JUNE 1985 VOL. 8 NO. 2

a quarterly bulletin
of the IEEE computer society
technical committee
on

Database Engineering

Contents

Special Issue on Concurrency and Recovery

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Memoership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the Editor

This issue of *Database Engineering* is devoted to **Concurrency and Recovery**. Thirteen papers authored by academic and industrial researchers give an overview of research results and implementation experiences relating to transaction management in centralized and distributed data base management systems. The papers cover the entire spectrum of commercial products, research prototypes, and unimplemented ideas.

The widespread adoption of the data base approach to managing data created an important requirement - the support for concurrent access to data and for recovery from different types of failures. While this requirement was recognized many years ago in the case of the mainframe data base systems, only now are the designers and implementors of the workstation/personal computer data base management systems beginning to realize it. Even in the case of the mainframe systems, the increasing demand for higher throughput has necessitated reductions in the overheads due to concurrency control and recovery mechanisms.

As the data base users have become more sophisticated they are starting to demand support for different kinds of objects and specialized operations on such objects. The transaction concept is gaining popularity in non-data base applications also. The papers of this issue reflect these trends.

The first two papers deal with commercial products: IBM's IMS/Fast Path and Tandem's Transaction Monitoring Facility (TMF). The next four papers discuss transaction management aspects of research prototypes still under implementation. The featured prototypes are Carnegie-Mellon University's TABS, Massachusetts Institute of Technology's ARGUS, Computer Corporation of America's DDM, and highly available and real-time data base systems, and IBM San Jose Research Laboratory's R*. Performance evaluation results relating to implemented and unimplemented concurrency control and recovery algorithms are discussed in the next three papers. The final set of papers present a number of novel research ideas.

I wish to thank all the authors for their contributions. I was pleasantly surprised at the overwhelming response to my invitations. I had to be strict about the page limit. One set of authors alone were permitted to exceed the 8-page limit since their results were in fact to have been presented in two separate papers, but to provide more cohesion the results were presented in a single paper.

C. Mohan

C. Mohan

May 1985

VARIETIES OF CONCURRENCY CONTROL IN IMS/VS FAST PATH

Dieter Gawlick
Amdahl Corporation
1250 E. Arques Avenue, Sunnyvale, CA 94088-3470

David Kinkade
Tandem Computers, Inc.
19333 Vallco Parkway, Cupertino, CA 95014

Abstract

Fast Path supports a variety of methods of concurrency control for a variety of different uses of data. This arose naturally out of efforts to meet the typical needs of large on-line transaction processing businesses. The elements of "optimistic locking" apply to one type of data, and "group commit" applies to all types of data. Fast Path development experience supports the idea that the use of a variety of methods of concurrency control by a single transaction is reasonable and is not as difficult as might be expected, at least when there is fast access to shared memory.

1. Introduction

IBM introduced the Fast Path feature of IMS/VS in 1976, to support performance-critical on-line transaction processing.

Originally, Fast Path transactions shared the communication network and the system journal with standard IMS transactions, but little else was shared. Fast Path transactions could not access standard IMS/VS data bases and standard IMS/VS transactions could not access Fast Path data bases. Now Fast Path is integrated into IMS/VS, and a single transaction can access both kinds of data bases with consistency and atomicity across all of them. This integration in itself meant that the different methods of concurrency control common to Fast Path and to standard IMS/VS had to be supported within a single environment. In addition to this variety, Fast Path itself has always supported a variety of methods of concurrency control, although this was less apparent in 1976 than it is today, because the methods were simpler then.

Fast Path supports three kinds of data, each with its own methods of concurrency control. Two of these methods were novel when developed:

- Very active data items ("hot spots") are supported by Main Storage Data Bases (MSDBs). Very active data can include branch summaries, total cash on hand, accounting information (the number of transactions of a particular type processed), or any other kinds of information likely to cause bottlenecks because of locking conflicts between simultaneous transactions. Also, MSDBs are useful for terminal-related data, data likely to be accessed by a high percentage of all transactions from a particular terminal.
- Less frequently accessed data is stored in the main portions of Data Entry Data Bases (DEDBs). The main portion of a DEDB is hierarchical, with randomized (hashed) access to the roots. The significant improvement over standard IMS/VS hierarchical randomized data bases is in data availability, not in concurrency control. Because the amount of

data is very large, access conflicts are rare and straightforward methods of concurrency control are adequate.

- Application journal data (application historical data) is also supported by DEDBs, in special "sequential dependent" portions of DEDBs. The implementation of sequential dependents avoids a bottleneck that could otherwise result from simultaneous transactions adding records at the end of an application journal.

Some Fast Path innovations (such as group commit and the elimination of "before" images from the system journal) are general, not for the benefit of particular types of data. Still, we will start by considering the special handling for highly active data "hot spots" and for application journal data.

2. Main-Storage Data Bases (MSDBs)

What innovations help with "hot spots"? Consider, for example, an ultra-hot spot: a counter updated by every transaction. To achieve a high transaction rate, we want a transaction to be able to access the counter without waiting for any other transaction. We also want to guarantee the integrity of the counter, consistency with other parts of the data base, etcetera. If we are to have high performance ($>$ 100 transactions per second), then there are three things we cannot afford to do:

a. We cannot afford to lock the record and keep other transactions from accessing it until our transaction in complete.

b. We cannot afford to read the record from disk and write the record to disk.

c. We cannot afford to wait for the previous transaction's update log records to be written to disk (this is not unique to MSDBs, and is handled by group commit, to be described later).

To avoid the bottleneck that would occur from all transactions needing exclusive control of the counter, we must be a little tricky. In our example, the transaction wants to add 1 to the counter. The transaction doesn't care about the old or new values of the counter, except that the old value of the counter must be less than the maximum value the counter can hold. Accordingly, the transaction program does not "read" and "write" the counter value. Instead, it uses two new operations:

a) VERIFY counter $<$ *(maximum counter value)*
b) CHANGE counter $+$ *1*

The VERIFY operation checks a field and makes sure the field is $<$, $<=$, $=$, $>=$, or $>$ a specified value. Multiple VERIFY operations can be used to specify upper and lower limits.

The CHANGE operation updates a field by adding or subtracting a specified value, or by setting a field to a specified value. It could have been extended to include multiplication and division if these had seemed useful.

Fast Path does VERIFY tests both at the time of the VERIFY operation and again during commit time. The first test gives the application program a chance to do something different if the value is already out of range. The second test, the "real" test, is performed as part of a high-speed replay of the VERIFYs and CHANGEs for this transaction. During commit processing the data is locked, but only for a few instructions.

CHANGE operations are performed only during commit processing.

Besides VERIFY and CHANGE operations, Fast Path also allows the more usual operations (reading and writing records) against MSDB data. This is a complication for the developers, but not relevant to this paper.

The VERIFY and CHANGE operations implemented for MSDBs, publicly introduced in 1976, contain the essential elements of "optimistic locking," although the term was then unknown. This is readily apparent if you consider the use of "VERIFY = ."

3. Data-Entry Data Bases (DEDBs)

DEDBs are for large data bases. Except for sequential dependents, we are willing to wait for data to be read from disk (Fast Path doesn't even use buffer look-aside, except within a transaction). We are willing to write each updated record to disk (multiple read-write heads can be assumed), and we are willing for one transaction to wait for the other if both transactions want to access the same record.

Since many readers of this paper will be unfamiliar with Fast Path, it seems reasonable to mention the innovations that improved availability, even though this is not relevant to the main topic of the paper. It will at least make you aware of DEDB "AREAs." Originally there were two big availability improvements over standard IMS/VS:

1. Fast Path allows you to divide a DEDB into up to 240 different partitions called "AREAs." Each AREA is a separate file, and can be taken off-line while leaving the rest of the data base on-line. The internal data structures are such that pointers never point from one AREA to another. (The use of multiple AREAs also allows the construction of very large databases. The limit is 960 gigabytes.)

2. Since Fast Path has no way of telling which transactions will access which AREAs, it lets a transaction try to access data in an unavailable AREA. Fast Path introduced a new IMS/VS status code to tell the application program that the data is unavailable and let the application program take alternate action. Until Fast Path, IMS/VS refused to run a transaction that might try to access unavailable data.

In 1984 another data availability enhancement was provided: data replication, the ability to have from 0 to 7 copies of an AREA. Zero copies, of course, amounts to having the AREA off-line. The data replication facilities allow a new copy to be added and brought into use or allow a copy to be removed and taken away for safe storage without interrupting access to the AREA.

4. DEDB Sequential Dependents

Sequential dependents allow the construction of an application journal. They are to contain historical information. For example, a root record with your account number as its key may show how much money you currently have in your account, but your bank must also keep a history of how much money you used to have, and what happened to change your account balance. The bank would like to be able to get this information without having to scan through all updates to all accounts. At other times the bank would like to be able to scan all the updates for a particular period of time very rapidly; this is typically part of a daily batch run.

These considerations lead to the following strategy:

1. Sequential dependents, once written, can neither be modified nor individually deleted. They are eventually discarded as obsolete.
2. Sequential dependents are stored in chronological order according to time of commit processing (time of completion of the transaction).
3. To support retrieval of sequential dependents related to some particular thing (such as your bank account), without having to scan a lot of unrelated data, sequential dependents are chained in LIFO order from a chain anchor in the root record. This is very efficient; it can be done without accessing previously written sequential dependents.

There are many ways this strategy could have been implemented. The method actually chosen imposes many restrictions. At the time, these restrictions were appropriate, since the general strategy had not yet been tested by use. Here are the rules:

1. A sequential dependent segment (record) must be added to the DEDB as a child of one and only one non-sequential DEDB segment. Only one sequential-dependent segment type is allowed, and this segment type must be a child of the root segment.
2. Once inserted, sequential dependent segments can be retrieved as children of their root segment. Retrieved this way, they are presented in LIFO order. For example, the root segment for your bank account would point to the most recent update for your account, the most recent update record would point to the update before that, and so on.
3. Sequential dependents can also be scanned sequentially, for high-speed bulk processing. In this case, all the sequential dependents for a chosen period of time are presented in chronological order by commit time.
4. Sequential dependent segments are stored in a portion of disk storage reserved for this purpose at the end of each DEDB AREA. This portion of disk storage is used and reused as a large circular buffer.
5. There is no facility for deleting individual sequential dependents. Instead, there is a pointer to the oldest sequential dependent not yet deleted. When this pointer is moved forward, all previous records are logically deleted, and the space they occupied is made available for reuse. Because the space is circularly reused, the most significant bits of all pointers are a "cycle" counter: the first use of the disk storage is cycle 1, the second use is cycle 2, etcetera. Accordingly, a later record always has a larger pointer.

For sequential dependents, the "hot spot" is not a data item; it is the next piece of unused space. To handle this "hot spot," a Sequential Dependent insertion goes through three stages:

1. When the application program tells Fast Path to add a new sequential dependent, the data provided is NOT assigned a location on disk; this would not be a good idea because other transactions could insert sequential dependents and complete before this transaction completes. Instead, the data provided is placed in a holding area, the same as for MSDB VERIFY or CHANGE data. Also, the added data is chained LIFO from the main-storage copy of the record it is a child of. The pointer from the root is

recognizable as a main-storage pointer (not a disk pointer) by the fact that its most significant bits (the "cycle" number) are 0.

2. During commit processing, Fast Path allocates the available space to the sequential dependents. After allocating the space, Fast Path revisits the root and all sequential dependents added by this transaction, converting main-storage pointers into disk storage pointers to the newly allocated locations. Also, Fast Path copies the data to the current buffer being filled with sequential dependents for this DEDB AREA.
3. The buffer the sequential dependents are copied to at commit time isn't written to disk until after the buffer is filled and all the commit records for all the transactions that added data to the buffer have been written to the system journal. This delay is not a problem; nobody has to wait for the buffer to be written to disk unless there is a shortage of buffers.

5. Resource Control

The design of Fast Path distinguishes between contention control (making sure this transaction and another transaction aren't incorrectly accessing the same data) and contention resolution (deadlock detection). Contention control is constantly needed. Accordingly, it is handled within Fast Path, using a variety of efficient mechanisms. Contention resolution is needed only when a conflict has actually occurred, and is much more expensive. It uses the IMS/VS Resource Lock Manager (IRLM). This is necessary because a deadlock could involve both Fast Path and standard IMS/VS (non-Fast Path) resources.

An historical note: Since the first release of Fast Path did not allow a single transaction to access data from both Fast Path and IMS/VS data bases, a common contention resolution mechanism was not required, and Fast Path originally had its own deadlock detection. At that time, contention control and contention resolution were not separated the way they are now.

Contention control in Fast Path is optimized towards processing simple cases with a minimum of CPU time. To do this, Fast Path uses hashing in the following way:

1. Fast Path creates a large number of hash anchors for each data base or DEDB AREA (file). The number of hash anchors is large compared to the number of resources likely to be in use at one time.
2. When a transaction requests a particular resource (requests a lock), one of the hash anchors is selected as a pseudorandom function of the resource name, in such a way that a single resource always hashes to the same hash anchor. The hash anchor is used to anchor a chain of all lock requests for resources that hash to the anchor point.

Since the number of hash anchors is normally much larger than the number of resources in use, most requests encounter unused hash anchors. Therefore, since there can be no resource conflict without a hash anchor conflict, the resource can usually be locked quickly. If we ignore the CPU time needed to create the resource information, such a lock/unlock pair typically costs about 20 instructions.

If the hash anchor is already in use, then there must be a check to see whether or not there is actually a conflict. These checks have to be made in such a way that only one process is checking and/or manipulating the chain off one anchor. This requires additional synchronization, but it still costs less than 100 instructions unless a conflict is found to actually exist.

If a conflict is found to actually exist, then a request must be forwarded to the contention resolution mechanism (the IRLM).

Fast Path could do some deadlock detection. However, neither Fast Path nor standard IMS/VS could detect all the deadlocks without information from the other, since cycles might involve both kinds of resources. With a full resource control system available, the natural choice is to use it for all deadlock detection. However, there are two challenging technical problems:

1. To avoid consuming an excessive amount of CPU time, Fast Path must avoid unnecessary interactions with the common resource manager (IRLM).

   The common resource manager should only know about those resources that are involved in a conflict. However. a conflict is only recognized when a resource is requested for the second time. Therefore, at the time a conflict is detected, two requests have to be sent to the common resource manager: the request for the current owner, which has to be granted; and a request for the competitor. This leads to the second problem.

2. When a conflict does arise, Fast Path must share all relevant information with the common resource manager.

   The difficulty is related to the fact that resources have to be obtained on behalf of other processes. That is, the IRLM has to give the lock to the current owner, not to the process that detected the conflict. Fast Path's solution to this problem depends on the details of IRLM, and is too complicated to describe here.

With all the difficulties to be solved in the case of actual conflict resolution, 2,000 instructions is a reasonable number for the cost of conflict resolution using the IRLM.

Although actual results depend on the characteristics of the data bases and application programs, in a typical case 98% of the requests might encounter unused hash anchors, and 98% of the other requests might be instances of two different resources hashing to the same hash anchor. Doing the arithmetic, we see that contention resolution is needed only 0.04% of the time, and a typical Fast Path lock/unlock pair winds up still costing only about 20 instructions. Again, this is not counting the instructions required to create the resource information.

For Fast Path, the type of resource information (the name, owner, chain fields, etc.) depends on the type of data, but creating the resource information typically takes about 30 instructions. Locking should not be charged with all this cost, since the resource information is also used for other purposes, including buffer look-aside within a transaction and group commit.

The logic just described works only as long as no data bases are shared between different Fast Path systems. If DEDBs are shared between systems, then all locking has to be done by the common resource manager (IRLM), and locking becomes much more expensive.

The lock granularity in Fast Path varies depending on the data base structure and the use of the data.

- Record granularity is always used for MSDBs.
- Page granularity is used for on-line non-sequential-dependent processing and sequential-dependent retrieval.
- AREA granularity is used for sequential-dependent insertion.
- Page clusters (named "Units of Work") are used for on-line utilities.

6. Commit Processing

Fast Path commit processing satisfies the following needs:

- To work together with standard IMS/VS data base handling, and to make a common GO/NOGO (commit/abort) decision.
- To do system journaling in a way that minimizes the amount of system journal data and that allows asynchronous checkpoints of data bases.
- To minimize the amount of time that critical resources such as MSDB records and/or application journal tails are held.

The commit process is structured in the following way:

1. Do phase one processing for standard IMS/VS data bases.
2. Get all locks that are required to do phase one of Fast Path. These locks represent the MSDB records that have to be verified and/or modified and the journal tails of DEDB AREAs for which inserts are pending. Locks are acquired in a specific order, minimizing the potential for deadlocks.
3. Journal Fast Path data base modifications. This includes building after images and copying them to the next available space in the system journal's main storage buffers; it does NOT include waiting for system journal records to be written to external media. All the Fast Path data base updates for a single transaction are grouped together. The use of after images allows easy implementation of asynchronous checkpoints.
4. Do phase two of commit processing. This includes updating MSDBs and DEDB application journal tails, and unlocking the resources related to MSDBs and application journals.
5. Do phase two processing for standard IMS/VS data bases.

Even after all these steps, the results of the transaction cannot yet be externalized (data base changes cannot yet go to DASD and the response cannot go to the terminal) until after the system journal records have been written. In step three, Fast Path does not wait for system journal records to be written; this is to reduce the time between step two and step four, when critical resources are held. This technique reduces the period for which critical resources are held to the millisecond range, allowing extremely high concurrency for these resources.

Even after step five, Fast Path does not force the system journal records out to external media. In this way, Fast Path attempts to minimize CPU time and disk/tape space for the system journal. The system journal records will eventually be forced out by a buffer becoming full, a time-limit expiring, or standard IMS/VS processing. Since resource ownership is recorded on an operating-system region (job) basis, Fast Path switches resource ownership to a dummy region. This lets the region start work on a new transaction.

When a system journal buffer is physically written out, it is likely to contain commit records for more than one transaction. Accordingly, a group of transactions all become committed at one time. This was named "Group Commit" long after the development of IMS/VS Fast Path.

7. Acknowledgments

We thank and acknowledge the work of the following people: Keith Huff, Wally Iimura, Cliff Mellow, Tom Rankin, and Gerhard Schweikert, who all contributed greatly to the Fast Path implementation and design; Jim Gray, who helped clarify and improve the design; Don Hyde and George Vogel, who managed the project.

8. Bibliography

Although this paper is based on personal experience and private discussions, the following literature would be helpful to anyone interested in further exploration of these ideas:

Anon., et al., *A Measure of Transaction Processing Power,* Tandem Technical Report 85.1, also (a shorter version) in: Datamation, April, 1985.

Galtieri, C. A., *Architecture for a Consistent Decentralized System,* IBM Research Report RJ2846, 1980.

Gawlick, D., *Processing "Hot Spots" in High Performance Systems,* Proceedings of COMPCON '85, 1985.

IBM Corp., *IMS/VS Version 1 Data Base Administration Guide,* Form No. SH20-9025-9, 1984.

IBM Corp., *IMS/VS Version 1 Application Programming,* Form No. SH20-9026-9, 1984.

Lampson, B., *Hints for Computer System Design,* in: Operating Systems Review, Volume 17, Number 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, pp. 33-48, 1983.

Reuter, A., *Concurrency on High-Traffic Data Elements,* in: Proceedings of the ACM Symposium on Principles of Database Systems (SIGACT, SIGMOD), pp. 83-92, 1982.

Strickland, J., Uhrowczik, P., and Watts, V., *IMS/VS: An Evolving System,* in: IBM Systems Journal, Vol. 21, No. 4, 1982.

Transaction Monitoring Facility (TMF)

Pat Helland
Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014
(408) 725-6000

ABSTRACT

Tandem's Transaction Monitoring Facility (TMF) is a fully functional log-based recovery and concurrency manager. In use at customer sites since 1981, TMF runs in a distributed environment which supports modular expandability. This paper outlines the existing facility, as well as recent and current research and development. This research has concentrated on the areas of high transaction rates and high availability.

I. The Current Product

TMF as a concurrency or recovery mechanism is mainly unusual in two respects:

-- It is a distributed system.

-- It is part of the underlying operating system. As such, TMF can provide protection for any disc file.

A. Environment

A Tandem node consists of two to sixteen processors, each with an independant memory, interconnected by dual 13 megabit-per-second interprocessor busses. This, combined with the message-based GUARDIAN operating system [Bart78], provides an environment that may be expanded modularly. Up to 255 of these nodes may be connected in a long-haul network. The network automatically provides route-through, reroute in the event of a link failure, and best-path selection. Groups of two to fourteen of these 255 nodes may be connected via FOX, Tandem's high-speed fiber-optics Local Area Network. Messages between nodes, via FOX, typically take ten percent longer than messages between processors in the same node.

B. Intra-Node TMF

TMF is a log-based recovery manager that uses the "Write Ahead Log" protocol [Gray 78].

Each disc volume (one physical disc and an optional mirror) is controlled by its own process, called a disc process. The disc process actually runs as two processes (a "process pair" [Bart78]) in the two processors physically connected to the disc controller. In addition to servicing record-level requests, its responsibilities include the generation of logical before and after images of the record-level requests and the maintenance of the B-trees for keyed access. Locks are maintained by each disc process.

At commit time, a two-phased protocol is used to coordinate the flushing of the disc processes' log buffers and the writing of the commit record to the log. This consists of the following steps:

1) Each processor is notified that the transaction is about to commit. The processor notifies each disc process running in it that has worked on the transaction. Each disc process sends its log buffer to the log. When all disc processes in the processor have sent their logs, the processor notifies a preselected coordinator processor that it has flushed.

2) When all processors have flushed, the coordinator processor writes the commit record to disc. All processors are notified that the transaction has completed. Each processor notifies any concerned disc processes, which release any locks held on the transaction's behalf.

TMF uses the log to provide transaction BACKOUT to cope with failures of the application process. Node failures are dealt with by AUTOROLLBACK, which uses the log to undo uncommitted transactions and redo recently committed transactions. AUTOROLLBACK can also be used to recover from disruptions of service to a single disc volume. In the event of a media failure, ROLLFORWARD uses the current log, tape archives of the old log, and an old copy of the database (called ONLINE DUMP) to reconstruct a consistent, up-to-date database.

Transaction protection may be provided for any disc file using TMF. When a process has performed a BEGINTRANSACTION, any request messages it issues automatically include a TRANSID that identifies the process's current transaction. By designating a file as an AUDITED file, all updates must be performed using a TRANSID and the file may be recovered to a consistent transaction boundary.

C. Inter-Node (Network) TMF

When a request that is marked with a transaction identifier leaves one node for another node, the two nodes establish the transaction as a network transaction. This logic is centralized into one process at each of the nodes. At transaction commit, a standard two-phased commit protocol is superimposed onto the intra-node (local) commit.

II. Performance Improvements

As the power and speed of Tandem processors has increased, a strong trend has emerged to use the machine for larger applications. The modular expandability that is inherent in the Tandem architecture supports this trend. Customers want and need a system that they can easily expand by purchasing additional hardware. This upward growth must have a very high ceiling to support very large systems. Requirements for 100 transactions per second are common today with growth to 1000 transactions per second expected in the near future.

Transaction loads exceeding 100 transactions per second can be expected to require multiple nodes and transactions spanning nodes inherently consume more resources than ones local to a single node. These transactions must be coordinated across different nodes and their outcome recorded in more than one log. By minimizing this additional expense, it should be possible to construct a system that executes a thousand transactions per second with 20 to 30% of these transactions spanning two or more nodes.

Achieving this goal will require work in a number of different areas:

A. DP2

A major step towards these performance goals has been accomplished with the 1985 release of Tandem's new disc process, known as DP2. A number of changes in strategy have been incorporated that offer significant performance gains.

1. New Logging Strategy

Under the original Tandem disc process, DP1, each process that participates in a transaction performs a separate I/O to the log. While this I/O might carry the log records for more than one transaction, the log records generated by different disc processes go to different places on disc. In addition, a completely separate log is maintained for commit and abort records.

Typically, under DP2, all log records are sent to one disc process on which the log resides. The log records are buffered until a commit or abort record is added to the log buffer and the buffer is written to disc. This allows the TMF log to be written, typically, with no more than one I/O per transaction.

2. Buffered Cache

DP2 supports buffered cache for database pages. Previously, DP1 would write through to disc whenever an update was performed.

3. Overlapped CPU and I/O

   DP1 is a single-threaded process which waits while a physical I/O is in progress. DP2 performs computation on behalf of one request while waiting for a physical I/O against another.

4. Less expensive Checkpointing

   Since a Tandem disc process runs as a "process pair" in two processors, the primary disc process must inform its backup disc process of its state periodically (called checkpointing). This is necessary because the backup disc process is expected to assume active duties should the primary disc process's processor fail.

   Under DP1, the primary disc process checkpoints enough information to the backup disc process that the backup can always carry forward any partially-completed operation if the primary disc process fails.

   DP2, on the other hand, does not try this approach. Should the backup notice the failure of its primary's processor, the backup aborts any transactions in which the primary has participated. This provides DP2 with the ability to serve a write request without telling the backup disc process about it, providing a significant reduction in checkpointing [Borr84]. For Debit-Credit transactions, half as many checkpoint messages are sent moving one fifth as many bytes.

B. Intra-Node TMF Performance

This area is concerned with the overhead of beginning and ending a transaction, including the coordination of moving log records to disc, writing commit records, and coordinating the release of record locks after a transaction's commitment.

1. Performance Impact of Adding Processors

   Because the work of a transaction takes place in different processors on the node, all the processors must know of the existence of the transaction. When the transaction is to be committed, all the processors must ensure that all log records buffered in them are sent to the log before the commit record is sent.

   This requirement of the architecture presents an inherent problem -- the addition of a processor imposes an added burden on the existing processors. This threatens the ability to expand performance linearly with the linear addition of hardware. It is important to note that this added burden is functionally dependent on the product of the transaction rate and the number of processors. To achieve linear performance expandability, the transaction rate should increase as the number of processors in the node increases. But the cost of informing all processors about the transaction is dependant on the number of processors in the node. The total cost of telling all processors about all transactions is proportional to the product of the number of transactions and the number of processors. This cost is N-squared in terms of the number of processors.

Now that DP2's new strategies have reduced the cost of a transaction dramatically, the number of transactions that can be handled by a processor has increased. As the transaction rate increases, the penalty of adding a processor becomes more noticeable.

By introducing buffers (boxcars) for the messages sent between processors, the work for several transactions can be done with one message. The cost then drops low enough to be acceptable at 100 transactions per second on 16 processors. Notice that this does not remove the N-squared nature of the algorithm. It simply lowers the fixed cost of this function to a level at which the non-linearity is acceptable up to its ceiling of 16 processors.

2. Buffering of Commit Records

Currently, separate buffers are maintained in each processor to hold commit records. The processor containing the process that began the transaction buffers the commit record. Using 16 buffers with 100 transactions per second would mean an average arrival rate of 6.25 commit records per second into each buffer. To wait for two or three records in the buffer would impose an unacceptable delay in response time. This means that the buffering (box cars) is ineffective.

By directing all commit records to one buffer, it becomes reasonable to write an average of 10 records every .1 second.

C. Inter-Node (Network) Performance

Two major changes are being investigated to enhance the performance of transactions that span 2 or more nodes (network transactions). The first involves the mechanism by which the state of a transaction is recorded on disc. The second uses a different two-phased commit protocol to reduce the number of disc I/Os necessary to commit a transaction.

1. Recording the State on Disc

   Currently, a file is used in which a section is dedicated to each active network transaction. To record a change in a transaction's state (e.g., the transaction has entered PHASE-1), the disc must seek to the dedicated location in the file and update the section. This approach ensures that an I/O must occur to change the state of a transaction. Any other transactions changing state at the same time will almost certainly use a different part of the disc.

   By using the log to hold records indicating changes in the transaction's state, we can use buffering (box cars) to amortize the cost. If the records indicating changes in the network transaction's state share the same buffer with the local transaction's commit records, the benefits are increased even more.

2. Reducing Network Changes of State

   In the normal "Two Phased Commit Protocol" [Gray78] used by TMF, the home node of the transaction must notify the remote node or nodes of the outcome of the transaction once the remote nodes enter Phase-One of commit. Since the home node must fulfill this obligation even if it crashes, it MUST record on disc which nodes are remote nodes for the transaction before asking them to enter Phase-One.

   By using a different protocol known as the "Presumed Abort Protocol" [MoLi83], this requirement can be removed. The presumed-abort protocol simply changes the rules so that if the home node fails or communication is lost, the remote node will ask the home node about the outcome of the transaction. If the home node has no record of the transaction, the remote node may presume that the transaction has aborted.

   The change to the presumed abort protocol will mean that one record on the remote node (Phase-One) and one record on the home node (commit) must be written before the user can be told that the transaction has been committed. This saves one record write on the home node (the one before Phase-One).

With the FOX communication line, the messages from node to node are very inexpensive. By changing to the presumed-abort protocol and writing records to the log (with buffering), the cost of a network transaction should be reduced dramatically.

III. New Features

In addition to improving performance, Tandem is investigating a number of features directed toward large distributed databases with a special emphasis on high availability.

A. Replicated Databases

Many customers could use the ability to have their database replicated at different nodes. This implies that any updates are automatically made to enough copies of the database to guarantee that a subsequent read of the data item will show the change. To be of significant use, this mechanism must be invisible to the user and available even if a subset of the nodes are available. This would employ both Gifford's algorithm [Giff79] and some form of time-staged delivery to reconcile unavailable nodes once they become available.

B. Database Migration

Suppose two nodes A and B were available in a network. Database migration would consist of instructing the system to take the database that resides on node A (or some part of it) and copy that database to node B. As the database is being copied, the TMF log is shipped to B as it is created (or slightly later). Node B would process the log as it arrives, and decide whether the updates in the log affect the part of the database that it has received so far. If the updates are to the part that node B has received, the partial database is updated to match.

Eventually, the entire database is copied from A to B. If updates to the database on A are momentarily suspended, B is allowed to catch up, and B's copy designated as the official version, the migration is complete.

Customers requiring 24-hour-a-day access to their database seriously need this mechanism. Any computer must have some planned down time occasionally. It may be once a year to install some new software, or once every 5 years to paint the computer room.

C. Contingency Site

When a customer's business depends on its database, it is imperative that the database be accessible. This must be true even in the face of fires, bombs, floods, or other calamities. Such customers can survive a few minutes outage, but must promptly regain service. The effects of transactions up through shortly before the failure must be available.

In the previous section on database migration, there was a point at which node A contained the official version of the database and node B contained a copy that lagged momentarily behind. This state could be the stable mode of operation. B would function as a contingency site.

When B detected the loss of A, it would need to examine the log that it had received so far. Any transactions that did not have a commit record in the part of the log received by B would be aborted and their updates undone. At this point, B would start functioning as the official database. When A is available again, database migration is employed to bring it online.

Summary

TMF is a fully featured distributed recovery and concurrency mechanism presently in use at many customer sites. The challenges of very high transaction rates, coupled with continuous availability, provide the incentive to further enhance TMF in novel ways.

References

[Bart78] Bartlett, J. F., "A 'NonStop' Operating System", Eleventh Hawaii International Conference on System Sciences, 1978.

[Bart81] Bartlett, J. F., "A NonStop Kernel", Proceedings of Eighth Symposium on Operating System Principles, ACM, 1981.

[Borr81] Borr, A. J., "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing", Proc. Seventh International Conference on Very Large Data Bases, September 1981.

[Borr84] Borr, A. J., "Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach", Proc. Tenth International Conference on Very Large Data Bases, August 1984.

[Giff79] Gifford, D. K., "Weighted Voting for Replicated Data", ACM Operating Systems Review, Vol 13, No 5, December 1979, pp 150/162.

[Gray78] Gray, J. N., "Notes on Data Base Operating Systems", IBM Research Report RJ 2188, February 1978.

[MoLi83] Mohan, C., and Lindsay, B. G., "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", Proc. Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983.

The TABS Project[1]

Alfred Z. Spector
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

To simplify the construction of reliable, distributed programs, the TABS Project is performing research in the construction and use of general purpose, distributed transaction facilities. As part of this work, it has constructed a prototype and data objects that are built on it. The goals of the work are to show that a distributed transaction facility can simplify programming some types of distributed applications and that its performance can be satisfactory.

1. Introduction

The fundamental hypothesis of the TABS Project is that a general purpose, distributed transaction facility is useful in simplifying the construction of distributed applications and that such a facility can be implemented efficiently. We define a *general purpose distributed transaction facility* as a service that supports not only such standard abstractions as processes and inter-process communication, but also the execution of transactions and the implementation of data objects that can be called from within them. Unlike a database system's transaction facility that has been specialized to support only a particular database model, a general purpose transaction facility is integrated into an operating system and supports a collection of data types in a uniform way.

There are many references in the literature as to why transactions simplify the synchronization and recovery of collections of operations on shared, long-lived data objects. Gray succinctly discusses the benefits of transactions [Gray 80]. A *general purpose* distributed transaction facility provides additional benefits. One is that arbitrary user-defined operations can be used within transactions. The facility can support local or remote objects such as the tables and B-Trees frequently found in database systems, but it can also support other types of objects, such as I/O objects and queue-like objects. Another is that the facility may operate with better performance, because it has been implemented within the operating system, possibly using specialized hardware. A final advantage is that a common transaction facility should simplify the composition of activities that are implemented in different subsystems, such as a relational database system and a hierarchical file system.

[1]This work was supported by the IBM Corporation, and the Defense Advanced Research Projects Agency, ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

However, the challenges in constructing distributed transaction facilities are manifold. For example, the algorithms for synchronizing and recovering typical database abstractions may not be efficient enough for other types of abstract objects [Schwarz and Spector 84, Weihl and Liskov 83]. Providing efficient-enough support for the fine grain operations that might be used in some applications is difficult. Defining the system interface is not clear; for example, there is a question of whether synchronization should be done explicitly by the implementor of a type, or automatically by the system. The complexity of the facility itself makes it difficult to determine how the system should be organized; there are many ways the facility's process management, inter-process communication, virtual memory management, recovery, transaction management, synchronization, and naming components can interact.

The TABS Project at Carnegie-Mellon has been considering questions such as these. The Project began in September 1982, and we have been considering the basic algorithms for using and implementing transaction facilities. For example, new recovery algorithms and a novel replication algorithm for directory objects are described in recent technical reports [Schwarz 84, Bloch et al. 84]. In addition, we have built a prototype distributed transaction processing facility, called the TABS Prototype [Spector et al. 84, Spector et al. 85], with two goals:

- The first goal is to learn how well various applications and data objects are supported by the TABS primitives. We have implemented queues, arrays, a recoverable display package, a B-Tree package, and a bank record type suitable for simulating ET1 [Anonymous et al. 85]. We are working on replicated objects that permit access despite the failures of one or more processing nodes.

- The second goal is to measure the performance of the facility and to use the resulting performance metrics as a base from which to predict how well general purpose transaction facilities could perform. Because of the certain artifacts of the present implementation, we do not expect its performance to be outstanding, but we can show conclusively why transaction facilities with high performance can be produced.

The TABS prototype is functioning well and is described after a very brief survey of related work. The last section presents the status of our work and our initial conclusions.

2. Related Work

Previous work on abstract objects and their implementation on distributed systems has influenced TABS. The *class* construct of Simula, the client/server model as described by Watson [Watson 81], and work on message passing and remote procedure calls, such as described by Rashid and Robertson [Rashid and Robertson 81] and Birrell and Nelson [Birrell and Nelson 84] have had major influences on us. In TABS, objects are implemented within server processes, and operations on objects are invoked via a location transparent message passing facility. Remote procedure calls are used to reduce the programming effort of message packing, unpacking, and dispatching. Type-specific locking, write ahead log-based recovery, and the tree-structured variant of the two-phase commit protocol are the bases of the transaction-specific components of TABS [Korth 83, Schwarz and Spector 84, Harder and Reuter 83, Schwarz 84, Lindsay et al. 79].

Related systems work includes the R* distributed database management system and the Argus programming language [Lindsay et al. 84, Liskov et al. 83]. R* logically contains a transaction facility similar to that of TABS, though it is primarily intended to support relational database servers, and is organized quite differently. Internally, Argus contains many facilities that are analogous to those of TABS and R*, but it has the more ambitious goal of making those facilities very easy to use.

3. TABS Prototype

The TABS Prototype is implemented in Pascal on a collection of networked Perq workstations [Perq Systems Corporation 84] running a modified version of the Accent operating system kernel [Rashid and Robertson 81]. At each node, there is one instance of the TABS system facilities and one or more user-programmed data servers and/or applications that access the facilities of TABS. (See Figure 3-1.)

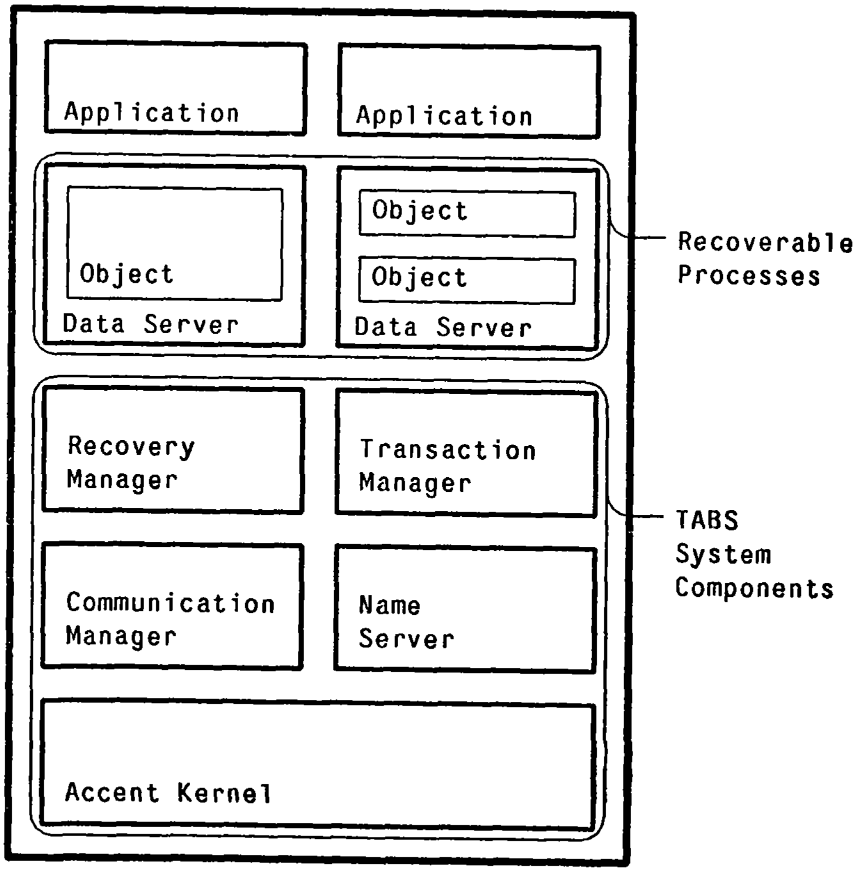


Figure 3-1: The Basic Components of a TABS Node

Each data server may encapsulate one or more objects of multiple types. Data objects are stored in the virtual memory of data servers and operations are performed via request and response messages directed to it. As mentioned, data servers use a write-ahead log-based recovery mechanism, which is implemented with the cooperation of the Recovery Manager and Accent kernel.

. TABS provides various libraries to support transaction initiation and commitment, synchronization (performed via type-specific locking), recovery, lightweight processes in the data servers, and naming. Example routines from the TABS libraries are listed in Table 3-1.

Routine	Purpose
ReadPermanentData(DiskAddress) returns (VirtualAddress, DataSize)	Startup
RecoverServer	Startup
AcceptRequests(DispatchFunction)	Startup
LockObject(ObjectID, LockMode)	Locking
IsObjectLocked(ObjectID) returns (Boolean)	Locking
PinObject(ObjectID)	Pinning
UnPinObject(ObjectID)	Pinning
PinAndBufferOldValue_Object(ObjectID)	Pinning, Logging
BufferNewValueAndLog_Object(ObjectID)	Logging, Pinning
BeginTransaction(TransactionID) returns(NewTransactionID)	Transaction Management
EndTransaction(TransactionID) returns(Boolean)	Transaction Management
AbortTransaction(TransactionID)	Transaction Management
TransactionIsAborted(TransactionID) [*exception*]	Transaction Management
SignIn(Name, Type, Port, ObjectID)	Naming
LookUp(Name, NodeName, RequestNumberOfPortIDs, MaxWait) returns(ArrayOfPortIDPairs, ReturnNumberOfPortIDs)	Naming

Table 3-1: Partial Listing of TABS Library Interface
This table includes about one-half of the routines provided in the TABS Library. Most of the routines are used by data servers for the purposes of initialization, synchronization, and recovery. Others are used to initiate, commit, and abort transactions. The remainder are used for naming.

With the exception of the coroutine facility and locking, which are implemented entirely with data servers, most of the functions in TABS are implemented by four component processes that run on a specially modified version of the Accent kernel (see Figure 3-1). A single TABS component process is assigned to each of the functions of name management, network communication, recovery and log management, and transaction commit supervision.

The functions of the TABS System components and the modifications to the kernel are summarized below:

- The Accent kernel provides heavyweight processes with 32-bit virtual address spaces [Rashid 83]. Processes communicate via messages addressed to ports. The TABS project has modified the kernel to permit data servers to use virtual memory to store permanent, failure atomic objects and to provide the Recovery Manager with special control over the paging of these recoverable data. The enhancements are necessary to implement our logging protocols.

- The Recovery Manager has two principal functions. During normal operation, the Recovery Manager acts as a log manager, accepting logging requests from data servers and the Transaction Manager and writing the log when required. During recovery from node failure, or during transaction abort, the Recovery Manager drives the Transaction Manager and data servers back to a transaction-consistent state.

- The Communication Manager is the only process that has access to the network. It sends inter-node datagrams on behalf of the local Transaction Manager in order to implement the distributed commit protocol, and it maintains a session for every communication link between a local process and a remote data server.

- The Transaction Manager coordinates initiation, commit and abort of local and distributed transactions.

- The Name Server maps object names into a set of possible locations.

Recent papers describe the implementation of TABS in more detail [Spector et al. 84, Spector et al. 85].

4. Discussion

TABS began to operate in the Fall of 1984, but only recently could it support use by non-implementors. As of March 1985, all the necessary facilities to support data servers with read/write locking and value logging were operational. A compatible, operation (or transition) logging algorithm that can be used in conjunction with the value logging algorithm has been fully implemented and tested, but the libraries do not yet contain the necessary code to interface to it. We are also presently working on the design of type-specific locking packages.

We have executed a variety of benchmarks on the prototype to gauge its performance. Typical performance numbers are 72 milliseconds for local read-only transactions that page-fault the data page and 244 milliseconds for local write transactions that modify an already resident data page. We project that slightly changing the paging system and integrating the Recovery Manager and the Transaction Manager into the kernel would improve these times to about 72 and 151 milliseconds, respectively. With faster hardware, more disks, and a more efficient message passing implementation, we believe the performance would be excellent [Spector et al. 85].

Our performance evaluation shows that the current implementation of TABS has major inefficiencies due to the division of the TABS system components into separate processes. Also, the TABS physical log I/O, message passing, and paging facilities need re-implementation or tuning. Future work on TABS should address these issues and look into programming and debugging support for data servers. We have neglected these latter issues. Once we have complete support for operation logging, a comparison of the relative merits of our two logging algorithms is needed.

Overall, our experience with programming data servers has made us conclude that the TABS Prototype will make some applications easier to program, and that at least some applications need the flexibility that TABS provides. Our performance analysis shows that a system based on the ideas of the prototype should be quite efficient. Certainly, additional research is required to determine more carefully the correct function and implementation of general purpose distributed transaction facilities, but our experience with TABS prototype has led us to believe they are a good idea.

Acknowledgments

Acknowledgments to Joshua Bloch, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Sherri Menees, Peter M. Schwarz who have contributed to the TABS Project in many ways, and to Ruth McKillop for assistance with this paper.

References

[Anonymous et al. 85]
Anonymous et al.
A Measure of Transaction Processing Power.
Technical Report TR 85.1, Tandem Corporation, January, 1985.
Submitted for Publication.

[Birrell and Nelson 84]
Andrew D. Birrell, Bruce J. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):38-59, 1984.

[Bloch et al. 84] Joshua J. Bloch, Dean S Daniels, Alfred Z. Spector.
Weighted Voting for Directories: A Comprehensive Study.
Carnegie-Mellon Report CMU-CS-84-114, Carnegie-Mellon University, April, 1984.

[Gray 80] James N. Gray.
A Transaction Model.
IBM Research Report RJ2895, IBM Research Laboratory, San Jose, CA, August, 1980.

[Harder and Reuter 83]
Theo Harder and Andreas Reuter.
Principles of Transaction-Oriented Database Recovery.
ACM Computing Surveys 15(4):287-318, December, 1983.

[Korth 83] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1), January, 1983.

[Lindsay et al. 79] Bruce G. Lindsay et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, CA, July, 1979.
Also appears in Droffen and Poole(editors), *Distributed Databases*, Cambridge University Press, 1980.

[Lindsay et al. 84] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost.
Computation and Communication in R*: A Distributed Database Manager.
ACM Transactions on Computer Systems 2(1):24-38, February, 1984.

[Liskov et al. 83] B. Liskov, M. Herlihy, P. Johnson, G. Leavent, R. Scheifler, W. Weihl.
Preliminary Argus Reference Manual.
Programming Methodology Group Memo 39, Massachusetts Institute of Technology Laboratory for Computer Science, October, 1983.

[Perq Systems Corporation 84]
Perq System Overview
March 1984 edition, Perq Systems Corporation, Pittsburgh, Pennsylvania, 1984.

[Rashid 83] Richard F. Rashid.
Accent Kernel Interface Manual.
November, 1983.

[Rashid and Robertson 81]
Richard Rashid, George Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the Eighth Symposium on Operating System Principles.* ACM, 1981.

[Schwarz 84] Peter M. Schwarz.
Transactions on Typed Objects.
PhD thesis, Carnegie-Mellon University, December, 1984.
Available as CMU Report CMU-CS-84-166.

[Schwarz and Spector 84]
Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
ACM Transactions on Computer Systems 2(3):223-250, July, 1984.
Also available as Carnegie-Mellon Report CMU-CS-83-163, November 1983.

[Spector et al. 84] Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, Peter M. Schwarz.
Support for Distributed Transactions in the TABS Prototype.
In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 186-206. October, 1984.
Also available as Carnegie-Mellon Report CMU-CS-84-132, July 1984. To appear in Transactions On Software Engineering.

[Spector et al. 85] Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Randy Pausch.
Distributed Transactions for Reliable Systems.
CMU Report CMU-CS-85-117, Carnegie-Mellon University, April, 1985.
Submitted for Publication.

[Watson 81] R.W. Watson.
Distributed system architecture model.
In B.W. Lampson (editors), *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 2, pages 10-43. Springer-Verlag, 1981.

[Weihl and Liskov 83]
W. Weihl, B. Liskov.
Specification and Implementation of Resilient, Atomic Data Types.
In *Symposium on Programming Language Issues in Software Systems.* June, 1983.

Atomic Data Types

William E. Weihl[1]
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
(617) 253-6030
Arpanet: weihl@mit-xx

Abstract

Atomic data types can be used to build transaction systems that exhibit high levels of concurrency, and at the same time have the modularity properties that normally accrue from using data abstractions. In this paper we summarize our work on atomic data types, focusing on three issues: what makes a type atomic, how to specify an atomic type, and how to implement an atomic type.

1 Introduction

There are many applications in which the manipulation and preservation of long-lived, on-line data is of primary importance. Examples of such applications include banking systems, airline reservation systems, office automation systems, database systems, and various components of operating systems. A major issue in such systems is preserving the consistency of on-line data in the presence of concurrency and hardware failures. In addition, such systems often have strong performance, reliability, and availability requirements. The goal of our research is to develop systematic techniques for structuring and maintaining programs for such applications. Our focus is on distributed systems, although many of the problems that we are studying arise as well in centralized systems. Our work is being done in conjunction with the Argus project at MIT [LISK83a,LISK83b].

Recent research, in both database systems (e.g., [ESWA76, GRAY78, GRAY81]) and distributed systems (e.g., [LISK83a, REED78, SPEC84]), indicates that one effective way of structuring systems to cope with concurrency and failures is to make the activities that use and manipulate data *atomic*. Atomic activities are often referred to as *actions* or *transactions*; they were first identified in work on databases. Atomic activities are characterized informally by two properties: serializability and recoverability. *Serializability* means that the concurrent execution of a group of activities is equivalent to some serial execution of the same activities. *Recoverability* means that each activity appears to be all-or-nothing: either it executes successfully to completion (in which case we say that it *commits*), or it has no effect on data shared with other activities (in which case we say that it *aborts*).

Atomicity simplifies the problem of maintaining consistency by decreasing the number of cases that need to be considered. Since aborted activities have no effect, and every concurrent execution is equivalent to some serial execution, consistency is ensured as long as every possible serial execution of committed activities maintains consistency. Even though activities execute concurrently, concurrency can be ignored when checking for consistency. In general, atomicity simplifies

[1]Support for this research was provided in part by a graduate fellowship from the Hertz Foundation, in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS79-23769.

reasoning about the correctness of a system, since one can reason about the partial correctness of an individual activity (e.g., whether it preserves consistency) without considering the other activities that might be present in the system.

We have been exploring an approach in which atomicity is achieved through the shared data objects, which must be implemented in such a way that the activities using them appear to be atomic. Objects that provide appropriate synchronization and recovery are called *atomic objects*; atomicity is guaranteed only when all objects shared by activities are atomic objects. By encapsulating the synchronization and recovery needed to support atomicity in the implementations of the shared objects, we can enhance modularity. In addition, by using information about the specifications of the shared objects, we can increase concurrency among activities. For example, if the specifications of the types imply that two operations commute, we can allow activities to execute the operations concurrently, even if both operations update the same object. Other researchers have also recognized the potential performance gains of using information about the specifications of objects in this way; numerous examples can be found in [ALLC83, REUT82, SCHW84, WEIH84, WEIH85a].

Atomic objects are encapsulated within *atomic abstract data types.* An abstract data type consists of a set of objects and a set of primitive operations; the primitive operations are the only means of accessing and manipulating the objects [LISK74]. In addition, the operations of an atomic type ensure serializability and recoverability of activities using objects of the type.

We have been studying the following three questions about atomic types:

- *What is an atomic type?* We need a precise characterization of the behavior of atomic types. For example, we need to know how much concurrency can be allowed by an atomic type.

- *How do we specify an atomic type?* What aspects of the type's behavior must appear in the specification of an atomic type, and how should the specification be structured?

- *How do we implement an atomic type?* What problems must be solved in implementing an atomic type, and what kinds of programming language constructs make this task simpler?

In the next three sections of this paper we summarize our work to date on these questions. We begin in Section 2 by discussing our specification method for atomic types. Then, in Section 3, we discuss what it means for a type to be atomic. Next, in Section 4, we discuss issues involved in implementing atomic types. Finally, in Section 5, we summarize our results and discuss directions for further research.

2 Specification Method

A type's specification serves as a kind of contract between the implementor of the type and its users: the implementor guarantees that the type's behavior will obey the specification, and the users rely on this guarantee. Most importantly, however, the only assumptions that the users of a type can make about the type's behavior are those given in the type's specification. Thus, the implementor of a type has complete freedom in making implementation decisions, as long as the implementation satisfies the specification.

In writing specifications for atomic types, we have found it helpful to pin down the behavior of the operations, initially assuming no concurrency and no failures, and to deal with concurrency and failures later. In other words, we imagine that the objects exist in an environment in which all activities are executed sequentially, and in which activities never abort. We call this specification of a type's operations the *serial specification* of the type. The serial specifications of atomic types are particularly useful in reasoning about an activity that uses atomic objects. The atomicity of activities means that they are "interference-free," so we can reason about the partial correctness of an individual activity without considering the other activities that might be sharing objects with it [BEST81]. This reasoning process is essentially the same as for sequential programs; the only information required about objects is how they behave in a sequential environment. The serial specification of an atomic type describes the assumptions that a single activity can make about the behavior of the type's objects, and serves to define the correct serial executions involving objects of the type.

As we illustrate in Section 3 below, many different protocols can be used to ensure atomicity. However, if different types use different protocols, atomicity can be violated. To be able to ensure that the protocols used by different types are compatible, the specification of an atomic type must include some information about how the type manages concurrency and failures. We use the *behavioral specification* of a type to describe the concurrent executions permitted by the type. In Section 3, we discuss constraints on the behavioral specifications of the types in a system that ensure that the types cooperate to guarantee atomicity.

A specification of an atomic type should probably include other requirements, perhaps related to performance. For example, for a system designer to be able to make statements about the level of concurrency among activities in a system, the specification of each type shared by the activities must include requirements on the amount of concurrency that must be provided by an implementation of the type. Similarly, a type's specification should include information about how implementations of the type are permitted to deal with deadlocks. The form and content of such additional requirements are the subject of future research.

3 Defining Atomicity

Our goal in defining atomicity for types is to ensure that, if all objects shared by activities are atomic, then the activities are serializable and recoverable — i.e., atomic. Atomicity of activities is a *global* property, because it is a property of all of the activities in a system. However, atomicity for a type's objects must be a *local* property: it deals only with the events (invocations and returns of operations and commits and abort of activities) involving individual objects of the type. Such locality is essential if atomic types are to be specified and implemented independently of each other and of the activities that use them.

In [WEIH84] we explored three local properties of types, each of which suffices to ensure that activities are atomic. In addition, each of the three properties is optimal: no strictly weaker local property suffices to ensure atomicity of activities. In other words, these properties define precise limits on the concurrency that can be permitted by an atomic type. The three properties characterize respectively the behavior of three classes of protocols: protocols like two-phase locking (e.g., see [ESWA76, MOSS81]), in which the serialization order of activities is determined by the order in which they access objects; multi-version timestamp-based protocols (e.g., see [REED78]), in which the serialization order of activities is determined by a total order based on when activities begin, rather than on the order in which they access objects; and hybrid protocols (e.g., see [BERN81a, CHAN82, DUBO82, WEIH85b]), which use a combination of these techniques. We use the terms *dynamic atomicity*, *static atomicity*, and *hybrid atomicity* to denote the three properties.

The existence of several distinct definitions of atomicity for types indicates that there are many different "kinds" of atomic types, not all of which are compatible. For example, atomicity of activities is ensured if all types shared by activities are dynamic atomic; however, atomicity is not guaranteed if static atomic types and dynamic atomic types are used in the same system. As a result, a system designer must choose which kind of atomic type to use in a given system.

In the remainder of this section we sketch the definition of dynamic atomicity. Our approach is informal, and is intended only to convey some intuition to the reader. More precise definitions of all three local atomicity properties that we have studied can be found in [WEIH84].

Dynamic atomicity characterizes the behavior of objects implemented with protocols which determine the serialization order of activities dynamically based on the order in which the activities access the objects. The essence of such protocols is the notion of *delay*: if the steps executed by one committed activity conflict with the steps executed by another committed activity, then one of the activities must be delayed until the other has committed. However, an implementation of dynamic atomicity need not actually delay activities to achieve this effect. All that is necessary is that the overall effect for committed activities be as if conflicts were resolved by delays. Indeed, an optimistic protocol, which resolves conflicts by aborting some activities when they try to commit, could be used to implement dynamic atomicity.

Typical definitions of atomicity in the literature state that two activities conflict if an operation executed by one does not commute with an operation executed by the other. For example, if one activity reads an object written by another (or vice versa), then the two activities conflict. We have found that straightforward generalizations of these definitions do not lead to an optimal local atomicity property. Since we are interested in understanding the limits on concurrency that are needed locally to ensure global atomicity, we have taken a slightly different approach. Rather than focusing on the conflicts between operations executed by activities, we look at the possible orders in which the committed activities can be serialized — i.e., the orders in which the activities can be executed serially so that they execute exactly the same steps. The existence of a conflict between two activities typically implies a constraint on the orders in which the activities can be serialized. For example, if one activity reads a value written by another, the first activity must be serialized after the second. This emphasis on the possible serialization orders leads to a more basic and more general definition of atomicity.

An important feature of our framework is that the question of whether an execution is serializable in a given order is resolved by looking at the serial specifications of the objects involved in the execution. In other words, whether activities are serializable depends only on their expectations about the objects as defined by the serial specifications, and not on some pre-defined model of execution (e.g., that all operations are reads or writes). In defining atomicity, we ignore what happens inside the implementation of an object, and focus on what is observable at the interface between the objects in a system and the activities that use them. Since the serial specification of each type defines what is acceptable in a serial execution, we define serializability in terms of it.

Our intuition about delays and conflicts can be restated as follows: if two activities *a* and *b* are concurrent (neither is delayed until the other has committed), then they must be serializable in both possible orders (*a* followed by *b* and *b* followed by *a*); if one is delayed until the other has committed, then they must be serializable in the order in which they commit. These ideas are captured more precisely in the following definitions.

First, we need to define the notion of *delay*. We do this as follows: given an execution *h*, define the binary relation *precedes*(*h*) on activities to contain all pairs <*a*, *b*> such that some operation invoked by *b* in *h* terminates after *a* commits in *h*.[2] The relation *precedes*(*h*) captures our intuitive notion of delay: if *b* is delayed until after *a* commits in an execution *h*, then <*a*,*b*> will be in *precedes*(*h*).

Now, if two committed activities are concurrent (i.e., unrelated by *precedes*), dynamic atomicity requires them to be serializable in both possible orders. In general, dynamic atomicity requires all executions *h* permitted by an object to satisfy the following property: the committed activities in *h* must be serializable in all total orders consistent with *precedes*(*h*). As noted in [WEIH84], natural restrictions on executions guarantee that the "precedes" relation is a partial order, ensuring that there are total orders consistent with it.

We have shown [WEIH83, WEIH84] that if all objects in a system are dynamic atomic, then the activities in the system are atomic. The proof hinges on the fact that objects never "disagree" about the "precedes" relation: there is always at least one total order that is consistent with all of the local "precedes" relations. Thus, if each object ensures serializability in all total orders consistent with its local "precedes" relation, then the objects will agree on at least one (global) serialization order. This means that dynamic atomicity satisfies our goals for a local atomicity property. As mentioned earlier, we have also shown [WEIH84] that dynamic atomicity is an *optimal* local atomicity property: no strictly weaker (i.e., more permissive) local property of objects suffices to ensure global atomicity. This shows that dynamic atomicity defines a precise limit on the amount of concurrency that can be permitted by an atomic type.

4 Implementation Issues

Many issues are raised by the problem of implementing atomic types. Among those that we have studied are: concurrency control protocols that use information about the specifications of types to permit more concurrency, structuring techniques for implementations of atomic types, and linguistic support for atomic types. Each of these is discussed below.

4.1 Protocols

The local atomicity properties described above define precise limits on the amount of concurrency that can be permitted by an implementation of an atomic type, but they do not indicate exactly how to implement an atomic type. An important problem is the design of general and efficient protocols that use information about the specifications of types to achieve some or all of the concurrency permitted by a given local atomicity property. We have focused most of our efforts to date on dynamic atomicity, in part because it characterizes the behavior of known two-phase locking protocols, which are probably the most widely used concurrency control protocols, and in part because it has been adopted as the standard local property for atomic types in Argus [LISK83a, LISK83b].

Two phase locking protocols (e.g., see [ESWA76, BERN81b, KORT81, SCHW84, WEIH84]) give relatively simple and efficient implementations of dynamic atomicity. However, dynamic atomicity is more general than two-phase locking, in the sense that it permits more concurrency than can be achieved by an implementation based on known two-phase locking protocols. These protocols have

[2] Note that there is a distinction between an activity completing by committing or aborting, and an operation terminating by returning information. An activity may execute any number of operations before completing. In addition, it is possible for *b* to commit after *a* in *h*, yet for all of *b*'s operations to terminate before *a* commits; in this case the pair <*a*,*b*> is not in *precedes*(*h*).

two structural limitations that prevent them from achieving all of the concurrency allowed by dynamic atomicity. First, they are conflict-based: synchronization is based on a pair-wise comparison of operations executed by concurrent activities. In contrast, dynamic atomicity depends on the *sequences* of operations executed by activities. Second, the protocols are history-independent: synchronization is independent of past history, in particular the operations executed by committed activities. In contrast, dynamic atomicity depends on the entire execution. A more detailed discussion of these limitations, and example implementations that achieve a level of concurrency that cannot be achieved by a locking implementation, can be found in [WEIH84].

4.2 Implementation Structure and Linguistic Support

A basic question that must be addressed for the potential benefits of atomic types to be realized is how to implement an atomic type. One implementation problem, discussed above, is the design of concurrency control and recovery protocols. Another problem of equal importance is how to structure the implementation of an atomic type. Given a particular structure for implementations, we can then consider the design of programming language constructs that support the chosen structure.

We have explored two different structures for implementations of atomic types. One is the structure supported by the constructs in Argus [WEIH85A]. In Argus, the programmer relies on the system to update the representation of an object as activities that used the object commit and abort. In addition, the names of the activities executing operations are not available to the user program. This approach can be characterized as *implicit* in that much of the management of synchronization state (e.g., locks) and recovery information is handled by the underlying system. An alternative that we explored in [WEIH84] is to permit user code to use the names of activities, and for the programmer to supply code that is run when activities complete to update the representations of objects. This approach is more *explicit* in that the programmer can manage synchronization and recovery directly. The main conclusion that we have drawn from analyzing these two approaches is that the Argus approach has relatively limited expressive power, particularly with respect to the efficiency of blocking activities when a conflict occurs and then unblocking them when the conflict is resolved. The alternative that we have explored can express certain structures more conveniently and more efficiently, but the resulting programs are sometimes more complicated. We note, however, that neither approach leads to simple implementations that are easy to understand.

5 Conclusions

We have been studying an organization for distributed systems based on the use of atomic activities and data abstractions. Atomicity provides two benefits: it allows us to specify an abstract data type in essentially the same way as we would specify a type in a sequential system, and it permits us to reason about the partial correctness of an individual activity while ignoring concurrency and failures. Data abstractions, as in sequential systems, are important for modularity. The implementation of an atomic type can be verified independently of which actions use objects of the type and of other types shared by those actions. In addition, our emphasis on abstraction permits us to achieve high performance without sacrificing atomicity. In particular, atomicity is only important at the abstract level; it is not necessary that activities appear atomic when viewed at the level of the representations of the shared atomic objects. The specification of a type determines the limits on the concurrency among activities that can be permitted by an implementation; as illustrated earlier, significantly more concurrency is permitted by the specification of many types than can be achieved, for example, by treating the operations as reads and writes.

The modularity properties resulting from this organization are especially useful if a system performs poorly because of internal concurrency limitations. In such a case, it may be possible to identify certain shared objects as bottlenecks, and to substitute more concurrent (albeit more complex) implementations for the types defining those objects. Thus, it may be possible to trade simplicity for concurrency systematically.

As we mentioned earlier, this work is being performed in conjunction with the Argus project at MIT. Argus is a programming language and system designed to support the implementation of distributed systems. As discussed above, it contains constructs that support the implementation of user-defined atomic types, as well as other constructs that help manage other problems of distribution and reliability. Argus is currently running on a collection of DEC VAX-11/750 machines connected by a network. We are beginning to build applications so that we can evaluate both the language and the overall approach to structuring distributed systems. We also hope to use Argus as a base for experimenting with other ways of structuring implementations of atomic types, such as the explicit approach discussed earlier.

6 Acknowledgements

I am indebted to all the members of the Programming Methodology Group at MIT for their many contributions to my work on atomic types, and for their combined efforts on Argus. In addition, I would like to thank Gary Leavens, Barbara Liskov, and Brian Oki for their helpful comments on drafts of this paper.

7 References

[ALLC83] Allchin, J., and McKendry, M. Synchronization and recovery of actions. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 31-44. Montreal, Canada, August, 1983.

[BERN81a] Bernstein, P., and Goodman, N. Concurrency control in distributed database systems. *ACM Computing Surveys* 13(2): 185-221, June, 1981.

[BERN81b] Bernstein, P., Goodman, N., and Lai, M.-Y. Two part proof schema for database concurrency control. In *Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 71-84. February, 1981.

[BEST81] Best, E., and Randell, B. A formal model of atomicity in asynchronous systems. *Acta Informatica* 16: 93-124, 1981.

[CHAN82] Chan, A., *et al. The implementation of an integrated concurrency control and recovery scheme.* Technical Report CCA-82-01, Computer Corporation of America, March, 1982.

[DUBO82] DuBourdieu, D. Implementation of distributed transactions. In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 81-94. 1982.

[ESWA76] Eswaran, K., *et al.* The notions of consistency and predicate locks in a database system. *CACM* 19(11): 624-633, November, 1976.

[GRAY78] Gray, J. Notes on Database Operating Systems. In *Lecture Notes in Computer Science*, Volume 60: Operating Systems — An Advanced Course. Springer-Verlag, 1978.

[GRAY81] Gray, J. The transaction concept: virtues and limitations. In *Proceedings of the 1981 VLDB Conference*, pages 144-154. IEEE, 1981.

[KORT81] Korth, H. *Locking protocols: general lock classes and deadlock freedom.* PhD thesis, Princeton University, 1981.

[LISK74] Liskov, B., and Zilles, S. Programming with abstract data types. In *Sigplan Notices,* Volume 9: *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages*, pages 50-59. ACM, 1974.

[LISK83a] Liskov, B., and Scheifler, R. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems* 5(3): 381-404, July, 1983.

[LISK83b] Liskov, B., *et al. Preliminary Argus reference manual.* Programming Methodology Group Memo 39, MIT Laboratory for Computer Science, October, 1983.

[MOSS81] Moss, J. *Nested transactions: an approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, 1981. Available as Technical Report MIT/LCS/TR-260.

[REED78] Reed, D. *Naming and synchronization in a decentralized computer system.* PhD thesis, Massachusetts Institute of Technology, 1978. Available as Technical Report MIT/LCS/TR-205.

[REUT82] Reuter, A. Concurrency on high-traffic data elements. In *Proceedings of the Symposium on Principles of Database Systems*, pages 83-92. ACM, Los Angeles, CA, March, 1982.

[SCHW84] Schwarz, P., and Spector, A. Synchronizing shared abstract types. *ACM Transactions on Computer Systems* 2(3), August, 1984.

[SPEC84] Spector, A., *et al. Support for distributed transactions in the TABS prototype.* Technical Report CMU-CS-84-132, Carnegie-Mellon University, July, 1984.

[WEIH83] Weihl, W. Data-dependent concurrency control and recovery. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 63-75. Montreal, Canada, August, 1983.

[WEIH84] Weihl, W. *Specification and implementation of atomic data types.* PhD thesis, Massachusetts Institute of Technology, 1984. Available as Technical Report MIT/LCS/TR-314.

[WEIH85a] Weihl, W., and Liskov, B. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, April, 1985.

[WEIH85b] Weihl, W. Distributed version management for read-only actions. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, August, 1985, to appear.

Improving Availability and Performance of Distributed Database Systems

Arvola Chan and Sunil Sarin

Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142
(617) 492-8860

Abstract

This paper surveys three ongoing projects at Computer Corporation of America aimed at improving the availability and performance of database systems through innovations in transaction management techniques.

1. Introduction

Concurrency control, replicated data handling, and recovery management all have major impact on the availability and performance of a database system, especially in a distributed environment. This paper surveys three ongoing projects at CCA aimed at improving the availability and performance of database systems through innovations in each of these areas. The first project, sponsored by DARPA and NAVELEX, involves the development of a prototype homogeneous distributed database system for a geographically dispersed environment. This system is designed to support transparent data replication in order to provide high availability in the face of site failures. However, it is not equipped to handle network partition situations. The problem of achieving high availability in the presence of arbitrary communication failures, including network partitions, is addressed in our second project. Sponsored by DARPA and the Air Force Systems Command, this latter project is aimed at designing and prototyping transaction management algorithms for applications which require continued operation and which can trade some loss of data integrity for high availability. Our third project, currently supported by internal R&D funds, is aimed at addressing the real time (or near real time) requirements of defense mission-critical applications.

2. Homogeneous Distributed Databases

LDM/DDM is a database system under development at CCA for operation within the Ada programming environment [CHAN81, CHAN83a, CHAN85b]. This system has been built from the ground up, starting with a local data manager (LDM) which can operate both stand alone and as a component of a distributed system. LDM incorporates a semantically rich data model and has been specially engineered to support operations of the distributed data manager (DDM). Applications and end users of DDM are shielded from almost all complexities that may arise from data allocation and site failures. The following is a brief summary of LDM/DDM innovations in the area of transaction management. As of this writing, the implementation of LDM is complete. The implementation of DDM is well underway and is expected to be completed by the end of the year. All of the DDM features described in the following discussions have been implemented and have undergone some preliminary testing.

Speeding up Read-only Transactions

Most database systems maintain before images of data objects in order to support transaction and system recovery. In LDM/DDM, these old versions of data objects are exploited to improve the achievable degree of concurrency among transactions, through the distinction of read-only transactions from update transactions. Concurrent access to the same

Portions of this research were supported by the Defense Advanced Research Projects Agency of the Department of Defense (DARPA), the Naval Electronics Systems Command (NAVELEX), and the Air Force Systems Command at Rome Air Development Center (RADC) under contracts N00039-82-C-0226 and F30602-84-C-0112. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NAVELEX, RADC, or the U.S. Government.

logical data object is facilitated by requiring that update transactions create new versions rather than overwrite old versions, and that appropriate versions are read by read-only transactions. The salient features of the multi-version scheme implemented in LDM [CHAN82] include:

1. Conflicts between read-only transactions and update transactions are completely eliminated.

2. Read-only transactions never have to be aborted because of the unavailability of appropriate object versions.

3. Garbage collection of old object versions can be carried out efficiently.

The scheme calls for each object version (page) in the physical database to be labeled with the identifier of its creating transaction and the use of a single common pool of pages, called the version pool, for storing the old versions of pages potentially needed by in-progress transactions. Each read-only transaction is required to obtain a Completed Transaction List (CTL) at the time of its initiation. The CTL is a compact representation of the identities of all update transactions that have completed execution (either committed or aborted) at a given time. Read-only transactions set no locks and therefore do not conflict with update transactions. They see a consistent snapshot of the database by reading the latest version of each needed data object that has been created by a transaction that is on its associated CTL. The efficacy of this multi-version scheme in an environment where long read-only transactions are present with update transactions has been demonstrated in a simulation study [CARE84]. An efficient extension for operation within a distributed environment is described in [CHAN85a]. The extension requires a minor modification to the protocol used for achieving atomic commitment of transactions. Certain transaction completion information (potential transaction dependency information), in the form of differential CTLs, must be propagated during the execution of the commit algorithm. Such dependency information is piggybacked on messages already needed for achieving atomic commitment such that no additional messages are introduced.

Speeding Up Update Transactions

Database fragments in DDM are collected into groups which form the units for replication [CHAN83b]. Let X be a fragment group that is to be replicated n times, at sites S1, ..., Sn. Two kinds of fragment group copies are distinguished. The DBA can designate k (less than or equal to n) of these as regular sites for the fragment group. The remaining (n-k) sites are treated as backup sites. Regular sites are used for storing copies to be used during normal transaction processing; backup sites are used to improve resiliency in the presence of site failures. In general, each replicated copy of the fragment group X can be in one of three states: online, offline, or failed. Online copies are accessible, up-to-date, and updated synchronously. Offline copies are accessible, normally out-of-date, and are updated only in a background fashion. Failed copies are of course inaccessible. Where possible, the system will maintain k online copies of fragment group X, preferrably at the k regular sites. As sites fail and recover, the system will automatically bring offline copies online and vice versa, in order to maintain the desired degree of resiliency and availability. By limiting the number of online copies at any one time, response to update transactions can be improved. At the same time, resilency against total failures [SKEE83] can be assured by specifying an adequate number of regular and backup copies. In the unlikely event that a total failure does occur, DDM is designed to provide automatic recovery without human intervention.

Site Status Monitoring

Data replication is used in DDM to improve data availability, so that retrievals and updates (in particular) can be performed even when some sites in the system are not operational. The correct implementation of distributed updates over replicated copies of data in DDM requires that site status be monitored constantly. DDM employs the available copies approach [BERN84] to replicated data handling (i.e., the policy of requiring read operations to read a single copy, and update operations to update all available copies). However, there is one subtlety in the implementation of two-phase locking under this policy, as employed in DDM. Consider a database that contains logical data items X and Y and copies Xa, Xb, Yc, Yd. T1 is a transaction that reads X and writes Y; T2 is a transaction that reads Y and writes X. The following execution (which obeys the above specified policy) is incorrect because the multiple copies do not behave like a single logical data item.

R(T1, Xa) --> d fails --> W(T1, Yc)

R(T2, Yd) --> a fails --> W(T2, Xb)

Part of the problem arises from the fact that the site coordinating T1 learns of site d's failure before that of site a; whereas the site coordinating T2 learns of the two sites' failures in an opposite order. A reliability algorithm that eliminates the incorrect behavior is discussed in [GOOD83]. DDM implements a more optimized version of the algorithm that makes use of the site status monitoring protocol proposed in [WALT82] for detecting site failures, and a primary site mechanism for propagating site failure information. (There are well-defined rules for succession to primacy if the current primary site fails.)

Incremental Site Recovery

When a failed site recovers, it is necessary to bring it up-to-date before it can participate in new transaction processing. This is especially the case when data is replicated and when updates continue in the presence of site failures. In early distributed systems that support data replication, such as SDD-1, site recovery is an all-or-none process. To be fully recovered, a site in SDD-1 [HAMM80] is required to process all messages that have been buffered for it since its failure. In [ATTA82], a more incremental approach is suggested. Essentially, the portion of a database that is stored at a site is subdivided into units for recovery. A site can prioritize the recovery of different units and bring them up-to-date one at a time. A unit that is up-to-date can be used immediately to process new transactions independently of the recovery of other units.

Instead of "spooling" all messages destined for a site that is not operational [HAMM80], the recovery algorithm in [ATTA82] reads an up-to-date copy of a recovery unit from another site. A more practical implementation scheme using a differential file approach is employed in DDM. A roll-forward log is maintained at the site of each up-to-date copy, so that only updates that have not been applied to the out-of-date copy need to be transmitted to the remote recovering site. The scheme also allows each operational site to keep track of when a cohort site (i.e., another replication site) fails, so as to determine which portion of the recovery log for a specific recovery unit needs to be sent. In contrast to a local database system, portions of a recovery log must be stored on disk to permit easy access. Portions that will not be needed for rolling forward currently out-of-date copies can be archived to tape. The site for each up-to-date copy maintains sufficient information in its log to determine what portion of the log can be migrated, so archival can be done independently at each up-to-date site without synchronization.

3. Partition Tolerant Distributed Databases

In a separate project, we are developing techniques for ensuring availability of a distributed database in the face of network partitions and other failures. In many applications, such as banking, inventory control, and transportation scheduling, continued servicing of database queries and updates is of extreme importance even when a site is disconnected from all other sites; the application would rather suffer some loss of integrity than lose the business and goodwill of its customers. Concurrency control techniques that guarantee serializability even in the face of network partitions are too conservative and do not provide sufficiently high availability, e.g., only allowing a majority group of sites to continue operation [GIFF79, THOM79] or severely restricting the kinds of transactions that can be performed by each group of sites [SKEE84]. Increased availability can only be achieved by relaxing transaction serializability and integrity requirements, allowing the database to be updated in both (or all) sides of the partition and reconciling inconsistencies after the fact.

Most existing approaches to this problem ([DAVI84, PARK83, GARC83]) assume "discrete" network partitions and reconnections: It is necessary for sites to agree on an "uplist" of accessible sites and possibly switch operating "modes" on each partition or reconnection, and further partitions and reconnections that may occur while switching modes (especially during the "merge phase" after a reconnection) introduce complications that are hard to deal with. If site failures cannot be distinguished from network partitions (and it has not been convincingly established that they can), then every site failure or recovery must be treated as a "partition" or "reconnection" that requires a change of mode. The approach we are taking (which was originally postulated in [BLAU83]) is for a site to operate in a single "mode" regardless of which and how many other sites it can communicate with. Each site is in effect always "partitioned" from the rest of the system, and continuously reconciles its database copy as it receives notification of database updates from other sites. Site failures and network partitions require no special treatment, because they manifest themselves only as very long communication delays.

For simplicity, we assume a fully replicated distributed database. (This is an assumption that we will relax at a later date, allowing partial replication in order to accommodate sites with limited storage capacity.) In addition to a copy of the application database, each site maintains a history of all database updates it has received so far. Updates issued by users and application programs "commit" without any inter-site synchronization. Sites are informed of updates only after the fact, with the system ensuring that every site eventually sees every update. At any given time, a site presents its users and application programs with the "best" possible view of the database that it can, reflecting those updates that it has received so far. Different sites may receive updates in different orders; to ensure eventual mutual consistency of the database copies, timestamps are used to determine a unique order in which updates should be reflected in the database.

Each site runs three software modules; the interaction of these with

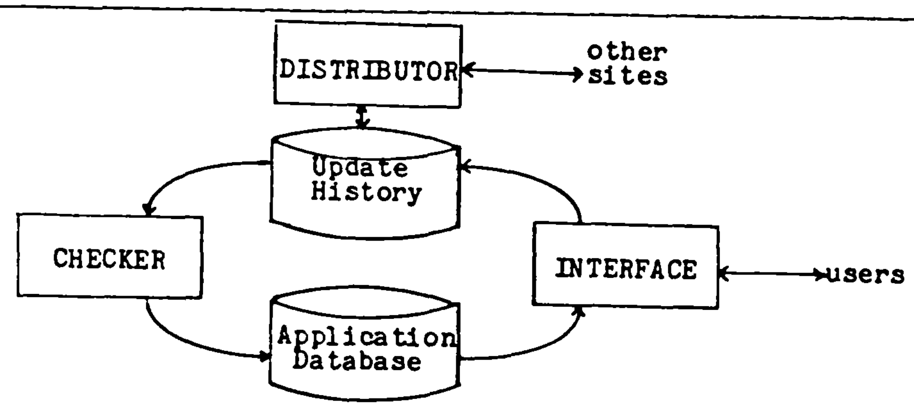


Figure 1. Site Architecture

the application database and the update history is shown in Figure 3.1. The Interface module interacts with users and application programs, querying the database and accepting updates which are appended to the history.

The Distributor module handles inter-site communication and ensures that all sites eventually see all updates. Updates issued by a site's Interface module are sent to all other sites using a reliable broadcast protocol based on [AWER84]. Updates received from other sites are appended to the local copy of the history.

The Checker reads updates from the history and installs their effects into the application database. Updates from different sites will not always be received in timestamp order, but the Checker must ensure that the effect on the database is as if the updates were processed in timestamp order. The

Checker uses a technique called Log Transformation to integrate newly-received updates with any higher-timestamped updates that have already been processed. Conceptually, each higher-timestamped update is rolled back so that the database state now reflects only those updates with timestamps smaller than the newly-received ones; then, the newly-received updates and the updates which were rolled back are executed in timestamp order. The database state now reflects all received updates executed in timestamp order. Log transformation exploits semantic properties of the updates in order to achieve the same effect more efficiently, i.e., with less rolling back and redoing of updates. Updates with non-intersecting read- and write-sets always commute; the same result is obtained regardless of the order of execution, and it is therefore unnecessary to rollback and redo such updates. Updates to the same data item in general do conflict, but again there are many special cases where rollback and redo can be avoided. "Increment" type updates (e.g., deposits to and withdrawals from a bank account) also commute, and can be applied in any order. Or, if an update overwrites (i.e., completely replaces) the value of a data item, an earlier-timestamped update to the same data item can be discarded. Log transformation was first introduced in [BLAU83] for the discrete partition scenario; it has since been refined and adapted for the continuous operation model and is described in [BLAU85]. We are currently implementing log transformation in a prototype distributed system that uses Ingres as the local DBMS at each site.

Because database updates are committed without inter-site synchronization, the application may need to compensate for non-serializable execution. For example, users at two sites may withdraw money from the same bank account, each believing that there is still money remaining in the account, but the total amount withdrawn may be such that the resulting account balance is negative. In this case, the bank might assess a service charge on the overdrawn account and send the customer a letter demanding payment. Compensation is handled by application programs, running in the Interface module, that are triggered [ESWA76] on the occurrence of exception conditions (such as negative account balances), perhaps after some specified delay. Triggered programs may perform actions on the external world (print a letter, or notify an authorized user to investigate the problem), and/or issue further database updates which are processed by the Distributor and Checker as already described.

Meanwhile, we are continuing research into making the system architecture more usable. This primarily involves designing inter-site communication protocols at both the system and application levels. We have developed protocols for: initialization of the distributed system; orderly system shutdown; dynamically adding new sites; removing sites that are malfunctioning or permanently dead; and determining when old updates are no longer needed for log transformation and can be discarded from the update histories. We are also beginning to develop methods for supporting: partial replication of both the application database and the update history; and schema update, which requires some extra mechanism in our system because a site may receive an update to a new field, say, before receiving the schema update that created the field.

At the application level, we are exploring probabilistic ways of combining the integrity advantages of traditional atomic transactions with the availability advantages of our architecture. One promising technique is for a site wishing to update the database to synchronize (using a message similar to a lock request) with as many sites as it can within a reasonable time interval, before issuing the update. Then, if all sites are up and communicating, almost all interactions will be serializable and the need to compensate after the fact will be minimized. When there are communication delays or partitions, interactions will not in general be serializable, but mutual consistency will never be compromised because of the underlying log transformation mechanism. This mixed approach has the advantage that compensation is invoked only when necessitated by failures and partitions, not when all sites are up and communicating normally. It also allows the application to trade integrity for availability, e.g., by adjusting the parameter that determines how long a site should wait for responses to a "lock request", or by making permissible updates at a site (e.g., the maximum amount that can be withdrawn from an account) be some function of how many sites are in communication with the given site. Such probabilistic synchronization among sites will also be useful in minimizing problems introduced by our architecture such as "over-compensation" (on noticing an exception condition, two or more sites concurrently perform compensating actions) and "oscillation" (a site noticing an exception issues a compensating action, but on receipt of another update realizes that the exception has been fixed and tries to compensate for the earlier compensating action).

In summary, our approach to maintaining database availability in the face of network partitions is to commit database updates without inter-site synchronization and to use timestamp ordering to ensure eventual mutual consistency of sites' database copies. The cost of the increased availability is that integrity of the database may suffer, and the burden is on the application to perform appropriate compensating actions. For many applications, this may be a viable alternative to currently popular approaches that enforce strict serializability but limit availability. In addition, applications need not give up all of the integrity advantages of atomic transactions; they may use similar concurrency control techniques in a probabilistic manner to find an appropriate balance between availability and integrity. In the long run, a language and code generation tools will be needed to make it convenient for application designers to use the proposed techniques; this will be the subject of future research.

4. Real Time Databases

The LDM/DDM prototype described in Section 2 has been designed to meet many of the requirements of distributed defense applications. Being implemented in Ada, the LDM/DDM technology has a good prospect for commercialization because of the recent DoD mandate that all mission-critical computer systems use Ada as the implementation language, and because of the increasing data requirements of such systems. On the other hand, LDM/DDM is designed to operate within the environment of a geographically distributed point-to-point network with disk residency of the database. Owing to communication delays in the underlying long-haul network and access latency in the storage devices, LDM/DDM may not provide a level of response adequate for many mission-critical applications that have stringent response requirements. Our current, internal R&D effort aims to develop real time (or near real time) distributed database management capabilities that will provide the orders of magnitude in performance improvement demanded by many mission critical applications. There are two key challenges. First, the DBMS will have to be aware of the deadlines of time critical transactions and organize its workload to meet these requirements. Second, the DBMS must support much higher transaction throughput rates than current systems. We expect to experiment with various concurrency and performance enhancing techniques in the context of LDM/DDM because of its modular architecture. In the long run, we expect the highly reusable and adaptable components of LDM/DDM to facilitate the construction of customized database systems.

One avenue for improving performance is to exploit the characteristics of alternative operating environments. Specifically, we hope to develop alternate system architectures in the context of multi-processor and local area network environments that will minimize the communication and synchronization costs. It has been observed that many transaction management algorithms originally developed for use in a point-to-point communications network can be greatly simplified in the context of a reliable broadcast network [CHAN84]. We also plan to investigate the impact of the availability of large quantities of main memory whereby significant portions of the database can be cached. One technique that can be used to further optimize LDM/DDM's multi-version mechanism is the notion of a database cache/safe system [ELHA84]. This latter scheme exploits the availability of large main memory to cache portions of the database and to avoid ever writing uncommitted updates back to the database proper. Commit processing is greatly expedited by eliminating the need to force write before images. Furthermore, after-images are only force written to a sequential safe which is used for recovering the cache in case of system failures. Committed updates are written back to the database proper as part of the cache replacement routine such that hot spots can be updated many times without being force written. When used with LDM/DDM's multi-version mechanism, the older versions of a data object can be kept in the cache, greatly speeding up the processing of long read-only transactions. When cache replacement requires replacing a version pool page, it will often be the case that the particular old version is no longer needed for recovery or reading purposes and does not need to be written back. Even if still potentially needed, the writing of a number of consecutive version pool pages can be batched, thus avoiding random accesses.

A second optimization strategy we will take is to trade flexibility for performance. Real time applications tend to be stable (i.e., ad hoc transaction are most likely to be absent). If the data, periodicity, and response time requirements of all types of transactions are known a priori, much preanalysis in the areas of access path optimization and concurrency control can be performed offline, greatly reducing the overhead that will have to be incurred at run time. Furthermore, the database applications and the DBMS can be loaded into a single program image, significantly reducing the process switching costs. A specific technique that may potentially be combined with the LDM's multi-version mechanism is the notion of hierarchical database decomposition based on preanalysis of transactions [HSU83]. The transaction analysis decomposes the database into hierarchically related data partitions, such that transactions that write into one partition will only read from the same data partition or from higher level partitions. The concurrency control technique enables read access to higher level data partitions to proceed without ever having to wait. The original scheme presented in [HSU83] requires that transactions within the same class be synchronized via initiation timestamps. Our preliminary investigations suggest that the scheme may be adapted to work with LDM's locking based concurrency control algorithm.

Finally, we plan to exploit characteristics of the application environment in order to simplify the system design. Much of our investigations will focus on transaction management. We will determine if the transaction serializability criterion can be relaxed in order to improve parallelism. We will also examine if the database can be structured in such a way that will facilitate pipelining, thereby reducing the lengths of critical sections (e.g., through the use of non-two-phase locking protocols). Likewise, we will consider if reliability mechanisms can be simplified. For example, if the database is highly volatile, it may be possible to relax the transaction persistence requirement in order to eliminate the logging overhead. Instead, it may be sufficient to take periodic checkpoints and roll back to the latest checkpoint in case of error. In order to identify the simplifications that become possible, we need a more thorough understanding of the specific application requirements. For this purpose, we plan to study a number of representative real time applications, such as computer aided manufacturing, process control, aircraft and missile tracking, military tactical situation monitoring, and real time data acquisition, so as to characterize their requirements.

5. References

[ATTA82] Attar, R., P.A. Bernstein, N. Goodman, "Site Initialization, Recovery, and Back-up in a Distributed Database System," Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks, 1982.

[AWER84] Awerbuch, B., S. Even, "Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network," Proceedings of the Symposium on Principles of Distributed Computing, 1984.

[BERN84] Bernstein, P.A., N. Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984.

[BLAU83] Blaustein, B.T., H. Garcia-Molina, D.R. Ries, R.M. Chilenskas, C.W. Kaufman, "Maintaining Replicated Databases Even in the Presence of Network Partitions," IEEE Electronics and Aerospace Conference (EASCON) Proceedings, Washington, D.C., September 1983.

[BLAU85] Blaustein, B.T., C.W. Kaufman, "Updating Replicated Data during Communications Failures," VLDB Conference Proceedings, 1985.

[CARE84] Carey, M.J., W.A. Muhanna, "The Performance of Multi-version Concurrency Control Algorithms," Computer Sciences Technical Report #550, University of Wisconsin at Madison, August 1984.

[CHAN81] Chan, A., S. Fox, W.T. Lin, D. Ries, "The Design of an Ada Compatible Local Database Manager (LDM)," Technical Report CCA-81-09, Computer Corporation of America, Cambridge, Massachusetts, November 1981.

[CHAN82] Chan, A., S. Fox, W.T. Lin, A. Nori, D. Ries, "The Implementation of an Integrated Concurrency Control and Recovery Scheme," ACM SIGMOD Conference Proceedings, 1982.

[CHAN83a] Chan, A., U. Dayal, S. Fox, N. Goodman, D. Ries, D. Skeen, "Overview of an Ada Compatible Distributed Database Manager," ACM SIGMOD Conference Proceedings, 1983.

[CHAN83b] Chan, A., U. Dayal, S. Fox, D. Ries, "Supporting a Semantic Data Model in a Distributed Database System," VLDB Conference Proceedings, 1983.

[CHAN84] Chang, J.M., "Simplifying Distributed Database Systems Design by Using a Broadcast Network," ACM SIGMOD Conference Proceedings, 1984.

[CHAN85a] Chan, A., R. Gray, "Implementing Distributed Read-only Transactions," IEEE Transactions on Software Engineering, Vol. SE-11, No. 1, February 1985.

[CHAN85b] Chan, A., S. Danberg, S. Fox, T. Landers, A. Nori, J.M. Smith, "A Database Management Capability for Ada," ACM Washington Ada Symposium Proceedings, 1985.

[DAVI84] Davidson, S.B., "Optimism and Consistency in Partitioned Distributed Database Systems," ACM Transactions on Database Systems, Vol. 9, No. 3, September 1984.

[ELHA84] Elhardt, K., R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems," ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984.

[ESWA76] Eswaran, K.P., "Aspects of a Trigger Subsystem in an Integrated Database System," Proceedings of the International Conference on Software Engineering, 1976.

[GARC83] Garcia-Molina, H., T. Allen, B. Blaustein, R.M. Chilenskas, D.R. Ries, "Data-Patch: Integrating Inconsistent Copies of a Database after a Partition," Proceedings of the Symposium on Reliability in Distributed Software and Database Systems, 1983.

[GIFF79] Gifford, D.K., "Weighted Voting for Replicated Data," Proceedings of the Symposium on Operating Systems Principles, 1979.

[GOOD83] Goodman, N., D. Skeen, A. Chan, U. Dayal, S. Fox, D. Ries, "A Recovery Algorithm for a Distributed Database System," Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1983.

[HAMM80] Hammer, M., D.W. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," ACM Transactions on Database Systems, Vol. 5, No. 4, December 1980.

[HSU83] Hsu, M., S.E. Madnick, "Hierarchical Database Decomposition - A Technique for Database Concurrency Control," Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1983.

[PARK83] Parker, D.S., G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," IEEE Transactions on Software Engineering Vol. SE-9, No. 3, May 1983.

[SKEE83] Skeen, D., "Determining the Last Process to Fail," Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1983.

[SKEE84] Skeen, D., D. Wright, "Increasing Availability in Partitioned Database Systems," Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984.

[THOM79] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979.

[WALT82] Walter, B., "A Robust and Efficient Protocol for Checking the Availability of Remote Sites," Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks, 1982.

Concurrency and Recovery in Data Base Systems

C. Mohan

IBM Research Laboratory
K55/281
San Jose, CA 95193
mohan.IBM@CSnet-relay

Introduction

This paper summarizes the work that the author and his colleagues have done on commit coordination, deadlock management, and concurrency control in data base systems. Some of the discussion relates to the design decisions of the R* distributed data base system implemented at the IBM San Jose Research Laboratory [LHMW84, LMHDL84]. The rest of it relates to the author's and his colleagues' work on non-two-phase locking protocols performed mostly at the University of Texas at Austin. Due to space limitations the descriptions are of a brief nature and the reader is assumed to be knowledgeable in the areas of concurrency and recovery.

Here, we suggest that complicated protocols developed for dealing with rare kinds of failures during commit coordination are not worth the costs that they impose on the processing of distributed transactions during normal times (i.e., when no failures occur). (Multi-level) Hierarchical commit protocols are also suggested to be more natural than the conventional two-level (one coordinator and a set of subordinates) protocols, given that the distributed query processing algorithm finds it more natural and easily manageable to decompose the distributed query for execution by a multi-level tree of processes, as opposed to a two-level tree.

As far as global deadlock management is concerned, we suggest that if distributed detection of global deadlocks is to be performed then, in the event of a global deadlock, it makes sense to choose as the victim a transaction which is local to the site of detection of that deadlock (in preference to, say, the "youngest" transaction which may be a nonlocal transaction), assuming that such a local transaction exists.

The R* related work described here was done jointly with B. Lindsay and R. Obermarck.

Commit Protocols and Recovery in R*

Several commit protocols have been proposed in the literature, and some have been implemented [Borr81, HaSh80, Lamp80, LSGGL80, MoSF83, Skee81]. These are variations of what has come to be known as the two-phase (2P) commit protocol. These protocols differ in the number of messages sent, the time for completion of the commit processing, the level of parallelism permitted during the commit processing, the number of state transitions that the protocols go through, the time required for recovery once a site becomes operational after a failure, the number of log records written, and the number of those log records that are synchronously (i.e., force) written to stable storage. In general, these numbers are expressed as a function of the number of sites or processes involved in the execution of the distributed transaction.

Some of the desirable characteristics in a commit protocol are: (1) Guaranteed transaction atomicity always, (2) ability to "forget" outcome of commit processing after a short amount of time, (3) minimal overhead in terms of log writes and message sending,

(4) optimized performance in the no-failure case, and (5) exploitation of completely or partially read-only transactions.

We concentrated on the performance aspects of commit protocols, especially the logging and communication performance during no-failure situations. We have been careful in describing when and what type of log records are written [MoLi83]. The discussions of commit protocols in the literature are very vague, if there is any mention at all, about this crucial (for correctness and performance) aspect of the protocols. We also wanted to take advantage of the read-only property of the complete transaction or some of its processes. In such instances, one could benefit from the fact that for such processes of the transaction it does not matter whether the transaction commits or aborts and hence they could be excluded from the second phase of the commit protocol. This means that the (read) locks acquired by such processes should be released during the first phase. No *a priori* assumptions are made about the read-only nature of the transaction. Such information is discovered only during the first phase of the commit protocol.

Keeping in mind these goals, extensions were made to the conventional two-phase (2P) commit protocol, which works for the case of a single coordinator and a set of subordinates, to a more general, multi-level tree of processes [LHMW84] and two variants, the Presumed Abort (PA) and the Presumed Commit (PC) protocols were developed [MoLi83]. In this more general hierarchical model, the root process which is connected to the user/application acts only as a coordinator, the leaf processes act only as subordinates, and the non-leaf, non-root processes act as both coordinators (for their child processes) and subordinates (for their parents). The root process initiates the execution of the commit protocol, by sending prepare messages in parallel to its subordinates, when the user issues the *commit transaction* command. A non-root, non-leaf process after receiving a prepare message propagates it to its subordinates and only after receiving their votes does it send its combined (i.e., subtree) vote to its coordinator. The type of the subtree vote is determined by the types of the votes of the subordinates and the type of the vote of the subtree's root process. If any vote is a *no* vote then the subtree vote is a *no* vote also (In this case, the process, after sending the subtree vote to its coordinator, sends abort messages to all those subordinates that voted *yes*). If none of the votes is a *no* vote and at least one of the votes is a *yes* vote then the subtree vote is a *yes* vote; otherwise, it is a *read-only* vote. A process voting *yes* force writes a prepare record in the log *before* sending the vote. The writing of this record by a DBM forces out to stable storage the UNDO/REDO log information which allows the actions of the process to be subsequently committed or aborted. This record also includes a list of the locks held by the process. Complications arise in R* due to the fact that more than one process of a transaction may be executing in the same DBM. Since these processes will be sharing some data, to avoid some inconsistencies the processes of a single transaction are made to access some components of the DBM serially (see [LHMW84]).

PA requires acknowledgements (ACKs) from subordinates for only commit messages while PC requires them for only abort messages. This means that in PA all processes that receive commit messages have to force write their commit records *before* acknowledging such messages. In PC, only the root process needs to force write the commit record. Both protocols exclude from the second phase those subordinates that vote *read-only*. In PC, every non-leaf process force writes a *collecting* record, which contains the list of subordinates, before sending the prepare messages to the latter. In PA, there is no need for such a log write. In both PA and PC, the list of subordinates, if any, that voted *yes* is included in the prepare record written by the non-root processes (if the need arises to write that record). In PA alone this kind of information is also included in the commit record written by the root process. In all cases the list of subordinates is needed by the recovery process to know from whom ACKs may have to be demanded while restarting after a site failure. Note that a read-only process does not write any log records in PA.

In summary, note that as far as 2P is concerned, all transactions appear to be completely update transactions and that under all circumstances PA is better than 2P. PA performs better than PC in the case of (completely) read-only transactions (saving the coordinator 2 log writes, including a synchronous one) and in the case of those partially read-only transactions in which only the coordinator does any updates (saving the coordinator a synchronous-write). In both of these cases, PA and PC require the same number of messages to be sent. In the case of a transaction with only one update subordinate, PA and PC are equal in terms of log writes, but PA requires an extra message (ACK sent by the update subordinate). For a transaction with n > 1 update subordinates, both PA and PC require the same number of records to be written, but PA will force n-1 times when PC will not. These correspond to the forcing of the commit records by the subordinates. In addition, PA will send n extra messages (ACKs). PA and PC can coexist in the same system and each transaction could independently choose to use either one of the two protocols. In fact, in R*, both protocols were implemented. More details about PA and PC can be found in [MoLi83].

In cases where the transaction manager (TM) and the data base manager (DBM) at a given site make use of the same file for inserting log information of all the transactions at that site, we wanted to benefit from the fact that the log records inserted during the execution of the commit protocol by the TM and the DBM would be in a certain order, thereby avoiding some synchronous log writes (Currently, in R*, the TM and the DBMs use different log files, but the commit protocols have been designed and implemented to take advantage of the situation when the DBMs and the TM use the same log). For example, a DBM need not force write its prepare record since the subsequent force write of the TM prepare record will force to disk the former. In addition to explicitly avoiding some of the synchronous writes, one can also benefit from the batching effect of more log records being written into a single file. Whenever a log page in the virtual memory buffers fills up we write it out immediately to stable storage.

Given that we have these efficient commit protocols and the fact that remote updates are going to be infrequent, the time spent executing the commit protocol is going to be much less compared to the total time spent executing the whole transaction. Furthermore, site and link failures cannot be frequent events in a well designed and managed distributed system. So the probability of the failure of a coordinator happening after it sent prepare messages, thereby blocking the subordinates that vote *yes* in the in-doubt state until its recovery, is going to be very low.

We have extended, but not implemented, PA and PC to reduce the probability of blocking by allowing a prepared process that encounters a coordinator failure to enquire its peers about the transaction outcome. The extensions require an additional phase in the protocols and result in more messages and/or synchronous log writes even during normal times. To some extent the results of [Coop82] support our conclusion that blocking commit protocols are not undesirable. The designers of Tandem's Transaction Monitoring Facility (TMF) also plan to implement the PA protocol [Hell85]. To handle the rare situation in which a blocked process holds up too many other transactions from gaining access to its locked data, we have provided an interface which allows the operator to find out the identities of the in-doubt processes and to forcibly commit or abort them. Of course, the misuse of this facility could lead to inconsistencies caused by parts of a transaction being committed while the rest of the transaction is aborted. In cases where a link failure is the cause of blocking, the operator at the blocked site could use the telephone to find out the coordinator site's decision and force the same decision at his site.

If we assume that processes of a transaction communicate with each other using virtual circuits, as in R* [LHMW84], and that new subordinate processes may be created even at the time of receipt of a prepare message by a process, for example to install updates at the sites of replicated copies, then it seems reasonable to use the tree structure to send the commit protocol related messages also (i.e., not flatten the multi-level tree into a two-level tree just for the purposes of the commit protocol). This approach avoids the need to set up any new communication channels just for use by the commit protocol. Furthermore, there is no need to make one process in each site become responsible for dealing with commit related messages for different transactions (as in ENCOMPASS [Borr81]).

Just as the R* DBMs take checkpoints periodically to throw away from the data base shadow versions of the modified pages and to force out to disk data and log buffers, the R* TM also takes its own checkpoints. The TM's checkpoint records contain the list of active processes that are currently executing the commit protocol and those processes that are in recovery (i.e., processes in the in-doubt state or waiting to receive ACKs from subordinates). TM checkpoints are taken without quiescing completely all TM activity (This is in contrast with what happens in the DBMs). During site restart recovery, the last TM checkpoint record is read by a recovery process and a transaction table is initialized with its contents. Then the TM log is scanned forward and, as necessary, new entries are added to the transaction table or existing entries are modified/deleted. Unlike in the case of the DBM log, there is no need to examine at any time the portion of the TM log before the last checkpoint. The time of the next TM checkpoint depends on the number of transactions initiated since the last checkpoint, the amount of log consumed since the last checkpoint, and the amount of space still available in the circular log file on disk.

Deadlock Management in R*

Assuming that we have chosen to do deadlock detection instead of deadlock avoidance/ prevention it is only natural, for reliability reasons, to use a distributed algorithm for global deadlock detection.[1] Depending on whether or not (i) the wait-for information transmission among different sites is synchronized and (ii) the nodes of the wait-for graph are transactions (as is the case in R*) or individual processes of a transaction, false deadlocks might be detected. Since we do not expect these types of false deadlocks to occur frequently, we treat every detected deadlock as a true deadlock.

In R*, there is one deadlock detector (DD) at each site. The DDs at different sites operate asynchronously. The frequencies at which local and global deadlock detection searches are initiated can vary from site to site. Each DD wakes up periodically and looks for deadlocks after gathering the wait-for information from the local DBMs. If the DD is looking for multi-site deadlocks also during that detection phase then any information about Potential Global (i.e., multi-site) Deadlock Cycles (PGDCs) received earlier from other sites is combined with the local information. No information gathered/generated during a DD detection phase is retained for use during a subsequent detection phase of the same DD. Information received from a remote DD is consumed during *at most* one deadlock detection phase of the recipient. This is important to make sure that false information sent by a remote DD, which during many subsequent deadlock detection phases does not have anything to send, is not consumed repeatedly by a DD, resulting in the repeated detection of, possibly, false deadlocks. If, due to the different deadlock detection frequencies of the different DDs, information is received from multiple phases of a remote DD before it is consumed by the recipient then only the remote DD's last phase's information is retained for consumption by the recipient. This is because the latest information is the best information.

The result of analyzing the wait-for information could be the discovery of some local/global deadlocks and some PGDCs. Each PGDC is a list of transactions (Note:

1 We refer the reader to other papers for discussions concerning deadlock detection versus other approaches [AgCa85, Ober82].

not processes) in which each transaction, except the last one, is on a lock wait on the next transaction in the list. In addition, the first transaction is known to be expected to send response data to its cohort at another site and the last transaction is known to be waiting to receive response data from its cohort at another site. This PGDC is sent to the site on which the last transaction is waiting if the first transaction's name is lexicographically less than the last transaction's name. Thus, on the average, only half the sites involved in a global deadlock send information about the cycle in the direction of the cycle. In general, in this algorithm only one site will detect a given global deadlock.

Once a deadlock is detected the interesting question is how to choose a victim. While one could use detailed cost measures for transactions and choose as the victim the transaction with the least cost, the problem is that such a transaction might not be in execution at the site where the deadlock is detected. Then, the problem would be in identifying the site which has to be informed about the victim so that the latter could be aborted. Even if information about the locations of execution of every transaction in the wait-for graph were to be sent around with the latter or if we pass along the cycle the identity of the victim, still, there would be a delay and cost involved in informing remote sites about the nonlocal victim choice. This delay would cause an increase in the response times of the other transactions that are part of the deadlock cycle. Hence, in order to expedite the breaking of the cycle, one can choose as the victim a transaction that is executing locally, assuming that the wait-for information transmission protocol guarantees the existence of such a local transaction. The latter is the characteristic of the deadlock detection protocol of R* [BeOb81, Ober82] and hence we choose a local victim. If more than one local transaction could be chosen as the victim then an appropriate cost measure (e.g. elapsed time since transaction began execution) is used to make the choice. If one or more transactions are involved in more than one deadlock, no effort is made to choose as the victim a transaction that resolves the maximum possible number of deadlocks.

Even though the general impression might be that our data base systems release all locks of a transaction only at the end of the transaction, in fact, some locks (e.g., short duration page-level locks when data is being locked at the tuple-level) are released long before all the locks are acquired. This means that when a transaction is aborting it would have to reacquire those locks to perform its undo actions. Since a transaction could get into a deadlock any time it is requesting locks, if we are not careful we could have a situation in which we have a deadlock involving only aborting transactions. It would be quite messy to resolve such a deadlock. To avoid this situation, we permit, at any time, only one aborting transaction to be actively reacquiring locks in a given DBM. While the above mentioned potential problem had to be dealt with even in System R, it is somewhat complicated in R*. We have to ensure that in a global deadlock cycle there is at least one local transaction that is not already aborting for being chosen as the victim.

This reliable, distributed algorithm for detecting global deadlocks is operational now in R*.

Workstation Data Bases

Now that the R* implementation has reached a mature state, providing support for snapshots [Adib80, AdLi80], migration of tables, distributed deadlock detection, distributed query compilation and processing [LMHDL84], and crash recovery, we are currently involved in building a data base system for use on workstations like the IBM PC. The ultimate aim of this effort is to have an R* like system running on a network of mainframes and workstations. In order to be able to benefit from our experiences with the implementations of System R and R*, we are designing the new system (WSDB) from scratch. In the areas of logging and recovery, WSDB will most likely do write-ahead logging (WAL) and in-place updating of data on disk to avoid the overhead of page pointer maintenance. We are exploring improving the performance of the currently known WAL protocols, extending them for record level locking, inventing new ones, avoiding writing compensation log records during rollback of transactions, and providing support for partial rollbacks of transactions.

Other Work

This section discusses the non-R* related work that the author and his colleagues have done at IBM and the University of Texas at Austin.

In [MoSF83], we have introduced the classes of omission and commission faults and studied the advantages of making use of Byzantine Agreement (BA) protocols in conjunction with the R* commit protocols. Our work was done in the context of the Highly Available Systems project and hence we were concerned about Byzantine types of failures also. This was the first serious attempt to apply the BA protocols for realistic situations. In the modified versions of the commit protocols, the identities of all the processes in the subtree are piggybacked onto the messages carrying the votes. If the root's decision is commit then that decision is broadcast using a BA protocol. In contrast, an abort decision is propagated implicitly via timeouts. While the protocols could tolerate Byzantine types of failures during the second phase, we did not find any significant performance advantages in using the BA protocols in the second phase. In [MoSF83], we have briefly outlined an approach for using the BA protocols for protecting the transaction from Byzantine types of failures during its complete execution.

In [HaMo83], we have presented a distributed algorithm for detecting global deadlocks in systems where all interprocess communication is via messages and there is no memory sharing (Note that the R* model is different). This algorithm combines the features of the algorithms of [ChMi82, Ober82]. It has the nice property that it detects only true deadlocks. A comparative study of many distributed data base prototype systems and algorithms can be found in [Moha84a, Moha84b]. The features considered include support for replicated data, and strategies for concurrency control and deadlock management. An annotated bibliography is also included in [Moha84b].

Non-two-phase protocols are intended to be used when data is organized in the form of a directed acyclic graph, where each vertex of the graph is a data item [SiKe82]. These protocols force the transactions to acquire locks on those items in some constrained ways. The edges of the graph could model logical or physical relationships. Our work on locking protocols has concentrated on extending previous work on two-phase and non-two-phase locking protocols to achieve a higher degree of concurrency and at the same time deal effectively with the deadlock problem [Moha81]. We introduced a new lock mode, called INV, with properties fundamentally different from lock modes previously studied and showed how this leads to increased concurrency in some existing very general non-two-phase protocols operating on directed acyclic graphs [MoFS82a, MoFS84] and in a new non-two-phase protocol developed by us [MoFS82b]. The impact of the extensions on deadlock-freedom were studied and simple conditions for resolving deadlocks were given. The INV mode does not grant any access privileges (read/write) to the holder of the lock on the associated data item. It is used solely for concurrency control purposes. Through the introduction of the INV mode, which is compatible with the X (exclusive) mode but not the S (shared) mode, a new principle of the theory of data base concurrency control was enunciated. This principle involves the separation of the effects of commutativity (which relates to serializability) and compatibility (which relates to deadlock-freedom) of data manipulation operations.

Thus we depart from the traditional approaches to concurrency control which do not make this separation.

We have also performed a systematic study of the consequences of allowing lock conversions (from X to S and S to X) in protocols, and shown how this leads to increased concurrency and affects deadlock-freedom [MFKS85, MoFS82b]. In [KeMS82], we have introduced an efficient method for dealing with deadlocks in an existing protocol using partial rollbacks. In addition to extending existing protocols and proposing new ones, we have also extended the existing theory of locking protocols [FuKS81, YaPK79] by including lock conversions and the INV mode in the directed hypergraph model of locking protocols [Moha81]. In so doing, we have obtained very general results concerning serializability and deadlock-freedom properties of all protocols satisfying a natural closure property.

While many of us have been working on non-two-phase locking protocols for many years now (see the references listed in the papers mentioned here), no serious effort has been made to make these protocols resilient to failures. The data base design problem has also not been addressed. In spite of the belief that these protocols are useful for managing concurrent access to ordinary data and that the protocols provide increased concurrency compared to two-phase locking, so far, to our knowledge, none of these protocols have been implemented. It is only in the context of index structures that some similar protocols have been found to be useful (see [Shas85] and the references listed there).

References

Adib80 Adiba, M. *Derived Relations: A Unified Mechanism for Views, Snapshots and Distributed Data*, **Proc. 7th International Conference on Very Large Data Bases**, Cannes, France, September 1981. Also IBM Research Report RJ2881, July 1980.

AdLi80 Adiba, M., Lindsay, B. *Database Snapshots*, **Proc. 6th International Conference on Very Large Data Bases**, Montreal, October 1980.

AgCa85 Agrawal, R., Carey, M. *The Performance of Concurrency Control and Recovery Algorithms for Transaction-Oriented Database Systems*, **Database Engineering**, Vol. 8, No. 2, 1985.

BeOb81 Beeri, C., Obermarck, R. *A Resource Class Independent Deadlock Detection Algorithm*, **Proc. 7th International Conference on Very Large Data Bases**, Cannes, France, September 1981.

Borr81 Borr, A. *Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing*, **Proc. International Conference on Very Large Data Bases**, September 1981.

ChMi82 Chandy, M., Misra, J. *A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems*, **Proc. ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing**, Ottawa, Canada, August 1982.

Coop82 Cooper, E. *Analysis of Distributed Commit Protocols*, **Proc. SIGMOD Int'l Conf. on Management of Data**, June 1982.

FuKS81 Fussell, D., Kedem, Z., Silberschatz, A. *A Theory of Correct Protocols for Database Systems*, **Proc. 7th International Conference on Very Large Data Bases**, Cannes, September 1981.

HaMo83 Haas, L., Mohan, C. *A Distributed Deadlock Detection Algorithm for a Resource-Based System*, IBM Research Report RJ3765, January 1983.

HaSh80 Hammer, M., Shipman, D. *Reliability Mechanisms for SDD-1: A System for Distributed Databases*, **ACM Transactions on Data Base Systems**, Vol. 5, No. 4, December 1980.

Hell85 Helland, P. *The Transaction Monitoring Facility (TMF)*, **Database Engineering**, Vol. 8, No. 2, 1985.

KeMS82 Kedem, Z., Mohan, C., Silberschatz, A. *An Efficient Deadlock Removal Scheme for Non-Two-Phase Locking Protocols*, **Proc. Eighth International Conference on Very Large Data Bases**, Mexico City, September 1982.

Lamp80 Lampson, B. 'Atomic Transactions', Chapter 11 in *Distributed Systems - Architecture and Implementation*, B. Lampson (Ed.), Lecture Notes in Computer Science Vol. 100, Springer Veralg, 1980.

LHMW84 Lindsay, B., Haas, L., Mohan, C., Wilms, P., Yost, R. *Computation and Communication in R*: A Distributed Database Manager*, **ACM Transactions on Computer Systems**, Vol. 2, No. 1, February 1984.

LMHDL84 Lohman, G., Mohan, C., Haas, L., Daniels, D., Lindsay, B., Selinger, P., Wilms, P. *Query Processing in R**, To Appear as a Chapter in Query Processing in Database Systems, W. Kim, D. Reiner, and D. Batory (Eds.), Springer-Verlag, 1984. Also IBM Research Report RJ4272, April 1984.

LSGGL80 Lindsay, B., Selinger, P., Galtieri, C., Gray, J., Lorie, R., Putzolu, F., Traiger, I., Wade, B. "Notes on Distributed Databases", IBM Research Report RJ2571, July 1979.

MFKS85 Mohan, C., Fussell, D., Kedem, Z., Silberschatz, A. *Lock Conversion in Non-Two-Phase Locking Protocols*, IEEE Transactions on Software Engineering, Vol SE-11, No 1, January 1985. Also IBM Research Report RJ3947.

MoFS82a Mohan, C., Fussell, D., Silberschatz, A. *Compatibility and Commutativity in Non-Two-Phase Locking Protocols*, Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, March 1982.

MoFS82b Mohan, C., Fussell, D., Silberschatz, A. *A Biased Non-Two-Phase Locking Protocol*, Proc. Second International Conference on Databases: Improving Usability and Responsiveness, Jerusalem, June 1982.

MoFS84 Mohan, C., Fussell, D., Silberschatz, A. *Compatibility and Commutativity of Lock Modes*, Information and Control, Volume 61, Number 1, April 1984. Also IBM Research Report RJ3948.

Moha81 Mohan, C. *Strategies for Enhancing Concurrency and Managing Deadlocks in Data Base Locking Protocols*, PhD Thesis, University of Texas at Austin, December 1981.

Moha84a Mohan, C. *Recent and Future Trends in Distributed Data Base Management*, Proc. NYU Symposium on New Directions for Data Base Systems, New York, May 1984. Also IBM Research Report RJ4240, 1984.

Moha84b Mohan, C. Tutorial: Recent Advances in Distributed Data Base Management, ISBN 0-8186-0571-5, IEEE Catalog Number EH0218-8, IEEE Computer Society Press, 1984.

MoLi83 Mohan, C., Lindsay, B. *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions*, Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983. Also IBM Research Report RJ3881, June 1983.

MoSF83 Mohan, C., Strong, R., Finkelstein, S. *Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors*, Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983. Also IBM Research Report RJ3882, June 1983.

Ober82 Obermarck, R. *Distributed Deadlock Detection Algorithm*, ACM Transactions on Database Systems, Vol. 7, No.2, June 1982.

Shas85 Shasha, D. *What Good are Concurrent Search Structure Algorithms for Databases Anyway?*, Database Engineering, Vol. 8, No. 2, 1985.

SiKe82 Silberschatz, A., Kedem, Z. *A Family of Locking Protocols for Database Systems that are Modeled by Directed Graphs*, IEEE Transactions on Software Engineering, November 1982.

Skee81 Skeen, D. *Nonblocking Commit Protocols*, Proc. ACM/SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, 1981, pp. 133-142.

YaPK79 Yannakakis, M., Papadimitriou, C., Kung, H.T. *Locking Policies: Safety and Freedom from Deadlocks*, Proc. 20th IEEE Symposium on Foundations of Computer Science, October 1979.

Performance Analysis of Synchronization and Recovery Schemes

T. Härder, P. Peinl, A. Reuter*
FB Informatik
University Kaiserslautern
P.O. Box 3049
D-6750 Kaiserslautern
West Germany

1. Introduction

Seven years ago our group "Data Management Systems" at the University Kaiserslautern (formerly at TH Darmstadt) started a performance measurement project to evaluate the DBMS [UDS] in cooperation with the system´s designers. We identifed very soon the critical functions and performance-determining bottlenecks of that system: locking and logging/recovery. After having studied the special locking problem for UDS, we investigated in subsequent projects a broader range of concurrency control methods (extended locking protocols, versions, optimistic variants, etc.) for different system structures, e.g. centralized DBMS, DB/DC-system, DB-sharing. In addition, various recovery schemes based on both collecting redundant information (logging) and mapping support by the DBMS (shadows) were designed and evaluated with respect to their performance relevant behavior.
To determine the performance of the algorithms, we soon abandoned the idea of pure analytic modelling, especially in the area of CC. Starting mostly with discrete event simulation we refined our models to use trace driven simulation. The simulation results then served for the validation of analytic models in areas where they could be applied appropriately.
We describe here the main aspects of our work performed during the last two years, our current research and our future plans in the area of synchronization and recovery.

2. Empirical Comparison of Concurrency Control Schemes

Research into CC-performance in our group is largely based on trace driven evaluation, as opposed to analytic models or random number driven simulation. The trace input, called "object reference strings" (ORS), is obtained from real DB-applications of different types and sizes. These are medium-sized scientific databases with a workload of long update transactions, bill-of-material applications, and order entry databases with their typical load of short interactive updates and inquiries. The ORSs we have been using until now have been recorded with only one DBMS, the CODASYL-like UDS, but currently we are preparing for getting the same type of data from IMS and ENCOMPASS.

2.1 Methods

The contents of an ORS can be briefly described as follows: There is a beginning-of-transaction (BOT)-record for each transaction, describing the type and the origin of that transaction. Each call to the DBMS´s lock request handler is written to the ORS, describing the type of request, the lock mode and the identifier of the object to be locked/unlocked. In UDS, all lockable objects are pages. Each end-of-transaction (EOT), whether successful or not, is also recorded. All ORS-entries are time-stamped, and the sequence of recording is the temporal sequence of the corresponding events in the original DBMS-execution. For investigating the performance of different CC-schemes, the algorithms are explicitly coded and their behavior under load situations as reflected in the ORSs is analyzed by driving the CC-components with the real lock requests. The basic scenario is shown in Fig. 1 (quoted from [PEIN83]).
For driving a lock-subsystem in a DBMS-like environment, there has to be a scheduler for transaction initialization and resource multiplexing. Depending on the type and speed of underlying hardware we assume, the level of multiprogramming has to be changed on behalf of the scheduler. Each ORS would have been recorded at a fixed level of multiprogramming in its original application. Hence, there must

* Current address: Institut für Informatik, University of Stuttgart, Azenbergstraße 12, D-7000 Stuttgart 1, West Germany

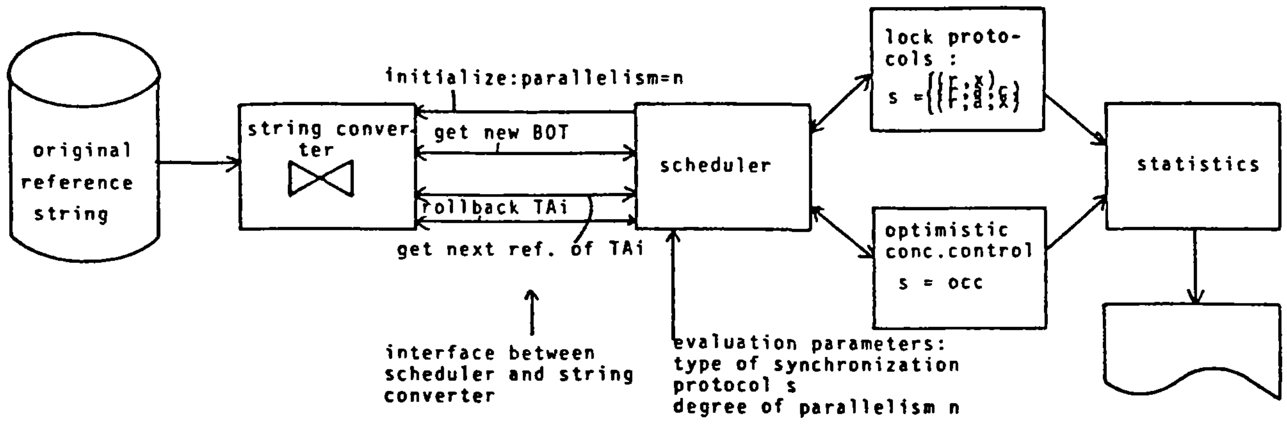

original reference string
string converter
initialize:parallelism=n
get new BOT
rollback TAi
get next ref. of TAi
scheduler
lock protocols :
optimistic conc.control
s = occ
statistics
interface between scheduler and string converter
evaluation parameters: type of synchronization protocol s degree of parallelism n

Fig. 1: Block diagram of the CC-evaluation environment

a way of making the same load look as though it was recorded at an arbitrary level of parallelism. This is achieved by the string converter, which upon initialization can display the ORS with any number of transactions executing concurrently. The string converter is also able to dynamically decrease the level of multiprogramming under thrashing conditions. Transactions that have to be rolled back because of deadlock or validation failure can be returned to the input ORS to be restarted at a later point in time.

This approach to performance analysis of CC-schemes and recovery algorithms is almost unique, because all other results available on this topic are based on either analytic models [FRAN83, GRAY81, MORR84, REUT83, TAY84b] or on discrete event simulation [CARE83, KIES83, RIES77, RIES79]. The problem with analytic models is that in order to keep the formulae tractable, a large number of simplifications have to be applied. Some of these simplifications are questionable from the beginning; e.g. expressing conflict probabilities by a simple term or some sort of distribution function does most likely not reflect the behavior of a real system. A second effect results from the accumulative impact of multiple simplifications. In [REUT83] it is shown that this effect can dastically change the qualitative characteristics of the desired results, which means that it is hard to tell how "real" the phenomena predicted by analytic models are. The most elaborate models are developed in [TAY84a], and some attention is paid to the above-mentioned problems.

Simulation basically suffers from the same difficulties. Although tractability is not an issue, the expressive power of classical simulation systems requires very simple-minded modeling of the database, its object structure, and the transaction reference patterns. Since concurrency control problems do arise only under very specific conditions, the question again is: How real are the results obtained by simulation, i.e. how realistic was the database and transaction model?

The survey presented in [TAY84c] reflects some of these problems, in that the comparison of results from different studies either turns out to be impossible, or reveals contradictory predictions.

We belive that trace driven evaluation allows for a much higher degree of realism than both analytic models and standard simulation techniques because it does not require any idealizations or assumptions that cannot be validated. With these techniques, important phenomena like high traffic locks, hot spots and the influence of lock request patterns cannot be modeled adequately. The drawback is that results based on traces do not yield "general" results, as e.g. analytic models do; they are confined to the given type of application. But since they are very close to a real system's behavior, they are extremely helpful in validating analytic models. If a model's prediction cannot be supported by trace driven evaluation, something must be wrong with the model.

2.2 Studies Based on Traces

We investigated the performance impact of CC in centralized DBMS using two increasingly detailed simulation models. Three variants of locking protocols (standard shared/exclusive and two improved protocols using an additional temporary copy and two versions of an object [BAYE80] to allow for one concurrent writer) were related to two schemes of optimistic CC (OCC), which differ in their scope of validation (backward oriented (BOCC) validates against committed transactions, while forward oriented (FOCC) validates against all active ones with considerable degrees of freedom on how to proceed in case of conflict [HÄRD84a]). Yet all the protocols are suited for page level synchronization, as required by the ORSs and seems promising in the centralized case from the performance perspective. In the first stage [PEIN83] the model consisted of a rudimentary transaction scheduler and the respective CC-modules, being functionally and implementationally roughly equivalent to their counterparts in a real DBMS (Fig. 1). The transaction scheduler dispatches references of as many transactions as specified by the prescribed level of multiprogramming (we examined 1-32) in a round-robin fashion to the CC-module and thereby gathers statistics about the simulation (frequency and cause of deadlock/invalidation, blocking situations, etc.). Performance of the algorithms was rated by the so called effective parallelism, which is roughly equivalent to the total throughput.
Some of the more important observations of this first attempt are the following. In almost all cases the optimistic approach (to be more specific FOCC, since BOCC produced unacceptable results) came out modestly to considerably better than the locking approaches, though OCC achieved this with a higher percentage of reexecutions of failed transactions. The novel locking protocols, as expected, outperformed the standard algorithm, but to a much lower degree than anticipated. Furthermore, they did not obtain the performance of OCC. The study revealed lock conversions (shared to exclusive) to be the most common cause for deadlocks using empirical ORSs. Moreover, special precautions had to be taken to limit the amount of work needed to reexecute aborted transactions and guarantee the execution of long batch transactions. Considerable performance improvements were achieved on the one hand by not immediately restarting failed transactions and on the other hand by employing a load balancing scheme, which actually lowered the degree of parallelism temporarily to facilitate the execution of critical transactions.
However, though the model yielded various valuable insights regarding the dynamic behavior of CC-mechanisms using empirical data, the selected performance measure also exhibits certain weaknesses that prohibit a final judgement on the CC-algorithms. Firstly, the simulation clock (which the effective parallelism is based on) is advanced by references and therefore only roughly related to physical time and secondly, none of the I/O related activities within the DBMS were covered by the first model. Thus, there was no possibility to obtain exact throughput and response time figures. This reasoning gave rise to the development of a considerably more sophisticated and realistic DBMS model, which is currently being evaluated.
For each request, a real DBMS, besides checking for correct synchronization, has to make the required page available in the system buffer. To take into account the delay due to this kind of I/O processing, it was decided to incorporate a buffer manager component and a representation of the DB on external storage media into the refined model. So the DB can be partitioned among several disks, the hardware characteristics of which may be set as parameters in addition to the size of the system buffer. Buffer management in an environment with versions and transaction specific temporary copies, as required by some of the CC-algorithms examined, introduces a considerable amount of algorithmic complexity. This complexity is additionally compounded, when several page types specific to our CODASYL-DBMS are treated separately. We ruled out the propagation of updates before EOT, since that would have required rather complex recovery algorithms. With that ¬STEAL policy (in terminology of [HÄRD83]) applied in the refined model, all that has to be done is the writing to disk out of the after images at EOT (¬FORCE), while the transaction is suspended from scheduling.
The transaction scheduler, as in the preliminary model, tries to keep active a number of transactions up to the predefined multiprogramming level, assigning each one an equal share of the CPU by processing references in round-robin fashion. In the refined model, different types of reference consume different amounts of CPU

time. Moreover, transactions can acquire the CPU only when not waiting for I/O completion. Thus, transaction execution times and other performance measures are directly related to physical time. Since the preliminary study clearly revealed that load balancing and restart policy can strongly influence overall systems performance, in the refined model the restart time of aborted transactions can be effectively controlled by several parameters. In addition there is more flexibility in controlling load balancing. All the CC algorithms listed for the first model are currently being evaluated using the available ORSs with multiprogramming level and buffer size as the primary parameters. Additionally, several restart and load balancing policies are being tested. Besides using throughput and response time as the key performance measures, the different constituents of the simulation model gather comprehensive statistics of their dynamic behavior. Only the more important statistics are briefly listed in the following. The device manager records device utilization and access distributions, buffer manager utilization and I/O traffic. The synchronization modules yield statistics about blocking durations and blocking events, number of deadlocks and their causes, validation overhead, conflicts and their causes, etc. The transaction scheduler computes the average number of nonblocked transactions, restarted transactions and the number and type of references pertaining to each transaction execution. We will come up by mid-1985 with a detailed study of the numerical results of the simulation efforts employing the refined model.
DB-sharing describes the situation where a centralized database is used by a number of DBMSs running on a system of closely or loosely coupled processors. The essential design goal of DB-sharing systems is the approximately linear increase of throughput (by n) of typically simple and short transactions thereby nearly preserving the response times gained in the case of one processor. In addition, a substantial improvement of system availability is required.
The general structure of a DB-sharing system together with its main functional components is sketched in Fig. 2 (quoted from [REUT85]). Synchronization and load control play the critical role for the system's performance goals. Therefore, our modelling efforts were focussed on these functions by reflecting them very accurately in the simulation model. The workload description (ORS) permitted a precise evaluation of synchronization related aspects like conflict frequency, waiting times, deadlocks, etc. Some form of load control using transaction types, locality of reference, etc. was necessary to gain reasonable results.

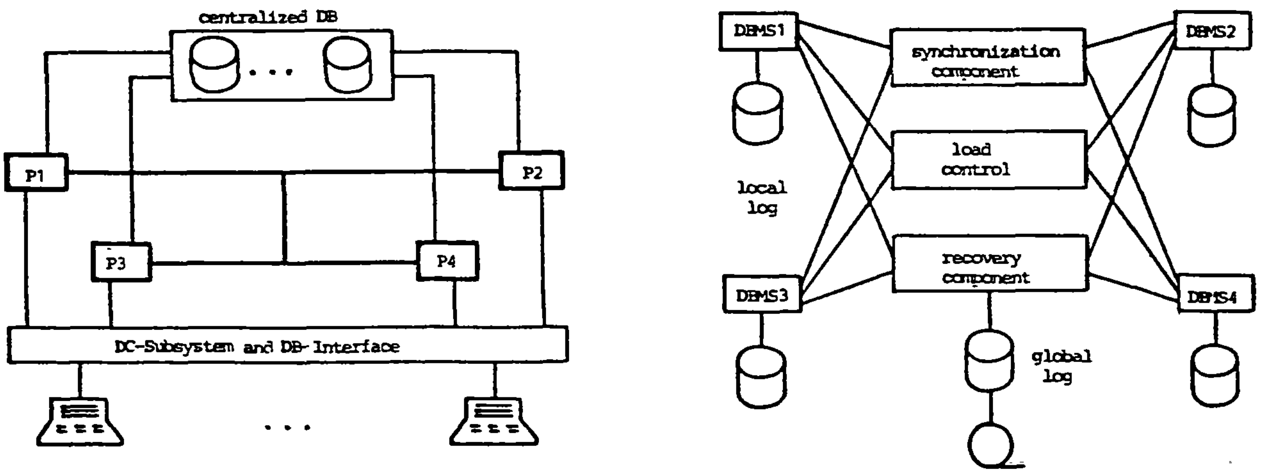


Fig. 2: DB-Sharing: system structure and functional components

We have currently implemented an IRLM-like (IMS Resource Lock Manager) locking protocol which supports a two-processor DB-sharing. It is essentially a token-ring approach where the IRLM attached to a processor can only decide upon global lock requests in its master phase when it posesses the token (buck). A copy of the global hash table (GHT) summarizing the current lock state of the entire system is used by each IRLM to grant as many as possible locks locally thereby minimizing the waiting times for synchronization requests. Details can be found in [REUT85].

In our simulation we tailored the CC-protocol to the needs of a CODASYL-system by introducing a special locking hierarchy and to the particularities of UDS by distinguishing various page types and by controlling to some degree the locality of DB-processing [HÄRD85].
The results were satisfactory as far as throughput improvement is considered (1.6 - 1.8). Response times, however, increased often by a factor of 2 and more compared to the case of one processor. They are critically dependent on the length of the master phase which determines the average (synchronous) wait time for a global lock request. Reducing the master phase, on the other hand, provokes increasing communication costs (frequency of buck exchange). Transaction parallelism suffers from too short master phases since only a limited number of transactions can be serviced within such a period.
Based on the same techniques, we are now starting to investigate different classes of synchronization algorithms for shared database systems. The IRLM-like approach described above is not extendible to more than two nodes, so better protocols are sought for the general n-processor database sharing. There is an initial investigation of algorithms derived from distributed DBMS synchronization mechanisms [REUT85], which we will use as a framework for our performance analyses. It classifies algorithms w.r.t. the following criteria: centralized vs. distributed control; snychronous/asynchronous/optimistic protocols; software vs. hardware implementations. Since in shared DBMSs there is an extremely critical throughput/response time tradeoff due to interprocessor communication and the level of multiprogramming, these evaluations must comprise a detailed mapping of events onto a simulated physical clock which is the same for all nodes in the shared system. Hence, we will have to combine our trace driven approach for the protocol part with standard simulation techniques for modelling a CPU, a communication subsystem, an I/O-subsystem, etc. First results pertaining to a centralized, hierarchical locking scheme are expected by mid-1985.

2.3 Studies based on random numbers

Another area of research is the integration of a DBMS into a general purpose operating system. In particular, the cooperation between application and DBMS processes as well as the synchronization of multiple DBMS processes and their influence on throughput and response time are investigated. The first problem essentially concerns the number of process switches and their inherent overhead which accompanies each DML-request. The second problem occurs when multiple DBMS processes called servers are used. Since now more than one preemptible server is accessing the global system tables (GST) like system buffer (SB), lock tables, etc. high traffic access to these resources has to be synchronized. For performance reasons, a direct protocol between the servers has to be applied (using compare-and-swap-like instructions). Since the GST-elements are typically hot spot data, there happens to be a lot of lock contention. When a server holding a high traffic lock is preempted due to page wait or time slice runout, there is the danger of establishing a convoy in front of the resource [BLAS79].
The process structures for the multiple DBMS servers are shown in Fig. 3. n application processes can cooperate during each DML-call with one of m servers

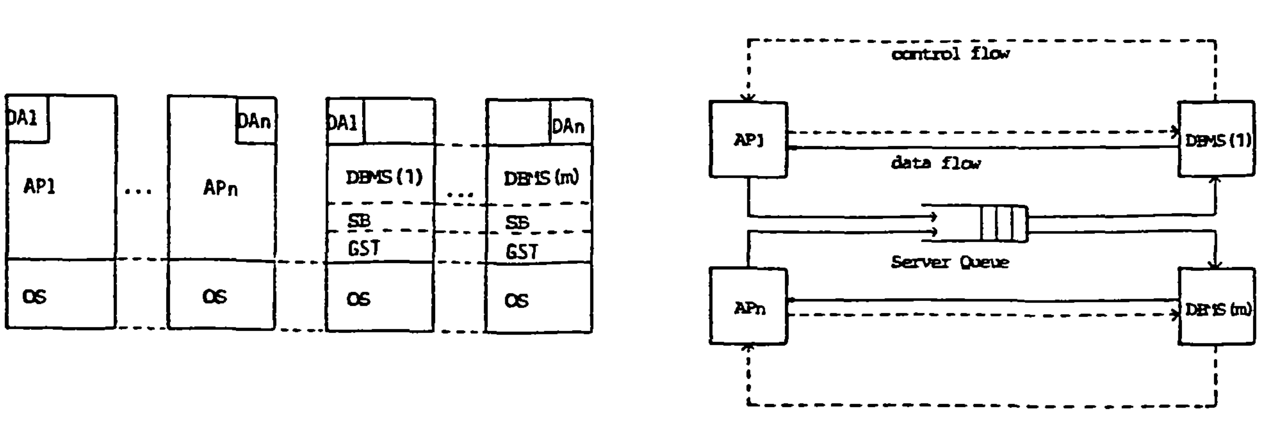


Fig. 3: Multiple DBMS servers: process structure and communication model

(n > m). Data exchange is direct via DAi. Global resources (GST, SB) are in shared virtual memory segments. The various processes can be executed by k tightly coupled processors (k < 4). In order to save process switches, the communication AP → DBMS is performed using a common server queue from which a free server fetches the next AP request.
Our first simulation study reported in [HÄRD84b] was restricted to one processor (k=1) and used a very simple workload model. The AP model consisted of a cyclic process which requested in each cycle a DML-service with a random number of logical page references and physical page fetches. The DBMS server model was fairly detailed providing 4 different latches (hot spot elements) per page access. This event driven simulation model was entirely based on random numbers, that is, DML generation, lock request, page fetch, length of DML request, etc. were all handled by random number generators with suitable bounds.
The experimental results have shown that in a well balanced system the average number of process switches per DML can be reduced from 2 to close to 1. The formation of convoys or waiting times due to critical preemption turned out to be no severe problem. However, process switching overhead in general (synchronous I/O) consumed a huge share of the CPU resources (about 50%).
We reformulated our simulation model for the multi-processor case using again the simplistic AP model and random numbers in each decision situation. The goal of this study was the evaluation of hot spot data accessed by different processors. In this situation other locking protocols may be used (spin lock as opposed to suspend lock in the one-processor case). Furthermore, a throughput and response time prediction should be achieved for the k-processor case with given MIPS-rates. We felt that our modelling efforts and the obtained experimental results were not accurate enough for this problem description.
For this situation, it seems to be important to have the notion of a transaction in the workload model. The execution models for BOT and EOT (Begin/End of TA) within the server are quite different from normal DML execution. Read and write statements should also be treated differently to gain greater accuracy in the simulation model.
Therefore, we are currently designing - in a third approach - a simulation model for the n/m/k-case where the workload model is derived from ORSs (described in 2.1). A server has incorporated four different execution models (BOT, EOT, read, write). The length of a DML statement (page references) is determined by the ORS. According to these references, simulation of lock and buffer management - as essential system components - can be precisely performed. Critical sections protected by latches should also be described by parameters like length, frequency of access, and location within the server model to adapt to the actual needs of simulating a particular system.

3. Empirical Comparison of Recovery Schemes

Before investigating the performance of CC-schemes, a major research topic in our group was DB-recovery. We started by devising some novel recovery algorithms [REUT80, HÄRD79] and then tried to identify basic concepts for classifying and describing the implementational techniques for DB-recovery. The result of this line of investigation has been published in [HÄRD83], from where Fig. 4 is quoted. It shows the classification tree for all conceivable logging and recovery schemes, based on four key concepts: propagation, buffer replacement, commit processing, and checkpointing. Details can be found in [HÄRD83]. In this paper it is argued that the classification criteria have not been established arbitrarily; they rather allow for a fairly precise characterization of performance and other properties of all algorithms belonging to the same class. This was supported by some very detailed performance analyses of ten different recovery schemes. Comparison is based on a fairly simplistic performance measure, namely number of I/O-operations for logging and recovery purposes between two subsequent system crashes, and the results are derived from analytic (mean value) models. For three algorithms we used the trace driven evaluation as a validation method, but investigating all algorithms with this technique was impossible. It turned out that coding all the logging- and recovery components of a given algorithm was much more expensive than coding a, say, r,x-lock manager - even though we did not employ a real-time scale. For the 3 algorithms chosen, the trace driven results supported the model prediction, and a description of this project can be found in

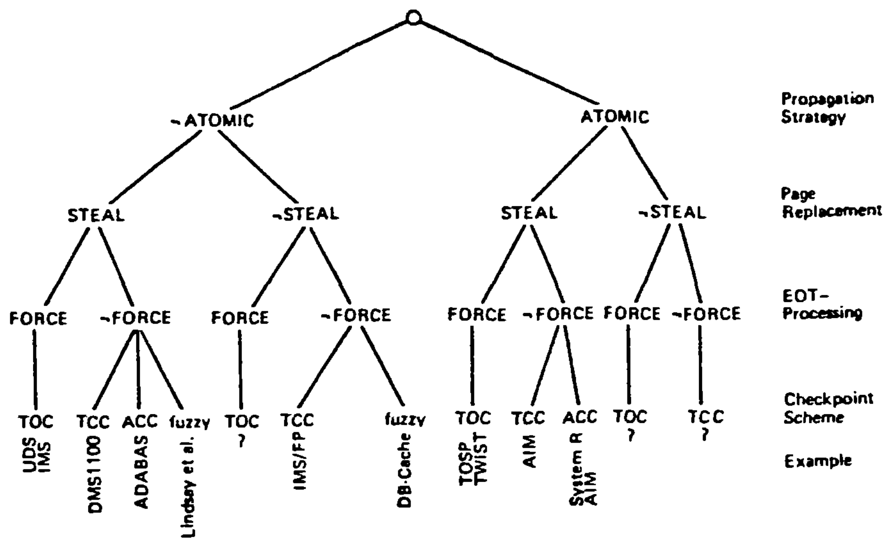


Fig. 4: Classification scheme for recovery techniques

[REUT84]. The most important conclusion to be drawn from these studies is that "advanced" recovery algorithms that aim at reducing the work to be done after crash recovery have to pay a penalty during normal operation that makes them less efficient than the "classical" write-ahead-log-schemes. In particular, it could be shown that
- indirect page mapping (e.g. shadow pages) does not pay;
- page oriented logging schemes are overly expensive;
- forcing modified pages at EOT means high overhead with large buffers.

On the other hand, page oriented schemes have two important virtues: First, they are simple and independent of storage structures; and second, a page log can be used even if the corresponding DB page is corrupted - which cannot be done with an entry log. There is a proposal to optimize page logging by using chained I/O and an appropriate log file format; this so-called DB-cache is described in [ELHA84]. To compare this approach with an optimized entry logging technique, we are currently conducting a trace driven evaluation of both. The results, which will be published by mid-1985, show that even with a number of tricky optimizations page logging is inferior to entry logging and imposes a much lower limit on total system throughput.

4. Future Research
As already mentioned our research is now concentrating on synchronization and recovery aspects in DB-sharing systems. In these systems, sophisticated mechanisms for load balancing with a tight coupling to the synchronization algorithm have to be integrated, too. Furthermore, we are looking into other types of multiprocessor architectures with particular emphasis on
- CC and new transaction concepts for non-standard applications (CAD, office, etc.)
- low level CC aspects concerning asynchronous actions (access path maintenance, replica management, etc.)
- fault tolerance improvement and recovery on single components.

5. Literature

BAYE80	Bayer, R., Heller, H., Reiser, A.: Parallelism and Recovery in Database Systems, ACM TODS, Vol. 5, No. 2, June 1980, pp. 130-156.
BLAS79	Blasgen, M. et al.: The Convoy Phenomenon, ACM Operating Systems Review, Vol. 13, No. 2, 1979, pp. 20-25.
CARE83	Carey, M.: Modeling and Evaluation of Database Concurrency Control Algorithms, UCB/ERL 83/56, PhD Dissertation, University of California, Berkeley, Sept. 1983.
ELHA84	Elhard, K., Bayer, R.: A Database Cache for High Performance and Fast Re-

start in Database Systems, ACM TODS, Vol. 9, No. 4, Dec. 1984, pp. 503-525.

FRAN83 Franaszek, P., Robinson, J.T.: Limitations of Concurrency in Transaction Processing, Report No. RC 10151, IBM Research, Yorktown Heights, Aug. 1983.

GRAY81 Gray, J., Homan, P., Korth, H., Obermarck, R.: A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System, Report No. RJ 3066, IBM Research, San Jose, Feb. 1981.

HÄRD79 Härder, T., Reuter, A.: Optimization of Logging and Recovery in a Database System, Data Base Architecture, Proc. IFIP TC-2 Working Conf., June 1979, Venice, Bracchi, G. and Nijssen, G.M. (eds.), North Holland Publ. Comp., 1979, pp. 151-168.

HÄRD83 Härder, T., Reuter, A.: Principles of Transaction Oriented Database Recovery, ACM Computing Surveys, Vol. 15, No. 4, Dec. 1983, pp. 287-317.

HÄRD84a Härder, T.: Observations on Optimistic Concurrency Control Schemes, Information Systems, Vol. 9, No. 2, 1984, pp. 111-120.

HÄRD84b Härder, T., Peinl, P.: Evaluating Multiple Server DBMS in General Purpose Operating System Environments, Proc. 10th Int. Conf. on VLDB, August 1984, Singapore, pp. 129-140.

HÄRD85 Härder, T., Rahm, E.: Quantitative Analyse eines Synchronisationsalgorithmus für DB-Sharing, Interner Bericht 127/85, FB Informatik, Uni Kaiserslautern, Feb. 1985 (in German).

KIES83 Kiessling, W., Landherr, G.: A Quantitative Comparison of Lockprotocols for Centralized Databases, Proc. 9th Int. Conf. on VLDB, Florence, NOv. 1983, pp. 120-131.

MORR84 Morris, R.J.T., Wong, W.S.: Performance Analysis of Locking and Optimistic Concurrency Control Algorithms, AT&T Bell Laboratories Report, 1984.

PEIN83 Peinl, P., Reuter, A.: Empirical Comparison of Database Concurrency Control Schemes, Proc. 9th Int. Conf. on VLDB, Florence, Nov. 1983, pp. 97-108.

REUT80 Reuter, A.: A fast Transaction Oriented Logging Scheme for UNDO-Recovery, IEEE Transactions on Software Engineering, Vol. SE-6, July 1980, pp. 348-356.

REUT83 Reuter, A.: An Analytic Model of Transaction Interference in Database Systems, Research Report 88/83, University of Kaiserslautern, 1983.

REUT84 Reuter, A.: Performance Analysis of Recovery Techniques, ACM TODS, Vol. 9, No. 4, Dec. 1984, pp. 526-559.

REUT85 Reuter, A., Shoens, K.: Synchronization in a Data Sharing Environment, IBM Research Report, San Jose, Calif. (in preparation).

RIES77 Ries, D.R., Stonebraker, M.: Effects of Locking Granularity in a Database Management System, ACM Trans. on Database Systems 2, Sept. 1977, pp. 233-246.

RIES79 Ries, D.R., Stonebraker, M.: Locking Granularity Revisited. ACM Trans. on Database Systems 4, June 1979, 210-227.

TAY84a Tay, Y.C.: A Mean Value Performance Model For Locking Databases, Ph.D. Thesis, Harvard University, 1984.

TAY84b Tay, Y.C., Goodman, N., and Suri, R.: Choice and Performance in Locking for Databases, Proc. Int. Conf. on Very Large Databases, Singapore, Aug. 1984, pp. 119-128.

TAY84c Tay, Y.C., Goodman, N., Suri, R.: Performance Evaluation of Locking in Databases: A Survey, Res. Report TR-17-84, Aiken Comp. Laboratory, Harvard University, 1984.

UDS UDS, Universal Data Base Management System, UDS-V4 Reference Manual Package, Siemens AG, Munich, West Germany.

The Performance of Concurrency Control and Recovery Algorithms for Transaction-Oriented Database Systems

Rakesh Agrawal

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

Michael J. Carey[†]

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

1. INTRODUCTION

Research in the area of transaction management for database systems has led to the development of a number of concurrency control and recovery algorithms, many of which are surveyed in [Bern81, Bern82, Verh78, Haer83]. Given the ever-growing number of concurrency control and recovery algorithm proposals, the database system designer is faced with a difficult decision: Which algorithm should be chosen? We have been addressing this question in our research, and this paper summarizes our efforts towards arriving at a satisfactory answer.

The organization of the paper is as follows. In Section 2, we describe the structure and characteristics of a simulator that has been used for a number of our concurrency control performance studies. This simulator captures all the main elements of a database environment, including both users (i.e., terminals, the source of transactions) and physical resources for storing and processing the data (i.e., disks and CPUs) in addition to the usual model components (workload and database characteristics). Section 3 contains the results of a recent study in which we addressed assumptions made in past performance studies (by us and others) and their implications, showing how differences in certain assumptions explain apparently contradictory performance results from past studies and how more realistic assumptions could have altered the conclusions of some of the earlier studies. In Section 4, we present the results of an earlier study which compared the performance of a number of alternative concurrency control algorithms. Section 5 describes the results of a performance and storage efficiency comparison of several multi-version concurrency control algorithms. Section 6 summarizes the results of an evaluation of a number of alternative deadlock resolution strategies. In Section 7, we present several parallel recovery algorithms and the results of a study of their performance. Finally, in Section 8, we discuss the results of a study of several integrated concurrency control and recovery algorithms. Section 9 points out some directions for future research in this area.

2. A SIMULATION MODEL

Our simulator for studying the performance of concurrency control algorithms is based on the closed queuing model of a single-site database system shown in Figure 1 [Care83b, Care84a]. Underlying this logical model are two physical resources, the CPU and the I/O (i.e., disk) resources. The physical queuing model is depicted in Figure 2. Table 1 describes the simulation parameters for the model. The model actually supports workloads with two classes of transactions, and each class has its own set of values for the second group of parameters in Table 1. A class probability parameter determines the mix of transaction classes in the workload.

3. CONCURRENCY CONTROL MODELS — ASSUMPTIONS AND IMPLICATIONS

One of our most recent studies of concurrency control algorithm performance [Agra85c] focused on critically examining the assumptions of previous studies and on understanding their apparently contradictory results. For example, several of our own studies have suggested that algorithms that use blocking to resolve conflicts are

[†]Research partially supported by National Science Foundation Grant Number DCR-8402818, an IBM Faculty Development Award, and the Wisconsin Alumni Research Foundation.

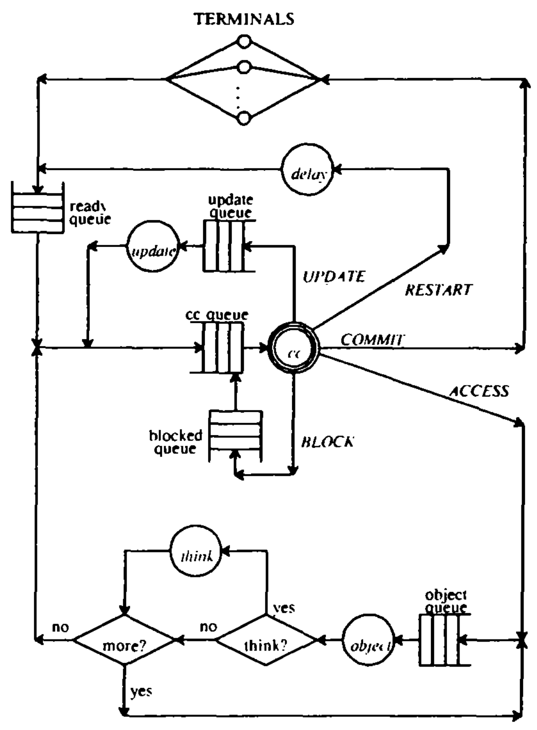


Figure 1: Logical Queuing Model.

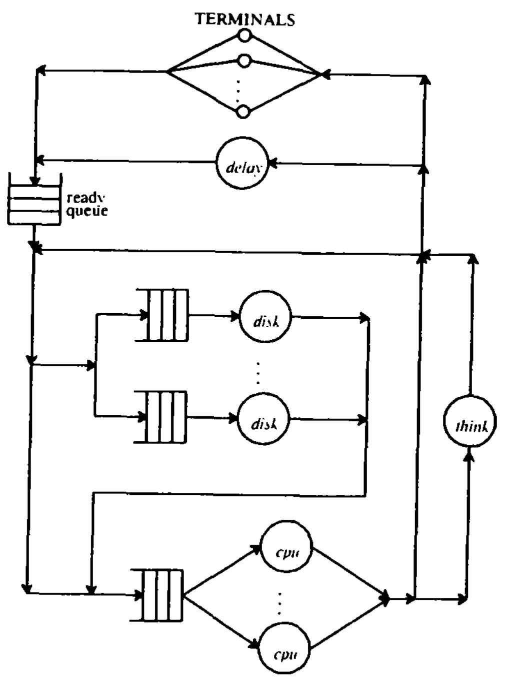


Figure 2: Physical Queuing Model.

Parameter	Meaning
db_size	number of objects in the database
gran_size	number of objects in a concurrency control granule
tran_size	mean transaction size
tran_dist	transaction size distribution (fixed, uniform, exponential)
tran_type	type of transaction (random or sequential accesses)
write_prob	Pr(write X | read X) for transactions
int_think_time	mean intra-transaction think time (optional)
num_terms	number of terminals
mpl	multiprogramming level
ext_think_time	mean time between transactions (per terminal)
restart_delay	mean transaction restart delay (optional)
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
startup_io	I/O time for transaction initiation
startup_cpu	CPU time for transaction initiation
cc_cpu	CPU time for a basic concurrency control operation
num_cpus	number of CPUs
num_disks	number of disks

Table 1: Simulation Parameters.

preferable to those that use restarts [Agra83a, Care84a], whereas there have been other studies that basically came to the opposite conclusion [Fran83, Tay84a, Tay84b]. The assumptions about system resources that underlie these studies are quite different — our previous studies assumed a database system with highly utilized resources, whereas the studies in [Fran83, Tay84a, Tay84b] assumed what we refer to as *infinite resources*, where transactions progress at a rate independent of the number of concurrent transactions (i.e., all in parallel).

In this study, we selected three algorithms to examine: The first algorithm studied was a *blocking* algorithm, basic two-phase locking with deadlock detection [Gray79]. The second algorithm studied was an *immediate-restart* locking algorithm, in which transactions restart immediately (instead of waiting) in the event of a lock conflict [Tay84a, Tay84b]. Restarted transactions are delayed for one average response time before being resubmitted. The last algorithm studied was the *optimistic* algorithm of Kung and Robinson [Kung81], where transactions are allowed to execute freely and are validated (and restarted if necessary) only after they have reached their commit points. These algorithms were chosen because they represent extremes with respect to when conflicts are detected and how a detected conflict is resolved.

We first examined the performance of the three strategies in the case of very low conflicts. As expected, all three algorithms performed basically alike for both infinite and finite resource levels under low conflicts. We then investigated the performance of the algorithms under a higher level of conflicts, first for infinite resources, then for a varied finite number of CPUs and disks, and finally for an interactive type of workload. Table 2 gives the parameters used for most of these experiments.

Figures 3 through 8 show the throughput results obtained for some of the experiments that we ran. As shown in Figure 3, the optimistic algorithm performed the best for the infinite resource case. The blocking algorithm outperformed the immediate-restart algorithm in terms of maximum throughput, but immediate-restart beat blocking at high multiprogramming levels. The performance of blocking increased to a point, then began to decrease due to excessive blocking. These results concur well with those reported in [Balt82, Fran83, Tay84a, Tay84b] under similar assumptions. Figure 4 shows the results obtained for the same workload when the system's resources consist of 1 CPU and 2 disks. As shown, the blocking strategy performed best in this case[1], followed by immediate-restart, then followed by the optimistic algorithm, which agrees well with the results of [Agra83a, Agra83b, Care83b, Care84a]. Figures 5 and 6 show the results obtained with 10 CPU's and 20 disks, and with 25 CPU's and 50 disks. As is evident, blocking won in the former case, and optimistic performed best in the latter case. In the (10,20) case, the maximum levels of useful disk utilization for the algorithms were in the 45-55% range, and they were in the range of about 25-35% in the (25,50) case. (The disks were the bottleneck for this set of parameters.) Useful utilization is the fraction of a resource used to process requests for transactions that eventually committed (as opposed to being restarted); we found the useful utilization

Parameter	Value
db_size	1000 pages
tran_size	8 page mean readset size
tran_dist	uniform on 4 to 12 pages
tran_type	random access pattern
write_prob	0.25
num_terms	200 terminals
mpl	5, 10, 25, 50, 75, 100, and 200 transactions
ext_think_time	1 second
obj_io	35 milliseconds
obj_cpu	15 milliseconds

Table 2: Simulation Parameter Settings.

[1] The fact that immediate-restart beat blocking at the highest level of multiprogramming in the limited resource case is a side-effect of its restart delay; blocking won for the same multiprogramming level when it was given a restart delay [Agra85c].

□ — blocking ○ — immediate-restart Δ — optimistic

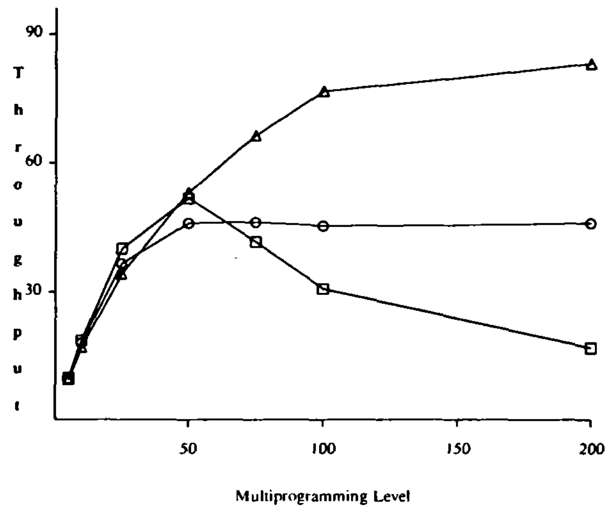


Figure 3: Throughput (Infinite Resources).

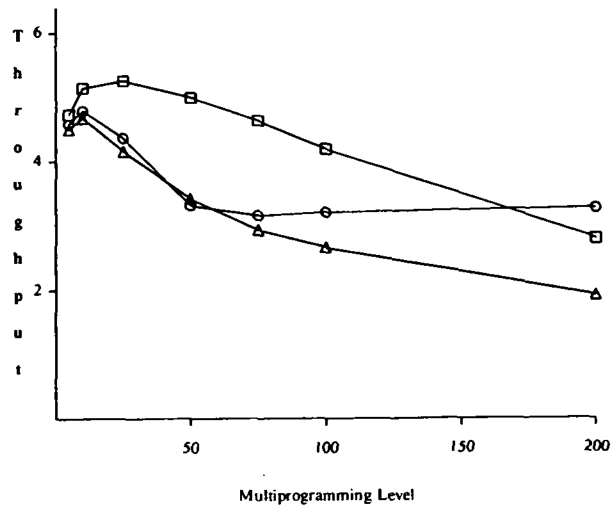


Figure 4: Throughput (1 CPU, 2 Disks).

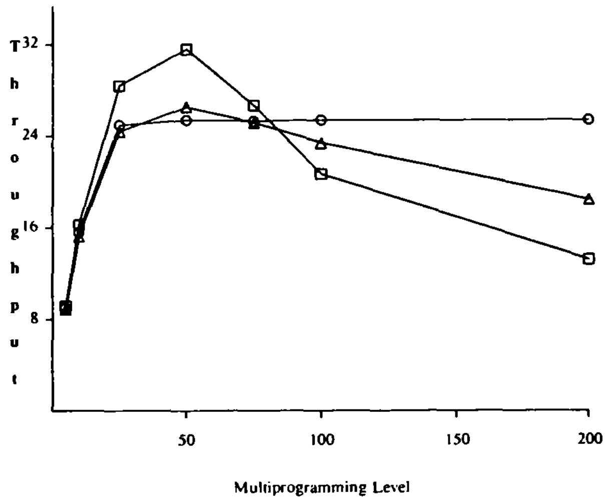


Figure 5: Throughput (10 CPUs, 20 Disks).

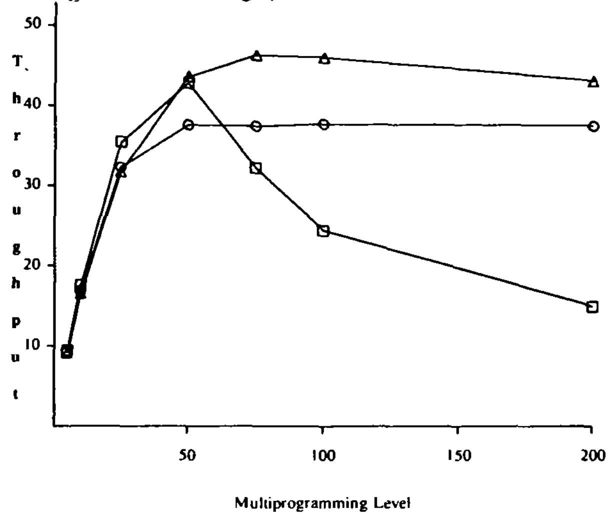


Figure 6: Throughput (25 CPUs, 50 Disks).

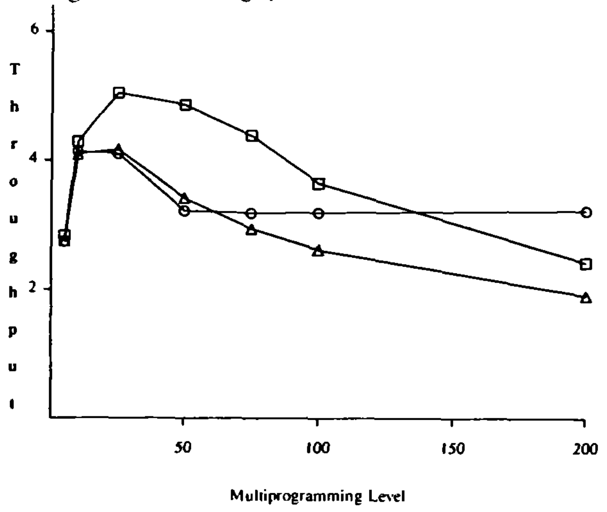


Figure 7: Throughput (1 Sec. Thinking).

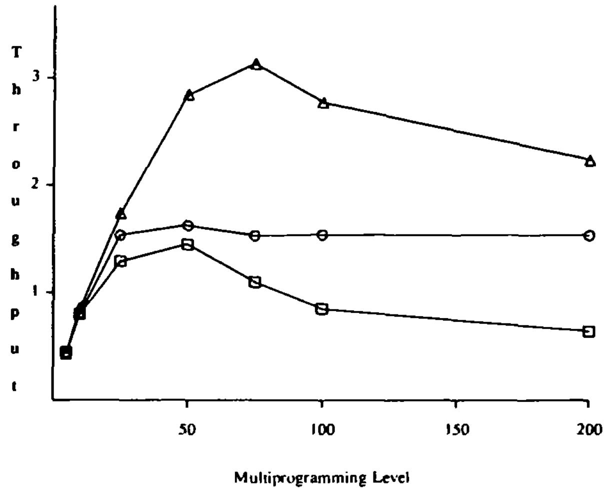


Figure 8: Throughput (10 Sec. Thinking).

and throughput curves to be highly correlated (almost exactly the same shape) in all cases. Finally, Figures 7 and 8 show the results for two interactive workloads, one with a 1-second think time between the read and write phases of transactions (and a 3-second external think time), and the other with a 10-second internal think time (and a 21-second external think time). As shown, the first case yielded results similar to the finite resource case, and the results of the latter test were closer to those of the infinite resource case. The maximum useful disk utilizations in the 1-second case were in the 70-90% range, and they were in the 25-55% range in the latter case.

The main conclusion from this study is that results which appeared to be contradictory are in fact not, being instead due to differences in their underlying resource-related assumptions. For higher resource utilizations, blocking is the conflict-resolution tactic of choice, and for low resource utilizations, restarting is preferable from a performance standpoint. (Such low resource utilizations can be due either to a multiplicity of CPU and disk resources or to an interactive workload with transactions that have long think periods during their execution.) This study also illustrates the fact that capturing the system resources is a necessity for a database performance model if one wishes to study algorithms for real database systems.

4. A STUDY OF ALTERNATIVE CONCURRENCY CONTROL ALGORITHMS

In one of our earlier studies [Care83b, Care84a], we considered the problem of choosing among seven different algorithms — four locking algorithms, two timestamp algorithms, and an optimistic algorithm. The locking algorithms considered were variants of both preclaimed and dynamic locking [Gray79], the timestamp-based algorithms were basic timestamp ordering with and without the "Thomas write rule" [Bern81], and the optimistic algorithm was the serial validation scheme of [Kung81]. Experiments were run for a number of different transaction sizes and mixes, assuming several different concurrency control costs and multiprogramming levels, all while varying the granularity of the database to vary the level of conflicts. These experiments were performed for a high resource utilization case.

All algorithms were found to perform equally well when conflicts were rare. When conflicts were more frequent, the algorithms that minimized the number of transaction restarts were generally found to be superior. In situations where several algorithms restarted about the same number of transactions, those that restarted transactions which had done less work tended to perform the best. Preclaimed locking consistently outperformed the other alternatives in this study[2], and dynamic two-phase locking also did quite well. The optimistic and timestamp algorithms generally performed much worse than the other algorithms under high conflict levels because they wasted the most resources through restarts.

The study in [Care83b] also considered the performance of hierarchical and multiversion concurrency control algorithms. The results for hierarchical versions of locking, basic timestamp ordering, and serial validation (see [Care83a] for algorithm details) were that, unless concurrency control overhead is a significant fraction of the overhead for reading and processing data objects, hierarchical algorithms are of little use under high resource utilization. The next section presents recent results for multiple version algorithms that supercede those of [Care83b].

5. MULTIVERSION CONCURRENCY CONTROL ALGORITHMS

In [Care84b], we studied the performance of three multiversion concurrency control algorithms — Reed's multiversion timestamp ordering algorithm [Reed83], the CCA version pool algorithm, which uses locking [Chan82], and a multiversion variant of serial validation [Care85]. In two experiments, a mix of small update transactions and large read-only transactions were used as the workload; the size of read-only transactions and the relative fraction of the two transaction types were varied. In these experiments, the parameter settings were such that virtually all conflicts were between readers and updaters (not between two updaters). All the multiversion algorithms were found to perform equally well in this situation, and they all outperformed their single version counterparts. The performance gains due to multiple versions were particularly pronounced for the timestamp and optimistic algorithms, as these algorithms performed worse than locking in the absence of multiple versions. In the last experiment, a workload that focused on updater-updater conflicts was tested, and the relative

[2] We assumed that the objects to be accessed are accurately known at transaction startup time — this is not the case in general-purpose database systems, but may be possible in some applications.

performance of the multiversion algorithms was found to be the same as that of their single version counterparts. All of these experiments were run assuming a high resource utilization situation.

We found that although the multiversion locking algorithm did not provide a remarkable increase in overall throughput as compared to single-version locking, it did offer a rather nice form of improvement — it traded a slight increase in the average response time for large read-only transactions, due to occasionally following version chains, for a huge decrease in the average response time for small update transactions. We also looked at the storage characteristics of the algorithms, and all three were similar — the size of the version pool increased with the size of read-only transactions, but its average size never exceeded 10-15% of the total database size in our experiments (even for the largest read-only transaction sizes). We also found that most read requests, more than 95% of them, could be satisfied in a single disk access, and nearly 85% of all transactions executed without needing more than one access for any read request. The cost of reading old versions was found to only slightly impact performance, and its impact was outweighed by the benefits of having multiple versions around to reduce the amount of blocking and restarts.

6. ALTERNATIVE DEADLOCK RESOLUTION STRATEGIES

Given that locking is the concurrency control algorithm of choice over a fairly wide variety of database workloads and system configurations (as described in Section 3), we recently studied the performance of a variety of deadlock resolution strategies [Agra85d]. Among those studied were the following alternatives:

Deadlock Detection. Strategies based on deadlock detection require that a waits-for graph [Gray 79] be explicitly built and maintained. We studied both *continuous detection* and *periodic detection.* We also considered a number of deadlock victim selection criteria, including picking the most recent blocker, a random transaction, the one with the fewest locks, the youngest one, and the one that has used the least amount of resources.

Deadlock Prevention. In deadlock prevention, deadlocks are prevented by never allowing blocked states that can lead to circular waiting. We considered the following set of deadlock prevention algorithms: *Wound-wait* [Rose78], where an older transaction can preempt a younger one; *wait-die* [Rose78], where a younger transaction restarts itself when it conflicts with an older one; immediate-restart [Tay84a], which we described in Section 3; *running priority* [Fran83], where a running transaction preempts any blocked transactions that it conflicts with (i.e., running transactions get priority), and *timeout*, where a blocked transaction times out after waiting longer than some threshold time period.

We investigated the performance of the various deadlock detection and prevention algorithms for two of Section 3's parameter settings, the resource-bound case (1 CPU and 2 disks, non-interactive), and the interactive case with a 10-second internal think time for transactions.[3] A major conclusion of the study is that the choice of the best deadlock resolution strategy depends on the system's operating region. In a low conflict situation, the performance of all deadlock resolution strategies is basically identical. In a higher-conflict situation where resources are fairly heavily utilized, continuous deadlock detection was found to be the best deadlock resolution strategy, and in this situation it is best to choose a victim selection criterion that minimizes transaction restarts and hence wastes little in the way of resources. The fewest locks criterion was found to provide the best performance of those examined. In the interactive case, however, due to low resource utilizations and to long think times during which locks are held, it was found that an algorithm like wound-wait, which balances blocking and restarts, provides the best performance. A deadlock resolution strategy such as immediate-restart, which exclusively relies on transaction restarts, was found to perform relatively poorly in both situations. Continuous deadlock detection consistently outperformed periodic deadlock detection due to higher blocking and restart ratios for periodic detection. Finally, our study also highlighted the difficulty in choosing an appropriate timeout interval for the timeout strategy; even with an adaptive timeout interval, the timeout strategy was never the strategy of choice.

[3] The interactive case was chosen to represent a low resource utilization situation — it is relatively inexpensive to simulate, and its results will also hold for other low utilization cases.

7. PARALLEL RECOVERY ALGORITHMS

In [Agra85b], we presented parallel recovery algorithms based on three basic recovery algorithms, *logging* [Gray79], *shadows* [Lori77], and *differential files* [Seve76], and we evaluated their performance. Although our algorithms were designed in the context of multiprocessor-cache database machines [Dewi81], they may easily be adapted for use in any high performance database management system. Our database machine architecture consists of query processors that process transactions asynchronously, a shared disk cache, and an interconnection device. A processor, designated the back-end controller, coordinates the activities of the query processors and manages the cache. We assume in all of our algorithms that the back-end controller also executes a page-level locking scheduler.

7.1. Parallel Logging

The basic idea of parallel logging is to make the collection of recovery data more efficient by allowing logging to occur in parallel at more than one log processor. When a query processor updates a page, it creates a log fragment for the page, selects a log processor, and sends it the log fragment. The log processors assemble log fragments into log pages and write them to the log disks. The back-end controller enforces the write-ahead log protocol [Gray79]. Details of the algorithm are given in [Agra85a], where it is also shown how recovery from failures may be performed without first merging logs from multiple log disks into one physical log, and how system checkpointing can be performed without totally quiescing the system.

7.2. Shadows

The major cost of the shadow algorithm is the cost of indirection through the page table to access data pages [Gray81, Agra83a]. The indirection penalty may be reduced by keeping page tables on one or more page-table disks (which are different from data disks) and using separate page-table processors for them. Alternatively, the indirection may be avoided altogether by maintaining separate shadow and current copies of data pages only while the updating transaction is active. On transaction completion, the shadow copy is overwritten with the current copy.

7.3. Differential Files

In the model proposed in [Ston81], each data file R is considered a view, $R = (B \cup A) - D$, where B is the read-only base portion of R, and additions and deletions to R are appended to the A and D files (respectively). The major cost overhead of this approach consists of two components [Agra83a] — the I/O cost of reading extra pages from the differential files, and the extra CPU processing cost (for example, a simple retrieval is converted into set-union and set-difference operations). While the number of differential file pages which must be read to process a query depends on the frequency of update operations and the frequency with which differential files are merged with the base file, the parallelism inherent in a database machine architecture may be exploited to alleviate the CPU overhead. In [Agra83b], parallel algorithms were presented for operations on differential files.

7.4. Performance Evaluation Results

The simulator used to evaluate the performance of these parallel recovery algorithms is described in [Agra83b, Agra84]. The details of the performance experiments and a sensitivity analysis of the results can be found in [Agra83b, Agra85b].

The differential-file algorithm degraded the throughput of the database machine even when the size of the differential files was assumed to be only 10% of the base file size. This degradation in throughput was due to the extra disk I/Os for accessing the differential file pages and the extra CPU cycles required for the set-difference operation, and the degradation was found to increase nonlinearly with an increase in the size of the differential files. Since I/O bandwidth is the factor limiting the throughput of the bare database machine [Agra84], these extra disk accesses had a negative impact. The extra processing requirement is not a problem as long as the query processors do not become the bottleneck. However, when the size of the differential files was larger than 10%, the query processors became saturated.

In the case of the "thru page-table" shadow algorithm, it was found that reading and updating the page-table entries could be overlapped with the processing of data pages for random transactions by using more than one page-table processor; there was virtually no degradation in performance. For sequential transactions, when it was assumed that logically adjacent pages could be kept physically clustered, the performance of the "thru page-table" shadow algorithm was very good. In practice, this assumption is difficult to justify, in which case the algorithm performed very poorly due to relatively large seek times. The overwriting algorithm maintains the correspondence between physical and logical sequentiality, but it performed worse than the "thru page-table" shadow algorithm due to extra accesses to the data disks — whereas accesses to the page-table disk in the "thru page-table" shadow algorithm may be overlapped with the processing of data pages, the overwriting algorithm is not amenable to such overlapping.

Overall, parallel logging emerged as the recovery algorithm of choice, as the collection of recovery data could be completely overlapped with the processing of data pages. The bandwidth of the communications medium between the query processors and the log processor had no significant effect on the performance of parallel logging. In particular, the performance of logging did not degrade when log pages were routed through the disk cache; hence, a dedicated interconnection for sending log pages between the query processors and the log processor was found to be unnecessary.

8. INTEGRATED CONCURRENCY CONTROL AND RECOVERY ALGORITHMS

In [Agra83a, Agra83b], we argued that concurrency control and recovery algorithms are intimately related, and we presented six integrated algorithms that perform the tasks of both concurrency control and recovery: log+locking, log+optimistic, shadow+locking, shadow+optimistic, differential file+locking, and differential file+optimistic. The granularity of objects for concurrency control purposes was assumed to be a page. We first explored how the interaction of concurrency control and recovery in the integrated algorithms influences data sharing, update strategies, commit processing, and overall performance [Agra83b]. The performance of these integrated algorithms was then evaluated using a combination of simulation and analytical models. Performance was evaluated assuming a resource-limited situation. Buffer availability was considered as well as CPU and disk overhead. Our evaluation indicated that log+locking had the best overall performance. If there are only large sequential transactions, the shadow+locking algorithm is also a possible alternative. In an environment with medium and large size transactions, differential file+locking is a viable alternative to the log+locking algorithm.

The operation of "making local copies globally available" at transaction commit time was found to be very expensive in the log+optimistic algorithm, resulting in its inferior performance as compared to log+locking. In the log+optimistic algorithm, the log records required for recovery are used as the local copies of updated objects required for validation. Thus, to make the updates of a transaction globally available at commit time, all log pages that were flushed to disk due to buffer size constraints have to be re-read. Worse yet, so do all data pages that are to be updated and that could not be kept in memory due to buffer limitations. In addition, optimistic concurrency control induces more transaction restarts than locking with deadlock detection. Also, nonserializability is detected after a transaction has run to completion with optimistic concurrency control, wasting the entire transaction's processing, whereas deadlock detection is performed whenever a request blocks with locking. Thus, with the optimistic approach, not only are there more transaction restarts, but each restart is also more expensive. These were the main factors responsible for the poor performance of the optimistic combinations as compared to their locking counterparts.

We found that it was more expensive to do transaction undo with log+locking as compared to shadow+locking or differential file+locking. However, the logging combination puts a smaller burden on a successful transaction. Since most transactions succeed rather than abort, log+locking emerged as the better algorithm. The major disadvantage of the shadow combination is the cost of indirection through the page table. The disadvantage of the differential file combination is the overhead of reading differential file pages plus the extra CPU overhead for processing a query.

9. FUTURE WORK

We have described the results of a number of our recent studies of concurrency control and recovery algorithm performance. While we (and, of course, others) have now taken a fairly exhaustive look at the performance of these algorithms for centralized database systems, several interesting problems remain to be addressed.

First, there have been a fair number of studies of distributed concurrency control algorithms, but work remains to be done. In particular, resource issues brought out in [Agra85c] indicate that future studies *must* be based on realistic models of the system resources. Only then can the issue of restarts versus blocking (and its associated deadlock detection message overhead) be resolved satisfactorily for the distributed case. Second, most studies with which we are familiar have been based on either analytical or simulation models. It would be very interesting to see results obtained from studies that stress the concurrency control and recovery mechanisms of actual database systems (or of prototype systems which allow these algorithms to be varied). Such studies would certainly shed light on the validity of the models and results of the performance studies performed to date. Finally, concurrency control and recovery algorithms are being proposed for new application areas such as expert database systems (e.g., Prolog-based database systems), CAD database systems, and transaction-based programming systems for distributed applications. To our knowledge, the performance of alternative algorithms for these and similar emerging areas has yet to be addressed.

REFERENCES

[Agra83a] R. Agrawal and D. J. DeWitt, *Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation*, Computer Sciences Tech. Rep. #497, Univ. of Wisconsin, Madison, March 1983. An expanded version is available as AT&T Bell Laboratories Research Report, Sept. 1984.

[Agra83b] R. Agrawal, *Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance Evaluation*, Ph.D. Thesis, Computer Sciences Dept., Univ. of Wisconsin, Sept. 1983.

[Agra84] R. Agrawal and D. J. DeWitt, "Whither Hundreds of Processors in a Database Machine", *Proc. Int'l Workshop on High-Level Computer Architecture '84*, May 1984, 6.21-6.32.

[Agra85a] R. Agrawal, "A Parallel Logging Algorithm for Multiprocessor Database Machines", *Proc. 4th Int'l Workshop on Database Machines*, March 1985.

[Agra85b] R. Agrawal and D. J. DeWitt, "Recovery Architectures for Multiprocessor Database Machines", *Proc. ACM-SIGMOD 1985 Int'l Conf. on Management of Data*, May 1985, to appear.

[Agra85c] R. Agrawal, M. J. Carey and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications", *Proc. ACM-SIGMOD 1985 Int'l Conf. on Management of Data*, May 1985, to appear.

[Agra85d] R. Agrawal, M. J. Carey and L. McVoy, *The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems*, Computer Sciences Tech. Rep. #590, Univ. of Wisconsin, Madison, March 1985.

[Balt82] R. Balter, P. Berard and P. Decitre, "Why the Control of Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management", *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Aug. 1982, 183-193.

[Bern81] P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys* 13, 2 (June 1981), 185-221.

[Bern82] P. Bernstein and N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control", *Proc. 8th Int'l Conf. on Very Large Data Bases*, Sept. 1982.

[Care83a] M. J. Carey, "Granularity Hierarchies in Concurrency Control", *Proc. Second ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, Atlanta, Georgia, March 1983.

[Care83b] M. J. Carey, *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. Thesis, Computer Science Division (EECS), Univ. of California, Berkeley, Sept. 1983.

[Care84a] M. J. Carey and M. R. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems", *Proc. 10th Int'l Conference on Very Large Data Bases*, Singapore, August 1984.

[Care84b] M. J. Carey and W. Muhanna, *The Performance of Multiversion Concurrency Control Algorithms* (with W. Muhanna), Technical Report #550, Computer Sciences Department, Univ. of Wisconsin, Madison, August 1984.

[Care85] M. J. Carey, "Improving the Performance of an Optimistic Concurrency Control Algorithm through Timestamps and Versions", *IEEE Trans. on Software Eng.*, to appear.

[Chan82] A. Chan, S. Fox, W. Lin, A. Nori, and D. Ries, "The Implementation of an Integrated Concurrency Control and Recovery Scheme", *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, 1982.

[Dewi81] D. J. DeWitt and P. Hawthorn, "A Performance Evaluation of Database Machine Architectures", *Proc. 7th Int'l Conf. on Very Large Data Bases*, Sept. 1981.

[Fran83] P. Franaszek and J. Robinson, *Limitations of Concurrency in Transaction Processing*, Report No. RC10151, IBM Thomas J. Watson Research Center, August 1983.

[Gray79] J. N. Gray, "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979.

[Gray81] J. N. Gray, P. R. McJones, B. G. Lindsay, M. W. Blasgen, R. A. Lorie, T. G. Price, F. Putzolu and I. L. Traiger, "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys* 13, 2 (June 1981), 223-242.

[Haer83] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys* 15, 4 (Dec. 1983), 287-318.

[Kung81] H. T. Kung, and J. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Trans. on Database Syst.* 6, 2 (June 1981).

[Lori77] R. A. Lorie, "Physical Integrity in a Large Segmented Database", *ACM Trans. Database Syst.* 2, 1 (March 1977), 91-104.

[Reed83] D. Reed, "Implementing Atomic Actions on Decentralized Data", *ACM Trans. on Computer Syst.* 1, 1 (February 1983).

[Seve76] D. Severence and G. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", *ACM Trans. on Database Syst.* 1, 3 (Sept. 1976).

[Ston81] M. R. Stonebraker, "Hypothetical Data Bases as View", *Proc. ACM-SIGMOD 1981 Int'l Conf. on Management of Data*, May 1981, 224-229.

[Tay84a] Y. C. Tay, *A Mean Value Performance Model for Locking in Databases*, Ph.D. Thesis, Computer Science Department, Harvard Univ., February 1984.

[Tay84b] Y. C. Tay and R. Suri, "Choice and Performance in Locking for Databases", *Proc. 10th Int'l Conf. on Very Large Data Bases*, Singapore, August 1984.

[Verh78] J. M. Verhofstadt, "Recovery Techniques for Database Systems", *ACM Computing Surveys* 10, 2 (June 1978), 167-195.

DISTRIBUTED COMPUTING RESEARCH AT PRINCETON

Rafael Alonso, Daniel Barbara, Ricardo Cordon
Hector Garcia-Molina, Jack Kent, Frank Pittelli

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-4633
Net address: hector%princeton@csnet-relay

1. Introduction

In this note we briefly summarize our current research in the area of distributed computing. In general terms, our emphasis is on studying mechanisms for reliable distributed data management, focusing on their performance. Our work can be roughly divided into seven categories: use of semantic knowledge, network partitions, recovery algorithms, vote assignments, experimental evaluation of crash recovery mechanisms, data replication, and load balancing and query optimization.

Due to space limitations, we concentrate on describing our own work and we do not survey the work of other researchers in the field. For survey information and references, we refer readers to some of our reports.

2. Semantic Knowledge

In a distributed database, concurrency control mechanisms may introduce unacceptable delays in transactions, mainly due to the relatively large communication times. These same mechanisms may cause transactions to block during a network partition. Both of these problems are undesirable, especially in applications where data availability is critical.

Many of these problems can be avoided if the application is known and understood. For example, consider an airline reservation application, where reservation for flights are being made. A transaction T_1 needs to reserve a seat on a flight F. The data for flight F is held by a computer X, so T_1 executes actions at X. Then, T_1 may have to execute actions at other computers in order to reserve other seats. Now, suppose that a second transaction T_2 wishes to reserve a seat on F. With conventional mechanisms, T_2 cannot commit until the fate of T_1 is known. In other words, the user who submitted T_2 must wait until T_1 completes its actions, and in a distributed environment this can take a significant amount of time. However, since we know the application, we can tell that if T_1 left sufficient seats for F in X, then T_2 does not have to wait for the outcome of T_1. Therefore, T_2 does not have to be delayed and may actually finish before T_1 does.

What we have done in this example is to exploit the application semantics in order to improve performance. Actually, the reliability has also been improved. For instance, suppose

This work has been supported by NSF Grants ECS-8303146 and ECS-8351616, Naval Electronic Systems Command contract N00039-84-C0485, and from grants from IBM and NCR corporations.

that a network partition causes T_1 to block (i.e., to be delayed until repair time). Now T_2 is not forced to block. That is, in spite of the failure and the suspension of T_1, the reservations data for F is still available to transactions like T_2.

Ideas such as the one presented in this example have been used for years, in an ad hoc fashion, in commercial applications where the semantics are relatively simple, e.g., in banking, airline reservations, and inventory control. (See for example the description of IMS Fast Path in [Date81].) In [Garc83b] we formalized this approach and developed a general transaction processing mechanism which lets system administrators enter certain types of semantic knowledge. This knowledge is used by the system to schedule transactions more efficiently and to reduce some of the delays associated with failures. With this mechanism, the semantic knowledge is given to the system in a structured and organized fashion, and does not have to be embedded in the application code as is currently done.

The proposed mechanism has higher overhead than a conventional one and may induce higher delays in transactions that cannot be interleaved with others, so it may not be advantageous in all situations. To better understand the conditions under which using semantic knowledge does pay off, we implemented a detailed event-driven simulator [Cord84a]. With it we identified the "probability of saved conflict" as the most important parameter in deciding whether a semantic based mechanism is advantageous. This is the probability that an object requested by a transaction is locked by another transaction that allows interleaving with it. We discovered that under very general conditions, this probability must be greater than 0.01 before semantic knowledge pays off. Thus, a system administrator can estimate the probability of saved conflict for an application (as discussed in [Cord84a]) and make a preliminary decision as to what mechanism is best. Of course, the number 0.01 is not an exact threshold, but it can be used to discriminate in applications where this parameter is clearly above or below the value.

Our simulation results also indicated that semantic knowledge can improve performance when long lived transactions (LLTs) are present. These transactions have a long duration or access many objects (or both), and are common in certain applications. We have started to investigate mechanisms for processing LLTs, and some of our preliminary observations can be found in [Garc83d]. In this report we also discuss how the database schema and the transactions themselves can be designed in order to improve the performance of the system.

We have also started to develop tools for analyzing transactions and checking if they can be interleaved with others. So far, we have considered transactions that consist of arithmetic actions (e.g., $A \leftarrow A + B - 30$), set actions (e.g., $S \leftarrow S \cup \{x\}$), simple conditional statements, and abort statements. For these, we have developed procedures for automatically checking compatibility of transactions [Cord84b]. These procedures are also useful because they tell us how transactions should be written in the first place to achieve compatibility. We are currently working on generalizing our results; however, as soon as we leave the relatively simple transactions, checking compatibility appears to be substantially more difficult.

3. Partitions in Distributed Database Systems

In a distributed database, a network partition occurs when the system is split into groups of isolated nodes. In order to preserve data consistency, most systems will allow transactions to be run in at most a single group. This means that users at the other nodes will not be able to access the data, even though they may have a copy of it.

In some applications data availability is paramount and halting transactions at some nodes may not be acceptable. Typically, in these applications either partitions are relatively common or they occur at critical moments when access to the data is imperative. Several strategies that provide this higher availability have recently been suggested. They allow more than a single group to process transactions, and the resulting schedule may even be non-serializable (but acceptable for the application). In order to better understand the available options, we have surveyed them and compiled our findings in [Davi84].

The approach we have focused on for database partitions is called Data-patch [Garc83c]. Here each group can update the database during the partition, and then application knowledge is used at repair time to integrate the divergent copies. We are currently modifying Data-patch so that it operates in a more dynamic environment. The goal is to have highly autonomous nodes and to eliminate the need for "clean" integration phases. Under the original Data-patch, when a partition was repaired, the nodes in the newly formed group had to cooperate with each other to arrive at a single integrated database. Any failures during this integration period could cause serious problems. With the new approach, nodes will continuously broadcast copies of their local updates. Each node will then integrate the remote updates with its local view of the database, on a continuous basis.

There are two fairly independent problems to be attacked here. One is the reliable distribution of the update messages. Although there has been some work on reliable broadcasts [Awer84, Chan84], we feel that there are still many open questions. Specifically, we are interested in eventually implementing the protocol on the ARPANET, and in this case the network switches cannot be programmed by the users. Thus, the reliable broadcast protocol must run on the hosts, and managing the broadcast from them, and not the switches, is a challenging task. In [Garc84b] we discuss the difficulties involved, and we suggest various algorithms for the broadcast.

A second problem is the integration of the possibly conflicting updates when they arrive. The basic philosophy will be that each node is autonomous. If a node can communicate with others, then it will try to ensure that the database copies do not diverge. Yet, if a network partition occurs and isolates the node, it will continue to process and update its data as best it can. When communications are restored (even if partially or temporarily), an attempt will be made to merge the databases into a single one. The resulting execution schedule may not, and probably will not, be equivalent to a serial schedule. However, the resulting data will satisfy consistency constraints specific to the application that is running. A more complete discussion of this philosophy and its implications is given in [Garc83c] and [Davi84].

Our preliminary studies show that very often the data can be divided in one of two classes: *image* and *derived*. Image data is associated with a physical object; updates to it must emanate from that physical object. Derived data can be entirely computed from the image data. For example, consider a car rental system. If a reservation cannot be modified or cancelled without first notifying the customer, then we would call this image data. If car assignments can be computed from the reservations and car availability data (both image data), then it would be derived data. It is easier to manage image and derived data in a partitioned system than it is other types of data because the physical objects control updates. In our example, only nodes that can contact a customer will be able to change his reservation, and thus, the types of conflicts that can arrise will be relatively simple. This classification also helps to divide the system in layers, where each layer uses the facilities of the previous one. The lowest layer is the distributor, in charge of reliably distributing each update to the nodes. The next level will implement the image database. The next will implement the derived database.

The system we will build will not be able to handle applications with data that is not either image or derived. However, we suspect that these applications may not be well well suited to a very high availability environment.

4. Recovery Algorithms

A recovery algorithm tries to ensure the continuation of a task after a failure occurs in the distributed computing system. The algorithms that have been proposed so far vary mainly in the types of failures they can cope with. We have concentrated on the so-called Byzantine Agreement (BA) recovery algorithms. The core of these algorithms is a protocol that enables a set of processors, some of which may fail in arbitrary ways, to agree on a common "value." One of the reasons why we became interested in these algorithms is that there has been considerable controversy regarding their use in practice. On the one hand, the developers of the algorithms cited multiple applications for the protocols, but on the other hand "practitioners" complained about the intolerable high message overhead.

In our work we studied when and how BA protocols can be used in general-purpose database processing [Garc84a]. We approached this from a practitioners point of view, and identified the pragmatic implications of BA. In summary, we discovered that some of the applications may have been overstated, but that there was one very important application. It arises in a fully redundant system where data and transaction processing is replicated. In this case, BA must be used to distribute the transactions from the input nodes to the processing nodes.

In [Garc84a] we also sketch how such a fully redundant general-purpose database system could operate using BA. The idea is similar to the *state machine approach* [Lamp84a, Schn82] and hardware modular redundancy [Schl83, Siew82], but we focus on the database and transaction processing issues that arise. Our study also exposed some challenging implementation questions, and thus, this type of system forms the basis for one of the systems we are implementing.

5. Vote Assignments

A number of distributed algorithms require that at most one connected group of nodes be active at a time, and this is achieved by assigning each node a number of *votes*. The group that has a majority of the total number of votes knows that no other group can have a majority, and can thus be active. (It is also possible that no group has a majority and is active.)

The way these votes are dispersed among the nodes can affect in a critical fashion the reliability of the system. To illustrate what we mean, consider a system with nodes A, B, C, and D. Suppose that we assign each node a single vote. This means that, for example, nodes A, B, and C could operate as a group (3 is a majority of 4), but A and B by themselves could not.

Next, consider a different distribution where A gets two votes and the rest of the nodes get one vote. Call this distribution Q and the previous one P. It turns out that Q is superior to P because any group of nodes that can operate under P can operate under Q, but not vice-versa. For example, A and B can form a group under Q (3 is a majority of 5) but not under P. Thus, if the system splits into group A, B and group C, D, there will be one active group under Q but no active group if P is used. So clearly, no system designer should ever use distribution P, even though it seems a very "natural" choice.

We have developed a theory for enumerating and comparing vote assignments [Garc83a]. In it the notion of "assignment Q is superior to P" is formalized. Vote assignments can also be represented by *coteries*, i.e., by an explicit list of the sets of nodes that can be active. For example, the assignment Q can be described by

$$\{A,B\},\{A,C\},\{A,D\},\{B,C,D\}.$$

However, we have shown that coteries are more powerful than vote assignments, that is, there are valid coteries that cannot be represented by votes. We have also shown that for 5 or less nodes, the number of choices (votes or coteries) is surprisingly small (12 choices for 4 nodes; 131 for 5 nodes). Thus, searching for the optimal assignment is possible in these cases.

We have also studied the assignment optimization problem in detail [Barb84a,b,c,d]. We defined two deterministic metrics for comparing assignments or coteries (node and edge vulnerability). We developed heuristics for evaluating these metrics in large systems, as well as heuristics for selecting a good assignment (useful when exhaustive enumeration of assignments is not feasible). We also investigated the impact the network topology has on the assignments, and obtained several theoretical results that simplify the task of choosing an assignment for a given network. We also studied a probabilistic metric (the steady state probability of being operative) and for special topologies we found the conditions for which distributing the votes among the nodes in the system pays off. Although open questions still remain, we feel that we have gained a good understanding of the properties of vote assignments and coteries, and that we now have techniques for selecting them in a practical environment.

6. The Database Crash Recovery Testbed

Many of the distributed mechanisms we are interested in require a reliable database store at one or more nodes. Thus, as a first step in our implementation effort we decided to build such a facility. However, it soon became apparent that this was not a trivial task, mainly because we did not know which of the many crash recovery strategies available we should implement. Several of the strategies have already been implemented (mainly in industry), but we were unsuccessful in our search for explicit performance numbers. Furthermore, all the systems we know of implement a single strategy, so even if we had found numbers we would have faced the tricky task of comparing results from entirely different systems running with different operating systems and hardware. In addition, many implemented systems are hybrids between different strategies (e.g., System R has both shadowed files and logging), so identifying the strengths of each philosophy is complicated.

Crash recovery strategies have been compared in various simulation and analysis papers. Although they were useful to us for understanding the main issues and tradeoffs, we felt that many implementation questions were still unanswered. For example, how much CPU overhead does a mechanism have? How does one go about selecting a block size for the database or the log? For what types of transactions does each mechanism perform best, and *by how much*?

Since these questions seemed interesting in their own right, we decided to implement *two* divergent strategies for crash recovery, and to perform experiments to evaluate and understand them fully. One of the strategies is undo-redo logging; the other is shadow paging. The mechanisms were implemented on a VAX 11/750 running Berkeley Unix 4.1. We have already run a few hundred hours of experiments. The details of the system and our preliminary results are given in [Kent84]. Some very brief and selected highlights follow:

- There is no single best strategy. For instance, logging is superior when updates are short and they access random database locations. Shadowing may be superior if updates modify contiguous pages.
- The CPU overhead of the mechanisms is comparable, although logging usually has slightly less. The CPU overhead of the two phase locking concurrency control algorithm we used was very low.
- The disk scheduler and controller have a strong influence on the overall performance. Some of our results even contradict what simulation and analytical models predict because the latter ignore such implementation "details."
- Contrary to what we expected, implementing shadowing was about as difficult as implementing logging. Shadowing mechanisms in the literature appear simpler because they do not consider updates by concurrent transactions.

Some of the trends observable in our results could have been predicted via simple analytic models (e.g., logging is better for random database probes). Others, however, can only be observed in a real implementation. But even for the ones that could have been predicted, our experiments have provided some firm comparison data points, again, something that simulators and analysis cannot do.

7. Replicated Data

The use of replicated data in a network becomes attractive due to the improved access time possible for read-only transactions. Furthermore, there is increased availability to the users in the case of a network partition. However, update transactions suffer a performance penalty since changes to all copies of the data must be synchronized. We are exploring the tradeoffs involved in deciding whether to replicate a given file, or more generally, determining the number and the location of the copies.

We are building the software required to support file replication, in order to measure its performance under a variety of conditions. These measurements, coupled with information about the performance of the networking software in our experimental systems, will be used to drive a simulation program which will help us analyze the cost-performance of file replication.

We are also exploring other techniques that, like replication, speed up the performance of read-only queries but that may pay a lower cost for concurrency control. For example, caching and pre-fetching are possible design alternatives for decreasing the data access time in a distributed system. We are trying to quantify the performance implications of such mechanisms as compared to data replication under a variety of conditions. We are also studying the benefits of using snapshots and "hints" in distributed systems. That is, in some cases, the semantics of a query may be such that the use of slightly out of date or stale data may be acceptable to the user, and snapshots may be used. Or there may be higher level protocols that detect incorrect information, making the use of hints appropriate. For example, the users of a name-server normally consider the addresses provided by that service as only having a high probability of being correct. Since they must use an application level protocol to ensure that they have contacted the correct object anyway, a name server may make good use of stale data.

8. Load Balancing and Query Optimization

Query optimization in a distributed database system is usually carried out by considering a variety of query plans and estimating the actual costs of executing each of them. In a typical optimizer, those costs consist of a weighted sum of the estimated amount of I/O, computation and message sending required by the plans. However, existing systems usually neglect to consider the effects of the system load on the performance of the plans. In [Alon85] we consider the implications of using load balancing to improve the query optimization process.

The goal of that work was to show that the performance of optimizers could be improved substantially by incorporating load balancing techniques into currently existing optimizers. We studied two different systems, R* [Will82] and distributed INGRES [Ston77], using a variety of tools (simulation and experimentation on a running system) and load metrics (number of active jobs and CPU utilization).

For both R* and distributed INGRES we were able to show that the gains due to load balancing could indeed be large under moderate load imbalances, and for systems that were never saturated. For example, in the R* experiment it was shown that for a typical join the gains were from about 10 to 30 percent, in a system that never became very loaded (always less than 90 percent CPU utilization). For both systems, the implementation approaches suggested involved minimal changes to the system. In the case of INGRES, the cost function was extended so that, instead of examining only transmission costs in order to make the fragment-replicate decision, the load average measurement gathered by UNIX systems was also considered. For R*, one of the two implementations described was to construct alternative plans and choose among them based on load information. The other approach was to invalidate plans at run-time if the load in the systems involved was not within a "normal" range. Finally, in all cases the load metrics used were readily available in the systems studied. Since the statistics used are normally gathered by the operating systems, implementing load balancing does not involve additional overhead for data acquisition, but only for communication among the systems involved. In a high bandwidth broadcast local area network such as the Ethernet, this communication overhead should be negligible unless an extraordinarily large number of hosts are involved in the load balancing scheme.

The results described in [Alon85] show that load balancing strategies are quite attractive for distributed database systems. Furthermore, designers would do well to focus on runtime issues such as the system loads, even at the expense of further work on compile time optimization. The effect of the former has been shown here to be significant, while any new gains due to further refinements in the techniques currently employed for query compilation may be quite marginal.

9. References

[Alon85] R. Alonso, "Query Optimization in Distributed Database Systems Through Load Balancing," *Ph.D. Thesis, U.C. Berkeley, to appear in 1985.*

[Awer84] B. Awerbuch, "Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network," *Proceedings Third Principles of Distributed Computing Systems Symposium,* August 1984.

[Barb84a] D. Barbara and H. Garcia-Molina, "Optimizing the Reliability Provided by Voting Mechanisms," *Proceedings Fourth International Conference on Distributed Computing Systems,* May 1984, pp. 340-346.

[Barb84b] D. Barbara and H. Garcia-Molina, "The Vulnerability of Voting Mechanisms," *Proc. Fourth Symposium on Reliability in Distributed Software and Database Systems,* October 1984.

[Barb84c] D. Barbara and H. Garcia-Molina, "The Vulnerability of Vote Assignments," Technical Report 321, Department of Electrical Engineering and Computer Science, Princeton University, July 1984.

[Barb84d] D. Barbara and H. Garcia-Molina "Evaluating Vote Assignments with a Probabilistic Metric," to appear *Proceedings FTCS 15,* June 1985.

[Chan84] J. Chang and N. Maxemchuck, "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems,* Vol. 2, Num. 3, August 1984, pp. 251-273.

[Cord84a] R. Cordon and H. Garcia-Molina, "The Performance of a Concurrency Control Mechanism that Exploits Semantic Knowledge," to appear *Proceedings Fifth International Conference on Distributed Computing Systems,* May 1985.

[Cord84b] R. Cordon and H. Garcia-Molina, "Checking for Semantic Transaction Compatibility," working paper, November 1984.

[Date81] C. Date, *An Introduction to Database Systems,* Vol. I, Addison Wesley, 1981.

[Davi84] S. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a Partitioned Network: A Survey," Technical Report 320, Department of Electrical Engineering and Computer Science, Princeton University, August 1984.

[Garc83a] H. Garcia-Molina and D. Barbara, "How to Assign Votes in a Distributed System," Technical Report 311, Department of Electrical Engineering and Computer Science, Princeton University, March 1983.

[Garc83b] H. Garcia-Molina, Using Semantic Knowledge for Transaction Processing in a Distributed Database, *ACM Transactions of Database Systems,* Vol. 8, Num. 2, June 1983, pp. 186-213.

[Garc83c] H. Garcia-Molina, T. Allen, B. Blaustein, M. Chilenskas, and D. Ries, "Data-patch: Integrating Inconsistent Copies of a Database after a Partition", *Proc. Third Symposium on Reliability in Distributed Software and Database Systems,* October 1983.

[Garc83d] H. Garcia-Molina, and R. Cordon, "Coping with Long Lived Transactions," working paper, December 1983.

[Garc84a] H. Garcia-Molina, F. Pittelli, and S. Davidson, "Is Byzantine Agreement Useful in a Distributed Database?" *Proceedings Third SIGACT-SIGMOD Symposium on Principles of Database Systems,* April 1984, pp. 61-69.

[Garc84b] H. Garcia-Molina, B. Blaustein, C. Kaufman, N. Lynch, O. Shmueli, "A Reliable Message Distributor for a Distributed Database," CCA Technical Report, September 1984.

[Kent84] J. Kent and H. Garcia-Molina, "Performance Evaluation of Database Recovery Mechanisms," *Proceedings Fourth Symposium on Principles of Datbase Systems,* March 1985.

[Lamp84a] Lamport, L., Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, *ACM Transactions on Programming Languages and Systems,* Vol. 6, Num. 2, April 1984, pp. 254-280.

[Schl83] Schlichting, R.D., and Schneider, F.B., Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems, *ACM Transactions on Computer Systems,* Vol. 1, Num. 3, August 1983, pp. 222-238.

[Schn82] Schneider, F., Comparison of the Fail-Stop Processor and State Machine Approaches to Fault-Tolerance, Technical Report TR 82-533, Department of Computer Science, Cornell University, November 1982.

[Siew82] Siewiorek D. P., and Swarz, R. S., *The Theory and Practice of Reliable System Design,* Digital Press, 1982.

[Skee82a] D. Skeen, "A Quorum-Based Commit Protocol", *Proceedings Sixth Berkeley Workshop on Distributed Data Management and Computer Networks,* February 1982, pp. 69-80.

[Ston77] M. Stonebraker and E. Neuhold, "A Distributed Database Version of INGRES," *Proceedings 1977 Berkeley Workshop on Distributed Data Management and Computer Networks,* 1977.

[Will82] R. Williams et al, "R*: An Overview of the Architecture," *Proceedings of the International Conference on Databases,* Israel, June 1982.

LAMBDA: A Distributed Database System for Local Area Networks

Jo-Mei Chang

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

LAMBDA, a Local Area Network Multiaccess/Broadcast Database System, is an experimental system, designed at AT&T Bell Labs, Murray Hill, for studying distributed database issues in a local area network. It provides high data availability through replicated data. In this paper, we give an overview of the LAMBDA design and demonstrate how LAMBDA simplifies the design of a distributed database system by exploiting the features in local area networks. In particular, LAMBDA employs a novel architecture to relegate most of the complexity arising from communication/site failure to the network level, where it is handled efficiently by one network protocol.

The features provided in LAMBDA are: (1) high data availability – one logical view is provided for all replicated data copies: as long as one replicated copy is operational, the logical data is available for reading and writing, (2) automatic recovery from site failure, (3) uninterrupted transaction processing during site recovery.

1. Introduction

In a broadcast network, either a local area network with broadcast feature or a satellite network, messages can be sent to a group of sites by using one broadcast message. In contrast to a point-to-point network where messages must be sent individually to each site, the broadcast network provides an economic way of communicating among a group of sites. Furthermore, for certain types of broadcast networks, such as Ethernet [METC76], there exists a unique sequence among the messages transmitted in the network: when messages are not lost, if broadcast message m_1 is received at one site before a message m_2, m_1 is received at all sites before m_2. In contrast to a point-to-point network, where messages from different sources can arrive at a site in reverse order in which they were sent, this type of broadcast network provides a more synchronized communication environment [BANI79]. The nature of distributed protocols in a broadcast network environment therefore needs to be reexamined.

A common task in a distributed database system is to send the same information to many or all the sites participating in a given database transaction, for example, transaction update information. Reaching agreement is another frequent task in a distributed system. Transaction commitment, where sites must unanimously decide to commit or abort a transaction, is the best known example of this. Agreement also lies at the heart of many update protocols for replicated copies: here each process must agree on whether a given copy is operational or not. In all the above, an efficient broadcast facility simplies the task while reducing its cost.

One major source of complexity in designing distributed database systems is due to communication failures and site failures. Consequently, database protocols, such as fault-tolerant commit protocols, are carefully designed to guard against these failures. Fault-tolerant protocols tend to be much more complex than their non-tolerant counterparts; in addition, fault-tolerant protocols, performing similar functions, must be introduced in several modules in a typical database system. It is not surprising then that fault-tolerance is a major source of complexity in a distributed database system and is a significant factor in system performance.

Unfortunately, due to the unreliable communication medium, the broadcast network cannot be fully utilized by the application programs: broadcast messages may not arrive at all intended recipients or may be lost altogether. The cost and complexity of fault-tolerance suggests that the first step toward both exploiting broadcast networks and simplifying distributed database design is to provide a *reliable broadcast communication environment that detects failures*. Once the underlying communication network is made reliable and made to detect site failures, the responsibility of the distributed database can be reduced and the system simplified. This is the approach taken in the LAMBDA database system [CHAN83a].

The author's current address is Sun Microsystems, Inc, 2550 Garcia Ave, Mountain View, CA 94303. 415-960-7241, sun ! jmc.

LAMBDA, a Local Area Network Multiaccess/Broadcast Database System, is an experimental research vehicle designed at AT&T Bell Laboratories, Murray Hill, to study the distributed database issues in a local area network. It aims at providing high data availability through replicated data.

In this paper we first describe the features of the reliable broadcast environment provided in LAMBDA, and then discuss how this environment can simplify the design of crash recovery, concurrency control, and transaction commit in a distributed database system. The remainder of the paper is structured as follows. In Section 2, we describe a reliable broadcast network environment – LAMBDA NET – and the network interface to database processes. Section 3 discusses database transaction commit, concurrency control and crash recovery issues. Section 4 gives our conclusion.

2. *A Reliable Broadcast Network*

A number of local area networks provide a broadcast feature [METC76]: broadcast/multicast messages can be received by all sites in the broadcast group. However, some intended receivers may lose broadcast messages because of transmission errors or buffer overflows. For this type of local network, a reliable broadcast environment, called LAMBDA NET, can be established by using a reliable broadcast protocol [CHAN83b] between the application program and the broadcast network.

2.1 *Services Provided by LAMBDA NET*

A system is *k-resilient* if it can continue normal operations in the presence of k or fewer failures. Under the assumption that site failures are benign – that is, sites do not lie about their state – LAMBDA NET employs a strong notion of k-resiliency: it always continues message processing when k or fewer site failures occur and can detect when it is safe to continue processing when more than k failures occur. The system often continues processing in the presence of many more than k failures.

LAMBDA NET provides the following features:

Atomic broadcast: Ensuring two very strong properties:

(1) failure atomicity (all-or-nothing property) – if a message is received by an application program at one site, it will be received by application programs at all operational sites; and

(2) message synchronization – if message m_i is received at one site before m_j, it is received at all operational sites before m_j.

System wide logical clock: Each broadcast message is timestamped with a unique incrementally increasing timestamp. A message with a larger timestamp can not influence the issuing of a message with a smaller timestamp.

Failure detection: When failures or recoveries are detected, a site status message is broadcast. (This message looks like any other broadcast message, although it is implemented differently.) Whenever a site is declared not operational, it will not (successfully) broadcast any messages until a site status message announcing its recovery is sent. Therefore, LAMBDA NET never delivers an obsolete message from a failed site.

Network partitioning announcement: Whenever more than a majority of sites have failed, a network partitioning is announced. This ensures that application programs are aware of the possibility of network partitioning and take precautionary measures, e.g., to block processing completely or to be willing to back out when the communication link is repaired.

When a broadcast message is first received by a site, it is considered *uncommitted* and can not be read by an application process. It remains uncommitted until the network can guarantee the two atomic broadcast properties given above. The message is then placed in a message queue for the intended process and is thereby *committed.* The above features apply only to committed messages. Property (1) under atomic broadcast can now be more rigorously stated: if any site commits the message, then all operational sites commit the message. Message sequencing is another important aspect of atomic broadcasts, simplifying the design of several database protocols. This is especially true when failures and recoveries are synchronized with normal messages. The above features are defined with respect to a broadcast group, which is a static set of sites. LAMBDA NET allows any number of broadcast groups; each one sequences its messages, maintains its own clock, and detects failures. Activities from different groups are not coordinated. A site may belong to any number of broadcast groups.

LAMBDA NET is implemented on top of an Ethernet. The overhead in LAMBDA NET is surprisingly low: it requires only one acknowledgement per broadcast message (instead of one acknowledgement per site per broadcast message). It also requires only limited storage for retransmission messages: for a broadcast group with N sites, the reliable broadcast protocol can operate with only N-1 messages and acknowledgements retained. Note that the above overhead is independent of k, the resiliency parameter. Extra resiliency in LAMBDA NET is obtained by introducing message delays rather than transmitting extra messages. Detailed descriptions and analysis of this protocol can be found in [CHAN83b] and [MAXE84].

2.2 A Database Interface

In the LAMBDA distributed database system, all sites participating in database operations are treated as one large broadcast group. Consequently, the entire database system sees a total ordering over all broadcast messages and has access to an unique global logical clock. A message is usually addressed to only a subset of these sites, a broadcast subgroup. Since there is only one relevant broadcast group, we will let the term "group" refer to the broadcast subgroup.

Transparent to database processes and lying between them and LAMBDA NET is a *message filter.* At each site, the filter process: (1) removes messages not addressed to a process at the site, (2) dispatches messages, in sequence, to the intended database processes, (3) forwards site status messages, in sequence, to database processes. Note that each process receives all messages, including site status messages, in increasing timestamp order.

A simplified failure assumption has been used: process failures result only from site failures. This assumes that processes do not die individually because of errors or user actions. If processes within a site can fail independently, mechanisms can be added to monitor process status, e.g., the message filter process can be devised to monitor critical database processes and to announce detected failures.

The network interface to database processes consists of a number of broadcast primitives and, of course, routines for standard point-to-point communication, the latter not providing message synchronization. The principal broadcast primitives are

1. the atomic broadcast, denoted Brdcst(MSG,Group$_i$).

   Group$_i$ is the broadcast subgroup that should receive message *MSG. Brdcst* is a function returning the subset of Group$_i$ that is defined by LAMBDA NET to be operational when MSG is sent.

2. a *broadcast-all protocol,* denoted Brdcst_All(MSG,Group$_i$).

   In the broadcast-all protocol, a coordinator broadcasts one request to a broadcast group and expects responses from each of the recipients of the request. A response here will either be a reply sent by the recipient or a failure notification sent by LAMBDA NET.

3. a *broadcast-select protocol,* denoted Brdcst_Select(MSG,Group$_i$,M).

   In a broadcast-select protocol, the coordinator broadcasts one request and solicits only M non-null responses. Often M is one. The protocol returns the set of M responses or returns *ERROR* if it becomes clear that the request cannot be satisfied. A null response indicates that the recipient can not respond to the request in a meaningful way. Null responses are often used by processes recovering from a recent failure and is necessary to prevent delaying the coordinator unnecessarily.

3. Distributed Database Design Issues

Atomic broadcast and failure detection are major primitives provided by LAMBDA NET. These are convenient and powerful primitives for distributed systems, and we now demonstrate this by showing how the LAMBDA NET environment simplifies the problems of transaction commit, concurrency control, and crash recovery.

For transaction processing, we assume the TM-DM model of [BERN81]. A transaction manager (TM) and a database manager (DM) reside at each site. The transaction managers supervise user queries and translates them into commands for database managers. Each transaction is executed by a single TM. A database manager maintains that part of the database stored at its site. A DM may be concurrently processing transactions from several TMs.

3.1 Transaction Commit

At the end of processing a transaction, a commit protocol is invoked to either commit or abort the transaction at all participating DMs. The most used protocol is the two-phase commit. Briefly, it is [GRAY78]:

Phase 1: The coordinating TM sends the "prewrite" message to all DMs updating their databases. Each DM prepares to write into the permanent database its changes, acquiring any needed locks and logging recovery data. Each DM responds with "accept" or "reject."

Phase 2: If all DMs accept, then the TM sends "commit" to all DMs; otherwise, it sends "abort."

The major weakness of this protocol is that it is not resilient to coordinator failures: if the coordinator and all DMs that know the decision (i.e., commit or abort) fail, then the protocol is blocked and the transaction can not be completed until some processes recover. Four-phase [HAMM80] and three-phase [SKEE81] nonblocking commit protocols are known, but these are more expensive, by 50 to 100 percent, in time and messages, and much harder to implement. They do have the feature that the transaction can always be correctly completed, without waiting on failed processes, if k or fewer failures occur (where k is the resiliency parameter).

COORDINATING TM:

```
Phase I:   Msgs := Brdcst_All("Prewrite" Group_i);

Phase II:  if all Msgs = "accept"
             then Brdcst("Commit" Group_i);
             else Brdcst("Abort" Group_i);
```

PARTICIPATING DMs:

```
Phase I:   wait for either a prewrite from TM or a failure notice;
           if "prewrite" is received
             then prepare to commit;
                  respond to TM with either "accept" or "reject";
             else abort the transaction;

Phase II:  wait for either a Commit, Abort, or TM failure notice;
           if "commit" received
             then commit the transaction;
             else abort the transaction;
```

Figure 1. A nonblocking 2-phase commit protocol using LAMBDA NET.

Using the LAMBDA NET environment, the simple two-phase commit protocol of Figure 1 provides the same nonblocking property. Let's examine why this protocol works, concentrating first on the second phase. The atomic broadcast feature ensures that if any DM receives a commit message, all operational DMs must receive the message. Clearly the strong all-or-nothing property of atomic broadcast is necessary here. Phase II requires not only the all-or-nothing property but also the message synchronization property, especially, the synchronization of failure notices with other messages that LAMBDA NET provides. If any slave receives a failure notice before a commit message, it is guaranteed that all slaves will receive the failure notice before the commit message. A slave can thus safely decide to abort the transaction.

It is not surprising that atomic broadcast greatly simplifies nonblocking commit protocols: the last two phases of the three-phase commit protocol is essentially implementing atomic broadcast at the database process level [SKEE83]. However, in the case of LAMBDA NET, the atomic broadcast is implemented at the network level, where it is cheaper and its complexity can be hidden from application programmers.

3.2 Replicated Databases

Consider now a database system supporting replicated data. For each logical data item X, there exists a number of physical copies $X_1, X_2, \cdots, X_m$, where $m \geq 1$. For our purposes, the unit of replication is not important. Replication complicates concurrency control and adds a new problem, that of maintaining consistency among copies.

3.2.1 Consistency among Copies. Transaction managers and other software components at a comparable level of the system normally deal with logical data items. Requests to logical item X must be mapped into requests to appropriate copies of X. Similarly, responses from copies of X must be mapped back into a single response. These mappings must preserve the intuitive notion of correctness: the multiple copies of X should behave as if they were a single copy [BERN83].

A popular scheme for maintaining one-copy behavior, which has been proposed for ADAPLEX [GOOD83], distributed-INGRES [STON79], and LAMBDA, is what we will call the *active group* method. To read X, a transaction manager reads from *any operational* copy of X. To write X, a transaction manager writes *all operational* copies of X. By operational copies, we mean copies that include the most recent updates, in contrast to failed or recovering copies. This method yields high read and write availability: as long as one copy is operational, the logical item is available for both reading and writing.

Implementing the active group method requires transaction managers to have consistent copy status information. Consider what may happen if two transactions T_i and T_j have inconsistent views of operational copies. Suppose T_i considers X as having two operational copies X_1 and X_2, and T_j considers X as having only one operational copy X_2. T_i and T_j can simultaneously read and write X: T_i reads from X_1 and T_j writes to X_2. Because the copy status information is itself dynamically changing and replicated, the active group method is difficult to implement in a general network environment.

Consider now implementing the active group approach in the LAMBDA NET environment. The problem of maintaining a consistent view of operational copy status among transaction managers is partially solved by using site status information. Since all processes receive the site status messages in the same sequence, all processes agree on the set of operational sites that have a copy of X. Processes can therefore use the broadcast primitives of LAMBDA NET to reach all operational copies. A broadcast will also reach non-operational copies, such as recovering copies, that reside at operational sites; but these copies can ignore the request or, if a response is required, return a null response.

Figure 2 describes in detail the read-one/write-all protocols used in the LAMBDA database system. A two phase locking protocol [GRAY78] is used to ensure correct execution of concurrent transactions. Depending on the lock status, a read or write request on operational data is answered either "yes" or "no". A recovering copy always returns a null response. (Due to the underlying failure detection facility, a slow response, due to a transaction waiting for a lock, can be differentiated from a failed site).

Let $Group_X$ represent the set of sites that have a copy of X. For a read operation on X, the transaction manager can arbitrarily select an operational site, send its request, and hope that the site has an operational copy. Alternatively, a transaction manager can use a broadcast-select protocol to broadcast the request to $Group_X$ and wait for the first response from an operational copy of X. For a write operation on X, a transaction manager uses a broadcast-all protocol to send the request to $Group_X$. It will receive the set of responses from all operational sites in $Group_X$, from which it can determine the operational copies (they return non-null responses) and verify that all were able to perform the request.

3.2.2 Concurrency Control. A problem arises when the active group approach is used with two-phase locking. Normally locks are held only at the copies that were actually accessed. Hence, if transaction T reads X, it locks only the one copy actually read. This single read lock prevents any other transaction from writing X since to write all operational copies must be locked. If however the copy locked by T fails, then this is no longer true. Lacking locks at any operational copy of X, T has effectively lost its lock on the logical data item X.

Whenever a transaction loses a lock on a logical item before it completes, that transaction has, in effect, violated the two-phase locking constraint. It is easy to construct examples where losing locks in this way leads to nonserializable executions [BERN83]. Note that problems can only arise with lost read locks, not write locks:

Read One protocol:

```
send("read X") to an arbitrary operational site in Group_X;
if (response = "null") {
        Res := Brdcst_Select("read X", Group_X, 1);
                if (Res = "yes")
                        read is successful;
                else read is unsuccessful;
        }
else if (response = "yes")
        read is successful;
else if (response = "no")
        read is unsuccessful;
```

Write All protocol:

```
Msgs := Brdcst_All(write X request, Group_X);
if ("no" ∈ Msgs)
        write is unsuccessful;
else if ("yes" ∈ Msgs)
        write is successful;
else    /*  all Msgs = "null" or "failed"  */
        write is unsuccessful;
```

Figure 2. Protocols for read and write operations.

losing all write locks for an item means that all copies are non-operational and, hence, the item is inaccessible.

A correct two-phase locking strategy to go with active groups is to check all read locks at commit time. If a transaction owns a read lock from a site that has failed since the read lock is granted, the transaction must be aborted. This check is easily performed in LAMBDA NET using the site status information provided. Note that the correctness of this scheme relies on failure notices arriving in the same order at all sites. [BERN83] proves a similar scheme correct.

Although failures pose no problems regarding write locks, recoveries do. If transaction T writes X, it locks all operational copies. This prevents any other transaction from reading (or writing) X since to read (or write) one (or all) operational copy must be locked. If however a copy of X changes its status from recovering to operational, then this is no longer true. This copy is now available for reading and can grant a read lock on the logical item X to a different transaction. Lacking a lock at an operational copy, T has not effectively write-locked the logical data item X. T has to be aborted, unless the recovering copy happens to carry the correct lock. A correct recovery strategy for the active group approach is to require a recovering copy to *inherit* write locks from other operational copies of X before it becomes operational.

So far, we have discussed only one type of concurrency control mechanism: two-phase locking. The LAMBDA NET environment can be used to simplify other types of concurrency control mechanisms as well, e.g., timestamp concurrency control [BERN80]. The timestamps provided by the LAMBDA NET can be used directly as timestamps for transactions. As pointed out in [CHAN83c], if the SDD-1 concurrency control algorithm is used in the LAMBDA NET environment, a number of deficiencies such as the requirement for "send null" and "null write" messages can be eliminated.

3.3 Crash Recovery

When data is replicated, upon recovery, the simplest method is to copy the entire data from an operational copy. However, this method is very expensive. Popular and much cheaper schemes use *logs* for recovery. For each physical copy X_1, a log file, log(X_1), records the sequence of updates (prewrite and commit messages, as described in the commit protocol) that have been applied to X_1. To recover, a recovering copy must identify the set of missing updates in the log of an operational copy of X and apply these updates.

Generally, log-based recovery is very messy because log files at different sites may record updates in different orders. This can happen because each DM may be concurrently processing several transactions and updates from different transactions may be executed in a different sequence at each site. The consequence of this is that it is difficult for a site to concisely identify the updates it is missing. To illustrate the problem, consider the following sequence of updates recorded in two log files log(X_1) and log(X_2), ignoring momentarily the | mark.

log(X_1) = { ..., Prewrite$_j$, Commit$_j$, Commit$_i$, ...}
log(X_2) = { ..., Commit$_i$, | Prewrite$_j$, Commit$_j$, ...}

In these log files, updates from transactions T_i and T_j are recorded in different orders. Now consider what happens if X_2 fails at the | mark. The last log record appearing in log(X_2) is Commit$_i$. Upon recovery, how does X_2 request the updates that it missed? If X_2 requests only what is in log(X_1) after Commit$_i$, the last update that X_2 has processed, then X_2 would miss both Prewrite$_j$ and Write$_j$ messages. Unless it is possible to bound how many transactions can be executed out of sequence at the failed site, X_2 has to send X_1 its entire commit history in order to identify the set of missing updates. (The latter approach has been proposed by several systems.)

We now show that in LAMBDA, the starting point of the missing updates can be bounded. Since each message in LAMBDA NET is timestamped, the set of missing updates in X_2 can be identified using timestamp information rather than according to the ordering in X_2's log. Furthermore, site status messages clearly define the duration that a site fails. Let t_i be the timestamp of the site status message announcing site 2's failure, where site 2 holds X_2. Let t_j be the timestamp of the site status message announcing site 2's recovery. X_2 clearly has missed all the updates with timestamps between t_i and t_j. (X_2 may also have missed some updates before t_i because site 2 actually fails sometimes before its failure is announced and because unprocessed messages may be lost when 2 fails, This problem, although subtle, can still be solved by using the timestamp information. Please refer to [CHAN83a] for a detailed solution.)

Another difficulty in performing log-based recovery in a general network environment is to determine when to change a copy's status from recovering to operational. A recovering copy is required to obtain the most recent updates before it becomes operational. Once operational, it must continue to receive all updates on the logical data item X. Since messages can arrive out of order – a message from site B can arrive at site C before an earlier message from site A arrives – the above recovery synchronization is very difficult to achieve. Note that, in general, failure synchronization performed in a general network only requires that failure messages are synchronized among the other failure messages. Recovery synchronization actually requires that recovery status changes be synchronized with respect to other messages, such as update messages, as well. One way of simplifying the problem is to not allow transactions to commit during the recovery period, as suggested in [GOOD83]. This increases recovery costs considerably: normal transaction execution is interrupted every time a copy recovers from failure. Other solutions execute complex protocols among the recovering copy, the copy that provides the log records, and the transaction managers of the on-going transactions to reach agreement on the copy status change.

LAMBDA provides recovery synchronization by exploiting the total ordering existing among all messages. Once site 2 recovers from failure, the active group method includes X_2 as one of the copies to receive updates on X. That is, X_2 receives all updates on X timestamped after t_j. Updates timestamped before t_j can be obtained from log(X_1). X_2 therefore obtains all updates that it missed. Once X_2 completes applying these updates, it changes its copy status to operational. The only externally visible effect of the copy status change is in the vote that X_2 sends back. (X_2 can now vote "yes" or "no" instead of "pass".) Switching from recovering to operational does not interrupt any normal transaction processing. In fact, transaction managers of on-going transactions will not even realize that this copy status has changed. Log-based recovery therefore is considerably simplified in the LAMBDA NET environment.

4. Conclusion

Atomic broadcast and failure detection are two powerful primitives in designing distributed database systems. Atomic broadcast forms the basis of many database protocols such as transaction commit protocols, election protocols, and the "include" protocol [GOOD83]. Failure detection is required by many fault-tolerant protocols, where all sites must have consistent view of failure status. In LAMBDA NET, the atomic broadcast and failure detection are provided as network primitives. Performing these functions at the network level not only simplifies database protocols but also better utilizes the broadcast network.

In this paper, we have shown that atomic broadcast and the failure detection facility simplify transaction commitment, concurrency control, and crash recovery in a distributed database system. The resulting distributed database system not only is simpler but runs more efficiently than a distributed database system in a point to point network.

5. References

[BANI79] J-S. Banino, C. Kaiser and H. Zimmermann, "Synchronization for Distributed Systems using a Single Broadcast Channel", Proceedings of the 1st International Conference on Distributed Computing Systems, Huntsville, Oct 1979.

[BERN80] P. A. Berstein, et. al, "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM Transaction on Database Systems, March 1980.

[BERN81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, June 1981.

[BERN83] P. A. Bernstein and N. Goodman, "The Failure and Recovery Problem for Replicated Databases", ACM Symposium on Principles of Distributed Computing, August 1983.

[CHAN83a] J. M. Chang "Simplifying Distributed Database Systems Design by Using a Broadcast Network", Proceedings of the ACM SIGMOD Int'l Conference on Management of Data, June 1984.

[CHAN83b] J. M. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols", ACM Transactions on Computer Systems, August 1984.

[CHAN83c] A. Chan, U. Dayal, S. Fox, D. Ries and J. Smith, "On Concurrency Control and Replicated Data Handling in Real-Time Distributed Database Management Systems", CCA position paper, Dec 1983.

[GOOD83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox and D. Ries, "A Recovery Algorithm for a Distributed Database System", ACM Symposium on Principles of Database Systems, March 1983.

[GRAY78] J. N. Gray, "Notes on Database Operating System" In Operating Systems: An advanced Course, Springer-Verlag, 1978, pp. 393-481.

[HAMM80] M. Hammer and D. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases", ACM Transactions on Database Systems, Dec 1980.

[MAXE84] N. F. Maxemchuk and J. M. Chang, "Analysis of the Messages Transmitted in a Broadcast Protocol", Proceedings of the Int'l Computer Conference , Amsterdam, May 1984.

[METC76] R. M. Metcalf, D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm. of ACM July 1976.

[SKEE81] D. Skeen, "Nonblocking Commit Protocol", Proceedings of the ACM SIGMOD Int'l Conference on Management of Data, April 1981.

[SKEE83] D. Skeen, "Atomic Broadcasts", paper in preparation.

[STON79] M. Stonebraker, "Concurrency Control and Consistency of Multiple copies of Data of Distributed INGRES", IEEE Trans. Software Eng. May 1979.

What good are concurrent search structure algorithms for databases anyway?

Dennis Shasha,

Courant Institute,
New York University
New York, New York 10012
shasha@nyu-csd2.arpa

1. Abstract

We discuss a strategy for designing concurrent search structure algorithms which make those algorithms fit well with standard database concurrency control and recovery techniques. The goal is to minimize the number of nodes that must be write-locked until transaction commit time. The strategy consists in isolating restructuring operations (such as B tree splits) into separate transactions. The strategy can be applied profitably to many search structure algorithms. We present an example on a B+ tree.

2. The Setting and the Problem

A dictionary is an abstract data type supporting the actions member, insert, delete, and update on a set consisting of key-value pairs. For example, the key may be the name of an employee and the value may be a pointer to a tuple of the employee relation. A search structure is a data structure used to implement a dictionary. Examples include B trees, hash structures, and unordered lists. Concurrent algorithms on search structures try to allow many transactions to access a search structure at the same time. These algorithms use locks on the nodes of the search structure when it is necessary to synchronize concurrent transactions for correctness purposes. Much work continues to be directed towards the goal of minimizing the number and reducing the exclusivity[1] of the locks (see [Good85] for a fairly complete listing and a brief survey).

This article looks at another problem: how to combine a concurrent search structure algorithm with the general concurrency control and recovery subsystems of a database management system. It may seem surprising that this should be a problem at all. Intuitively, the concurrent search structure algorithm should be a subroutine of the general concurrency control algorithm. While this is possible in principle, combining these two algorithms naively may result in low concurrency due to the recovery algorithm.

2.1. An Example of the Problem

How did recovery get into this? Consider a transaction T that issues an insert(x) and

[1] For example, suppose the rules are that (1) any number of transactions may hold read locks on the same node; and (2) if one transaction holds a write lock, then no other transactions may hold any lock on the node. Then reducing the exclusivity means substituting read locks for write locks whenever possible.

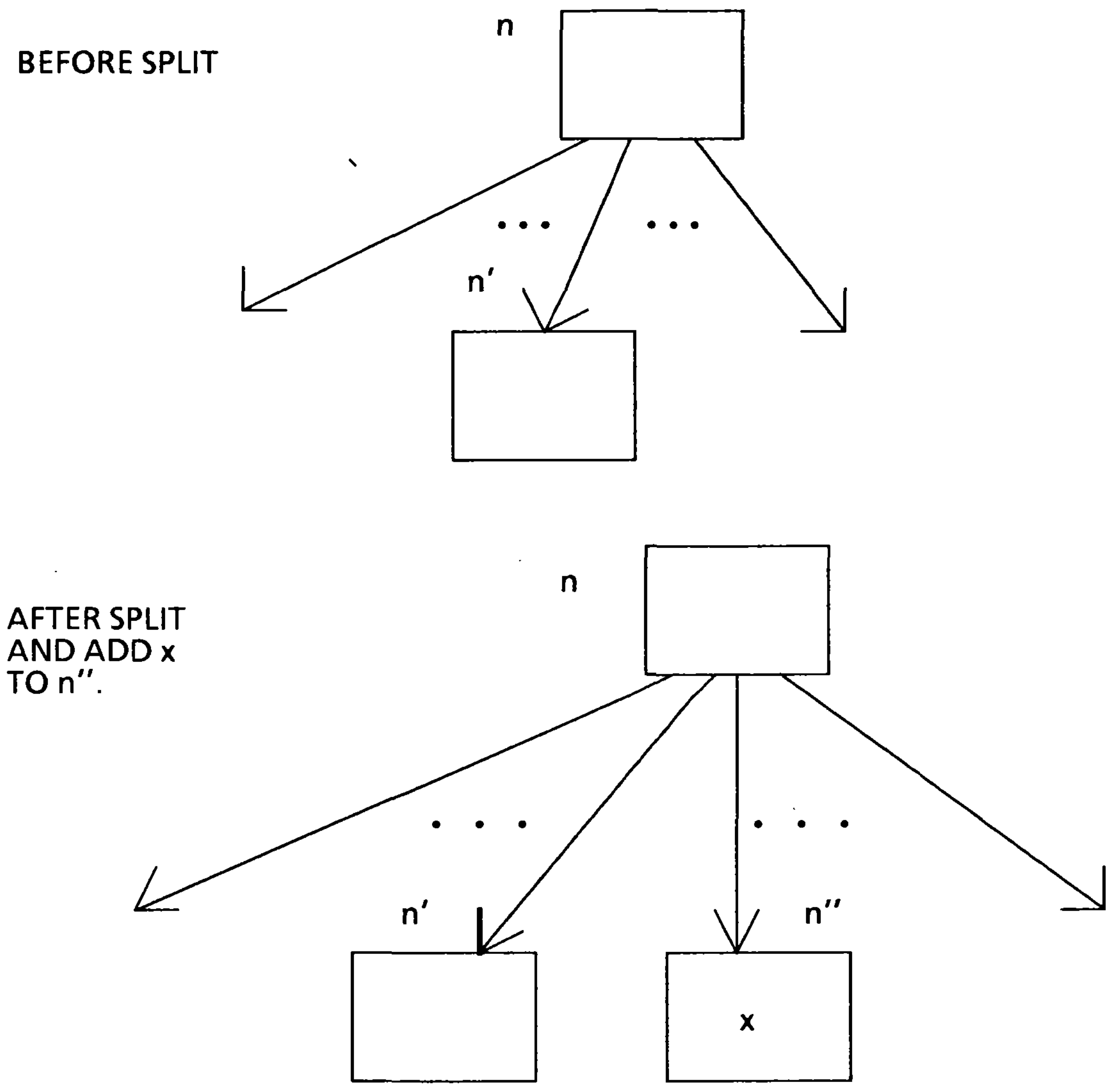


Figure 1 -
The split changes n and n' and creates n'', but does not change the global contents of the structure. Adding x to n'' does.

In a naive approach, the recovery subsystem would hold write locks on n, n', and n''. This is recovery overlock.

In our approach, the split is a separate transaction so the recovery subsytem only holds a lock on n'', the node into which the item x is added.

several other dictionary actions. Suppose the insert causes a split[2] in a B+ tree [Come79], as shown in figure 1. If the transaction is ever aborted, then the split would be undone in most practical recovery schemes changing three nodes (collapsing n' and n'' into one node and changing n to recover the before split configuration in figure 1).

But this is actually unnecessary, since all that has to happen is that x should be removed from one node (n'' in figure 1). Removing x alone would require changing one node instead of three. Also, other transactions would still be able to navigate the structure properly. The reader might object that removing x alone might leave one node (n'') a little sparse, but this entails only a minor [Guib78] performance penalty.

So what? Transactions are seldom aborted and failures are (we hope) also rare, so why should the number of nodes that must be undone matter? The reason is that most recovery subsystems hold write locks on all nodes that might be undone. The write locks ensure that no other transaction can read uncommitted changes [Bern83, Haer83]. If three nodes must be write-locked instead of one, concurrency may decrease a lot. (Holding a write lock on n in figure 1 is particularly bad since n has many children.) We call this problem the *recovery overlock* problem.

3. Two possible solutions

Ideally, the recovery subsystem would hold locks only on the nodes that hold (or could hold, in the case of a delete) the key-value pairs that have been updated. In the case of figure 1, this node would be n''. The recovery overlock problem appears when the recovery subsystem holds locks on other nodes as well. Recovery overlock is a potential problem for all concurrent algorithms on search structures, since those algorithms are only concerned with maximizing the concurrency when there is one insert or delete per transaction. There are at least two remedies to the recovery overlock problem.

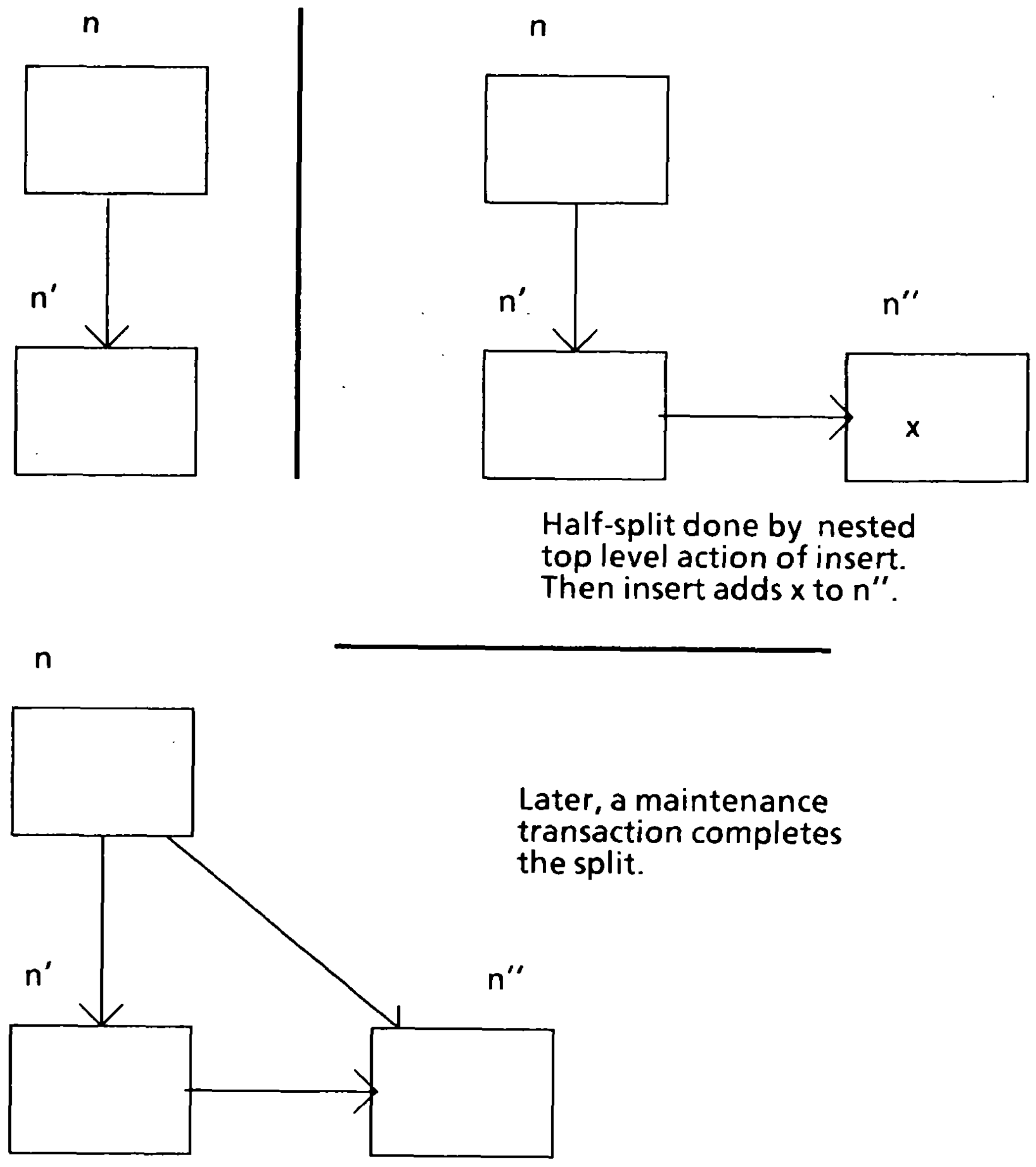
3.1. Two-scheduler approach

The first remedy would be to change the concurrency control and recovery algorithms to operate at a semantic level. That is, data item concurrency control would lock keys whereas the search structure concurrency control would lock nodes. The recovery algorithm in turn would undo updates to key-value pairs by issuing reverse updates (e.g. to reverse insert(x), issue delete(x)). Figure 2 is a schematic of the scheduler architecture.

In our example above, the item lock manager would hold locks on x and any other data items accessed by the transaction. The dictionary actions themselves would only hold node locks as long as required by the concurrent search structure algorithm. (That is, if a transaction issues insert(x), then waits, then issues another dictionary action, it will hold no node locks during the waiting period.) The concurrent search structure algorithm could abort a given dictionary action without aborting the entire transaction.

This is an elegant approach in that it completely divorces search structure concurrency control (concerned with nodes) from data item concurrency control (concerned with keys and values). We discuss this approach in detail in [Shas84, chapters 5 and 6]. The approach has several disadvantages however. It uses two schedulers. Each modifying dictionary action requires an inverse action [Haer83]. As it happens, the inverse action of a dictionary action is one or more other dictionary actions, but this requirement still makes undoing more complicated than simply replacing changed nodes by the before-images of those nodes.

[2] All the reader has to know about splits is that they move the data from one node into two nodes, readjusting the parent of the two nodes appropriately. Altogether three nodes are affected.



Half-split done by nested top level action of insert. Then insert adds x to n''.

Later, a maintenance transaction completes the split.

Figure 3 - Mixed nested and asynchronous restructuring.

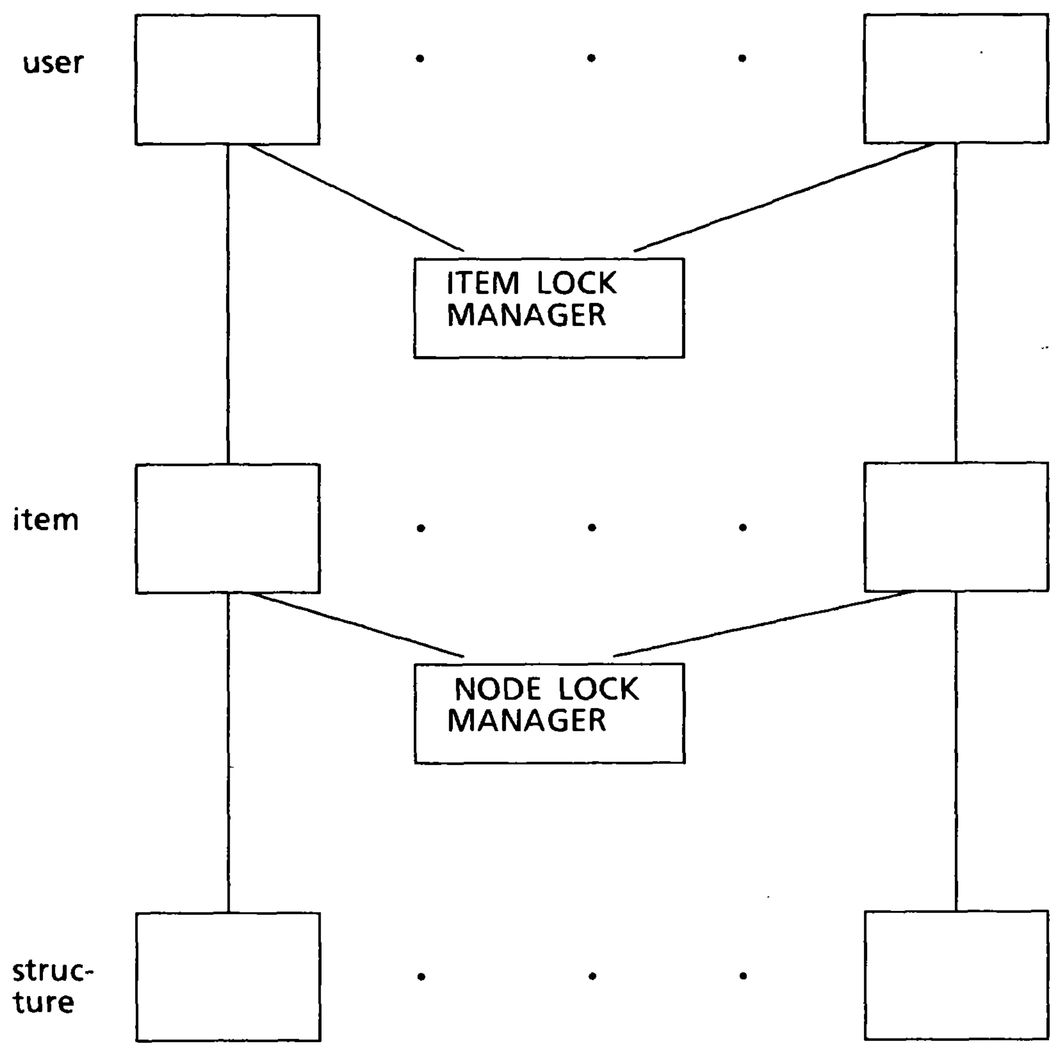


Figure 2 - Two-level architecture to remedy recovery overlock.

Item lock manager used by standard concurrency control algorithm such as two phase locking.

Node lock manager used by concurrent search structure algorithm.

3.2. Isolated restructuring approach

The second remedy would be to alter the architecture of the search structure algorithm itself to "fool" the recovery subsystem into locking only the nodes with updated key-value pairs. That is, we isolate restructuring operations as separate transactions. (For search structures, *restructuring operations* are operations that may change nodes or edges, but do not change the set of key-value pairs contained in the entire structure. Splits are examples of restructuring operations.) Thus, when an insert discovers that it must split a node, the split becomes a separate transaction. This has two implications:
1) if the split has to be aborted, only the changes caused by the split itself are undone; and
2) if the split completes but the insert is undone, the split is not undone.

This strategy may be implemented either by having separate maintenance transactions do all restructuring [Manb82,Shas84], or by making each restructuring operation a special subtransaction within a transaction. The Argus langauge [Lisk82,Moss81,Weih82] allows such special subtransactions, calling them "nested top-level actions."

In the example of figure 1, the insert(x), upon discovering that a split is necessary, might issue the split as a nested top level action. Then the insert would add x to the node n''. The recovery subsystem would hold n'' write-locked until the transaction issuing the insert committed. If the transaction were aborted, the recovery subsystem would replace n'' by its image before $add(x,n'')$, but would not undo the split.

Sometimes we may want to use both special subtransactions and separate maintenance transactions for restructuring. For example, in an algorithm in the spirit of [Lehm81] that we [Lani85] are designing, splits occur in two stages (figure 3). In the first stage, the insert issues a special subtransaction to create a new node n'', to move appropriate data from n' to n'', and to add a forward pointer from n' to n''. The forward pointer ensures that other dictionary actions can reach the data in n''. The insert then calls on a maintenance process to complete the split. This two-stage approach avoids deadlock and has the property that neither the insert nor the maintenance transaction ever locks more than one node.

4. Conclusion

The recovery overlock problem arises when the recovery subsystem holds more locks than necessary. This occurs when the recovery subsystem holds locks on nodes in order to undo restructuring operations (such as splits and merges). The problem may occur no matter how concurrent the search structure algorithm is by itself. We present two solutions to this problem -- one based on undoing dictionary actions by issuing their inverses and the second based on isolating restructuring operations as separate transactions. We think the second solution is more practical.

We can generalize the notion of restructuring operation to any operation on a data structure underlying an abstract data type that leaves the abstract data type state unchanged. For example, an operation that moves a part of the waits-for [Bern81] directed graph used by distributed deadlock detection from one processor to another is a restructuring operation. We apply this generalized notion of restructuring to the transaction processing component of Ultrabase [Shas85], a database machine project based on the New York University ultracomputer [Gott83].

REFERENCES

[Bern81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13, 2, pp. 185-221, June 1981.

[Bern83] P. A. Bernstein, N. Goodman, and V. Hadzilacos, "Recovery Algorithms for Database Systems," Proc. IFIP Congress 1983, Paris, France.

[Come79] R. Comer, "The Ubiquitous B-tree," ACM Computing Surveys, vol. 11, pp. 121-138, 1979.

[Good85] N. Goodman and D. Shasha, "Semantically-based Concurrency Control for Search Structures," Proc. ACM Symposium on Principles of Database Systems, pp. 8-19, March 1985.

[Gott83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer," IEEE Transactions on Computers, vol. C-32, no. 2, pp. 175-189, February 1983.

[Guib78] L. J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," Proc. 19th Annual Symposium of Foundations of Computer Science, pp. 8-21, 1978.

[Haer83] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," ACM Computing Surveys, vol. 15, no. 4, pp. 12-83, December 1983.

[Lani85] V. Lanin and D. Shasha, "A Highly Concurrent, Deadlock-Free B tree algorithm," in preparation.

[Lehm81] P. L. Lehman and S. B. Yao "Efficient locking for concurrent operations on B-trees," ACM Trans. on Database Systems, vol. 6, no. 4, pp. 650-670, December 1981.

[Lisk82] B. Liskov and R. Scheifler, "Guardians and actions: linguistic support for robust, distributed programs," Proc. of the ninth ACM Symposium on Principles of Programming Languages, pp. 7-19, January 1982.

[Manb82] U. Manber and R. E. Ladner, "Concurrency control in a dynamic search structure," Proc. ACM Symposium on Principles of Database Systems, pp. 268-282, 1982.

[Moss81] J. E. B. Moss, "Nested transactions: an approach to reliable distributed computing," MIT/LCS/TR-260, Massachusetts Institute of Technology.

[Shas84] D. Shasha, "Concurrent Algorithms for Search Structures," TR-12-84, Harvard University Center for Research in Computing Technology, June 1984.

[Shas85] D. Shasha and P. Spirakis, "Join Processing in a Symmetric Parallel Environment," Courant Institute, New York University, TR # 158, April 1985.

[Weih82] W. Weihl and B. Liskov, "Specification and Implementation of Resilient, Atomic Data Types," Computation Structures Group Memo 223, MIT, December 1982.

Reducing the Cost of Recovery from Transaction Failure

Nancy D. Griffeth
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
(404)894-2589
nancy@gatech

Abstract. Some algorithms are presented for reducing the cost of recovery from transaction failure. The cost is reduced primarily by reducing the cost of preparing for recovery. This cost is likely to predominate, since failure will often be infrequent, while preparations for recovery must go on while the system is operating normally. Performance evaluations of some of these algorithms are also described.

1. *Introduction*

The cost of preparing for and performing recovery is a major cost of maintaining a database system. This cost has several parts. One part arises from scheduling operations when the system is up so that transactions can be recovered after it has gone down. A second part arises from creating the log. A third part arises from transaction commitment, which must guarantee that all changes to data are written to stable storage. And the fourth part, the cost of the recovery itself, arises after a failure has occurred.

An ongoing project at Georgia Tech has been investigating the principles that govern correct recovery from failure. We have chosen to concentrate on preparation for recovery (i.e., scheduling, logging, and commitment) rather than on recovery after failure. This is because most of the impact of recovery processing is felt during normal operation, when the transaction system is preparing for recovery, rather than during failure recovery. It is an especially frustrating fact that, although the preparatory work is necessary, we actually hope that it will prove to have been unnecessary. To date in this investigation, we have focussed on transaction failure in order to solve a simple version of the problems involved in general failure. Future work will also address processor and media failures.

We have identified some novel methods of recovery scheduling, logging, and rollback. The performance of some of these new methods has been shown by both analytic and simulation studies to be much better than that of some commonly-used recovery procedures. Section II of this paper describes five algorithms for scheduling transactions for recoverability. Section III describes the performance studies that have been carried out on the scheduling algorithms. Section IV describes algorithms for logging transaction undo operations and for performing aborts and undos.

This material is based partly on work supported by the National Science Foundation under grant number MCS-8200854 and partly on work supported by the Army Institute for Research in Management Information and Computer Systems under grant number DAAK70-79-D-0087.

II. *Scheduling Transactions for Recoverability*

Transactions are supposed to simulate "atomic" actions, that is, actions which occur indivisibly. To simulate an atomic action with an action which actually takes place in several steps, we must make sure that no other action sees any partial results of the action. We must also make sure that all of its steps are actually completed, or, if this proves impossible, that none of its effects are installed permanently in the database.

We prevent transactions from viewing partial results of other transactions by requiring that any interleaving of steps of different transactions is **serializable**. This means that if transaction 1 writes A and B and transaction 2 also accesses them, then 2 dos NOT see the value of A written by 1 together with an earlier value of B. Another way of looking at it is that 2 appears to have begun after 1 completed (or vice-versa). Serializability has been addressed at length elsewhere and will not be discussed further here (see BERN81 for a comprehensive discussion of algorithms for enforcing serializability and PAPA79 for the relevant theory).

We also require that all steps of a transaction must be completed before its effects can be made permanent. We say that a transaction has been **committed** when its effects have been made permanent in the database. Before we commit a transaction which has completed all of its steps, it must satisfy a second condition. The second condition requires **recoverability**, that is, no transaction is allowed to commit before all of the transactions whose effects it has seen have committed. If we do not have recoverability, then it is possible for an incomplete transaction to have a permanent effect on the database via another transaction which has seen its effects. For a more detailed discussion of recoverability, see also HADZ83 .

In this work, the individual transactions steps are **read** and **write** operations. A transaction which reads a data value **depends** on the last transaction that wrote it. A data value is **dirty** if it was written by a transaction which has not yet committed. Any transaction which reads such a value is reading **dirty data**. A transaction may also request a **commit** when it has completed. An **abort** operation may also appear in the schedule, introduced by the transaction itself or by the scheduler. (Some authors work with a more general class of operations, with dependency appropriately defined for such classes. See ALLC83,KORT83,SCHW84 .)

There are two simple, well-known ways to schedule transactions for recoverability. One of these blocks any transaction which tries to read or over-write dirty data. This is the **pessimistic** recovery protocol. The other maintains uncommitted versions of the data locally and does not block reads and writes, but allows only committed versions to be seen by any read. This is the **deferred write** recovery protocol.

Three other scheduling protocols have been developed which guarantee "recoverable" execution of transactions in a distributed database system. These protocols are described in detail in GRAH84b . The protocols are:

(1) the **optimistic** protocol, which blocks a transaction from committing until all transactions on which it depends have been committed. Reads and

writes are allowed to execute as soon as they are requested. A dependency graph is maintained, in which dependencies of one transaction on another are recorded as they develop. When a transaction requests a commit, it is blocked if it depends on an uncommitted transaction. When a transaction commits, all dependencies on it are removed from the dependency graph, so that some other transactions may be allowed to commit. When a transaction requests an abort, all transactions which depend on it are aborted. (These other transactions must actually be aborted first, as discussed in secion IV).

(2) the **pessimistic** protocol, which blocks reads and over-writes of dirty data. The reads and writes are enqueued on the data item until its last writer has committed. Then a sequence of reads may be dequeued, and after they have completed, one write can be dequeued. A waits-for graph must be maintained to break deadlocks. One of the deadlocked transactions must be selected as a victim and aborted. The effect of a commit or an abort is to turn dirty data into clean data, so that the enqueued reads and writes can be dequeued, as described above.

(3) the **realistic** protocol, which blocks only reads of dirty data. Multiple versions of writes are maintained. A read is enqueued on the most recent write version when it arrives and is dequeued when that version has committed. If the transaction that wrote the version should abort, then the read will refer to the most recent version preceding the one it had been waiting on. It may be enqueued again if that version has not been committed. When a transaction commits, all reads which have been waiting on versions written by that transaction are dequeued. When a transaction aborts, all reads which have been waiting on versions written by that transaction are moved back to the most recent previous write version. A waits-for graph must also be maintained for the realistic protocol, unless it is known that in all transactions, all read operations precede all writes. In this case no deadlocks can occur.

(4) the **paranoid** protocol, which aborts any transaction which tries to read or over-write dirty data. For this protocol, it is necessary only to keep track of dirty data. When a read or write is requested, it executes unless the data is dirty. If the data is dirty, the transaction aborts. When a transaction commits or aborts, the data it has written becomes clean. There is also a "realistic" version of this protocol, in which over-writing dirty data is allowed. A queue of uncommitted writes must be maintained in case one or more of the writing transactions aborts.

(5) the **deferred write** protocol, similar to that used in optimistic concurrency control KUNG81 , postpones all writes until a transaction requests a commit. If this protocol is used, serializability can only be enforced by checking for it at commit point.

A number of questions were raised about these protocols. First, does the recovery protocol introduce aborts of transactions that would otherwise have committed? Second, what is the effect of a recovery protocol on the meaning of the schedule? In other words, suppose that a schedule is correct, in that the interleaving of reads and writes gives a meaningful result (this would usually mean serializable). Will the recovery protocol change the meaning of the schedule so that we can no longer be sure that it is correct? Third, if we know that serializability is the condition for a schedule to be correct, then what is the effect of the recovery protocol on the serializability of the schedule (whether it changes the meaning or not)?

Introduction of Aborts. The paranoid protocol introduces aborts to enforce recoverability. We would expect the largest number of transaction aborts from this protocol. The pessimistic and realistic protocols introduce them only to break deadlocks. It is interesting to note that if all reads are known to precede all writes in transactions, then no deadlock can occur using the realistic protocol and no aborts will be introduced. Finally, the optimistic protocol cascades aborts. One result of the performance studies described below was that aborts rarely cascaded. Crude analysis indicated that this was at least in part because all reads did precede all writes in the transactions used in the simulations. In this case, the probability of the first cascaded abort is small and the probability of the second is infinitesimal.

Meaning-preservation. The most obvious change to the meaning of operations comes with the deferred write protocol. Since this protocol postpones writes while letting reads proceed, the value which would have been read may not yet be available when the read is executed. The pessimistic and realistic protocols may also change the meaning of a read operation. This happens when the write on which a blocked read waits is rolled back because the transaction requesting the write has been aborted. The read must then access the previous value of the data rather than the one it was waiting on. The paranoid protocol changes the meaning of a read as a result of a slightly different sequence of events. Suppose that transaction A writes a value in record 1 which would be read by transaction B in the original schedule. Suppose also that A must be aborted by the paranoid protocol because, after writing record 1, it tries to read dirty data. Then the the meaning of B's read has been changed. In this case also, the read will access the previously written data value. The optimistic protocol does not rearrange operations in any way and will abort a transaction rather than change the meaning of any of its operations. Thus the meanings of the operations will be preserved.

Preservation of Serializability. Although general serializability is preserved only by the optimistic protocol, there is a very important subclass of the class of serializable schedules for which these protocols are much better behaved. The class of DSR schedules PAPA79 is preserved by the optimistic, pessimistic, paranoid, and realistic protocols. Only deferred writes fails to preserve this class. Since this is the largest known class of schedules which is recognizable in polynomial time, and since all practical schedulers recognize only schedules in this class, we view DSR as the most important class to be preserved. It would seem to be a serious failing of the deferred write protocol that it does not preserve DSR.

III. Performance Studies

A queuing network model of the execution of database transactions has been developed and validated against simulation. The most interesting feature of this model is that it can be used even when transactions (i.e., the customers) must hold multiple resources simultaneously. As discussed in GRIF84 , the only assumptions necessary to use this analysis are (1) that the number of locks held and the waiting time of a blocked transaction can be closely approximated by their means and (2) that the service rate increases linearly with the number of transactions. The result gives a close approximation to the steady state of the system, where the state is a vector nl,...,nk,bl,...,bk , ni is the number of transactions executing their ith operation, and bi is the number of transactions blocked before their ith operation. From the steady state, we can compute all of the interesting measures such as throughput, number of blocked transactions, number of locks held, space required for queues or versions, and so forth.

A simulation study of the scheduling protocols has been carried out with somewhat surprising results. In comparing the realistic and optimistic protocols we found that although in many cases the throughput for the optimistic protocol is slightly higher, it suffers more performance degradation on an unreliable, low-capacity system. The pessimistic protocol had surprisingly poor performance. Although its throughput was quite good when there are a small number of writes compared to the number of reads, it was normally in a dead heat with the paranoid protocol. When there are many write operations, the pessimistic protocol is nearly an order of magnitude worse than the realistic protocol. Hence if we were to rank protocols in order of throughput, we would have to say that the realistic protocol edges out the optimistic protocol for first place, while the pessimistic and paranoid protocols are in a dead heat for a distant last place. We might expect that we would pay for the throughput of the realistic protocol with the extra space for multiple versions of written data values. In comparison with the pessimistic protocol, this is not so: the queues of blocked writes, in the pessimistic protocol, will require about the same amount of space as the multiple versions in the realistic protocol.

IV. *Using Semantic Knowledge to Reduce Log Size and Processing.*

A database can be viewed naturally in terms of its "levels of abstraction". In a relational database, the relations would occupy the highest level. These would be implemented at the next level by files (containing sequences of tuples) and indexes. Files and indexes would then be implemented by disk pages. How to use knowledge of these levels of abstraction to reduce log size and processing is described in the next few paragraphs.

The central idea is suggested by a trick which can be applied to dynamic structures such as B-trees. Suppose that a transaction adds a record to an indexed relation. It must first add the record to the relation and then add the record key to the index. Suppose also that the transaction continues operating on the database while other concurrently executing transactions cause changes to the index structure. For example, the other transactions might split or coalesce nodes of the B-tree. At some point in the execution, it may become necessary to abort the initial transaction. One might think that, as a consequence, it would also be necessary to roll back every action that changed the B-tree structure after the initial transaction added the record key to the index. In this way, the B-tree structure would be restored to its state at the time the key was added.

Because this is such a cumbersome process, concurrent operations on the B-tree would usually be prohibited. Of course, it is not really necessary to be this conservative. The trick mentioned above works as follows. We can simply delete the key from the B-tree to rollback the operation that added the key. After all, we do not really care about the structure of the B-tree. All we really care about is the set of keys it contains.

In GRAH84a , the ideas used in the B-tree trick have been formalized and can be applied in the general case. We assume multiple levels of abstraction. At each level of abstraction we define **abstract actions** which are implemented by state-dependent sequences of **concrete actions**. The concrete actions at one level are the abstract actions at the next lower level (except of course at the lowest level). In the B-tree example, the highest-level abstract actions would be the operations on relations: add a tuple, delete a tuple, query the relation. These would be implemented by concrete actions which operate on files and indexes: add a record to a file, add a key to an index, and so on. The file and index operations are then the abstract actions at the next lower level. They are implemented by page operations, which are the lowest-level concrete actions.

We assume that with every action, we are supplied a collection of state-dependent undo actions. The addition of a key to an index illustrates why multiple state-dependent undo actions must be available. If the key was already in the index, then nothing needs to be done to undo the add action. An identity action should be supplied as the undo for this case. If the key was not already in the index, then it must be removed to undo the add action. A delete action is supplied as the undo for this case.

If actions A and B, both at the same level of abstraction, do not conflict with each other, then it is possible to roll back action A without first rolling back a later action B. For recovery purposes, actions A and B **conflict** if undo(A), for the prior state of A, does not commute with B. (This definition is slightly different from the definition of conflict for the purpose of serializability, where actions A and B conflict if they do not commute.) To undo an action, we must first undo all later actions which conflict with it.

The situation is complicated somewhat if we must consider the concrete level as well. When an abstract action must be rolled back before it has finished executing, that is, before completion of the sequence of concrete actions which implement it, then we cannot undo it at the abstract level but we can **abort** it. To abort it, we recursively abort incomplete concrete actions and roll back complete concrete actions in reverse order to their order of execution. (The concrete actions at the lowest level must be atomic to halt this recursion.) An abstract action **depends** on an earlier abstract action if it has a child (i.e., a concrete action implementing it) which conflicts with a child of the earlier abstract action. We must not abort an action before we abort every action which depends on it. This guarantees that no action is undone before any later conflicting action has been undone. We formalize this intuition about the correctness of an abort in the following definition. A schedule is **revokable** if every undo action is the child of an abort action and no action is aborted before every dependent action has been aborted. This definition is symmetric to the definition of **recoverability**, that no action can commit until every action on which it depends has committed.

Using the above algorithm for rolling back has very nice implications for the size of the log. Once an abstract action has been completed, we can record its state-dependent undo and throw away the undo actions for the concrete actions implementing it. Thus the size of the log could be reduced considerably.

The above definitions require conflicting actions and dependent actions to be ordered. It is possible to interleave the concrete actions for some collection of abstract actions in such a way that no reasonable order can be defined on the abstract actions. Such interleavings would be undesirable. The existence of the required order on the abstract actions can be guaranteed by a modified form of serializability. First, let us say that actions A and B **conflict** if either A or undo(A) does not commute with B. Next, let us say that a schedule of concrete actions implementing a schedule of abstract actions is **serializable** if the schedule can be transformed to a serial schedule by swapping non-conflicting actions. (This is a version of conflict-preserving serializability.) We require serializability at each level of abstraction. That is, if we treat the abstract actions as transactions and consider the schedule of concrete actions implementing them, this schedule is serializable.

We would like to know that, at the top level of abstraction, the resulting state is exactly what we would get from a serial schedule of the lowest-level concrete actions. It follows from a result of BEER83 that this is guaranteed by the level-by-level form of serializability. Furthermore, this type of serializability provides an ordering on the abstract actions, as required for recovery. Finally, a result in GRAH84a establishes that the abstract state resulting from a history of operations which is level-by-level serializable, recoverable, and revokable will be the same as the abstract state resulting from the history after all aborted operations have been deleted. And from the above-cited result in BEER83 , it follows that the abstract state will be the same as that resulting from a serial execution of the actions which were not aborted.

V. Summary and Further Work.

This paper has described several ways to reduce the overhead of recovery processing from transaction failure during normal operation of a transaction system. We are continuing to look at this issue. We intend to extend the methods to include system and media failure. Other problems we are addressing include: the interaction of the serializer with the recovery scheduler, techniques for optimizing the choice of a recovery scheduler, adaptive scheduling, and the use of levels of abstraction to reduce the overhead of checkpointing and redo processing.

Bibliography

ALLC83 Allchin, James E. *An Architecture for Reliable Distributed Systems*, Ph. D. dissertation, Georgia Tech Technical Report GIT-ICS-82 23, 1983.

BEER83 Beeri, C., P.A. Bernstein, N. Goodman, M. Y. Lai, and D. Shasha. "A Concurrency Control Theory for Nested Transactions", *Proceedings of the 1983 ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, August 1983.

BERN81 Bernstein, P. A., and N. Goodman. "Concurrency Control in Distributed Database Systems", *Computing Surveys* vol. 13, no. 2 (June 1981), pp. 185-222.

GRAH84a Graham, Marc H., Nancy D. Griffeth, and J. Eliot B. Moss. "Recovery of Actions and Subactions in a Nested Transaction System", Georgia Tech Technical Report GIT-ICS-84/12, March 1984.

GRAH84b Graham, Marc H., Nancy D. Griffeth, and Barbara Smith-Thomas. Reliable Scheduling of Transactions on Unreliable Systems, *Proceedings of the ACM-SIGACT/SIGMOD Symposium on Principles of Database Systems*, April 1984.

GRIF84 Griffeth, Nancy D. and John A. Miller. "Performance Modeling of Database Recovery Protocols", *Proceedings of the IEEE Symposium on Reliability in Distributed Software and Database Systems*, Oct 15-17 1984; also, to appear in *IEEE Transactions on Software Engineering*, June 1985.

HADZ83 Hadzilacos, Vassos. "An Operational Model of Database System Reliability", *Proceedings of the 1983 SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, August 1983.

KORT83 Korth, H. F. "Locking primitives in a Database System", *Journal of the ACM*, vol. 30 no. 1, January 1983.

KUNG81 Kung, H. T., and J. T. Robinson. "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6 No. 2, June 1981.

PAPA79 Papadimitriou, Cristos H. "The Serializability of Concurrent Database Updates", *Journal of the ACM*, vol. 26 no. 4, 1979.

SCHW84 Schwartz, Peter M., and Alfred Z. Spector. "Synchronizing Shared Abstract Types", *ACM Transactions on Computer Systems*, vol. 2 no. 3, August 1984.

Achieving High Availability in Partitioned Database Systems

Dale Skeen

IBM Research
San Jose, California

Introduction

As factories and businesses become more automated, they are increasing their reliance on computers to store data that is critical to their day to day operations. Since their operations can be crippled or halted entirely if their data bases are not accessible, their requirements for availability are increasing dramatically. Clearly any failures or scheduled downtime of the computers storing critical data must be minimized.

One solution for increasing data availability is to replicate critical data on several processors. This solution is gaining in popularity due to recent hardware trends that are making loosely-coupled networks of small and medium sized computers an economical alternative to large centralized mainframes. By having multiple processors and storage systems with independent failure modes, it is possible to confine and localize the effects of system component failures. Through replication, it is possible, in theory, to provide arbitrarily high availability of important data.

However, the realization of higher availability is problematic. As the number of components increase, so does the likelihood that one or more components will be down due to failures or required maintenance. Moreover, loosely-coupled systems introduce system failure modes that are either impossible or occur with negligible frequency in closely-coupled architectures. Perhaps the most disruptive of these failure modes are *partition failures*, communication failures that fragment the network into isolated subnetworks, called *partitions*. Unless detected and recognized by all affected processors, such failures provide the opportunity for independent, uncoordinated updates to be applied to different replicas of the data, thereby compromising data integrity.

Traditional algorithms for ensuring data integrity during partition failures are based on restricting read/write access to the data on a per data record basis [EAGE83,ELAB85,GIFF79, THOM79]. These algorithms, which we will call *voting algorithms*, enforce the constraint that a data record can not be read in one partition and written in another partition. Although attractive for their simplicity and generality, such algorithms have a number of weaknesses.

One weakness of voting algorithms is that they can not easily account for correlated data accesses, e.g., where a group of records are normally accessed together. Clearly, if a record is only accessed in conjunction with other data records of a group, it should be made accessible within a partition only if some of the other members of the group are accessible.

Another weakness of these algorithms is that they are overly restrictive, as illustrated by the following example. Consider a bank customer having a checking account and a saving account at different branches of the same bank. Assume

The work described herein was performed in collaboration with D. Wright, while the author and he were at Cornell University, Ithaca, New York. Most of the results appear in [SKEE84].

that it is the policy of the bank to allow a customer to overdraw his checking account as long as the overdraft is covered by funds in his saving account. If the branches are partitioned, no voting scheme would allow a checking withdrawal (which requires reading the balance of both accounts) to occur in one partition and allow a savings deposit to occur in the other partition. Clearly, executing these transactions in parallel in different partitions violates neither the banks policy nor the standard notion of correctness for replicated data.

The above example illustrates well a key point: in general, it is more important to provide high availability of the *operations* frequently performed on the data, than it is to provide the capability of reading and writing individual data records. Certainly this is true for the bank customer, who cares little that his checking account record is read and write accessible in a partition, but cares a great deal that a debit transaction is not executable on his account. The most significant weakness of most voting algorithms is that they can not optimize the availability of high-level data operations.

Emphasizing high-level data operations is in accordance with recent programming methodology trends. For many applications, the database can be viewed as a repository for abstract objects, where each object is defined by its abstract state (that is physically represented in one or more data records) and by a set of operations that can access and change the abstract state. High availability of an object in this context means high availability of the operations defined on the object.

This paper discusses one approach, termed *class conflict analysis*, to the problem of maximizing the availability of desired data operations in the presence of communication failures. The basic idea is that each processor identifies its most important data operations and assigns a preference to each one. In the event of a partition failure, the processors in a subnetwork combine their preferences and make executable a subset of the most preferred operations, a subset whose execution is guaranteed to preserve data integrity.

To make the approach as general as possible, we make only a few assumptions about the database system. The principal ones are: (1) the data base is a set of records, (2) all operations on the data are executed atomically, i.e., all operations are *transactions*, and (3) operations can be characterized by their *potential* readsets and writesets. No particular data model is assumed. Our approach accommodates nicely object-based systems, but does not require them. In fact, the "operations" whose availability is maximized could simply be the most frequently executed transactions.

The proposed approach is considered to be a *syntactic approach*, since in checking correctness, it makes no assumptions about the state of the database or about the semantics of the operations performed on the data. For applications where state information or properties of the data operations can be used, our approach provides a basis on which such information can be added.

Background

A database is a set of *logical data objects* that support the basic operations *read* and *write*. The granularity of these objects is unimportant; they could be records, files, relations, etc. The *state* of the database is an assignment of values to the logical data objects. The value of each logical object x is stored in one or more *physical data objects*, which are referred to as the *copies* of x. (The database is *nonreplicated* if each logical object is implemented by exactly one physical object; otherwise, it is *replicated*.) For brevity, logical data objects are subsequently called data objects or, more simply, objects.

Transactions issue read and write operations on logical data objects, which are mapped by the database system to corresponding operations on physical copies. The set of objects read by a transaction is called its *readset*. Similarly, the set of objects written is called its *writeset*. A transaction is assumed to be correct: when executed in isolation, it transforms a correct database state into a new correct database state.

The correctness criteria for a replicated database system is an intuitive one: *the concurrent execution of a set of transactions on replicated data must be equivalent to some serial execution of the same transactions on nonreplicated data.* (Two executions are equivalent if every transaction reads the same values in both executions and the final database states are the same.) This property is known as *one-copy serializability* [BERN83]

Transactions interact with one another indirectly by reading and writing the same data objects. Two operations on the same object are said to *conflict* if at least one of them is a write. Conflicting operations are significant because their order of execution determines the final database state.

For nonreplicated databases, the order of execution of two conflicting operations prescribes a simple *order dependency* between the transactions executing them, a dependency that constrains the execution order of the transactions in any equivalent serial execution. For example, if transaction T_1 executes operation o_1 before transaction T_2 executes operation o_2 and the operations conflict, then T_1 must execute before T_2 in any equivalent serial execution. To show that an execution on a nonreplicated database is serializable, one must show that all order dependencies implied by conflicting operations can be preserved in some serial execution. An easy way to do this is through the use of a *serialization graph*, a directed graph where the transactions are represented by nodes and order dependencies by directed edges. A fundamental theorem in serializability theory states that an execution is serializable if the corresponding serialization graph is acyclic [ESWA76,PAPA79]

Determining whether an execution on a replicated database is serializable turns out to be significantly more difficult. In such a database, conflicting operations can be executed in parallel by operating on disjoint sets of an object's copies. Moreover, depending on how writes and reads on logical objects are implemented, it may be possible to read a copy containing a stale (i.e. old) value of the data object. Note that neither the parallel execution of conflicting operations nor the reading of old data values necessarily implies that transaction execution will be nonserializable (consider the banking example of the introduction where parallel conflicting operations on the savings account were allowed). However, such activity, if unrestricted, normally does result in nonserializable executions. An example based on the banking example is given in Figure 1.

When conflicting operations are executed in parallel on a replicated database, the order dependencies introduced between the corresponding transactions are no longer the simple dependencies illustrated previously. In fact, each pair of conflicting operations introduces a system of dependencies. This complicates both the modeling of executions and the verification of serializability, as evidenced by the serializability graph model introduced by Bernstein and Goodman [BERN83]

However, if it is assumed that a transaction's readset contains its writeset, then a much simpler graph model can be used to represent transaction execution in a replicated database. In particular, all conflicts can be modeled by simple order dependencies (like those in the nonreplicated case). Such a graph model was used by Davidson [DAVI84] to verify the serializability of transaction execution in a partitioned database system.

Partially because of the complexities involved in managing replicated data and partially because

```
 PARTITION 1          PARTITION 2

CHecking: $100       CHecking: $100
SAvings:  $200       SAvings:  $200

if CH+SA>$200        if CH+SA>$200
then CH:=CH-200      then SA:=SA-200

CHecking:-$100       CHecking: $100
SAvings:  $200       SAvings:  $  0
```

Figure 1. The execution of a checking withdrawal and a saving withdrawal resulting in the violation of the integrity constraint: CH + SA > = 0. The anomaly is due to the concurrent execution of conflicting operations in separate partitions.

of the uncertainties concerning the nature and extent of a partition failure, most voting protocols have been extremely conservative in the level of availability provided during a partition failure. In fact, the cornerstone of their correctness is that they do not allow conflicting data operations to be executed in different partitions.

In the next section, we described a less conservative protocol that does not prohibit conflicting operations in different partitions. It uses the assumption that readsets contain writesets to simplify the modeling and the analysis of transaction execution.[2] In particular, only simple order dependencies are considered.

Class Conflict Analysis

Class conflict analysis consists of three distinct steps--transaction classification, conflict analysis, and conflict resolution--which are described in the next three sections. It is assumed that a correct concurrency control protocol serializes transaction execution within each individual partition. (This, of course, does not imply that transaction execution will be serializable across partitions.)

Transaction Classification

Prior to a communication partitioning the processors must agree to a classification of the transactions. The criteria used to classify transactions depends on the way applications (and human users) view the data and on the system availability goals. If a data object is viewed as an instance of some abstract data type, then each operation defined on the abstract type can be considered a class. For example, a "checking withdrawal" is a well defined operation on an object of type "checking account." Even in systems where the data records are normally not viewed as instances of abstract types, the applications using the data may interact with the system using a small number of "canned" transactions, and these become natural candidates for classes. In systems lacking any high-level knowledge on how data is used, a class may simply be all transactions satisfying a certain input/output specification, e.g., the class containing all transactions reading and writing a subset of items a, b, and c. In such a system, performance monitoring can be employed to reveal sets of data items that are frequently accessed as a group.

Classes are characterized by their readset and writesets. The *readset* (respectively, *writeset*) of a class is the union of the readsets (respectively, writesets) of all of its member transactions. Two classes are said to *conflict* if one's readset intersects the other's writeset. A conflict between a pair of classes indicates the *potential* for a pair of transactions belonging to these classes to issue conflicting data operations and, as a result, for the existence of an order dependency between the transactions. (Since transactions are required neither to read all objects in the readsets of their classes nor to write all objects in the writesets of their classes, transactions from conflicting classes may not issue conflicting operations and, therefore, may not have an order dependency.)

Each processor assigns to each class a *preference*, a nonnegative number that reflects the processor's need to execute that class when a partition occurs. The higher a class's preference, the more likely the processor will be able to execute transactions from the class during a partition failure. Preferences may be based on execution frequencies, administrative priorities, etc. The method of assigning preferences is not important to the rest of the discussion.

Prior to a partition failure, each processor must make its preferences known to all other processors.

Conflict Analysis

Let us now discuss the processing that takes place in an arbitrary partition when a partitioning occurs. It is important to remember that the processing described occurs in all partitions independently and in parallel.

2 The assumption is used only for the purpose of modeling transaction executions, it is not enforced in actual executions.

Immediately after a partitioning is detected, each processor executes a protocol to determine the other processors within its communication partition. The members of the partition then combine their individual preferences to calculate partition-wide preferences for all classes. The partition-wide preferences are used to determine a set of classes from which processors can execute transactions.

Individual preferences can be combined to form partition-wide preferences in any number of ways. If the individual preferences are based on execution frequencies, then the partition-wide preferences could be assigned the sums of the individual preferences. If, on the other hand, individual preferences are assigned by administrative policy, then partition-wide preferences could be set to the maximum of the individual preferences. The only requirement for combining preferences is that the inclusion of an additional processor in a partition can never decrease the computed partition-wide preferences.

Based on their knowledge of the network topology and on the cause of the partitioning, the members of a partition must estimate the memberships of all other partitions. To avoid confusion, the partition performing the estimates will be called the *home partition* and the estimated partitions will be called the *estimated foreign partitions*, or the *foreign partitions*, for short. For availability reasons the estimates should be as precise as possible; for correctness reasons the estimates can be high but not low. In the absence of any conclusive evidence on the membership of other partitions, the home partition can simply assume that there exists a single foreign partition, which contains all other processors. In addition to estimating partition membership, partition-wide class preferences must also be estimated using the prespecified processor preferences.

Once the computation of preferences (actual and estimated) is complete, the classes with preferences exceeding a small prespecified minimum are considered for possible execution. In general, not all of the prospective classes can be executed in parallel if one-copy serializability is to be ensured. Consequently, the conflicts between prospective classes must be analyzed, and sets of classes that can lead to non-serializable executions identified. The preferences, both estimated and actual, are the basis for deciding which partitions will be allowed to execute transactions from which classes.

The analysis uses a graph model that is similar to the one used in serializability theory, the major difference being that serializability graphs give all *actual* order dependencies between conflicting transactions; whereas, class conflict graphs give all *potential* order dependencies between conflicting classes. Defined below is a simplified version of the model presented in [SKEE84].

A node in a *class conflict graph* represents the occurrence of a given class in a given partition. Edges are drawn between occurrences of conflicting classes according to the rules given below. Let C_i and C_j be classes such that readset(C_i) and writeset(C_j) *intersect*.

1. If C_i and C_j are in the same partition, then a pair of edges pointing in opposite directions connects them.

2. If C_i and C_j are in different partitions, then a directed edge extends from C_i to C_j.

An edge between two conflicting classes indicates the possibility of an order dependency between transactions from those classes, with the direction of the edge indicating the direction of the order dependency. If two conflicting classes belong to the same partition, then the concurrency control algorithm will resolve conflicts between transactions from those classes. In doing so, the algorithm is free to choose which of the two conflicting transactions to execute first. Hence, the edge pair in rule 1 signifies that either order dependency is possible between transactions in the adjacent classes.

Between classes in different partitions the situation is quite different. The direction of an order dependency between two transactions is determined by the fact that the values produced in one partition are not available for reading in another. Consider the case of executing transaction T_i from class C_i and transaction T_j from C_j (and assume that the transactions read the entire readset and writeset for their respective classes). Since T_i can not read the updates of T_j, the resulting database state[3] is the same as the serial execution of T_i followed by T_j. Clearly, there is an order dependency from T_i to T_j and, therefore, in the conflict graph there is an edge from class C_i to class C_j.

Figure 2 contains a class conflict graph depicting four classes: the checking withdrawal (class C_w) used in previous examples; a class for computing and adding an interest payment to a savings account (C_s); a class for modifying the current interest rate (C_i); and a class that reads the current interest rate and the current checking balance (C_r).

In identifying class conflicts that could lead to nonserializable executions, cycles play a key role. However, not all cycles are bad. Among class

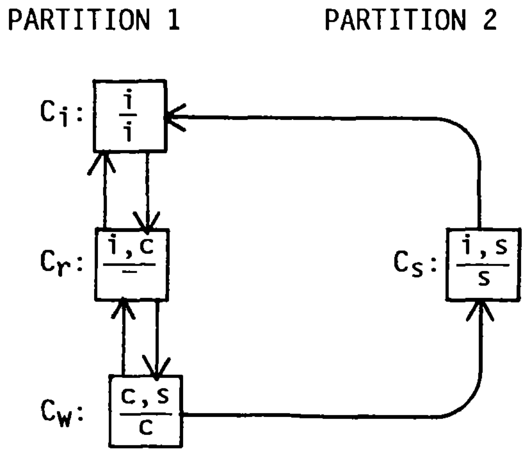


Figure 2. A class conflict graph for the banking example with four classes and three logical data objects–interest rate (i), checking balance (c), and savings balance (s). The readset of the class appears on the top half of the box and the writeset of the class in the bottom half.

occurrences in the same partition, cycles are both common, since in this case all edges are paired, and harmless, since the concurrency control algorithm operating in the partition will prevent nonserializable executions.

On the other hand, cycles spanning two or more partitions are not harmless, since there is no global mechanism that can prevent them from occurring in an execution. Hence, *multipartition cycles indicate the potential for nonserializable executions.* In the example above, if transactions from classes C_i, C_r, and C_w execute--in that order--in partition 1 and a transaction from C_s executes in partition 2, the result is nonserializable. execution.) Note that not all executions of these transaction are nonserializable; in fact, any other order of execution of the transactions in partition 1 is serializable.

Conflict Resolution

Whenever the initial class conflict graph contains multipartition cycles, further constraints on transaction processing must be imposed. Of course, to satisfy our availability goals, the minimum constraints are desired. The process of finding an appropriate set of constraints is called conflict resolution.

The most obvious resolution technique is to prohibit certain transaction classes from executing in certain partitions. In the graph model, this is represented by deleting the nodes corresponding to the appropriate class instances. Deletion of class nodes must continue until the class conflict graph is rendered multipartition acyclic. In the above example, exactly one class--the one with lowest preference--must be deleted to render the graph multipartition acyclic.

The interesting optimization problem imposed by our availability goals is to find the set of classes with smallest summed preferences that render the conflict graph multipartition acyclic. Unfortunately, this optimization problem is an instance of the feedback vertex set problem, a well known

3 Recall that the database state is defined to be the values of the logical objects, not the physical replicas.

NP-complete problem. This, of course, does not rule out the existence of good heuristics.

In practice, the NP-completeness result for our optimization problem is not the major complication in finding a good class deletion algorithm; rather, the major complication is that the preferences ascribed to classes in foreign partitions are only estimates, which may not bear any semblance to their actual preferences. Not only may "optimal" solutions based on estimates be suboptimal with respect to actual preferences, but, more important, they may be incorrect. ("Deleting" a class in a foreign partition is an unenforceable assumption that the foreign partition will not execute transactions from that class. If two partitions estimate different preferences for each other, then they will compute different sets to be deleted.) Whenever there exists uncertainty about the network topology after a partition failure, resolution algorithms tolerating inaccurate estimates are required.

There exist several easily implemented heuristics for ensuring that a solution to the class deletion problem tolerates errors in preference estimation. The simplest heuristic breaks all estimated cycles by deleting class instances from only the home partition. Given that home partition can not underestimate the number of actual cycles, the scheme is obviously correct, since it requires no action on the part of foreign partitions. A significant weakness of this heuristic is that a cycle spanning k partitions will be broken k times (once in each partition), irrespective of the how good the estimates are.

A less naive heuristic divides the responsibility of breaking multipartition cycles evenly between the partitions. It uses the rule that the home partition assumes responsibility for breaking a multipartition cycle if the least preferred class in the cycle is in the home partition; otherwise, a foreign partition must break the cycle.[4] (The home partition, however, assumes nothing about how a foreign partition breaks a cycle.) Given that the home partition can not underestimate the preferences of classes in foreign partitions, it is assured that at least one partition will assume responsibility for breaking each cycle. For the cycles that the home partition must break, any algorithm can be used.

Although class deletion is the most intuitive conflict resolution technique, it often restricts transaction execution more than necessary. A less restrictive resolution technique, called *edge deletion*, guarantees serializable executions by restricting the order of execution of transactions within an individual partition.

Consider again the four classes illustrated in Figure 2. The only way a nonserializable execution can occur is if an interest transaction (of class C_i) executes and then a transaction of class C_r executes. Therefore serializability can be preserved if the system disallows the execution of C_r transactions after the first interest transaction has been executed (but C_r transactions *can* execute up until the time the first interest transaction is executed). Note that the same effect can be achieved if C_r transactions are allowed to execute after interest transactions have executed, but are required to read the original (unmodified) version of the interest rate (data object i).

The resolution technique of edge deletion, though less restrictive than class deletion, is also less general. It can not be used to break all multipartition cycles; in particular, it can not break a cycle where each class is from a different partition (prohibiting a certain order of transaction execution in this case is meaningless). Nonetheless, edge deletion is an attractive technique, and the theory underlying it is rich and involved, encompassing both standard concurrency control theory and multiversion concurrency control theory. For more details, the reader is referred to [SKEE84].

[4] The assignment of responsibility can be performed in $O(n^4)$ steps, where n is the number of class instances, even though the number of cycles may be exponential in n.

Conclusions

We have discussed a method of increasing data availability in a partitioned database system by first classifying the transactions and high-level operations that manipulate the database and then preanalyzing potential class conflicts that could lead to data integrity violations. The analysis is syntactical: only knowledge about the classes' readsets and writesets is used. In contrast to popular approaches, the method emphasizes the availability of high-level data operations, rather than the availability of individual data records.

An obvious extension to this work is to enrich the analysis by including either database state information or database semantics. Along the same lines, the analysis could be enriched by viewing the database as a repository for objects of declared abstract types, and by making use of type-specific information. Type-specific information has already been applied to standard concurrency control theory [SCHW85] and to voting algorithms [HERL85]. Given the popularity of programming methodologies based on abstract types, this is a particularly promising avenue of investigation.

Bibliography

[BERN83] Bernstein, P.A. and Goodman, N., The Failure and Recovery Problem for Replicated Databases, *Proc. ACM Symp. on Principles of Distributed Systems* (Montreal Canada, August 1983) pp. 114-122.

[DAVI84] Davidson, S.,Optimism and Consistency in Partitioned Distributed Database Systems, *ACM Trans. on Database Systems* 9 (September 1984) pp. 456-482.

[EAGE83] Eager, D.L. and Sevcik, K.C.,Achieving Robustness in Distributed Database Systems., *ACM Trans. on Database Systems* **8** (September 1983) pp. 354-381.

[ELAB85] El Abbadi, A., Skeen, D., and Cristian, F.,An Efficient, Fault-Tolerant Protocol for Replicated Data Management, *Proc. ACM Symp. on Principles of Database Systems* (Portland, Oregon, March 1985) pp. 215-229.

[ESWA76] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L.,The Notions of Consistency and Predicate Locks in a Database System, *Commun. ACM* **19** (November 1976) pp. 624-633.

[GIFF79] Gifford, D.K.,Weighted Voting for Replicated Data., *Proc. 7th Symposium on Operating System Principles* (Asilomar, Calif., December 1979) pp. 150-162.

[HERL85] Herlihy, M.,Using Type Information to Enhance the Availability of Partitioned Data, *CMU tech. report* (Pittsburgh, Penn., April 1985).

[PAPA79] Papadimitriou, C.,The Serializability of Concurrent Database Updates., *Journal Assoc. Comp. Mach.* **26** (October 1979) pp. 631-653.

[SKEE84] Skeen, D. and Wright, D.D.,Increasing Availability in Partitioned Database Systems, *Proc. 1984 ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (March 1984).

[SCHW85] Schwarz, P. and Spector, A.,Synchronization of Shared Abstract Types, *ACM Trans. on Computer Systems* **2** (August 1984) pp. 223-250.

[THOM79] Thomas, R.H.,A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases, *ACM Trans. Database Syst.* **4** (June 1979) pp. 180-209.

IEEE COMPUTER SOCIETY
TECHNICAL COMMITTEE APPLICATION

Please complete the form on the reverse to apply for member or correspondent in a Technical Committee. Upon completion, please return this form to the following address:

IEEE COMPUTER SOCIETY
1109 Spring Street, Suite 300
Silver Spring, Maryland 20910

We look forward to having you with us!

DEFINITIONS:

TC MEMBER—Actively participates in TC activities; receives newsletters and all TC communications. Active participation means that you do or are willing to do something for the TC such as review papers, help organize workshops, conferences, etc., participate in standards development, help with TC operations, etc.

TC CORRESPONDENT—Does not participate actively in TC activities; receives newsletters and other TC communications.

TECHNICAL COMMITTEE CODES:

Computational Medicine (**01**)
Computer Architecture (**02**)
Computer Communications (**03**)
Computer Elements (**04**)
Computer Graphics (**05**)
Computer Languages (**06**)
Computer Packaging (**07**)
Computers in Education (**08**)
Computing and the Handicapped (**09**)
Data Base Engineering (**10**)
Design Automation (**11**)
Distributed Processing (**12**)
Fault-Tolerant Computing (**13**)
Mass Storage Systems & Technology (**14**)
Mathematical Foundations of Computing (**15**)
Microprocessors & Microcomputers (**16**)
Microprogramming (**17**)
Multiple-Valued Logic (**18**)
Oceanic Engineering & Technology (**19**)
Office Automation (**20**)
Operating Systems (**21**)
Optical Processing (**22**)
Pattern Analysis & Machine Intelligence (**23**)
Personal Computing (**24**)
Real Time Systems (**25**)
Robotics (**26**)
Security and Privacy (**27**)
Simulation (**28**)
Software Engineering (**29**)
Test Technology (**30**)
VLSI (**31**)

IEEE COMPUTER SOCIETY

IEEE COMPUTER SOCIETY TECHNICAL COMMITTEE APPLICATION

INSTRUCTIONS: PLEASE PRINT IN INK OR TYPE (ONE CHARACTER PER BOX. INFORMATION WRITTEN OUTSIDE OF BOXES WILL NOT BE RECORDED). BECAUSE OF PACKAGE DELIVERY SERVICES, STREET ADDRESSES ARE PREFERRED RATHER THAN, OR IN ADDITION TO, POST OFFICE BOX NUMBERS. INTERNATIONAL MEMBERS ARE REQUESTED TO MAKE BEST USE OF AVAILABLE SPACE FOR LONG ADDRESSES.

LAST NAME ☐ FIRST NAME ☐ INITIAL ☐ DR./MR./MRS./MS./MISS/PROF., ETC ☐

COMPANY/UNIVERSITY/AGENCY NAME ☐ DEPARTMENT/MAIL STOP/BUILDING/P.O. BOX/APARTMENT/ETC ☐

STREET ADDRESS (OR POST OFFICE BOX) ☐

DATE: MONTH ☐ DAY ☐ YEAR ☐

CHECK ONE:
☐ NEW APPLICATION
☐ INFORMATION UPDATE

CITY ☐ STATE ☐ ZIP CODE ☐

COUNTRY ☐ OFFICE PHONE ☐ HOME PHONE *(Optional)* ☐

ELECTRONIC MAIL NETWORK ☐ ELECTRONIC MAIL ADDRESS *(Mailbox)* ☐

801

TELEX NUMBER *(Optional)* ☐ IEEE MEMBER/AFFILIATE NO. ☐

I am a Computer Society Member ☐ YES ☐ NO

CHECK ONE: **TC CODE:** ☐ *(from reverse)*

☐ TC MEMBER
☐ TC CORRESPONDENT

PLEASE INDICATE YOUR TC ACTIVITY INTERESTS *(Please circle)*:

	HIGH INTEREST					NO INTEREST
Conference Organizing Committee	5	4	3	2	1	0
Workshop Organizing Committee	5	4	3	2	1	0
Conference/Workshop Session Chairman	5	4	3	2	1	0
Help with Newsletter	5	4	3	2	1	0
Write Survey/Technical Papers	5	4	3	2	1	0
Review Papers	5	4	3	2	1	0
Standards Development	5	4	3	2	1	0
Educational Activities	5	4	3	2	1	0
Help with TC Operations	5	4	3	2	1	0

Other *(Specify)*: ____________________

Technical Interests/Specialties: ____________________

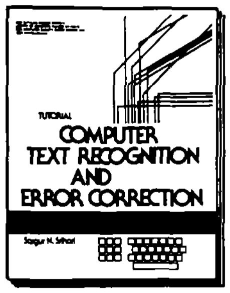
____

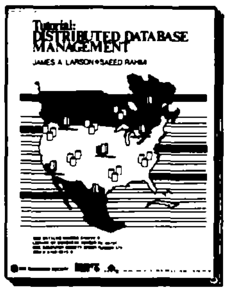
____

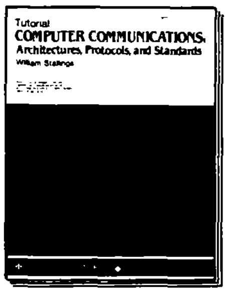
FOR OFFICE USE ONLY

Approved *(TC Chair/designate)*	Date
Key Entered	Date
Returned to TC Chair/Designate	Date

MVL NL 284







Tutorial: Computer Text Recognition and Error Correction
by Sargur N. Srihari
Designed for computer researchers interested in developing flexible techniques for text processing and computer vision, this tutorial is concerned with transferring a degree of intelligence to text processing systems. In particular, the ability to automatically detect and correct spelling and typographical errors and to interpret digitized video images of print and script whose iconic representations (visual symbols) are ambiguous.
CONTENTS: Introduction; Text Interpretation: Visual Processing, Contextual Post-Processing Based on n-Gram Statistics, Lexicon-Based Methods; Text Error Correction: String Edit Distance Computation, Principles of Practical Programs, Systems Programming Applications; Dictionary Organization.
CJ579 (ISBN 0-8186-0579-0): December 1984, 363 pp.,
list price $36.00/member price $24.00

Tutorial: Recent Advances in Distributed Data Base Management
by C. Mohan
This tutorial assumes prior exposure to centralized data base management concepts and therefore is intended for systems designers and implementors, managers, data base administrators, students, researchers, and other technical personnel.
CONTENTS: Introduction; Distributed Data Base Systems Overview; Distributed Query Processing; Distributed Transaction Management; Distributed Algorithm Analysis; Annotated Bibliography.
CJ571 (ISBN 0-8186-0571-5): December 1984, 350 pp.,
list price $36.00/member price $24.00

Tutorial: Distributed Database Management
by J.A. Larson and S. Rahimi
This tutorial provides a written description of the basic components of distributed database management systems and examines how those components relate to each other.
CONTENTS: Introduction; Transforming Database Commands; Semantic Integrity Constraints; Decomposing Requests; Concurrency and Replication Control; Distributed Execution Monitor; Communications Subsystem; Design of Distributed DBMSs; Case Studies; Glossary.
CJ575 (ISBN 0-8186-0575-8): January 1985, 678 pp.,
list price $36.00/member price $24.00

Tutorial: Software Quality Assurance: A Practical Approach
by T.S. Chow
This tutorial provides the reader with a comprehensive view of software quality issues—the various components of a software quality assurance plan and how they are implemented. Intended for managers and engineers who are interested in understanding software quality and in familiarizing themselves with the latest advances.
CONTENTS: Introduction; Software Quality: Definitions, Measurements, and Applications; Managerial Issues: Planning, Organization, and Control; Managerial Issues: Standards, Practices, and Conventions; Technical Issues: Requirements, Design, and Programming; Technical Issues: Testing and Validation; Software Tools; Implementation of Software Quality Assurance Programs; Subject Index.
CJ569 (ISBN 0-8186-0569-3): January 1985, 506 pp.,
list price $36.00/member price $27.00

Tutorial: Computer Communications: Architectures, Protocols, and Standards
by William Stallings
This tutorial presents the motivations for, and design priciples of, a communications architecture. It also includes a broad overview of communication protocols while exploring the following general categories: principles, services and mechanism, and standards.
CONTENTS: Communications Architecture; Physical and Data Link Protocols; Network Access Protocols; Internetworking; Transport and Session Proposals; Presentation and Application Protocols; Glossary; Bibliography.
CJ604 (ISBN 0-8186-0604-5): March 1985, 496 pp.,
list price $39.00/member price $24.00

Tutorial: Local Network Equipment
by Harvey A. Freeman and Kenneth J. Thurber
This tutorial is offered as a means of learning more about the products and systems that are being used today or will be used in the near future. The articles describe the various companies' experiences in building their products, in supporting the many applications for local networks, and in dealing with the theoretical issues of local networks in practical terms. The Text assumes that the reader has some knowledge of the theory, terms, and issues involved.
CONTENTS: Introduction; Local Area Networks; High-Speed Local Networks; Digital Switches and Computerized Branch Exchanges; The Network Interface; Performance; Internetworking; VIII Design Issues.
CJ605 (ISBN 0-8186-0605-3): March 1985, 380 pp.,
list price $36.00/member price $24.00

2

PUBLICATIONS ORDER FORM

Return with remittance to:
IEEE Computer Society Order Department
P.O. Box 80452
Worldway Postal Center
Los Angeles, CA 90080 U.S.A.

Discounts, Orders, and Shipping Policies:

Member discounts apply on the FIRST COPY OF A MULTIPLE-COPY ORDER (for the same title) ONLY! Additional copies are sold at list price.

Priority shipping in U.S. or Canada, ADD $5.00 PER BOOK ORDERED. Airmail service to Mexico and Foreign countries, ADD $15.00 PER BOOK ORDERED.

Requests for refunds/returns honored for 60 days from date of shipment (90 days for overseas).

ALL PRICES ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL BOOKS SUBJECT TO AVAILABILITY ON DATE OF PAYMENT.

ALL FOREIGN/OVERSEAS ORDERS MUST BE PREPAID.

Minimum credit card charges (excluding postage and handling), $15.00.

Service charge for checks returned or expired credit cards, $10.00.

PAYMENTS MUST BE MADE **IN U.S. FUNDS ONLY**, DRAWN ON A U.S. BANK. UNESCO coupons, International money orders, travelers checks are accepted. **PLEASE DO NOT SEND CASH.**

ORDER HANDLING CHARGES (based on the $ value of your order—not including sales tax and postage)

For orders totaling:	Add:
$ 1.00 to $ 10.00	$ 3.00 handling charge
$ 10.01 to $ 25.00	$ 4.00 handling charge
$ 25.01 to $ 50.00	$ 5.00 handling charge
$ 50.01 to $100.00	$ 7.00 handling charge
$100.01 to $200.00	$10.00 handling charge
over $200.00	$15.00 handling charge

PLEASE SHIP TO:

NAME

AFFILIATION (company or attention of)

ADDRESS (Line - 1)

ADDRESS (Line - 2)

CITY/STATE/ZIP-CODE

COUNTRY

(required for discount)
IEEE/COMPUTER SOCIETY MEMBER NUMBER

PHONE/TELEX NUMBER

PURCHASE ORDER NUMBER

AUTHORIZED SIGNATURE

QTY	ORDER NO.	TITLE/DESCRIPTION	M/NM PRICE	AMOUNT

If your selection is no longer in print, will you accept microfiche at the same price?

☐ Yes ☐ No

SUB TOTAL $ ______
CALIFORNIA RESIDENTS ADD 6% SALES TAX $ ______
HANDLING CHARGE (BASED ON SUB-TOTAL) $ ______
OPTIONAL PRIORITY SHIPPING CHARGE $ ______
TOTAL $ ______

METHOD OF PAYMENT (CHECK ONE)

☐ CHECK ENCL. ☐ VISA ☐ MASTERCARD ☐ AMERICAN EXPRESS

CHARGE CARD NUMBER

EXPIRATION DATE

SIGNATURE

CJ

PUBLICATIONS ORDER FORM

Return with remittance to:
IEEE Computer Society Order Department
P.O. Box 80452
Worldway Postal Center
Los Angeles, CA 90080 U.S.A.

Discounts, Orders, and Shipping Policies:

Member discounts apply on the FIRST COPY OF A MULTIPLE-COPY ORDER (for the same title) ONLY! Additional copies are sold at list price.

Priority shipping in U.S. or Canada, ADD $5.00 PER BOOK ORDERED. Airmail service to Mexico and Foreign countries, ADD $15.00 PER BOOK ORDERED.

Requests for refunds/returns honored for 60 days from date of shipment (90 days for overseas).

ALL PRICES ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL BOOKS SUBJECT TO AVAILABILITY ON DATE OF PAYMENT.

ALL FOREIGN/OVERSEAS ORDERS MUST BE PREPAID.

Minimum credit card charges (excluding postage and handling), $15.00.

Service charge for checks returned or expired credit cards, $10.00.

PAYMENTS MUST BE MADE **IN U.S. FUNDS ONLY**, DRAWN ON A U.S. BANK. UNESCO coupons, International money orders, travelers checks are accepted. **PLEASE DO NOT SEND CASH.**

ORDER HANDLING CHARGES (based on the $ value of your order—not including sales tax and postage)

For orders totaling:	Add:
$ 1.00 to $ 10.00	$ 3.00 handling charge
$ 10.01 to $ 25.00	$ 4.00 handling charge
$ 25.01 to $ 50.00	$ 5.00 handling charge
$ 50.01 to $100.00	$ 7.00 handling charge
$100.01 to $200.00	$10.00 handling charge
over $200.00	$15.00 handling charge

PLEASE SHIP TO:

NAME

AFFILIATION (company or attention of)

ADDRESS (Line - 1)

ADDRESS (Line - 2)

CITY/STATE/ZIP-CODE

COUNTRY

(required for discount)
IEEE/COMPUTER SOCIETY MEMBER NUMBER

PHONE/TELEX NUMBER

PURCHASE ORDER NUMBER

AUTHORIZED SIGNATURE

QTY	ORDER NO.	TITLE/DESCRIPTION	M/NM PRICE	AMOUNT

If your selection is no longer in print, will you accept microfiche at the same price?

☐ Yes ☐ No

SUB TOTAL $ ______
CALIFORNIA RESIDENTS ADD 6% SALES TAX $ ______
HANDLING CHARGE (BASED ON SUB-TOTAL) $ ______
OPTIONAL PRIORITY SHIPPING CHARGE $ ______
TOTAL $ ______

METHOD OF PAYMENT (CHECK ONE)

☐ CHECK ENCL. ☐ VISA ☐ MASTERCARD ☐ AMERICAN EXPRESS

CHARGE CARD NUMBER

EXPIRATION DATE

SIGNATURE

CJ

CALL FOR PAPERS

DATA ENGINEERING

The Second International Conference on Data Engineering

Bonaventure Hotel
Los Angeles, California, USA
February 4-6, 1986

Sponsored by the IEEE Computer Society

Committee

Honorary Chairman:
C. V. Ramamoorthy
University of California, Berkeley, CA 94720

General Chairman:
P. Bruce Berra
Syracuse University, Syracuse, NY 13210
(315) 423-4445

Program Chairman:
Gio Wiederhold
Dept. of Computer Science
Stanford University, Stanford, CA 94305
(415) 497-0685

Program Co-Chairpersons:
Iris Kameny, SDC, *Santa Monica, CA 90406*
Ming T. (Mike) Liu, *Ohio State Univ., Columbus, OH 43210*
Richard L. Shuey, *Schenectady, NY 12309*
Joseph Urban, *Univ. S.W. Louisiana, Lafayette, LA 70504*
Benjamin W. Wah, *Purdue Univ., W. Lafayette, IN 47907*

Tutorials:
Peter Ng, *Univ. of Missouri, Columbia, MO*

Treasurers:
Lily Chow, *IEEE, Silver Spring, MD 20910*
Aldo Castillo, *TRW, Redondo Park, CA 90278*

Local Arrangements:
Walter Bond, *Cal State Univ., Dominquez Hills, CA 90747*
(213) 516-3580/3398

Publicity:
Mas Tsuchiya, *TRW Colorado Springs, CO 80916*
(303) 570-8376

Committee Members

J.L. Baer
Bharat Bhargava
Olin Bray
Richard Braegger
Alfonso Cardenas
John Carlis
Nick Cercone
Peter P. Chen
John Steve Davis
Christoph F. Eick
Ramez ElMasri
Robert Epstein
Michael Evangelist
Domenico Ferrari
King-Sun Fu
Hector Garcia-Molina
Georges Gardarin
Sakti P. Ghosh
Yang-Chang Hong
Lee Hollaar
Sushil Jajodia
Arthur Keller
Charles Kellogg
Roger King
Virginia P. Kobler
Henryk Jan Komorowski
Robert R. Korfhage
Tosiyasu L. Kunii
Winfried Lamersdorf
James A. Larson
Matt LaSaine
Victor Li
Witold Litwin
Jane W.S. Liu
Raymond A. Liuzzi
Yuen Wah Eva Ma
Mamoru Maekawa
Gordon McCalla
Toshimi Minoura
N.M. Morfuni
Jaime Murow
Sham Navathe
G.M. Nijssen
Ole Oren
Peter Rathmann
Z. Meral Ozsoyoglu
C. Parent
Domenico Sacca
Giovanni Maria Sacco
Sharon Salveter
G. Schlageter
Edgar Sibley
Peter M. Stocker
Stanley Y.W. Su
Denji Tajima
Marjorie Templeton
A.M. Tjoa
Y. Udagawa
Susan D. Urban
P. Valduriez
R.P. VanDeRiet
Ouri Wolfson
Harry K.T. Wong
Helen Wood
David Y.Y. Yun

For further information write to:
Second Data Engineering Conference
c/o **IEEE Computer Society**
1109 Spring Street, Suite 300
Silver Spring, MD 20910
(301) 589-8142
TWX: 7108250437 IEEE COMPSO

SCOPE

Data Engineering is concerned with the role of data and knowledge about data in the design, development, management, and utilization of information systems. As such, it encompasses traditional aspects of databases, knowledge bases, and data management in general. The purpose of this conference is to continue to provide a forum for the sharing of experience, practice, and theory of automated data and knowledge management from an engineering point-of-view. The effectiveness and productivity of future information systems will depend critically on improvements in their design, organization, and management.

We are actively soliciting industrial contributions. We believe that it is critically important to share practical experience. We look forward to reports of experiments, evaluation, and problems in achieving the objectives of information systems. Papers which are identified as such will be processed, scheduled, and published in a distinct track.

TOPICS OF INTEREST

- Logical and physical database design
- Data management methodologies
- Distribution of data and information
- Performance Evaluation
- Design of knowledge-based systems
- Architectures for data- and knowledge-based systems
- Data engineering tools

We also are planning a special workshop track:
- Performance models and measurement of relational database systems

and solicit papers which report or evaluate such findings.

Awards, Student Papers, and Subsequent Publication:
An award will be given for the best paper at the conference. Up to three awards of $500 each to help defray travel costs will be given for outstanding papers authored by students. Outstanding papers will be considered for publication in the IEEE Computer Magazine, the Transactions on Computers, and the Transactions on Software Engineering. For more information, contact the General Chairman.

Paper submission:
Four copies of papers should be mailed before July 1, 1985 to:
Second Data Engineering Conference
IEEE Computer Society
1109 Spring Street, Suite 300
Silver Spring, MD 20910
(301) 589-8142

Conference Timetable:
Manuscripts due: July 1, 1985
Acceptance letters sent: October 1, 1985
Camera-ready copy due: November 15, 1985
Tutorials: February 3, 1986
Conference: February 4-6, 1986

See you in Los Angeles!

IEEE COMPUTER SOCIETY   IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

CJ

IEEE COMPUTER SOCIETY

Administrative Office

P.O. Box 639
Silver Spring, Maryland
20901

Non-profit
Organization
U.S. Postage
Paid
Silver Spring, MD
Permit No. 1398