RJ2571(33471)7/14/79
Computer Science

Research Report

NOTES ON DISTRIBUTED DATABASES

B. G. Lindsay
P. G. Selinger
C. Galtieri
J. N. Gray
R. A. Lorie
T. G. Price
F. Putzolu
I. L. Traiger
B. W. Wade

IBM Research Laboratory
San Jose, California 95193

IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

RJ2571(33471)7/14/79
Computer Science

NOTES ON DISTRIBUTED DATABASES

B. G. Lindsay
P. G. Selinger
C. Galtieri
J. N. Gray
R. A. Lorie
T. G. Price
F. Putzolu
I. L. Traiger
B. W. Wade

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: This paper is a set of notes on various subjects in single site and distributed database management systems. It is intended as our contribution to a set of notes for a course on distributed databases. Although the discussion is couched in relational terminology, the technology is relevant to other data models as well. Five subjects are covered in this paper. The first is replicated data, with an emphasis on summarizing several update strategies and discussing their impact upon data consistency and currency. The second subject covered is authorization and views. Access control to files, logical views, and application programs is described in both local and multi-site environments. The next subject covered is an overview of distributed transaction management. The fourth discussion details recovery management protocols for transaction abort during normal processing, site recovery to a transaction consistent state, and media recovery for damaged on-line data. The last subject covered is multi-site transaction initiation, migration, and termination. Two commit protocols are presented which achieve uniform and atomic transaction commit in the presence of site failures during a multi-site transaction.

Notes on Distributed Databases

by

Bruce G. Lindsay, Patricia G. Selinger,
Cesare A. Galtieri, James N. Gray,
Raymond A. Lorie, Thomas G. Price,
Franco Putzolu, Irving L. Traiger,
and Bradford W. Wade

CONTENTS

Chapter I: Replicated Data 1
1.1: Introduction. 1
1.2: Why replicate? 1
1.3: Update Strategies 1
1.3.1: The Unanimous Agreement Update Strategy 2
1.3.2: Single Primary Update Strategy 2
1.3.3: Moving Primary Update Strategy 2
1.3.4: Majority Vote Update Strategy. 3
1.3.5: Majority Read Update Strategy 4
1.4: Data Consistency 4
1.4.1: Currency 5
1.5: Summary. 5
Chapter II: Authorization and Views 7
2.1: Introduction. 7
2.2: The Local Access Control Mechanism. 8
2.2.1: The GRANT Command 9
2.2.2: Revocation 10
2.2.3: Recursive Revocation 11
2.2.4: Views. 14
2.2.5: Authorization on a View 14
2.2.6: Granting Views 14
2.2.7: Revoking and Dropping Views. 15
2.2.8: Other Authorization Issues 16
2.2.9: Summary of the Single Site Authorization Mechanism. 16
2.3: Authorization of Application Programs 17
2.3.1: Application Program Authorization: A User's Viewpoint 17
2.3.2: The RUN Privilege. 18
2.4: Handling Multiple Authorization Environments 18
2.5: Further Issues 19
2.6: Conclusion. 20
Chapter III: Introduction to Distributed Transaction Management 21
3.1: Transaction Management Issues 22
3.2: Transaction Initiation and Termination. 22
3.3: Concurrency Control 24
3.4: Recovery Management. 25
Chapter IV: Recovery Facilities 28
4.1: A Log File Protocol 30
4.2: Page Fetch. 31
4.3: Page Update 31
4.3: Page Write 32
4.4: Transaction Commit. 32
4.5: Transaction Abort 33
4.6: Checkpoint 33
4.7: Restart 34
4.7.1: Restart Analysis. 35
4.7.2: Restart Undo 36
4.7.3: Restart Redo 37
4.8: Media Recovery. 37
4.9: Recovery Conclusions. 39
Chapter V: Transaction Initiation, Migration, and Termination. 40

5.1: Transaction Initiation. 40
5.2: Transaction Migration 41
5.3: Transaction Commit. 43
5.3.1: The Two-Phase Commit Paradigm 44
5.3.2: Recoverable Transaction States. 44
5.3.3: The Linear Two-Phase Commit Protocol 45
5.3.4: The Centralized Two-Phase Commit Protocol 46
5.3.5: Comparison of Linear and Centralized Commit Protocols. 50
5.4: Transaction Management Conclusions 50
REFERENCES. 55

Chapter I: Replicated Data

1.1: Introduction

Among the desirable properties of distributed database systems is the ability to have a local repository of frequently used data, while still being able to access data stored at other network sites. Our picture of a distributed database, then, is one where there are multiple sites each of which stores data. These sites communicate over a slow, unreliable communication network. Such a network can lose messages, duplicate messages, and deliver messages out of order. We will assume for this discussion that duplicate and out of order messages can be handled by the data communication component of each DBMS, but that lost messages must be reported to the component which sent the message. Generally, lost messages will be detected by lack of a response before timeout.

Because communication lines are slow and distributed data protocol overheads are significant, frequently accessed data should be stored locally. If two or more sites frequently access the same data, then there is reason to replicate that data, subject to certain tradeoffs. We will speak of data as replicated at the file level. If only certain portions of a file are accessed frequently, then one can conceive of replicating only that sub-file. This is a special case of partitioned data, which is not covered in this chapter. Replication of only some partitions of a file can be handled with the mechanisms we will present by treating each separate partition as a file for replication purposes.

1.2: Why replicate?

The arguments for replicated data are well-known, and we will only summarize them here. If a file is stored at N sites, then the availability of the data for a read request is higher than if the file were stored only once. If p is the probability that a site is up, then the probability that a read request can be satisfied is p if the data is not replicated. If the data is replicated at N sites, the probability that a read request can be satisfied is $1-(1-p)^N$, since reads can be satisfied by any one of the replicas. Replicated data enhances locality of reference by satisfying more read requests locally. This reduces the response time for a request, and reduces traffic on the communication network.

Another advantage of replication is that it offers the opportunity to distribute the workload on a frequently accessed file. Furthermore, replicated data provides greater reliability through backup copies from which data can be recovered in the event of media failures at up to N-1 of

the N sites containing replicas.

1.3: Update Strategies

1.3.1: The Unanimous Agreement Update Strategy

So far, only read requests have been discussed. It is with write requests that having replicas becomes expensive. Most proposals about replicated data have as their goal the presentation to the end user of a single copy image of data. One of the easiest designs to reflect a single copy image of replicated files is to propagate updates to all replicas immediately. Unanimous acceptance of the proposed update by all sites having replicas is necessary in order to make a modification, and all of those sites must be available for this to happen. Updates are refused unless they have unanimous acceptance.

Consequently, in this design, the availability of a replicated file for update requests is p**N, if there are N copies. For p<1 and large N, this implies that the overall system will seldom be available for updates. This means that a replicated file is *less* available for update than a single copy file. The system must support additional message traffic in order to send the update to all copies and confirm or cancel it, based on whether or not unanimous agreement was obtained. Furthermore, all these sites become participants in the updating transaction, and therefore must be included in the end-of-transaction processing (see the chapter on two phase commit). One can conclude that for this design to be cost-effective, a replicated file must have a high ratio of read requests to update requests.

1.3.2: Single Primary Update Strategy

As described above, under the unanimous update agreement mechanism, the probability that an update succeeds is p^N, which goes to zero as the number of copies N becomes large. Many solutions to the availability problem have been suggested which do not drive the success of an update to zero for large N.

We will list several of these solutions, and describe each one briefly. The first such solution is to designate one replica as PRIMARY, and the remaining replicas as SECONDARYs. Updates requests are issued to the primary replica, which serves to serialize updates and thereby preserve data consistency. Under this scheme, the secondaries diverge temporarily from the primary. After having performed the update, the primary will broadcast it to all the secondaries at some later time. There are different proposals for this broadcast, accompanied by limitations on the data currency and consistency seen by transactions run before all secondaries have current copies again. Some proposals call for the update requests to be sent to the secondaries 'immediately'; others package updates and broadcast them at end of transaction. Still others broadcast updates only at specified intervals -- once an hour, overnight, etc.

In all of these primary-secondary schemes, the delay of update propagation from primary to secondary can be perceptible to a user unless the DBMS is careful. A user who issues an update followed by a read wants to see the updated value. If the update is at a remote primary, and the read is at the local (secondary) replica, then there is a possibility/probability that the update will not have been propagated to the local replica yet. In a later section, this race condition will be further discussed.

If p is the probability that a node fails, then the availability of the system for update is p, the probability that the primary is up and communicating. In the next section, a variation is

presented which improves on this.

1.3.3: Moving Primary Update Strategy

This discussion is a summary of the algorithm proposed by Alsberg et al [1]. It is a variation of the algorithm in section 1.3.2 and provides greater resiliency to failures. The two-host resilient scheme is described, and can be generalized to n-host resiliency. There is a designated primary site and N-1 secondary sites. Update requests are made to the primary or any secondary -- in general updaters are unaware of which site is functioning as the primary at any particular time. When an update request is received, the receiving site is either a primary or not. If it is the primary, it performs the update, then it sends a cooperation request to a secondary, informing it of the update. The secondary performs the update, and broadcasts an acknowledge to the primary, and also to the original update requestor. It also passes the request on to the next secondary. Once the primary host has received the ack from the secondary, it is certain that two-host resiliency has been attained. In other words, the update is not lost unless the primary and the secondary which received the cooperation request both fail. If the original update request was received by a secondary server, it forwards the request to the primary, and the algorithm proceeds as above.

If the primary fails, it will be discovered by the secondaries when they forward their next request. Upon discovery, they begin to recoverably elect a new primary from amongst themselves. In a two host resilient scheme, all N-1 secondaries must participate in this election. In general, for a m-host resilient scheme, at least N-m+1 secondaries must participate in order to elect a new primary. An easy form of election is to choose the next primary to be the secondary with the highest site number in the participating set. Once all sites have been informed of the identity of the new primary and have acknowledged, then the new primary has been recoverably elected with m-host resilience and can begin functioning in its new role, accepting update requests. When the old primary next attempts to request cooperation in an update, at least one of the secondaries to which it makes its request will have been one of the electors of the new primary. This elector will inform the old primary that it has been deposed, and its request for cooperation will be treated as the forwarding of an update request to the (new) primary. Henceforth the old primary will behave like a secondary. The race condition mentioned in the previous section also occurs for this scheme.

1.3.4: Majority Vote Update Strategy

Another proposal to accomplish updates of replicated data with greater availability than the unanimous acceptance algorithm involves designating all sites with a replica as 'peers'. In this scheme an update succeeds if it is accepted by a majority of peers. A version number is associated with a particular majority which agreed on a set of updates. Each time a site or link fails, it causes a new (different) majority to be formed. When the majority changes, the version number is increased by the 'next' majority. The version number is monotonically increasing and unique to a given majority since every two majorities have at least one member in common. This member in common (namely, the member with the highest version number) will insist that other members of his majority group (the newly formed one) 'catch up' to his latest version of the data before any updates are accepted. Thus, successive updates may be accepted by different majorities, as nodes and links fail or recover, but only one majority at a time will be processing updates, and these updates will always be applied to the latest version of the data.

When a site has an update request, it must broadcast update request messages to all members of the last majority in which it participated. Either the update request will succeed (that majority is still in power and has no other pending updates on the same item), or it will fail due to a conflicting update, or fail because the ruling majority has changed. If an update request fails because the majority has changed, the site requesting the update may be the first to

detect that the former majority is no longer viable. If so, it attempts to form a new majority by a broadcast to all peers, followed by a 'catch up' of all out of date peers who respond. If a majority of peers responds, the set of responding peers is instituted as the new majority whose version number is one larger than the maximum version number previously held by any of the new majority members.

On the other hand, a site requesting an update may contact the members of the ruling majority associated with his version number, and receive a response that there is a new ruling majority. In this case, it catches up to the responding site with the highest version number, and requests to join the ruling majority. Joining a ruling majority causes the ruling majority to change, and the version number to increase.

Under the rule that a node may participate in an update if and only if it has the 'latest' version of the replicated data, it is ensured that there are no lost updates. Thus a monotonic, increasing version number and a majority list of peers is associated with each replicated table. With this scheme, the probability of a successful update is

$$\sum_{K>\lfloor \frac{N}{2} \rfloor} p^K (1-p)^{N-K} \binom{N}{K}$$

which approaches 1 for large N.

1.3.5: Majority Read Update Strategy

The proposals considered so far have concentrated on forcing the work of update to be done at update time, with associated complexity in the update algorithm. This next scheme places the work at the time of a read request. The scheme performs update requests on a replicated file at the local site of the request. The update is then broadcast to whatever sites are on-line. The update request succeeds if it is accepted by a majority of peers. A number is assigned to each update, and is monotonically increasing at each update. Each peer which accepts the update also stores the version number of that update. Updates are accepted at a given peer only in order. If update i is received then a peer may accept it only if it has accepted update i-1. A catchup mechanism is required to allow a peer to obtain updates it has missed.

This means that the overall system is available to update requests so long as there is a majority of peers which are in communication. Recall that in all the previous algorithms, read requests were always local (modulo some race condition problems). In this algorithm, read requests must be sent to a majority of peers. Each peer responds to the read request with the answer and the version number of the last update it has seen. The read requestor takes the answer with the highest version number.

This approach is feasible only when messages are fast and cheap, and when it is not a tremendous burden for a majority of peers to answer read requests when only one answer will eventually be used. It is not clear what distributed database management systems satisfy these characteristics; perhaps they are satisfied most closely by local networks of high speed CPUs.

1.4: Data Consistency

Transactions in a distributed environment can achieve the appearance that all data is stored as a single copy at a single site. For data which is not replicated, it has been shown that a distributed database management system need only obey the rules for non-distributed database management [19]:

1. Lock entities before using.
2. Hold all locks until end of transaction.

For replicated data, two additional principles are required in order to achieve single site, single copy equivalency:

3. Updates must be broadcast to all replicas before the transaction ends.
4. If updates are not immediately broadcast and performed by all replicas, then all accesses after an update must be to an updated copy.

It has been shown that these four conditions are sufficient to guarantee the equivalent of a single copy, single site, serial user system [39].

1.4.1: Currency

In the above update strategies, some peers may not receive updates at the end of the updating transaction. Thus rule 3 above was relaxed. This section investigates the effect that this has on data consistency. Having to raise the issue of currency at all is another price to be paid in order to achieve greater availability for update requests. If updates were performed immediately and only by unanimous agreement of all peers, then values would always be current at every peer. However, since rule 3 is relaxed by all of the non-unanimous algorithms above, there is a distinction between a 'currency' read request and an ordinary read request.

If a user wishes to read *the* most current value, then in the update strategies which have a primary site, that read must be issued to the primary. In the majority of peers strategies, the user must issue requests to all peer sites, and the read will succeed if and only if at least a majority of peers answer, at which point the answer with the highest version number is taken as the *current* value. It is clear that since all of the algorithms except the last permit local reads, that a read request for a current value is more expensive than an ordinary read request.

How can we characterize the value obtained by an 'ordinary' read request? The value was the current value as of the time of the last update request which was performed on the replica satisfying the read request. If the site is the primary, or in the current 'ruling majority', then this is the same value as would be retrieved by a 'currency' read request.

On the other hand, sites which have a replica, but are secondaries or have not participated in the latest update majority may continue to read and use the older data, so long as they do not also access newer data or any data 'derived' from newer data. Without this restriction transactions may see time appearing to flow backwards, e.g. by reading Tuesday's data followed by Monday's in the same transaction. This would occur if a transaction performed a read at a 'new' site followed by a read at an 'old' site. This would also occur in the race conditions mentioned above where an update followed by a read of the same item occurs at new (update reflected) and old (update not yet done) sites respectively. The data seen in these situations is inconsistent and does not provide a single copy view of data to the distributed database user.

Several alternative proposals have been made to enforce the restriction that new and old sites cannot participate in the same transaction. These schemes involve the exchange and comparison of version number and majority list for each replicated file whenever a transaction makes a request to another site. In this way, transactions spread 'gossip' about the relative newness of the data they have accessed so far, and compare it to the version of the data at the new site. Based on their current gossip, transactions refuse to run at sites known to contain older data. They may, however, encounter sites with newer gossip which tells the transaction that a site it previously visited really contained 'old' data which is not consistent with other

data the transaction has seen. In this case, the transaction must abort.

1.5: Summary

A number of proposals to read and update replicated files have been described. The update proposals trade off availability of the system for update against complexity of the update algorithm and the amount of bookkeeping needed to preserve data consistency. None of the proposals are simple, but the primary-secondary approaches appear slightly less complex and correspondingly more appealing.

Chapter II: Authorization and Views

2.1: Introduction

A multi-user data base system whether or not it is distributed, must permit users to selectively share data, while still retaining the ability to restrict data access. There must be a mechanism to provide protection and security, permitting information to be accessed only by properly authorized users. Further, when tables or restricted views of tables are created and destroyed dynamically, the granting, checking, and revocation of authorization to use them must also be dynamic.

A user of any database management system, whether distributed or not, interacts with the data base in a series of transactions or success units. Each transaction consists of one or more commands to the user interface of the system, such as an INSERT statement or a query. A transaction is the basic unit of integrity, consistency, and recovery.

The scenario of this chapter is that a user is logged on to his local DBMS, having undergone appropriate authentication. He is known to this DBMS by some USERID. This DBMS is in conversation with its sibling DBMSs at other sites of the distributed database management system. This DBMS-to-DBMS conversation is initiated at system restart and continues to the next crash or restart. Other terms used for this conversation are 'virtual circuit' or 'session'. One could also think of this conversation as being established on an as-needed basis, and ended upon termination of the transaction which uses it. Alternatively, one can conceive of this conversation being established upon first request to a remote DBMS, and then continuing until the next crash or restart. The tradeoff among these three choices is dependent upon the number of sites in the network, the expense of setting up and maintaining a conversation, and the frequency of remote access.

The initiation of this conversation includes each DBMS exchanging information and agreeing on protocols, formats, flow control, cryptographic techniques, and most importantly, authentication. By a public key cryptosystem or some other secure mechanism, each of the two DBMSs authenticate to their own satisfaction that they are indeed conversing with each other [18]. Once this conversation is established, then for the purposes of our discussion, it will be presumed that the systems subsequently trust each other and are communicating over a secure channel.

When a user issues a request to his local DBMS that involves a remote DBMS request, the local DBMS sets up a conversation with the remote DBMS if one does not already exist, and passes through the request along with the user's USERID. To the receiving remote DBMS, this will be treatd just as if that USERID were directly logged on to the remote DBMS. With such a mechanism, each site then performs local authorization checking on its own local data. A site uses conversations to pass through requests to remote data, which are then checked by the remote site using the same mechanism that performs local access control.

Each site, then, keeps access control information about its local data, and checks authorization upon every access. Authorization is by USERID, which must be unique across all systems participating in the distributed database. One easy way of achieving this uniqueness without a central USERID distributor is to prefix each USERID with the name (unique network address) of the site which originally issued the USERID. A consequence of this naming scheme is that a USERID is equally authorized regardless of the site from which he issues a request. User authorization status is transparent to the site from which the user is operating. S/he can issue any request for which s/he is authorized from any site in the network.

User location transparency is either an advantage or a disadvantage, depending on the degree of security needed. Some database systems today authorize on a USERID, TERMINAL, SITE basis; this would imply, for example, that bank tellers can only debit other people's bank accounts when they are at their teller stations. A DBMS can achieve this level of security on top of the mechanism presented here by obtaining USERID from the user's logon id concatenated with some system-produced information such as terminal number, site-id, etc.

In the following sections, we describe how the local access control mechanism should work, then describe how application programs are authorized, present a way to avoid access checking on every request, and finally discuss the conflict between dynamic object deletion and node autonomy.

2.2: The Local Access Control Mechanism

In current single-site data base management systems, the ability to grant authorization to perform actions on objects resides with a central 'data base administrator,' or with the creator of the object. Many systems rely on password schemes, which are vulnerable to guessing. Password schemes also require disclosing the password when a grant is made, thereby making authorization completely external to the DBMS; thus revocation and auditing become difficult. In addition, password schemes do not permit data-dependent access control. This chapter addresses the problems of dynamically authorizing data-independent and data-dependent operations, and of revoking such authorization, in an environment in which more than one user may grant privileges on the same object.

This discussion will use relational terminology, and it will be assumed that the reader is familiar with the relational data base approach as described by Codd [10, 11, 12, 13, 14], Date [15], and Boyce and Chamberlin [7, 8]. Although we will discuss the issues of authorization and their solutions in a relational context, we believe that many of our solutions are applicable to problems common to any multi-user data base system in which authorization is granted, checked, and revoked dynamically. The work described here is a summary and extension of the ideas presented in [9] and [24].

The basic objects in the data base are relations, which are sets of n-tuples. Each n-tuple in a relation has n columns; every column is named. An example of a relation is

EMPLOYEE (NAME, SALARY, MANAGER, DEPARTMENT)

The EMPLOYEE relation has a tuple (row) for each employee, giving his name, salary,

manager, and department.

In our model, there are two types of relations: *base relations*, which are physically stored, and *views* A view [9]. is a virtual relation which is a dynamic window on the data base. In response to a query, the tuples of a view are materialized from the base relation(s) on which it is defined. An update to a view is performed (if possible) by updating the underlying base relation.

Any query whose result is a relation may be used to define a view. Therefore, a view may be

- a row and column subset of a relation. For example, a view can be defined consisting of the names and salaries of the employees who work in the toy department.

- a summary of the information in a relation. For example, a view can be defined consisting of the average salary of each department.

- a join [10] of the information in two or more relations. For example, if the DEPT relation contains the number of the floor on which the department is located, one may define a view of employee names together with the floors on which they work.

It is to be emphasized that views are dynamic windows, and not static copies. As the information stored in a base relation changes, the information visible through views defined on that relation changes with it.

The view mechanism is a means of granting access to row and column subsets, granting 'statistical' access, and granting access to other transformations of relations. In the first part of this section, we will describe our authorization mechanism as it applies to all objects in the system, regardless of whether they are views or relations. We will refer to both views and relations by the collective name 'tables.' Later, we will describe the extensions necessary to accommodate a dynamic view mechanism.

2.2.1: The GRANT Command

In our model, there is no central data base administrator, in the usual sense of the term. Any data base user may be authorized to create a new table. When s/he does, s/he is fully and solely authorized to perform actions upon it. (If the table is a view, her/his authorization may be restricted by the authorization s/he possesses on the underlying tables.) If s/he wishes to share her/his table with other users, s/he may use a GRANT command to give various privileges on that table to individual users. Among the privileges that may be granted on a table are:

- READ: the privilege to use this relation in a query. This privilege permits a user to read tuples from the relation, to define views based on the relation, etc.
- INSERT: the privilege to insert new rows (tuples) into the table
- DELETE: the privilege to delete rows from the table
- UPDATE: the privilege to modify existing data in the table. This privilege may optionally be restricted to a subset of the columns of the table.

The user may grant a set of privileges with the GRANT option. The GRANT option permits the grantee to further grant his acquired rights to other users. This is analogous to the copy flag of Lampson [26] and of Graham and Denning [21], or sealed capabilities as in [31] or [40]. For example, let A be the creator of the EMPLOYEE relation, and assume that he issues the command

GRANT READ, INSERT ON EMPLOYEE TO B.

After the grant, B possesses the read and insert privileges on the EMPLOYEE relation. If B attempts to grant these privileges on EMPLOYEE to any user, the data base system will refuse the command, for B has not been given the GRANT option.

If B has been given the grant option in addition to the READ and INSERT privileges, then he may make such a grant. Assume that the sequence of grants is:

A: GRANT READ, INSERT ON EMPLOYEE TO B WITH GRANT OPTION
A: GRANT READ ON EMPLOYEE TO X WITH GRANT OPTION
B: GRANT READ, INSERT ON EMPLOYEE TO X.

Both grants by A succeed, since A is the creator of EMPLOYEE. The grant by B also succeeds, since he has been given the GRANT option.

What rights may the user X grant in the above example? Clearly he possesses READ and INSERT privileges on EMPLOYEE, but his INSERT privilege is not grantable. X's source of INSERT privilege is B, and B did not give X the right to further grant it. In fact, B may be completely unaware that X has been given the grant option. We therefore subdivide a grantee's privileges into two classes: grantable and not grantable. A user may not grant a privilege P that he has been given unless he was given P with the grant option.

When a user issues a GRANT command, the authorization module is consulted to determine whether the user is authorized to issue this particular GRANT command. The set of grantable privileges possessed by the grantor is intersected with the set of privileges named in the grant, and the rights which are actually granted are those in this intersection.

2.2.2: Revocation

Any user who has granted a privilege may subsequently withdraw it, by issuing the REVOKE command. The revoked privilege on the named table is denied to the revokee, unless the revokee has another (independent) source of the privilege.

REVOKEing previously granted privileges significantly complicates the authorization mechanism. The authorization system must retain records of each grant and the identity of the grantor, to make sure that a grantor is allowed to revoke only those privileges which he previously granted. Furthermore, upon revocation, the system must (recursively) revoke authorization from those to whom the grantee granted the table.

We will discuss the problem of revocation with respect to a given table. Let the sequence of grants of a specific privilege on a given table by any user before any REVOKE commands be represented by:

$$G_1, G_2, ..., G_{i-1}, G_i, G_{i+1}, ..., G_n$$

where each G is the grant of a single privilege. (We represent grants or revocations of several privileges as a sequence of individual grants or revocations.) If $i < j$, then grant G occurred at an earlier time than did G . Now suppose that grant G is revoked. The sequence becomes:

$$G_1, ..., G_{i-1}, G_i, G_{i+1}, ..., G_n, R_i.$$

We formally define the semantics of the revocation of G to be as if G had never occurred. This implies that the set of authorized privileges after the above sequence of grants should be identical to the set after the sequence

$$G_1, ..., G_{i-1}, G_{i+1}, ..., G_n.$$

The state is the same as if the grant of the revoked privilege had never been made*.
For example, consider the sequence

```
A:  GRANT READ, INSERT, UPDATE ON EMPLOYEE TO X
B:  GRANT READ, UPDATE ON EMPLOYEE TO X
A:  REVOKE INSERT, UPDATE ON EMPLOYEE FROM X
```

After the revoke, X retains READ and UPDATE privileges on EMPLOYEE.

The general rule is that if the revokee possesses other grants of the revoked privilege from an independent source, then he retains these privileges. We will discuss in the next section what we mean by 'independent source.'

2.2.3: Recursive Revocation

Consider the following sequence of grants (assume that A is the creator of the EMPLOYEE relation):

```
A: GRANT ALL RIGHTS ON EMPLOYEE TO X WITH GRANT OPTION
X: GRANT ALL RIGHTS ON EMPLOYEE TO Y
A: REVOKE ALL RIGHTS ON EMPLOYEE FROM X
```

According to the semantics of REVOKE, this sequence is equivalent to:

```
X: GRANT ALL RIGHTS ON EMPLOYEE TO Y
```

which fails, since X has no rights on EMPLOYEE. Therefore when a REVOKE command is issued, the system must not only modify the revokee's privileges, but it must also effect a revocation of the revokee's grants of these privileges. If the revoked privileges were not grantable, no further action need be taken.

The decision about exactly which privileges are to be revoked is not obvious. One might expect that if the revokee possesses other grants of the revoked right, then recursive revocation should not take place. The problem is that such an algorithm does not detect cycles in the chain of grants following the revokee. A difficulty arises if a revokee has in addition to the grant from

* This is why we defined a grant of privileges in excess of the available rights as the intersection, rather than rejecting the grant.

the revoker, another grant of the same privilege from another grantor. In this case, the revocation algorithm must distinguish between the case where the second grant is part of a cycle originating at the revokee (i.e. he has, possibly indirectly, granted the privilege to himself), versus the case where the grant stems from the table creator on a path which does not go through the revoker.

Effectively, the correct algorithm traces the grant chains from revokee back to the creator of the table. If every such path passes through the revoker, then revokee's privilege should be revoked. However, if there exists a path back to the creator which does not pass through the revoker, then revokee should retain the privilege after the REVOKE. Unless there are paths from the creator to the revokee not passing through the revoker, the revokee's privileges on the object are decreased or deleted. The revokee's remaining privileges on the table, if any, will be consulted to decide exactly which of his grants should be recursively revoked.

To decide whether to revoke recursively or not, without explicitly tracing the grant graph, the authorization records contain a timestamp indicating the relative time of a grant. Since a distributed DBMS checks authorization for remote requests at the site of the data, all authorization information is kept locally and checked locally. Thus the timestamp can be obtained from any local clock or counter; its important characteristics are that it is monotonically increasing, and that no two GRANT commands are tagged with the same timestamp. (Privileges granted in the same command are tagged with the same timestamp.)

For example, assume that A, B, and C grant certain privileges on the EMPLOYEE table to X, who in turn grants them to Y. After this sequence of events, the relevant section of the authorization records look like:

USERID	TABLE	GRANTOR	READ	INSERT	DELETE
X	EMPLOYEE	A	15	15	0
X	EMPLOYEE	B	20	0	20
Y	EMPLOYEE	X	25	25	25
X	EMPLOYEE	C	30	0	30

Suppose that at time t=35, B issues the command REVOKE ALL RIGHTS ON EMPLOYEE FROM X. Clearly, the (X, EMPLOYEE, B) grant must be revoked. In order to determine which of X's grants of EMPLOYEE must be revoked, we form a list of X's remaining incoming grants:

USERID	TABLE	READ	INSERT	DELETE
X	EMPLOYEE	(15,30)	(15)	(30)

as well as a list of X's grants to others:

USERID	TABLE	READ	INSERT	DELETE
Y	EMPLOYEE	(25)	(25)	(25)

The grant of the DELETE privilege by X at time t=25 must be revoked, since his earliest remaining DELETE privilege was received at time t=30. But X's grants of READ and IN-

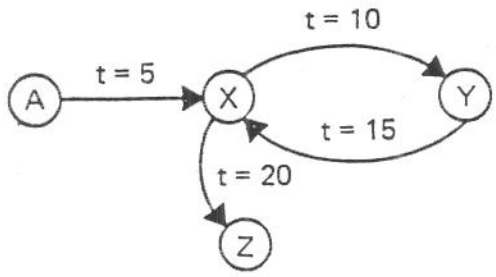
SERT are allowed to remain because they are still 'supported' by incoming grants which occurred earlier in time.

The revocation algorithm is as follows:

```
REVOKE:procedure ( grantee, privilege, table, grantor );
    comment  turn off the grantee's authorization
                    for <privilege> obtained from <grantor>;
    set <privilege> = 0 in the (grantee, table, grantor)
                    authorization record;
    comment  find the minimum time stamp for the grantee's
                    remaining grantable <privilege> on <table>;
    m <-- current timestamp;
    for each grantor u such that (grantee, privilege,
                    table, u, grantable) is recorded do
        if privilege ≠ 0 and privilege < m
                    then m <-- privilege;
    comment revoke grantee's grants of <privilege> on
                    <table> which were made before time m;
    for each user u such that (u, privilege, table, grantee)
                    is recorded do
        if privilege < m
                    then REVOKE ( u, privilege, table, grantee );
    return
    end REVOKE
```

Example:

Consider the following graph of grants, assuming that all privileges are granted each time:



and let the grant from A to X be revoked. Then,

- On the first pass, only the X-to-Y grant is revoked. The X-to-Z grant is *not* revoked, because of the incoming grant at t=15.
- On the second pass, the Y-to-X grant is revoked.

• Finally, on the third pass, the X-to-Z grant is revoked.

If the same privilege is granted by the same grantor to the same user on the same table, then both grants must be recorded. Otherwise, a REVOKE at a later time might cause the revocation of some earlier legitimate grants [20].

2.2.4: Views

Up to now, we have only been concerned with the granting and revoking of privileges on existing tables. However, the DBMS should provide the ability to define views [7, 9] on top of base relations, and on top of other views. The view mechanism is a method which permits data value-dependent access to a table. In order to grant privileges on selected row and column subsets, or to grant 'statistical' access, the user defines a view of the table, and then grants access to that view. The grantee may then query that view, even though he does not necessarily have any access at all to the underlying table (and therefore cannot even issue the query that defines the view). He may also issue INSERT, DELETE, and UPDATE commands against that view, subject to his granted privileges on that view, and to the semantics of data modification through views.

Other relational systems permit a similar type of data-dependent access control. INGRES [35] performs query modification to achieve this. Whenever a user makes a query on a table, the predicate representing his data-dependent authorization on that table is ANDed to the query. The MacAIMS system [30] uses filters to implement data-dependent authorization.

2.2.5: Authorization on a View

When a user creates a base relation, he is fully and solely authorized to perform actions on that relation. Similarly, when a user defines a view, he is solely authorized to perform actions upon it. However, he is not *fully* authorized, for two distinct reasons:

(1) view semantics: certain operations may not be performed on any view; for instance, the creation of an index. Other operations may or may not be allowed, depending upon the view itself. For instance, 'statistical' views are not updateable.

(2) the definer's authorization on the underlying tables: clearly a user with read-only access to the EMPLOYEE table should have read-only access to any view which he defines on top of it. We restrict a user's authorization on a view to be the set of privileges held by the user on the underlying table(s). If the view involves more than one underlying table, the user's privileges are constrained to the intersection of the privileges which he holds on the underlying tables. Furthermore, a privilege P on the defined view is grantable if and only if the definer holds a grantable privilege P on all underlying tables. Recall that the update privilege may be restricted to a subset of the columns of a table; we may then consider the update privilege to be held column by column. Therefore, the update privilege is held on a column of a view only if it is held on that column in all of the underlying tables in which it occurs.

Therefore, at view-definition time, its authorization record indicates the definer's privileges on the view, as computed from considerations (1) and (2). The timestamp associated with each

privilege is the time of view definition.

2.2.6: Granting Views

Views may be granted to other users, and the set of grantable privileges is computed in the same manner as for base relations: namely, it is the union of all grantable privileges from all grantors.

2.2.7: Revoking and Dropping Views

While the granting of views proceeds very similarly to the granting of base relations, the presence of a view mechanism makes revocation more difficult. A view may be built on top of granted tables; then this view may be granted, and used to construct other views. A view may be dropped (its definition deleted) at any time. As a result, views built on top of the dropped view must also be automatically dropped. Therefore, at REVOKE time we need not only to recursively revoke grants of a privilege or an object, but also to drop or re-authorize views which have been built using that object. (A view is re-authorized by computing its remaining set of privileges.)

To introduce the view mechanism into our formal model of the semantics of revocation, consider a sequence of grants and view definitions, followed by a revocation or a drop. The semantics of this sequence is defined to be the same as if the corresponding grant or view definition had never been made. A revocation or a drop affects both grants and view definitions. All grants of dropped views are automatically revoked. All view definitions based on dropped views, or on tables for which the definer has no remaining privileges, are deleted.

To illustrate the implementation of REVOKE in a system with a view mechanism, and the implementation of DROP in the presence of a grant mechanism, we will present our REVOKE and DROP algorithms as they apply to a restricted version of our authorization mechanism. Then we will indicate how to extend the algorithms to apply to the general case. Consider a system which is restricted in the following manner: Individual privileges cannot be granted; on any given table, each user has either all privileges (and can grant them) or no privileges. A user may possess only one grant of any given table.

In such a restricted system, the set of grants and view definitions ultimately based on any given table may be described by a tree. The nodes of the tree are (user, table) pairs, representing the fact that the user has all privileges on the table; the arcs of the tree represent grants and view definitions. A grant of table-1 from user-1 to user-2 is represented by an arc connecting a (user-1, table-1) node to a (user-2, table-1) node. A view definition by user-1 of table-1 using table-2,..., table-n is represented by connections to a (user-1, table-1) node from each of the nodes (user-1, table-2),..., (user-1, table-n).

The commands REVOKE and DROP cut a grant arc or a set of view definition arcs, respectively. Each also prunes the entire subtree originating at the removed edge(s). To implement REVOKE and DROP, we need two recursive procedures with quite similar structures. Each procedure performs the atomic act of cutting an arc, and then calls itself and the other recursively in order to prune the subtree. Each procedure relies on the system records of which views are defined using which underlying tables. A record (UNDERLYINGTABLE, VIEW, DEFINER) records the fact that the user DEFINER used the UNDERLYINGTABLE in defining the VIEW.

REVOKE:*procedure* (grantee, table, grantor);
delete the record for the (grantee, table, grantor) grant;

```
        for each u such that (u,table,grantee) was granted do
                REVOKE ( u, table, grantee );
        for each view such that (table,view,grantee) was defined do
                DROP ( view );
        return;
        end REVOKE;


DROP: procedure ( view );
        delete the view definition from the system;
        for each u1 and u2 such that (u1, view, u2) was granted do
                REVOKE ( u1, view, u2 );
        for each v and u such that (view,v,u) was defined do
                DROP ( v );
        return;
        end DROP;
```

To extend this simplified model to encompass the full power of our authorization mechanism, we proceed as follows: to properly process revocations from users who have received several grants of the same table, and to detect circular grants, we issue timestamps as before. After a DROP or a REVOKE, those grants are revoked which grant a dropped view, or which are not supported by any remaining earlier incoming grants; those views are dropped which are built using a dropped view, or which are built on a granted table which was not received prior to the time of the view definition.

Selective granting and revocation of individual privileges is handled as before; affected views are re-authorized, and dropped only if their definer no longer holds any privileges on the underlying table.

2.2.8: Other Authorization Issues

The reader may have observed that a user who has been granted a view has access to the tuples of that view, even though he may not be able to execute the query which defines the view (because he may not be able to access the underlying tables). Associated with each view is a 'recipe' for materializing tuples of the view, in terms of operations on the base relations. The recipe is generated automatically by the system from the definition of the view. The granting of authorization on a view is equivalent to authorizing the use of the associated recipe. Therefore, once it has been established that a user is authorized to use a view, the recipe is made available, and no further authorization need be performed.

When a user defines a view, a check is made to determine whether he has the necessary authorization on the underlying tables. If he does, then the view definition is accepted, and a 'recipe' is created for it.

2.2.9: Summary of the Single Site Authorization Mechanism

We have presented a mechanism which permits the users of a single-site shared data base to maintain private data, and permits them to share a set of privileges on their data with a selected group of other users, or with all users. Subsets of a user's data, derived data, and other transformations of data may be shared by defining a view and sharing that view. Privileges on an object, once granted, may be revoked. We have defined the semantics of revocation within a shared data base, and presented a recursive algorithm which effects these semantics. Upon revocation of a privilege from a user, the algorithm revokes his grants of that privilege which

were made before his oldest remaining receipt of the privilege. Consequently, privileges legitimately obtained from other sources, and his grants of those privileges, are retained; privileges obtained circularly via a collusion of users are revoked. Examples of the techniques described are presented within the context of a relational model of data. However, we feel that these techniques are applicable to any data base management system which performs authorization dynamically.

2.3: Authorization of Application Programs

In addition to being authorized to perform operations on tables, a USERID can be authorized to run application programs on the system. An application program contains statements in some programming language plus database requests. The execution of an application program makes zero or more requests to the database. The first request to the database is always a BEGIN TRANSACTION command; subsequent requests can either be data requests or an END TRANSACTION command, possibly followed by another BEGIN TRANSACTION. Recall from the discussion above that a transaction consists of one or more logically related DBMS requests, which form a unit of locking, recovery, and consistency of data. Thus an application program can contain zero or more DBMS transaction units. In a distributed DBMS, a transaction's requests (and thus those in an application program) can be on data stored at several different sites. In handling these requests, the local DBMS sends requests concerning remote data to the remote DBMS, which does authorization checking as described above, and then performs the requested operations on its data.

In an ad-hoc environment, the user sits at a terminal and types in DBMS requests, which may require remote data. Thus the locus of activity of the application program which is the terminal interface migrates among the sites, performing work authorized against the USERID which invoked this application program. One can also conceive of application programs as those which are installed in the DBMS and then run repeatedly, possibly by many users. Such programs would be written by knowledgeable database or system programmers, and invoked by end users who have little if any database knowledge. In this case, the application programmers should be authorized to perform the operations of the programs they write, but the end users should only be authorized to run those programs. End users should not be given broad powers of UPDATE, say, on the ACCOUNTS table. Instead, they should only be able to issue the carefully written DEBIT and CREDIT application programs which ensure the consistency of the ACCOUNTS data. The authorization mechanism presented above must handle this case differently than the case where it authorizes requests against the current logged-on USERID, which in this case is the end user. This mechanism must be extended to encompass the installation and authorization of a transaction by one highly authorized user, and the subsequent running of that transaction by weakly authorized users.

2.3.1: Application Program Authorization: A User's Viewpoint

An application program undergoes two stages: installation and invocation. These stages may be under the control of different users, with different authorization privileges. The dichotomy of installation versus invocation, under the authorization control of different types of users, described below, justifies a new object type 'application program' in the DBMS. This application program object has access controls which are separable from the privileges utilized within the application program.

End users are usually limited to pushing buttons which cause forms to appear on the display screen. After filling in a form, another button causes the form to be validated, and if it passes the test, to be acted upon by the system.

The application programmer defines and installs application programs. Depending on the degree of care exercised by the programmer, he may be able to prevent the end users from destroying the application-defined consistency and validity of the data base. For example, the program might refuse to handle withdrawals of more than five hundred dollars without the branch manager's approval.

Installing an application program consists of an application programmer entering it into the DBMS catalogs, storing the program in a safe place (usually this place is within the DBMS where only the system can access it), and authorizing one or more end users to use it. The installation component of the DBMS checks the installer's authorization against the set of DBMS requests in the program. Installation is successful if and only if the installer is authorized to perform each request. A successful installation results in the installer obtaining the RUN privilege on the application program.

2.3.2: The RUN Privilege

One reason for defining application programs is to encapsulate objects so that others may use them without violating the integrity of the constituent objects. In just the same way, we provided views in order to authorize (value-dependent) portions of data without fully authorizing use of the underlying table(s). The application program is a mechanism for authorizing certain specific consistency-preserving sequences of actions without authorizing exercise of the constituent actions themselves. Installed application programs become authorized objects in the database, and as we saw above, the creator or the installer of an object becomes fully and solely authorized to perform all actions on the object. In the case of an application program, the only privilege on application programs is RUN, the ability to invoke the program.

After installation, the installer may GRANT the RUN privilege on the program if and only if he was authorized to GRANT all the privileges required to perform the DBMS requests contained in the program. For example, if a program reads table T and updates view V, then the installer can grant RUN on that program only if he has a grantable read privilege on T and a grantable update privilege on V. As in the case of privileges on tables and views, the RUN privilege on programs may be GRANTed WITH GRANT OPTION, the ability to further grant the RUN privilege to other users.

For example, a banking system provides programs which credit and debit accounts (according to certain rules) rather than granting direct access to the accounts file. This effectively encapsulates the procedures of the bank and insures that all users of the data follow these procedures. An application programmer would write the programs and grant RUN authority to the tellers of the bank and grant RUN authority WITH GRANT OPTION to the branch managers so that they could authorize new tellers at their branches.

Revocation of the RUN privilege on an application program proceeds as for revocation of a privilege on a table, with the recursive revocation of further grants of the revoked privilege.

When an application program is installed, a record is kept of the tables it references, and the authorization of the installer on those tables. If any of these tables are dropped, or privileges on those tables are revoked from the installer of the program, then the program is invalidated. This invalidation also causes all the RUN privileges on that application program to

be revoked.

2.4: Handling Multiple Authorization Environments

We have described how authorization works for individual operations and how a highly authorized programmer can install a transaction and allow a weakly authorized end user to RUN it without granting to that end user all of the programmer's privileges. In the case of a terminal interface program which accepts ad hoc DBMS requests from the user who invoked it, the authorization to perform those requests must be checked against the USERID of the invoker.

Thus the authorization checking for an application program distinguishes between DBMS requests which appear explicitly in the program, and those which are accepted as input at execution time and are passed through by the program to the DBMS. Explicit requests contained in the program are written by the application programmer, are examined by the DBMS at installation time, and are authorized against the privileges held by the application programmer installer. Execution-time requests are submitted by the application program invoker, are examined by the DBMS at invocation time, and are authorized against the privileges held by the application program invoker.

2.5: Further Issues

It has been described that the installation of an application program causes its authorization to be checked, and then the program is stored in the database. For programs all of whose requests involve data at only one site, it is clear that the program should be stored at that site so that all accesses will be local. Thus all the access information about the application program as well as the access information about the tables the program references are all stored at this site. The revocation of a privilege on the application program changes only authorization records at that site. Similarly, revoking or dropping a table involves checking locally for the application programs and views which reference it, and modifying or revoking their privileges.

If an application program references tables at two or more sites, it is not clear which site should store the program or how revocation/invalidation should work. If the application program is stored at both sites, then there is a replicated file update problem (see chapter on replicated data). Both sites must be up in order for the revocation or invalidation to occur (in the absence of more complex update algorithms for replicated data.) This is a serious loss of autonomy for an individual site -- a local user cannot revoke privileges on or drop a local object unless the site is in communication with every site which has an application program which references that object.

If an application program is stored at only one site, even though it references objects at several sites, then the same problem still exists. The sites which do not store the program must still record the program's dependency on their local objects and still have to reflect drops and revokes on local objects to the site which has the application program. Regardless of where or how many times an application program is stored, local sites have to do remote accesses in order to perform essentially local actions. This same situation also occurs for view definition. If a user at site A defines a view on a table at site B, then either the view definition should be stored at A or at B, or at both. If it is stored only at B, then local drops and revokes can be reflected

in the privileges on the view by the local DBMS alone. If the view definition is stored at A, or if a view references a table at site C as well as a table at B, then revokes and drops at B must involve the participation of multiple sites.

An alternative to this loss of autonomy is to permit storage of view definitions and application programs at any or all of the sites with referenced objects. A revoke or drop of an object at a site will be performed locally, and will not affect the view definitions or application programs stored remotely. The next time a remote site attempts to access that object, the remote site will be refused access, and can then revoke or invalidate its references to that object. This scheme depends on being able to check every data request to an object at execution time, which conflicts with the efficiency improvement obtained by performing all authorization checks on explicit DBMS requests at installation time. This could make data requests considerably more expensive (e.g. a factor of 2 or 3).

2.6: Conclusion

Mechanisms have been presented for authorization of privileges on tables, views, and application programs in a single site DBMS. These mechanisms are also applicable in a distributed DBMS. The authorization records for an object are kept at the site where the object is stored, and privileges may be granted or revoked dynamically.

For reasons of efficiency, authorization is checked as early as possible. When an application program is installed the ability of the installer to access the objects the program references is checked. Similarly, if the program is a user interface program which at execution time takes in user commands, then those requests will be authorized in the context of the invoking user. If all DBMS requests in an application program are explicit, no authorization tests will be performed at invocation time (except for validation that earlier authorization decisions have not been revoked). A conflict exists between this early authorization checking for efficiency of execution and authorization checking at every access in order to maintain site autonomy. This conflict must be resolved in choosing where to store application programs or view definitions.

Chapter III: Introduction to Distributed Transaction Management

Distributed database management systems can be expected to operate in an environment consisting of many processing sites connected by a relatively slow, unreliable, and expensive communication network. Each site may be controlled by a different organization or individual. This means that distributed data accesses are achieved by *cooperation* among independent and *autonomous* database managers. Cooperation is achieved through the acceptance and use of common protocols which define the meaning of sequences of *messages* exchanged between sites.

In order to organize and control access to a shared database, the database management system must provide some mechanism for identifying and defining groups of logically related database **actions**. An action is a call to the database management system. The notion of a **transaction** has been introduced to represent a delimited sequence of database actions which together perform a logical unit of work [6],[19],[33]. The sequences of actions which constitute a transaction are identified with and defined by **application programs** which issue the database calls comprising the logical unit of work.

For example, an application program can define a transaction which transfers funds from one account to another and produces an audit trail for the funds transfer. Another application might define transactions which (interactively) accept order invoices, transactions which maintain inventory data, and transactions which produce shipping orders based upon order invoices and inventory levels. In each case, several database actions are required to complete the work of a single transaction as defined by the application program.

We must distinguish between the application program which defines the sequences of actions in a transaction from an *instance* of a transaction. A single application program invocation may initiate many instances of the transaction(s) it defines. An application creates transaction instances by executing call to the database manager actions for initiating and terminating transactions. The sequence and parameters of the database calls comprising the transaction can be determined by inputs to the application program as well as by values of the database items accessed by the application. For example, the operator may type an account number which is used by the application program to select particular account records. The values found in the account records may in turn be used by the application program to control whether or not a check can be cashed.

Normally, the application program which defines the sequence of actions comprising a transaction type must be installed in the database management system before it can be invoked to access the database. An application program may be invoked by the arrival of a message at a site, as a batch program, or from an interactive console. Once active, the application program may be able to converse with the operator and may complete zero or more transactions before terminating.

Actions on database objects are implemented by the database management system in such a way as to be **atomic**. An atomic action either completes and (possibly) modifies the state of the database or it does not complete and has no effect on the database state. If two (or more) atomic actions on a given data object complete, then the result will be equivalent to a serial

execution of the actions. Transactions extend action atomicity to the sequence of actions comprising the logical unit of work as defined by the application program. Either all the actions of the sequence are completed and have their effect on the database or the transaction has no effect.

Transactions remain atomic despite failures of the hardware, software, or the database storage. Either all the effects of transaction are applied to the database or the transaction has no effect upon the database. In order to preserve transaction atomicity, recovery management must be able to reapply the effects of committed transactions whenever a database site fails without having written all of a transaction's updates to non-volatile storage or when some portion of the database storage has failed. Also the recovery management facility must be able to remove the effects of partially completed transactions from the database whenever a site fails while some transactions are incomplete or whenever the system or the application program requests a transaction abort. The recovery facilities relieve the application programmer / designer from having to consider how to return the database to a transaction consistent state following a system failure. Also, partially completed transactions can be aborted by the by the application or by the system without having to write application specific programs to undo the effects of the partially complete transaction.

These chapters discuss the design of transaction management facilities which give the database user the ability to define logical units of work which are recoverable and atomic. Transaction atomicity eliminates interactions between separate transactions and insures that only complete sequences of actions are posted to the database. Recovery facilities insure database integrity and allow the system or the application program to abort the effects of incomplete transactions.

3.1: Transaction Management Issues

Transaction management is a component of the database management system which oversees and controls the execution of the database actions comprising a transaction. Transaction management responsibilities fall into three basic areas.

- Transaction initiation and termination: identifying the sequence of actions comprising a transaction and unambiguously committing or withdrawing all of the transaction's results.

- Concurrency control: synchronizing accesses to database entities in order to control interactions between different transactions.

- Recovery: insuring database integrity in the presence of hardware, software, communication, and storage media failures.

Transaction management interfaces to the other components of the data management system to control the progress of transactions in the system. Among the components which interact with transaction management are the data communication manager, the data model manager, the

lock manager, and the log manager.

3.2: Transaction Initiation and Termination

In order to be able to maintain isolation between concurrent transactions and to be able to guarantee that all the effects of committed transactions will be retained in the database, each database action must be associated with a particular transaction. All the actions of a transaction, including accesses at different sites, must be identified with the same transaction. A new transaction is created when an application program asks the database management system to start a transaction. Beginning a new transaction creates a recovery scope and defines a concurrency partition. Subsequent database actions on behalf of the transaction are recovered together.

The site at which the transaction is initiated is known as the **site of origin**. Once begun, the transaction can access database entities locally and at other sites of the distributed database. Remote accesses require the participation of database managers at the remote sites. When non-local data is required, the transaction **migrates** to the site where the required data is available. Transactions migrate by sending ***work*** request messages describing the database accesses required. When the remote accesses are complete, the results are return in an ***answer*** message. Section 5.2/ discusses the ***work*** and ***answer*** message protocol.

In order to minimize the number of work request and answer exchanges required to complete the work of a transaction, the semantic level of the work request message should be as high as possible. The amount of database activity required to satisfy a work request message should be large enough to amortize the processing overhead and transit delays involved in sending and receiving messages. Increasing the amount of work requested by a single message can be achieved by packaging (bundling) several small work requests together, or by expressing a larger unit of work in a higher level protocol. For example, one can send a sequence of requests for individual database records or one can send a single request for all records satisfying some search criteria.

Once a transaction has been initiated it must eventually complete. If all goes well, the transaction can **commit** its updates to the database and notify the user of completion. Committed updates are ' visible ' to other transactions and will be recovered if the system or the storage fails. If the transaction has updated database entities at more that one site, extra effort (messages) is required to insure that all the sites involved in the transaction commit together. Section 5.3 discusses the details of the protocols required to insure uniform commit at all sites of a transaction.

Instead of committing, a transaction can be **aborted** at user or system request. When a transaction aborts, all of the transaction's updates are withdrawn or undone. Transactions are aborted whenever:

- the system hardware or software fails before all the actions have occurred,
- the system elects to kill the transaction, or
- the application program decides to abandon the transaction (e.g., insufficient funds).

Transaction abort allows both the application program and the database manager to undo the effects of an incomplete transaction without having to supply application dependent

programs to drive the transaction in reverse. System initiated transaction abort also allows the database manager at any site to retrieve the local resources held by a transaction by unilaterally backing-out and aborting the transaction.

3.3: Concurrency Control

While this presentation does not include a complete discussion of the techniques and mechanisms for controlling concurrency among transactions, we will touch upon some of the issues as they relate to transaction management. Recalling our earlier definition, atomic actions either complete or have no side effects *and* the result of two (or more) atomic actions is the same as if they had executed serially, one after another. The same effects are desired for transactions.

If transactions run one at a time, then each transaction sees a database state which reflects only the effects of completed transactions. Each transaction produces a new database state which depends only upon the old database state, the transaction definition, and the input to the transaction. If all transactions were simple and the database was stored in primary memory at a single site, there would be no need to interleave the execution of different transactions. However, long transactions and delays caused by accesses to information in secondary storage or at remote sites require that several transactions execute concurrently in order to improve system response time and throughput and to improve system resource utilization.

Unrestricted interleaving of database actions from different transactions can lead to anomalies which threaten the integrity of the database and compromise transaction atomicity. The database integrity anomalies which occur because of uncontrolled transaction concurrency may give incorrect answers and leave the database in a state which it would never reach if transactions were executed one at a time at a single site. These database anomalies include:

- lost updates: If two transactions read and increment the value of the same database record, overlapping transaction execution may cause the record to be incremented only once. (e.g., A transaction which posts deposits to a savings account may leave an incorrect balance if two deposits to the same account are processed concurrently.)

- dirty reads: If transactions can read outputs from incomplete transactions, then they may 'see' database states which could not be observed in a system without concurrency. (e.g., A funds transfer transaction credits a sink account and then checks the balance of the source account. If the source has insufficient funds to cover the transfer, the sink is debited to restore its balance. Without concurrency controls, other transactions could read the intermediate sink balance.)

- inconsistent reads: If related data records are being read by one transaction while another transaction updates them, the first transaction may not see consistent record values. (e.g., Transaction T1 computes the mean and standard deviation of successive versions of a sample set. Without concurrency controls, another transaction might read the mean produced by one execution of T1 and the standard deviation from a subsequent execution of T1.)

Locking can be used to control the interleaving of database accesses from different transactions and to prevent the above anomalies. Database entities at various granularities can be locked by one transaction to prevent other transactions from reading or updating the locked entity [22]. It has been proved that if all transactions 1) lock each entity before accessing it

and 2) hold all locks until the end of transaction, then the resulting interleaving of database accesses is equivalent to some serial execution of the transactions [19]. In the distributed case, database locking at the site of the entity being accessed is sufficient to insure a globally serializable schedule of database accesses [39]. Additional information about locking to control transaction concurrency can be found in [23], [29], [33].

Unless concurrency controls are used to isolate transactions from each other, undoing one transaction may require undoing other transactions. The concurrency anomalies discussed above also occur when we consider transaction abort. The lost update anomaly occurs if transaction T1 increments a value which is in turn incremented by transaction T2. If T1 is subsequently aborted and the old value seen by T1 is restored, T2's update is lost. Dirty reads allow transactions to see updates which may be undone by a transaction abort. In general, if transactions are not protected against lost updates and dirty reads, aborting one transaction may invalidate the results of other completed transactions. To undo a committed transaction one must *compensate* by running another transaction which corrects the mistakes of the original transaction [16]. It is partly because of this 'domino effect' that most database management systems provide some form of concurrency control.

Waiting for locks on database entities can lead to local or global (inter-site) deadlocks among waiting transactions. Distributed deadlock detection and resolution requires extra mechanism in lock management. This is one of the reasons that there has been recent interest in concurrency control mechanisms for distributed database management systems which are based upon the use of timestamps on data and transactions. The timestamp schemes seem to allow less concurrency and imply longer delays (extra messages) than locking. Readers interested in finding out more about timestamp concurrency control algorithms are referred to [32], [4], [5], [38].

3.4: Recovery Management

In order to provide reliable service the database management system must be able to preserve the integrity of the database over long periods of time. In particular, the database management system should be able to restore the database to a transaction consistent state following hardware, software, communication, or storage media failures. If the transactions applied to the database transform the database from one consistent state to another, then the database will be transaction consistent when it reflects only the effects of *committed* transactions.

In order to preserve database integrity, recovery management facilities are designed to be able to bring the database to a transaction consistent state which includes all the effects of *all* committed transactions. Whenever a transaction has not been committed, recovery management must be able to remove (undo) its effects from the database. On the other hand, if a transaction has been committed, recovery management must insure that all of its effects are retained in the database despite the various failures which might befall the database management system. When the database is distributed, recovery management at the various sites must be carefully coordinated so as to uniformly retain or reject the effects of multi-site transactions.

In order to design effective recovery facilities, it is important to define clearly what kinds of failures are expected and their characteristics. Failures which affect the database management system fall into three broad categories:

- processor and software failures which cause the database management system at one site to be stopped and restarted,

- storage media failures which make it impossible to retrieve data from the on-line database storage, and

- failures in communication between sites of the distributed database management system.

Recovery procedures are designed to cope with one or another of these failure classes. In general, site restart and storage media recovery depend upon redundant descriptions of transaction activity and database updates. Recovery from communication failures depends, in general, upon being able to abort the transaction experiencing the failure.

Site restart failures are fairly frequent (on the order of one per week) and the mean recovery time varies between minutes and days. We assume that the volatile memory is lost during a site restart. Site restart failures are detected or made known to the database management system when the database is brought back up. In order to begin from a consistent state, when the database is brought up the recovery manager insures that only the effects of all committed transactions are incorporated into the database. Transactions which had not committed (and were not in the process of committing) when the system stopped are backed-out.

Storage failures are the second class of exceptions to which the recovery facilities must respond. Storage can be divided into three classes:

- *volatile storage:* main memory and paging space,

- *non-volatile on-line storage:* database and log disks, and

- *non-volatile off-line storage:* tape, archive and off-site storage.

We assume that the failure modes of the three types of storage are independent. For example, a power failure will affect volatile storage but not the other two storage classes. A disk head crash damages the non-volatile on-line data but does not affect off-line media. It is assumed that a fire will not affect both the on-line and off-line storage.

Any piece of storage may spontaneously fail. A particle of dust may settle upon a disk record or a black hole may pass through an archive store making it impossible to retrieve the stored information. We assume that one can detect when stored information has been lost. Error detecting codes can be written with the data to validate the data when it is read.

The database is stored in non-volatile on-line storage (e.g., the disks). Data is transferred non-volatile and volatile storage in blocks called pages. Pages are moved to main memory (volatile) where they may be modified. Subsequently they are written back to the disks. We assume that when a page is written, the target page either receives correct value, is detectably messed up, or retains its original value [27].

Updates by committed transactions must not be lost. If the updated pages have not been written to non-volatile storage when the volatile memory fails (e.g., system restart), the recovery facility must be able to reconstruct the updates belonging to completed transactions. If a page of the on-line non-volatile storage fails, the lost page must be reconstructed from off-line data plus on-line records. The mechanisms employed by the recovery facility to reconstruct lost data are discussed in detail in chapter III.

In distributed database management systems, database managers at different sites will communicate by exchanging messages. From the point of view of the sending data base manager, a message is intended to cause some action and / or solicit some results from the recipient. It is of little or no interest to the sender to know whether or not the message is delivered to the recipient. The sender wishes to know whether or not the message has been *acted upon* by the recipient and the sender wants to know the outcome of the action. For this reason, the low level delivery acknowledgments provided by most communication facilities are of little use to the database manager. To achieve reliable distributed function we will require that (almost) every message sent is acknowledged by a response *from the recipient.*

Given a message or response to transmit, the communication facility will either:

- deliver the correct message to the right recipient,
- deliver a garbled message to the right recipient,
- deliver a message to the wrong recipient, or
- lose the message and deliver it to no one.

Furthermore, the communication facility may deliver messages in an order different from the order in which they were sent and some messages may be duplicated and delivered more that once. Low level facilities can eliminate cases two and three above as well as preserve message order and eliminate duplicates.

The difficulty in recovering from communication failures is that it is not easy for the sender to determine what has happened when a response from the recipient is not forthcoming. If the response to a message is slow to arrive, the reason could be any of the following:

- the message was lost or garbled,
- the response was lost or garbled,
- the recipient is dead or died after receiving the message but before responding, or
- the recipient is simply slow to respond or the communication facility has delayed the original message and / or the response.

It is very difficult for the sender to distinguish between these cases. Without knowing why the response to some message has not arrived, the sender has to choose between retransmitting the message or aborting the transaction which caused the message to be sent. Retransmission must be handled carefully to avoid causing the recipient to act twice (e.g., the response was lost). In general, recovery from communication failures must be considered in the context of the particular message which has been sent. Recovery from communication failures is considered in sections 5.2 and 5.3.

The distributed transaction management system must establish protocols for initiating, migrating, and terminating transactions. Transactions can also be aborted at user or system request. Transaction commit and abort interact with concurrency control and recovery management. The transaction management facility is charged with controlling interactions among transactions, managing recovery and transaction abort, and supervising transaction initiation and termination. The following sections will discuss recovery facilities and the distributed transaction protocols used to coordinate the activity of multi-site transactions.

Chapter IV: Recovery Facilities

The recovery facility is designed to cope with many different kinds of failures. However, recovery facilities can be designed and discussed in terms of only three different recovery activities.

- *Transaction abort:* undoing the effects of an incomplete transaction during normal system operation.
- *Site restart:* bringing the on-line database to a transaction consistent state following an unplanned interruption of database processing at a single site.
- *Media recovery:* repairing damaged portions of the on-line, non-volatile database.

In-transaction back-out aborts a transaction and insures that no effects of the transaction remain in the database. It is used to recover from communication failures, system resource limitations, and also supports user requested transaction abort. Site restart recovers from hardware and software failures, including CPU and volatile memory failures, as well as any software initiated database restarts. Media failures occur whenever any portion of the on-line, non-volatile database becomes un-readable for whatever reason.

All three forms of recovery rely upon redundant representations of the database state. Usually a log or journal file is maintained as an alternative representation of the database. Logically, the log is a semi-infinite sequence of records which document all changes to the database state. One point of view is that the log is the real representation of the database because it contains the history of all modifications ever applied to the database. The on-line directly accessible data pages can be seen as an optimized database representation which allows rapid access to the most recent (current) database state.

Log records document all recovery relevant database events. Each (recoverable) database update is recorded. Transaction termination, both commit and abort, are recorded in the log. Recovery management also makes various notations in the log to facilitate its recovery procedures. These records include checkpoint records designed to bound the amount of log which has to be scanned during restart recovery.

The log is stored in volatile memory and in non-volatile storage. The non-volatile log may be duplexed for reliability. The writing of log records to non-volatile storage must be carefully synchronized with events affecting the database. Sometimes, it is necessary to *force* log records to the non-volatile log file in order to make some event recoverable. It is also important to control the order in which log records and the corresponding changed database pages reach the non-volatile store.

Database pages contain representations of the entities recognized by the database management facility. These entities include, but are not limited to, *files, record types, record instances, fields* in records, *indices* (record access paths including B-trees or hash tables), and *linked lists* (parent-child sets or sequences of record instances). When a database entity is updated,

created, or deleted the page(s) containing the entity representation are read into the volatile memory **buffer pool**, modified while in the buffer pool, and copied back to the non-volatile database. When a modified page leaves the buffer pool, it can be copied to its original position in the non-volatile store replacing the earlier version of the page or it can be copied to a new location in non-volatile storage leaving the old version available for recovery purposes.

Instead of updating the original copy, the page can be written to what is called a 'side file' or 'shadow page'. Eventually the shadow page must replace the original page. The shadow page can be copied to the original page slot at some later time or control information (e.g., file descriptor blocks) can be updated to refer to the shadow as the original. Shadow or side file recovery protocols tend to require extra I/O to update the original version of the page and may disrupt physical contiguity of logically related pages of data. Instead of investigating shadow page recovery mechanisms, we will be describing a recovery protocol which writes pages back to their original positions in non-volatile storage. Persons interested in shadow page mechanisms are referred to [27], [28], and [34].

Whenever a database entity is updated, inserted, or deleted, a record which describes the entity update is also added to the log file. The log record describes the entity update either in terms of the **physical** changes made to specific data pages or in terms of the **logical** database operations performed. For example, a logical log record for a record insertion would simply describe the file, record type, and field values of the new record. Insertion of the record into the access paths (e.g., indices) defined for the record type is implied by the logical log record. On the other hand, physical logging records the file addresses of and changes to all the pages modified by the insertion of the new record. This includes the page receiving the record as well as the index or hash table pages modified to support accesses to the new record. Physical logging tends to require that more information be recorded in the log but it also leads to simpler recovery procedures. In particular, media recovery is simplified because the log records relevant to particular file pages can be recognized easily.

Log records can contain information to either **redo** the logged action, to **undo** the logged action, or both. To redo an action, the new values or entities inserted into the database must be described. To undo an action, the values or entities destroyed must be described. Logs containing only redo records are called **forward** logs, while logs containing only undo records are called **backward** logs. Logs with both undo and redo records are known as **bi-directional** logs. Different recovery implementations use various combinations of forward and backward logs to support different aspects of recovery.

Transaction abort uses a backward log to undo the transaction's effects. If modified buffer pool pages are allowed to replace original pages in non-volatile storage before the transaction commits, then a backward log is required to undo the effects of uncommitted transactions during site restart recovery. Forward logging allows restart recovery to apply the effects of committed transactions to non-volatile pages which were not copied from the buffer pool when the transaction committed. In general, forcing modified buffer pool pages to non-volatile storage when the transaction commits will cause extra I/O, especially for high traffic pages. Forcing the relevant redo log records plus the transaction commit record to non-volatile will leave sufficient information in the log to recover the effects of committed transactions even if modifications in the volatile memory buffer pool are lost before being written to non-volatile storage. A forward log is also used to recover from media failures. An archived version of a damaged page can be brought up to date by applying the relevant redo log records of committed transactions. Note that the media recovery forward log does not need to contain any records for aborted transactions (but there probably won't be many anyway).

The log can be stored on any non-volatile media. In our discussions of recovery we assume that the log never fails. Actually, the log should be duplexed on devices with independent

failure modes so that a single log device failure will not prevent database recovery. In order to facilitate transaction abort, a backward log should be stored on some direct access device. Direct access enables one or more transactions to read old log records to perform transaction abort without disturbing logging for other ongoing transactions. Storing a bi-directional restart log on direct access devices also avoids tape handling during restart. On the other hand, tape storage provides good long term integrity and high capacity for the media recovery log.

If the on-line restart and transaction abort logs are stored in direct access storage, they must be confined to a bounded buffer. A fixed sized circular buffer allows new log records to be added to the head of the log until the buffer is filled. A log record at the tail of the buffer can be overwritten only if all of the following conditions hold:

- It is an undo record which is no longer needed for transaction abort: the corresponding transaction has already committed or aborted.
- It is a redo or undo record which is not needed for site restart: the corresponding transaction has completed and the page has been written to non-volatile storage.
- It is a redo record which is not needed for media recovery: the committed redo record has been copied to an archive tape or the corresponding data page has been copied to an image dump.

There must always be enough log space available to perform the logging required during restart plus enough to take system checkpoints. If necessary, the oldest transaction can be aborted, the changed pages of completed transactions can be forced to non-volatile storage, or redo records can be copied to an archive log tape to liberate space on the on-line log device.

Site restart recovery brings the database to a transaction consistent state using information in the on-line log. Restart determines which transactions committed ('winners'), which transactions aborted ('losers'), and which transactions were in progress at the time of the crash (also 'losers'). (Multi-site transactions introduce transactions whose restart status is in doubt. See section 5.3.) Insuring that all the effects of winners are in the database and that no effects from losers are present may require redoing database actions for winners and undoing the actions of losers.

Media failures could be handled by processing all redo log records from the archive and the on-line log to reconstruct the current state of damaged pages from scratch. However, the number of archive log records which have to be examined quickly becomes important. In order to limit the amount of log which has to be scanned, **image dumps** of the contents of all database pages are periodically constructed. Media recovery only needs to scan archive and on-line log records more recent than the image dump.

4.1: A Log File Protocol

We will now describe a recovery facility some detail. These recovery algorithms were designed with considerable help from my friends I.L. Traiger and J.N. Gray. The log is considered to be a sequence of log records each byte of which is numbered. Each log record is identified by the **log sequence number** (LSN) of its *last* byte. Log records are appended to the log by copying then into volatile log page buffers. Log page buffers are written to the non-volatile log file(s) when they fill or when some log record must be *forced* to non-volatile storage.

Every page update is described by a physical, bi-directional modify type log record. The modify log record identifies the updated page and describes the old and new page states. Also, all of a transaction's log records are linked together and contain the transaction-id. Each database page contains a **page-LSN** equal to the LSN of the latest modify record for the page. Figure 4.1 gives a pseudo declaration of the information fields of a modify log record.

```
DECLARE modify-record,          /* modify page log record */
    2 XACT-ID,                      /* transaction-id */
    2 PREV-LSN,                     /* preceeding log record for */
                                    /*   this transaction */
    2 PAGE,                         /* file address of page */
    2 MODIFICATION,                 /* undo & redo information */
        3 OFFSET,                       /* offset in page */
        3 SIZE,                         /* number of bytes modified */
        3 OLD-LSN,                      /* previous page-LSN  */
        3 OLD-VALUE,                    /* previous value */
        3 NEW-LSN,                      /* LSN of this log record */
        3 NEW-VALUE;                    /* new value */
```

Figure 4.1: Modify log record format.

We require that all pages modified by a transaction be locked exclusively before being updated and that the locks be held until the end-of-transaction. This restriction cannot be relaxed without careful consideration and modifications to the recovery algorithms being described. Page locking means that only one incomplete transaction can be updating a page. In particular, updates to a page by one transaction will never be interspersed with accesses to the page by other transactions.

Our discussion of the restart facility continues with descriptions of the protocol rules (algorithms) for the recovery relevant database activities. The relevant activities are page fetch, page write, page update, transaction commit, transaction abort, checkpoint, site restart, and media recovery. Log record types and recovery data structures will be introduced as needed.

4.2: Page Fetch

Whenever a database page must be read into the buffer pool, the buffer pool manager selects an empty buffer slot and then reads the required page into the buffer. If the page is un-readable (e.g., I/O error), media recovery is required. Otherwise, if the page is read successfully, the buffer pool manager adds a fetch record to the log. The fetch record contains only the page address of the page just read. It is used at restart to determine which pages were in the buffer pool at the time of the crash. The buffer pool manager associates the LSN of the fetch record with the buffer. The fetch-LSN gives a lower bound for the log records which may

have to be re-applied to the page during restart.

4.3: Page Update

The data managers which update the various database entities must all obey the recovery protocol in their use of the buffer pool and the log. Before updating a page, the page must be locked exclusively and **pinned** in the buffer pool. Pinning the page holds it in the buffer pool and will cause it to be fetched if it is not already present. A page cannot be updated unless it is pinned. Before unpinning a modified page, the data manager must add a modify record to the log and the LSN of (the last byte of) the modify record must be recorded in the page's page-LSN.

Only allows unpinned pages can be written to disk, so a page can be inconsistent during an update. Since a page update is accomplished as part of a single database action, it can usually be arranged that a page will not be pinned for very long. A page can be unpinned as soon as the update is complete, the log record has been inserted into the log buffer, and the page-LSN has been updated.

4.3: Page Write

The buffer pool manager is allowed to select and write any dirty, unpinned page as long as the log records for all updates to the page have already been written to non-volatile storage. If log records for all updates on the page have not been written, the page cannot be written back to non-volatile storage. However, if the dirty page selected for replacement has been in residence or unused for long, it is likely that the relevant log records will already have been written. This rule for synchronizing logging and updates to non-volatile storage is called the Write Ahead Log rule. Not only are uncommitted updates allowed to replace previous values before the end-of-transaction, but also, updated pages do not have to be written at the end-of-transaction. This flexibility in page migration allows the buffer pool manager to use any page replacement strategy it chooses and also avoids extra I/O at end-of-transaction.

When the buffer pool manager writes a page and is notified of the completion of the I/O, we assume that either the correct page has been updated correctly or the correct page is detectably incorrect (un-readable). The buffer pool manager records page write completions by adding an end-write record containing the page address to the log. End-write records are used during restart in conjunction with fetch records to determine which pages may have been in the buffer pool at the time of the crash.

4.4: Transaction Commit

Transaction commit adds a commit record to the log and then forces the commit record and all preceeding log records to non-volatile storage. This insures that records of all of the transaction's updates (i.e., modify records) are copied to non-volatile storage along with the indication that the updates are committed. Once the commit record has been written, all of the transaction's locks can be released. Writing the commit record is the atomic action which commits the transaction's updates to the database. If the transaction has not updated any

database entities, no commit record is required because commit and abort are the same thing for read-only transactions. Section 5.3 will elaborate upon the mechanisms necessary to insure atomic and uniform transaction commit for multi-site transactions.

Note that transaction commit does not require that changed pages be forced to non-volatile storage. The modify records, plus the LSNs in the data pages are sufficient to reconstruct page updates if restart occurs before updated pages are written from the buffer pool. The commit record reliably indicates the fate of the transaction so that the transaction's updates will be redone if necessary during restart.

4.5: Transaction Abort

Transaction abort must undo all of the transaction's updates to database pages. However, we would like to avoid having to log undo activity. Otherwise, we would have to think about redoing an undo during restart.

Abort reads the transaction's modify records, starting with the most recent. Because each transaction's log records are linked together, abort does not need to search the log to find the needed undo records. Each modified page is pinned (may cause a fetch) and the logged update is undone using the old values from the modify record. Next the page-LSN is *reset* to the old LSN from the log. Finally, the page is unpinned. Modify records are read and undone until all of the transaction's updates have been undone. (The modify record with no predecessor in the transaction is the last one to undo.)

After all of the transaction's updates have been undone, an abort record is added to the log and the log is forced. Once the abort record is in non-volatile storage, the transaction's locks on modified pages can be released. Updates by aborted transactions may have to be undone (again) during restart if the pages are not written before a crash. However, only pages which were fetched before the abort record was written and were not written before the crash are candidates for undo during restart.

4.6: Checkpoint

From time to time, the recovery facility takes a **checkpoint**. The purpose of checkpoint is to limit the amount of log which must be scanned and the number of pages which must be fetched during restart. Checkpoint flushes old pages from the buffer pool and records the currently active transactions and buffer pool occupancy in a checkpoint log record. High traffic, dirty pages are not allowed to remain indefinitely in the buffer pool without being written to non-volatile storage. Otherwise, restart might have to go arbitrarily far back into the log to recover committed changes to un-written pages. Snapshots of current transaction activity and buffer pool occupancy allow restart to establish which transactions were active and which pages were in the buffer pool at the time of the crash.

Checkpoint begins by purging old pages from the buffer pool. For each (dirty) page in the buffer pool, the fetch-LSN associated with the buffer is compared with the LSN of the last checkpoint. If the page was fetched before the last checkpoint, it is flushed from the buffer pool. Pages are flushed by first pinning them to insure internal page consistency and then writing the page to non-volatile storage. The log may have to be forced to obey the Write

Ahead Log rule if there are recent updates to the page. After logging an end-write for the flushed page, a new fetch record is logged and the buffer's fetch-LSN is updated before unpinning the page. Flushing old pages before the checkpoint insures that no committed (or aborted) updates recorded before the penultimate checkpoint will have to be redone (or undone) during restart. Purging old pages is required not only to limit the amount of log which must be scanned during restart, but also because on-line log space cannot be reused so long as the records may be needed for restart.

After the old log pages have been flushed, checkpoint quiesces the system to capture an accurate snapshot of current transaction activity and buffer pool occupancy. The system is briefly halted at an action consistent point. That is, new database actions are deferred and on-going actions are completed (or backed-out) before the checkpoint snapshot is taken. Once the checkpoint record has been created and appended to the log the system is allowed to resume normal operation. The checkpoint record contains:

- a list of active transactions with the LSN of the first log record of each active transaction (the AT list),
- a list of pages in the buffer pool with the fetch-LSN of each page (the BP list).

The AT and BP lists allow restart to determine which transactions were incomplete and what pages were in the buffer pool when the crash occurred. Once the checkpoint record is logged, the log is forced and the log file address of the checkpoint is recorded in special non-volatile storage called the *emergency restart record*. This record quickly locates the most recent checkpoint during restart.

4.7: Restart

When the database management system has been halted without warning, the non-volatile database may be left in an inconsistent state. Uncommitted updates by incomplete transactions may have been written to the database. Futhermore, committed (or aborted) updates to pages in the buffer pool may not have been written to non-volatile storage. In the first case, modify log records written ahead of the data pages supply the information necessary to undo the uncommitted updates. In the second case, modify records in the log ahead of the commit (or abort) record are used to redo (undo) the committed (aborted) updates.

To restore the database to a transaction consistent state, restart recovery must:

- undo the effects of incomplete transactions which have been written to non-volatile storage,
- undo the effects of aborted transactions whose modified pages were written but whose undone modifications were left in the buffer pool,
- redo the effects of committed transactions whose updates were never written from the buffer pool.

For completed (aborted or committed) transactions, only pages left in the buffer pool have to be considered for undo or redo. On the other hand, any page at all may have to be undone to remove the effects of incomplete transactions. Establishing which transactions are incomplete and which pages were in the buffer pool is a prerequisite to undoing and redoing updates to the non-volatile database. Reflecting this, restart consists of three phases.

- *Analysis:* determines which transactions were incomplete and which pages were in the buffer pool at the time of the crash.

- *Undo:* removes the effects of incomplete and aborted transactions from the database.
- *Redo:* insures that all committed updates are present in the database.

Restart makes three scans over the log. Figure 4.2 illustrates the the restart log scans. The log is scanned forward during analysis and backward to undo incomplete and aborted transactions. Finally, a forward log scan redoes committed updates. Care must always be taken to make sure that if a crash occurs before the end of restart, another restart is still possible.

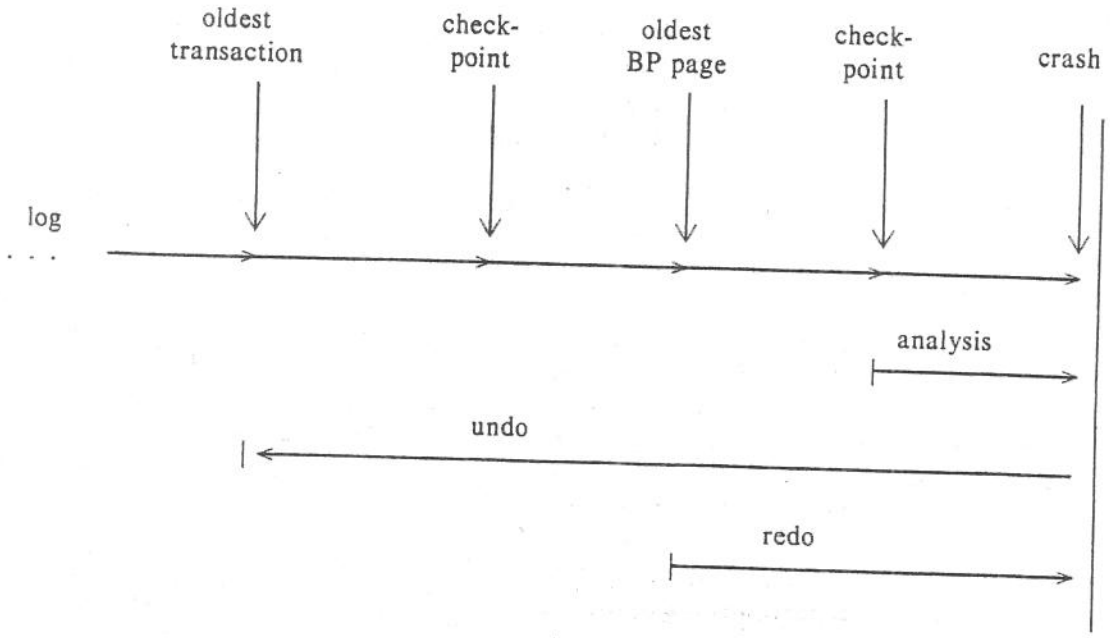


Figure 4.2 Restart Log Scans

4.7.1: Restart Analysis

Restart begins by reading the database emergency restart record to find the log address of the latest checkpoint record. The checkpoint record is read and the active transaction (AT) and buffer pool (BP) page lists used during restart are initialized from the checkpoint record. The log is then scanned forward from the checkpoint to the end of the log. Restart analysis reads each log record and processes them as follows:

fetch: If the page is not in the BP list, then add this page and the LSN of the fetch record to BP list. No modify records with LSNs less than the fetch-LSN will have to be redone.

end-write: If the page is in the BP list, remove it. The page was written to non-volatile storage so no earlier committed (or aborted) updates will need to be redone (or undone).

modify: If the transaction is not in the AT list, then add this transaction. As new transactions are encountered, they are entered into the AT list which will be used during restart undo to identify incomplete transactions.

commit (or abort): The transaction is removed from the AT list. Commit (and abort) records mark the completion of a transaction. Only pages in the buffer pool list may have to be redone (undone) to apply (remove) the transaction's effects.

When the end of the log is reached, the AT list will contain exactly those transactions which had not forced a commit or abort record to the log before the crash. The BP list will contain pages which were in the buffer pool at the time of the crash *or* had been written so recently that the end-write log record had not been written. While the AT list is exact, the BP list may overestimate the number of unwritten buffer pool pages.

4.7.2: Restart Undo

Restart undo insures that the effects of all aborted and incomplete transactions are removed from the database. Any update logged by an incomplete transaction may have been written to non-volatile storage and may have to be undone during restart undo. Updates by aborted transactions may have been written before the update was undone by a transaction abort. Aborted updates may have been left in the buffer pool if the fetch-LSN of a page in the BP list is less than the LSN of the transaction's abort record.

Restart undo also prepares for the redo phase. During redo, only committed updates should be applied to pages in the BP list. However, when reading the log forward, it is unclear whether modify records need to be applied because the commit or abort record comes later in the log. Restart undo scans the log backwards. Therefore, it sees the commit or abort record before the modify records. Restart undo identifies aborted updates and adds this information to the BP list. Log intervals containing modify records which should not be applied to pages in the BP list are linked to BP list entries during the undo phase.

Restart undo begins scanning backwards from the end of the log. All modify records of incomplete transactions must be processed to see if they must be undone. Therefore, restart undo must continue at least until all modify records of incomplete transactions have been processed. In addition, all updates by transactions which aborted after a page in BP was fetched must be processed. Restart undo begins by making a copy of the AT list and then marks each member of the original AT list as 'incomplete'. Also, all original members of the AT list are given abort-LSNs equal to the end of the log. The AT list copy will be used at the end of restart to log aborts for the incomplete transactions. During undo, aborted transactions are added to the AT list and transactions are removed after all their log records have been processed. Restart undo continues until the AT list is empty and all records with LSNs greater than the lowest fetch-LSN in the BP list have been scanned.

Only two log record types are relevant to restart undo. Abort and modify records are processed as follows:

abort: If the abort record LSN is greater than the minimum fetch-LSN in the BP list, then the transaction is added to the AT list. The abort-LSN and transaction-id are added to the AT and the transaction is marked 'complete'.

modify: The modify record does not have be be processed if the transaction is not in the current AT list. If the transaction is in the AT list and if the transaction is 'incomplete' then the page must be checked to see if the update must be undone. The page must also be checked if it is in the BP list and the page's fetch-LSN is less than the transaction abort-LSN. To check a page, the page is fetched and the page-LSN is compared to the modify record LSN. If they are *equal* then the update is undone and the page-LSN is reset to the old LSN from the modify record.

If the transaction is in the AT list ('complete' or 'incomplete'), and the page is in the BP list, then an aborted interval is added to the BP list entry. The aborted interval is from the modify record LSN to the abort-LSN of the transaction. Finally, if this is the first modify record for the transaction (i.e. no predecessor), the transaction is removed from the AT list. When the AT list is empty and the log position is prior to the lowest fetch-LSN in the BP list, restart undo is complete.

4.7.3: Restart Redo

When restart undo has completed undoing the effects of incomplete and aborted transactions, restart redo applies the effects of committed transactions to pages in the BP list. Only pages in the BP list need to be considered because committed updates will only be missing from pages which were updated but not written from the buffer pool before the crash. Restart redo can begin at the lowest fetch-LSN in the BP list. Only modify records need to be processed during redo. They are processed as follows:

modify: If the page is in the BP list, then check to see if the modify record LSN is in one of the aborted intervals associated with the page. If the modify is *not* in an aborted interval, then compare the modify record LSN with the fetch-LSN in the BP list. If the modify LSN is greater than the fetch-LSN, then fetch the page. If the page-LSN of the fetched page is equal to the *old* LSN in the modify record, then redo the update and set the page-LSN to the modify record LSN.

When restart redo reaches the end of the log, redo is complete. Restart recovery is completed by adding abort records to the log for all 'incomplete' transactions (using the copy of the AT list) and then taking two checkpoints. The abort records are needed to leave the log in a consistent state. Taking two checkpoints has the effect of purging the buffer pool of all updated pages and leaves the system in a clean state and ready for normal processing. Note that restart can be interrupted by another restart without getting confused.

4.8: Media Recovery

The recovery facility being described also supports the recovery of damaged database pages using dumps, log tapes, and the on-line log. Media recovery is easy to implement because physical logging allows us to quickly identify which pages are are updated by which log records. Media recovery can be invoked without stopping the database management system, but damaged pages cannot be referenced until they have been completely recovered. First we discuss the formation of image dumps and the creation of the archive log. Then, media recovery for a single damaged page is described.

An image dump is formed by scanning over the pages of the database and making a copy of each page on the dump media (usually tape). If the dumped pages are transaction consistent, then media recovery does not have to undo changes to dumped pages. A transaction consistent image dump can be obtained by first halting and then restarting the system. (Recall that restart recovery brings the database to a transaction consistent state.) Next, the entire database is copied to the dump media. Finally, normal database activity is resumed.

If it is impractical to halt database operations for the time it takes to perform the image dump, a **fuzzy dump** of individual, transaction consistent pages can be obtained. Fuzzy dump locks each page for reading, copies the page to the dump media, and then unlocks the page. The read lock insures that no incomplete transaction has modified the page. As in the case of the

brute force image dump, fuzzy dump makes a copy of all the pages of the database. Page images include the page-LSN. The dump should be organized to make it easy to locate particular pages in the dump.

The **archive log** contains log records used to redo updates to image dump pages. Let dump-LSN be the LSN at the time the (fuzzy) image dump was begun. The archive log used to recover pages from the image dump must contain redo records for all updates by committed transactions which were logged after the dump-LSN. Media recovery is simplified if the archive log does not contain records for updates which were undone because the transaction was aborted. A forward only archive log can be derived from the on-line log by eliminating all records from aborted transactions.

The archive log is created by starting at the dump-LSN in the on-line log and scanning forward. During archive logging an Active Transaction list is maintained as follows:

> If the current modify record's transaction is not in the AT list, then scan forward in the log to find the commit or abort record for the transaction. If the transaction is not complete (end of log reached), then archive logging cannot proceed and must wait for the transaction to complete. Otherwise, add the transaction to the AT list with an indication of whether it committed or aborted. When the transaction owning a modify record is in the AT list and it was committed, then redo information from the modify record is added to the archive log. The page address, the new values, the old LSN, and the modify record LSN are recorded in the archive log. If the transaction was aborted, the modify record is skipped. When abort or commit records are encountered while scanning the on-line log, the corresponding transaction is removed from the AT list.

Archive logging can be continuous or it can be activated periodically to capture redo records before they are overwritten in the on-line log. Archive log tapes should be stored and labeled by the range of LSNs they contain. Media recovery will require specific archive log tapes depending upon which image dump is used to supply the page.

Media recovery for a single page is fairly simple. First the page is locked exclusively to keep transactions from trying to access the page while it is being recovered. Next, the page is located and read from some image dump. Any dump will do but the most recent one is best. Locating and accessing the page image will require tape handling. Updates to the base page are then redone using the archive logs and then the on-line log.

Restoration of the page begins with the archive log tape containing the dump-LSN of the image dump. This archive log tape and all subsequent archive log tapes must be located and mounted in order. The archive log is scanned for updates to the damaged page. If the old LSN in the archive log equals the current page-LSN of the page being reconstructed, then the logged update is redone and the page-LSN is updated to the modify record LSN from the archive log.

When the end of the archive log is reached, the on-line log must be scanned for more recent updates to the damaged page. Committed updates are located in much the same way as they are when the archive log is created. The on-line log scan is begun at the LSN of the last record in the archive log. When a modify record for the damaged page is located in the on-line log, the commit or abort record for the corresponding transaction must also be located. (It must exist because media recovery has the page locked so no incomplete transaction can have modified the page.) If the update was committed, media recovery redoes the update to the damaged page. Aborted updates are skipped during media recovery.

When the end of the on-line log is reached, the damaged page has been brought up to its most recent transaction consistent state. Writing the page to its home slot in non-volatile

storage and unlocking the page completes the media recovery. Clearly, more that one page may be recovered at once. However, the pages cannot be stored into their home slots in non-volatile storage before they are completely recovered because site restart will not remember that the pages were damaged and locked. Restart and media recovery can be extended to allow use of home slots and the buffer pool for damaged pages under repair but care must be taken not to allow partially repaired pages to be used like new following site restart.

4.9: Recovery Conclusions

We have described a complete, single site recovery facility. The log file protocol supports transaction abort during normal processing, site recovery to a transaction consistent state, and media recovery for damaged on-line data. The Write Ahead logging rule allows modified database pages to be written back to their original slots both before or after transaction commit. The local site recovery facility described must be extended only slightly to support recovery for distributed databases. Section 5.3.2 discusses some of the extensions required to recover multi-site transactions.

Chapter V: Transaction Initiation, Migration, and Termination

We turn our attention to the management of multi-site transactions in a distributed database system. Once initiated at some site, the transaction is allowed to migrate to other sites of the system. No special set-up messages are required to migrate the transaction to new sites. The requestor does not need to know whether the transaction is already active at the target site. We will discuss the information which is exchanged to insure reliable cooperation among the sites of the distributed database. We also explain how multi-site commit can insure that either all on none of the sites in a transaction commit their updates to the database.

5.1: Transaction Initiation

A new transaction is created when an application program running at some site asks its local database manager to initiate a new transaction. The initiating site is known as the transaction site of origin. The local database manager assigns a *globally unique* transaction-id and allocates and initializes the local transaction descriptor. The transaction descriptor is stored in volatile memory and is *not* recovered following a site restart. The local transaction descriptor includes:

- *Transaction-id:* a handle uniquely identifying the transaction and it's site of origin
- *Transaction recoverable state:* state to which the transaction will recover if there is a local restart
- *Transaction timestamp:* provides timestamps for messages sent by the transaction
- *Transaction low water mark (LWM):* transaction timestamp of the earliest event at this site
- *Sites used:* a list of sites used by this transaction along with low water marks for those sites

The transaction-id distinguishes the transaction from all other transactions throughout the distributed database. All resources used, messages sent, and recovery records written by the transaction will be labeled with the transaction-id. A globally unique transaction-id is constructed by concatenating a local (recoverable) counter to the site of origin's (inter)network address. To avoid issuing a duplicate transaction-id, the local counter must never be allowed to recycle or reset. The recoverable transaction state is used to coordinate transaction commit for multi-site transactions. The recoverable state is designated as UNKNOWN before the transaction is created, during transaction execution, and after completion of the transaction. Only during multi-site transaction commit are other recoverable states required.

The transaction timestamp provides sequence numbers for messages sent by the transaction. It is initially zero at the site of origin. The low water mark (LWM) is the timestamp value of the event which caused the transaction to become known at the site. It is zero at the site of origin and equal to the timestamp of the first work request message received at other sites. Sites used is the set of sites used directly and indirectly by this site to accomplish the work of the transaction. Associated with each site used is the reported LWM of the site. Initially, the sites used list contains only the local site and LWM.

5.2: Transaction Migration

Once a transaction has been created at its site of origin, it can perform accesses to the local database and it can **migrate** to other sites to access remote data. We will not discuss the details of establishing, authentication, and protecting communications between database managers at different sites. We simply assume that local database managers can fabricate and send messages which are delivered with non-zero probability to database managers at other sites. When access to a remote database is required, the local database manager prepares a ***work*** request message describing the remote database actions required. The remote database system performs the requested work and sends the ***answer*** to the requesting site.

After sending a work request, the transaction at the sending site waits for the answer. We assume, unless otherwise stated, that the transaction will wait for an answer before sending any more work requests. Although we allow transactions to migrate from site to site without any restrictions, we allow only the last site visited to generate new work requests. This avoids unrestricted parallelism within a transaction. In particular, we avoid problems of transaction indeterminacy which can occur when parallel work requests are allowed. This restriction can be relaxed in many situations.

Satisfying a work request might require the receiving site to send work requests to other sites to accomplish the work of the original message. As the semantic content of the work request message is increased, subsidiary work requests become more likely (e.g., 'order more wigits from the lowest bidder'). In any case, the transaction management facility should not place any constraints upon how a transaction migrates through sites of the distributed database. In particular, transactions must be allowed to migrate to a site by different routes (see figure 5.1). Also, a site which is waiting for a answer to an outstanding work request should be able to accept and service incoming work requests (see figure 5.2).

The work request and answer messages carry transaction management information to coordinate multi-site transactions and detect site failures. The transaction clock is used to timestamp all messages of the transaction. It is incremented before sending a work request or answer and the new value is included in the message. The message timestamp establishes a global ordering of events in the transaction. The work request message includes:

- Transaction-id: globally unique identity which contains the address of the site of origin
- Timestamp: the requestor's transaction clock
- Requestor-id: return address for the answer
- Work request: describes the database accesses to be performed

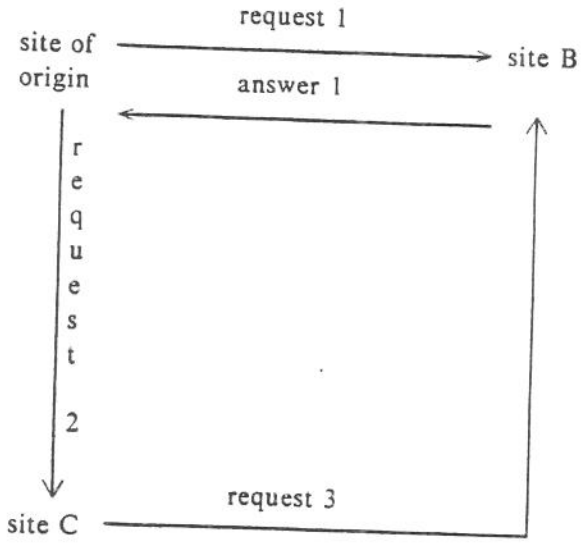


Figure 5.1: Transaction migrates to site B from both site of origin and site C.

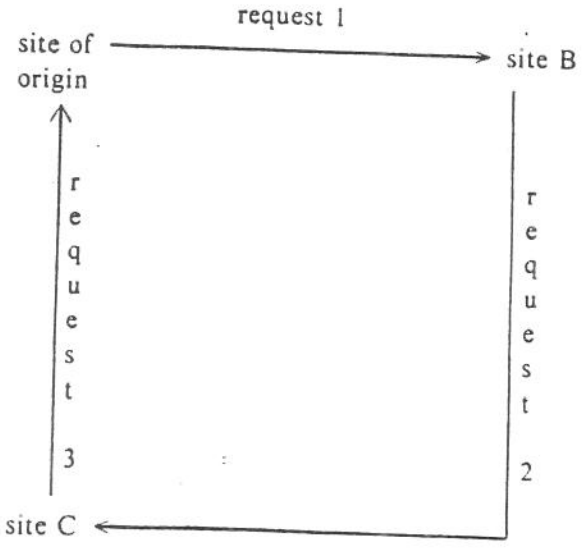


Figure 5.2: Transaction migrates back to site of origin before request 1 is completed.

When the requested work is completed by the receiving site, an answer message is constructed and sent to the requestor. The answer message which includes:

- Transaction-id
- Timestamp of the work request causing this answer
- Local transaction clock (timestamp of the answer)
- Sites used with their low water marks
- The results or answers

When a database manager receives a work request, a check is make to see if the transaction is already active at the site. If not, a recovery scope and concurrency partition is initialized for the transaction and a local transaction descriptor is allocated and initialized. The transaction clock and the local low water mark are initialized to the value of the timestamp in the work request message. The recoverable state is **UNKNOWN** and the sites used set contains the local site and LWM. If the transaction is already active at the site, only the transaction clock needs to be updated. If the incoming timestamp is greater than the local transaction clock, the local clock is set equal to the work message timestamp.

Automatic transaction initialization upon receipt of a work request must be handled carefully. We are assuming that any site can unilaterally abort the local effects of any transaction as long as the transaction has not begun the commit procedure. If a site unilaterally aborts a transaction between the receipt of two work requests, the second work request will re-establish the transaction without complaining to the requesting site. Both the requesting and the aborting site will think everything is fine unless some precautions are taken.

The local low water marks and monotonic message timestamps are designed to solve this problem. Each site maintains a low water mark equal to the timestamp of the first work request message serviced for the transaction. Also, each site collects a list of sites used directly or indirectly by the transaction along with the reported low water mark of each site. When answering a work request, the worker includes its site list with the answer. The requestor compares the received site list to its local site list. If a site is not in the local sites used list, the site and its low water mark are added to the local site list. If the site is already in the list (has been used before), the reported low water mark is compared to the low water mark in the local sites used list. If they differ, then the site in question has aborted and restarted the transaction. When the low water mark protocol detects a mid-transaction abort by some site, the entire transaction must be abandoned (aborted) because some of the early work of the transaction has been lost. Note that including the sites used list in the answer propagates sites used information towards the site of origin. Indeed, whenever the site of origin receives and processes an answer, it's sites used list includes *all* sites used so far by the transaction. The transaction commit protocols used for multi-site transactions will make use of this fact.

5.3: Transaction Commit

Transaction management in a distributed processing environment is concerned with achieving transaction atomicity regardless of the number of sites which have participated in the transaction. That is, the transaction either succeeds or fails everywhere. Success means that all of the transaction's updates (and action or commit messages) occur irreversibly at all sites

visited by the transaction. Failure means that there are no side effects of the transaction anywhere (except for conversational I/O). When more that one site becomes involved in a transaction, special protocols are required to insure that all the sites involved succeed or fail uniformly. The distributed transaction commit protocols require extra messages to insure that all sites make the same decision with respect to committing or aborting the transaction.

In order to preserve the autonomy of each site involved in a transaction, we wish to allow any site to unilaterally abort or backout a transaction (and reclaim the resources locally allocated to the transaction) at any time. Unfortunately, we know of no scheme which allows this while preserving the all or nothing character of the distributed transaction. However, we will describe protocols which allow participating sites to abandon a transaction at any time up to the moment when the site agrees to participate in the commit procedure. The commit protocols to be described insure that all the sites involved in the transaction come to the same decision about whether to commit or abort despite lost messages and site failures. The protocols to be described are called **two-phase** commit protocols [23], [27]., [33]. Not only do the two-phase commit protocols require extra messages to insure consistency, but they also imply a loss of local autonomy to permit global coordination of the transaction outcome.

5.3.1: The Two-Phase Commit Paradigm

The two-phase commit protocol allow multiple sites to coordinate transaction commit in such a way that all participating sites come to the same conclusion as to the disposition of the transaction. One site, the transaction **coordinator**, makes the final commit / abort decision *after* all the other sites in the transaction have already agreed to respect the coordinator's decision. During the first phase of the two-phase commit protocol, all sites of the transaction are queried as to whether they can commit their part of the multi-site transaction (e.g., they haven't already unilaterally aborted). Each site becomes *recoverably* prepared to go either way and awaits the coordinator's decision. Once all sites are prepared to commit, the coordinator makes its decision and all sites are notified to commit the transaction during phase two of the commit protocol

Being prepared to commit means that the site will not require any additional resources to commit the transaction. Obtaining additional resources could cause deadlock which might back-out the transaction. Futhermore, the resources already obtained are **sequestored** until the outcome of the transaction is determined and the site is notified to commit or abort. Once a site expresses its willingness to commit, it is no longer allowed to unilaterally abandon the transaction. Forbidding sites from unilaterally abandoning a transaction during commit processing compromises local site autonomy but it allows the coordinating site to assume that sites will remain able to commit (or abort) until the coordinator is able to decide which way to go.

5.3.2: Recoverable Transaction States

The multi-site commit protocol must preserve transaction atomicity and uniformity despite lost messages and site failures. This implies that sites must remember that they are involved in multi-site transaction commit despite the loss of virtual memory during a site restart. Therefore, we define the *local recoverable state* of a transaction at a site to be the state to which the transaction will return following restart recovery. In order to be recoverable, the transaction state must be recorded in non-volatile storage. The local log file makes a suitable repository for the recoverable transaction state. Other authors (Lampson and Sturgis) use special records in non-volatile storage to record the recoverable transaction state.

We have already mentioned that the initial transaction state is UNKNOWN. This corresponds to a transaction which has no state change records in the log. Also, the state is UNKNOWN when the transaction is completed and all volatile records of the transaction have been flushed. It is important to be able to unify the recoverable states of unbegun and complet-

ed transactions because, without unlimited memory for recalling completed transactions, there is no way to quickly distinguish between new and old transactions. Therefore, the commit protocols to be described are designed to take correct action without having to distinguish between unbegun and completed transactions.

Besides the **UNKNOWN** state, multi-site transactions can assume the **READY** state. Transactions enter the **READY** state after being asked to prepare for multi-site commit but before responding to the request. Becoming **READY** implies that the site will not abort or commit the transaction until it receives notice of the coordinator's decision. Futhermore, the site is able to recover the effects and resource reservations of the **READY** transaction following a site failure. When a transaction becomes **READY** at a site, the shared (read access) resources held by the transaction can be released. Because commit or abort are the only continuations possible for the **READY** transaction, no further references to the shared resources will be made. Therefore, updates by other transactions will not affect the **READY** transaction. If only shared resources are held at the site, it may be possible to avoid passing thru the **READY** state because commit and abort are the same thing to a read-only transaction. In any case, all resources held exclusively (update access) must remain sequestered until the commit or abort is complete.

There are some other recoverable states which will be introduced as needed in the descriptions of the two-phase commit protocols. Besides the recoverable transaction states, there is unrecoverable transaction status which is may be lost during restart. In particular we distinguish between 'inactive' transactions and transactions which are 'active' and have a transaction descriptor in volatile memory at the site. It is assumed, during normal operation, that if a transaction is 'inactive' at a site, its recoverable state is **UNKNOWN**. Note, however, that 'active' transactions can also be (and usually are) in the **UNKNOWN** state.

5.3.3: The Linear Two-Phase Commit Protocol

We begin by describing a commit protocol in which the sites of a multi-site transaction are ordered into a sequence and the commit proceeds as messages travel from site to site in the sequence. This protocol was described by J.N. Gray in his 'Notes on Data Base Operating Systems' [23]. The commit procedure begins when the site of origin decides to initiate the commit. This site, having received answers to all its work requests, has a complete list of the sites used by the transaction. Each site, starting with the site of origin, enters the **READY** state and then notifies the next site is the list to prepare for commit. When the last site is notified to prepare, all the other sites are already **READY** so the last site can make the commit decision and notify its predecessor of its decision. As each site is notified to commit, it commits and propagates the commit to its predecessor.

Let us examine the sequence of actions and the transaction state changes accompanying the actions more closely. Figure 5.3 depicts the messages sent and the new states assumed during a successful execution of a three site linear commit. Figure 5.4 shows the state transitions of the linear commit protocol. Finally, figure 5.5 gives a complete description of all the events, messages and states of the linear commit protocol.

To begin with, the site of origin initiates the commit process by becoming recoverable **READY** to commit. This requires that an in-doubt log record containing the list of sites used be forced to non-volatile storage. After forcing the in-doubt record, the site of origin sends a ***prepare*** for commit message to the next site in the sites used list. The ***prepare*** message contains the transaction-id and the sites used list with the site of origin at the head of the list.

Any site receiving a ***prepare*** message must check to see whether the indicated transaction is 'active' at the site. If not, the ***prepare*** is acknowledged with an ***abort*** message to the sender. Otherwise, the receiver enters the **READY** state by forcing a in-doubt record (with the sites list)

to the local log and then ***prepare*** message is sent to the next site (if there is one). Because a site must remain **READY** until it is notified of the coordinator's decision, the sender of a ***prepare*** message depends upon the receiver to respond eventually. If no response is forthcoming, the sender can resend the ***prepare***, but no other actions by the **READY** sender are allowed.

The last site in the sites list becomes the commit coordinator. It is the coordinator who makes the decision to commit or abort. If the transaction is 'active' at the last site and no other conditions prevent the commit, the last site directly enters the COMMITTING state (bypassing **READY**) by forcing a commit record to the log. Writing the commit record to the log at the last site is the atomic action which irrevokably commits the transaction. Once committed, the coordinator can release the transaction's local locks and resources.

After committing, the last site sends a ***commit*** message to its **READY** predecessor. Recall that each **READY** site is depending upon its successor to respond to the ***prepare*** request with either a ***commit*** or ***abort*** answer. Upon receiving a ***commit*** response, the **READY** site enters the COMMITTING state by forcing a commit record to the log. Once committed, the COMMITTING site acknowledges the ***commit*** from its successor and sends a ***commit*** to its predecessor. Each COMMITTING site must maintain its volatile memory transaction descriptor until the ***commit*** message is acknowledged. This is required because the ***commit*** message be lost in transmission. When the ***commit*** is acknowledged, the site enters the **UNKNOWN** state by logging done, dropping the transaction descriptor from volatile memory, and forgetting about the transaction.

When the site of origin receives the ***commit***, it commits by forcing a commit record. After acknowledging the ***commit***, return to the **UNKNOWN** state by dropping all volatile memory records of the transaction. ***Abort*** propagates in the same way as ***commit***. Interested readers are encouraged to study the state diagrams to understand how abort is handled.

5.3.4: The Centralized Two-Phase Commit Protocol

Instead of propagating the ***prepare*** and ***commit*** messages from site to site, a single site can send all the ***prepare***s and ***commit***s. A protocol similar to the centralized commit protocol has been described by Lampson and Sturgis [27] and by Jim Gray [23]. The centralized commit protocol assigns the role of coordinator to the site of origin. Centralized control of the commit process allows all the sites of the transaction to prepare and commit in parallel. Figure 5.6 illustrates the sequence of messages sent during a successful execution of a three site centralized commit procedure. Figure 5.7 shows the state transitions of the centralized commit protocol, while figure 5.8 gives a complete description of all the events, messages and states of the centralized commit protocol.

Commit is initiated at the site of origin where a complete list of the sites used in the transaction is available. First the coordinator recoverably enters the COLLECTING state by forcing a ***collect*** record with the list of sites used to the log. As soon as the coordinator has recoverably recorded that commit has begun, it sends a ***prepare*** message to every other site in the transaction. Each site is asked to vote on whether to commit or abort the transaction. The ***prepare*** message causes the other sites to get **READY**, so the coordinator is responsible for saving the other sites from the **READY** state.

Upon receipt of a ***prepare*** message, any site at which the transaction is still active sequesters its resources and recoverably enters the **READY** state (i.e., force an ***in-doubt*** record to the log). Once **READY**, the site sends a ***vote yes*** response to the coordinator. The **READY** site then waits for the coordinator's decision to commit or abort. If the transaction at the site is read-only, then the site can drop out of the commit protocol early. The read-only site can send a special ***vote yes*** informing the coordinator that the site does not need to participate in phase two

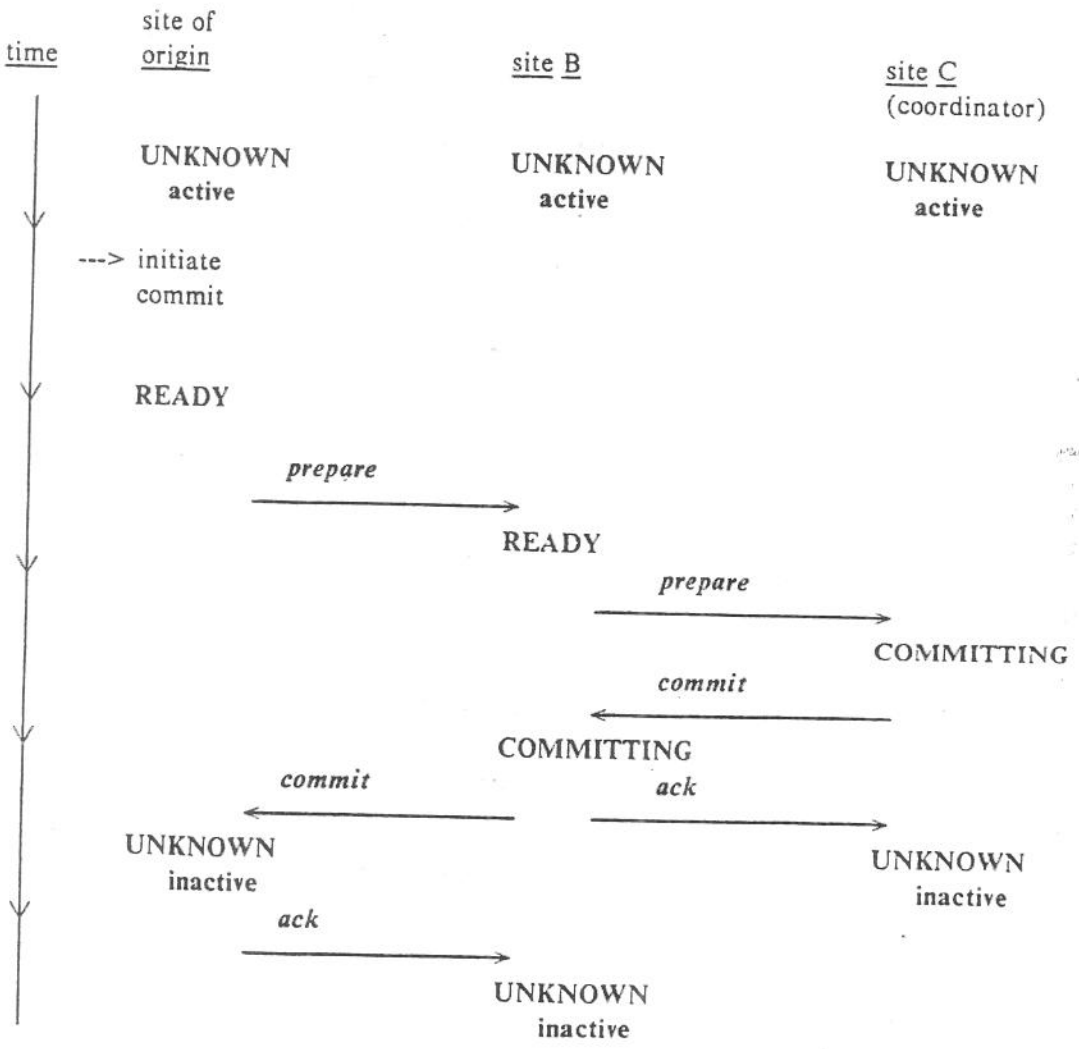


Figure 5.3: Linear Commit Protocol

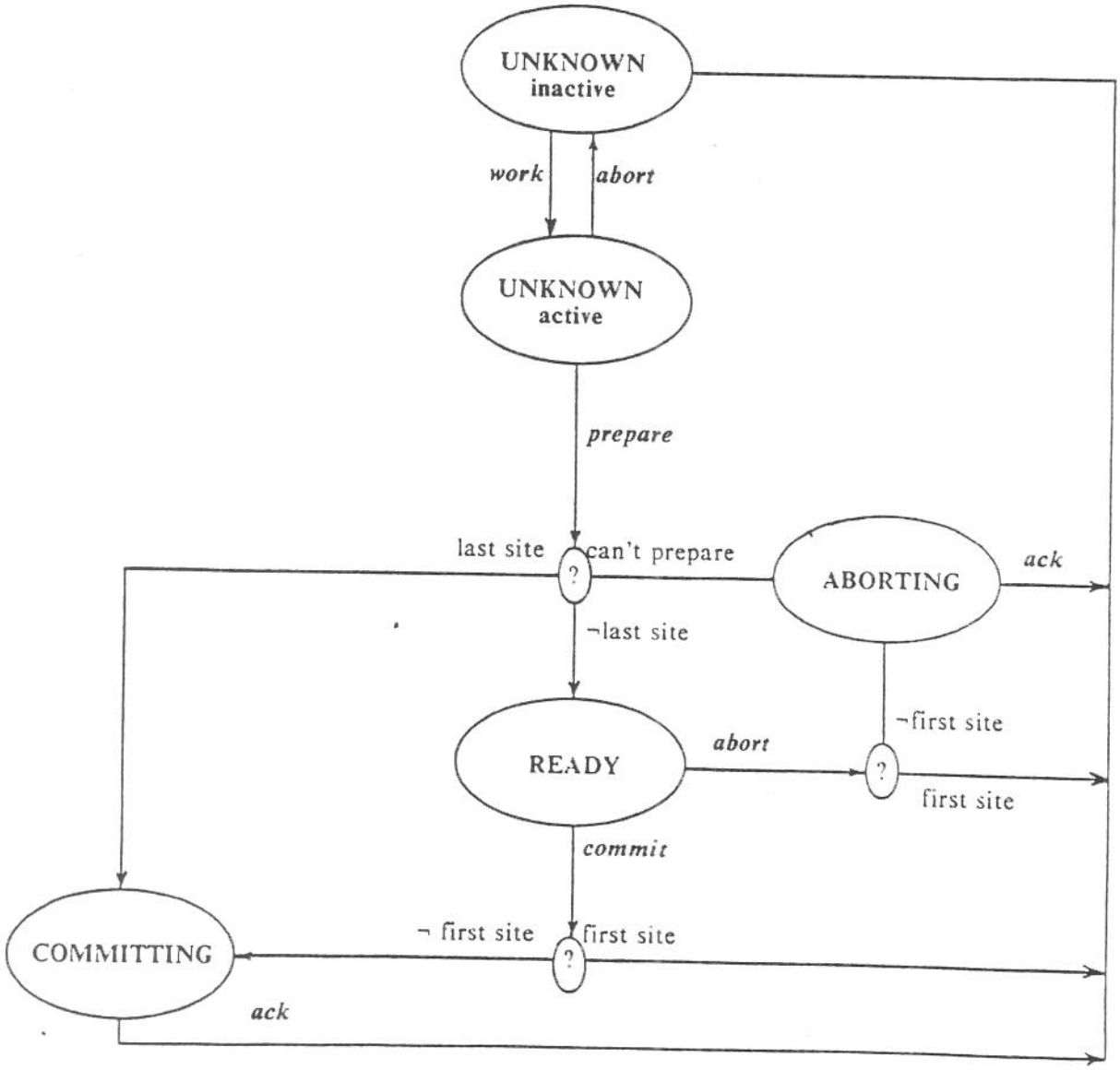


Figure 5.4: Linear Commit State Diagram

Messages & Events	UNKNOWN inactive	UNKNOWN active	READY	COMMITTING	ABORTING
work	UNKNOWN active *answer*	UNKNOWN active *answer*	Protocol Error	Protocol Error	Protocol Error
answer	Protocol Error	UNKNOWN active	Protocol Error	Protocol Error	Protocol Error
prepare	UNKNOWN inactive *abort*	can't prepare--> UNKNOWN inactive *abort* ¬ last site --> READY *prepare* to successor last site --> COMMITTING *commit* to predecessor	READY *prepare* to successor	COMMITTING *commit* to predecessor	ABORTING *abort* to predecessor
commit	UNKNOWN inactive *ack*	Protocol Error	¬first site --> COMMITTING *commit* to predecessor *ack* to successor first site --> UNKNOWN inactive *ack* to successor	COMMITTING *ack* to successor	Protocol Error
abort	UNKNOWN inactive *ack*	UNKNOWN inactive	¬first site --> ABORTING *abort* to predecessor *ack* to successor first site --> UNKNOWN inactive *ack* to successor	Protocol Error	ABORTING *ack* to successor
ack	UNKNOWN inactive	Protocol Error	Protocol Error	UNKNOWN inactive	UNKNOWN inactive
initiate commit	Not Allowed	READY *prepare* to successor	Not Allowed	Not Allowed	Not Allowed
timeout	Not Applicable	UNKNOWN inactive *abort* to site of origin	READY *prepare* to successor	COMMITTING *commit* to predecessor	ABORTING *abort* to predecessor
local restart	UNKNOWN inactive	UNKNOWN inactive	READY *prepare* to successor	COMMITTING *commit* to predecessor	ABORTING *abort* to predecessor
local abort	Not Applicable	UNKNOWN inactive *abort* to site of origin	Not Allowed	Not Allowed	Not Allowed

Figure 5.5: Linear Commit State Transition Rules

of the commit protocol. After voting for commit, the read-only site can immediatly drop its volatile memory transaction records. (Note that if the special ***vote yes*** is lost in transmission, the transaction will eventually be aborted. The reader should try to figure out why this is so.) If the site cannot commit (e.g., the transaction was already locally aborted), a ***vote no*** response is sent to the coordinator.

The coordinator collects the votes from the rest of the sites. If any ***vote no*** messages arrive, then the transaction will be aborted. On the other hand, if *all* the sites respond with a ***vote yes***, then the transaction can be committed. As soon as the coordinator knows that all the other sites are **READY** and waiting, it commits the transaction by recoverably entering the COMMITTING state (i.e., forcing a ***commit*** record to the log). Once the ***commit*** record is written, the coordinator sends ***commit*** messages to the other sites and releases the transaction's local resources. After sending the ***commit*** messages, the coordinator must collect acknowledgments before all records of the transaction can be purged from volatile memory.

When a **READY** site receives a ***commit*** message, it forces a commit record to the log and acknowledges the ***commit*** to the coordinator. The transaction's local resources are released and all volatile memory transaction records are released. If the coordinator sends ***abort*** instead of ***commit***, the **READY** site undoes the local effects of the transaction, forces an abort record to the log, and acknowledges the ***abort*** to the coordinator. The coordinator collects all the ***commit*** (or ***abort***) acknowledgments before releasing its volatile memory records of the transaction.

Site restart will cause the coordinator to resend all the ***prepare***, ***commit***, or ***abort*** messages depending upon the recoverable state at the time of the crash. The **READY** sites will wait indefinitely for the coordinator to tell them what to do. Local restarts will return **READY** transactions to the **READY** state and reacquire all the sequestered resources. Lost messages will cause timeouts which will either abort the transaction or cause some messages to be retransmitted. The reader is urged to study the state diagrams to more completely understand the behavior of the centralized commit protocol.

5.3.5: Comparison of Linear and Centralized Commit Protocols

The two protocols described have different advantages and disadvantages. The linear protocol always requires fewer messages but the centralized protocol requires fewer message delays when three or more sites are involved. Figure 5.9 compares the performance of the two protocols. Note that the centralized protocol can drop read-only sites from the second phase while the linear protocol cannot. A mixed approach in which two (and possibly three) site transactions use the linear protocol while all other multi-site transactions use the centralized protocol is possible.

5.4: Transaction Management Conclusions

Distributed databases require special message protocols to control the migration and termination of multi-site transactions. Special care is required to achieve uniform and atomic transaction commit if individual sites are allowed to unilaterally abandon incomplete transactions. Resources must be sequestered and extra messages exchanged to insure uniform transaction commit in the presence of site and communication failures.

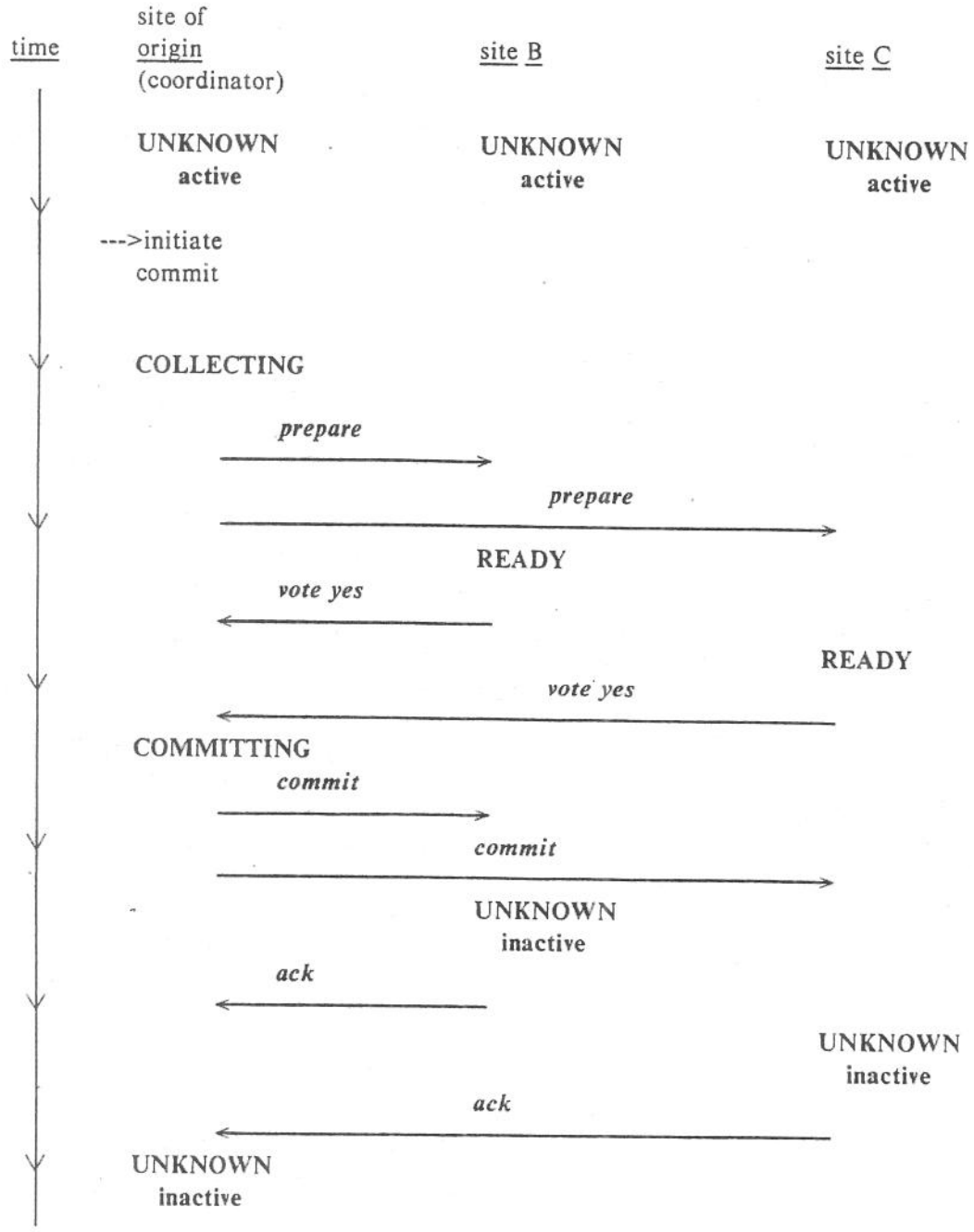


Figure 5.6: Centralized Commit Protocol

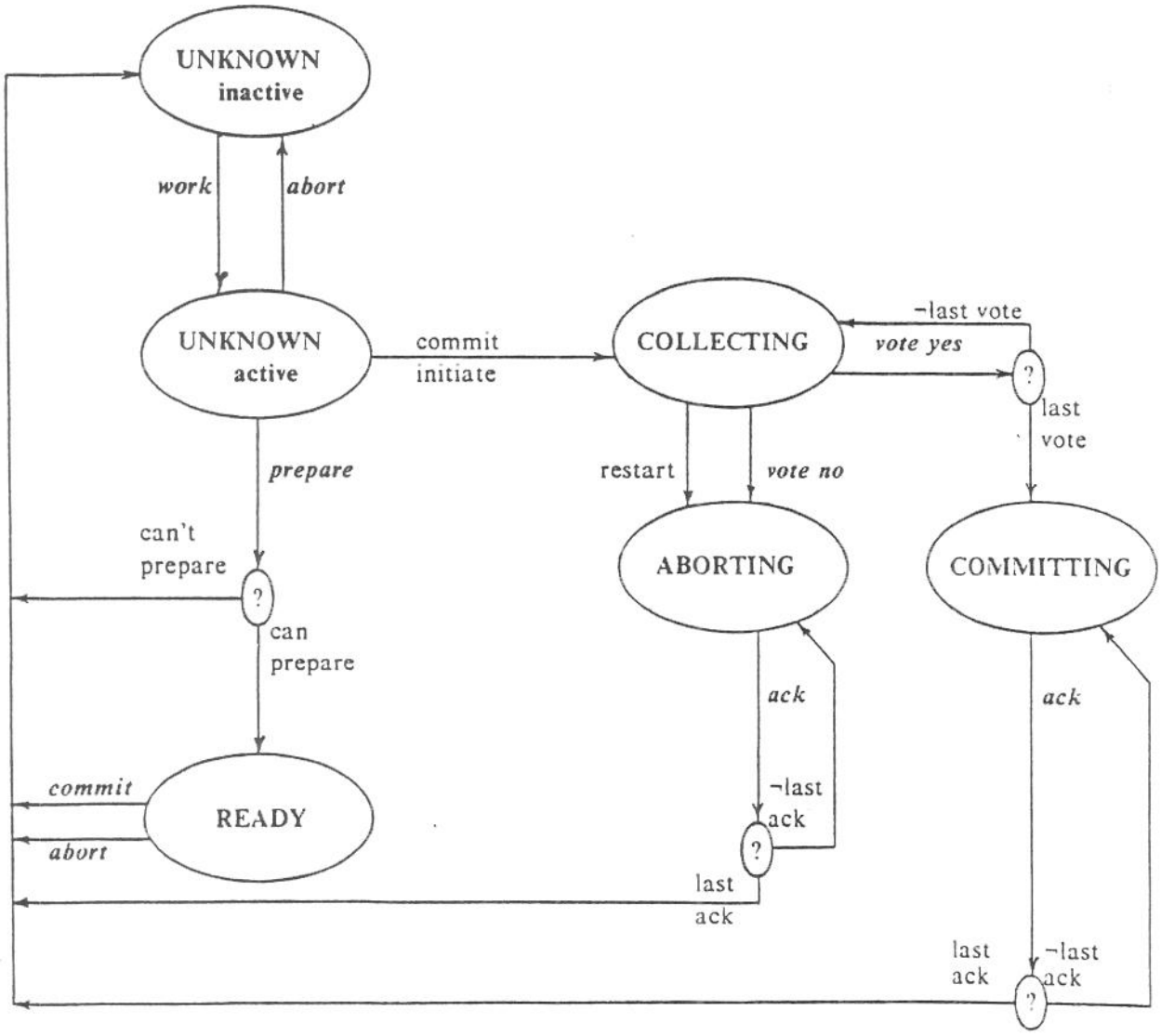


Figure 5.7: Centralized Commit State Diagram

Messages & Events	UNKNOWN inactive	UNKNOWN active	READY	COLLECTING	COMMITTING	ABORTING
work	UNKNOWN active *answer*	UNKNOWN active *answer*	Protocol Error	Protocol Error	Protocol Error	Protocol Error
answer	Protocol Error	UNKNOWN active	Protocol Error	Protocol Error	Protocol Error	Protocol Error
prepare	UNKNOWN inactive *vote no*	can't prepare--> UNKNOWN inactive *vote no* can prepare --> READY *vote yes*	READY *vote yes*	Protocol Error	Protocol Error	Protocol Error
vote yes	Protocol Error	Protocol Error	Protocol Error	last vote --> COMMITTING *commit* to all sites ¬last vote --> COLLECTING	COMMITTING	Protocol Error
vote no	Protocol Error	Protocol Error	Protocol Error	ABORTING *abort* to all sites	Protocol Error	ABORTING
commit	UNKNOWN inactive *ack*	Protocol Error	UNKNOWN inactive *ack*	Protocol Error	Protocol Error	Protocol Error
abort	UNKNOWN inactive *ack*	UNKNOWN inactive *ack*	UNKNOWN inactive *ack* to coordinator	Protocol Error	Protocol Error	Protocol Error
ack	UNKNOWN inactive	Protocol Error	Protocol Error	Protocol Error	last ack --> UNKNOWN inactive ¬last ack --> COMMITTING	last ack --> UNKNOWN inactive ¬last ack --> ABORTING
initiate commit	Not Allowed	COLLECTING *prepare* to all sites	Not Allowed	Not Allowed	Not Allowed	Not Allowed
timeout	Not Applicable	UNKNOWN inactive *abort* to site of origin	READY successor	ABORTING *abort* to all sites	COMMITTING *commit* to all sites	ABORTING *abort* to all sites
local restart	UNKNOWN inactive	UNKNOWN inactive	READY successor	ABORTING *abort* to all sites	COMMITTING *commit* to all sites	ABORTING *abort* to all sites
local abort	Not Applicable	UNKNOWN inactive *abort* to site of origin	Not Allowed	Not Allowed	Not Allowed	Not Allowed

Figure 5.8: Centralized Commit State Transition Rules

	Linear	Centralized
total messages	3(N - 1)	4(N - 1)
message delays	2N - 1	4
message delays to commit point	N - 1	2
recoverable state changes	2N	2N
read-only sites	no help	-2 messages per read-only

Figure 5.9: Comparison of Linear and Centralized Commit

REFERENCES

[1] Alsberg, P.A. and Day, J.D., 'A Principle for Resilient Sharing of Distributed Resources', *Proc. 2nd International Conf. of Software Engineering*, San Francisco, CA, October 1976, pp. 562-570.

[2] Astrahan, M. M., et. al., 'System R: A Relational Approach to Data Base Management', *Transactions on Database Systems 1, 2* (June, 1976).

[3] Bell, LaPadula, 'Secure Computer Systems', ESD-TR-73-278, (AD 770768, 771543, and 780528). MITRE, Bedford Mass., Nov. 1973.

[4] Bernstein, P.A., *et.al.*, 'Concurrency Control in SDD-1: A System for Distributed Databases; Part 1: Description', Computer Corp. of America Technical Report CCA-03-79, January 1979.

[5] Bernstein, P.A., *et.al.*, 'The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)', *IEEE Transactions on Software Engineering, SE-4*, 3 (May 1978), pp. 154-168.

[6] Bjork, L.A., 'Recovery Scenario for a DB / DC System', *Proc. ACM National Conference*, Atlanta, GA, 1973.

[7] Boyce, R. F., and Chamberlin, D.D., 'Using a Structured English Query Language as a Data Definition Facility', IBM San Jose Research Report RJ1318, December 10, 1973.

[8] Chamberlin, D. D., and Boyce, R. F., 'SEQUEL: A Structured English Query Language', *Proc. ACM-SIGMOD Workshop on Data Description, Access, and Control*, Ann Arbor, Michigan, May 1-3, 1974, 249-264.

[9] Chamberlin, D. D., Gray, J. N., and Traiger, I. L., 'Views, Authorization, and Locking in a Relational Data Base System', *Proc. National Computer Conference*, Anaheim, Calif., May 19-22, 1975, 425-430.

[10] Codd, E.F., 'A Relational Model of Data for Large Shared Data Banks', *Comm. ACM 13, 6* (June, 1970), 377-387.

[11] Codd, E. F., 'A Data Base Sublanguage founded on the Relational Calculus', *Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control*, San Diego, Calif., November 11-12, 1971, 35-68.

[12] Codd, E. F., 'Further Normalization of the Data Base Relational Model', Courant Computer Science Symposia *6*, 'Data Base Systems', Prentice-Hall, New York City, May 24-25, 1971, 33-64.

[13] Codd, E. F., 'Relational Completeness of Data Base Sublanguages', Courant Computer Science Symposia *6*, 'Data Base Systems', Prentice-Hall, New York City, May 24-25, 1971, 65-98.

[14] Codd, E.F., 'Recent Investigations in Relational Data Base Systems. *Information Processing '74* (Proc. IFIP Congress, Stockholm, Sweden, August 5-10, 1974), North-Holland, Amsterdam.

[15] Date, C. J., *An Introduction to Data Base Systems.* Addison-Wesley, 1975.

[16] Davies, C.T., 'Recovery Semantics for a DB / DC System', *Proc. ACM National Conference,* Atlanta, GA, 1973.

[17] Dennis and Van Horn, 'Programming Semantics for Multiprogrammed Computations', CACM Vol. 9, No. 7, July 1977. pp. 145-155.

[18] Diffie, W. and Hellman, M.E., 'New Directions in Cryptography', *IEEE Transactions on Information Theory* IT-22, 6 (Nov. 1976), 644-654.

[19] Eswaran, K.P., *et.al.*, 'The Notion of Consistency and Predicate Locks in a Database System', *Comm. ACM, 19,* 6 (November 1976), pp. 624-633.

[20] Fagin, R., 'On an Authorization Mechanism', *ACM Transactions on Database Systems, 3,* 3 (September 1978), pp. 310-319.

[21] Graham, G. S., and Denning, P. J., 'Protection -- Principles and Practice', *Proc. Spring Joint Computer Conference,* Atlantic City, New Jersey, May 16-18, 1972, 417-429.

[22] Gray, J.N., *et.al.*, 'Granularity of Locks and Degrees of Consistency in a Shared Data Base', in *Modelling in Data Base Mangement Systems,* (ed G.M. Nijssen), North Holland, 1976.

[23] Gray, J.N., 'Notes on Data Base Operating Systems', in *Operating Systems An Advanced Course,* Lecture Notes in Computer Science 60, (ed Goos & Hartmanis), Springer-Verlag, 1978, pp.393-481.

[24] Griffiths, P., and Wade, B., 'An Authorization Mechanism for a Relational Database System', *ACM TODS, Vol. 1,* No. 3, Sept. 1976. pp. 242-255.

[25] Jones, A. K., 'Protection in Programmed Systems', Ph. D. thesis, Carnegie-Mellon University, 1973.

[26] Lampson, B. W., 'Protection', *Proceedings 5th Annual Princeton Conference,* Princeton University (March 1971), 437-443. Bedford Mass., Nov. 1973.

[27] Lampson, B.W. and Sturgis, H.E., 'Crash Recovery in a Distributed Data Storage System', *Comm. ACM,* to appear.

[28] Lorie, R.A., 'Physical Integrity in a Large Segmented Database', *ACM Transactions of Database Systems, 2,* 1 (March 1977), pp. 91-104.

[29] Nauman, J.S., 'Observations on Sharing in Data Base Systems', IBM Technical report TR03.047, Santa Teresa Laboratory, San Jose, CA, May 1978.

[30] Owens, R. C., Jr., 'Primary Acess Control in Large-Scale Time-Shared Decision Systems', Project MAC Technical Report TR-89, July, 1971.

[31] Redell, D. D., 'Naming and Protection in Extendible Operating Systems', Ph. D. thesis, University of California (Berkeley), Sept. 1974.

[32] Reed, D.P., 'Naming and Synchronization in a Decentralized Computer System, Ph.D. thesis, MIT, Cambridge, Mass, September 1978.

[33] Rosenkrantz, D.J., Stearns, R.E., and Lewis, P.M., 'System Level Concurrency Control for Distributed Database Systems', *ACM Transactions on Database Systems, 3,* 2 (June 1978), pp. 178-198.

[34] Severance, D.G. and Lohman, G.M., 'Differential Files: Their Application to the Maintenance of Large Databases', *ACM Transactions of Database Systems, 1,* 3 (September 1976), pp. 256-267.

[35] Stonebraker, M. R., and Wong, E., 'Access Control in a Relational Data Base Management System by Query Modification', Electronics Research Lab Memo No. ERL-M438, U. C. Berkeley, May, 1974.

[36] Stonebraker, 'Implementation of Integrity Constraints and Views by Query Modification', ACM SIGMOD Conference. May 1975. pp. 554-556.

[37] Summers, R. C., Coleman, C. D., and Fernandez, E. B., 'A Programming Language Approach to Secure Data Base Access', IBM Los Angeles Scientific Center Tech. Report G320-2662, May 1974.

[38] Thomas, R.H., 'A Solution to the Consurrency Control Problem for Multiple Copy Data Bases', *Proc. IEEE Compcon,* Spring 1978.

[39] Traiger, I.L., *et.al.*, 'Transactions and Consistency in Distributed Database Systems', IBM Research Report RJ2555, IBM Research Laboratory, San Jose, CA, June 1978.

[40] Wulf, W., et al., 'HYDRA: The Kernel of a Multiprocessor Operating System', *ACM Communications,* Vol. 17, No. 6 (June, 1974), 337-344.