Overview of an Ada* Compatible Distributed Database Manager

Arvola Chan
Umeshwar Dayal
Stephen Fox
Nathan Goodman**
Daniel R. Ries
Dale Skeen***

Computer Corporation of America
Four Cambridge Center
Cambridge, Massachusetts 02142

Abstract

Adaplex is an integrated language for programming database applications. It results from the embedding of the database sublanguage DAPLEX in the general purpose programming language Ada. This paper provides an overview of the DDM: a distributed database manager (DDM) that supports the use of Adaplex as an interface language. The important technical innovations we have incorporated in the design of this system include:

1. An advanced data model that captures more application semantics than conventional data models.
2. Support for flexible data distribution options that improve locality of reference and efficiency of query processing.
3. Extensive query optimization that combines compile time access path optimization with run time site selection.
4. Efficient transaction management that reduces transaction conflicts and improves the resiliency of replicated data.
5. Robust, incremental recovery management that provides for automatic recovery from certain "catastrophic" failure conditions.

---

*Ada is a trademark of the Department of Defense (Ada Joint Program Office).

**Nathan Goodman's current address: Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138.

***Dale Skeen's current address: Computer Science Department, Cornell University, Ithaca, NY 14853.

This project is supported jointly by the Advanced Research Projects Agency of the Department of Defense (DARPA) and the Naval Electronics Systems Command (NAVELEX) under contract N00039-82-C-0226. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NAVELEX, or the U.S. Government.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM -0-89791-104-0/83/005/0228 $00.75

1. Introduction

Many military computer applications require access to information stored in databases. In the future, these applications will be implemented by programs written in the DoD's new programming language, Ada. Therefore, it is essential that database management capabilities that are closely coupled to Ada be made available. Toward this end, Computer Corporation of America (CCA) has designed Adaplex, the Ada-based composite language for programming database applications. Adaplex results from embedding in Ada the database sublanguage Daplex [SHIP81]. Two systems that support the Adaplex language are currently being developed at CCA. The implementation of a centralized DBMS, called the Local Database Manager (LDM), has been underway since October '81, and it is scheduled for completion in March '84. The LDM has been designed as a high-performance stand-alone system for managing local data [CFLR81] [CFLN82] [CDFL82]. The design specifications of a distributed DBMS, called the Distributed Database Manager (DDM), which interconnects multiple LDMs in a computer network, recently has been completed [CDFG82] [CDFG83]. Implementation of the DDM began in October '82.

Functionally, the DDM provides the same capabilities that one expects of any modern centralized DBMS. Like the LDM, the DDM supports applications written in either Daplex or Adaplex. Each user interacts with a distributed database precisely as if he were accessing a single-user, centralized, integrated database. The DDM shields users from all complexities introduced by data distribution, replication, and potential site failures. More specifically, the DDM provides the following important facilities:

1. An integrated global schema that encompasses data stored at all sites. The DDM maintains a global directory in order to keep track of the distribution and replication of data. It automatically maps transactions on the global schema into subtransactions on data stored at individual LDMs.

2. Complete physical data independence. The database administrator (DBA) is free to tune parameters involving the physical distribution, replication, and organization of the stored data, without affecting the external view of the database.

3. Mutual consistency of replicated data. Users deal with logical data only. The propagation of updates to redundant copies of updated data is managed by the DDM.

4. Continued operation in spite of site failures. Users can continue to perform update operations on redundantly stored data, even though some copies may be temporarily inaccessible. These latter copies are brought up to date by the DDM before being used for processing subsequent transactions.

5. Interleaved executions of multiple transactions. The DDM guarantees "atomicity" for each transaction. No partial effects of one transaction will be seen by another; and if a transaction is unable to complete, all of its effects on the database are undone automatically.

6. Efficient access to replicated data. The DDM will optimize the selection of sites that are used for accessing replicated data. This selection takes into consideration the availability of sites at run time and the actual data requirements of a given transaction.

7. Dynamic integration of new sites into the system. No quiescence of on-going activities is necessary for reconfiguration of the system.

In the following sections, we give an overview of the DDM's mechanisms for handling distributed and replicated data. Section 2 summarizes the database management facilities provided by the system. Section 3 outlines facilities for distributed database administration. Section 4 sketches the decompilation approach we employ for supporting the Adaplex language. Section 5 highlights our strategies for optimizing Daplex transactions. Section 6 summarizes our mechanisms for synchronizing concurrent transactions. Section 7 identifies our innovations in the area of recovery management. Finally, Section 8 reports the implementation status of the system.

2. The Adaplex Database Languages

The core language for the definition and manipulation of databases, both in the DDM and in the LDM, is called Daplex [SHIP81]. Daplex supports the notions of entities, functions, and generalization hierarchies [SS77]. These high level concepts provide more modelling capabilities than conventional hierarchical, network, or relational data model. Daplex has been extended to provide two user interfaces:

1. The Adaplex Programming Language Interface is designed primarily for application programmers. It integrates the complete programming language Ada with Daplex.(1) In addition to data definition and manipulation facilities, the Adaplex Programming Language Interface also provides access to a report writer utility. Furthermore, database administrators can define views, Ada procedures, and computed functions for controlling access to data.

2. The Interactive Language Interface is designed for use by both application developers and database administrators. Interactive users have complete access to the Daplex language. They also can create temporary databases, interface with a report writer utility, a bulk load utility, and a test data generator utility. In addition, database administrators can enter database authorization specifications and interface with a database reorganization utility.

3. Distributed Database Design

An important design goal of the DDM is to realize the advantage of locality of reference, by storing data items where they are most frequently used. In order to achieve locality of reference for retrieval transactions, it is often desirable to introduce replication. At the same time, replication is needed for improving data availability in the presence of site failures. Flexible options for database fragmentation and replication are supported in the DDM.

3.1 Fragmentation and Grouping

In order to provide efficient support for Daplex's function inheritance semantics(2) without introducing undue complexity, it is preferable to keep all information about a conceptual object at one place. Therefore, we do not consider the vertical fragmentation of properties of any given entity across sites. At the same time, Daplex allows for "direct" pointers that point from one entity to another. In order to facilitate the maintenance of such pointers, we introduce the notion of a <u>fragment group</u> (FG) for collecting together related entities. That is, we put related fragments from different generalization hierarchies together into groups, and we use these FGs as units for physical design. We expect that most databases can be designed in such a way that frequently followed inter-entity references, from entities within

---

(1) No changes whatsoever have been made to the existing Ada language. New (non-Ada) constructs have been introduced in ways that clarify their similarities to (and differences from) existing Ada constructs. The objective has been to make Adaplex easy to learn by trained Ada programmers.

(2) That is, a subtype inherits all functions applicable to its supertypes. In processing an entity of a given type, the Adaplex language allows for access to functions of all relevant supertypes.

a given FG, can be made self-contained (i.e., local to the FG). This will eliminate the cumbersome maintenance of nonlocal pointers. It is important to note that we do not require FGs be defined in such a way that all interentity references are localized. However, interentity references that span FG boundaries have to be processed through (more expensive) value-based join operations on distributed data.

Since we have ruled out the possibility of vertically partitioning the functions (attributes) of individual entities across sites, all fragmentation definition must be on the base types of generalization hierarchies. Partitioning of entities within a subtype can be implicitly defined through partitioning of the corresponding base type. Each fragment of a base entity type is defined in terms of a conjunction of conditions. Each condition can specify subtype membership, subtype nonmembership, or value for a single-valued function. Since fragments are required to be disjoint, their corresponding definition predicates must be non-overlapping.

In addition to fragmentation based on local properties of entities, we also support fragmentation based on associations. (In relational terminology, we support fragmentation based on join conditions.) The fragmentation of a generalization hierarchy can be made dependent on that of another generalization hierarchy. (This dependence must be induced by a single-valued function mapping entities from the former hierarchy to entities in the latter.) For example, assume there is a single-valued function mapping person entities to department entities. If a department fragment D_1 for storing department entities that are located on the first floor has been defined, then the specification of a corresponding person fragment P_1 for storing person entities that reference department entities in D_1 also is permissible. However, the fragment P_1 is required to be stored in the same FG as the fragment D_1.

3.2 Fragment Group Organization

Each group of logically related fragments is treated as a unit for replication, distribution, and physical organization. We distinguish two kinds of replicated copies of an FG. Let X be an FG that is to be replicated n times, at sites $S_1, \ldots, S_n$. The DBA can designate k (less than or equal to n) of these as regular sites for the FG. The remaining (n-k) sites are then treated as backup sites for the FG. Regular sites are intended for storing copies to be used during normal transaction processing; backup sites are used to improve resiliency in the presence of site failures. (The treatment of replicated data on transaction update and site recovery is discussed in Sections 5 and 6.)

The physical organization for each FG is specified in the same way as physical organization is specified for a local database in the LDM. For implementation simplicity, we require that all copies of an FG be organized identically. Discussions on the physical design options available in the LDM can be found in [CDFL82].

The DBA is free to change the replication, distribution, and physical organization parameters at any time. The DDM is designed to accommodate all reorganization of a physical nature (i.e., including changes in distribution and replication parameters) without incurring the overhead of rewriting or recompiling application programs.

4. Implementing the Adaplex Language

For the purpose of portability, A Preprocessor is used to translate an Adaplex program into a modified, pure Ada program with embedded calls to the underlying DBMS. One complication arises from the tight coupling of Daplex with Ada. While this tight coupling enhances the ease of use of the resultant composite language, it often will not be efficient to perform the database access operations in the order specified. In order to facilitate access path optimization, the Preprocessor employs a decompilation strategy. It first isolates the database access (Daplex) component of an Adaplex transaction specification from its general purpose computation (Ada) component. It then converts the Daplex component into a nonprocedural internal representation, called an envelope, which is in a form more amenable to optimization. The envelope is passed on to the DBMS to describe the data that is needed for running the entire transaction.

4.1 Decompilation Advantages

Decompilation is intended to yield two important performance advantages: it reduces synchronization overhead in the system; it also reveals opportunities for access path optimization.

4.1.1 Synchronization Reduction

In many database management systems, the task architecture shown in Figure 1 is employed. All data management functions, such as concurrency control and actual physical data access, are carried out by a separate DBMS task. This architecture

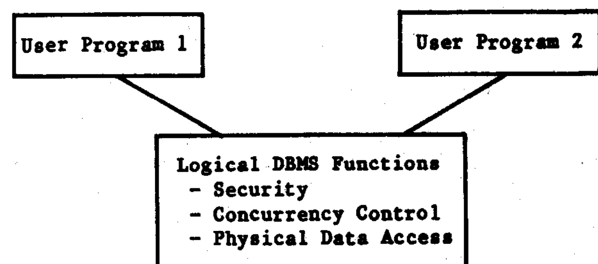

Figure 1. DBMS Architecture

serves to prevent undue corruption of shared data by erroneous user programs. The use of such a task architecture, in a DBMS that supports only record-at-a-time retrievals and updates, will result in two context switches between the user's task and the DBMS task for each accessed record. The expense of this communication, as compared to using high-level DBMS commands that manipulate sets of records, has been discussed in [RS81].

If the LDM/DDM simply translates the "for each" loops in an Adaplex program into "get first" and "get next" operations, the same excessive communication overhead will result. Instead, the Preprocessor converts a number of nested "for each" statements into a non-procedural set-oriented expression called an envelope. The result of executing an envelope is a sequence of hierarchically related records that correspond to entities selected from the database. The modified, pure Ada program makes a single invocation on the underlying DBMS to obtain the collection of data described by an envelope. Either the entire set or a suitably buffered subset of the required data is returned to the user task with a single context switch.

4.1.2 Access Path Optimization

The separate identification of a required set of data permits the DBMS to perform more thorough optimization. The decompilation of Adaplex, in fact, allows for more optimization than is normally found in embedded relational query language systems like Pascal/R [SCHM77] and RIEGEL [RS79]. These latter systems optimize relational expressions that govern the execution of individual iteration loops only. Decompilation in the LDM/DDM, on the other hand, identifies a set oriented expression that describes the data required in a number of arbitrarily nested loops constituting all or part of a database transaction.

The need for access path optimization is illustrated by the query in Figure 2.

This query retrieves the names of students who have taken 4-credit courses, along with the titles of those courses. The obvious approach for process-ing this query is to follow the control structure of the program: a student entity is retrieved; then a course entity in which the student is enrolled is retrieved; if the course is a 4-credit one, then the student's name and the course's title are printed. The retrieval of course entities is repeated until all of the courses for the given student have been exhausted. Next, another student entity and the corresponding course entities are retrieved and processed. This is repeated until all of the student entities have been considered. If the database contains many student entities, and comparatively few 4-credit course entities, and there exists direct access paths for locating the 4-credit courses, then an alternate approach may be better. First, the 4-credit courses are retrieved. Next, the students who are taking those courses are retrieved. This can be done by exploiting available auxiliary structures that represent the inverse of the enrollments function, or by performing a relational join operation. Finally, the name and title pairs are grouped by student. This can be achieved by sorting on the internal identifiers of student entities. An important purpose of decompilation is to decouple the data selection requirements from the output format requirements, in order to uncover such optimization opportunities.

```
for each S in STUDENT
 loop
  for each C in ENROLLMENTS(S)
   where CREDITS(C) = 4
    loop
     PRINT(NAME(S), TITLE(C));
    end loop;
 end loop;
```

Figure 2. An Adaplex Program Segment

4.2 Preprocessing

An Adaplex program goes through a preprocessing stage in order to prepare for multiple run-time invocations. Preprocessing results in a pure Ada program with embedded procedure calls to the underlying DBMS and embedded procedures for unbuffering data retrieved from the database. During the preprocessing stage, the envelopes for defining the data needed by the program are identified; the strategy for obtaining data prescribed by each envelope is also optimized. The description of the optimized execution steps are stored away as <u>access plans</u> that can be invoked at run-time.

In the modified, pure Ada program, the original data manipulation constructs are converted into two basic types of procedure calls. The first type of procedure call initiates the execution of a corresponding access plan. The second type of procedure call obtains data needed for each iteration of an Ada looping construct. The Preprocessor generates both types of procedure calls and the looping constructs. It also generates the descriptions for the Ada record structures for holding the scalar values that are used inside the looping constructs.

The run-time execution of an Adaplex program invokes the Ada procedures described above. An envelope invocation procedure will first initialize parameters from values of run-time Ada program variables, if necessary. The underlying DBMS then retrieves the corresponding access plan for execution. The procedures that access data needed by the modified Ada loops will unbuffer the returned data, and will store the database values in the appropriate Ada records, as expected for the execution of the set of statements within each loop.

5. Envelope Optimization

The objective of envelope optimization is to minimize total processing cost, which is computed as a weighted sum of the costs of intersite data movement and local processing. We have extended

distributed query optimization techniques used in existing systems like SDD-1 [BGWR81] and R* [SELI80]. Both SDD-1 and R* focus on conjunctive queries (i.e., queries involving only selections, projections, and joins). SDD-1's strategy is to reduce data by a sequence of semijoins, then transfer the reduced data to a single site, where the result is produced by a final series of join operations. As in R*, the DDM considers mixed join and semijoin strategies. We differ from SDD-1 and R* in the handling of horizontally partitioned data. Consider, for example, the query R $\bowtie$ S, where R is horizontally partitioned into R_1, R_2, and S is horizontally partitioned into S_1, S_2. Both SDD-1 and R* replace the original query by the union of four subqueries, $R_i \bowtie S_j$, $1 <= (i,j) <= 2$. Algebraically, this is equivalent to distributing joins over unions. This is not always a good strategy. It is sometimes cheaper first to construct R and S by unions, and then to join them. In the DDM, we consider the tactic of distributing joins over unions only if it is profitable. (Left distribution, right distribution, and both left and right distribution, are all considered.)

Additional extensions to the DDM's optimizer are necessitated by the richness of Adaplex, compared to the relational interfaces supported by SDD-1 and R*. The implications of the richness of the Adaplex language on query processing in a centralized environment have been detailed in [RCDF83]. The extensions developed there also are applicable to the distributed environment. We sketch a few important extensions here. More details can be found in [DAYA83b].

First, the DDM attempts to optimize several nested "for each" loops in a query simultaneously. Many embedded relational query language systems, such as Pascal/R [SCHM77] and RIGEL [RS79], optimize the qualification of one "for each" loop at a time. Optimization over several loops allows the DDM to consider more join strategies, and it can reduce the communication and synchronization overhead between a user program and the DDM. The _unidirectional outer join_ operation(3) is used to model nested iteration loops with embedded output statements in an Adaplex program. A straightforward approach to processing queries containing unidirectional outer joins would be to process all the regular joins before any unidirectional outer join, since these two types of joins do not commute in general. We could restrict the strategy space to only strategies that correspond to join sequences in which all regular joins precede all unidirectional outer joins. However, in order to increase the strategy space for optimization, we define a new operation, called a _partial join_, with the property that a unidirectional outer join followed by a partial join is equivalent to a regular join followed by a unidirectional outer join [RCDF83]. All three kinds of joins are in fact special cases of the graft operation introduced in [DAYA83b]. In essence, a larger number of join order permutations for an envelope is permissible, provided regular joins are replaced by partial joins, where appropriate.

Second, the DDM optimizes queries with nested quantifiers. In [DAYA83a], it is shown that queries that have mixed quantifiers can be processed using a two-phased technique. In the first or _graft_ phase, all edges representing universal quantification are turned into unidirectional outer join edges, and the resultant query is processed using graft operations. In the second or _prune_ phase, the result of the graft phase is processed to account for quantifications. This graft and prune strategy is potentially more efficient than those based on Codd's Reduction Algorithm [CODD72] [PALE72] [JS82]. The DDM optimizer constructs a good strategy for the graft phase. The result of the graft phase is made available at the transaction's home site, where the prune phase is executed locally.

Third, the DDM optimizer takes advantage of the explicit entity-to-entity functions that are specified in the functional data model. These are implemented by pointers and serve as direct access paths for performing joins between the pairs of entity types that are linked. In case the target entities are stored within other fragment groups, the sets of pointers from the source entities can first be collected in order to pinpoint the set of fragment groups that contain the target entities.

The overall optimization technique employed in the DDM is the following. First, localize selections.(4) Next, collapse joins that can be performed at the same site. Then, enumerate join orders using a leveled hill climbing search strategy: the user can specify an optimization level, N; the optimizer then enumerates join sequences of length N at each stage of the optimization process. When N is one, this heuristic is similar to the "greedy strategy" of INGRES [WY76]. When N is equal to or larger than the number of joins in the query, this heuristic resembles System R's exhaustive enumeration strategy. For a given (partial) join order, and for each join of horizontally partitioned entity sets, determine whether it is cost-effective to distribute the join over the union. For each nonlocal operation, select a site for performing the operation and determine whether it is cost-effective to do semijoin reductions before each join.

In order to amortize the overhead of decompilation and access path optimization, a two-step optimization approach is taken. Access path optimization is performed during the preprocessing stage (i.e., when a repetitive class of transactions is first defined). It involves the ordering of high level database operations and the selection of available access structures to efficiently process these operations. Optimization in this phase does not take into consideration the presence of replicated data. Each fragment group is treated as a distinct logical site, and the cost of an explicit

---

(3) This is similar to one of the dissymmetric joins discussed in [LP76].

(4) The distribution of selection over union is always possible and desirable.

transfer is always included for an operation whose operands are obtained from different fragment groups. The objective of access path optimization at compile (preprocess) time is to minimize total processing cost, using a worst case assumption about data distribution. Run time optimization is performed when an instance of a repetitive transaction class is to be executed. Materialization selection is carried out in this phase. That is, logical sites are bound to physical sites. It takes into consideration the availability of sites, as well as the locality of replicated data that needs to be accessed, in order to select the set of sites for running the transaction. The objective of this phase is to minimize intersite data transmission. Heuristics for materialization selection are discussed in [CDFG83].

6. Concurrency Control

The concurrency control mechanisms for the DDM are generalizations of those used in the LDM [CFLR81] [CFLN82]. We extend the LDM's two-phase locking mechanism in order to synchronize transactions that access distributed and replicated data. We also augment the LDM's multi-version mechanism in order to permit each read-only transaction to read a consistent snapshot of the database, without having to synchronize with concurrent update transactions.

6.1 Update Synchronization

The DDM's concurrency control mechanism for update transactions is based on distributed two-phase locking. In the following discussions, we speak of a distributed transaction as being coordinated by a single coordinator and serviced by a number of distributed agents. The agent of a transaction at a given site is responsible for acquiring locks for the transaction at that site. The processing that needs to be done by an agent is described to the agent nonprocedurally at the envelope level by the transaction's coordinator. The agent then executes the access plan for the envelope and sets locks incrementally as necessary. These locks are set at the page level and are held until the end of the transaction. Because of the incremental locking strategy, deadlocks are possible. They are detected and resolved on a periodic basis, both at the local and global levels.

Each replicated FG copy can be in one of three states: online, offline, or failed. Online copies are accessible and up to date. Offline copies are accessible but out of date. Failed copies are inaccessible. Let X be an FG that is to be replicated at n sites. Let k of these sites be regular sites designated by the DBA. Where possible, the DDM will endeavor to maintain k online copies for FG X. Under normal circumstances, the copies at the k regular sites are kept online. When a regular site that stores an online copy fails, an appropriate offline copy at a backup site is rolled forward and brought online. This is done in order to sustain the degree of resiliency and availability specified by the DBA. Similarly, when a regular site for FG X that has previously failed recovers, its copy is rolled forward and brought online. (A backup site that is storing an online copy will demote its copy to offline status, if appropriate.)

In general, online copies are updated synchronously; offline copies are updated in a background fashion. All online copies of an FG are treated as equals for the purpose of transaction processing. The transaction's coordinator selects a single agent in order to perform read and write operations on a single online FG copy. In case an FG copy is updated, the agent is responsible for propagating the updates to other online copies of the FG. This is done when the agent receives the "End of Transaction" indication from the coordinator. An agent sends positive acknowledgement to the coordinator only when it receives positive acknowledgements from the replication sites that store all the other online copies. Each site that stores a copy of an FG maintains a fragment group status table for that FG. This table identifies the sites that contain online copies of that FG, and it is used by a transaction's agent to determine the set of sites to which updates on the FG must be propagated.

6.2 Multiversion Mechanism

The LDM's multiversion scheme [CFLN82] is extended for operation in the distributed environment. The extended scheme guarantees that each distributed read-only transaction sees a semantically consistent snapshot of the database. Its salient features include:

1. Conflicts between read-only transactions and update transactions are completely eliminated.
2. Read-only transactions never have to be rolled back.
3. Garbage collection of old versions of data objects can be carried out efficiently.

At each LDM, the stored data is organized as a collection of segments. (Each segment is a collection of consecutively numbered logical pages.) There is a single common pool of pages, called the version pool, which is used for storing the old versions of segment pages that are potentially needed by in-progress transactions. Update transactions synchronize by using two-phase locking at the segment page level. Before performing in-place update to a segment page, an update transaction is required to first copy the current version of the page to the version pool. Thus, the multi-version mechanism deals with multiple versions of logical pages. Old versions in the version pool serve dual purposes: rolling back update transactions that have to be aborted, and providing consistent snapshots to read-only transactions.

Consider the example illustrated in Figure 3. The read-only transaction T_r needs to read logical pages P_1 and P_2. T_r has read P_1 when T_u comes along and updates P_2, causing the old version of P_2 to be copied to the version pool. When T_r tries to

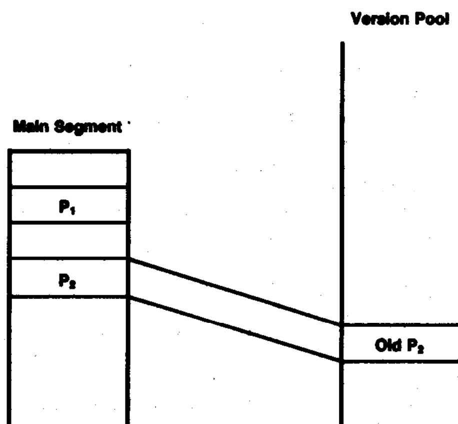

Figure 3. Elimination of Conflicts

read P_2, it still can be made to see a consistent snapshot of the database if it is presented with the old version of P_2, instead of the latest version created by T_u.

The consistent snapshot to be seen by a read-only transaction is defined in terms of a completed transaction list (CTL) that identifies the set of transactions that already have finished execution when the read-only transaction initiates.(5)

An important prerequisite for implementing the multi-version reading algorithm is that it be possible to determine exactly what transactions have completed execution at a given point in time. This is a trivial problem in a centralized environment. For a distributed read-only transaction that reads from multiple sites, the problem is more formidable. The local CTLs are potentially incomplete with respect to transaction dependencies. That is, an update transaction might be added to a CTL at a site before the transactions that it depends upon are added. Our solution for the distributed environment requires the coordinator of the distributed read-only transaction to retrieve the CTLs from sites where the read-only transaction is to perform reading. The coordinator computes the union of retrieved CTLs and retransmits the union-list to the reading sites for use with the multi-version reading algorithm. Some additional requirements are imposed on the atomic commitment protocol: potential transaction dependency information must be propagated during the execution of the commit algorithm. Such dependency information is piggybacked on messages already needed for achieving atomic commitment. Thus, no additional messages will be introduced. Details of the distributed multiversion algorithm can be found in [CG83].

(5) In the DDM, a global transaction identifier can be represented in terms of an originating (coordinating) site identifier, along with a chronologically assigned number that is unique within that site. Thus, a CTL will be made up of a number of sublists. Each sublist is associated with one site and represents completion information for transactions that have been initiated from that site. A sublist can in general be represented by three components: a base number, an exception list, and a bit-map. The exception list identifies transactions whose numbers are below the base number that are still in-progress. The bit-map identifies completed (i.e., committed or aborted) transactions whose numbers are above the base number. The sublist information is periodically recompacted by changing the base number and the corresponding exception list and bit map.

7. Recovery Management

The goals of the recovery management subsystem in the DDM provide for extremely high data availability and resilient transaction processing. The recovery management subsystem consists of three components:

1. The fragment group manager (FGM) provides the desired level of availability for each FG copy, and maintains mutual consistency among copies of an FG.

2. The transaction manager (TM) provides atomic commitment of transactions, even in the presence of failures.

3. The enhanced network (EN) provides the network services required by the other two components.

7.1 The Enhanced Network

The enhanced network layer effects a clean separation between the communication network level and the higher-level managers. In addition to message transmission, it provides services that facilitate failure recovery of TMs and FGMs:

1. It maintains a site status table (SST) that stores the lastest known status of each site in the network (i.e., whether the site is up or down). Higher layers are allowed to query this table.

2. It implements a watch facility whereby a process can register a "recovery watch" or a "failure watch" on a specified site. If the enhanced network layer observes the recovery (failure) of the watched site, it interrupts the registering process.

The SST is implemented by a set of Reliable Status Control Protocols. The goals of these protocols are to propagate failure information (and to a lesser extent, recovery information) in a timely, low-cost fashion. An entry in the SST consists of a unique site identifier, the site's incarnation number, and information indicating whether it is up or down. A site's incarnation number is incremented every time it recovers. By labeling a message with both the sender's id and its incarnation number, it is possible to determine during which of the site's operational periods a given message was sent.

The watch facility is implemented by observing changes in the local SST and it does not require the sending of messages to the watched sites. It eliminates the need for high-level timeouts. A straightforward implementation of watches associates a list of local processes with each site in the SST. The list identifies processes that will be interrupted whenever the SST entry for the corresponding site changes. With this implementation, the watch facility is almost free.

The Reliable Status Control Protocols consist of two components: a component that detects failures and recoveries, and a component that propagates failure/recovery information. The CONTROL protocol lies at the heart of the detection component; the CHANGE protocol lies at the heart of the propagation component.

The CONTROL protocol is essentially the one proposed in [WALT82], modified slightly to interface with our CHANGE protocol. A major difference between the CHANGE protocol used in the DDM and the one proposed in [WALT82] is that we have a "primary" site which is given the responsibility of actually disseminating the status change information. (The use of a primary site provides an efficient means for precluding the anomaly of a message affected by the failure of a site, say A, arriving at another site, say B, before the new SST reporting A's failure arrives at B. This anomaly, if allowed to occur, would lead to errors in our concurrency control algorithm. In order to support the primary site mechanism, the Site Status Table is augmented with an ordering field and a unique version number. The ordering field imposes a linear ordering among the operational sites and determines the order of succession if the current primary should fail. The version number designates the primary site and a number that is assigned in strictly increasing order by the primary site.)

The CONTROL protocol uses a static virtual ring. Each site asynchronously and periodically sends a HI message to its nearest operational successor in the ring and to all of the failed sites in between. A failure of a predecessor is detected when a HI message is not received after the elapse of an interval dependent on the sending rate of HI messages and the maximum message delivery time. Once a failure has been observed, the observer should find and notify the current primary, which is responsible for propagating the failure information.

Our mechanism for the propagation of status change information combines an active component and a passive component. The active component requires that each observation of a failure (recovery) be broadcasted to all sites. It is remotely based on the CHANGE protocol proposed in [WALT82]. The passive component piggybacks recent failure information on messages that may have been influenced by the failure, by appending the version number of the sender's SST to each message. At the receiving site, the Enhanced Network delays delivery of a message until its own version of the SST is at least as recent as the sender's. Thus, failure information is guaranteed to be delivered to sites that might be influenced by the failure, at a rate determined by the intensity of the message traffic.

7.2 Fragment Group & Transaction Management

At the conceptual level, a (replicated) FG is managed by an abstract process known as its fragment group manager. Such a process can be accessed directly from a number of different sites and is extremely fault-tolerant. A given FGM provides facilities to read and update one FG. Physically, an FGM is a closely-coupled confederation of processes (at least one process executing at each site that contains a copy), communicating among themselves in order to maintain mutual consistency. Each of these processes watches out for the failure of the others. Each is responsible for deciding, in a decentralized fashion, if its stored copy of the FG should be brought online or offline. It communicates with other processes within the confederation in order to decide what portion of the recovery log stored at one site should be sent over to the other, in order to roll-forward an out-of-date FG copy. Thus, recovery of replicated FG copies at a recovering site is performed in an incremental fashion [ABG82], one FG at a time, at the initiative of the recovering site. As soon as the locally stored copy of an FG has been rolled forward, it can be made accessible to new transactions. Thus, the availability of data stored at a recovering site is improved.

The execution and commitment of a transaction is managed by an abstract process called the transaction manager. The TM interacts with the managers of all FGs involved in the transaction. With respect to recovery management, its primary responsibility is to commit the transaction atomically at all involved FGMs. Like the FGM, a TM is physically implemented as a closely-coupled confederation of processes that span multiple sites in order to achieve high fault tolerance.

Our design for the FGM and TM in the DDM solves a number of reliability problems not addressed in previous prototypes like R* [WDHL82] and SDD-1 [RBFG80]. R* uses a two-phase commit protocol [GRAY78] that is liable to block if the site that is coordinating the commitment fails. SDD-1 makes use of backup coordinators in order to improve resiliency [HS80]. However, no provision is made in SDD-1 to allow for recovery from the situation wherein the primary coordinator and all its backups fail simultaneously. The DDM uses an improved version of SDD-1's commit algorithm in order to permit automatic recovery from such commit

catastrophes. Whereas R* handles no replication, SDD-1 makes use of a spooler mechanism to collect update messages destined to a nonoperational site in order to facilitate the site's recovery [HS80]. When all spoolers for a given site fail at the same time, a spooler catastrophe is said to occur, and human intervention becomes necessary for recovery. The DDM makes use of the audit trails at replication sites in order to recover a failed site. It is designed to recover automatically from a total failure situation wherein all replication sites of a given FG have failed simultaneously.

8. System Status

The design specifications for the DDM recently have been completed. Implementation is now underway. In order to ensure the portability of the resulting system, Ada is being used as the implementation language. The initial system is targeted for Digital Equipment Corporation's VAX 11/780 running the VMS operating system. A two-phase implementation is planned. The first phase is scheduled for completion in September '84. It will not include implementation of the Adaplex Language Preprocessor, the Report Writer, the View Mapping Processor, and the Reorganization Utility. These latter modules are to be implemented in the second phase.

9. References

[ABG82]
Attar, R., B. Bernstein, and 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.

[BGWR81]
Bernstein, P. A., N. Goodman, E. Wong, C. Reeve, and J. B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981.

[CDFG82]
Chan, A., U. Dayal, S. Fox, N. Goodman, D. Ries, and D. Skeen, "DDM Design Specifications," Computer Corporation of America, 1982.

[CDFG83]
Chan, A., U. Dayal, S. Fox, N. Goodman, D. Ries, and D. Skeen, "DDM: An Ada Compatible Database Manager," COMPCON Digest of Papers, 1983.

[CDFL82]
Chan, A., S. Danberg, S. Fox, W. K. Lin, and D. Ries, "Storage and Access Structures to Support a Semantic Data Model," VLDB Conference Proceedings, 1982.

[CFLN82]
Chan, A., S. Fox, W. K. Lin, A. Nori, and D. Ries, "The Implementation of an Integrated Concurrency Control and Recovery Scheme," ACM SIGMOD Conference Proceedings, 1982.

[CFLR81]
Chan, A., S. Fox, W. K. Lin, and D. Ries, "The Design of an Ada Compatible Local Database Manager (LDM)," Technical Report CCA-81-09, Computer Corporation of America, November, 1981.

[CG83]
Chan, A., and R. Gray, "Implementing Distributed Read-only Transactions," submitted for publication.

[CODD72]
Codd, E. F., "Relational Completeness of Database Sublanguages," in R. Rustin (ed.) Database Systems, Courant Computer Science Symposium, Prentice Hall, Englewood Cliffs, N.J., 1972.

[DAYA83a]
Dayal, U., "Processing Queries With Quantifiers: A Horticultural Approach," Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983.

[DAYA83b]
Dayal, U., "Query Optimization in the Distributed Database Manager (DDM)," Technical Report, Computer Corporation of America, Cambridge, Mass., in preparation.

[GRAY78]
Gray, J. N., "Notes on Database Operating Systems," Operating Systems: An Advanced Course, Spring-Verlag, 1978.

[HS80]
Hammer, M., and D. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," ACM Transactions on Database Systems, Vol. 5, No. 4, December 1980.

[JS82]
Jarke, M., and J. W. Schmidt, "Query Processing Strategies in the Pascal/R Relational Database Management System," ACM SIGMOD Conference Proceedings, 1982.

[LP76]
Lacroix, M., and A. Pirotte, "Generalized Joins," ACM SIGMOD Record, Vol. 8, No. 3, September 1976.

[PALE72]
Palermo, F. P., "A Database Search Problem," IBM Research Report RJ 1072, July 1972.

[RBFG80]
Rothnie, J. B., Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong, "Introduction to a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, Vol. 5, No. 1, March 1980.

[RCDF83]
Ries, D., A. Chan, U. Dayal, S. Fox, and W. K. Lin, "Decompilation, Optimization, and Pipelining for Adaplex: A Procedural Database Language," Technical Report, Computer Corporation of America, in preparation.

[RS79]
Rowe, L. A., and K. A. Schoons, "Data Abstraction, Views and Updates in RIGEL," ACM SIGMOD Conference Proceedings, May 1979.

[RS81]
Rowe, L. A., and M. Stonebraker, "Architecture for Future Database Systems," ACM SIGMOD Record, Vol. 11, No. 1, January 1981.

[SCHM77]
Schmidt, J. W., "Some High Level Language Constructs for Data of Type Relation," ACM Transactions on Database Systems, Vol. 2, No. 3, September 1977.

[SELI80]
Selinger, P. G., and M. Adiba, "Access Path Selection in Distributed Database Management Systems," Proceedings of the International Conference on Databases, University of Aberdeen, Aberdeen, Scotland, 1980.

[SHIP81]
Shipman, D., "The Functional Data Model and the Data Language Daplex," ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

[SS77]
Smith, J. M., and D. C. P. Smith, "Database Abstractions: Aggregation and Generalization," ACM Transactions on Database Systems, Vol. 2, No. 2, June 1977.

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

[WDHL82]
Williams, R., D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost, "R*: An Overview of the Architecture," Proceedings of the 2nd International Conferences on Databases: Improving Usability and Responsiveness, 1982.

[WY76]
Wong, E. and K. Youssefi, "Decomposition -- A Strategy for Query Processing," ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976.