide a backup copy for the database.

A crash recovery mechanism in database systems essentially consists of two phases: preparation for the recovery by saving necessary information during normal operation of the system, and the coordination of actual recovery. The preparation for recovery is usually performed through checkpointing and logging which record database changes to a separate stable medium such as a disk. The disk activity associated with the preparation for recovery can be a bottleneck for traditional disk-oriented database systems. For a main memory database system with no need for disk activity for data access, this disk activity may have even worse effects on system performance, possibly nullifying the advantages of having all the data in main memory. Thus, constructing a recoverable main memory database system using volatile memory appears to be one of the most challenging issues to the system designer. One important requirement in constructing a checkpoint is that the interference of the checkpointing procedure with the transaction processing should be kept as small as possible. Consistency of the constructed checkpoint is also important since no undo operation would be necessary during recovery, reducing the recovery time.

On-line log compression is necessary to keep the log short to achieve a rapid restart. Compression can be used by any database system to improve restart time, but is essential for main memory database systems which may achieve very high transaction throughput. Most of the recovery techniques assume that a portion of memory can be made non-volatile by using batteries as a backup power supply [DEW84, HAG86, SAL86]. By exploiting this portion of non-volatile memory, log compression can be achieved effectively. It has been shown that the amount of such memory the system would need is not great [SAL86].

2.2. An Approach

In this section, we present a recovery scheme that is based on the techniques of non-interfering incremental checkpointing and log compression using non-volatile areas of memory. They are combined to increase the transaction throughput by reducing the number of log write operations, and minimizing the interference of checkpointing with the transaction processing. The scheme also provides a rapid restart of the database system from failures.

To make transactions durable in spite of system failures, the system must ensure that before a transaction commits, either all pages updated by the transaction are flushed into the backup copy, or a complete redo information is recorded in the log on the stable storage. The latter (saving redo information) achieves better performance than the former in general, since the former may involve more I/O activity. Our recovery scheme ensures the latter.

The checkpointing in this scheme is performed concurrently with transaction activity while constructing a transaction-consistent checkpoint on disk, without requiring the database quiesce to save a consistent state. This is particularly important in an environment where many update transactions generate a large number of updated pages between checkpoints, or in a real-time application where high availability of the database is desirable.

To construct a transaction-consistent checkpoint, the updates of a transaction must be either included in the checkpoint completely, or they must not be included at all. To achieve this, transactions are divided into two groups according to their relationships to the current checkpoint: *included-transactions* (INT) and *excluded-transactions* (EXT). The updates belonging to INT are included in the current checkpoint while those belonging to EXT are not included. Time-stamps are used to determine membership in INT and EXT. A transaction-consistent state of the database is always maintained on disk. At a checkpoint time, only the portion of the saved state which has been changed since the last checkpoint is replaced. When a checkpoint

completes, next one begins with only a negligible interruption of service to delete old versions that have been checkpointed.

A checkpointing begins by setting the checkpoint number (CN) as the current clock value. Transactions with the time-stamps smaller than the CN are the members of INT; others are the members of EXT. The transaction activity continues with the checkpointing process. When an EXT updates data objects updated by INT, new in-memory versions are created, leaving the old versions to be written to disk.

The checkpointing process waits until there is no active INT in the system. It then begins to write the pages updated by INT to disk, constructing a portion of the state to be replaced in the checkpoint. Let δ_1 be the set of pages that have been updated by INT since the last checkpoint. First, the members of δ_1 must be identified. This might be done with hardware assist by setting a dirty bit when the page is updated by INT. Then δ_1 must be written on disk without the loss of previous checkpoint in the case of a system failure. To achieve this, a variation of stable storage technique [LAM81] can be used in two steps. During the first step, δ_1 is written to a new disk file δ_2. When it is successfully completed, then δ_1 is written into the desired location in disk. If the system crashes while the checkpoint state is being replaced by δ_1, a consistent checkpointed state can be reconstructed from δ_2.

After δ_2 has been successfully created, a "begin checkpoint" record is written to the log, indicating the completion of the first step of checkpointing procedure. When δ_1 is written to their original locations on disk, making the checkpoint state on disk identical to the database state in memory as if all transaction activities are completed and no more transaction has been started after CN, an "end checkpoint" record is written to the log. It indicates that the checkpoint in disk is a complete transaction-consistent state of the database, and it includes all the updates of transactions active at time CN. The address of the checkpoint record and the value of CN are saved in non-volatile memory so that they can be used in locating the most recent checkpoint record in the log during recovery. When a checkpointing procedure completes, pages in δ_1, updated by EXT, are overwritten by the new versions, and their dirty bits are set. They are the initial members of δ_1 for the next checkpoint. Then the next checkpoint begins with the new CN.

The checkpointing approach presented here is different form that of [DEW84] in two important ways. First, it constructs a transaction-consistent state of the database instead of action-consistent state. Second, it does not require the database quiesce. It is also different from the "fuzzy dump" checkpointing in [HAG86], which constructs an inconsistent state of the database.

The non-volatile memory is used to hold an in-memory log so that the log writes to disk can be delayed. In-memory log can be considered as a reliable disk output queue for log data. When the log data must be written to disk since in-memory log buffer is full, many transactions that have entries in the log may now be completed. For aborted transactions, the whole entries in the log may be eliminated; for committed transactions, the undo part of log records may be eliminated. In addition, the log entries can be stored in disk in a more compact form. In the conventional approach, log entries for all transactions are intermixed in the log. A more efficient alternative is to maintain the log on a per transaction basis, and it is possible by using the non-volatile memory. Non-volatile memory also assists in reducing the recovery time by maintaining the information such as the log entry of the oldest update that refers to an uncheckpointed page, to help to determine the point in the log from which recovery should begin.

3. Distributed Real-Time Database Systems

Falling cost of hardware and advances in communication technology over the last decade have triggered considerable interests in distributed database systems. In such systems, data objects are spread over a collection of autonomous computer systems (called *sites)* that are connected via a communication network. Since the physical separation of sites ensures the independent failure modes of sites and limits the propagation of errors throughout the system, distributed database systems must be able to continue to operate correctly despite of component failures. However, as the size of a distributed system increases, so does the probability that one or more of its components will fail. Thus, distributed systems must be fault tolerant to component failures to achieve a desired level of reliability and availability. Asserting that the system will continue to operate correctly if less than a certain number of failure occurs is a guarantee independent of the reliability of the sites that make up the system. It is a measure of the fault tolerance supported by the system architecture, in contrast to fault tolerance achieved by using reliable components.

Distributed database systems are of great importance to real-time applications because they offer several advantages such as higher system availability, over a single-site database system. In many situations, real-time systems naturally include distributed database systems because they are inherently distributed. In distributed real-time database systems, the main issue is how to make several sites work collectively so that a high level of performance is obtained. In this section, we discuss approaches to the design of high-performance distributed real-time database systems.

3.1. Using Semantic Information

We can enhance the performance of distributed real-time database systems by exploiting semantic information of transactions. A read-only transaction is a typical example of the use of transaction semantics. A read-only transaction can be used to take a checkpoint of the database for recovering from subsequent failures, or to check the consistency of the database, or simply to retrieve the information from the database. Since read-only transactions are still transactions, they can be processed using the algorithms for arbitrary transactions. However, it is possible to use special processing algorithms for read-only transactions in order to improve efficiency, resulting in high performance. With this approach, the specialized transaction processing algorithm can take advantage of the semantic information that no data will be modified by the transaction.

Serializability has been accepted as the standard correctness criteria in database systems. It means that the concurrent execution of a group of transactions is equivalent to some serial execution of the same group of transactions. However, people actually developing large real-time systems are unwilling to pay the price for serializability, because predictability of response is severely compromised due to blocking or preemption. For read-only transactions, correctness requirements can be divided into two independent classes: the currency requirement and the consistency requirement.

The currency requirement specifies what update transactions should be reflected by the data read. There are several ways in which the currency requirement can be specified; we are interested in the following two:

(1) Fixed-time requirement: A read-only transaction T requires data as they existed at a given time t. This means that the data read by the transaction must reflect the modifications of all update transactions committed in the system before t.

(2) Latest-time requirement: A read-only transaction T requires data it reads reflect at least all update transactions committed before T is started, i.e., T requires most up-to-date data available.

The consistency requirement specifies the degree of consistency needed by read-only transactions. A read-only transaction may have one of the following requirements:

(1) Internal consistency: It only requires that the values read by each read-only transaction satisfy the invariants (consistency constraints) of the database.

(2) Weak consistency: It requires that the values read by each read-only transaction be the result of a serial execution of some subset of the update transactions committed. Weak consistency is at least as strong a requirement as internal consistency, because the result of a serial execution of update transactions always satisfies consistency constraints.

(3) Strong consistency: It requires that all update transactions together with all other read-only transactions that require strong consistency, must be serializable as a group. Strong consistency requirement is equivalent to serializability requirement for processing of arbitrary transactions.

We make a few comments concerning the currency and consistency requirements. First, it might seem that the internal consistency requirement is too weak to be useful. However, a read-only transaction with only internal consistency requirement is very simple and efficient to process, and at least one proposed algorithm [FIS82] does not satisfy any stronger consistency requirement. Second, it is easy to see that strong consistency is a stronger requirement than weak consistency, as shown by the following example. Suppose we have two update transactions, T_1 and T_2, two read-only transactions, T_3 and T_4, and two data objects, X and Y, stored at two sites A and B. Assume that the initial values of both X and Y were 0 before the execution of any transactions. Now consider the following execution sequence:

T_3 reads 0 from X at A.
T_1 writes 1 into X at A.
T_4 reads 1 from X at A.
T_4 reads 0 from Y at B.
T_2 writes 1 into Y at B.
T_3 reads 1 from Y at B.

The values read by T_3 are the result of a serial execution of $T_2<T_3<T_1$, while the values read by T_4 are the result of a serial execution of $T_1<T_4<T_2$. Both of them are valid serialization order, and thus, the execution is weakly consistent. However, there is no single serial execution of all four transactions, so the execution is not serializable. In other words, both read-only transactions see valid serialization orders of updates, but they see different orders.

Clearly, strong consistency is preferable to weak consistency. However, as in the case of internal consistency, it can be cheaper to ensure weak consistency than to ensure strong consistency. For the applications that can tolerate a weaker requirement, the potential performance gain could be significant.

Finally, one might wonder why fixed-time requirement is interesting, since most read-only transactions may require information about the latest database state. However, there are situations that the user is

interested in looking at the database as it existed at a given time. For an example of a fixed-time read-only transaction, consider the case of a general in the army making a decision by looking at the database showing the current position of the enemy. The general may be interested in looking at the position of the enemy of few hours ago or few days ago, in order to figure out the purpose of their moving. A read-only transaction of a given fixed-time will provide the general with the desired results.

3.2. Using Replication and Multiversion

An obvious approach to improve system availability is to keep replicated copies of critical data at multiple sites so that the system can access the data even if some of the copies are not available due to failures. In addition, replication can enhance performance by allowing transactions initiated at sites where the data are stored to be processed locally without incurring communication delays, and by distributing the workload of transactions to several sites where the subtasks of a transaction can be processed concurrently. These benefits of replication must be seen in the light of the additional cost and complexities introduced by replication control.

A major restriction of using replication is that replicated copies must behave like a single copy, i.e., *mutual consistency* of a replicated data must be preserved. By mutual consistency, we mean that all copies converge to the same value and would be identical if all update activities cease. The inherent communication delay between sites that store and maintain copies of a replicated data makes it impossible to ensure that all copies are identical at all times when updates are processed in the system [SON87].

Maintaining multiple versions of data objects is another approach to improve system responsiveness by increasing the degree of concurrency. The objective of using multiple versions is to reduce the conflict probability among transactions and the possibility of rejection of transactions by providing a succession of views of data objects. One of the reasons for rejecting transactions is that its operations cannot be serviced by the system. For example, a read operation has to be rejected if the value of data object it was supposed to read has already been overwritten by some other transactions. Such rejections can be avoided by keeping old versions of each data object so that an appropriate old value can be given to a tardy read operation. In a system with multiple versions of data, each write operation on a data object produces a new version instead of overwriting it. Hence, for each read operation, the system selects an appropriate version to read, enjoying the flexibility in controlling the order of read and write operations.

There are several problems that must be solved by an algorithm that uses multiple versions. For example, selection of old versions for a given read-only transaction must ensure the consistency of the state seen by the transaction. In addition, the need to save old versions for read-only transactions introduces a storage management problem, i.e., methods to determine which version is no longer needed so that it can be discarded. In the next section, we focus our attention on these problems.

4. A Resilient Synchronization Algorithm

In the algorithm to be presented below, we use the notion of *tokens* to support fault-tolerant distributed real-time database systems in increasing both the availability of data and the degree of concurrency, without incurring too much storage and processing overhead. Each data object has a predetermined number of tokens. Tokens are used to designate a read-write copy, and a token copy is a single version representing the latest value of the data object. The site which has a token copy of a data object is called a *token site*, with respect to the data object. The number of tokens for each data object can be used as a tuning parameter to adjust the robustness of the system.

Multiversions are stored and managed only at read-only copy sites. For read-only copies, each data object is a collection of consecutive versions. A read-only transaction does not necessarily read the latest committed version of a data object. The particular old version that a read-only transaction has to read is determined by the time-stamp of the read-only transaction (for the latest-time requirement) or by the given time (for the fixed-time requirement). The time-stamp is assigned to a read-only transaction when it begins, while the time-stamp for an update transaction is determined as it commits. When a read-only transaction with time-stamp T attempts to read a data object, the version of the data object with the largest time-stamp less than T is selected as the value to be returned by the read operation.

We assume that update transactions use two-phase locking [ESW76], with exclusive locks used for write operations, and shared locks for read operations. Lock requests are made only to token copies, and there is no locks associated with read-only copies. In addition, update transactions use the two-phase commit protocol and stable storage [LAM81] to achieve fault-tolerance to site failures. When a new version is created, it is created at all copy sites, including read-only copy site. However, any new versions are not accessible to other transactions until they are finalized through the two-phase commit protocol. Upon receiving the commit message from the coordinator, new versions of data objects created by the transaction replace

transactions. Recall that each data object keeps track of the read-only transactions that have accessed the data object, along with a lower bound on the time-stamp chosen by each transaction. Data objects can use the following rule to decide which versions to keep and which to discard.

Rule for retention:

A version with time-stamp TS must be retained if

(1) there is no version with time-stamp greater than TS (i.e., current version), or

(2) there is a version with time-stamp TS' > TS, and there is an active read-only transaction whose time-stamp 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, the read-only site effectively agrees to retain the current version and any later versions, until it knows which of those versions is needed by the read-only transaction. When the read-only site finds out the time-stamp 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 time-stamp 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 time-stamp to all read-only sites when it chooses the time-stamp.

The version management described above is effective at minimizing the amount of storage needed for versions. For example, unlike the "version pool" scheme in [CHA85], 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 read-only site 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 time-stamp, a process that requires communicating with all data objects it needs to access.

Because the time-stamp for a fixed-time read-only transaction is determined by the user, the number of versions that needs to be retained to process fixed-time read-only transactions cannot be bounded as in the case for latest-time read-only transactions. In order to process all the potential fixed-time read-only transactions, the system must maintain all the versions created up to the present, which may require huge amount of storage. There are several alternatives to keep a history instead of saving all the versions created for each data object. One of the simplest and efficient alternative would be to keep a log of all the update transactions. Fixed-time read-only transactions can be processed by examining the log in reverse chronological order until the desired version of the data object can be reconstructed. Since fixed-time read-only transactions must examine the log, their execution depends on the availability of the log, and their execution speed would be slower than that of latest-time read-only transactions. One important advantage of the transaction log mechanism is that in many systems the log is required anyway for crash recovery. Thus, in these systems, keeping the log for fixed-time read-only transactions represents no real overhead.

Read-only transactions are never aborted and their response time is reduced because they do not need to go through two-phase commit protocol. Furthermore, access requests from read-only transactions do not require to access token copies, and hence no blocking is introduced by update transactions. This results in further reduction of response time of read-only transactions. In general, the number of aborts and average transaction response time for a given set of transactions depend on system parameters and read-set/write-set of transactions, making analytical evaluation complicated. A prototyping tool for experimenting distributed database systems is being developed at the University of Virginia [SON88b], and a quantitative evaluation of the proposed algorithm will be performed and reported in a separate paper.

5. Concluding Remarks

A real-time database system supports applications which require severe performance constraints such as fast response time and continued operation in case of subsystem failures. Real-time database systems are still in the state of infancy, and issues and alternatives in their design are not very well explored. In this paper, we have addressed two approaches to design of real-time database systems. The first approach is aiming to reduce I/O delays by having main memory resident database. The second approach calls for exploiting semantic information and data replication especially in distributed environments.

Replication is a key factor in making distributed database systems more reliable than single-site systems. The algorithm presented in this paper exploits the multiple versions of data objects and the semantic information of read-only transactions in achieving improved system performance. Multiple versions are maintained only at the read-only copy sites, hence the storage requirement is reduced in comparison to other multiversion mechanisms [REE83, CHA85].

High performance and high reliability do not come for free. There is a cost associated with each approach: storage requirement and complicated control in synchronization and recovery. For appropriate management of multiple versions, some communication cost is inevitable to inform data objects about activities of read-only transactions. There is also a cost associated with maintaining the data structures for keeping track of versions and time-stamps. In many real-time applications of database systems, however, the cost of those approaches is justifiable. Further work is clearly needed to develop alternative approaches for recovery in main memory database systems and for exploiting semantic information of transactions and data objects, and to investigate performance of different approaches.

REFERENCES

BHA86 Bhargava, B., Ruan, Z., Site Recovery in Replicated Distributed Database Systems, *Proc. 6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, pp 621-627.

BIT86 Bitton, D., The Effect of Large Main Memory on Database Systems, *Proc. ACM SIGMOD Conference on Management of Data*, May 1986, pp 337-339.

CHA85 Chan, A. and Gray, R., Implementing Distributed Read-Only Transactions, *IEEE Trans. on Software Engineering*, Feb. 1985, pp 205-212.

DEW84 DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D., Implementation Techniques for Main Memory Database Systems, *Proc. ACM SIGMOD Coference on Management of Data*, 1984, pp 1-8.

ESW76 Eswaran, K.P. et al, The Notion of Consistency and Predicate Locks in a Database System, *Communications of ACM 19*, Nov. 1976, pp 624-633.

FIS82 Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, *IEEE Trans. on Software Engineering*, May 1982, pp 198-202.

HAG86 Hagmann, R., A Crash Recovery Scheme for a Memeory-Resident Database System, *IEEE Trans. on Computer Systems*, Sept. 1986, pp 839-843.

IBM88 *IBM Real-Time Systems Requirements and Issues Workshop*, Manassas, Virginia, April 1988.

LAM81 Lampson, B., Atomic Transactions, Distributed Systems: Architecture and Implementation, *Lecture Notes in Computer Science*, Vol. 105, Springer-Verlag, 1981, pp 246-265.

ONR87 *ONR Real-Time Computing Initiative Workshop*, San Jose, California, Dec. 1987.

REE83 Reed, D., Implementing Atomic Actions on Decentralized Data, *ACM Trans. on Computer Systems*, Feb. 1983, pp 3-23.

SAL86 Salem, K. and Garcia-Molina, H., Crash Recovery Mechanisms for Main Storage Database Systems, *Technical Report CS-TR-034-86*, Department of Computer Science, Princeton University, April 1986.

SHI87 Shin, K. G., Introduction to the Special Issue on Real-Time Systems, *IEEE Trans. on Computers*, Aug. 1987, 901-902.

SKE81 Skeen, D., Nonblocking Commit Protocols, *Proc. ACM SIGMOD Conference on Management of Data*, 1981, pp 133-142.

SON86 Son, S. H. and Agrawala, A., An Algorithm for Database Reconstruction in Distributed Environments, *Proc. 6th International Conference on Distributed Computing Systems*, May 1986, pp 532-539.

SON87 Son, S. H., Synchronization of Replicated Data in Distributed Systems, *Information Systems 12*, 2, June 1987, 191-202.

SON88 Son, S. H., Real-Time Database Systems: Issues and Approaches, *ACM SIGMOD Record*, Special Issue on Real-Time Database Systems, Vol. 17, No. 1, March 1988.

SON88b Son, S. H., A Message-Based Approach to Distributed Database Prototyping, *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, pp 71-74.