DISTRIBUTED TRANSACTION MANAGEMENT IN JASMIN

Ming-Yee Lai W. Kevin Wilkinson

Bell Communications Research, Inc.

ABSTRACT

In this paper, we describe the architecture of JASMIN, a functionally distributed database machine which uses replicated software modules (DM, RM, IS) to achieve high degrees of throughput. We discuss some issues in distributing data and metadata in JASMIN. We describe our distributed multiversion validation technique along with the two phase commit protocol which we use to achieve concurrency control and crash recovery for data and metadata. The scheme also solves the version consistency problem in the multiprocessor environment.

1. INTRODUCTION

JASMIN is an experimental database machine intended for research in large database, high volume applications. It is designed as a functionally distributed database machine, in the spirit of DBC [BAN78], SABRE [GAR83], and SPIRIT [KAM79]. A common goal of these projects is to achieve high system throughput by decomposing the work of a database management system into modules and running those modules on separate processors. JASMIN is different in that other work has tended to emphasize the decomposition of a database manager, resulting in many different modules (each perhaps on its own processor). We feel that too fine a decomposition can hurt performance by increasing the number of control messages required per transaction. We use a simple 3-level decomposition. Our plan is to achieve higher degrees of multiprocessing by replicating software modules.

A uniprocessor prototype of JASMIN has been completed. Its performance encouraged us to extend our work to a distributed environment. In this paper, we describe our solutions to the problem of distributed transaction management for data and metadata. These problems are interesting due to the novel design of the concurrency control mechanism and the need to transparently replicate software modules. The next section gives an overview of the JASMIN architecture. More details and some performance results may be found in [FIS84]. Section 3 describes how we coordinate access to distributed data and Section 4 explains our approach to metadata access and update.

2. JASMIN ARCHITECTURE

Two of our design goals for JASMIN are that the design should allow incremental growth in processing power, and support transparent distribution of data and software. To satisfy our goals, we have devised a software architecture with three layers of database services. Each successive layer represents a higher level of abstraction. The layers are implemented as processing modules (servers) that communicate with each other via messages. Because intermodule communication is restricted to messages, the module-to-processor assignments can be changed with no effect on the software. Further, the modules are written in a fashion that allows each to coexist with clones of itself. Thus, JASMIN can be configured with as many modules and processors as processing requirements dictate.

The layered architecture consists of IS, RM and DM modules, each built on the previous layer. The Intelligent Store (IS) [ROO82] performs page and physical block management and transaction management (concurrency control, and crash recovery). The Record Manager (RM) performs record and index management, and single relation query processing using the facilities of the IS. The Data Manager (DM) builds on the RM to provide full relational query processing. The message passing mechanism is provided by the JASMIN kernel.

2.1 The Kernel

The JASMIN kernel provides a simple, but powerful, set of facilities on which to build distributed system software. It offers three types of services: tasking, scheduling, and message passing. A task is the unit of execution. It has a priority level, a stack and a private data segment. A module consists of several tasks, all sharing a public data segment. Task scheduling is based on priority; a task is not preempted by another task at the same or lower priority until it releases the processor to receive a message. The kernel does no swapping or paging of memory. All inter-task communication is via messages that are sent over communication "paths" similar to the links used in Roscoe [SOL79] and DEMOS [BAS77].

The kernel includes no database-specific features and even device drivers are not included. Thus, the kernel is a sparse environment with little of the overhead found in general purpose operating systems [STO81]. It gives the database system implementor complete control over how memory is scheduled, swapped, and paged.

2.2 Intelligent Store

The Intelligent Store (IS) is a sophisticated page manager that maps pages into secondary storage. The IS underlies the JASMIN database management facilities, providing data consistency, concurrency control, and crash recovery. The IS is transaction oriented. It permits multiple transactions to access and update pages stored on one or more disk subsystems. Page requests are based on (logical) page identifiers. The IS translates between logical page id's and physical block addresses. Thus, an IS user cannot know the physical location of a page. This provides a degree of physical database independence. The IS also implements rollback and recovery from system crashes. A detailed discussion of the IS appears in [ROO82].

One of the more novel features of the Intelligent Store is its optimistic concurrency control method. The IS uses a versioning scheme to guarantee each transaction a consistent view of the database as of the transaction's start time (as if the entire database were copied). Note that the IS is really a page manager; it knows nothing about databases (relational or otherwise). Thus, the IS may be used for implementing network or other data models as well as providing a base for other services, such as a secure file manager.

2.3 Record Manager

The Record Manager (RM) [LIN82] provides access to data stored in the form of records. It maps RM-style records into IS pages. Variable size records with missing and repeated fields are supported. The RM supports multiple record types, each of which must have an associated primary index. Record types may have any number of secondary indexes. Retrieval requests are associative: they specify only the record type and a search expression that selects the desired records. The physical aspects of storing and retrieving index and data pages are handled by the IS. The RM accommodates multiple concurrent users (transactions), both updates and retrievals, and relies on the concurrency control provided by the IS. The RM operations are limited to those that can be computed in one pass over one record type, thus including select and project, but excluding sort and join.

Since the RM user interface is associative, details of database and record structure are hidden from the user. This is important because it provides a degree of logical data independence. As with the IS, the RM is not tied to any data model and it may be used as a base for other non-relational database managers. It may also be used for applications which do not require the full generality of a relational data manager and can live with access to only one record-type per request.

2.4 Data Manager

The Data Manager (DM) provides a relational view of data, mapping relations into RM records. It offers a QUEL [STO76] interface and read/write protection of data down to fields within records. The DM uses the RM to process one or more single relation queries. Query processing is accomplished in a pipeline. One or more streams of RM output are sorted, joined, aggregated, and projected by a network of tasks set up to handle the specific query. Use of the pipeline eliminates the need to create temporary relations during query processing. We believe the pipeline orientation will accommodate special purpose processors for selected operations. Like the RM, the DM is supported by the concurrency control and recovery mechanisms provided by the IS.

2.5 Configurations

A JASMIN configuration consists of a database spread across one or more Intelligent Stores (Figure 1). Any number of RM's retrieve and update records in the IS's, each RM having equal access to every IS. Similarly, multiple DM's use those RM's in implementing relational operations on the database. However, a given DM may access multiple sets of RM's, providing access to different databases. Thus, an IS manages a set of pages, an RM manages a set of record types (a database) and a DM manages sets of databases.

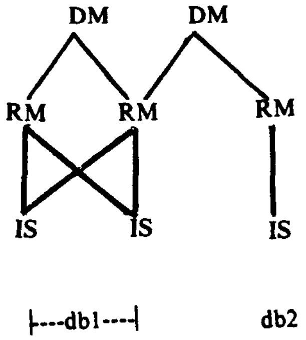

Figure 1

3. DISTRIBUTING THE INTELLIGENT STORE

There are two dimensions to distributing a system: distributed data and distributed processing. In JASMIN, distributed data implies the use of multiple Intelligent Stores, each managing a partition of the database. Distributed processing implies running multiple IS, RM and DM modules on separate processors. In this section, we describe the problems of coordinating multiple Intelligent Stores. To motivate the problems, we first describe the IS in a centralized environment.

3.1 Centralized IS

The Intelligent Store employs the optimistic approach to concurrency control. In this method, a transaction has three phases of execution: read, validate, commit. During the read phase, a transaction reads objects (i.e. pages) from the public database. An object is written by creating a new version of the object, leaving the public database unchanged (until commit). When finished reading and writing, a transaction enters the validation phase. This phase ensures that the transaction, if allowed to commit, will produce a valid serialization order. During the commit phase, the transaction's updates are applied to the public database.

The IS concurrency control method differs from [KUN81] and [SCH81] in that the Intelligent Store uses a versioning scheme to guarantee that transaction sees a consistent view of the database as of that transaction's start time. Thus, each transaction sees the output of all transactions that committed before it started but sees no output from transactions which committed after it began. When a transaction, T, starts, it is assigned a read timestamp, rts(T). Any read request by T returns the version of the object that existed in the database at time rts(T). Note that read-only transactions always see consistent data and, thus, always commit without violating database consistency.

When a transaction enters its validation phase, it is assigned a write timestamp, wts(T). The validation procedure checks that the part of the database seen by the transaction has not changed during the lifetime of the transaction. Specifically, let RS denote the readset of T and WS the writeset, and let Ti denote any transaction that committed between rts(T) and wts(T). Validation computes the set:

$$(\cup \text{ WSi}) \cap (\text{RS} \cup \text{WS})$$

where ∪ denotes set union and ∩ denotes set intersection. If this set is empty, then T is committed. Otherwise T is aborted.

During the commit phase, the new versions of objects written by T are made the public versions. Note that an object may have more than one version (the old versions are needed to give other transactions a consistent view of the database). Eventually, old versions are destroyed ("retired") when they are no longer needed.

In this scheme, update transactions serialize in order of their write timestamps. Read-only transactions serialize in order of their read timestamps. Also, the validate and commit phases constitute a critical section on the set of objects that are being updated.

Crash recovery uses a shadowing approach, that is different from the scheme in [LOR77]. In response to a write request, a shadow page is created. There is no in-place updating. The commit procedure is done in two phases. First, the page map blocks for the shadow pages are written to intention lists in stable storage. Then, the page map blocks are swapped using a careful replacement algorithm and the old version is kept for read requests to older versions. Each phase is executed atomically. This approach never needs to undo a commit, but may need to redo a commit (after a crash). This shadowing scheme has been described in [ROO82] and [AGR83].

3.2 Coordinating Multiple Intelligent Stores

The read phase in the distributed case is similar to the centralized case, except the timestamp chosen by the IS that initiates the transaction must be a global timestamp. The creation of unique global timestamps is based on the algorithm in [LAM78]. If the transaction visits another IS (becomes global), the new IS must use the read timestamp chosen by the initiating IS. If the new IS cannot provide a consistent view of the database as of that read timestamp, the transaction must abort.

When the transaction enters the validation phase, we use the two-phase commit protocol to coordinate multiple IS's. Any IS participating in the transaction may act as coordinator. First, the coordinator chooses a global timestamp for the validation phase. Then, the coordinator sends a message to each participating IS to start its validation phase. Each IS validates the transaction independently and sends a vote back to the coordinator. The coordinator then initiates the commit phase by sending a commit or abort message to the other IS's. Note that the use of globally unique timestamps ensures that transactions will validate in the same order at every IS.

3.2.1 Nonatomic Validate-Commit In the centralized IS, the validate and commit phases were one critical section; i.e. if the transaction was successfully validated, the updates were immediately made public. However, the use of two-phase commit means that validate-commit cannot be atomic. This has two major consequences. First, the consistency model changes. Formerly, a transaction saw the output of any transaction which committed before it started. Now, in the distributed case, a transaction sees the output of any transaction that *validated* before it started. The second consequence is that an IS does not know the outcome of a transaction at validation (it must wait until the commit phase). This makes it difficult to implement the consistency model, since validating a transaction does not make it safe to release its updates (the transaction may yet abort). These "pending" transactions (those between validation and commit) must be accomodated in both the read phase and the validation phase.

The effect of pending transactions in the validation phase is that the validation test must change slightly. The test,

$$(\text{RS} \cup \text{WS}) \cap (\cup \text{ WSi}) = \emptyset$$

must include the writesets of pending transactions as well as committed transactions. This may generate unnecessary aborts (e.g. if a transaction conflicts with a pending transaction that later aborts). But the alternative is waiting for the outcome of the pending transaction to be determined. This introduces deadlock.

In the read phase, we could have the situation where a transaction needs to read the output of a pending transaction (i.e. its read timestamp is greater than the write timestamp of the pending transaction). There are several ways to do resolve this. [ROO80] suggests locking the write set of pending transactions. Conflicting reads would be deferred until the pending transaction finished. Another possibility is to reset the read timestamp of the reading transaction to precede the write timestamp of the pending transaction. In this case, the reader gets the old version of the updated object. Of course, this only works for read-only transactions.

Our plan is to release the output of the pending transaction to the reader. This scheme is also proposed in [SCH81]. If the pending transaction subsequently aborts, the reader must also abort (possibly propogating more aborts). This solution has the advantage of not blocking a transaction during its read phase. However, it may block transactions at commit (as they wait for pending transactions to be resolved). In particular, read-only transactions may be blocked. Under the optimistic assumption, conflicts are rare and most (pending) transactions succeed. So, we expect this to be a safe solution.

3.2.2 Out-of-Order Requests One problem with basing the consistency model on transaction timestamps is that start and commit requests may arrive out of timestamp order. There are two situations where this can occur. An IS might receive a transaction start request with a read timestamp earlier than an already committed transaction. This is easy to fix; aborting a start request is cheap. The other situation involves a read request. Suppose transaction T2 with start timestamp n reads object x. Later, transaction T1 updates object x and commits with write timestamp n-1. Since T2 has a later timestamp it should see T1's version of x. But since T2 has already read the old version, we are out of luck. [ROO80] gives a solution for this which involves putting constraints on the commit time of T1.

3.2.3 Version Retiring A final issue is version retiring. In a centralized system, it is a simple matter to determine that a version is no longer needed. One only needs to keep track of the oldest active transaction. When that transaction finishes, the prior versions of pages it updated may be retired, since there is no older transaction that would want to read those versions. So, the problem in the distributed case translates into the problem of determining the oldest active transaction among all sites. This can probably be done most easily by piggybacking the time of the oldest active transaction onto the clock synchronization messages which must be periodically passed among sites.

In summary, there are several approaches to distributed optimistic concurrency control ([SCH81], [CER82]). We do not have space to compare these schemes. However, our approach is different in that it integrates a versioning mechanism with the concurrency control mechanism to provide transactions a consistent view of the database. The issues of managing replicate data is discussed in

[WIL84].

4. METADATA ACCESS AND UPDATE

Metadata refers to the information managed by each module to describe the user data under its control. There are three levels of metadata in JASMIN, the IS, RM and DM metadata. The IS metadata contains only local information about the contents of the disks under its control. Since the RM provides access to a single database, its metadata contains descriptors for each record type in the database. The RM also maintains distribution information for records that are dispersed across multiple IS's. The DM provides access to multiple databases. In addition to the information required by the RM, the DM also maintains database descriptors. Due to space limitations, we only describe the mechanisms used to access and update RM metadata. The technique for DM metadata is similar.

The metadata for each RM is stored as a separate record type in a single IS. We could distribute the metadata across multiple IS's, but the volume of data is so small as to not merit distribution. Also, distributed metadata would require meta-metadata to describe the distribution criteria so we are back to the original problem. Because the metadata must be accessed by every transaction, it is reasonable to cache a database's metadata in every RM serving that database. We assume the amount of metadata is small enough to be wholly kept in a main memory cache in the RM. We also assume that metadata changes are infrequent (or else there is little benefit in caching it).

Recall that our concurrency control model provides each transaction with a snapshot of the database as of the transaction start time. This also applies to metadata. We assume that each transaction gets its own (consistent) version of the metadata at start time by copying the RM's cache. However, this scheme requires that the RM metadata cache always be up-to-date. Since there are multiple RM's, each with their own cache, we have a cache update problem.

There are really two problems for an RM cache: detecting that the cache is out of date and detecting conflicts between simultaneous updates of the metadata. The source of these problems is that transactions get their metadata from the RM cache rather than by reading it from the IS. This circumvents the concurrency control mechanism provided by the IS. The IS is unaware that the metadata has been read so it cannot detect read-write conflicts.

4.1 Cache Version Numbers

Our solution is to use cache version numbers (CVNs). A CVN is an integer (part of the database metadata) which is incremented whenever the metadata changes. Each RM has a copy of the CVN as does the IS containing the metadata. When a transaction starts at an RM, the RM compares its CVN with the IS CVN. If the RM CVN is less than the IS CVN, the RM cache is out of date and must be refreshed by re-reading metadata from the IS. This detects the outdated cache and provides a new transaction with the latest version of the database metadata.

To solve the concurrent update problem, at validation time the RM compares its CVN with the IS CVN. If they are the same, then the metadata has not changed during the lifetime of the transaction so it may commit (assuming no other conflicts). If the RM CVN and IS CVN differ, another transaction has updated the metadata, so the transaction must be aborted. When a transaction wants to update the metadata, it first makes changes to its local copy of the metadata. It must also issue page writes to the IS to update the disk version of the metadata (this includes a new version of the page containing the CVN). At validation time, (assuming validation succeeds), the RM running the transaction instructs the IS to increment its CVN, copies the transaction metadata cache to to its own RM cache and increments its own CVN. If two transactions concurrently udpate the metadata, they must both update the disk page containing the CVN. The IS will detect this write-write conflict and one of the transactions will abort.

4.2 Read-Only Cache Protocol

Use of the CVN introduces a performance problem. Whenever the metadata is updated (causing the IS CVN to change), all active transactions are aborted. A malicious (or ignorant) user could cripple JASMIN by repeatedly forcing cache updates, reducing throughput to zero. The problem may be avoided by using the following optimization. In most cases, transactions that only read metadata need not be aborted when another transaction concurrently updates metadata. Below, we give conditions on when this is true and an informal argument for its correctness. A more formal proof requires a formalization of the notions of metadata and database consistency.

Recall how the IS detects conflicts. Assume transaction T1 updates metadata and commits during the lifetime of T2. Then T1 and T2 conflict iff:

$$(RS2 \cup WS2) \cap (WS1) \neq \emptyset$$

Since the readset of a transaction is composed of metadata and user data, we may rewrite the conflict definition:

$$(MRS2 \cup URS2 \cup WS2) \cap (WS1) \neq \emptyset$$

or

$$(MRS2 \cap WS1) \cup (URS2 \cap WS1) \cup (WS2 \cap WS1) \neq \emptyset$$

where MRS is the metadata readset and URS is the non-metadata readset. By reading metadata objects from the RM cache instead of the IS, we prevent the IS from detecting metadata read-write conflicts (since it never sees the metadata read). In effect, we eliminate the first term in the conflict definition above. The interesting question is: when may we safely ignore this type of read-write conflict without violating database consistency?

In the example above, the serialization order for the transactions is T1, T2 (since transactions serialize in validation order). Consider the metadata read-write conflict ($MRS2 \cap WS1$) in detail. Although the transactions conflicted when run concurrently, we would like to know if the conflict "really mattered", i.e. if we ran the transactions serially, would the output of T2 change? If not, then the conflict may be safely ignored. When run serially, the T1 metadata update could affect the output of T2 in three ways:

(1) The update has no effect on T2 (e.g. adding a new field to a record descriptor).

(2) The update affects the output of T2 directly. In this case, T2 used metadata to compute an output value (e.g. counting the number of record types).

(3) The metadata update indirectly affects the output of T2 (e.g. a record type is deleted).

The first case does not violate database consistency since T2 is unaffected by T1. It may safely be ignored. However, in the second and third cases the output of T2 may change. To preserve database consistency the conflict must be detected.

The second case may be detected by the following simple rule: the metadata cache may only be read for query compilation. If a user explicitly requests to read metadata (as part of transaction execution), the metadata must be read from the disk (not from the RM cache). In this way, the IS concurrency control mechanism gets involved and will detect metadata read/write conflicts.

The source of the problem in the third case is that a side-effect of updating the metadata has caused a conflict that could affect other transactions (e.g. deleting a record type causes all the record instances to be deleted). However, we claim that such conflicts will always be detected by their side-effects, i.e. as read-write or write-write conflicts involving user data. The reason is that there are implied consistency constraints between the metadata and other parts of the database. When the metadata is updated, those parts of the database must also be affected (e.g. if T1 deletes a record type, it also deletes all record instances). Any transaction that would be indirectly affected by the updated metadata will access that part of the affected database. This allows the IS to detect a conflict on user data.

Thus, metadata read-write conflicts may be safely ignored so long as the metadata cache is not used to provide data values during query execution. This result applies to any concurrency control scheme where transactions serialize in their validation order. To take advantage of this, we modify our CVN protocol slightly. Instead of aborting all active transactions when the metadata changes, we need only abort transactions that concurrently update the metadata.

5. SUMMARY AND FUTURE WORK

In this paper, we described the architecture of JASMIN, a functionally distributed database machine which uses replicated software modules (DM, RM, IS) to achieve high degrees of throughput. We then discussed some issues in distributing data and metadata in JASMIN. We adopted the distributed multiversion validation technique along with the two phase commit protocol to achieve concurrency control and crash recovery for data and metadata. The scheme also solves the version consistency problem in the multiprocessor environment.

An important goal for JASMIN is that it be useful as a testbed for ongoing research in database machines and database management in general. Our future work with JASMIN will include:

- Construction of a file system on the IS and its integration with the DM. This will permit additional data types such as text, graphics, and voice in the database.
- Study the reliability of JASMIN configurations and design of enhanced reliability features.
- Study the advantages and disadvantages of optimistic concurrency control for centralized and distributed databases, and comparison of its performance relative to locking.
- Study the performance of various configurations of DM, RM, and IS modules.

6. ACKNOWLEDGMENTS

We would like to thank Rakesh Agrawal, Dan Fishman, John Linderman, and Bill Roome for their insightful suggestions and comments on this paper. The JASMIN project is a continuation of earlier work (through the uniprocessor prototype) done at Bell Laboratories. Quite a number of people have made significant contributions to this project. These include, in addition to the authors, Micah Beck, Bill Burnette, Rudd Canaday, Dan Fishman, Susan Fontaine, Mary Hesselgrave, Phil Karn, Hikyu Lee, John Linderman, U. V. Premkumar, Bill Roome, Edith Schonberg, and Chung Wang.

7. REFERENCES

[AGR83] Agrawal, R., and DeWitt, D. J. "Integrated Concurrency Control and Recovery," *Computer Sciences Technical Report #497*, University of Wisconsin, Madison, Wisconsin, 1983.

[BAN78] Banerjee, J., Baum, R. I., and Hsiao, D. K. "Concepts and Capabilities of a Database Computer," *ACM TODS 3*, 4 (Dec. 1978), 347-384.

[BAS77] Baskett, F., Howard, J. H., and Montague, J. T. "Task Communication in DEMOS," *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, November 1977, 23-31.

[CER82] Ceri, S., and Owicki, S. "On the Use of Optimistic Methods for Concurrency Control," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, Feb. 1982.

[FIS84] Fishman, D. F., Lai, M. Y., and Wilkinson, W. K. "An Overview of the JASMIN Database Machine," *SIGMOD 84*, June, 1984

[GAR83] Gardarin, G., Bernadat, P, Temmerman, N., Valduriez, P., and Viemont, Y. "Design of a Multiprocessor Relational Database System," *INRIA Technical Report, TR 224*, July, 1983.

[KAM79] Kamibayashi, N., Kato, H., Kiyoki, Y., Ozawa, H., Seo, K., and Aiso, H. "SPIRIT: A New Relational Database Computer Employing Functional-Distributed Multi-Microprocessor Configuration," *International Conference on Distributed Computing Systems*, June, 1979.

[KUN81] Kung, H. T., and Robinson, J. T. "On Optimistic Methods for Concurrency Control," *ACM TODS 6*, 2 (June 1981), 213-226.

[LAM78] Lamport, L. "Time, Clocks and Ordering of Events in a Distributed System," *Comm. of ACM*, July 1978, 558-565.

[LIN82] Linderman, J. P. "Issues in the Design of a Distributed Record Management System," *Bell System Technical Journal 61*, 9 (Nov. 1982), Part 2, 2555-2566.

[LOR77] Lorie, R. L. "Physical Integrity in a Large Segmented Database," *ACM Transactions on Database Systems*, March, 1977, 91-104.

[ROO80] Roome, W. D. Unpublished work, Bell Laboratories, Murray Hill, NJ, June, 1980.

[ROO82] Roome, W. D. "A Content-Addressable Intelligent Store," *Bell System Technical Journal 61*, 9 (Nov. 1982), Part 2, 2567-2596.

[SCH81] Schlageter, G. "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. VLDB*, 1981.

[SOL79] Solomon, M. H., and Finkel, R. A. "The Roscoe Distributed Operating System," *Proceedings of the 7th Symposium on Operating Systems Principles*, December 1979, 108-114.

[STO76] Stonebraker, M., Wong, E., Kreps, P., and Held, G. "The Design and Implementation of INGRES," *ACM TODS 1*, 3 (Sept. 1976), 189-222.

[STO81] Stonebraker, M. "Operating System Support for Database Management," *CACM 24*, 7 (July 1981), 412-418.

[WIL84] Wilkinson W. K. and Lai, M. Y. "Managing Replicate Data in JASMIN," Submitted to *4th Symp. on Reliability in Distributed Software and Database Systems*, 1984.