DISASTER RECOVERY FOR TRANSACTION PROCESSING SYSTEMS

Christos A. Polyzois

CS-TR-371-92

June 1992

Disaster Recovery for Transaction Processing Systems

Christos A. Polyzois

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JUNE 1992

Acknowledgements

I would like to express my deepest gratitude to my advisor, Hector Garcia-Molina. He has been an endless source of knowledge, advice, help, and energy. Apart from a wonderful advisor, he has also been an invaluable friend.

I would like to thank my readers, Rafael Alonso and David Hanson, for their comments that improved this thesis as well as for their guidance and their always witty remarks during these years. Many thanks also to Kai Li for his help.

Abstract

A remote backup is a copy of a primary database maintained at a geographically separate location and is used to increase data availability. Remote backup systems are usually log-based and can be classified as either 2-safe or 1-safe, depending on whether transactions commit at both sites simultaneously or they commit first at the primary and are then propagated to the backup.

This thesis describes 1-safe algorithms that can exploit multiple log streams to propagate information from the primary to the backup. An experimental distributed database system is used to evaluate the performance of these algorithms and compare the 1-safe with the 2-safe approach under various conditions. Techniques for processing read-only queries at the backup are also presented.

Contents

List of Figures

Chapter 1

Remote Backups

As computing becomes more crucial to the operation of many enterprises, service disruptions become less tolerable. To achieve truly continuous operation, critical database applications often maintain an up-to-date backup copy at a geographically remote site, so that the remote site can take over transaction processing if the primary site fails. Such a remote backup database (or *hot standby* or *hot spare*) should be able to take over processing immediately. When the primary site recovers, the backup should provide it with a valid version of the database that reflects the changes made by transactions processed while the primary was down, which enables the primary to resume processing. During normal operation, the overhead at the primary for maintaining the backup should be kept low.

Local replication techniques (e.g., dual processors, mirrored disks) can mask many hardware failures. However, such techniques are inadequate for extensive failures (*disasters*) such as environmental hazards (e.g., fire, flood, earthquake), power outage, malicious acts, etc. Remote backups protect against these types of failures because their geographic separation reduces the likelihood that a disaster will affect both copies.

A remote backup may provide better protection against failures that can be masked by local fault-tolerance techniques because failures tend to propagate. To illustrate, consider mirrored disks, a common form of local replication. A repairman fixing one of a pair of mirrored disks accidentally damages the good disk, which is physically located next to its faulty mirror image [21]. The remote backup copy technique decouples the systems physically, so that failures are isolated and the overall system is more reliable.

Physical isolation can also contain some failures caused by operator and software errors. For example, an operator could destroy the database by reformatting its disks or by eliminating a critical file. Such incidents have been reported [21]; as hardware components become more reliable, human errors emerge as a major source of failures. It is much harder for one operator to destroy a remote database under the control of another operator. Similarly, software errors triggered by particular timing events at the primary will probably not occur at the backup. The backup will have errors of its own, but these are likely to occur at different times. Thus, remote backup copies provide a relatively high degree of failure isolation and data availability, and they are used in practice [22].

Disaster recovery systems have received significant attention recently. Jim Gray and Andreas Reuter [24] state:

> "As you can sense, we are very enthusiastic about disaster recovery systems. They offer a fertile ground for new algorithms, and also offer the promise of much higher availability ... It is fair to say that this is the most active area of transaction processing research."

Apart from taking over transaction processing in case of disaster, the backup may find use during planned downtime of the primary. For example, hardware upgrades, maintenance, database schema or software updates can be performed without interrupting transaction processing by using the backup while the maintenance, etc. is being performed at the primary and then reversing the roles to update the backup. If it has spare capacity, the backup can be used to perform useful processing during normal operation. For example, it can process read-only transactions.

1.1 Taxonomy

Recovery mechanisms produce either a consistent or an inconsistent backup copy. Consistency is highly desirable in most cases, since applications assume that they operate on correct data, and error handling in these programs is rarely comprehensive. In case of disaster, the backup takes over transaction processing and the applications would have to use an inconsistent copy if the backup does not preserve consistency. The inconsistency may lead to delays in transaction processing or even to crashes. Thus, compromising the consistency of the database may endanger its continuous operation.

Order-preserving backups preserve the logical order in which transactions committed at the primary. *Non-order-preserving* backups may commit transactions in a different order than the commit order at the primary. Non-order-preserving backups are usually easier to maintain, but they may lead to inconsistencies between the two copies, so they are rarely used [10]. In this thesis, we consider only consistent, order-preserving backups.

Backup systems can run *1-safe* or *2-safe* transactions [23, 30]. In 2-safe algorithms, the primary and the backup copy are kept in lockstep; all updates to the data are applied to both copies in synchrony, typically by employing some variance of the two-phase commit protocol [4, 20, 32, 41]. In 1-safe systems, transactions commit first at the primary site and are then propagated to the backup.

Two-safe is the way in which replicated data is traditionally handled in distributed systems [1, 4] and it offers two main advantages: conceptual simplicity (since applications are presented with a single logical view of the data) and guaranteed survival of all committed transactions in case of disaster (since changes are applied simultaneously to the two copies). However, 2-safe systems also have disadvantages. The agreement protocol must be executed once for every transaction and increases transaction response time by at least one primary-backup round trip delay plus some processing time at the backup. These delays may exceed one second [30] and force transactions to hold resources (e.g., locks, workspace, etc.) longer, thus increasing contention and decreasing throughput.

There are systems that can tolerate the delays mentioned above and run 2-safe transactions. Systems that cannot use 1-safe transactions. A disaster can cause some committed transactions to be lost. Consider, for example, a transaction T that executed and committed at the primary. If a disaster occurs at the primary before the backup receives the information that enables it to install T (typically the log entries written by T), transaction T will not survive the disaster. These losses occur only when disaster hits and are "economically acceptable" in applications with "very high volumes of transactions with stringent response time requirements. Typical applications include ATM networks and airline reservations systems [30]."

Apart from reducing contention for resources, 1-safe transactions have some other advantages as well. Suppose we have a system with some response time requirement. Typically, this may be something like "90% of the transactions must have response time below t_r." Say that when using 2-safety, the system can achieve a maximum throughput w. If we switch from 2-safety to 1-safety, the response time will drop for all transactions. Consequently, we can increase the load of the system beyond w and still meet the response time requirement. Thus, we can trade the gain in response time for an increase in throughput.

A third advantage of 1-safety is the simplification of processing when the backup becomes unreachable. With 2-safety, the primary site must either suspend processing or change the way it processes transactions to skip the agreement phase. When the backup becomes reachable again, it must catch up with the primary in a special processing mode. Then the two sites must synchronize, revert to the 2-safe mode, and resume normal processing. With 1-safety things are easier: if the backup becomes unreachable, the messages are simply accumulated at the primary and are sent to the backup later. No deviation from normal processing occurs. This behavior is especially convenient for short periods of disrupted communication; longer failures of communication links may require re-initialization of the backup anyway, since the backlog of messages may have grown too big.

It is easier to support multiple backups of the same database with 1-safety than it is with 2-safety. When 2-safety is used, the coordinator of a transaction must wait for messages to be received from all of the participants. Thus, the latest response sets the pace. The more sites there are, the more likely it becomes that one of them will introduce some delay and cause the other sites to wait. Furthermore, when the configuration changes (sites leave or join the set of active sites), all sites have to be notified. The problems mentioned above do not occur under 1-safety: each site operates at its own pace and independently.

Finally, 1-safe mechanisms can be combined with 2-safe mechanisms to yield hybrid schemes that run some transactions as 1-safe and some as 2-safe. For example, in a banking application, transactions involving large amounts of money could run as 2-safe (to ensure that they will not be lost in case of disaster), whereas the bulk of the transactions, involving small amounts, can run as 1-safe.

1.2 Thesis Outline

Unfortunately, very little has been published in the area of remote backup systems. Most publications come from industrial environments and describe particular products, focusing on their functionality and specifications rather than on the underlying principles. There are few performance evaluation studies and their scope is always limited to a particular system. We attempt a more systematic study of remote backups. In Chapter 2, we present existing systems and identify their shortcomings and limitations, which leads to the definition of our more general system model [15]. We highlight the difficulties that this model introduces, give formal definitions for the correctness of 1-safe backup algorithms, and state the design goals for such algorithms [14]. Chapters 3 and 4 present the dependency reconstruction algorithm [12, 13] and the epoch algorithm [11], respectively, which are two of our 1-safe algorithms. Chapter 5 describes our transaction processing testbed and our experimental results on the relative performance of various 1-safe and 2-safe schemes [17]. In Chapter 6, we discuss how a remote backup can assist in case of partial failures at the primary [15], and we present techniques for processing read-only queries at the backup [16] in Chapter 7.

Chapter 2

Framework and Problems

Commercial products that provide backup services are available. Tandem provides a remote duplicate database facility (RDF) [31, 44] and is currently developing RDF-2, the second generation. IBM markets an extended recovery facility (XRF) [25], which is primarily for local backups. There is also an ongoing project at IBM [5, 34] to support remote backups for IMS databases. These packages provide utilities for dumping databases, monitoring them, and propagating modifications to a backup database.

The RDF and the IBM project are similar in their approach. It is not our intention to describe the full packages; we are interested in only the algorithms used for maintaining and initializing the backup database. We give an overview of their main features in order to characterize current technology and establish the basis for a comparison with our work. These systems can run either as 1-safe or as 2-safe. Like all existing backup products, they are log-based: a log of the transaction processing activity at the primary is sent to the backup so that it can install the same changes that were installed at the primary. This log is maintained at the primary for local recovery purposes, so it is not an additional overhead.

The log entries written at the primary are sent to the backup where they are received by a *control process* in the same order in which they were sent. The updates are actually installed by a number of *writing processes* that operate in parallel and are assigned tasks by the control process. Care is taken to preserve the relative order of modifications to the same data item. For example, modifications to a certain data item are always assigned to the same writing process.

Current 1-safe systems assume a *two-site* model: there is a primary site and a backup site. This configuration is shown in Figure 2.1. Multiple processors are allowed within each site. However, the logs of all primary processors are merged into a *single master* log stream

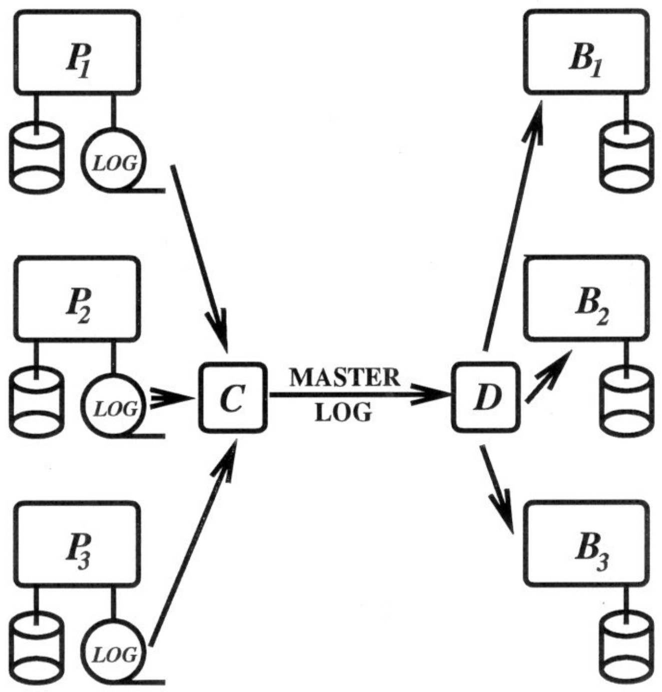


Figure 2.1: Log concentration

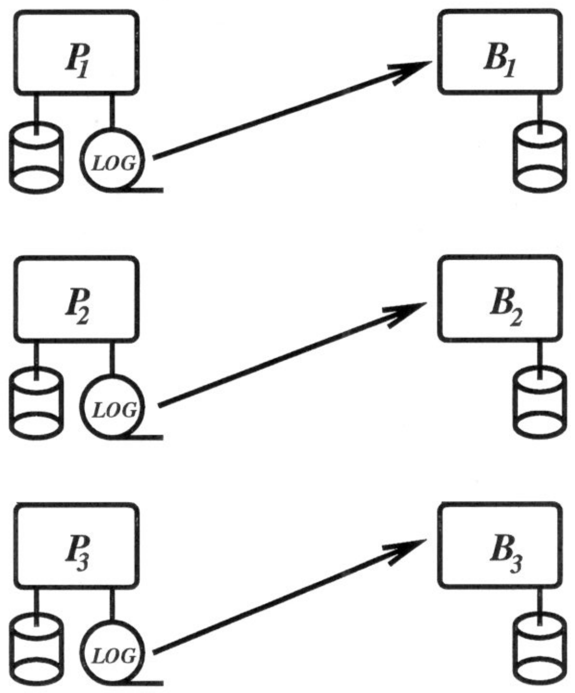


Figure 2.2: Multiple log streams

by a concentrator (denoted by C in Figure 2.1). The stream is sent to the backup and distributed among the backup processors by a distributor (denoted by D).

A preferable arrangement is to have multiple log streams between the primary and the backup computers. For example, there may be a stream between each pair of primary and backup peer processors, as shown in Figure 2.2. A single master log stream does not let a system scale upwards, since all information sent to the backup must go through the same concentrating processor that is connected to the communication line. The bandwidth of the single line to the backup is not usually a problem: very high-bandwidth lines are available. The bottleneck is usually the processing load at the concentrating computer. Suppose that this computer has a capacity of P instructions/sec and that processing a message containing log or control information requires x instructions. These instructions are necessary to run the communication protocols with primary computers, to merge the messages into the stream and to repackage them for retransmission. If a transaction generates m messages on the average and there are N processors running transactions, the throughput of each processor is limited to $P/(x \times m \times N)$ transactions per second, independent of the bandwidth of the communication link. A similar argument applies to the distributor at the backup.

Multiple log streams can also exploit multiple communication links. Furthermore, if the primary (or the backup) computers are geographically distributed, each interaction of a primary (backup) computer with the concentrator (distributor) involves a remote operation. Thus, merging and distributing the logs incurs unnecessary inter-site network traffic, so that it may actually be cheaper to send the log messages to the backup directly. The concentrator may also cause additional delays, since transactions must wait for an acknowledgement that their log writes have completed before they commit at the primary.

2.1 Our Generalized Model

Our model drops the two-site restriction. We assume that there is a single system, distributed across a collection of geographically separate sites $S_1, \ldots, S_n$. We place no restriction on what a site can be: it can range from a single processor to a number of computer installations covering an entire geographical region (e.g., computers on the east coast may be viewed collectively as a site). However, we expect that in many practical configurations the computers constituting a site will be in physical proximity and will be connected with communication links significantly faster and cheaper than the links between computers at different sites.

The database is logically partitioned into a number of fragments $F_1, \ldots, F_k$. A partition usually reflects the logical organization of the data, e.g., a relation or part of a relation. We assume that there is a *single logical* partition of the database for both the primary and the backup copy (or copies). The physical partition may vary, e.g., depending on the hardware used to store each copy. It may sometimes be useful to allow different logical partitions of the data at different copies, so our model covers this case as well.

Suppose we have two partitions, $P_a = \{F_{a_1}, \ldots, F_{a_s}\}$ and $P_b = \{F_{b_1}, \ldots, F_{b_t}\}$. Partition P_b is a *refinement* of P_a if every fragment F_{b_i} of P_b is a subset of some fragment F_{a_j} of P_a. Given a partition P_a, we can use its refinement P_b equivalently, if whenever a fragment of P_b is stored at a site, all other fragments of P_b that are subsets of the same fragment of P_a are also stored at the same site.

If we have a number of partitions $P_a, P_b, \ldots, P_q$, we can always find a partition P_r that is a refinement of all of the given partitions. Partition P_r is a *common refinement* of $P_a, P_b, \ldots, P_q$. Thus, we can always assume (without loss of generality) that there is a single partition of the database. If there is not, we can use a common refinement to replace all other partitions in the way indicated above.

The fragment is the smallest unit of data that can be allocated to a site. Fragments are assumed to be non-overlapping and they should cover the entire original database, i.e., each data item is in exactly one fragment. Fragments may be replicated at various sites. A fragment may also be replicated within a site, e.g., in mirrored disks, to increase its reliability. We assume that such redundancy is hidden by a lower level of abstraction that presents a virtual fragment, which fails only when local replication cannot mask the failure. Local replication can be taken into account in our model by adjusting the failure rates of

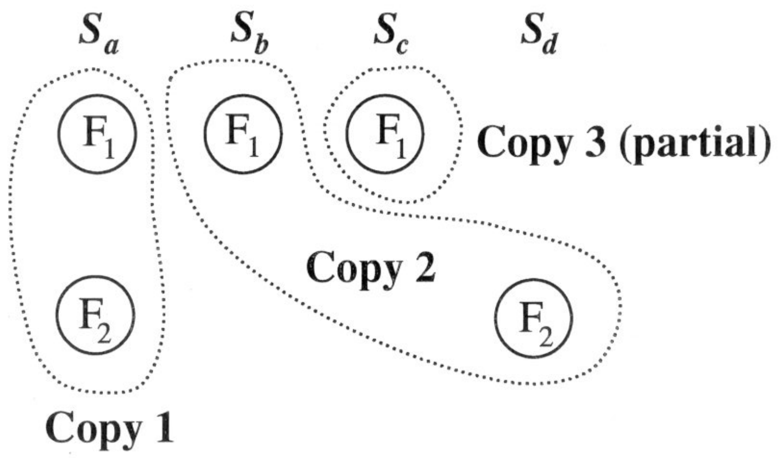


Figure 2.3: Database copies

such virtual fragments.

In 1-safe backups, there may be *multiple* logical versions of the database. For instance, one version may reflect a set T of transactions, another version may reflect a subset of T. Thus, different copies of the same fragment may be associated with different logical versions of the database. Unlike 2-safe copies, it is necessary to specify the association between fragment copies and database versions explicitly, as the example in Figure 2.3 illustrates. There are four sites, S_a, S_b, S_c, S_d, and fragments F_1, F_2. The copy of F_1 stored at S_a can be taken together (and kept consistent) with the copy of F_2 stored at S_a to form the primary copy of the database. A backup database copy can be formed by taking the copy of F_1 at S_b and the copy of F_2 at S_d and keeping them consistent. Finally, a partial copy of the database can be obtained by taking the remaining copy of F_1 at S_c.

It is not necessary for the primary copy to reside at a single site. For example, copy 2 could be the primary. In some cases, it may be more efficient for the primary copy to be at a single site, e.g., when many transactions access both fragments. However, in other cases, a multi-site primary may be desirable. For instance, the two fragments may be mostly accessed by disjoint, geographically separate sets of users running few distributed transactions, so it is best to place each fragment where it is most frequently accessed.

A database copy is a mapping of fragments to sites. We represent such a mapping with a set of ordered pairs of the form (*fragment, site*). For example, $C_i = \{(F_1, S_{i_1}), (F_2, S_{i_2}), (F_3, S_{i_3}), \ldots, (F_k, S_{i_k})\}$ implies that copy C_i has its F_1 fragment at site S_{i_1}, its F_2 fragment at site S_{i_2}, etc. A complete copy must specify a site for every fragment. The mapping must be total; partial mappings lead to incomplete copies, which are sometimes useful but are not considered here. However, the mapping does *not* have to be *onto* sites, since a copy may have no data stored at a particular site. Furthermore, the mapping can be many-to-one, since one site may hold many fragments. Mappings need not be static; failures may force a copy to map a fragment to a different site.

The rest of our discussion is in terms of database *copies*. A transaction is distributed if it accesses data in more than one fragment of a copy.

Different database copies correspond to different mappings. Two mappings C_j and C_k are different if they differ in at least one component. Since we are interested in disaster recovery, we would like our copies to be disjoint: to differ in *all* components, i.e., for copies $j, k\ \forall i, S_{j_i} \neq S_{k_i}$. Disjoint copies can tolerate the failure of *any* site, since each fragment is always stored in two sites. In general, if one has m copies any two of which are disjoint, one can tolerate the loss of *any* $m - 1$ sites. Depending on the particular backup scheme employed, the copies that remain operational may not be up-to-date or even consistent.

The model can encompass many practical configurations, e.g., a two-site model is a special case where there are two copies each mapped entirely to one site. Another interesting configuration is that of *cross-backups*, where a site acts as the primary for one part of the database and as the backup for another part. For example, if the database is distributed between Los Angeles and New York, the primary copy is $C_p = \{(F_{LA}, LA), (F_{NY}, NY)\}$ and the backups for each site are specified by $C_b = \{(F_{LA}, NY), (F_{NY}, LA)\}$.

2.2 1-Safe Processing

A single database copy (as defined in the previous section) is designated as primary and one or more other copies are designated as backup. If there are multiple backup copies, they track the primary copy directly and independently of each other; each backup copy is running a different instance of the backup algorithm. It is also possible to have second-level backup copies tracking a first-level backup, but doing so is analogous to a first-level backup tracking a primary, so we do not discuss this approach.

The computers that hold the fragments of each copy can communicate with each other over a wide-area network, a local-area network or bus, or even through shared memory. Thus, our techniques apply to shared-memory architectures as well as loosely coupled systems.

We assume that the primary copy is running a relational database management system. The database contains a set of *tables*, and each table contains a set of *records*. The tables have unique *names* and the records have unique *record ids*. Requests can be made to create or delete tables and to insert, select, update or delete records. Each of these requests provides the appropriate parameters. For example, updating a record requires the id of that record and its new version. The system will create and maintain indices on tables, with any subset of the fields forming the key. Basic transaction processing functions, such as locking, abort and commit operations, are also supported. Such requests are always associated with a transaction identifier, which is established with a *begin_transaction* request during the initialization phase of a transaction. We assume that strict two-phase locking is used for concurrency control.

Walking through the execution path of a transaction at both the primary and the backup establishes a reference point for subsequent chapters. A transaction may access data at one or more primary fragments. Each fragment maintains its own independent log, where it records the update activity by transactions to its data. Contrast our model with other models for parallel logs, e.g., Reference [2], where multiple logs are allowed, but all transactions have to go through a central processor referred to as the *back-end* controller.

The log contains only redo information or a combination of undo/redo information, and it can also be used for local recovery. The information may be at the logical level (action description) or the physical level (bytes changed within a page). For concreteness we assume a redo-only, logical log, but our discussion is applicable to any other kind. Thus,

field	meaning
`tid`	transaction identifier
`act`	action descriptor (e.g., update)
`tbl`	name of table involved
`key`	record identifier of record modified
`val`	after image of record modified

Table 2.1: Structure of log records

log entries have the fields shown in Table 2.1. Not all log entries contain all of these fields. For example, the log entry for a delete does not contain an after image.

When a transaction T completes its requests at all primary fragments, one of the fragments acts as the coordinator for the transaction and initiates a two-phase commit protocol to ensure atomicity. The coordinator notifies the participant fragments that the end of the transaction has been reached. Those participants that have executed their part of T successfully make a *prepare* entry in their logs, denoted by $P(T)$, and send a positive acknowledgement (a *participant_ready* message) to the coordinator. We assume that the $P(T)$ entry includes the identity of the coordinator.

Participants that were not able to complete their part of T successfully write an *abort* entry in their logs and send a negative acknowledgement to the coordinator. If a positive acknowledgement is received from all participants, the coordinator makes a *commit_coordinator* entry in its log, denoted by $CC(T)$, and sends a *commit* message to all participants. The participants place a *commit_participant* entry in their logs ($CP(T)$) and send an acknowledgement to the coordinator. $C(T)$ denotes either $CP(T)$ or $CC(T)$.

Running between the primary and backup fragments are several communication lines, which let primary fragments send their logs to the backup fragments. Control messages are also exchanged over these lines. The connections between primary and backup fragments are assumed to be virtual circuits [45], so that the order of messages is preserved. The existence of enough bandwidth to propagate the changes is assumed; however, for 1-safe transactions, communication delay is not critical. Note that redo-only logs reduce the amount of information that must be propagated to the backup.

The backup fragments receive the log entries from their primary peers on the communication lines in parallel and independently of each other. Their task is to install the updates of committed transactions atomically and in the same relative order as the primary (when the order matters). As we show in the next section, this task is difficult when there are multiple, independent log streams. After the logs are safe at a backup fragment (they have been installed or they have been written on non-volatile storage), the backup fragment sends an acknowledgement to its primary peer. The primary can then erase its logs, since the backup will not need them again.

An interesting issue in 1-safe backups is when the user submitting a transaction gets a response. If the response is sent after the transaction has committed at the primary site, then, in a disaster, the transaction may be lost if it does not commit at the backup. If the response appears after the transaction has committed at the backup, it is guaranteed that the effects of the transaction will not be lost. The transaction is 1-safe in both cases; only the user's information about the fate of the transaction is different.

As failures occur at the primary, the system tries to recover. We assume the fail-stop model for processors [39]. Multiple failures may slow down the primary or even stop it. At this point, a *primary disaster* is declared and the backup attempts to take over transaction processing. The declaration of a disaster will probably be done by a human because the backup cannot distinguish between a catastrophic failure at the primary and a break in the communication lines. User terminals are usually connected to both the primary and the backup. The backup connection is normally on standby and is used in case of disaster to route input transactions. For simplicity, we assume that when a primary disaster occurs, the hardware at the backup is fully operational. A *backup disaster* is similarly declared when failures impair the backup, and we assume that the primary is operational during a backup disaster.

A copy is a primary, a backup, or is recovering. At most one of the copies can be a primary at any time. Under normal operation, processing of the transactions takes place at the primary copy and sufficient information is sent to the backup copy to allow it to install the changes made at the primary. When a primary disaster is declared, the copy previously operating as backup starts operating as primary and all of the transactions are directed to it. When the failed copy comes up, it enters a recovering mode, which allows it to get a consistent, up-to-date copy of the database and (perhaps later) resume normal processing. The recovering mode is also used for creating the backup copy at system initialization.

2.3 Difficulties with Multiple Logs

Let T_x and T_y be two transactions such that T_x commits before T_y. A dependency $T_x \rightarrow T_y$ exists between transactions T_x and T_y if both transactions access a common data item and at least one of them writes it [4, 28]. Dependencies can be classified as write-write (W-W), write-read (W-R), or read-write (R-W) depending on the actions that generate them.

When a single log stream is used, two important properties are guaranteed.

- When the commit log message for a transaction is received at the backup, all of the transaction's actions have been received at the backup.

- When a transaction T is received at the backup, all of the transactions on which T depends have also been received at the backup.

These properties enable the backup to install the updates it receives on the log stream without further checking. It is tempting to generalize this approach to many log streams. However, in the case of multiple logs, the two properties mentioned above do not hold, and the consistency of the database may be violated. We illustrate with two examples.

Consider a distributed transaction T that accessed data at primary fragments P_1 and P_2. The transaction wrote log entries at both fragments and committed at both fragments. The log entries written at each fragment may follow different paths to the backup fragments. If a disaster occurs, it is possible that backup fragment B_1 (peer of P_1) receives the log entries of T while fragment B_2 does not. If B_1 installs the changes of T, transaction atomicity is violated. For example, suppose T is a funds transfer, crediting $Account_1$ at fragment P_1

and debiting $Account_2$ at fragment P_2. We do not want to execute only the credit at the backup.

It seems that an agreement protocol among backup computers might solve these problems, but it does not. Assume that there was a transaction T' such that $T \rightarrow T'$ at P_1. Since transaction T' depends on T and T cannot commit, B_1 should not commit T'. Doing otherwise may violate the consistency of the database. For example, suppose that T' is a withdrawal for which availability of funds depends on successful completion of the funds transfer by T. Thus, although T' may have been fully received at the backup, it is not allowed to commit, which may in turn prevent other transactions that depend on T' from committing, so that we have a *cascading* aborts effect. Some of these transactions may be distributed, so the effect may ripple across several backup fragments.

As these examples illustrate, in the presence of multiple logs, indiscriminate installation of updates is not possible. The backup must ensure atomicity of transactions and keep track of their dependencies, which explains why most systems do not use multiple log streams. For example, RDF permits multiple log streams between primaries and backups only if each log stream corresponds to an independent database, i.e., only if there are no distributed transactions that access data at computers with separate logs. 2-safe processing is imposed for transactions that access data at computers that do not share a log.

The algorithms presented in Chapters 3 and 4 can handle distributed transactions in the presence of multiple logs. Before presenting them, we give formal definitions of the correctness of 1-safe backup algorithms.

2.4 Correctness Criteria

Commercial systems are imprecise about what it means to "lose a few transactions." For instance, it is important to specify that if a transaction is lost, any transactions that depend on it cannot be processed at the backup.

The transaction processing mechanism at the primary ensures that the execution schedule PS of a set of transactions T is equivalent to some serial schedule. Schedule PS induces a set of dependencies on the transactions in T. The backup will execute a subset of the actions in PS, denoted BS. Read actions are not performed at the backup since they do not alter the database. The write actions that are executed simply install the value that their counterpart installed at the primary. Because of failures, not all write actions of PS appear in BS.

Requirement 2.1 (Atomicity) *If one action of a transaction T_x appears in BS, then all write actions of T_x appearing in PS must appear in BS. This requirement disallows partial propagation of a transaction.*

R is the set of transactions whose writes are in BS, $R \subseteq T$.

Requirement 2.2 (Mutual Consistency) *Assume T_x and T_y are in R. If dependency $T_x \rightarrow T_y$ holds in BS, dependency $T_x \rightarrow T_y$ must hold in PS. This requirement guarantees that the backup schedule is equivalent to the primary, at least as far as the propagated write actions are involved. Since no read actions take place at the backup, this requirement does not apply to R-W or W-R dependencies.*

Let M be the set of transactions that were not fully propagated to the backup before a failure and hence were not installed. In addition to these transactions, there may be other transactions that must not be installed at the backup. For example, suppose that T_x and T_y execute at the primary and T_x writes a value read by T_y. If T_x is not received at the backup (i.e., $T_x \in M$), we do not want to install T_y.

To illustrate this situation, say that T_x is the transaction that sells an airline ticket. It inserts a record giving the customer's name, date, flight involved, payment information, and so on. Transaction T_y issues a seat assignment. The updates produced by T_y cannot be installed at the backup without those of T_x: there would be no passenger record to update. Thus, we have the following requirement.

Requirement 2.3 (Local Consistency) *No transaction in R can depend on a transaction in M. That is, suppose there is a transaction $T_a \in M$ and there is a sequence of W-W* *and W-R dependencies (not R-W) in PS:* $T_a \rightarrow T_b \rightarrow T_c \rightarrow \cdots \rightarrow T_r$. *Then none of* $T_a, T_b, \ldots, T_r$ *is allowed to be in R. If C is the set of transactions that depend in this way on M transactions, then* $R \cap (M \cup C)$ *must be empty.*

R-W dependencies do not cause violations of the local consistency constraint. If $T_a \rightarrow T_b$ and the only dependencies between these two transactions are R-W, then the values installed by T_a cannot possibly affect the values installed by T_b. Thus, the backup can install the updates made by T_b and have a consistent database, even if T_a does not reach the backup.

Local consistency is similar to recoverability as defined in Reference [4]. Since we are also allowing W-W dependencies (not just W-R) in the dependency path of the definition above, local consistency comes closer to the concept of strict executions [4]. The motivation, however, is different in the two cases. In Reference [4], strict executions are introduced to avoid problems associated with uncommitted data being read or overwritten by another transaction and are intended to achieve correctness of a schedule within a *single* copy. In our case, local consistency is required to ensure the equivalence of *two* schedules. It only happens that both problems are dealt with in the same way: the actions of some transactions are delayed until some other transactions have committed. Since the two notions are quite different in context, we use a distinct name for our requirement.

Requirement 2.4 (Minimum Divergence) *The backup must be as close to the primary as possible, i.e., the backup must commit as many as possible of the transactions it is given. In other words, if a received transaction does not depend on any transaction in M (i.e., does not belong to C), then it has to belong to R, i.e.,* $T = R \cup M \cup C$.

We have implicitly assumed that the primary and backup schedules PS and BS run on the same initial database state. In Chapter 3, we present a procedure for initializing the backup database so that this property holds.

Most replicated data mechanisms described in the literature (e.g., References [1] and [4]) would not allow what we call missing transactions. That is, they have $M = \emptyset$ (2-safe transactions). As discussed in Chapter 1, they use a two-phase commit protocol to achieve this property. The weaker constraints we have defined in this section admit more efficient algorithms. In essence, transactions commit at the primary (using a local two-phase commit that involves primary fragments), release their locks, and only then are propagated to the remote backup.

A remaining issue is the semantics of missing transactions. In particular, is it valid to lose transactions that have committed at the primary? Is it reasonable to discard transactions that were propagated, but violate the local consistency requirement? As discussed in Chapter 1, real applications do allow missing transactions and can cope with the consequences.

The example of Section 2.3 illustrates the consequences. Consider a banking application where a transaction T_x transfers \$1,000 into an empty account, and a transaction T_y withdraws \$400 from the same account. Both of these transactions run at the primary just before a disaster. Transaction T_x is lost, but T_y arrives at the backup. To satisfy the local consistency constraint, T_y is not installed at the backup. The backup is consistent, but not consistent with the real world. The inconsistencies with the real world can be detected and corrected manually. For instance, when the customer checks the balance of the account (at the backup) and sees zero instead of \$600, the customer can present the bank with the transfer receipt and ask for a correction. The bank, knowing that a disaster occurred, will be willing to make amends for missing transactions. The bank might lose some money (especially if withdrawal transactions are lost), but this cost might be much less than the cost of providing 2-safe transactions. Furthermore, not all transactions have to be 1-safe; transactions that involve large sums of money can be performed with 2-safe protocols.

Finally, there is the issue of whether T_y should be discarded even if it arrived at the backup. Installing T_y makes the backup inconsistent (e.g., there are not enough funds in the account to cover this withdrawal). On the other hand, discarding T_y creates an inconsistency with the real world, and it is more important to maintain database consistency. As discussed in Chapter 1, without consistency, it is very hard to continue processing transactions. Furthermore, T_y can be saved for special manual processing. A bank employee can look at the T_y record and decide on the best compensating transaction to run. Since disasters are rare, the price to pay for this special processing of the relatively few transactions that are discarded is probably acceptable.

2.5 Design Goals

Subsequent chapters present 1-safe algorithms for maintaining remote backup copies. The algorithms were designed with the following goals.

Database Consistency As mentioned in Chapter 1, the consistency of the backup is important to ensure continuous transaction processing. The consistency criteria were defined in the previous section.

Scalability Existing algorithms often have a component that must process all transactions in some way, e.g., a master log. As systems scale upwards and use multiple computers for transaction processing, the performance of this component eventually forms a bottleneck, even if the amount of processing per transaction is small. We want algorithms that avoid such bottlenecks, so that they apply to very large systems.

Parallelism at the Backup Although it is rather difficult to give a measure for parallelism, a disaster recovery scheme should exploit the parallelism at the backup. For example, suppose that transactions are processed sequentially at the backup. If the primary uses multiprocessing to allow parallel execution of transactions, the backup will be unable to keep pace with the primary's higher throughput, and the backup copy will become out-of-date.

Primary Overhead Minimization We try to minimize the overhead induced by the backup algorithm at the primary. During normal processing, the backup is simply installing updates, as opposed to the primary, which is actually running the transactions. Yet, the backup should be capable of processing transactions after a disaster, so it will have spare capacity during normal processing. Thus, if we have an option of performing some task either at the primary or at the backup, we do it at the backup.

These goals are desirable in many applications, but they do not hold in every case. For example, the backup may not have spare capacity during normal processing (e.g., if it is being used for some other, non-critical processing), or database consistency may not be significant.

Chapter 3

The Dependency Reconstruction Algorithm

This chapter presents the dependency reconstruction algorithm. Commit-order timestamps (*tickets*) are assigned to transactions by the primary fragments. At the backup, two-phase commit ensures atomicity, while the timestamps are combined with two-phase locking to reconstruct and resolve data dependencies and to achieve consistency (mutual and local).

3.1 Tickets

As mentioned in Chapter 2, the primary fragments track their actions on behalf of a transaction in a redo log. As explained below, the dependency reconstruction algorithm requires that read sets of read-write transactions must also be recorded. The structure of redo log entries is described in Section 2.2.

When a transaction completes at the primary, one of the primary fragments initiates a local two-phase commit protocol. If the transaction has accessed data at the coordinating fragment, the coordinator is also a participant. Upon receipt of a *commit* message from the coordinator, a participant produces a local ticket number by atomically incrementing and fetching a counter. Intuitively, if a transaction gets a ticket t at fragment P_i, the transaction observed state $t-1$ of the fragment. If the transaction only read data at that fragment, the ticket of the transaction becomes the value of the counter plus one, but the counter is *not* incremented (since the state of the fragment did not change). For example, if the counter has the value 25, the committing transaction gets ticket 26. If the transaction wrote this fragment, the counter becomes 26; otherwise, it stays at 25.

Each fragment creates a commit entry in its redo log, with the following data:

field	meaning
`tid`	transaction identifier
`act`	action descriptor, in this case *commit*
`ticket`	local ticket number

After writing the commit entry in their logs, the participant fragments send an acknowledgement to the coordinating fragment and release the locks they held on behalf of the transaction.

From the viewpoint of the primary, transaction processing is similar to what it would be without a backup. Ticket generation is the only difference. Tickets are generated locally at each fragment and independently of any other fragment. There is no global ticket generation process, which avoids global bottlenecks. Each transaction identifier may be associated with several ticket numbers, one for each fragment at which actions of the transaction were executed.

Ticket generation is not an additional overhead. Most systems generate a monotonically increasing sequence of numbers — log sequence numbers (LSNs) — whenever they write something in the log. These numbers may be used by various components of the system. For example, the buffer manager may use them to ensure that a modified page gets written to disk only after the log entry for the last modification on the page has been written. Thus, there are already critical regions in the execution path of transactions, so adding an atomic increment of a counter is insignificant.

3.2 Dependencies at the Backup

The logs of primary fragments are propagated in parallel to their backup peers, and installing updates indiscriminately may violate transaction atomicity. Thus, before the updates of a transaction can be installed at a backup fragment, a *local* two-phase commit protocol is run among the backup fragments at which the transaction accesses data in order to ensure that all parts of the transaction have been received.

The two-phase commit protocol allows each fragment to proceed with the installation of a subset of the transactions it has received. If the updates of this subset of transactions are installed in strict ticket order, mutual consistency (Requirement 2.2) is guaranteed. Suppose we have two transactions T_x and T_y, such that $T_x \rightarrow T_y$ at the backup copy. Let z be a data item that caused this dependency. If actions are installed at the backup in local ticket order, the ticket number of T_x is smaller than that of T_y at the particular fragment, which implies that at the primary T_x got its ticket before T_y. When T_x got its ticket, it held a lock on z, which was not released until T_x committed. The lock was incompatible with the lock on z requested by T_y. Thus, $T_x \rightarrow T_y$ at the primary.

However, installing updates in ticket order does not guarantee local consistency, since dependencies on missing transactions cannot be detected. For example, suppose a transaction with local ticket 23 cannot be installed at backup fragment B_1 because two-phase commit detected that another part of it was not received at fragment B_2. Installing transactions with tickets higher than 23 at B_1 may violate the consistency of the database even if the installation is done in ticket order; some of these transactions may depend on the transaction with ticket 23 and should not commit before that transaction.

Furthermore, installing updates in ticket order may be inefficient. Suppose that T_1 and T_2 access disjoint data sets at some primary fragment and $\mathit{ticket}(T_1) = \mathit{ticket}(T_2) - 1$. At the backup, the writes for T_2 cannot be installed until those for T_1 are installed (as dictated by strict ticket order). Thus, T_2 must wait until T_1 *commits*, which may involve waiting for other fragments to agree, and then wait until the writes of T_1 are actually executed. This process is inefficient, especially if the fragments have a capacity for executing writes in parallel, e.g., have multiple disks. Even with one disk, efficient disk scheduling of the writes of T_1 and T_2 is not possible.

In order to preserve the consistency of the backup and to achieve parallelism, it is essential that backup fragments detect dependencies between transactions. If T_1 and T_2 write a common item, T_2 must wait for T_1 to commit and write. However, even if T_1 and T_2 write disjoint sets, there may still be a dependency. These dependencies can be detected only if read sets of transactions are also propagated to the backup. To illustrate, suppose that at some primary fragment transaction T_1 wrote item y and that T_2 read y and wrote z. There is a dependency $T_1 \rightarrow T_2$, so T_2 cannot commit at the backup unless T_1 does (local consistency, Requirement 2.3). The corresponding backup fragment cannot detect the dependency from the write sets ($\{y\}$ and $\{z\}$). If the read sets are sent, the dependency can be detected, and T_2 can be delayed until T_1 finishes.

Sending undo/redo logs (as opposed to redo logs) does not eliminate these problems. If undo logs are included, it is possible for backup fragments to install updates in local ticket order disregarding the state of other fragments. If it later turns out that some transaction does not commit, its updates can be undone. It might appear that it is possible to avoid these problems (especially cascading aborts) and their processing overhead by deferring commit decisions until disaster time and doing some extra processing then to abort those transactions that cannot commit. However, the commit decisions must still be made at some point and involve dependency detection. It is a poor idea to delay all commits until a disaster hits, mainly for two reasons.

- The undo logs for all transactions must be kept until the commit decision is reached, since potentially any transaction can be affected by a cascading abort. Determining that a transaction can no longer be affected by a cascading abort is equivalent to making a commit decision. The logs grow with time, and it may be impossible to keep them because of space limitations.

- Processing of new transactions at the backup after a disaster would be delayed until the commits complete, which may take a long time, since the cascading aborts mentioned above can lead to extensive searching and analysis of the logs.

Thus, undo logs still require that a commit protocol run as transactions are received, and it would be similar to the one we describe above for redo logging only.

3.3 Installing Actions at the Backup

Transactions should not wait for transactions they do *not* depend on, which happens if actions are done in *strict* ticket order. This section describes a mechanism that achieves these goals and proves its correctness. In what follows, the notations T_x and $T(s_x)$ are equivalent and both denote the transaction with ticket number s_x. The fragment is usually implied by the context.

The intuition behind our solution is to detect exactly those cases where waiting is necessary and to let all other cases take advantage of parallelism. This detection is achieved through locks on the data items accessed by the transactions, which are *granted to the transactions in ticket number order.* Write (exclusive) locks are requested for items that are to be updated, read (shared) locks are requested for other items in the read set. Additionally, a read lock on every table name accessed is requested so that the table is not deleted while accessing one of its records; if the table is created or destroyed, this lock must be exclusive.

As a concrete example, suppose that T_x has a smaller ticket number than T_y at one of the fragments. If they access a data item in conflicting modes, our mechanism ensures that the lock is granted to T_x, the transaction with the smaller ticket number. Transaction T_y cannot get the lock until T_x releases it, i.e., until T_x commits. If, on the other hand, there is no dependency between the two transactions, they do not ask for conflicting locks and can proceed in parallel.

When the redo logs for transaction T_x with ticket number s_x arrive at backup fragment B_j, transaction T_x goes through the following states.

`LOCKING` The transaction arrives and requests locks for all of the records and tables it accesses. It waits until all transactions with smaller ticket numbers have entered (or gone past) the `SUBSCRIBED` state (so that conflicts can be detected), and only then enters the `SUBSCRIBED` state.

`SUBSCRIBED` The transaction is waiting until it is granted the locks it requested. When these are granted, the transaction proceeds to the `PREPARED` state.

`PREPARED` An acknowledgement for the first phase of the two-phase commit protocol has been sent to the backup coordinating fragment.

`COMMITTED` The message for the second phase has been received for this transaction. All updates have been made public and all locks have been released.

When T_x arrives, B_j sets the state of T_x to `LOCKING` and starts requesting the locks required. Transaction T_x asks for a lock on data item z by inserting itself in a list of

```
$state(T) \leftarrow$ LOCKING
$t \leftarrow ticket(T)$ at this fragment
for every object $z$ accessed by $T$
    add lock request with $t$ to list for $z$
wait until $counter \geq t - 1$
if $T$ writes at this fragment then
    $counter \leftarrow counter + 1$
$state(T) \leftarrow$ SUBSCRIBED
for every object $z$ accessed by $T$
    wait until request by $T$ is at head of list for $z$
send acknowledgement to coordinating fragment
$state(T) \leftarrow$ PREPARED
wait until commit message for $T$ is received
for every object $z$ accessed by $T$
    if $T$ wrote $z$ then
        install new value for $z$
    remove lock request of $T$ from list for $z$
$state(T) \leftarrow$ COMMITTED
```

Figure 3.1: Pseudo-code for backup transaction processing

transactions asking for a lock on z; the list is sorted by ticket number. Each fragment has a counter that keeps track of the local ticket sequence, showing the ticket of the last transaction that became SUBSCRIBED at this fragment.

The locking procedure is summarized in Figure 3.1. After all of the locks for T_x have been requested, T_x waits for the counter to reach $s_x - 1$, which implies that T_x waits for all transactions with smaller ticket numbers to become SUBSCRIBED. Some or all of these transactions may be further ahead, e.g., PREPARED or COMMITTED. The important point is that they must have become at least SUBSCRIBED before T_x can do so. When this happens, T_x becomes SUBSCRIBED. If T_x writes data at this fragment, it increments the counter, which may trigger $T(s_x + 1)$ to become SUBSCRIBED, and so on. For example, if the current value of the counter is 25, then if transaction T_x with ticket 26 is LOCKING, it becomes SUBSCRIBED and increments the counter to 26, which may cause the transaction with ticket 27 to become SUBSCRIBED and so on. If T_x were read-only for this fragment, it would become SUBSCRIBED without incrementing the counter.

A transaction only waits for parts of other transactions that executed at the *same* fragment to become SUBSCRIBED. The parts of a transaction at different fragments proceed *independently* of each other, which also avoids deadlocks. The coordinating fragment ensures transaction atomicity by waiting for all parts of the transaction to become PREPARED before issuing a commit message and allowing any part to become COMMITTED. If one part is delayed in becoming PREPARED, the other parts must wait. In case of disaster, one or more parts may never become PREPARED. In this case, the transaction cannot commit, and the parts that did become PREPARED must be rolled back.

Waiting for transactions with smaller ticket numbers to become SUBSCRIBED is necessary if multiprocessing or multiprogramming is used at the backup. For example, two transactions may try to become SUBSCRIBED out of ticket order because of delays introduced by scheduling or because one of them had to ask for more locks than the other.

When SUBSCRIBED, T_x waits until all of its lock requests reach the head of their corresponding lists. A read request is also considered at the head of a list if all requests with smaller ticket numbers are for read locks. When this condition is met, T_x becomes PREPARED and informs the coordinating fragment. After commit, all of the requests of T_x are removed from the corresponding lists.

When a failure occurs at the primary, the backup is informed it will take over primary processing. The backup tries to commit all of the transactions that can commit and aborts the ones that cannot. It then becomes the primary.

Atomicity (Requirement 2.1) is enforced by the local two-phase commit protocol that is executed by the backup fragments.

Since locks are granted in ticket order at the fragments, updates inducing dependencies are installed in ticket order. As discussed at the beginning of Section 3.2, mutual consistency (Requirement 2.2) holds.

Local consistency (Requirement 2.3) also holds. Suppose $T_a \rightarrow T_b \rightarrow T_c \rightarrow \cdots \rightarrow T_r$ and T_a cannot commit at the backup (recall that the dependencies are non R-W). Further, assume that $T_a \rightarrow T_b$ occurs at fragment B_1, $T_b \rightarrow T_c$ at B_2, and so on (these fragments are not necessarily different). Since the dependency is non R-W, the ticket number of T_b is *higher* than that of T_a. If the part of T_a at B_1 (which causes the dependency $T_a \rightarrow T_b$) is missing, the ticket counter at B_1 will not be incremented by T_a and no transaction with higher ticket will be able to become `SUBSCRIBED` at that fragment, let alone `COMMITTED`. If the part of T_a at B_1 is received but T_a cannot commit (e.g., another part of it may be missing), transaction T_a holds a write lock on the data item that causes the dependency with T_b. Thus, T_b cannot obtain that lock and will not commit. At fragment B_2, transaction T_b will in turn hold a write lock on the data item that caused the dependency $T_b \rightarrow T_c$, which prevents T_c from committing, and so on for all transactions on the dependency path. Thus, local consistency holds.

Minimum divergence (Requirement 2.4) also holds. We have assumed that the order of messages between primary and backup is preserved. Consider a transaction T_x whose parts have been properly received at the backup fragments and that does not depend on any transactions that cannot commit at the backup. Since the order of messages is preserved, all transactions with smaller ticket numbers at those fragments have arrived, so that all transactions with smaller tickets can request their locks. Since there is no gap in the ticket sequence, all of these transactions eventually become `SUBSCRIBED` at these fragments. Some of these transactions may (in case of disaster) never become `COMMITTED` or go beyond `SUBSCRIBED` perhaps because they would violate atomicity or because of the cascading aborts effect, if they depend on other transactions that cannot commit. The important point is that these transactions *can* become `SUBSCRIBED`, which allows T_x to do so also. Transactions on which T_x depends commit and release their locks eventually, so T_x will be able to obtain all of its locks, install its changes, and commit.

3.4 Initialization of the Backup Database

We must now consider the system with one copy in the primary mode and one in the recovering mode, which is the case at system initialization and when a previously failed copy recovers. The mechanism we use is similar to a fuzzy dump [38], which commercial systems use for media recovery (e.g., Reference [8]). However, there are some differences. A conventional fuzzy dump gives a dump of the database and a log of the actions performed while the dump was in progress. In order to restore the database, the dump is installed and the log is replayed. No transaction processing takes place during restoration. The correctness criterion for the restoration is defined clearly: the database must be brought to the consistent state existing at the time of the last committed transaction in the log. In our application, there are multiple logs and transaction processing cannot be suspended during recovery: dump generation, dump installation and log playback must all occur *simultaneously*, which complicates both the implementation and the correctness proof.

The basic idea is for the primary to scan the entire database and transmit it to the recovering copy along with the changes that occur while this scan is taking place and may therefore not be reflected in the scan. These changes are essentially a redo log and are transmitted over the communication lines used for normal operation. If the scan data is too big to be transmitted over the same communication lines, it can be written on tape at the primary and carried to the backup. The order in which scan messages are received at the recovering copy is irrelevant for our method, so that some scan messages may be sent over the communication lines while others are written on tape. Our scheme allows the use of multiple tapes to expedite the process.

3.4.1 Scanning the Primary

Each primary fragment P_i sends its data to its backup peer B_i. When copying starts, P_i records the current ticket number s_x and sends it to B_i. B_i sets its ticket number to s_x, creates a legal but empty database and starts accepting redo logs with ticket numbers higher than s_x; redo logs with lower ticket numbers are simply ignored because their effects will be reflected in the scan copy. B_i remains in the recovery mode until initialization completes. If the primary fails while the backup is in recovery mode, nothing can be done: the backup is useless.

Fragment P_i starts scanning its portion of the database. At the same time, normal

```
for every table
    get a table read lock
    send a scan message for table creation
    for every record
        get a read lock on the record
        send a scan message with the image of the record
        release the lock held on the record
    release the lock held on the table
```

Figure 3.2: Pseudo-code for the scan process

processing continues and redo logs are sent to B_i. The scanning of the database is like a long-lived transaction that reads the entire database. For each object scanned (table or record), a scan message is sent to B_i with enough information for it to create the object. The scan process is described in Figure 3.2.

Locks are held only while an object is scanned. Records are locked one at a time; groups of records do not have to be locked together. When the scan process tries to lock an object that is already exclusively locked by a transaction, the scan process does not block: it reads the before image of the record. Tables or records created by transactions with ticket numbers greater than s_x do not have to be scanned, since they will be transmitted in the redo log of the transaction that created them. However, it is harmless if they are transmitted by the scan process too. The same holds for objects that have been updated after time s_x; it does not matter whether we send the object value that existed at time s_x or at some later time.

3.4.2 Backup Processing

The backup fragment receives two types of data: messages from the scan process and normal redo logs from transactions. The backup fragment processes both types of messages, using the simple rule of always trying to keep the most recent copy for every object. In particular, when a redo log arrives (with ticket number greater than s_x), it is processed as usual, except for the following steps.

- If a record update action is to be performed, but the record does not exist yet, the update is treated as an insert. (This action assumes that the update log entries contain the full after image of the modified record.)

- If a record (or table) insert arrives, but the record (or table) already exists, the values in the insert replace the existing ones. That is, the end result should be as if the record (or table) did not exist and the insert were normally executed.

- Deletes do not actually delete the record (or table). They simply mark it as deleted, but it is still reachable through the primary key index. If a delete arrives and the object does not exist, a dummy record is inserted with the record key and a flag indicating it is deleted.

When scan messages arrive at B_i, they are treated as insertions of the record or table. However, if the object already exists (even in deleted form), the scan message is ignored because the scan message may contain obsolete data (i.e., data that have been superseded by a later version). Even if the scan message contains a later version than the one already existing, no problem arises because the later version will be installed when the redo log for the corresponding transaction arrives.

The motivation for the above rules is that during the initialization phase, there will be two types of objects, those accessed by transactions and those not accessed at all. If the object is not accessed, we want the scan to give us the image of this object as it exists in the primary fragment. However, if the object is modified by a transaction, we really do not need the scan for the object since the normal processing will deliver the new image. Our rules ensure that a useless scan is harmless. For example, consider a record z that exists at time s_x in P_i. The scan starts and sends the image of z. Some time later a transaction deletes z. If the delete arrives at B_i before the scan, we could end up with a copy of z that should not exist. But since the delete creates a dummy record, the scan message will find it, and the scan copy will be discarded. After initialization, the deleted objects can be removed.

3.4.3 Correctness

There is a subtle point regarding the time at which we may resume normal processing at the backup, i.e., finish the recovery mode. It is not sufficient to wait until the scan process finishes, say at time s_y. Suppose that T_i writes data items a and b, which satisfy some consistency constraint (e.g., $a + b = const$). First, the scan process holds a read lock on a and transmits its before image (with respect to T_i). Then T_i gets write locks on both a and b, updates them, commits and releases the locks. The scanner gets a read lock on b, transmits its after image, and finishes (say b was the last unscanned object). Suppose that the scan messages for a and b arrive at the backup ahead of the redo log message for T_i, and that the T_i message never makes it because of a failure. If we let the backup copy resume normal processing at this point, we end up with an inconsistent copy — we are left with the before image for a and the after image for b.

The following rule avoids this type of problem. Suppose that the scan process finishes at the primary fragment P_i when the ticket number is s_y. It is safe to resume normal processing at B_i after all transactions through $T(s_y)$ have committed at B_i, and all of the scan messages sent by P_i have been processed at B_i. The backup can resume normal processing once all of its fragments are ready for normal processing.

The above two conditions are sufficient for correctness. Suppose the scan starts at primary fragment P_i at time s_x (i.e., after transaction $T(s_x)$ ran), and ends at time s_y. Let Z_0 be the state of P_i at time s_x and let T_{scan} be the set of transactions that committed at the primary during the scan.

At B_i, we start with an empty database and install a set of transactions T_{back} that includes *all* transactions in T_{scan}. The resulting schedule is SCH_{back}. Concurrently, we process all scan messages and arrive at a database state Z_α. T_{back} may contain more transactions than those in T_{scan}, since transactions that committed after time s_y at P_i can arrive before the last scan message.

We can show that Z_α is identical to Z_β, the state resulting from running SCH_{back} on Z_0, by considering each object z in the database.

Case 1 Object z was not modified by any transaction in T_{back}. Since $T_{scan} \subseteq T_{back}$, z was not modified by any transaction at the primary during the scan. Hence, the scan message for z contains the value of z in Z_0, i.e., $Z_0(z)$. This value will be installed, so $Z_\alpha(z) = Z_0(z)$. This value is the value of $Z_\beta(z)$.

Case 2 Object z was modified by some transaction in T_{back}. Let T_j be the last transaction in SCH_{back} to have modified z, writing value z_j. That is, $Z_\beta(z) = z_j$. Next, consider when z_j is installed by T_j at B_i. If the scan message for z arrives after this time, it is ignored. If it arrives before this time, T_j overwrites z. In either case, $Z_\alpha(z) = z_j$. Thus, $Z_\alpha(z) = Z_\beta(z)$.

We have shown that after SCH_{back} is run, the state of B_i is as if we had started with the initial state Z_0. Given the correctness of normal processing, the subsequent states of B_i will also have this property. So after the two conditions are satisfied, our implicit assumption that the initial states of the primary and the backup are identical holds (see Section 2.4).

3.5 Discussion

Our algorithm for maintaining and initializing a remote backup of a database is relatively straightforward and can be implemented using well known concepts and techniques, such as locking and logging. Chapter 5 examines its performance and compares it with other backup schemes.

Database consistency is preserved as shown in Section 3.3. The overhead at the primary is minimal: logs are usually maintained for other purposes, so the only extra processing is incrementing the ticket counter. The mechanism also scales: there is no component that must process all transactions in some way.

There is little we can prove regarding parallelism, but the proposed mechanism is not a processing bottleneck at the backup. Transactions are received by the backup fragments in parallel. At each fragment, locks are requested concurrently by multiple transactions. If the transactions with consecutive ticket numbers at one fragment access disjoint data sets, they can acquire their locks at that fragment in parallel. The only part that is executed serially is incrementing the counter, which is relatively fast and it is *not* a global bottleneck, since the increment at one fragment is independent of the increment at another fragment even for the same transaction. On the other hand, if the transactions access common data items at a fragment, a certain amount of parallelism is lost, because they will acquire their locks sequentially. However, at the primary, these conflicting transactions also executed sequentially, so the backup introduces no new delays.

In the remote event that the counter increment turns out to be a bottleneck, there is a workaround. One can divide the data residing in a fragment into *chunks* and apply our proposed solution generating ticket numbers per chunk instead of per fragment. This workaround gains parallelism by reducing the critical sections: there is less contention for getting the ticket numbers at the primary chunks, less contention for changing state (from `LOCKING` to `SUBSCRIBED`) at the backup chunks, etc.

We assume that two-phase locking is used for concurrency control at the primary. Two-phase locking is conceptually simple and has been studied extensively, so adopting it simplifies the presentation and helps concentrate on the novel issues. However, two-phase locking may not be the method of choice for some systems [35].

The dependency reconstruction algorithm can also be applied to systems using other concurrency control mechanisms. The only requirement is that the concurrency control algorithm assign tickets to transactions at each fragment so that they have the following property: if at the primary two *logical* actions A_1 and A_2, executed by transactions T_x and T_y respectively, induce a dependency $T_x \rightarrow T_y$, then $\mathit{ticket}(T_x) < \mathit{ticket}(T_y)$. Our proofs are independent of the details of the particular concurrency control algorithm.

The above requirement implies that the schedule of logical actions must be serializable. For every concurrency control method that generates serializable schedules there is a way to determine a serial order because, for example, serializable schedules can be topologically sorted [4]. In two-phase locking, a serial order can be obtained easily by incrementing a counter before releasing locks at commit time. In timestamp ordering, the timestamps themselves can be used as tickets.

In other concurrency control methods it may be more difficult to determine (on-line) an equivalent serial order for transactions and use that order as ticket. In such cases, one can use multiple tickets for a transaction (each for a set of data accessed by a transaction). These tickets have a limited scope, i.e., they are used only at the backup to determine the order of conflicting accesses to the set of data items for which the ticket was issued at the primary. The particular way to assign such tickets depends on the details of the concurrency control method.

An optimization can be made in the scanning process. Some of the messages sent by the scanning process are ignored at the backup. We can decrease the number of such messages if we track what has been scanned at P_i. For simplicity, assume that each record has a scan bit that indicates if it has been scanned. (This bit might actually be implemented with a cursor showing how far the scan process has progressed.) As each record is scanned, its bit is set. If a transaction modifies a record that has not been scanned, the after image for that record need not be sent since the scan process will send the after image of the record. However, the redo log entry containing the ticket must be sent, so that the ticket sequence will not be broken.

The recovery method suggested in Section 3.4 assumes that the primary database is lost during a disaster. If this is not the case, it may be possible to bring the failed primary up-to-date by sending it the redo logs for the transactions it missed instead of a copy of the entire database. The recovering primary also has to undo transactions that did not commit at the backup before redoing the missed transactions.

The use of redo logs for recovery may or may not be feasible, depending on the time it takes for the failed copy to be repaired. If this time is long, the logs will grow too big. The relative performance of the two methods also depends on the size of the logs. Our method is general and works when the primary loses its copy entirely without excluding the use of the other method. For example, one could combine the two strategies as follows: when a failure of the primary is detected, the backup takes over transaction processing and tries to keep the logs for as long as it can in case the primary recovers and still has its copy.

When its capacity overflows, it starts discarding the logs; when the failed copy recovers, the method proposed in Section 3.4 is used.

Finally, the proofs presented in the previous sections except for the minimum divergence proof can be shown to hold even if log messages are received at the backup out of order. Thus, our methods apply for connections between primary and backup that are not virtual circuits (e.g., datagrams).

Chapter 4

The Epoch Algorithm

The epoch algorithm takes a different approach than the dependency reconstruction algorithm. It installs transactions in batches at the backup rather than individually, which is more efficient. Read sets are not propagated to the backup, which leads to a simpler consistency criterion that is slightly stronger than the one mentioned in Chapter 2. Only the weaker criterion is actually necessary, but the epoch algorithm guarantees the stronger one. In what follows, $W(T, d)$ denotes the write at the backup of data item d by transaction T. The atomicity requirement is the same as in Chapter 2.

Requirement 4.1 (Atomicity) *If $W(T_x, d)$ appears in the backup schedule, then all of the write actions of T_x must appear in the backup schedule.*

Requirement 4.2 (Consistency) *Consider two transactions T_i and T_j such that $T_i \rightarrow T_j$ at the primary. Transaction T_j may be installed at the backup only if T_i is also installed (local consistency: dependencies are preserved). Furthermore, if they both write data item d, $W(T_i, d)$ must occur before $W(T_j, d)$ at the backup (mutual consistency: the direction of dependencies is preserved).*

4.1 Overview of the Epoch Algorithm

The general idea is that, periodically, special markers are written in the logs by the primary fragments. These markers delimit groups of transactions ("epochs") that can be committed safely by the backup. The primary fragments must write these markers in their logs in some synchronized way. Each backup fragment waits until all backup fragments have received the corresponding portion of the transaction group, i.e., all backup fragments have seen the next delimiter. Then, each fragment starts installing from its log the changes for the transaction group. This installation phase is performed almost independently of other fragments.

B_i	B_j
$\bigcirc_n$	$write(T_1)$
$write(T_1)$	$P(T_1)$
$P(T_1)$	$CC(T_1)$
$CP(T_1)$	$\bigcirc_n$

Figure 4.1: Markers in the logs

In the log, each delimiter includes an integer that identifies the preceding epoch. We represent the delimiter as a small circle with the epoch number as a subscript, e.g., $\bigcirc_n$ is the delimiter at the end of epoch n. At the primary, each fragment i keeps track of the current epoch number in a local counter $E(i)$. One fragment is designated as the *master* and periodically makes a $\bigcirc_n$ entry in its log where n is the current epoch number, increments its epoch counter, and broadcasts an *end_epoch*(n) message to all fragments at the primary. All recipients make a $\bigcirc_n$ entry in their logs, increment their epoch counters, and send an acknowledgement to the master. The master receives acknowledgements from all other fragments before it repeats the above process.

As we have described it, transactions can straddle the *end_epoch* markers, as shown in the logs of Figure 4.1 (the last record received at the backup is at the bottom). If epoch n is committed at the backup, the updates of T_1 at B_j are installed. However, the updates of T_1 at B_i appear after the *end_epoch* marker and will not be installed, which violates atomicity.

There are two ways to avoid this situation. The first way is to let transaction processing proceed normally and place the delimiters more carefully with respect to the log entries for actions of transactions. The second way is to write the delimiters asynchronously, but to delay some actions of some transactions so that the log entries for these actions will be placed more carefully with respect to the delimiters. These two approaches give rise to two versions of the epoch algorithm, which are described in Sections 4.2 and 4.4, respectively.

In what follows, we specify the fragment where an action took place and a log entry was written by adding an extra argument to the log entry. For example, the notation $\bigcirc_n(P_i)$ denotes the event when P_i writes a $\bigcirc_n$ entry in its log. Similarly, $C(T, P_i)$ denotes the event when fragment P_i writes a commit entry for transaction T in its log.

The symbol "$\Rightarrow$" denotes "occurs before," i.e., $A \Rightarrow B$ means that event A occurred before B, in the sense used in Reference [29]. When using a relation $A \Rightarrow B$, we do not distinguish whether A and B are the actual events or the corresponding entries in the log; we assume the log preserves the relative order of events within the same fragment. Do not confuse the "$\Rightarrow$" symbol with the symbol "$\rightarrow$" used for transaction dependencies. Suppose a dependency $T_x \rightarrow T_y$ exists at fragment P_d between two transactions T_x and T_y. Transactions T_x and T_y were coordinated by fragments P_x and P_y. The following property relates "$\Rightarrow$" and "$\rightarrow$".

Property 4.1 *If $T_x \rightarrow T_y$ at P_d, then*

$$CC(T_x, P_x) \Rightarrow CP(T_x, P_d) \Rightarrow P(T_y, P_d) \Rightarrow CC(T_y, P_y).$$

Proof: We prove the property in the case when strict two-phase locking is used for concurrency control. Transaction T_x does not release its locks until it commits, and transaction T_y cannot commit before T_x releases its locks, because the two transactions access some common data. Thus, P_x writes the *commit* message for T_x in its log, then P_d (if different from P_x) writes the *commit* message for T_x, the locks are released, T_y runs to completion, P_d writes the *prepare* entry for T_y, and finally P_y writes the commit message for T_y. Thus, the property holds for strict two-phase locking. Other concurrency control mechanisms also satisfy this property, but we will not discuss them. □

4.2 The Single-Mark Algorithm

The first version of the epoch algorithm places circles in the log more carefully. Circles are generated as described in the previous section, but some additional processing rules are followed. When a participant fragment i writes a *prepare* entry in its log and sends a *participant_ready* message to the coordinator, the local epoch number $E(i)$ is included in the message. Similarly, the epoch number is included in the *commit* message sent by the coordinator to the participants of a transaction. When a message containing an epoch number n arrives at its destination j, it is checked against the local epoch counter. If $E(j) < n$, it is inferred that the master has broadcast an *end_epoch*$(n-1)$ message that has not yet arrived. Thus, the fragment acts as if it had received the *end_epoch*$(n-1)$ message and makes a $\bigcirc_{n-1}$ entry in its log, sets $E(j)$ to n, sends an acknowledgement to the master, and *then* processes the incoming message. If the *end_epoch*$(n-1)$ message is received later from the master (when $E(j) > n-1$), it is ignored. The idea of incrementing an epoch when a message with a larger epoch number is received is similar in principle to incrementing logical clocks [29].

The logs (including the circle entries) are propagated to the backup. As the logs arrive at a backup fragment, they are saved on stable storage. The backup fragment does not process them immediately. Instead, it waits until a $\bigcirc_n$ mark has been seen by all backup fragments in their logs, which can be achieved in various ways. For example, when a fragment receives a $\bigcirc_n$ mark, it broadcasts this fact to other fragments and waits until it receives similar messages from all other backup fragments. Alternatively, when a fragment sees a $\bigcirc_n$ in its log, it notifies a master at the backup. The local master collects such notifications from all fragments and then lets all backup fragments know that they can proceed with installing the logs for epoch n.

To install the transactions in epoch n, B_i examines the newly arrived log entries from $\bigcirc_{n-1}$ to $\bigcirc_n$. However, there can also be entries pending from previous epochs that need to be examined. These entries correspond to transactions that did not commit in previous epochs. Thus, at the end of epoch n, B_i examines *all* of the log records appearing before $\bigcirc_n$ corresponding to transactions that have not been installed at B_i. The following rules are used to decide which new transactions will commit as part of epoch n.

- For a transaction T, if $C(T) \Rightarrow \bigcirc_n$ in the log, T is committed.
- If a transaction T does not fall in the above category, but $P(T) \Rightarrow \bigcirc_n$ in the log of B_i ($P(T)$ could possibly be in some *previous* epoch), deciding whether to commit T depends on whether P_i was the coordinator for T at the primary. (Recall from

Chapter 2 that the coordinator is included with every $P(T)$ log entry.) If P_i was the coordinator, T does not commit at the backup during this epoch. If some other fragment P_j was the coordinator at the primary, a message is sent to B_j requesting its decision regarding T. (B_j will reach a decision using these rules, i.e., if B_j finds $CC(T) \Rightarrow \bigcirc_n$, it commits T.) If B_j says T committed, B_i also commits T; otherwise T is left pending.

- Transactions for which none of the above rules applies do not commit during this epoch.

After having made the commit decisions, B_i reexamines its log. Again, it starts with the oldest pending entry (which may occur before $\bigcirc_{n-1}$) and checks the entries in the order in which they appear in the log. If an entry belongs to a transaction for which a commit decision has been reached, the corresponding change is installed in the database and the log entry is discarded. If no commit decision has been made for this transaction, the entry is skipped and will be examined again during the next epoch.

During the first scan of the log the only information from previous epochs that is actually necessary is for which transactions a $P(T)$ entry without a matching $C(T)$ entry has been seen. If this information is maintained across epochs and updated accordingly as *commit* and *prepare* messages are encountered in the log, the first scan can ignore pending entries from previous epochs and start from $\bigcirc_{n-1}$. It is still necessary for the second scan (installing the updates) to examine all pending entries.

4.3 Why the Epoch Algorithm Works

The following lemmas help prove that the epoch algorithm satisfies the correctness requirements stated in the beginning of the chapter.

Lemma 4.3.1 *If $C(T) \Rightarrow \bigcirc_n$ is in the log of a fragment P_i, then $CC(T) \Rightarrow \bigcirc_n$ is in the log of the coordinator P_c of T.*

Proof: If $P_i = P_c$, the lemma is trivially satisfied. Suppose that $P_i \neq P_c$, that $CP(T) \Rightarrow \bigcirc_n$ is in the log of P_i, and that $\bigcirc_n \Rightarrow CC(T)$ is in the log of P_c. The commit message from P_c to P_i includes the current coordinator epoch $n+1$. Upon receipt of this message, P_i writes $\bigcirc_n$ if it has not already done so. Thus, $\bigcirc_n \Rightarrow CP(T)$, which is a contradiction. □

Lemma 4.3.2 *If $CC(T) \Rightarrow \bigcirc_n$ is in the log of the coordinator for T, then $P(T) \Rightarrow \bigcirc_n$ is in the logs of the participants.*

Proof: Suppose $\bigcirc_n \Rightarrow P(T)$ at some participant. When the coordinator received the acknowledgement (along with the epoch) from that participant, it bumped its epoch (if necessary) and then wrote the $CC(T)$ entry. In either case, $\bigcirc_n \Rightarrow CC(T)$, which is a contradiction. □

Property 4.2 *The single-mark epoch algorithm satisfies the atomicity requirement.*

Proof: Suppose the changes of a transaction are installed by a backup fragment B_i after the logs for epoch n are received. If $C(T) \Rightarrow \bigcirc_n$ is in the log of B_i and the transaction was coordinated by P_c at the primary, by Lemma 4.3.1 $CC(T) \Rightarrow \bigcirc_n$ is in the log of B_c. If B_i does not encounter a $C(T)$ entry before $\bigcirc_n$, it must have committed because the coordinator told it to do so, which implies that in the log of the coordinator $CC(T) \Rightarrow \bigcirc_n$. Thus, in any case, $CC(T) \Rightarrow \bigcirc_n$ is in the coordinator's log. According to Lemma 4.3.2, $P(T) \Rightarrow \bigcirc_n$ is in the logs of *all* participants. The participants for which $CP(T) \Rightarrow \bigcirc_n$ will commit T anyway. The rest of the participants will ask B_c and will be informed that T can commit. Thus, if the changes of T are installed by one fragment, they are installed by all participating fragments. □

Property 4.3 *The single-mark epoch algorithm satisfies the consistency requirement.*

Proof: We prove the first part of the consistency requirement by showing that if $T_x \rightarrow T_y$ at the primary and T_y is installed at the backup during epoch n, T_x is also installed during the same epoch or an earlier one. Suppose the dependency $T_x \rightarrow T_y$ is induced by conflicting accesses to a data item d at a fragment P_d. By Property 4.1, $C(T_x, P_d) \Rightarrow P(T_y, P_d)$. Since T_y committed at the backup during epoch n, $P(T_y, P_d) \Rightarrow \bigcirc_n(P_d)$, which implies that $C(T_x, P_d) \Rightarrow \bigcirc_n(P_d)$. Thus, T_x must commit during epoch n or earlier (see Lemmas 4.3.1 and 4.3.2). For the second part of consistency: suppose $T_x \rightarrow T_y$ and they both write data item d. As we have shown in the first part, if $T_x \rightarrow T_y$ at the primary, T_x commits at the same epoch as T_y or at an earlier one. If T_x is installed in an earlier epoch, it writes d before T_y does, i.e., $W(T_x, d) \Rightarrow W(T_y, d)$. If they are both installed during the same epoch, the writes are executed in the order in which they appear in the log, which is the order in which they were executed at the primary. Since $T_x \rightarrow T_y$ at the primary, the order must be $W(T_x, d) \Rightarrow W(T_y, d)$. □

4.4 The Double-Mark Epoch Algorithm

In the single-mark algorithm, *participant_ready* and *commit* messages must include the current epoch number. The overhead, in terms of extra bits transmitted, is minimal, but the commit protocol does have to be modified to incorporate these, which might be a problem in adding the epoch algorithm to an existing system. The double-mark algorithm does not require any such modifications to the primary system; it positions transactions more carefully in the log with respect to delimiters.

At the primary, a master periodically writes a $\bigcirc_n$ entry in its log ($E(master) = n$), sets $E(master)$ to $n+1$, and sends an *end_epoch*(n) message to all fragments. Recipients make a $\bigcirc_n$ entry in their logs, stop committing new transactions, and send an acknowledgement to the master. Commit processing does *not* cease entirely. Transactions can still be processed; only *new* commit decisions by coordinators cannot be made, i.e., after writing the $\bigcirc_n$ entry in its log, a fragment cannot write a $CC(T)$ entry for a transaction T for which it is the coordinator. It can continue to receive and process *prepare* and *commit* messages for transactions for which it is not the coordinator. Except for the master, fragments do not need to remember the current epoch in the double-mark version.

When the master collects *all* acknowledgements, it starts a similar second round: it writes a $\diamond_n$ entry in its log (the counter is not incremented in this round) and sends a *close_epoch*(n) message to all fragments. The recipients make a $\diamond_n$ entry in their logs, send another acknowledgement to the master, and resume normal processing, i.e., new commit decisions can now be made. The master cannot initiate a new epoch termination phase (i.e., write a new $\bigcirc_{n+1}$ entry in its log) until all second-round acknowledgements have been received.

The logs (including $\bigcirc$ and $\diamond$ entries) are propagated to the backup and stored on stable storage. A backup fragment does not process the log entries of epoch n until all backup fragments have seen $\diamond_n$ in the logs they receive. Then each fragment B_i examines all of the log entries before $\diamond_n$, including entries pending from previous epochs,[1] to decide which transactions can commit after this epoch according to the following rules.

- If $C(T) \Rightarrow \bigcirc_n$ is in the log, a decision to commit T is made.
- If a transaction T does not fall in the above category, but $P(T) \Rightarrow \diamond_n$ is in the log of B_i ($P(T)$ could possibly be in some *previous* epoch), the decision whether to commit T depends on whether P_i was the coordinator for T at the primary. (Recall from Chapter 2 that the coordinator is included with every $P(T)$ log entry.) If P_i was the coordinator, T does not commit at the backup during this epoch. If some other fragment P_j was the coordinator at the primary, a message is sent to B_j requesting its decision regarding T. (B_j will reach a decision using these rules, i.e., if B_j finds $CC(T) \Rightarrow \bigcirc_n$, it commits T.) If B_j says T committed, B_i also commits T; otherwise T is left pending.
- If none of the above rules applies to a transaction T, the transaction does not commit during this epoch.

After the commit decisions have been made, the log entries up to $\diamond_n$ are examined and the actions of the committed transactions are installed as in the single-mark algorithm.

[1]The comment made in the single-mark algorithm about avoiding examination of entries from previous epochs when making commit decisions applies to the double-mark version as well.

In the correctness proofs of the double-mark version, we use the following property, which is a consequence of the master receiving all acknowledgements for *end_epoch*(n) before sending *close_epoch*(n) messages.

Property 4.4 $\forall i, j, n : \bigcirc_n(P_i) \Rightarrow \Diamond_n(P_j)$.

Lemma 4.4.1 *If* $C(T) \Rightarrow \bigcirc_n$ *is in the log of a fragment* P_i*, then* $CC(T) \Rightarrow \bigcirc_n$ *is in the log of the coordinator* P_c *of* T*.*

Proof: If $P_i = P_c$, the lemma is trivially satisfied. Now suppose that $P_i \neq P_c$. Then,

$$\begin{array}{ll} CC(T, P_c) \Rightarrow CP(T, P_i) & \text{(by two-phase commit)} \\ CP(T, P_i) \Rightarrow \bigcirc_n(P_i) & \text{(by hypothesis)} \\ \bigcirc_n(P_i) \Rightarrow \Diamond_n(P_c) & \text{(by Property 4.4)} \end{array}$$

By transitivity, $CC(T, P_c) \Rightarrow \Diamond_n(P_c)$, and since no commit decisions are allowed for coordinators between the $\bigcirc$ and $\Diamond$ entries, $CC(T, P_c) \Rightarrow \bigcirc_n(P_c)$. $\square$

Lemma 4.4.2 *If* $CC(T) \Rightarrow \bigcirc_n$ *is in the log of the coordinator for* T*, then* $P(T) \Rightarrow \Diamond_n$ *is in the logs of the participants.*

Proof: Consider a participant fragment P_i. Then,

$$\begin{array}{ll} P(T, P_i) \Rightarrow CC(T, P_c) & \text{(by two-phase commit)} \\ CC(T, P_c) \Rightarrow \bigcirc_n(P_c) & \text{(by hypothesis)} \\ \bigcirc_n(P_c) \Rightarrow \Diamond_n(P_i) & \text{(by Property 4.4)} \end{array}$$

By transitivity, $P(T, P_i) \Rightarrow \Diamond_n(P_i)$. $\square$

Property 4.5 *The double-mark epoch algorithm satisfies the atomicity requirement.*

Proof: Suppose the changes of a transaction are installed by a backup fragment B_i after the logs for epoch n are received. If $C(T) \Rightarrow \bigcirc_n$ is in the log of B_i and the transaction was coordinated by P_c at the primary, by Lemma 4.4.1, $CC(T) \Rightarrow \bigcirc_n$ is in the log of fragment B_c. If B_i does not encounter a $C(T)$ entry before $\bigcirc_n$, it must have committed because the coordinator told it to do so, which implies that $CC(T) \Rightarrow \bigcirc_n$ is in the log of the coordinator. Thus, in any case, in the coordinator's log holds $CC(T) \Rightarrow \bigcirc_n$. According to Lemma 4.4.2, $P(T) \Rightarrow \Diamond_n$ is in the logs of *all* participants. The participants for which

$CP(T) \Rightarrow \bigcirc_n$ will commit T anyway. The rest of the participants will ask B_c and will be informed that T can commit. Thus, if the changes of T are installed by one fragment, they are installed by all participating fragments. □

Property 4.6 *The double-mark epoch algorithm satisfies the consistency requirement.*

Proof: We prove the first part of the consistency requirement by showing that if $T_x \rightarrow T_y$ at the primary and T_y is installed at the backup during epoch n, T_x is also installed during the same epoch or an earlier one. Suppose that at the primary the coordinators for T_x and T_y are P_x and P_y respectively. Since $T_x \rightarrow T_y$, by Property 4.1 $CC(T_x, P_x) \Rightarrow CC(T_y, P_y)$. Since T_y committed at the backup, we infer from our processing rules that

$$\begin{array}{ll} CC(T_y, P_y) \Rightarrow \bigcirc_n(P_y) & \\ \bigcirc_n(P_y) \Rightarrow \diamond_n(P_x) & \quad \text{(by Property 4.4)} \end{array}$$

By transitivity, $CC(T_x, P_x) \Rightarrow \diamond_n(P_x)$, and since no commit decisions are made by coordinators between $\bigcirc$ and $\diamond$ entries, $CC(T_x, P_x) \Rightarrow \bigcirc_n(P_x)$, which implies that transaction T_x must commit during epoch n or earlier. The proof for the second part of consistency is identical to the proof for the single-mark version. □

4.5 Discussion

The epoch algorithms are scalable: no component processes all transactions. Each fragment processes only those transactions that access the data it holds.

The size of the epoch counters could be a problem; how big should they be? If only one epoch can be pending at any time, a fragment only needs to distinguish between its epoch and the epoch of a fragment that is possibly one epoch ahead or behind. Thus, a counter with 3 values is sufficient. In general, if the epochs of two fragments can differ by at most k, the epoch counter must accommodate $2k + 1$ values.

We assumed that the backup fragments start the first scan of the logs after they have received the marker for the next epoch. However, the first scan can occur while the logs are being received. Thus, the local commit decisions will be available when the next delimiter is seen and installation can begin immediately. Commit decisions that require interaction with another fragment must still be deferred until the epoch has been received.

The protocols have a low overhead and their cost is amortized over an entire epoch. There are three factors that contribute to the overhead: terminating an epoch at the primary, ensuring reception of an epoch at all backup fragments, and resolving the fate of transactions for which a $P(T)$ entry without a matching $C(T)$ has been seen. For the first two factors, the number of messages is proportional to the number of fragments at each copy. For the third factor, the average number of transactions for which a fragment cannot make a decision by itself can be estimated as follows for the single-mark version. The transactions for which a $P(T)$ was written before $\bigcirc_n$ and a $CP(T)$ after $\bigcirc_n$ are those transactions whose $P(T)$ entry falls within a time window t_c before the $\bigcirc_n$ mark, where t_c is the average delay necessary for a *participant_ready* message to reach the coordinator and the *commit* answer to return. Thus, the expected number of transactions for which information must be obtained from another fragment is $w_d t_c$, where w_d is the rate at which a fragment processes distributed transactions for which it is not the coordinator. If the global system processes distributed transactions at a rate w_g, there are n fragments at each copy and a global transaction accesses data at m fragments on the average, then $w_d = w_g(m-1)/n$. The number of messages that must be sent could be less than the number of transactions in doubt, since requests to the same fragment can be batched. Finally, these overheads are paid once per epoch. If an epoch contains a large number of transactions, the overhead per transaction is small.

The single-mark algorithm requires one less round of messages for writing delimiters at the primary at the end of each epoch. Also, the single-mark algorithm does not suspend commits. However, the transaction processing mechanism has to be modified to include the local epoch number in certain messages and to update the epoch accordingly when such messages are received. On the other hand, the double-mark algorithm may require fewer modifications to an existing system. The double-mark epoch termination protocol can be viewed as the commit phase of a special transaction with a null body. The *end_epoch*(n)

messages correspond to *prepare* messages and the *close_epoch*(n) messages correspond to *commit* messages. The only system interaction in the double-mark protocol is in suspending coordinator commits, which may be easy to achieve by simply holding the semaphore. Depending on the system, holding a commit semaphore may disable all commits, not just coordinator commits, which may be acceptable if the time between the $\bigcirc_n$ and the $\diamond_n$ is short. If this delay is unacceptable, a new semaphore can be added.

The algorithms satisfy atomicity and consistency. However, they do not achieve minimum divergence. If a disaster occurs, the last epoch may not have been fully received by the backup fragments. The epoch algorithms will not install any of the transactions in the incomplete epochs, even though some of them could be installed. This problem can be addressed by running epochs more frequently (to limit the number of transactions per epoch) or by having another mechanism for dealing with incomplete epochs, e.g., individual transaction commit or a mechanism like the dependency reconstruction algorithm.

It is interesting to compare the epoch and the dependency reconstruction algorithms. The epoch algorithm has less overhead, but it does not achieve minimum divergence. The dependency reconstruction algorithm achieves minimum divergence, which implies that the *takeover* time — the time between the point when a disaster occurs at the primary and the point when the backup starts processing new transactions — is shorter. Our experience in implementing both algorithms suggests that the dependency reconstruction algorithm makes it easier to run the same software at both the primary and the backup. Running the same software on both reduces maintenance and takeover times.

Algorithms similar to the epoch algorithm have been used for obtaining snapshots [7] and checkpointing databases [42], but there are differences between those approaches and the epoch algorithm. The epoch algorithm is log based. Minimal modifications to an existing system are necessary and minimal overhead is imposed at the primary. The epoch snapshot is not consistent. Enough information is included to allow a consistent snapshot to be extracted from the propagated logs, but some work is still necessary at the backup to clean up $P(T)$ entries with no matching $C(T)$.

4.6 Another Application: Distributed Group Commit

Group commit (for a single fragment) [9, 19] is a technique that can be used to achieve efficient commit processing of transactions in computer systems with a large main memory, which can hold the entire database (or a significant fraction of it). When the end of a transaction is reached, its log entries are written into a log buffer that holds the tail of the master log. The locks held by the transaction are released, but the log buffer is not flushed immediately to disk (to avoid synchronous I/O). When the log buffer becomes full, it is flushed and the transactions it holds commit as a group. The updates made by these transactions are installed in the database after the group commit. Care must be taken to preserve the dependencies of transactions that are members of the same group and have made conflicting accesses.

Under the above scheme, transactions are permitted to read uncommitted data. This is not a problem, since a transaction T can only depend on transactions in the same group or previous groups, which are installed before or when T is.

It would be desirable to apply the same technique to distributed systems. However, in a multicomputer environment it is not possible for each individual fragment to flush its own log independently of other fragments, since that could violate transaction atomicity. A simple example illustrates: suppose transaction T completes at fragments P_1 and P_2, fragment P_1 flushes its log (and commits T in the database) while P_2 does not. If a failure occurs and the contents of P_2's log buffer (in volatile memory) are lost, transaction atomicity is violated.

The epoch algorithm can be used to achieve distributed group commit. One can think of the main memory as being the primary and of the disks as the backups. Transactions run in a main memory database and their logs are written into log buffers, but their changes are not propagated to the disk. Distributed transactions still use a two-phase commit protocol to achieve atomicity. $P(T)$ and $C(T)$ entries are made for all transactions that finish processing successfully and their locks are released, but the logs are not flushed.

Periodically, delimiters (e.g., circles) are written by all fragments in their logs. When the delimiter is written, the log buffer is written on disk, but the group commit does not take place until it is confirmed that all fragments have saved their log buffers. Then, each fragment starts to install the changes of the transactions in the disk copy of the database, in the same way the backup fragments did in Section 4.2.

There are several advantages of group commit. Local transactions that execute only at one fragment avoid synchronous I/O and release their resources as soon as they complete. Holding resources for a shorter time decreases contention and increases throughput. Furthermore, the cost of log I/O is amortized over many transactions. Distributed transactions must still pay for the agreement protocol to ensure atomicity, but this cost may actually be a little smaller, since the individual *prepare* and *commit* decisions do not need to be written to disk, and thus their responses can be sent immediately. Distributed transactions also benefit from amortizing the cost of log I/O over several transactions.

Chapter 5

Evaluation of the Algorithms

This chapter evaluates the performance of the epoch algorithm, the dependency reconstruction algorithm, and the 2-safe mechanism. We cannot say definitively which backup technique is best because the answer depends on the configuration, the workload, and the performance requirements of a particular system. We can, however, understand the basic tradeoffs between the algorithms.

5.1 The Testbed

Our experimental transaction processing system uses eight IBM RT-PC 6150 computers running Unix 4.3BSD. Four computers are the primary set and four are the backup. Each machine holds part of the corresponding database copy (primary or backup) on its local disk and communicates with the machines that hold the rest of the copy as well as with its remote peer over a 10Mb/sec Ethernet. The primary computers receive their workload from a DECstation 5000/125 over the network. Many of the design decisions presented below have been dictated by this environment.

We built a real system instead of a simulator for several reasons. First, a real system is a proof of concept for our algorithms. The confidence in the correctness of the algorithms is higher when the state of the primary and the backup databases can be compared and checked for consistency. Tricky details of the protocols are easy to miss with a simulator. Finally, we made fewer assumptions about the system parameters, since some of them were fixed by our environment.

Real transaction processing environments have more powerful processors and larger and faster I/O systems. Our system is a scaled-down version. The network delays from primary to backup computers may be unrealistically short, since the machines are attached to the same Ethernet. Real systems have longer delays, which we can simulate by introducing a

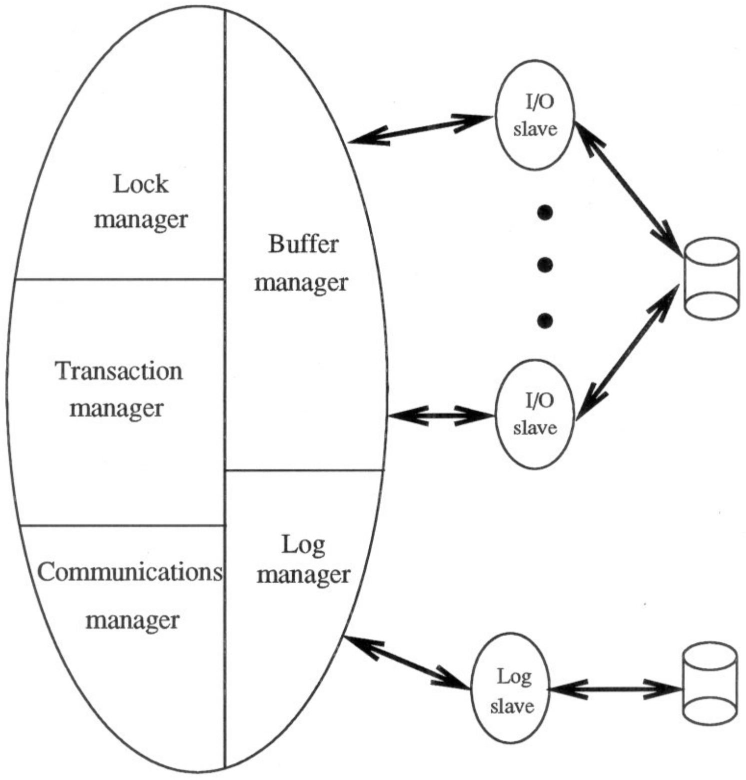


Figure 5.1: Structure of the testbed

range of artificial delays.

Certain high-level database functions, such as query processing and optimization, were not implemented in our system, since they were not relevant to our study. All access requests have the form read/update/insert/delete record-id and assume that other mechanisms have converted higher-level requests into this form. On the other hand, all-low level DBMS modules are fully implemented, including logging, locking, record and buffer management, etc., since these modules play an important role in backup algorithms.

Figure 5.1 shows a block diagram of the main parts of the testbed. The lack of shared memory and threads in 4.3BSD forced us to build a monolithic process that implements the transaction logic and does its own internal context switch when a transaction gets suspended on an I/O request. The lock manager, the log manager, and the buffer manager are also parts of this process. If these modules ran in separate address spaces, accesses to shared data (e.g., a lock request) would incur the cost of interprocess communication. Finally, the process handles the network communications with other local computers and with the remote peer.

Our recovery mechanisms need to determine if modified data is in volatile storage or it has been written to disk. Consequently, we use "raw" Unix I/O device to control the location of the data and do our own disk management. The problem with this approach is that I/O system calls block and all processing freezes for the duration of the call, so we could not overlap I/O with other processing.

To avoid this problem, we use slave I/O processes. When the central process issues an I/O request, it selects one of the slaves and passes the request (along with the data, if the request is a write) to that slave. The slave performs the actual blocking I/O operation and notifies the central process, passing back the page, if the request was a read. While the slave is blocked on the I/O system call, the central process can continue processing. Using slaves incurs the overhead of the interprocess communication, but this cost is taken into account when reporting CPU loads.

Records are always accessed through a record identifier unique within a table. Hashing maps records to pages and locates them on disk. The operations that can be performed on records are insert, delete, update and read. Insert and update requests also include the image of the record. Since we use small records (30 bytes), updates include the new image for the entire record. In a real system, records may be larger and updates would include only the modified fields. All record accesses are preceded by a request for the appropriate lock (shared or exclusive). Our lock manager also performs lock upgrades. Deadlocks are broken with timeouts. Finally, the system can create tables; these requests are executed serially.

The log manager uses a slave process to perform raw disk I/O without blocking the central process, which is analogous to the scheme used for accessing the database. Since the log is written to a raw device, it is flushed at page boundaries. If a certain interval elapses without enough logging activity to fill the tail page of the log, it is padded to the next page boundary and flushed. Later, when additional log data is appended and the tail page fills, it is rewritten.

We tested the system by writing the log to disk and verifying it. Real systems typically have a separate log disk because the log is written sequentially and sequential disk access is an order of magnitude faster than random access. Our RT's only have one disk, which makes log I/O compete with database I/O. In our experiments, we changed the log device to a dummy device (/dev/null). All of the processing associated with the log (system calls, etc.) is performed as if the data were written to a separate disk.

The buffer manager implements a simple LRU page replacement policy. There may be a limited number of active transactions at any time, and this number is a system parameter. We found that ten transaction slots provide enough multiprogramming to exploit the capacity of our machines fully.

We give a brief description of transaction processing for the 1-safe mechanism at the primary. A transaction is read from the load-generating machine over the network and it is assigned a slot. It performs its actions (setting the appropriate locks), but its changes are made to private copies of the data. When the transaction reaches the end of its execution, it writes log entries for its modifications, a *commit* entry (if the transaction is local) in the log, and waits until that portion of the log has been flushed. Then it writes its changes to the pages in the shared buffer pool, so that they can be seen by other transactions, and releases its locks. The changes may actually be written to disk later when the buffer manager flushes a page.

We use a chain model for distributed transactions. Assume that a distributed transaction T is executing at node A. When T completes its processing at A, it writes a *prepare* rather than a *commit* message in the local log, it holds onto its resources (locks, transaction slot, etc.) and sends a message to the next node, say B, to start processing on behalf of T. When T finishes at B, it moves to the next node and so on, until it reaches the last. The last node on the chain also acts as the coordinator. The commit/abort outcome of the transaction is propagated to the participants on the reverse path. The messages that trigger the execution of a distributed transaction at the next node also serve as messages for the agreement protocol: when a message is sent from A to B telling B to start executing T, the message is implicitly conveying information that all nodes in which T has executed so far are prepared to commit T. This information enables the last node on the chain to commit.

In the 2-safe mechanism, all updates of a transaction must be applied synchronously to both the primary and the backup. Thus, an agreement protocol is required; we use two-phase commit. Consider a transaction T that executes at only one primary P_i. When P_i finishes executing T, it writes a *prepare* message in its log but does not release any of the resources held by T. The logs are propagated to B_i, the backup peer of P_i. When the backup receives the logs, it extracts T's actions. The *prepare* entry in the log serves as both a delimiter for the actions of T and an indicator that the primary is prepared to commit T.

Transaction T has already run at the primary, so the backup performs only the writeback phase to externalize the modifications of T. The backup must also ensure that changes to the same data item by different transactions are applied locally in the same order as at the primary. To do so, the backup sets locks on the data items to be modified by T. The locks that have been obtained by the primary site would suffice to ensure the same sequencing at both sites, provided that these locks would not be released until after both the primary and the backup had installed the changes. This scheme would require additional synchronization between the primary and the backup and would cause the primary to hold locks longer.

After the locks are obtained, the backup writes log entries for the actions of T and a *commit* message in a local log. The backup then acts as the coordinator for T between P_i and itself. The backup waits until the log is flushed to disk and then sends a *commit* message about T's fate to the primary. Upon receipt of the *commit* message, the primary commits T, externalizes its changes, and releases its locks. The backup independently does likewise.

In the case of a 2-safe transaction that must access data at more than one primary site, the transaction finishes processing at one site and moves on to the next. As the backups receive the logs of their primary peers, they mirror their peer's processing. If a transaction

T moved from primary P_i to P_j, when backup B_i receives the logs for T, it writes a *prepare* message in its log and notifies B_j, which implies that all backups on the execution chain of T up to and including B_i are prepared. Assume that after P_j transaction T accesses data at P_k, and that P_k is the last computer on the execution chain of T at the primary. When B_k receives the logs from P_k, all primaries are prepared. When it receives the message from B_j, all backups are also prepared. Thus, all participants are prepared to commit T, so B_k can act as the global coordinator for T. In general, the backup peer of the transaction's last hop acts as the coordinator for all primaries and all backups.

We verified the correctness of the system by comparing the logs produced against the input transactions. We ran various workloads and verified that the primary and the backup databases were identical; we used workloads that created deadlocks and verified that the deadlocks were broken properly; we verified that the system acts correctly when transactions are aborted. Finally, we simulated disasters by killing the processes running at the primary and verified that the backup reached a consistent state and performed the takeover properly.

During the performance evaluation experiments, the primaries process all of their input and terminate, after ensuring that they have sent all of their log entries to the backup. The backups treat the termination of the primaries as a disaster and reach a consistent state, which is the same as the final state of the primaries, since all of the log entries have been received. At this point, the backups would start processing incoming transactions, but for the performance experiments they simply halt.

5.2 Experimental Results

One of the goals of our experiments was to compare the primary load against the backup load, in order to estimate the overhead of maintaining a remote backup. A second goal was to compare the behavior of 1-safe and 2-safe backup techniques under various conditions in order to determine the performance cost of 2-safe processing. One expects 2-safe processing to be more expensive, since it provides a higher level of reliability, but how *much* more expensive? Is the performance improvement with 1-safe large enough to merit the potential loss of a few transactions? A third goal was to study the performance impact of the epoch length on the epoch algorithm. We also compare the epoch and dependency reconstruction algorithms, and experiment with single versus multiple log streams.

Each computer in the base configuration of our system holds a 40 Mbyte database; 35 Mbytes hold data and 5 Mbytes are empty and provide overflow pages. The 1-Kbyte pages are half-full with 30-byte records; all records belong to the same table. The buffer cache is 4 Mbytes. Log entries are 16 bytes; inserts and updates are followed by the full image of the record. These base settings are varied in the experiments presented in subsequent sections.

Each primary processes 15,000 transactions. Runs typically take 20–25 minutes. Epoch lengths are 15 seconds. Each transaction accesses exactly 4 records, and the access pattern is uniform. The fraction of read-write transactions is 30%, 70% are read-only. Read-write transactions do at least one write action and the rest of their accesses are half reads and half writes. Write actions can be inserts, deletes, or updates with equal probability. Distributed transactions are a series of local transactions at more than one primary. Distributed transactions access data at a number of computers that is uniformly distributed between 2 and 4, the maximum number of primaries. In the base case, 28% of the transactions executed at each computer were distributed.[1] For this workload and running the epoch algorithm, the I/O and CPU utilization were 98% and 50%, respectively. Average response time for local read-write transactions was 700 msec and the throughput of each primary computer was 10.6 transactions per second. There are 4 primaries, so the total throughput is 43 transactions per second.

5.2.1 I/O Utilization

A backup performs only the fraction of the primary I/O that corresponds to writes. For the base settings, the I/O utilization at the backup is 29%. However, there is an important

[1]This fraction cannot be set directly: a computer can adjust only the number of distributed transactions it generates; however, it must also execute distributed transactions that have originated elsewhere. Thus, we count the number of distributed transactions during execution and determine their fraction at the end of a run.

difference in I/O load between the 2-safe and the epoch algorithm. Assuming input transactions are processed at the primary at a uniform rate, the I/O load at the backup is also uniform for the 2-safe case, e.g., 29%. In contrast, in the epoch algorithm the load comes in bursts: when an epoch is fully received at the backup, the I/O subsystem sees a large volume of I/O, which keeps the I/O subsystem at 100% utilization for 29% of the duration of an epoch. For the rest of the epoch, the I/O subsystem is idle.

The periodic nature of I/O in the epoch algorithm may have positive performance implications for systems with more sophisticated I/O subsystems. Long I/O queues can lead to better disk scheduling and increased I/O throughput [40, 43]. In both the 2-safe and the epoch algorithm, the spare I/O capacity at the backup can be used for useful processing (e.g., to answer read-only queries). However, the idle periods in the epoch algorithm obviate the need for synchronization between updates and reads, so that the read-only queries can run without locking between epochs when the database is consistent.

5.2.2 CPU Utilization

Figure 5.2 shows the CPU utilization[2] at the primary for the epoch algorithm (circles) and the 2-safe algorithm (squares) as a function of the ratio of read-only and read-write transactions. When there are few read-write transactions (10%), the CPU requirements are almost the same for both algorithms. As the fraction of read-write transactions increases, the CPU requirements increase faster for the 2-safe than for the epoch.

Both algorithms must do more logging and initiate more I/O for more write actions, which explains the trend shown in Figure 5.2. The 2-safe algorithm must also pay for the agreement protocol with the backup, which accounts for its faster increase in CPU demand.

In our system, transactions pay a relatively high CPU overhead when initiating I/O (see Section 5.1). Nonetheless, our base case may underestimate the CPU requirements of a real system's primary. Real systems perform tasks such as terminal handling, input validation, query processing, actual computation within queries, etc. To cover these costs,

[2] In our system, the central process polls the slave I/O and log processes even during idle periods. CPU utilization is the actual utilization minus the time spent polling.

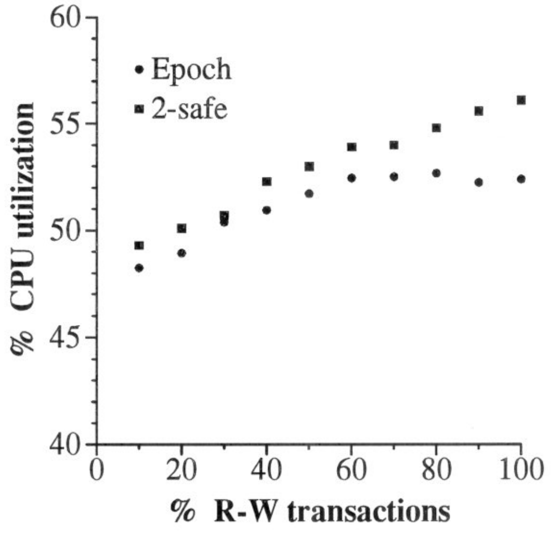
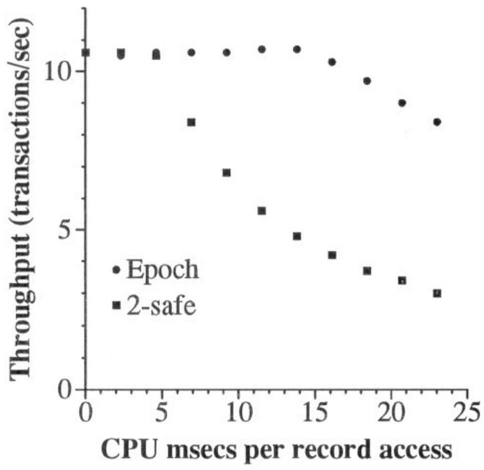


Figure 5.2: Primary CPU utilization



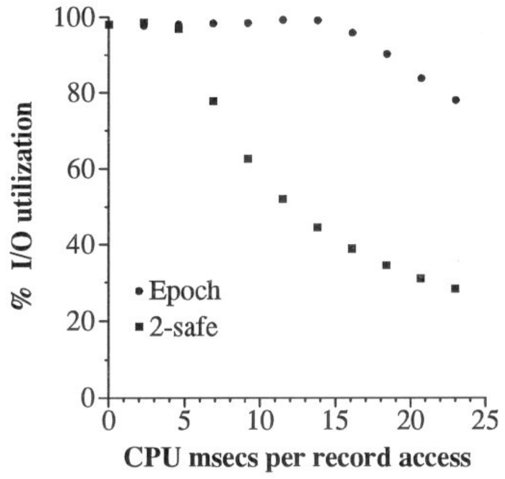
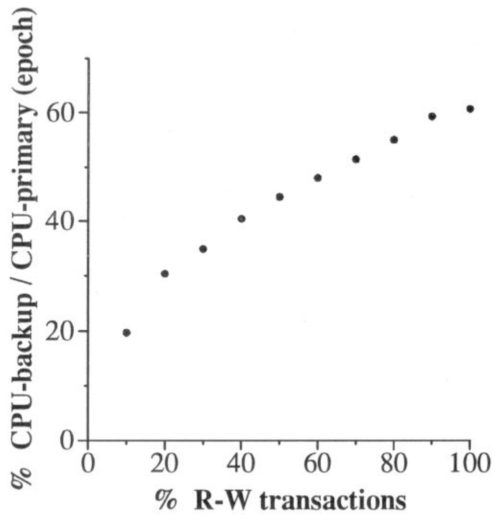


Figure 5.3: CPU-bound performance: throughput (left) and I/O utilization (right)

we added some CPU processing to each action executed by each transaction and observed the behavior of the system under each algorithm.

The left graph in Figure 5.3 shows the throughput of the epoch and the 2-safe algorithm as the CPU processing per action increases. The horizontal axis is the CPU processing added for each access to a record in milliseconds. Both algorithms suffer as the CPU load increases because the CPU becomes saturated, but the epoch algorithm sustains higher CPU loads before its performance drops. The graph on the right shows the corresponding



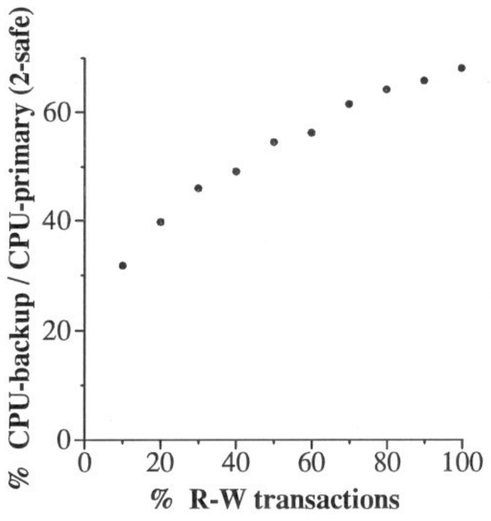


Figure 5.4: Backup CPU utilization relative to primary for epoch (left) and 2-safe (right)

I/O utilization in the two algorithms as the CPU load increases. As expected, throughput and I/O are similar, because unsaturated I/O is proportional to throughput.

Consider the backup's CPU requirements in the I/O-bound case. The backup only executes the write actions of read-write transactions in both algorithms, thus its CPU requirements are lower than those at the primary, as shown in Figure 5.4, which plots the fraction of the CPU cycles at the primary that is required at the backup for the epoch algorithm (left) and the 2-safe algorithm (right) against the read-write transaction ratio. These curves are not comparable since they depict fractions of *different* quantities, but both fractions rise with the fraction of read-write transactions since the backup has to replicate a rising fraction of the primary's work.

While our base case may underestimate the primary's CPU requirements, it does not underestimate the backup's CPU requirements because a real backup system would not perform any more tasks than our system. Thus, Figure 5.4 may overestimate the ratio.

Finally, Figure 5.5 shows that the CPU requirements of the epoch algorithm at the backup are always less than those of the 2-safe algorithm. Figure 5.5 plots the overhead $\frac{CPU_{2safe} - CPU_{epoch}}{CPU_{epoch}} \times 100$ versus the read-write transaction ratio, where CPU_z is the CPU usage of algorithm z at the backup.

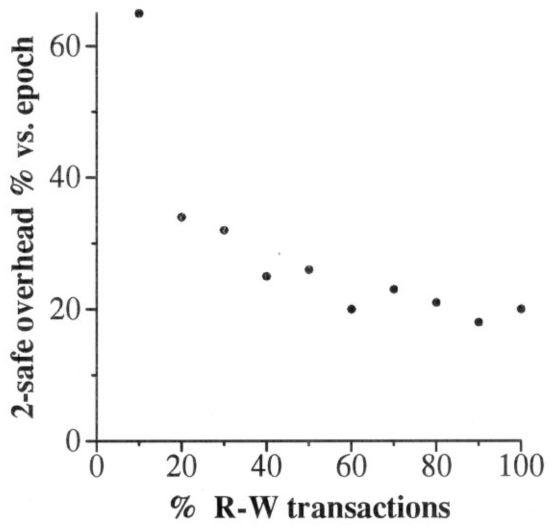
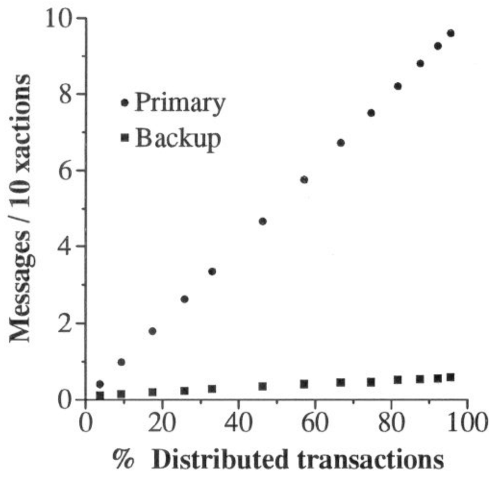


Figure 5.5: CPU overhead of 2-safe vs. epoch at backup

5.2.3 Network Utilization

We separate network traffic into two classes: one for messages that propagate the log information to the backup and one for messages for the agreement protocols and other forms of synchronization (e.g., the epoch termination messages). Only the number of messages in the second class varies with the algorithm, so we restrict our attention to those messages.

For a read-write transaction ratio of 30%, we varied the percentage of distributed transactions and observed the number of messages generated by primaries and backups. In this experiment, distributed transactions accessed data at two primaries. Figure 5.6 shows the average number of messages generated by a computer per 10 transactions at the primary and the backup for the epoch algorithm (left) and the 2-safe algorithm (right). As the percentage of distributed transactions increases, so do the messages generated by the primaries in both algorithms due to the agreement protocol. The epoch algorithm pays for $N - 1$ messages per epoch, where N is the number of primary computers (4). In these experiments the epoch duration was 15 seconds, so there were 3 messages every 15 seconds at the primary. This overhead is small to negligible; Section 5.2.7 discusses other aspects of epoch length selection.



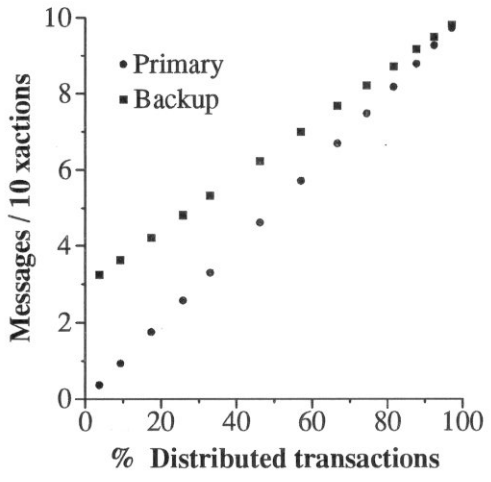


Figure 5.6: Messages generated by epoch (left) and 2-safe (right)

At the backup, the two algorithms exhibit different behavior. The 2-safe algorithm requires the same number of messages at the backup as at the primary to run the agreement protocol between backups, plus messages for agreement between primaries and backups (when a backup, acting as coordinator, sends a commit message to its primary peer). The latter messages are necessary even for read-write transactions that access data at only one primary. In contrast, in the epoch algorithm, the backups exchange a fixed number of messages to ensure that an epoch has been received by all, and this number is independent of the fraction of distributed transactions. There is also a small number of messages to negotiate the fate of transactions whose *prepare* and *commit* messages straddle the epoch marker. The number of these messages increases slowly with the fraction of distributed transactions since the probability of such straddling increases.

Figure 5.7 illustrates the savings in the number of messages achieved by the epoch algorithm over the 2-safe algorithm; it displays the ratio of the number of messages generated at the backup by the 2-safe over the number of messages generated by the epoch as a function of the distributed transaction percentage. For example, when 20% of the transactions are distributed, the 2-safe backup needs 20 times as many messages as the epoch backup. These savings are due mainly to the batching of transactions in the epoch algorithm. In the 2-safe scheme, there is no batching of transactions at the backup because the primary and the

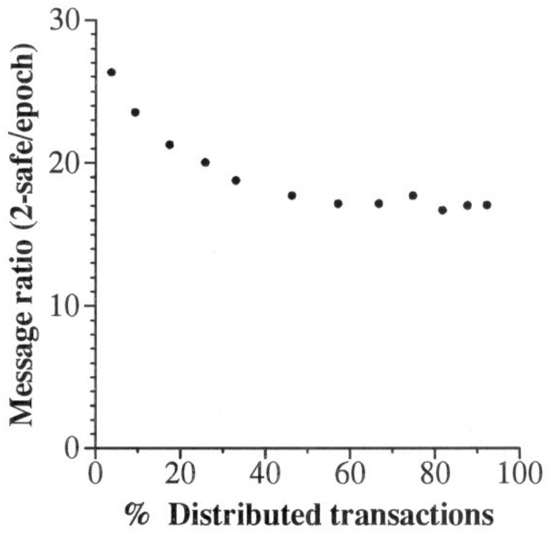


Figure 5.7: Backup message ratio (2-safe/epoch)

backup are coupled. If the 2-safe primary did not commit transactions until an entire batch was received and acknowledged by the backup, response time and throughput would suffer.

The actual cost difference may be higher than shown in Figure 5.7. We count all messages generated at the backup as having equal cost. In the epoch algorithm, all messages are between backups, whereas in the 2-safe algorithm messages are also sent to the primary. If the backups are physically close to each other, but separated from the primaries, messages from a backup to its primary peer may be more expensive than those to other backups.

5.2.4 Network Delays

We emulated longer primary-to-backup delays by having senders hold messages before sending them over the Ethernet to their remote peers.

In the epoch algorithm, the primary is decoupled from the backup, so the performance at the primary is almost immune to network delays. The only change is the time by which the backup lags the primary. In the 2-safe algorithm, transaction response time increased by the round-trip delay. Throughput dropped by 7.5% for a round-trip delay of 1 second and 28.9% for a round-trip delay of 2 seconds. For fast networks, the round-trip delay will be below 1 second, so that throughput loss may be insignificant.

The small decrease in throughput is due to a pipelining effect: the level of multiprogramming was high enough so that while some transactions were waiting for the acknowledgement from the backup, other transactions could do useful processing and keep the critical components of the system (in our case I/O) busy.

However, there are cases where this pipelining effect is not feasible and the impact of delays is more dramatic. Contention is an example. In the presence of contention, a transaction waiting for a commit acknowledgement can prevent other transactions from running because it holds resources (e.g., a lock) that they need.

Contention is a major problem in database systems and special efforts are made to reduce it. For example, IMS [19] first checks data without locking and obtains locks only if the check succeeds, which decreases lock hold times and alleviates contention. More recent attacks on contention hot spots include Reference [33]. In real systems, hot spots can arise when many transactions need to write the same data. For example, a branch balance may have to be updated for every transaction at a bank.

We introduced various levels of contention in our system by creating workloads with hot spots, i.e., heavily accessed records. Each transaction accessed exactly one hot spot record and three ordinary records. When there were more than one hot spot record in the database, the transaction picked one at random with uniform probability. Contention level was determined by the number of hot records and the fraction of read-write transactions. The three levels of contention tested were low (40% read-write transactions, 3 hot records), medium (50% read-write transactions, 2 hot records) and high (40% read-write transactions, 1 hot record). Even in the high contention case, there is some parallelism: read-only transactions can run in parallel with each other and while one transaction holds a lock on the hot spot, other transactions can access other data.

Figure 5.8 displays the ratio of epoch throughput over 2-safe throughput as a function of network round-trip delays for the three levels of contention. Even for round-trip delays as low as 250 msec, the epoch algorithm achieves throughput 2.5 times as high as the 2-safe algorithm.

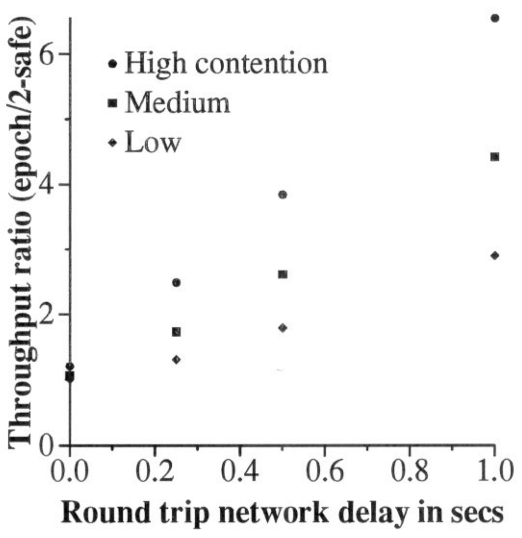


Figure 5.8: Throughput ratio with contention (epoch/2-safe)

5.2.5 Dependency Reconstruction

As for the epoch and 2-safe algorithms, the throughput of the dependency reconstruction algorithm with the base parameter settings at the primary is limited by the 98% I/O utilization. Although logs for read actions are propagated to the backup, read actions are not actually performed there, so the I/O utilization at the backup is again the fraction of primary I/O utilization that corresponds to write actions. Furthermore, the I/O load at the backup is uniform over time as in the 2-safe algorithm.

We varied the read-write transaction ratio and observed the CPU utilization at the primary. As the ratio increases, so does the CPU utilization because write actions are more expensive. There is less than 1% difference between the CPU utilization of the dependency reconstruction algorithm and that of the epoch algorithm, so the dependency reconstruction algorithm performs as well as the epoch algorithm and better than the 2-safe algorithm (see Figure 5.2).

We added extra CPU processing to test the dependency reconstruction algorithm under a CPU-bound workload. Its performance was identical to that of epoch (see Figure 5.3). Two different 1-safe algorithms outperform the 2-safe approach, reconfirming our conclusion

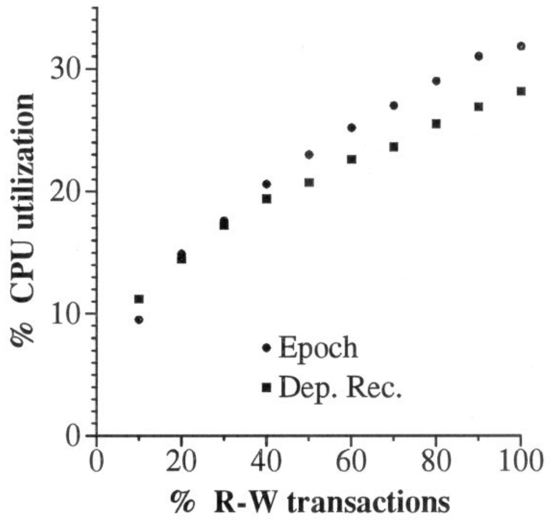


Figure 5.9: CPU utilization at the backup

that 1-safe algorithms can tolerate higher CPU-bound loads.

At the backup things are more interesting. Since the dependency reconstruction algorithm performs more tasks than the epoch algorithm (sets locks, receives and scans log entries for read actions, runs the commit protocol separately for each distributed transaction), one would expect it to have higher CPU requirements. Figure 5.9 shows the CPU utilization at the backup for the epoch and dependency reconstruction algorithms versus the read-write transaction ratio. For few read-write transactions, this expectation is true. However, CPU utilization increases faster for the epoch algorithm and passes the CPU utilization of the dependency reconstruction algorithm at about 20%. The epoch algorithm scans the log twice, once as it is received to detect the epoch marker, and once as actions of committed transactions are extracted and applied, but the dependency reconstruction algorithm scans the log once and scanning logs is expensive. As the read-write transaction ratio increases, the logs grow and the double scanning costs more than is saved in avoiding locking, agreement, and processing of read logs.

The network bandwidth necessary to propagate the logs to the backup is higher in the dependency reconstruction algorithm than in either the 2-safe or epoch algorithms because read actions of read-write transactions must also be propagated. The actual difference depends on the relative mix of read and write actions in read-write transactions. The number of messages exchanged at the primary is the same as in the other two algorithms. The number of messages exchanged at the backup is the same as at the primary (since the backup executes the same agreement protocol). Thus, in terms of messages at the backup, dependency reconstruction lies between the other two algorithms (see Figure 5.6).

Since it is 1-safe, the dependency reconstruction algorithm is almost immune to network delays. In the presence of contention, it has the same performance as epoch, confirming our conclusion that 1-safe algorithms perform better in the presence of contention and network delays.

The performance of the dependency reconstruction algorithm is generally between the epoch and 2-safe algorithms, but it commits transactions at the backup as soon as possible. Thus, in case of disaster, the backup is always ready to start processing new transactions immediately, which reduces takeover time. In contrast, in the epoch algorithm, on average half an epoch will be pending and must be processed at takeover. Furthermore, in the epoch algorithm, if one of the backups falls behind or crashes, processing cannot proceed in any of the other backups. In the dependency reconstruction algorithm, the backups can process local transactions independently of each other; if one backup falls behind, only distributed transactions can be delayed.

5.2.6 Single vs. Multiple Log Streams

We modified our system to use a central log stream: six computers act as primary transaction processors and a seventh acts as the log concentrator. In this experiment, we focused only on the primaries and ignored backups. The concentrator merges the log entries of all primaries and sends them to yet another machine, which simulates the machine that would receive and distribute the logs at the backup; in our case, it discards them. A primary writes its log entries in a local log buffer. To flush the log, it sends the log tail to the concentrator. The concentrator writes these entries in its master log and sends an acknowledgement to the creator of the entries after they have been flushed to disk.

Since log writes take longer, the response time of transactions increases with a log concentrator. However, the throughput remains almost the same because of the pipelining effect mentioned in Section 5.2.4. The six primaries use most of the CPU capacity of the concentrator. We did not have more processors, but we expect that if the number of primaries increases, there will be significant delays in the response of the concentrator, transaction response time will increase, and throughput will decrease. This behavior indicates that log concentration does not scale well and that it can be expensive in terms of hardware: merging the logs requires a dedicated processor, which could otherwise be used to process transactions. On the other hand, central logging works well for a small number of processors and it requires only a single log device for all primaries instead of a separate log device per computer.

A log concentrator can also hurt throughput when there is contention. We verified this effect experimentally by running the medium contention workload mentioned in Section 5.2.4 on our log concentrator testbed. The throughput with multiple log streams was 36% higher than the throughput with the concentrator. Interestingly, contention caused the CPU utilization of the concentrator to drop. Some transactions get blocked waiting for locks to be released and do not generate logs, so the concentrator receives logs at a lower rate and its CPU utilization drops.

5.2.7 Epoch Length Selection

Selecting an appropriate epoch length is a potential disadvantage of the epoch algorithm. Fortunately, the performance of the algorithm is nearly insensitive to the epoch length. The epoch algorithm requires some synchronization between all primaries when an epoch is terminated and some synchronization between backups when an epoch is received. We varied the epoch length from 2 to 20 seconds and observed only tiny changes in CPU utilization (1%) at both the primary and the backup, which indicates that the CPU overhead of synchronization in the epoch algorithm is negligible. The I/O utilization was also independent of the epoch length at both the primary and the backup.

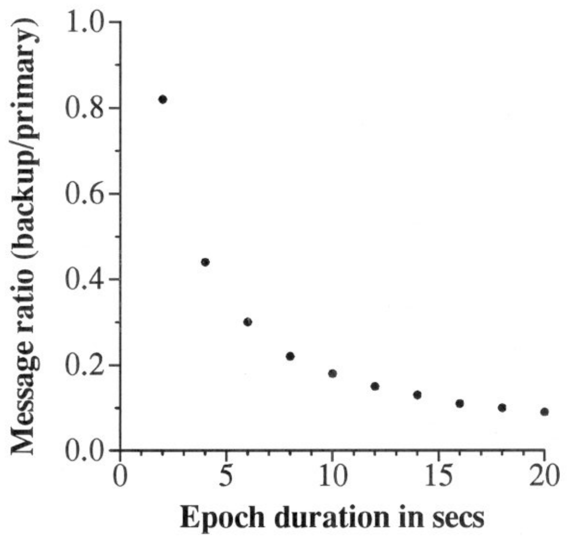


Figure 5.10: Message ratio (backup/primary) for epoch

The time interval between the moment a primary writes the epoch marker in its log until all backups receive and acknowledge the epoch was 150 msec. For wide-area networks, one should add the difference in round-trip delay between Ethernet and the wide-area network. As expected, this interval did not change with the epoch length.

The number of messages exchanged by backups was the only aspect of the system affected by the epoch length. Since installing an epoch requires a fixed number of messages, longer epochs require fewer messages. Figure 5.10 shows the ratio of the messages required at the backup over the messages required at the primary for various epoch lengths.

The epoch length also affects the lag between the backup and the primary. For longer epochs, there are more transactions in the last, possibly unfinished, epoch in case of disaster, and these transactions must be processed before the system can accept new transactions. Thus, epoch length affects takeover time. However, it does not affect the window of vulnerability for transactions: a transaction is vulnerable only until it is received at the backup, not until its epoch is installed. If a disaster occurs and an epoch has been partially received at the backup, the transactions in the incomplete epoch can be processed individually. On the average, half an epoch's worth of transactions will have to be processed in this way, and the time required is proportional to the length of an epoch.

Finally, the epoch length may also affect the performance of hybrid 1-2-safe schemes, where most transactions are run as 1-safe and some important transactions are run as 2-safe. In order to achieve mutual consistency, a 2-safe transaction may have to wait for all of its 1-safe predecessors to commit at the backup before it can commit. If the epoch algorithm is used to process 1-safe transactions in a hybrid system, the response time of 2-safe transactions depends on the epoch length.

Chapter 6

Partial Failures

In a *partial* failure, only some of the primary copy's fragments become inaccessible, so a complete takeover by the backup may be undesirable. A complete takeover involves considerable effort, may cause transaction processing to stop temporarily, and requires that all transactions be redirected to a new location. Furthermore, the primary may be a better place to run transactions (e.g., it may have more computing resources). Thus, there may be less disruption to the system if, instead of a complete takeover, one or more of the backup fragments substitute for their inaccessible primary peers to form a new primary.

Partial takeover may not always be preferable. For example, consider a configuration like that of Figure 2.3, but with copy 2 residing entirely at site S_b rather than being distributed across S_b and S_d. Suppose F_2 at site S_a fails. If we do a partial takeover, the new primary will span S_a and S_b and distributed transactions will be expensive. If, on the other hand, we do a complete takeover, the new primary will reside entirely at S_b. There may be a higher initial cost for switching to site S_b, but better performance during subsequent transaction processing may offset this initial cost. Thus, choosing between complete and partial takeover depends on the ratio of distributed transactions in the workload, the network speeds, the cost of switching transactions to a new processing center and, above all, the algorithm for partial takeover, which is the topic of the rest of this chapter.

With 2-safe backups, partial takeovers are easy, since the logical view of the data at a backup fragment is indistinguishable from that of its primary peer, which makes the substitution transparent to all applications. This property does not hold for 1-safe backups. For instance, suppose a system has fragments F_1 and F_2, and the primary is stored at sites P_1 (holding F_1) and P_2 (holding F_2). A transaction T executes and commits at both P_1 and P_2, and immediately P_1 fails without propagating the updates of T to the backup. If the backup fragment for F_1 at site B_1 is promoted to primary, the new primary at sites B_1 and P_2 only partially reflects T. Thus, a new primary formed by injecting fragments from backup copies may be inconsistent.

The problem outlined above is unrelated to any particular algorithm. Rather, it is inherent to 1-safe backups where copies of a fragment are not updated simultaneously. To bring the new primary database to a consistent state after the partial takeover, explicit actions are necessary. Consistency can be reestablished in two ways. The first option is to "roll forward" the newly injected fragments by providing them with the changes that had been applied by their failed primary peers but did not make it to the backup. This approach makes 1-safe backups equivalent to 2-safe backups with respect to partial failures since no transactions are lost. The second option is to "roll backward" the fragments surviving from the original primary, in order to make them compatible with the injected fragments.

6.1 Roll Forward

Roll forward requires that the logs of a computer remain accessible when the computer fails. In case of partial takeover, the logs of the failed computer are used to bring its promoted backup peer up-to-date with respect to the other primary fragments. Transaction processing resumes in a normal fashion. The ways in which log survival can be achieved vary depending on the extent of the failures one wants to tolerate as partial. If the log is kept on disk (as is usually done), then dual-ported disks can be used, connected to two different processors.

If log data can be destroyed, multiple copies of the log must be made. The extra copies are needed only until the log is propagated safely to the backup, which suggests the following approach. Suppose that a computer P_a holds a primary fragment and B_a is the remote site that holds a backup of the fragment. In many practical configurations, P_a may not be the only computer at its location, so assume that there is another computer, X, and that P_a and X are connected by a fast network with spare capacity. Suppose P_a routes its log stream to B_a through X rather than directly. When a log entry is written by P_a, it is also sent to X, which saves the message (in its buffers, not necessarily on disk) and sends an acknowledgement. The logs are forwarded from X to B_a in the normal 1-safe way. The log entry is considered written by P_a only after both the local disk write has completed and X has acknowledged receipt; this scheme can be viewed as an implicit two-phase commit between P_a and X.

The disk write and the network transmission can proceed concurrently, and they usually take the same amount of time, so response time is not affected significantly. If the writes proceed concurrently, care must be taken in case P_a fails and is restarted: some log data may have made it to X (and then to the backup), without having made it to P_a's disk. At restart time, P_a must check with X and the backup for such log data, so P_a must know about X and the backup. Alternatively, P_a can first perform its local write and then the remote write to X.

Under this scheme, even if P_a fails and the log on its disks is inaccessible, the log entries that have not made it to the backup are available at X, making partial takeovers feasible. If both P_a and X fail, the log will be lost, and a complete takeover is necessary.

There are many variations on this idea. In general, the tradeoff is between the extra computing resources (e.g., local network bandwidth and the additional hop on the path to the backup) and the additional flexibility. Many systems already have an extra hop; for example, the computers processing transactions may have to send the log stream to the backup via one of possibly several gateways. In this case, the extra cost incurred by the above mechanism is small. Finally, systems often maintain redundant log information at another computer for local recovery purposes; this information can be used to bring the promoted backup up-to-date.

6.2 Roll Back

The other alternative for reestablishing consistency is to roll back the fragments that survived from the primary, in order to make them compatible with the injected promoted fragments. This alternative does not require the survival of the redo logs of the failed computer, but it involves more complicated compensating actions upon failure. More specifically, the effects of transactions seen by the surviving fragments but missed by the promoted ones must be eliminated.

There are two main issues in rolling back the survived fragments. First, there must be a way of knowing what changes a transaction made in order to decide how to compensate for it in case of failure. For example, undo information may be kept during normal processing for local recovery. However, since transactions have already committed, it may be more appropriate to use compensating actions, which can be generated and stored during normal processing. These compensating actions are somewhat different from conventional compensating transactions. A conventional compensation is run after a transaction has committed, in order to undo its semantic effects [27]. Here, we must also undo the effects of a committed transaction, but some of these effects may have been lost. For example, a forward *reserve_flight* transaction may have updated a customer's record and a flight reservation record. Because of the failure, the update of the customer's record has been lost and the promoted fragment does not reflect that update. Now we wish to cancel the reservation, but we do not have the latest customer record available (e.g., we may not know how much the customer paid). Thus, compensating transactions must cope with missing data.

To address this problem, the forward transaction can be viewed as a collection of fragment actions (e.g., update customer record, update flight record), and compensation can be performed selectively on a per-fragment basis. Compensation is unnecessary for the part of a transaction reflected on a failed fragment; it is necessary only for the parts of a transaction reflected on surviving fragments. The additional information (undo log or compensating actions) must be kept only until the backup acknowledges that it has installed the transaction.

The second issue is determining how far to go back. An example illustrates the difficulties. Assume that a distributed transaction T executed at fragments F_1 and F_2, that the primary computer that holds F_1 failed, and that the log for the changes of T on F_1 never made it to the backup. Thus, the backup computer that holds a copy of F_1 and is promoted to primary has not seen T. To preserve consistency, the computer that holds F_2

must compensate for T and efface its changes. Since T is effaced, another transaction T' that depends on T may also have to be effaced. Whether or not T' is effaced depends on how T is rolled back.

For example, consider an airline reservation system and assume that T sold the last ticket on a particular flight and that T' subsequently found that flight booked and assigned a passenger to another flight. If only T is effaced, the database enters a correct state, which could have never been reached in any serializable schedule, but may be nonetheless tolerable. In other cases, this state may not be tolerable. For example, consider a withdrawal whose feasibility depends on a previous deposit. Then, transactions like T' must be dealt with. Identifying transactions like T' requires extensive searching and analysis of the logs. If any such transactions are found, they must be effaced using compensation actions, which may trigger further searching and rollbacks. To avoid these cascading rollbacks, a checkpointing mechanism is useful: all fragments go back to the last checkpoint seen by the promoted fragment. The epoch algorithm provides a natural way for establishing such checkpoints.

If there are multiple backup copies, yet another compatibility issue arises. The new primary that is formed after the partial takeover may have missed a few transactions. If there are other backups, they must now track the new primary. Before doing so, they must synchronize with it, since backup algorithms assume identical initial states for the primary and the backup. A transaction that is not in the new primary may have made its way to another backup and must be effaced from that backup.

After a partial takeover, the backups that conceded some of their fragments to the new primary are incomplete. Their remaining fragments may still track the new primary. An incomplete copy can assume that missing fragments reside in a *phantom* (non-existent) site. To maintain consistency, the partial copy can assume that the phantom site never fails, receives all information sent to it and makes decisions in a way that is most convenient for the real sites that hold data.

When the system recovers from the partial failure, the fragments that were promoted to primary are demoted to backup and returned to their copy of origin. In the meantime, however, the returned fragments may have seen more transactions than the fragments they are rejoining. Thus, the restored backup copy may consist of fragments that belong to different database versions, so care must be taken to preserve consistency and make the fragments compatible. For example, processing may halt at the demoted fragments until the other backup fragments are brought up-to-date with respect to the demoted ones.

6.3 Transaction Durability

Partial takeovers can lead to effacing committed transactions, and disasters under 1-safe processing can lose committed transactions. Transactions are usually characterized as atomic, consistent, isolated, and durable. Commitment is strongly related to durability; after commitment the effects of a transaction should persist even in the presence of failures. When compensating transactions are used, our processing rules seem to violate this condition, since committed transactions may be undone in case of a subsequent failure.

It is impossible to guarantee durability against all failures. No matter how many copies of data are used, they cannot be prevented from all failing simultaneously. Adding more copies can only reduce the likelihood of such an event. Thus, each system selects a set of *expected* undesirable events, i.e., a set of undesirable events that it will tolerate, and assumes that other failures never occur. Durability is always defined with respect to these expected events.

A difference between 1-safe and 2-safe is the set of expected undesirable events they can tolerate. After a transaction has been installed at the backup, the two methods offer the same degree of protection. In 2-safe systems, a failure during the propagation of a transaction to the backup is in the set of expected undesirable events. In 1-safe systems, committed transactions are vulnerable during propagation, and a failure during this small time window is an unexpected undesirable event. Thus, 1-safe systems trade protection for performance.

As we have mentioned, a hybrid 1-2-safe scheme can be employed for important transactions. A transaction that cannot tolerate the vulnerability window of 1-safe transactions can run as 2-safe: it holds on to its resources at the primary and commits after it is processed at the backup. To preserve mutual consistency, all of its predecessors (including those that are 1-safe) must be processed at the backup before the 2-safe transaction commits.

In the epoch algorithm, there are two ways to implement the hybrid scheme. The first is to delay the commit of the 2-safe transaction until the epoch terminates, which may introduce significant delays for this transaction. The second is to end the current epoch early, which reduces the delay for the 2-safe transaction, but may introduce additional overhead depending on the frequency of 2-safe transactions. In either case, the response time of 2-safe transactions can be improved if these transactions wait only for the logs of their predecessors to be received at the backup, not necessarily to be installed.

Chapter 7

Processing Read-Only Queries

Previous chapters considered remote backups for recovery purposes. A remote backup represents a significant investment: the computing resources at the backup are comparable to those at the primary; in addition, network resources are required to propagate the logs to the backup. Some critical applications need the protection offered by a remote backup and are willing to pay its price. However, these applications might profit from exploiting the backup for useful computation as well, rather than just for monitoring the primary. Such use would lower the cost of maintaining the backup, and could possibly make a remote backup affordable for less critical applications.

This chapter examines mechanisms for running read-only queries at the backup. A number of mechanisms have been suggested for processing read-only queries [3, 6, 18, 36, 38]. These include multi-version algorithms that allow queries to read older snapshots of the database. The disaster recovery scenario also involves multiple copies, one at the primary and one at the backup, and queries will read an older snapshot of the data. However, the way in which the updates are installed at the backup distinguishes the disaster recovery scenario from others.

As mentioned in previous chapters, existing backup systems are invariably log-based. Log-based backup management *decouples* the installation of updates at the backup from the primary transaction processing and leads to improved performance at the backup. In particular, updates can be applied more efficiently in batches. For instance, a group of updates can be sorted and applied in physical disk order, reducing seek distances.

Batch installation of updates is possible in 1-safe (e.g., epoch) and in 2-safe systems. In 2-safe systems, a transaction cannot commit at the primary until its log records are safely received, but not necessarily installed, at the backup. The backup must acknowledge receipt of the logs immediately, but it can install the changes later, after the transaction commits at the primary. We use the term *group* for a batch of updates to denote that our results apply to all types of backups, not just the epoch algorithm.

Query algorithms at the backup can also benefit from the primary/backup decoupling, so that query processing will not impact primary transaction processing. Query processing still interferes with update installation at the backup, but given the nature of updates, several key optimizations can be performed. For instance, queries executed when no updates are being installed need no concurrency control. Queries that run during update installation may know *in advance* what objects will be modified by a batch of updates and can use this knowledge to improve performance. In this chapter, we study such optimizations, and present several simple but efficient algorithms for query processing at the backup. The performance of these algorithms is studied through an analytical model.

We have presented log-based, decoupled backup copies in the context of disaster recovery. However, this scenario may be justified even when reliability is not the major consideration. In some cases, queries arise naturally at one computer and updates at another. For example, a company may use front-end computers at stores (modifying inventory, recording orders, etc.), and a back-end computer at headquarters for queries (trend analysis, budget summaries, etc.). In cases where the query load is significant, decoupled query processing may actually provide the most effective solution. Finally, it may be desirable to run queries on a computer with different software than the one running updates. In all these scenarios, best performance can be obtained with the query processing algorithms we study here, while also achieving high reliability.

7.1 Motivating Example

As shown in Chapter 5, the backups have significant spare CPU and I/O capacity, which we want to exploit. The application of updates in groups causes backup I/O traffic to be periodic. The I/O subsystem is idle while the backups are waiting for the next group to accumulate. Such periodic activity can provide long I/O queues, which may lead to better disk arm scheduling and higher I/O throughput [40, 43]. In order to verify the applicability of these results to our testbed, we sorted the logs in physical layout order before applying them. The installation time for the sorted logs was half the time for the unsorted logs. This result is only preliminary. We have no control over the disk arm scheduler. The central process issues writes in sorted order, but they reach the disk via independent I/O slave processes, so that CPU scheduling of those processes may change the sorted order. Our database occupies only a fraction (25%) of the disk, so seek times are short. In Unix, it is possible to issue only one I/O request per system call. In systems without these limitations, the improvement achieved by sorted logs may be greater.

When the logs are sorted, the order in which the updates are applied may not correspond to any serializable schedule. However, consistency is preserved as long as the state of the database after the application of an entire group is the same as it would be if the log entries had been applied in sequence.

Since queries compete with updates for the same resources, it is interesting to investigate when it is most appropriate to run queries. We experimented with two techniques. The first was to *separate* queries from updates in time and to run queries only during periods of update quiescence. This technique implies that no concurrency control is needed and that queries observe a consistent version of the data. However, response time of queries may suffer, since they are postponed until the end of an update group. An alternative is to run updates and queries concurrently, *dividing* the CPU and I/O resources between them. For example, in our testbed we divided the resources in each backup computer as follows. Ten threads installed updates and three threads processed queries concurrently without any synchronization between queries and updates; the queries may observe inconsistent data. Our experiments showed that the separation technique always provides higher throughput and a higher level of consistency. Figure 7.1 plots the percent gain in query throughput of the separation technique over the division technique for various fractions of read-write transactions on a 80MB database; other parameters are as in the base case of Section 5.2. When all transactions update, the separation technique can process 18% more queries.

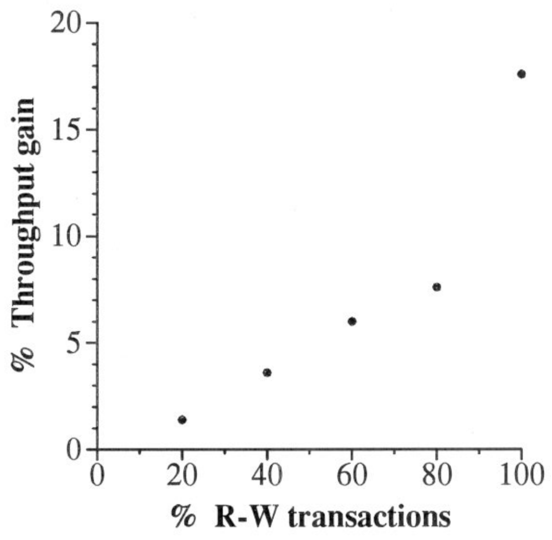


Figure 7.1: Gain of separation vs. division technique

These experimental results indicate that there are significant differences between query processing techniques. Rather than presenting additional experimental results, we focus on understanding the choices and tradeoffs involved for query processing in this environment. We cannot get this understanding directly from the experiments because our observations depend on implementation details, e.g., should we use more than three threads for queries, is our system too I/O-bound, is our buffer manager biasing the results? Thus, we develop a simple model and describe a suite of implementation choices for query processing. The model highlights the critical parameters and ignores the secondary ones. Our intention is not to predict the actual performance of systems, but to provide insight and guidance to system builders, so that they can select the most appropriate technique.

We consider only one primary and one backup computer; the situation is analogous when multiple computers are involved. The logs are installed in sorted groups in order to gain in efficiency, as mentioned above. The spare capacity is used for query processing. We assume that queries are ordered batch [3], i.e., they must access their data in a specific order. If they block on a particular access request, they cannot proceed until that request is satisfied.

We consider only queries that require a consistent view of the database, which means that the division technique must be extended to provide consistency. We also consider only relatively short queries, i.e., they can complete within the duration of a single group. Long queries that demand consistency can be processed with multiversion mechanisms. Finally, we assume that queries can tolerate slightly out-of-date data and that they do not have strict response-time requirements, which implies that queries can either view the previous consistent version of the database, or they can be deferred until the current group has been installed. If queries have stringent response-time requirements and need absolutely recent data, they should run at the primary. We consider other types of queries briefly in Section 7.5.

7.2 Algorithms for Processing Read-Only Queries

During a cycle, i.e., between the termination of two consecutive groups n and $n+1$, a backup computer performs three tasks: it receives the logs for group $n+1$, installs the updates for group n, and processes the queries that view the database state after group n. The first task is *logically* independent of the other two and can be performed in parallel with them. Often it is also *physically* independent, since a separate disk may be used to hold the log (if the log is stored on disk; see Section 7.5). However, applying updates and executing queries may conflict, since they may need to access the database in conflicting modes.

The query workload that is executed during a particular cycle can be divided into two classes: a *prescheduled* portion of the query workload is available at the beginning of the cycle. During the cycle, there may be *incidental* incoming queries. The system can decide to execute an incidental query immediately (during the current cycle) or to defer it until the next cycle. The decision can be based on various parameters, e.g., the current load on the system, the response-time requirement of the query, etc. If the query is deferred for the next cycle, it becomes a prescheduled query for that cycle.

In what follows, we assume for simplicity that the logs contain the after images of entire pages. Such logs can be applied directly to the database. However, it is possible to have

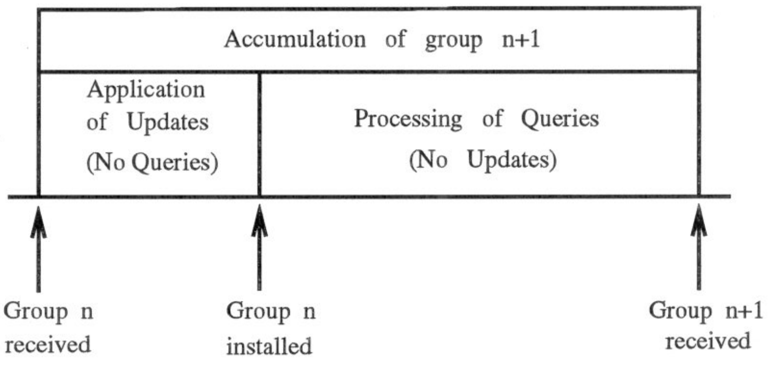

Accumulation of group n+1
Application of Updates (No Queries)
Processing of Queries (No Updates)
Group n received
Group n installed
Group n+1 received

Figure 7.2: Separating updates from queries

logs that need processing; for example, the log may contain logical actions or it may only contain the portion of a page that has been changed. In this case, the logs need to be processed and the before images of pages need to be read, increasing the read traffic.

In examining various possible mechanisms for performing the updates and answering the queries during a cycle, we consider four access conflict resolution techniques: separation of updates and queries, pointer-conflict detection, blocking map-conflict detection, and map-conflict detection with on-demand installation of updates. The first approach is the separation method discussed in Section 7.1: the updates are performed first followed by the queries (see Figure 7.2). While the updates are being applied, read access to the data is inhibited. While the queries are being executed, the data is left unchanged. Disk bandwidth utilization can be very high under this scheme.

If the logs do not require significant CPU processing, the CPU may be underutilized while the updates are being installed, since no queries are run. Furthermore, since queries must wait for the installation of the updates, their response time may increase. The main advantage of the separation scheme is its simplicity: there is no synchronization overhead.

Another approach is to perform updates and queries simultaneously. To guarantee consistency, concurrency control must be used. However, instead of using conventional mechanisms, such as two-phase locking, we can take advantage of the specific circumstances. In particular, only one process updates and all its updates are known in advance (at the beginning of the application of group n). The following algorithms are adaptations of locking to these circumstances.

One strategy is to apply the updates in the physical layout order of the data on the disk and to track the progress of the update process using a pointer. This strategy is similar to many two-color algorithms [36, 38]. The update process is

```
pointer ← address_of(first_data_item)
while there are more updates
    q ← next_update in physical layout order
    write(q)
    temp ← pointer
    pointer ← address_of(q)
    release all queries blocked between temp and pointer
```

A query can only read data that precede the current position of the pointer.

```
if p ≤ pointer then read(p) else block on p
```

This method detects conflicts easily, but it denies access unnecessarily if the data is not modified by the current group but simply lies beyond the current pointer position.

Another way to detect conflicts is to keep a map (e.g., a simplified lock table) of the pages modified by the group being installed. The map can be constructed as the logs are received and scanned and before any queries that need the new state of the data are processed. Under this scheme, before a query can access a page, it must first check the map to see if an update is pending for that page. If an update is pending, then the query blocks until the new version of the page is installed by the update process.

```
if map(p) is set then block on p else read(p)
```

When the update process installs a new page, it resets the corresponding entry in the map.

```
while there are more updates
    q ← next_update
    write(q)
    reset map(q)
    release all queries blocked on q
```

Yet another possibility is to detect conflicts using the map, but to resolve them on a demand basis: when a query needs to access a page for which an update is pending, the update is fetched from the log and installed before the time at which it would normally be installed. The query then proceeds.

```
if $map(p)$ is set then
    $write(p)$
    reset $map(p)$
    release all queries blocked on $p$
$read(p)$
```

The update process is the same as in the previous case. This option improves response time, but it requires that the modified page be installed out of sorted sequence, which may degrade the performance of update application.

In the pointer and map methods, the updates are installed concurrently with queries and the database contains a mixture of two versions. Since the updates are installed gradually, the database is closer to version $n-1$ than to version n at the beginning of the cycle. This observation suggests a "dual copy access" optimization: incidental queries that arrive at the beginning of the cycle view version $n-1$. In this way, they do not wait for updates to be installed, but read older data. If a query comes across data that has already been updated by group n, the attempt fails; the query is aborted and retried with version n.

Another optimization is possible when all updates for a group fit in the main memory buffer cache. While the log is being received, the new images of pages are stored in the database cache. When the group is to be installed, rather than write the pages to disk, the data structure that tracks the valid pages in the cache is updated to reflect the presence of the new pages. The modified pages can be written to disk at a convenient point (e.g., when the disk arm services a read request at the same or an adjacent cylinder), which hides the cost of many writes under the cost of the reads.

7.3 Performance Evaluation

We discuss the suitability and relative performance of the query processing schemes for various query workloads, focusing on query throughput, not on response time. We assume that the duration of a group is 1. The factors that affect this choice are discussed in the next section. We adopt a very simple system model and consider only two computing resources, CPU and I/O. Although very simple, the model applies to real systems as we have verified experimentally. We consider only the I/O operations on the database disks and ignore the log disks.

A query performs r steps, each involving some CPU processing and some I/O. The CPU can process A steps in unit time, if no updates are present. Thus, the CPU can process a single step in time $1/A$. The disk takes time $\frac{\lambda}{A}$ to satisfy the I/O requirements of a step, where λ characterizes the query workload, since it determines the relative utilization of I/O and CPU by a query. As mentioned in Section 7.1, when a query initiates a step that cannot be processed immediately (because the data it wants to access is not available), the query blocks until the data it needs becomes available. Furthermore, we assume that all queries are prescheduled queries and available at the beginning of the cycle. In what follows, we calculate how many queries the various processing schemes can run in unit time.

The update process is assumed to use only I/O and have negligible CPU requirements. We also assume that updates are installed most efficiently if there is no read traffic. In particular, we assume that all of the updates can be installed in physical layout order in time u. A random read request that is processed in the middle of the sequence of updates incurs an overhead, since it causes the disk heads to move to serve the read request and disrupts the efficient installation of updates. We consider this overhead part of the read request and assume that the I/O of a step executed during the update period takes time $\frac{\lambda p}{A}$ rather than $\frac{\lambda}{A}$, where p is a penalty factor, $p > 1$.

Results from our testbed confirm the I/O interference of updates and queries. Figure 7.1 showed that mixing queries with updates reduced throughput. Since the system is I/O-bound and there is no concurrency control in that particular configuration, the reduced throughput implies that mixing effectively increases I/O time. For the settings of Figure 7.1, we estimate that $p = 1.14$. In this case, the database is 80MB, or 25% of the disk capacity. For larger and more realistic databases that fill most of the disk, p would be larger.

First, consider the case $\lambda > 1$, i.e., I/O-bound queries. For this type of query workload, the separation algorithm is the most appropriate. Since the I/O capacity is the limiting

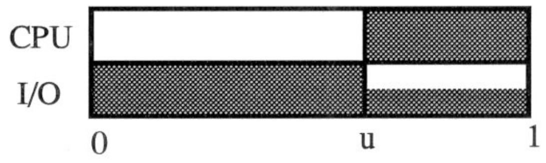


Figure 7.3: CPU-bound queries

factor, we want as much I/O capacity as possible available for queries, which implies that the updates must consume as little I/O capacity as possible. According to our model, this effect can be achieved if the updates are installed separately from queries. Again, the experimental results from our testbed, which is I/O-bound, confirm this conclusion.

The rest of this section considers the case $\lambda < 1$, i.e., when the queries are CPU-bound. If the separation algorithm is used for query processing, resource utilization is as shown in Figure 7.3, where shaded areas correspond to utilized resources. From time 0 to time u, while updates are being applied, the CPU is idle, because no queries are executed. From time u to time 1 CPU-bound queries are run and the CPU is saturated. During this period, only a fraction λ of the I/O capacity is utilized, so that the I/O area is not entirely shaded in Figure 7.3. The number of query steps executed is $A(1 - u)$.

7.3.1 Analysis of Map Method

Since CPU cycles are wasted during the update period and CPU is at a premium, the pointer and map algorithms try to run more queries. Consider map detection under a best case scenario: queries never block, i.e., they access either data that did not change during the group or data for which the new version has already been installed. This case gives an upper bound on the expected improvement in the number of queries that can run within a group period. Actual performance will approach this bound when a small fraction of the total database is modified by a group.

Suppose we move t seconds of query processing into the idle period. We must also move the corresponding amount of I/O, λt, into this period. According to our model, the interference with the updates multiplies the cost of this I/O by p, so its new cost is λpt, and the update period grows by that amount as well. The net effect will be a gain if the

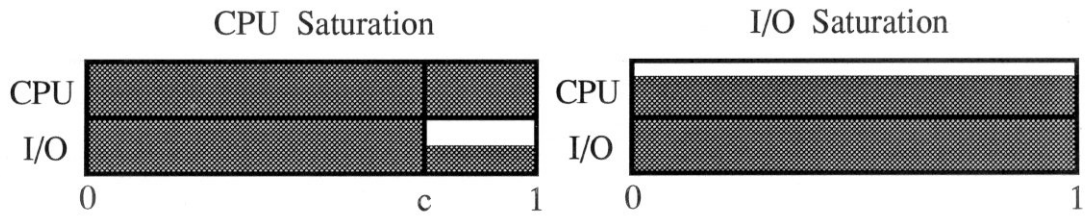


Figure 7.4: Saturation of CPU (left) and I/O (right)

CPU time moved into the update period is longer than the growth in time of the update period, i.e., if $t > \lambda pt$ or $\lambda p < 1$.

If $\lambda p < 1$, moving query processing into the update period leaves some spare time in the query period, which can run more queries. We can keep doing this rearrangement to fit as many queries as possible into a group period. However, since the CPU utilization grows faster than the disk utilization during the update period, the CPU utilization will catch up with the disk utilization. This effect is shown in Figure 7.4 (left), where the catch-up point is denoted by $c = u + \lambda pc$, so $c = \frac{u}{1-\lambda p}$. In this case, the CPU utilization during the group is 100%, so we can run A steps. The relative increase gained with respect to the separation algorithm in the number of steps executed during the group cycle is

$$G = \frac{A - A(1-u)}{A(1-u)} = \frac{u}{1-u}$$

However, depending on the values of the parameters, we may exhaust disk capacity (i.e., the update period reaches the end of the group) before the CPU utilization catches up with the disk utilization (Figure 7.4, right). In other words, the catch-up point may lie beyond the end of the group. This situation occurs if $\frac{u}{1-\lambda p} > 1$, and only qA steps get executed, where q is the CPU utilization. To compute q, note that qA steps generate $\frac{qA\lambda p}{A} = q\lambda p$ of I/O activity. In addition, u I/O time is taken by the updates and $u + q\lambda p = 1$ (I/O saturates). Thus, $q = \frac{1-u}{\lambda p}$. The relative throughput increase with respect to the separation case is

$$G = \frac{qA - A(1-u)}{A(1-u)} = \frac{1-\lambda p}{\lambda p}$$

Figure 7.5 shows the relative increase G for $u = 0.4$ (left) and $u = 0.7$ (right) as a function of λ. The curves correspond to $p = 2$ (solid line), $p = 3$ (dashed line) and $p = 5$

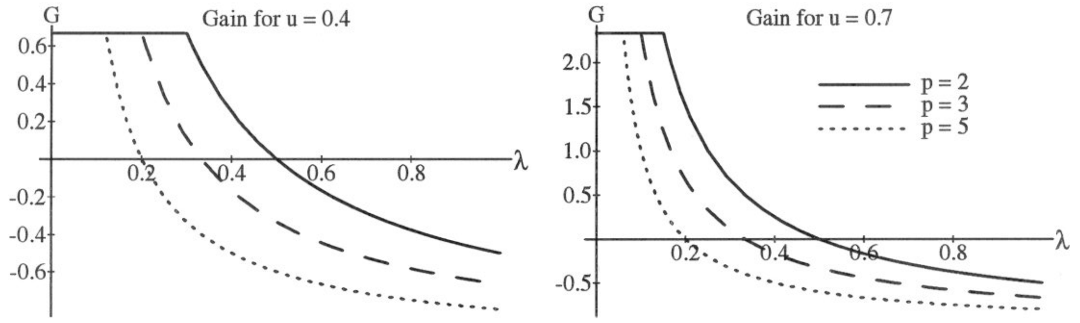


Figure 7.5: Relative throughput gain (map vs. separation)

(dotted line). The gain can be significant. For example, for $u = 0.7$ (70% of I/O capacity is used for update installation), the map method may process more than three times as many queries as the separation method. The gain could be smaller if queries block often. The gains grow as u grows, since more CPU cycles are wasted in the separation method and exploited in the map method. The gains shrink as p grows, since each step needs more I/O time in the update period and I/O saturates with a smaller number of steps. When $\lambda p = 1$, $G = 0$, i.e., the map and the separation method perform the same. If $\lambda p > 1$, the map algorithm is counterproductive because I/O saturates. The slope of the curves shows the sensitivity to λ. The steeper the slope, the more sensitive the gain is to small changes in λ.

7.3.2 Analysis of Pointer Method

In the pointer method, queries may block because the data they want to access lie beyond the position of the pointer, and the CPU may be underutilized. Using the pointer method, we can perform a fraction v of the maximum number of steps within a group period. In other words, we can execute vA steps, which corresponds to $\frac{vA}{r}$ queries, since each query issues r steps. Our goal is to calculate v.

Let $f(t)$ be the fraction of the data that has been scanned by the pointer and $B(t)$ be the number of steps that must be performed after time t because they will be issued by blocked queries. For example, if a query must issue 4 steps and at some moment it is blocked on its second step, it contributes 3 steps to the backlog at that moment. Assuming uniform

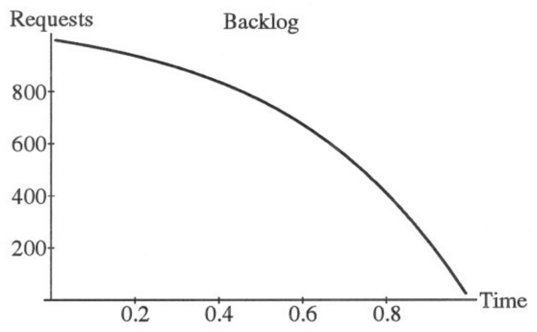


Figure 7.6: Backlog

access to the data, a step issued by a query succeeds with probability $f(t)$ and blocks with probability $1 - f(t)$. Thus, when the pointer has scanned a fraction f, the probability that a query contributes k steps to the backlog is the probability that the first $r - k$ steps of the query can be satisfied with data that precede the pointer, while the next (i.e., $r - k + 1$st) step requests data that lie beyond the pointer. The expected backlog is the average number of backlogged steps per query times the number of queries:

$$B(f) = \frac{vA}{r} \sum_{k=1}^{r} k f^{r-k}(1-f)$$

In what follows, we approximate the actual value of the backlog with its expected value. Figure 7.6 shows the expected backlog as a function of f for $vA = 1000$ steps and $r = 4$ requests per query. In the beginning, a very small portion of the data is accessible, so most of the queries block and the backlog is nearly vA. The derivative $\frac{-dB}{dt}$ gives the rate at which steps are released by the advance of the pointer. In other words, this rate is the rate at which the CPU is presented with work. As long as this rate is lower than the capacity of the CPU (i.e., A), the CPU can consume the released amount of work immediately and is underutilized. At some crossing point t_c, with a fraction f_c scanned, the release rate becomes equal to the capacity of the CPU, and then exceeds it, so CPU utilization is 100%. In order to compute this point, we must solve $\frac{-dB}{dt} = A$ or $\frac{-dB}{df}\frac{df}{dt} = A$. We can replace the factor $\frac{df}{dt}$ by noticing that at the crossing point (and thereafter) the query processing activity saturates the CPU and therefore takes up a fraction λp of the I/O capacity. We know that originally (in the separation method) the updates were installed at a rate $\frac{1}{u}$.

Now, the updates are installed by the remaining I/O capacity, i.e., at a rate $\frac{1-\lambda p}{u}$. If we assume that the updates are scattered uniformly on the data, the rate of update installation is also the rate at which the pointer is scanning the data. Thus, we can take $\frac{df}{dt} = \frac{1-\lambda p}{u}$ at the crossing point, so that $\frac{-dB}{df}\frac{df}{dt} = A$ becomes

$$\frac{v}{r}\left[\sum_{k=1}^{r} k f_c^{r-k} - \sum_{k=1}^{r-1} k(r-k) f_c^{r-k-1}(1-f_c)\right]\frac{1-\lambda p}{u} = 1 \tag{7.1}$$

Time t_c is the time it takes to install the fraction f_c of updates plus the time it takes to satisfy the I/O requests for the query steps that completed before t_c. Since the installation of all updates requires time u, the installation of fraction f_c requires uf_c. Before the crossing time, $vA - B(f_c)$ steps completed. These steps require $\frac{\lambda p(vA-B(f_c))}{A}$ I/O time, thus,

$$t_c = f_c u + \frac{\lambda p(vA - B(f_c))}{A} \tag{7.2}$$

After t_c, the CPU is fully utilized processing the remaining $B(f_c)$ steps, which take time $1/A$ each, so that

$$1 - t_c = \frac{B(f_c)}{A} \tag{7.3}$$

Combining Equations 7.2 and 7.3 yields

$$\frac{(1-\lambda p)B(f_c)}{A} + f_c u + \lambda p v = 1 \tag{7.4}$$

Equation 7.4 is independent of A, since the A in the denominator cancels with the A in $B(f_c)$. We can solve simultaneous Equations 7.1 and 7.4 numerically for f_c and v. When we obtain the value of v, we calculate the relative gain in throughput with respect to the separation algorithm

$$G = \frac{vA - A(1-u)}{A(1-u)} = \frac{v-(1-u)}{(1-u)} \tag{7.5}$$

For values of λ such that $u + \lambda p \leq 1$ or $\lambda \leq \frac{(1-u)}{p}$, the I/O never saturates and CPU utilization is limited only by the backlog effect. For $\lambda > \frac{(1-u)}{p}$, I/O saturates at some point λ_s. For values of $\lambda < \lambda_s$, our analysis holds. For $\lambda > \lambda_s$, our analysis does not hold, since the CPU is not saturated. It is difficult to characterize the point λ_s, so we proceeded by inspecting points beyond $\lambda = \frac{(1-u)}{p}$ manually. If the I/O was not saturated,

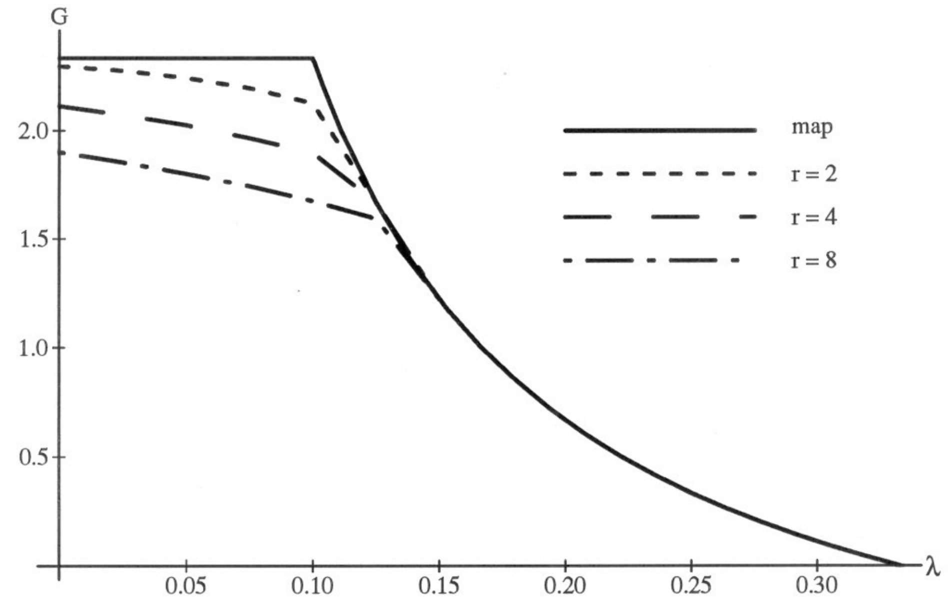


Figure 7.7: Gain in the pointer method for $u = 0.7$, $p = 3$

then Equation 7.5 was used. If I/O was saturated, the gain was the same as in the case of I/O saturation under the map method.

Figure 7.7 shows the relative increase in throughput (with respect to the separation method) as a function of λ, for $u = 0.7$, $p = 3$ and for $r = 2$ requests per query (dotted), $r = 4$ (dashed) and $r = 8$ (dot-dashed). For the same value of λ, the gain decreases as r increases, since the backlog effect becomes stronger. For comparison, the solid line is the curve for the best case for the map method (no blocking). The actual map method blocks on some accesses, but it will do so less frequently than the pointer method, since it will block only on modified data, not data that just happens to be located beyond the pointer. Thus, the pointer method gives a lower bound on the performance of the map method, and the real map-method curve lies between the pointer-method curve and the best-case map-method curve.

The main parameter that dictates query performance at the backup is λ. If the I/O time required by a query is less than 20–40 percent of its CPU time (roughly $\lambda < 0.2$ to 0.4), mixing update and query processing pays off. In such a situation, the map method is probably more effective, although the performance difference with the simpler pointer method is insignificant.

7.4 Group Size

A major factor that determines group duration is the takeover time. If a disaster occurs at the primary, the backup must complete the installation of the pending group and start accepting incoming transactions. Thus, the takeover time requirement puts an upper bound on the time it takes to install a group and on the size of the group. If updates are installed on a demand basis, it is possible to start transaction processing without having completed the installation of the group. Performance may be degraded initially, since transactions may have to wait for the application of updates, but the takeover will not be delayed.

Query parameters also affect group size. The group must be long enough to allow the biggest query to complete in a single cycle under this mechanism. On the other hand, long groups may increase the response time of queries that have to wait for the next cycle.

Throughput generally improves as groups become longer, since the longer the group the closer processing comes to batch mode. However, certain optimizations apply only when the group length is limited. For example, storing the updates in the buffer cache can be used only if the group size is small enough for all of the updates to fit in the cache.

7.5 Discussion

Our performance model is simple, but helps to understand the key tradeoffs. For instance, the parameter p captures the cost of interleaving update and query I/Os. In practice, this parameter depends on the disk scheduling algorithm, the disk geometry, the load, the seek functions, and so on. Introducing all these parameters and factors obscures the issues. At a later stage, when a particular system and application are known, a detailed model or actual experiments would be required to measure the values of p and other parameters.

It is efficient to apply the updates in physical layout order, but doing so requires sorting the log. To avoid sorting, the incoming log stream can be broken into substreams, each covering a contiguous part of the disk and having an expected size equal to the buffer cache.

When updates are installed, each substream is read into memory and its updates are applied in physical layout order.

In the case of multiple disks, our discussion applies to each disk separately. Since it is reasonable to assume that update installation proceeds in parallel and at the same rate on all disks, the fraction f of scanned data for the entire database is the same as the fraction f for each individual disk.

When the backup is distributed across several computers, each computer can use our techniques to install the local updates and run the local queries efficiently. Distributed queries (which access data at more than one computer) require that the database be in a *globally* consistent state after the installation of a group, which is a thorny issue [6]. The epoch algorithm of Chapter 4 can be used to achieve this property. Its overhead may be another factor in determining the size of groups.

An interesting issue is where the received logs are saved at the backup. If they are stored in non-volatile storage, they can survive crashes of the backup; their receipt can be acknowledged immediately to the primary, which can discard them. If the logs are stored in memory (as in the buffer cache optimization), the primary cannot discard them until the updates have actually been installed at the backup; otherwise, a crash at the backup would lose the logs. Furthermore, if the received logs are stored on disk, the system can tolerate a disaster at the primary with a simultaneous crash at the backup. If the logs are stored in memory, this scenario would result to loss of the last group received and not installed.

In the previous sections, we addressed ordered batch queries with an order different from the physical layout order. This scenario is the worst case for queries. *Random* batch queries can issue access requests in any order. Thus, the queries do not block when the data for their current request is unavailable and performance gains are as shown in Figure 7.5. Queries that scan the database in physical layout order (scan queries) can be processed efficiently: as the disk head moves to install the updates in layout order, it can also access the pages that the scan queries want to read. In our terminology, p is 1, since the scan queries do not disrupt the installation of updates.

Chapter 8

Conclusions and Future Work

Remote backups are gaining ground as a technique for achieving high availability. We have demonstrated efficient and scalable algorithms for maintaining a remote backup copy. These algorithms apply to a wide class of system configurations.

Our algorithms can be implemented with a moderate amount of effort and can be added to existing systems without requiring extensive modifications. Furthermore, the extra computing resources consumed at the primary are insignificant.

At the backup, the computing resources required to maintain the copy up-to-date are significantly fewer than the primary, because the backup needs to replicate only part of the primary's work and because primary/backup decoupling allows efficient backup processing schemes. These facts suggest that the backup can use less powerful processors or that a single backup processor can serve multiple primaries. The backup can use spare capacity to process read-only queries and thus further lower its cost.

Network technology is the dominant factor in the choice between 1-safe and 2-safe backups. Fast and reliable networks favor the 2-safe approach, which provides more protection and maintains a single logical view of the data. Slow networks can hurt performance when there is contention in the transaction load. The extra cost of a high-performance network required by 2-safe must also be weighed against the economical cost of the potential loss of a few transactions in case of disaster under 1-safe.

Finally, the choice between a centralized logging scheme and multiple independent log streams should be based on the size of the system and the resources available. A central log works well for a small number of processors and requires only one log device. However, log concentration is inefficient in terms of CPU utilization and does not scale to large numbers of processors.

An interesting area for future work is in fine tuning the algorithms, especially at the backup. The results from applying the logs in sorted order have been encouraging. We would like to experiment with other forms of log preprocessing. For example, we can replace multiple modifications to the same data item during an epoch with the last one, which might reduce updates to hot spots. The interaction between the log installation process and the buffer manager becomes important: when the logs are preprocessed, the access pattern at the backup is entirely different from the pattern at the primary, so LRU may no longer be an appropriate page replacement policy.

Database initialization deserves further study. Different techniques may be suitable for different kinds of update workloads. Furthermore, it may be possible to exploit primary/backup decoupling in order to reduce the interference between the scan process and the regular installation of updates. This reduction will shorten catch-up time.

Cross-backups are a special case of interest. Both sites perform both primary and backup tasks, so it is irrelevant whether some task is performed at the primary or the backup; the sites can trade equal amounts of processing. For example, the logs could be preprocessed while they are written and *before* they are sent. This preprocessing might eliminate the first pass through the log, which is an expensive operation as our experiments indicated. Furthermore, preprocessing might reduce the log volume and save communication costs.

In a nutshell, the techniques presented in this thesis manage replicas with asynchronous, log-based propagation of updates. It is appealing to investigate the applicability of these techniques to a broader area. File systems are a good target, since they offer the largest community of potential users. Furthermore, an increasing number of file systems use a log for reliability and performance.

A *write-ahead* log ensures that the consistency of the file system can be preserved if a failure occurs in the middle of a critical operation. Recovery time is also reduced, because only the tail of the log must be examined instead of the entire file system. In contrast, when a Unix file system recovers, a system process traverses the entire file system to detect and correct inconsistencies such as unreferenced inodes, incorrect link counts in inodes or the superblock, missing blocks in the free list, etc. Performance is improved with a log because synchronous I/O is done sequentially to the log rather than randomly to the disk location of the data. The log-structured file system (LFS) [37] carries the log idea to an extreme and uses the log as the *only* copy of the data; performance improvement is achieved through temporal locality of the data rather than logical locality.

In systems that use logs, the modifications must eventually be reflected on the master copy of the data, which can be done asynchronously. It is interesting to investigate how our techniques could be used to increase disk bandwidth for synchronous writes by allowing multiple logs and to update the master copy of the data from the logs efficiently.

In LFS, the log is the only copy of the data. Periodically, the log is "cleaned" by copying live data onto clean segments to compact space. Cleaning is another instance of asynchronous, log-based installation of updates. Rather than maintain the log structure across the copying, it would be interesting to investigate the possibility of installing the updates in a conventionally structured file system with logical locality. This approach would preserve the advantage of fast, sequential synchronous disk writes and eliminate the disadvantage of slow sequential reads of files that have been written randomly, which is the weak point of LFS.

Asynchronous, log-based propagation of information has been used recently for reasons of autonomy. An example is the Coda file system [26], designed to provide file-system availability in cases of disconnected operation. During disconnected operation, a client maintains a log of the update activity to the file system. This log is used at reintegration time to propagate the changes to the central data server.

In the current Coda system, changes are propagated at reintegration time in a single log stream and processed with a low level of parallelism. Furthermore, update activity is suspended during reintegration. Multiple log streams from the same client would expedite the reintegration process, and data availability would be higher if the update activity were not suspended. Our techniques might provide this functionality.

In Reference [26], the performance of the reintegration process is studied for a single server and a single client at a time. If the reason of disconnection is a failure of the communication network or the central file server, it is likely that several clients will reconnect when the failure is repaired. The logs from different clients should be processed in parallel, but our techniques must be extended to deal with this case. In this thesis, we assume that the independent log streams reflect update activity on disjoint parts of the database, so that there are no conflicts between operations that appear on different logs. The logs from different clients of the same file server may contain conflicting accesses to common files. Such conflicts must be detected and reported, and they prevent independent installation of each log.

Bibliography

[1] R. K. Abbott and H. Garcia-Molina. Reliable distributed database management. *Proceedings of the IEEE*, 75(5):601–620, May 1987.

[2] R. Agrawal. A parallel logging algorithm for multiprocessor database machines. In *4th International Workshop on Database Machines*, pages 256–276, Grand Bahama Island, March 1985. Springer Verlag.

[3] R. Bayer. Consistency of transactions and random batch. *ACM Transactions on Database Systems*, 11(4):397–404, December 1986.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[5] D. Burkes and K. Treiber. Design approaches for real time recovery. Presentation at the Third International Workshop on High Performance Transaction Systems, Pacific Grove, CA, September 1989.

[6] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, February 1985.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[8] R. A. Cruz. Data recovery in IBM Database 2. *IBM Systems Journal*, 23(2):178–188, 1984.

[9] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD*, pages 1–8, Boston, June 1984.

[10] W. Finkelstein and M. Cappi. Experiences with large networks of computers. Presentation at the International Workshop on High Performance Transaction Systems, Pacific Grove, CA, September 1985.

[11] H. Garcia-Molina, R. Hagmann, and C. A. Polyzois. Two epoch algorithms for disaster recovery. In *16th Int'l Conf. on Very Large Data Bases*, pages 222–230, Brisbane, Australia, August 1990.

[12] H. Garcia-Molina, N. Halim, R. P. King, and C. A. Polyzois. Overview of disaster recovery for transaction processing systems. In *IEEE 10th ICDCS*, pages 286–293, Paris, May 1990.

[13] H. Garcia-Molina, N. Halim, R. P. King, and C. A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Transactions on Database Systems*, 16(2):338–368, June 1991.

[14] H. Garcia-Molina and C. A. Polyzois. Issues in disaster recovery. In *IEEE Compcon*, pages 573–577, San Francisco, February 1990.

[15] H. Garcia-Molina and C. A. Polyzois. A generalized disaster recovery model and algorithm. In *Fourth International Workshop on High Performance Transaction Systems*, Asilomar, CA, September 1991.

[16] H. Garcia-Molina and C. A. Polyzois. Processing of read-only queries at a remote backup. Technical Report CS-TR-354-91, Department of Computer Science, Princeton University, December 1991.

[17] H. Garcia-Molina and C. A. Polyzois. Evaluation of remote backup algorithms for transaction processing systems. In *ACM SIGMOD Int'l Conf. on Management of Data*, San Diego, CA, June 1992. To appear.

[18] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

[19] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *Data Engineering Bulletin*, 8(2):3–10, June 1985.

[20] J. N. Gray. Notes on database operating systems. In R. Bayer et al., editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer Verlag, Berlin, 1979.

[21] J. N. Gray. Why do computers stop and what can be done about it? Presentation at the Fifth Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, January 1986.

[22] J. N. Gray and M. Anderton. Distributed computer systems: Four case studies. *Proceedings of the IEEE*, 75(5):719–726, May 1987.

[23] J. N. Gray and A. Reuter. Transaction processing. Course Notes from CS#445, Stanford University, Spring Term, 1988.

[24] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1991.

[25] IBM. *IMS/VS Extended Recovery Facility (XRF): General Information*, March 1987. Document Number GG24-3150.

[26] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[27] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *16th Int'l Conf. on Very Large Data Bases*, pages 95–106, Brisbane, Australia, August 1990.

[28] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, New York, 1986.

[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[30] J. Lyon. Design considerations in replicated database systems for disaster protection. In *IEEE Compcon*, pages 428–430, San Francisco, February 1988.

[31] J. Lyon. Tandem's remote data facility. In *IEEE Compcon*, pages 562–567, San Francisco, February 1990.

[32] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Montreal, August 1983.

[33] C. Mohan and I. Narang. Solutions to hot spot problems in a shared disks transaction environment. In *Fourth International Workshop on High Performance Transaction Systems*, Asilomar, CA, September 1991.

[34] C. Mohan, K. Treiber, and R. Obermarck. Algorithms for the management of remote backup databases for disaster recovery. IBM Research Report RJ 7885R, IBM Almaden Research Center, San Jose, CA, June 1990.

[35] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

[36] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(3):271–287, 1986.

[37] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[38] D. J. Rosenkrantz. Dynamic database dumping. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 3–8, Austin, TX, May 1978.

[39] R. D. Schlichting and F. D. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[40] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Winter 1990 USENIX*, pages 313–323, Washington, D.C., January 1990.

[41] D. Skeen. Nonblocking commit protocols. In *ACM SIGMOD Conf. on Management of Data*, pages 133–147, Orlando, FL, June 1982.

[42] S. H. Son and A. K. Agrawala. Distributed checkpointing for globally consistent states of databases. *IEEE Transactions on Software Engineering*, 15(10):1157–1167, October 1989.

[43] C. Staelin and H. Garcia-Molina. Clustering active disk data to improve disk performance. Technical Report CS-TR-283-90, Department of Computer Science, Princeton University, September 1990.

[44] Tandem Computers. *Remote Duplicate Database Facility (RDF) System Management Manual*, March 1987.

[45] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, NJ, 1988.