Consistency in a Partitioned Network: A Survey

Susan B. Davidson*
Hector Garcia-Molina*
Dale Skeen

TR 84-617
June 1984

Susan B. Davidson
University of Pennsylvania

Hector Garcia-Molina
Princeton University

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This material is based upon work partially supported by the National Science Foundation under Grant ECS-8303146.

Consistency in a Partitioned Network: A Survey

Susan B. Davidson *

University of Pennsylvania

Hector Garcia-Molina *

Princeton University

Dale Skeen *

Cornell University

ABSTRACT

Recently, several strategies for transaction processing in partitioned distributed database systems with replicated data have been proposed. We survey these strategies in light of the goal of maintaining reliability. Extensions and combinations are then discussed, and guidelines for the selection of a strategy for a particular application are presented.

* This material is based upon work partially supported by the National Science Foundation under Grant ECS-8303146.

INTRODUCTION

In distributed database systems, data is often replicated to achieve reliability. Although reliability is difficult to define precisely, one aspect is *availability*: copies of data-items should remain accessible even in the presence of failures. Another aspect of reliability is *correctness*: copies of data-items must be mutually consistent, and the values of data-items must "make sense." Correctness is normally defined by the concurrency control mechanism and integrity constraints used by the system.

The most disruptive failures in a distributed database system are communication failures that partition the system into subsets of nodes that can no longer communicate. Whenever such failures occur, the correctness or consistency of the data is threatened. For example, suppose an Airline Reservation System implemented by a distributed database splits into two groups P_1 and P_2 due to a failure in the communication network. If at the time of the failure all the nodes have one seat remaining for PAN AM 537, and reservations are made in both P_1 and P_2, correctness has been violated: who should get the last seat? There should not be more seats reserved for a flight than physically exist on the plane. (Some airlines do not implement this constraint and allow overbookings.) Since data has been replicated to achieve reliability, there should be a strategy for maintaining consistency in the face of such failures. Furthermore, this strategy should be part of the design process of the system rather than a concession to the inevitable once the system is operating.

Not all partitionings are caused by failures in the communication network. Site failures can partition many networks (e.g., the ARPANET), and slow responses from certain sites can cause the network to appear partitioned even when it is not. In many cases the cause or extent of a partitioning (or apparent partitioning) can not be discerned by the sites themselves, further complicating the design of an appropriate processing strategy. Moreover, in some systems, partitionings are part of the normal operation of the system and can often be anticipated. Mobile nodes, such as airplanes and ships, may be able to communicate only during predictable intervals. In a few cases partitioned operation is the default mode, and communication is established between nodes only when necessary.

As far back as 1977, Rothnie and Goodman in their well-known survey paper [ROGO77] identified partitioned operation as one of the important and challenging open issues in distributed data management. Since then our understanding of the problem has increased dramatically, while numerous and diverse solutions have been proposed. This paper surveys several of the more general solutions, as well as discussing current research trends.

Section 1 discusses the goals and trade-offs involved in designing a strategy. Section 2 defines the database model. Sections 3 and 4 survey the current solutions for partitioned operation, and suggest extensions and combinations. Section 5 discusses the problem of partitioning during the atomic commit phase of transaction processing. Guidelines for the selection of a strategy are presented in section 6, along with suggestions for future research.

Although our discussion is couched within a database context, most results have more general applications. In fact, the only essential notion in many cases is that of a transaction. Hence, these strategies are immediately applicable to mail systems, calendar systems, object-oriented systems—applications using transactions as their underlying model of processing. Only a few strategies use concepts indigenous to database systems.

1. BASICS

1.1. Issues

When designing a system which is reliable when partitioned, the two competing goals of availability—the normal function of the system should be disrupted as little as possible—and correctness—data must be correct when recovery is complete—must somehow be met. These goals are not independent; hence, trade-offs are involved.

Correctness can be achieved simply by suspending operation in all but one of the partition groups and forwarding updates at recovery; however availability has been severely compromised.

In some applications, this is not acceptable. Typically in these applications either partitions occur frequently or occur at critical moments when access to the data is imperative. For example, in the Airline Reservation System it may be too expensive to have a high connectivity network and partitions may occur periodically. Many transactions are executed each second (TWA's centralized reservations system [GISP84] estimates 170 transactions per second at peak time), and each transaction that is not executed may represent the loss of a customer. In a military command and control application, a partition can occur because of an enemy attack, and it is precisely at this time that we do not want transaction processing halted.

On the other hand, availability can be achieved simply by allowing all nodes to process transactions "as usual" (note that transactions can only execute if the data they reference is accessible). However, correctness may now be compromised. Transactions may produce "incorrect" results (e.g. reserving more seats than physically available) and the databases in each group may diverge. In some applications, such "incorrect" results are unacceptable. For example, a transaction which effectively hands $1,000,000 to a customer may not be able to be undone.

Since it is clearly impossible to satisfy both goals simultaneously, one or both must be relaxed to some extent depending on the application's requirements. Relaxing availability is fairly straightforward; one simply disallows certain transactions at certain sites. However, correctness does not seem to be such a relative term; a database is either correct or incorrect. Relaxing the definition of correctness has been addressed in unpartitioned systems in the interest of increasing throughput; results here are applicable to the partitioned scenario.

1.2. Anomalies

What does correct processing mean in a distributed database system? Before defining the concept formally, let us look at a couple of examples of anomalies caused by the concurrent execution of transactions in different partitions. The database, which will serve as a source of examples throughout the survey, contains a checking account and a savings account for a certain customer. A copy of each account is stored at both Site A and Site B, and a communications failure has isolated the two sites.

Figure 1 shows the result of executing a checking withdrawal at A (for $100) and two checking withdrawals at B (totaling $100).

SITE A	**SITE B**
Checking: $100 **Savings: $200**	**Checking: $100** **Savings: $200**
Checking := Checking – $100	*Checking := Checking – $25* *Checking := Checking – $75*
Checking: $ 0 **Savings: $200**	**Checking: $ 0** **Savings: $200**

Fig. 1 An anomaly due to write-write conflicts.

Although the resulting copies of the checking account contain the same value, we know intuitively that the actions of the system are incorrect—the account owner extracted $200 from an account containing only $100. The example illustrates a *write-write conflict* occurring between transactions executing in different partitions.

An interesting aspect of this example is that the resulting database is mutually consistent in the sense that all copies of the checking account have the same value. Mutual consistency is not a sufficient condition for correctness in a database system, although it commonly used the correctness criterium for replicated file systems. It is also not a necessary condition: consider the example where A executes the $100 withdrawal while B does nothing. Although the resulting copies of the checking account contain different values, the resulting database is correct if the system recognizes that the value in A's copy is the most recent one.

A different type of anomaly is illustrated in Figure 2, where the semantics of the checking withdrawal allows the account to be overdrawn as allowed as long as the overdraft is covered by funds in the savings account (i.e., $checking + savings \geq 0$). In the execution illustrated, however, these semantics are violated: $400 is withdrawn, whereas the accounts together contain only $300. The anomaly was not caused by write-write conflicts (none existed since the transactions updated

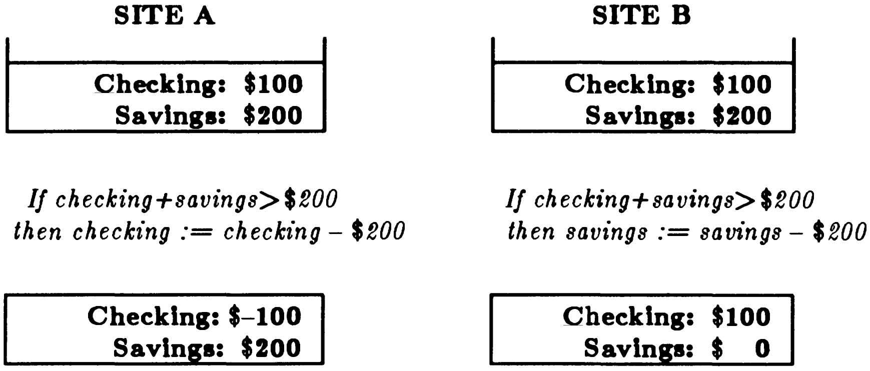


Fig. 2 An anomaly due to read-write conflicts.

different accounts), but instead by *read-write* conflicts resulting from one transaction reading the account changed by the other.

Read-write conflicts and write-write conflicts are two potential sources of inconsistencies in a partitioned system, but they are not the only sources—more will be identified in later sections. Nor do they always cause inconsistencies. For example, if the savings withdrawal in Figure 2 is changed to a deposit, the intended semantics of the database would not be violated. However, the above examples are typical of the anomalies that can occur if conflicting transactions are executed in different partitions.

2. BASIC CONCEPTS

2.1. Database Model

A database consists of a set of *logical data items* (hereinafter, just data items). The granularity of these items is not important: the items could be files, records, or relations. The *state* of the database is an assignment of values to these data items. A *transaction* is a program that issues read and write operations on the data items. In addition, a transaction may have effects that are external to the database, such as dispensing money or displaying results on a user's terminal. The items read by a transaction constitute its *readset;* the items written, its *writeset*. A *readonly transaction* neither issues write requests nor has external effects.

Transactions are assumed to be correct. More precisely, *a transaction, when executed alone, transforms an initially consistent database state into another consistent state* [TGGL82].

The database system is responsible for executing transactions so that they appear to users as indivisible, isolated actions on the database. This property, referred to as *atomic* execution, is achieved by guaranteeing the following properties:

(1) Each transaction is an "all or nothing" operation: either all of its writes and external operations are performed or none are performed. (In the former case the transaction is said to be *committed;* in the latter case, *aborted.*) The property is known as *atomic commitment.*

(2) If several transactions execute concurrently, they affect the database as if they executed serially in some order. The execution is then said to be *serializable.*

The first property is established by the commit and recovery algorithms of the database system; the second, by the concurrency control algorithm.

Given a serializable execution, an equivalent serial order of execution is called a *serialization order*. This order defines the apparent or logical order of execution (which may differ from the temporal order of execution). It need not be unique.

Two transactions *conflict* if and only if they operate on a common data item and at least one of the operations is a write. The order of execution of the two transactions is significant (i.e., reflected in the database state) only if they conflict. If one reads and the other writes, it is a *read-write conflict;* if they both write, a *write-write conflict.* (These definitions are the same as the ones given in Section 1. Readers may recognize that the examples of that section illustrate *non*serializable executions.)

Atomic transaction execution together with the consistency-preserving assumption imply that the execution of any set of transactions transforms a consistent initial database state into a consistent state. (This follows from a simple induction argument on the number of transactions.) As a correctness criteria, atomicity is appealing for its simplicity and generality: it is an operational constraint independent of the semantics of the data being stored and of the transactions manipulating it. It is also appealing because it corresponds to most users' intuitive model of processing—that of sequential processing.

Some systems allow additional correctness criteria to be expressed in the form of *integrity constraints*. Unlike atomicity, these are semantic constraints. They may range from simple constraints—e.g. the balance of checking account must be nonnegative—to elaborate ones that relate the values of many data items. In systems enforcing integrity constraints, a transaction is committed only if its execution is atomic and its results satisfy the integrity constraints.

In a distributed implementation of a database system, the data items are stored at multiple processing sites connected by a communications network. Sites can communicate only through the network—there is no shared memory. The distributed database is *replicated* if at least some of the items are physically stored at more than one site. The stored representations of an item are called its *copies*.

A generally accepted philosophy is that the user's[1] view of the database should not depend on the form of implementation—whether the database is centralized or distributed, nonreplicated or replicated. Hiding distribution from the user is known as *network transparency;* hiding replication, *replication transparency*. Replication transparency, in particular, is subtle to implement: accesses to an item must be mapped into accesses on its copies so as to appear that a single copy of the item is being manipulated (this is known as *one-copy equivalence*). Although there are several techniques for maintaining one-copy equivalence, for purposes of exposition we will assume the most intuitive: each write on an item is implemented as a write on all of its copies. Hence, copies are always kept mutually consistent.[2] (Other methods for one-copy equivalence are contained in GIFF79, STON79, and GSCDFR83. A good discussion of the requirements for maintaining one-copy equivalence in the presence of failures can be found in BEGO83.)

[1]The term "user" is used broadly, referring to either a human user or an application program making requests on the database.

[2]As in the introduction, we interpret "mutual consistency" to mean that the copies of an item contain the same value. This is the narrowest interpretation of several uses of the term that appear in the literature. Some authors use mutual consistency synonymously with one-copy equivalence.

The literature on the model and problems discussed above is extensive. The transaction concept was first introduced in EGLT76. A single-site recovery algorithm is presented in GMBLL81. Single-site concurrency control algorithms are too numerous to list, but two influential proposals are two-phase locking [EGLT76] and optimistic concurrency control [KURO81]. The seminal paper on serializability theory is PAPA79. BLAU81 discusses the enforcement of integrity constraints. GRAY78 contains an in-depth treatment of many issues in the implementation of a database system.

For non-partitioned, distributed database systems, concurrency control algorithms are surveyed in BEGO81 and KOHL81. Atomic commitment protocols are discussed in GRAY78, HASH80, and SKEE82b.

2.2. Partitioned Operation

As mentioned in the introduction, we are interested in partitioned networks, where the communication connectivity of the system is broken, by failures or by anticipated communication shutdowns, dividing the network into isolated subnetworks. Each subnetwork is called a *partition.* We assume that any two sites in the same partition can communicate and any two sites in different partitions can not. If the partitioning is due to failures, we assume that the failed components have simply stopped. Not considered are Byzantine failures, where components can act arbitrarily and even maliciously, or failures allowing one-way but not two-way communication between sites (e.g., a site's transmitter, but not its receiver, fails).

Network and replication transparency can not be maintained across partitions, but they can be maintained within a partition to the extent that the data items accessible within the partition appear to reside at a single site. (The algorithms accomplishing this are adapted from the ones for an unpartitioned network.) Except for the property that it can further partition, we can view a partition as a single site executing atomic transactions.

Among the problems introduced by processing transactions in multiple partitions, the simplest is the loss of mutual consistency between copies in different partitions whenever updates are allowed in either partition. As we saw in the introduction, this is not necessarily bad. However, it does introduce a problem: when two partitions are reconnected, updates made in one partition must be propagated to the appropriate copies in the other partition. This is easily solved by extra bookkeeping whenever the system partitions.

The best known and perhaps most interesting problem is that of maintaining serializability across all partitions. (This is the problem addressed by most papers on partitioned networks.) As illustrated by the last example of Section 1, nonserializable executions can occur even if each item is updated in only one partition and within each partition processing is serializable. Unlike the mutual consistency problem, there are no simple solutions or clearly superior approach. The problem of enforcing integrity constraints across partitions is similar to that of maintaining global serializability. Inconsistency may arise if an update in one partition invalidates an integrity check in another partition.

A problem of a different nature is the atomic commitment of transactions. The problem is not with transactions that are submitted after the system has partitioned—they will be atomically committed within one partition and their results propagated to other partitions after reconnection—but with the transactions in execution when the partitioning occurs. Since the commitment problem concerns only those transactions executing at the time of partitioning, the number of transactions affected is not dependent on the duration of the partitioning (unlike the problems above). If the partitioning can be anticipated, this problem can be avoided altogether. Note that it is present even if the database is not replicated (again, unlike the problems above).

Mutual consistency, serializability, integrity constraints, and atomic commitment are the elements of partial correctness in the traditional model of a database system. Ensuring these four properities then is the crux of the partitioned network problem. Our emphasis will be on serializability and, to a lesser extent, integrity constraints and methods for ensuring them when transactions execute in different partitions. Establishing mutual consistency after partition reconnection is simple and will not be discussed further. The atomic commitment problem is discussed briefly in a separate section.

2.3. Classification of Strategies

Strategies for reconciling inconsistencies introduced by the parallel execution of transactions in different partitions can be classified along two orthogonal dimensions.

The first dimension concerns the tradeoff between consistency and availability. At one extreme lies the *pessimistic strategies,* which prevent inconsistencies by limiting availability. (They operate under the pessimistic assumption that if an inconsistency can occur, it will occur.) These strategies differ primarily in the policy used to restrict transaction processing. Since they ensure consistency, it is straightforward to merge the results of individual partitions at reconnection time. The only activity required is the propagation of updates from copies in one partition to their counterparts in the other partitions.

At the other extreme lie the *optimistic strategies,* which do not limit availability. Any transaction may be executed in any partition containing copies of items in its readset and writeset. Hence, inconsistencies may be introduced. (These strategies operate under the optimistic assumption that inconsistencies, even if possible, rarely occur.) During reconnection, the system must first *detect* inconsistencies and then *resolve* them by, for example, undoing the effects of certain transactions. Optimistic strategies differ in how they detect and resolve inconsistencies. Although optimistic policies allow global inconsistences, transaction processing within each partition is consistent. Thus, no user staying within a single partition could detect an inconsistency.

The second dimension in the classification concerns the type of information used in determining correctness. *Syntactic* approaches use serializability as their sole correctness criteria and check serializability by examining readsets and writesets of the executed transactions. Hence neither the semantics of the transactions (i.e. how the read items are used to generate the result) nor the semantics of the data items are used in ascertaining correctness. Syntactic approaches are the natural extensions of general-purpose concurrency control algorithms, such as two-phase locking [EGLT76], which also use only syntactic information.

Semantic approaches use either the semantics of the transactions or the semantics of the database in defining correctness. Although this is somewhat of a "catch-all" category, there are two discernible subcategories. The first uses serializability as the correctness criteria but also uses the semantics of the transactions in testing serializability. The second abandons serializability altogether and instead defines correctness in terms of the contents of the database itself. Supposedly, the correctness criteria captures the semantics of the data stored in the database. Such semantic constraints fall outside of the traditional model discussed in the previous section.

3. SYNTACTIC APPROACHES

All approaches in this section use serializability as the correctness criteria and check serializability by comparing transactions' readsets and writesets. We assume that a correct concurrency control mechanism coordinates transaction execution within a partition; hence, transaction execution within a partition is serializable. We also assume that at the time of the partitioning all copies are mutually consistent and there are no in-progress transactions.

3.1. Optimistic Strategies

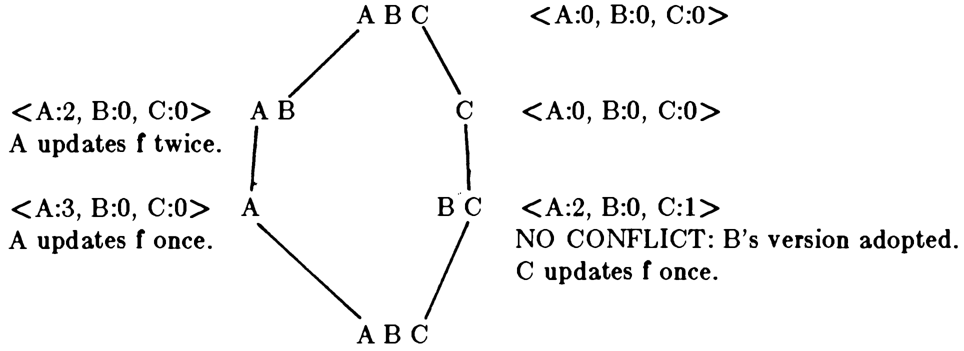
Version Vectors [PPR81]

Version vectors, proposed for use in the distributed operating system LOCUS [POPE81], detect write-write conflicts between copies of files. Each copy of a file f has associated with it a *version vector* consisting of a sequence of n pairs, where n is the number of sites at which f is stored. The i^{th} vector entry $(S_i:v_i)$ counts the number v_i of updates to f originating at site S_i. Conflicts are detected by comparing version vectors.

Vector $\mathbf{v}$ is said to *dominate* vector $\mathbf{v}'$ if $\mathbf{v}$ and $\mathbf{v}'$ are version vectors for the same file and $v_i \geq v_i'$ for $i=1, \cdots, n$. Intuitively, if $\mathbf{v}$ dominates $\mathbf{v}'$, the copy with vector $\mathbf{v}$ has seen a superset of the updates seen by the copy with vector $\mathbf{v}'$. Two vectors are said to *conflict* if neither dominates. In this case, the copies have seen different updates. For example, <A:3, B:4, C:2> dominates <A:2, B:1, C:2> since $3>2$, $4>1$ and $2=2$, but <A:3, B:1, C:2> and <A:2, B:4, C:2> conflict since $3>2$ but $1<4$.

When two sites discover that their version vectors for f conflict, an inconsistency has been detected. How to resolve the inconsistency is left up to the system administrator.

EXAMPLE: Consider the following partition graph for file f. Sites A, B and C initially have the same version of f. The system then partitions into groups AB and C, and A updates f twice. Hence both A and B have version vectors of <A:2, B:0, C:0>, while C is <A:0, B:0, C:0>. Site B then splits off from site A and joins site C. Since C did not update f and B has the current version, there is no conflict (<A:2, B:0, C:0> dominates <A:0, B:0, C:0>), and B's version (and vector) is adopted for the new group BC. During this new partition failure, A updates its version of f once, making group A's version vector <A:3, B:0, C:0>, and C updates its version of f once, making group BC's version vectors <A:2, B:0, C:1>. When groups A and BC now combine, there is a conflict and neither of <A:2, B:0, C:1> or <A:3, B:0, C:0> dominates the other.



CONFLICT: 3>2, 0=0, but 0<1.
Manual assistance required.

Version vectors detect write-write conflicts only. Read-write conflicts can not be detected because the files read by a transaction are not recorded. Hence, the approach works well for transactions accessing a single file, which are typical in many file systems, but not for multi-file transactions, which are common in database systems.

EXAMPLE: Consider applying version vectors to the banking example of Figure 1, where communication between sites A and B fail. During the failure, the transaction executed at A and updates the checking balance based on the value of saving balance; the transaction executed at B, updates the savings balance based on checking balance. No conflict will be detected, even though the above is clearly not serializable.

checking balance

A B <A:0, B:0>

<A:1, B:0> A B <A:0, B:0>

A B

NO CONFLICT detected
A's version adopted.
<A:1, B:0>

savings balance

A B <A:0, B:0>

<A:0, B:0> A B <A:0, B:1>

A B

NO CONFLICT detected
B's version adopted.
<A:0, B:1>

To extend the version vectors algorithm so that read-write conflicts are detectable, readset and writes of transactions must be logged. This leads to an algorithm very similar to the one presented in the next section.[3]

The Optimistic Protocol [DAVI82]

The optimistic protocol detects inconsistencies by using a precedence graph, which models the necessary ordering between transactions. Precedence graphs, which are used to checking serializability across partitions, are adapted from serialization graphs [PAPA79], which are used to check serializability within a site. In the following we assume that the readset of a transaction contains its writeset. (The reason for this assumption is to avoid certain NP-complete problems in checking serializability.)

In order to construct the precedence graph, each partition maintains a log, which records the order of reads and writes on the data items. From this log, the readsets and writesets of the transactions and a serialization order on the transactions can be deduced. (A serialization order exists since, by assumption, transaction execution within a partition is serializable.) For partition i, let $T_{i1}, T_{i2}, \cdots, T_{in}$ be the set of transactions, in serialization order, executed in i.

[3]Historical note: such an extension was proposed in PARA82. Their conflict detection algorithm, however, is incorrect: it does not detect all inconsistencies and falsely detects inconsistencies.

The nodes of the precedence graph represent transactions; the edges, interactions between transactions. The first step in the construction of the graph is to model interactions between transactions in the same partition. Two types of edges (interactions) are identified:

(a) *(Data) Dependency Edges*[4] ($T_{ij} \rightsquigarrow T_{ik}$): these edges represent the fact that one transaction T_{ik} read a value produced by another transaction T_{ij} in the same partition $(WRITESET(T_{ij}) \cap READSET(T_{ik}) \neq \emptyset,\ j<k)$

(b) *Precedence Edges* ($T_{ij} \longrightarrow T_{ik}$): these edges represent the fact that one transaction T_{ij} read a value that was later changed by another transaction T_{ik} in the same partition $(READSET(T_{ij}) \cap WRITESET(T_{ik}) \neq \emptyset,\ j<k)$

A dependency edge from T_{ij} to T_{ik} indicates that the output of T_{ij} influenced the execution of T_{ik}. The meaning of a precedence edge T_{ij} from T_{ik} is more subtle: T_{ik} does not influence T_{ij} only because T_{ij} executed before it. In both cases, an edge from T_{ij} to T_{ik} indicates that the order of execution of the two transactions is reflected in the resulting database state. Note that the graph constructed so far must be acyclic since the orientation of an edge is always consistent with the serialization order.

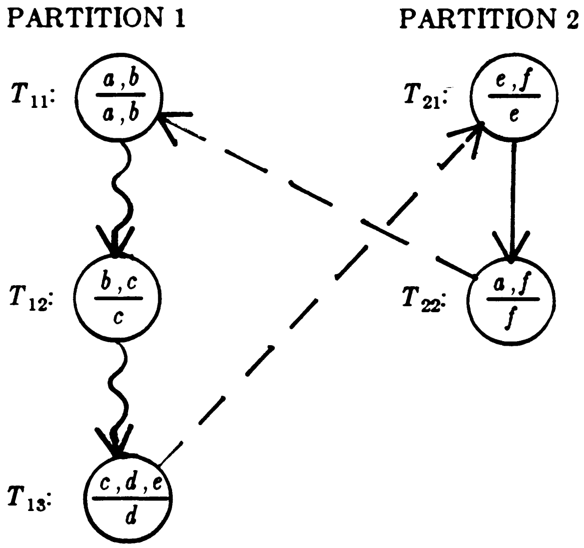
To complete the precedence graph, conflicts between transactions in different partitions must be represented. A new type of edge is defined for this purpose:

(c) *Interference Edges* ($T_{ij} \dashrightarrow T_{lk},\ i \neq k$): these edges indicate that T_{ij} read an item that is written by T_{lk} in another partition $(READSET(T_{ij}) \cap WRITESET(T_{lk}) \neq \emptyset,\ j<k)$.

The meaning of interference edge is the same as a precedence edge: an interference edge from T_{ij} to T_{lk} indicates that T_{ij} logically "executed before" T_{lk} since it did not read the value written by T_{lk}. An interference edge signals a read-write conflict between the two transactions. (A write-write conflict manifests as a pair of read-write since each transaction's readset contains its writeset.)

[4] In [DAVI82], these edges are called *ripple edges*.

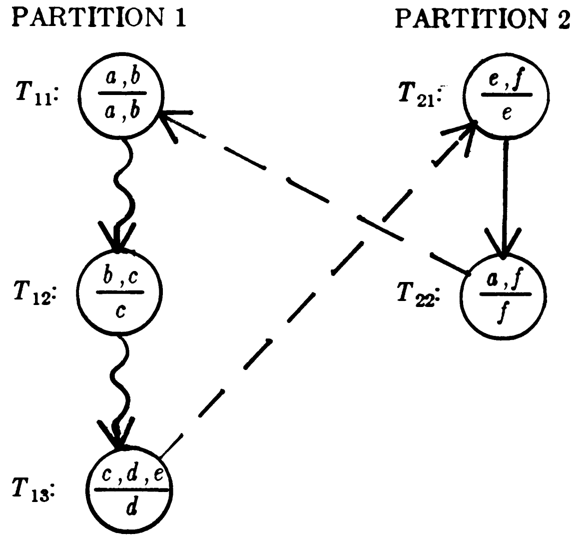
EXAMPLE: Suppose the serial history of transactions executed in P_1 is $\{T_{11}, T_{12}, T_{13}\}$, and that of P_2 is $\{T_{21}, T_{22}\}$. The precedence graph for this execution is given below, where the readset of a transaction is given above the line and the writeset below the line. (Thus, transaction T_{12} reads b,c and writes c.)



Intuitively, cycles in the precedence graph are bad: if T and T' are in a cycle then the database reflects the results of T executing before T' and of T' executing before T—a contradiction. Conversely, the absence of cycles is good: *the precedence graph for a set of partitions is acyclic if and only if the resulting database state is consistent* [DAVI82]. An acyclic precedence graph indicates that the transactions from both groups can be represented by a single serial history, and the last updated copy of each data-item is the correct value. A serialization order for the transactions can be obtained by topologically sorting the precedence graph.

Inconsistencies are resolved by rolling back (undoing) transactions until the resulting subgraph is acyclic. When a transaction is rolled back, transactions connected to it by dependency edges must also be rolled back, since these transactions read the values produced by the selected transaction. Hence rolling back one transaction may precipitate the rolling back of many, a problem known as *cascading rollbacks*. Transactions connected to a rolled back transaction by precedence edges are not rolled back since they did not read the results of the rolled-back transaction. In the above example, if T_{11} is selected, then T_{12} and T_{13} must also be selected. However, simply selecting T_{13}, T_{21}, or T_{22} also breaks the cycle and involves only one transaction.

EXAMPLE: Suppose the serial history of transactions executed in P_1 is $\{T_{11}, T_{12}, T_{13}\}$, and that of P_2 is $\{T_{21}, T_{22}\}$. The precedence graph for this execution is given below, where the readset of a transaction is given above the line and the writeset below the line. (Thus, transaction T_{12} reads b,c and writes c.)



Intuitively, cycles in the precedence graph are bad: if T and T' are in a cycle then the database reflects the results of T executing before T' and of T' executing before T—a contradiction. Conversely, the absence of cycles is good: *the precedence graph for a set of partitions is acyclic if and only if the resulting database state is consistent* [DAVI82]. An acyclic precedence graph indicates that the transactions from both groups can be represented by a single serial history, and the last updated copy of each data-item is the correct value. A serialization order for the transactions can be obtained by topologically sorting the precedence graph.

Inconsistencies are resolved by rolling back (undoing) transactions until the resulting subgraph is acyclic. When a transaction is rolled back, transactions connected to it by dependency edges must also be rolled back, since these transactions read the values produced by the selected transaction. Hence rolling back one transaction may precipitate the rolling back of many, a problem known as *cascading rollbacks*. Transactions connected to a rolled back transaction by precedence edges are not rolled back since they did not read the results of the rolled-back transaction. In the above example, if T_{11} is selected, then T_{12} and T_{13} must also be selected. However, simply selecting T_{13}, T_{21}, or T_{22} also breaks the cycle and involves only one transaction.

A selection algorithm should strive to minimize some cost function, for example, the number of rolled-back transactions, or the sum of the weights of the rolled-back transactions (where the assignment of weights can be application dependent). Unfortunately minimizing either the number of transactions or the sum of their weights is an NP-Complete problem [DAVI82]. Hence, heuristics must be used.

The most promising heuristics use the following observation: breaking all two-cycles in a precedence graph tends to break almost all cycles. Two cycles can be broken optimally using a polynomial algorithm. After the two-cycles have been broken, the few remaining cycles can be broken by a greedy algorithm, one that repetitively selects the lowest weight transaction involved in a cycle. Simulation studies have shown that such heuristics work very well, out-performing all other strategies tested [DAVI82].

The performance of the optimistic protocol is studied in DAVI82. A probabilistic model is developed that yields a formula for estimating rollback rate given the number of transactions, a model of the average transaction, and the size of the database. Simulation results in the same paper yield additional insight into rollback rates. These studies indicate that the optimistic protocol performs best when:

(1) a small percentage of items are updated during the partitioning, and

(2) few transaction have large writesets.

Whenever (1) holds, the probability that a given transaction will be rolled back depends more on the size of its writeset than its readset. Regarding (2), not only is the occasional large transaction more likely to conflict with another transaction, but in addition its rollback is likely to cause other rollbacks. Consequently, the rollback rate is quite sensitive to variance in transaction size.

3.2. Pessimistic Approaches

The first group of pessimistic strategies, primary site (copy), tokens, and voting, were initially proposed as distributed concurrency control mechanisms. However, they can also be used to prevent conflicts between transactions when the network partitions. The last approach, designed specifically for partitioned networks, strives to increase availability by exploiting known characteristics of the workload.

Primary Site, Copy [ALDA76]

Originally presented as a resilient technique for sharing distributed resources, this approach suggests that one copy of an item be designated the primary and as such be responsible for that item's activity. Other copies of the item are used as back-ups in case the primary fails, in which case a new primary is elected. In a partitioned system, only the partition containing the primary copy is allowed to access the item.

This approach works well only if site failures are distinguishable from network failures. Whenever it is uncertain whether the primary failed or the network failed, the back-ups can not safely process transactions.

Tokens [MIWI82]

This approach is very similar to that above except that the primary copy of an item can change for reasons other than site failure. Each item has a token associated with it, permitting the bearer to access the item. In the event of a network partition, only the group containing the token will be able to access the item.

The major weakness with this scheme is that accessibility is lost if the site with the token or the communication medium fails.

Voting [GIFF79]

The first voting approach was the majority consensus algorithm described in THOM79. What we now describe is the generalization of that algorithm proposed by Gifford [GIFF79].

In this approach, every copy of a replicated item is assigned some number of votes. Every transaction must collect a read quorum of r votes to read an item, and a write quorum of w votes to write an item. Quorums must satisfy two constraints:

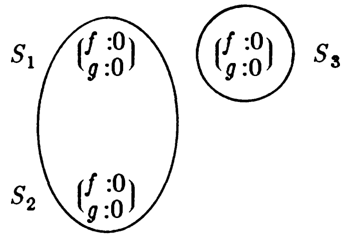
(1) $r + w$ exceeds the total number of votes v assigned to the item, and

(2) $w > \frac{v}{2}$.

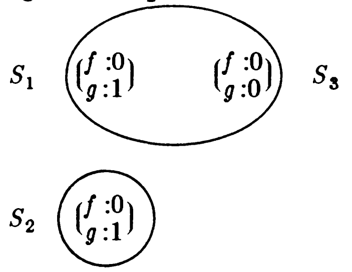
The first constraint ensures that there is a non-null intersection between every read quorum and every write quorum. Any read quorum is therefore guaranteed to have a current copy of the item. (Version numbers are used to identify the most recent copy.) In a partitioned system, this constraint guarantees that an item can not be read in one partition and written in another. Hence, read-write conflicts can not occur between partitions.

The second constraint ensures that two writes can not happen in parallel or, if the system is partitioned, that writes can not occur in two different partitions. Hence, write-write conflicts can not occur between partitions.

EXAMPLE: Suppose sites S_1, S_2 and S_3 all contain copies of items f and g, and that a partition P_1 occurs, isolating S_1 and S_2 from S_3. Initially, f=g=0, each site has 1 vote for each of f and g, and r=w=2 for both f and g.



During the partitioning, transaction T_1 wishes to update g based on values read for f and g. Although it cannot be executed at S_3, it is executed at S_1, and the new value g=1 is propagated to S_2.



Now suppose P_1 is repaired, and a new failure P_2 isolates S_1 and S_3 from S_2. During this new failure, transaction T_2 wishes to update f based on values read for f and g. It cannot be executed at S_2 since it cannot obtain a read quorum for g, or read and write quorums for f. However, it can be executed at S_3. Using the most recent copy of g=1 (obtained by reading copies at both S_1 and S_3 and taking the latest version) T_2 computes the new value f=1 and propagates the new value to S_1.

Notice that the above example reduces to a majority vote [THOM78] since each copy has exactly one vote and r and w are a simple majority

Varying the weight of a vote can be used to reflect the needed accessibility-level of an item. For example, in a banking application, a customer may use certain branches more frequently than other branches. Suppose there are 5 branches of the bank, and that the customer uses branches 1, 2, and 3 with equal frequency, but never goes to branches 4 or 5. Assigning r=w=2 and his account at branches 1, 2 and 3 a vote of 1 but 0 elsewhere would reflect this usage pattern.

The quorum algorithm differs from those previously discussed in two important ways. First, by choosing $r < v/2$, it is possible for an item to be read-accessible in more than one partition, in which case it will be write-accessible in none. Read-accessibility can be given a high priority by choosing r small. Second, the algorithm does not distinguish between communication failures, site failures, or just slow response. A serious weakness of the previous schemes is that availability is severely compromised if a distinction can not be made.

A weakness of the quorum scheme is that reading an item is fairly expensive. A read quorum of copies must be read in this scheme, whereas a single copy suffices for all other schemes.

Class Conflict Analysis [SKWR84]

The pessimistic strategies discussed so far enforce the following two constraints:

(1) An item can be updated in only one partition.

(2) No item can be written in one partition and read in another.

The first constraint ensures the absence of write-write conflicts between partitions, and the second, read-write conflicts.

Any strategy enforcing the above constraints prevents inconsistencies. This can shown formally by examining the precedence graphs (see section entitled "Optimistic Protocol") of transaction executions allowed under these constraints. The constraints ensure that no interference edges are present in these graphs. The absence of interference edges implies acyclicity of the graph and acyclicity implies serializability.

The complete absence of interference edges suggests that these constraints are stronger than necessary. This is indeed the case, as the next example illustrates.

EXAMPLE: Consider again the banking example in Figure 2 with an additional transaction—a saving deposit that increments the savings account by a user-supplied amount. Suppose sites A and B are partitioned and a checking withdrawal executes at A and a savings deposit at B:

SITE A	SITE B
Checking: $100 **Savings: $200**	**Checking: $100** **Savings: $200**
If checking+savings>$200 *then checking := checking – $200*	*Savings Deposit of $100*
Checking: $–100 **Savings: $200**	**Checking: $100** **Savings: $300**

Note: the result is serializable: the withdrawal precedes the deposit.

Even though the execution in the example is correct, it violates the second constraint; hence, it is allowed in none of the above strategies.

The goal of Class Conflict Analysis is to constrain transaction processing less than previous pessimistic approaches while tailoring the set of allowable transactions to the workload and to the semantics of the data. High priority or high volume transactions should be allowed whenever possible. Semantics may preclude certain types of transactions, and this should be exploited. For example, in an airline database, a transaction that reads the number of passengers and changes the amount of fuel is to be expected, but not one that reads fuel and changes passengers.

The approach assumes that transactions are divided into classes as proposed in BSR80. A class may be a well-defined transaction type, such as the savings withdrawal used in Figure 2, or it may be syntactically defined, e.g., the class containing all transactions reading and writing a subset of items a, b, and c.

Like transactions, classes are characterized by their readset and writesets. The readset (resp. writeset) of a class is the union of the readsets (resp. writesets) of all of its member transactions. As before, it is assumed that a class's readset contains its writeset, so that certain NP-complete problems are avoided. Two classes *conflict* if one's readset intersects the other's writeset. A class conflict indicates a *potential* read-write conflict between member transactions of the classes. A conflict may not actually occur because the transactions' readsets and writesets may be proper subsets of the classes' readsets and writesets.

The first step in the approach is the preliminary assignment of classes to partitions. This is normally performed in parallel by each partition at the time of partitioning, using prespecified assignment rules. A class is assigned to a partition whenever it is desirable to execute transactions from that class in the given partition. Note that this assignment reflects only what is desirable not necessarily what is safe. Classes may be assigned to more than one partition, and there may be conflicts between classes in different partitions.

The second step is to analyze the assignment and discover the class conflicts that can lead to nonserializable executions. The analysis uses a graph model that is similar to the one used in the optimistic protocol. Whereas the precedence graphs used in that protocol give the *actual* orderings between conflicting transactions, class conflict graphs give all *potential* orderings between conflicting classes. Defined below is a simplified version of the model presented in SKWR84.

A node of the *class conflict graph* represents the occurrence of a given class in a given partition. Edges are drawn between occurrences of conflicting classes according to the rules given below. Let C_i and C_j be classes such that $readset(C) \cap writeset(C_j)$ is not empty.

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

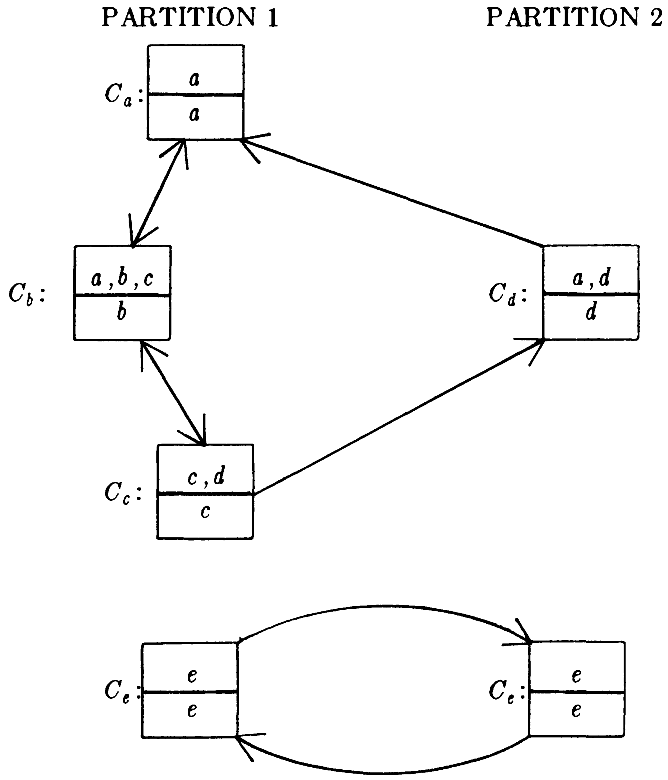
(2) If C_i and C_j are in different partitions, then a directed edge extends from C_i to C_j.

The direction of the edges indicate the possible logical orderings of transactions from conflicting classes. If two conflicting classes belong to the same partition, then the concurrency control algorithm will resolve conflicts between transactions from those classes. In doing so, the algorithm is free to choose which of the two conflicting transactions to execute first. Hence, the edge pair in

(1) means literally that the order is important but indeterminate a priori.

Between classes in different partitions the situation is different: The logical ordering between conflicting transactions is determined not by the concurrency control algorithm but by the fact that the values produced in one partition are not available for reading in another. In the case of classes C_i and C_j above, the transactions of C_i can not logically succeed those of C_j because C_i's transactions can not read the updates of C_j's transactions. The only order possible is that all transactions of C_i precede all transactions of C_j, as indicated by the single directed edge.

EXAMPLE: A class conflict graph for two partitions. Boxes are classes: readsets are shown above the line; writesets, below. Note that class C_e appears in both partitions.



The third step in the analysis is to identify those assignments that could lead to nonserializable executions. Cycles play a key role here; however, in contrast to previous graph models, not all cycles are bad. Among class occurrences in the same partition, cycles are both common, since in this case all edges are paired, and harmless, since the concurrency control algorithm operating in the partition will prevent nonserializable executions.

On the other hand, cycles spanning multiple (>1) partitions are not harmless, since there is no mechanism preventing them in an execution. Hence, *multipartition cycles indicate the potential for nonserializable executions*. In the example above, if transactions from classes C_a, C_b, and C_c execute in that order in partition 1 and a transaction from C_d executes in partition 2, the result is nonserializable. (This can be checked by constructing the precedence graph for the execution.) Note that not all executions of these transaction are nonserializable; in fact, any other order of execution of the transactions in partition 1 is serializable.

Whenever the preliminary class assignment yields a (multipartition) cyclic graph, further constraints on transaction processing must be imposed. The most straightforward approach is to delete classes from partitions until the class conflict graph is rendered multipartition acyclic. In the above example, one occurrence of class C_e must be deleted along with one of C_a, C_b, C_c, or C_d. Rendering a class conflict graph acyclic is similar to rendering a precedence graph acyclic, except for one major difference: when deleting a class occurrence, adjacent classes in the graph need never be deleted. Nonetheless, the problem of minimizing the weight of the set of deleted classes is still NP-complete, and the heuristics used for the optimistic protocol can be adapted for use here.

Although in this discussion we have assumed that the complete state of the network is known to all partitions, this assumption is not required in applying class conflict analysis. SKWR84 discusses some modifications to the basic algorithm when network status information is incomplete. In addition, the same paper discusses refinements that affords more availability than the the analysis presented here.

3.3. Integrity Constraint Checking

In the presentation of the above strategies, the problem of checking integrity constraints was ignored. This checking can be incorporated into syntactical approaches in a simple manner. Suppose that during the execution of transaction t the constraint I must be checked. In order to verify I the system must read all items referenced in I; therefore, these items can be considered part of the readset of t. In general, the readset of each transaction could be extended with all items referenced in the integrity constraints it must satisfy.

This solution, although simple, has a major weakness: it can greatly inflate the sizes of readsets. As a transaction's readset size increases, the probability of the transaction being executable in a partitioned system decreases. As an alternative, it may be desirable to curtail integrity checking during a partition, suspending, in particular, the checking of constraints accessing many items.

3.4. Discussion

Optimistic versus Pessimistic

One basis for comparing the two types of approaches is in terms of an appropriate cost model. The model should include *overhead*, the cost of *repairing* inconsistencies, and the cost of *lost opportunities*. In the following, costs common to all approaches, such as the propagation of updated values, are omitted.

Optimistic policies have two sources of overhead. The first is the log, which must be maintained while the system is partitioned, recording the readset and writeset of each transaction in order to construct the precedence graph and recording sufficient information to rollback transactions. Many database systems already maintain a log, called an *undo log,* for rolling back transactions in case of site failures or a transaction failures (e.g., deadlocks) [GMBLL81]. This same log can be used to roll back conflicting transactions in a partitioned system. However, in order to construct the graph, undo logs must be augmented with records of transactions' readsets (which are normally not recorded since they are not needed to roll back a transaction). This increases the complexity of the logging algorithms, but it does not significantly increase the cost of logging in most systems.

The second and most significant source of overhead is the conflict detection algorithm, which constructs the graph, checks the graph for cycles, and then selects transactions to roll back. Graph construction requires a single pass over the entire log, which can be quite expensive for a partition of long duration. The selection algorithm can be made arbitrarily expensive, depending on the quality of heuristics used. The best heuristics require time $O(N^{2.81})$ where N is the number of transactions. However, linear time heuristics often yield acceptable solutions.

The cost of repair in an optimistic approach is simply the rollback rate times the cost of rolling back a transaction. We have already discussed rollback rate. The rollback cost is often a significant fraction of the transaction's execution cost, and may, in fact, exceed the execution cost if the transaction has external side-effects (e.g., a customer may be entitled to compensation if her reservation is canceled). Consequently, the rollback rate must be kept reasonable small (certainly less than 20%) if optimistic approaches are to cost-effective.

The goal of optimistic approaches is to minimize lost opportunity, the cost associated with needlessly delaying a transaction. These costs can be substantial when user satisfaction is important as, for example, in a banking application. Lost opportunities still occur in these approaches because of the allocation of resources to transactions that are destined to be rolled back. Such transactions may displace valid transactions during the partitioning, and rolling them back may cause further delays after the partitions are reconnected. Still, for most applications, we speculate that other costs dominate.

Pessimistic approaches have no repair costs and, except for conflict class analysis, almost no overhead. Even in class conflict analysis, the overhead is likely to be substantially less than in an optimistic strategy, because although conflict analysis and conflict detection are procedurally similar, the number of predeclared classes in conflict analysis is likely to be substantially less than the number of transactions in conflict detection.

The major cost of a pessimistic approach is, of course, the cost of lost opportunities. Included in this cost are not only opportunities lost to real partitioning but also opportunities lost to "apparent" partitionings, for example, site failures that are indistinguishable from real partitionings. In many systems, apparent partitionings occur more frequently than real partitionings; therefore they must be included in any cost analysis.

In summary, the cost of an optimistic strategy is the overhead of conflict detection plus the repair cost, whereas the cost of a pessimistic strategy is the cost of opportunities lost to real and apparent partitionings. Unfortunately, except for repair costs, informed estimates for these costs are not easily obtained. No one has measured the overhead associated with any of the strategies, and the cost of lost opportunities is hard to quantify (although one component in a pessimistic strategy is the cost of underutilization of processing resources).

Combining Strategies.

Instead of using one strategy during a partitioning, strategies can be combined vertically over time. That is, the system could start out using one strategy and switch to another as circumstances dictate. For example, the rollback rate for the optimistic protocol increases linearly with time. (This means that the number of transactions rolled back increases quadraticly with time.) Since it is usually impossible to predict how long a partitioning will last, the database administator could set a ceiling on the rollback rate (say 10%). The system could then start out using the optimistic protocol and monitor the performance using the formula in [DAVI82] to estimate rollback rates. If this ceiling is reached, the system could switch to a more pessimistic approach, such as primary site, for the remainder of the failure.

Strategies can also be combined horizontally over time [SKEE82c]. One approach is to assign items different levels of consistency. Items in level 0 (the highest level) are immutable during a partitioning; items in level 1 are updated according to a pessimistic strategy; and items in level 2 are updated according to a optimistic strategy. Updates to level 1 items are globally consistent and guaranteed to persist, while updates to level 2 items are consistent within the partition but may not be globally consistent and, hence, are subject to rollback. Although a transaction may update items in only one level, it may read items of the same level and higher.

Another way to combine approaches horizontally is to divide transactions, instead of items, into groups. For each partition, transactions are divided into two groups: high priority transactions that can not be rolled back, and low priority transactions that can. Class conflict analysis is used to determine a group of high priority transactions for each partition. The low priority group for a partition consists of all transactions not writing an item read by a high priority transaction in the same partition. (A low priority transaction, though, can write an item read by a high priority transaction in a *different* partition.) When partitions are reconnected, the optimistic protocol is used to construct a precedence graph containing all transactions executed; however, only low priority transactions are liable to rollback. (An approach similar to this is used in APWI84.)

4. SEMANTIC APPROACHES

The first three approaches presented in this section illustrate three different ways of using semantics to increase availability. The first approach, log transformations, uses the standard notion of correctness, namely serializability, but uses the semantics of transactions in checking serializability. The second approach relaxes slightly the standard notion of serializability in order to enrich the set of transactions allowed in a partitioned system. The semantics of the application determine when serializability can be relaxed. The third approach, Data-Patch, abandons serializability as a correctness criterion altogether, using instead an application-specific definition of correctness. All three approaches are optimistic. As a matter of fact, to our knowledge, no one has suggested a pessimistic, semantic strategy, probably because semantics are usually introduced to increase availability, not to ensure correctness.

This section ends with a brief discussion of some other proposed ideas for increasing availability.

Log Transformations [BGRCK83]

This approach is similar to the optimistic protocol. During the partitioning, logs are kept of which transactions were executed and in what order. After reconnection, a rerun log is constructed which indicates what should be reflected as having happened during the failure. To achieve this, transactions in each group may have to be backed out and rerun. It differs in that transactions are pre-defined, and semantic properties of pairs of transactions are declared to avoid needlessly backing out and re-executing transactions. These properties can include commutativity ($T_i T_j = T_j T_i$) and overwriting ($T_i T_j = T_j$). There is also a notion of "absolute time" in each group during the failure so that transactions can no longer be merged based merely on their readsets and writesets, but must follow some agreed upon time ordering.

EXAMPLE: Suppose that during a partition, P_1 has executed T_2, T_4, T_6 and that P_2 has executed T_1, T_3, T_5, where the subscripts indicate the absolute timing of the transactions. The rerun log would be $T_1, T_2, T_3, T_4, T_5, T_6$. If we ignored any semantic properties of transactions, merging the database at P_1 would involve backing out transactions T_2, T_4, T_6 and reexecuting the rerun log. If we assume that backing out transaction T can be achieved by running an inverse transaction T^{-1}, then the entire merging operation at P_1 can be represented by the backout (or rollback) log $T_6^{-1}, T_4^{-1}, T_2^{-1}$ followed by the redo log. Similarly, the merge operation at P_2 involves executing the backout log $T_5^{-1}, T_3^{-1}, T_1^{-1}$ followed by the redo log. Let us call the combined backout, redo log the *merge log*.

If we know that T_1 commutes with T_2, then the merge log at P_1 can be reduced to

$$T_6^{-1}, T_4^{-1}, T_1, T_3, T_4, T_5, T_6$$

To see that the result of executing P_1's merge log is equivalent to the result of executing $T_1, T_2, T_3, T_4, T_5, T_6$ in order, consider the entire sequence of transactions executed by P_1 (that is, the original execution followed by the merge log):

$$T_2, T_4, T_6, T_6^{-1}, T_4^{-1}, T_1, T_3, T_4, T_5, T_6.$$

Since T_6, T_6^{-1} and T_4, T_4^{-1} are equivalent to the null transaction, the above is equivalent to

$$T_2, T_1, T_3, T_4, T_5, T_6.$$

And, by the commutativity of T_1 and T_2, this is equivalent to the desired sequence.

If in addition we know that T_1 and T_3 commute with T_4 and T_6, and that T_6 overwrites T_5, then the P_1 merge log can be further reduced to

$$T_1, T_3$$

(that is, after the partition we only have to run T_1, T_3 without backing out any transactions). At P_2, this same semantic information only reduces the merge log to

$$T_5^{-1}, T_3^{-1}, T_2, T_3, T_4, T_6.$$

The process of reducing in size the merge log is called log transformation. The process can be automated with the aid of a graph formalism presented in BGRCK83. With it, merge logs are represented as graphs, and each log transformation is represented as a graph transformation.

One advantage of the log transformation approach is that the merge processes at the sites are independent of each other. That is, as each site finds out about transactions that executed elsewhere, it can proceed to integrate them locally, regardless of what the other sites are doing. Thus, this approach may be useful in an environment where failures are common and communications unreliable.

Weak Consistency [GAWE82]

Garcia and Weiderhold argue in GAWE82 that conventional correctness criteria—in particular, serializability—may be stronger than needed for many readonly transactions. Since such transactions do not change the database state, their execution under a weaker correctness criterion can not generate an inconsistent state. Relaxing the serializability constraint is especially attractive for partitioned systems since it would allow a richer mix of readonly transactions. (The original motivation for a weaker correctness criterion was to speed up the processing of readonly transactions in a distributed system.) Since readonly transactions occur frequently in most systems, allowing a richer mix of them often substantially increases the number of transactions executed while partitioned.

In GAWE82, readonly transactions are divided into two classes: those requiring strong consistency and those requiring weak consistency. A strongly consistent transaction is processed in the normal fashion: its execution must be serializable with respect to update transactions and other strongly consistent transactions. A weakly consistent transaction must see a consistent database state (the result of a serializable execution of update transactions), but its execution need not be serializable with respect to other readonly transactions. The following example illustrates.

EXAMPLE: Consider again the banking database of the first section with Sites A and B partitioned. The following sequence of transactions occur:

SITE A		**SITE B**	
C:	*checking deposit of \$50*	D:	*savings deposit of \$100*
A_A:	*read checking and savings accounts*	A_B:	*read checking and savings accounts*

Notice that the two update transactions, considered alone, are serializable. In fact, since they access different items, both $C;D$ and $D;C$ are valid serialization orders. However, when the accounting transactions A_A and A_B are included, the execution is not serializable. The database state read by A_A is possible only if C executes before D, while the state read by A_B is possible only if D executes before C. (Both A_A and A_B see a valid serialization order of the updates; the problem is that they see different orders.)

If A_A and A_B required only weak consistency, the above execution would be "correct": the update transactions alone are serializable and each weakly consistent transaction sees the result of a serializable execution of update transactions.

The use of different consistency levels can be integrated with any of the syntactic approaches discussed in the previous section. In a pessimistic strategy a transaction requiring only weak consistency can be executed at any time in any partition, as long as the partition contains copies of items read by the transaction. The transaction will always see a consistent database state since all update transactions are guaranteed to be (globally) consistent. In an optimistic strategy, such a transaction sees a consistent state only if it does not read the result of an update transaction that is eventually rolled back.

The choice of a consistency level for a readonly transaction depends on how the information returned by the transaction is used. An accounting transaction reporting cash flow within a bank probably requires strong consistency. Inventory reporting and transactions computing summary statistics often need only weak consistency.

Fischer and Michael give an important application of weak serializability in their algorithms for directory systems[FIMI82]. A directory supports only three types of transactions: insert a unique item, list all items, and delete an item. Mail systems, calendar systems, and other familiar applications can be cast as directories. Exploiting the property that the list operation requires only weak consistency, they give an algorithm allowing unrestricted transaction processing in the presence of communication failures, including but not limited to failures partitioning the system.

Data-Patch [GABCR82]

Data-Patch is a tool which aids the database administrator in the development of a program to automatically integrate divergent databases. As in the previous optimistic strategies, transactions are executed "normally" during the failure. At reconnection, the final database state is constructed according to an integration program. Serializability is no longer the correctness criterion; rather, the integration program defines the "correct" final database. This is based on the premise that users may already have observed the effects of a non-serializable execution, thus restoring the database to a serializable state may not be the most sensible thing to do. For example, in an Airline Reservation System, if a flight becomes overbooked it may not be desirable to cancel reservations since a promise has been made to customers and normal passenger cancellations could take care of the problem.

The major design principle involved is identifying *image* and *plan* relations. Image relations are observable entities or relationships, and must reflect that in the final database. For example, in a database for Girard bank, the relation GIRARD(BRANCH, CASH, ...) might be used to record the amount of cash at each branch. The value of CASH in each tuple at recovery should reflect the actual amount of cash at that branch. This might be obtained as the latest value for CASH in each partition group. Plan relations do not represent observable entities and the DBA can therefore have more freedom in selecting the final values. In the next example, ACCOUNT is a plan relation.

EXAMPLE:

ACCOUNT (CUSTOMER, BALANCE, ...)
DEPOSIT (CUSTOMER, AMOUNT, DATE, ...)
WITHDRAWAL(CUSTOMER, AMOUNT, DATE, ...)

DEPOSIT and WITHDRAWAL are records of account activity. If during a partition a customer overdraws his account according to the records from each group, he may be assessed a penalty charge. Thus BALANCE would reflect the sum of withdrawals and deposits to the account, plus the penalty charge. If, on the other hand, a customer is mistakenly assessed a penalty charge because a DEPOSIT did not appear during a failure, the penalty charge may be dropped.

The above example shows that not only must a final database state be chosen, but corrective actions must be specified. That is, if integrity constraints are violated after the image and plan relations have been constructed, some sort of compensating or corrective action must be issued (e.g. penalty for overdraft, as above).

The Datapatch integration program is defined through a set of rules that specify how the integrated database can be obtained from two databases that exist after a partition. Some rules specify how differing facts are to be combined. For example, consider a field that represents the location of a ship. In this case, the DBA can select a "latest value" rule: if the field has a different value in each partition, in the integrated database use the value with the latest timestamp. If the field represents the number of reservations for a flight, the "arithmetic rule" can be used: the integrated value is the sum of the two partitioned values minus the value that existed when the p partition started. Other rules specify the corrective actions to be taken. For instance, a rule might specify that if the withdrawals exceed the deposits to an account (after the integrated database has been obtained), then a dunning letter should be sent to the customer.

Other Ideas.

Numerous ad-hoc techniques for exploiting the semantics of an application to increase availability have been proposed. Many of these can best be illustrated by examples.

The first example illustrates the idea of *splitting* a data item. In an Airline Reservation System [HASH80], let $SEATS$ represent the number of seats available on a particular flight. When a partition occurs, P_1 creates $SEATS_1$ containing 40% of the value of $SEATS$, and P_2 creates $SEATS_2$ containing 60% of the value of $SEATS$ (or other percentages reflecting the relative booking rates for that flight). At recovery,

$$SEATS = SEATS_1 + SEATS_2$$

would restore $SEATS$ to its correct value. Splitting can be used whenever the value of the data item represents a partial summation and each term in the summation is not dependent on the current value of the data item.

The second example comes from Incomplete Information Systems [DAVI82, LIPS79]. Suppose we have a tuple representing John Doe's age as less than 30. During a partition, P_1 gathers more information and concludes that his age is between 20 and 30, while P_2 concludes it to be between 15 and 25. At recovery, the intersection of these ranges, 20 to 25, may be taken as John Doe's age.

The last example illustrates the use of *failure-mode integrity constraints*. Recall the banking example of Figure 2, where overdrafts on the checking account were allowed as long as *checking balance* + *saving balance* ≥ 0. That example described a scenario where this constraint was violated during a partitioning. This anomaly could have been avoided by enforcing a failure-mode integrity constraint disallowing checking account overdrafts when the system is partitioned.

These ideas can be used with a pessimistic approach such as primary copy to allow more transactions to be executed: a portion of $SEATS$ would be available in each group, although the actual or current value for $SEATS$ could not be obtained due to possible bookings in the other group. However, the flight would never be overbooked if neither group sold more than their allotment of seats. It can also be used with more liberal approaches such as the Optimistic Protocol and Data-Patch to avoid conflict and possible transaction backouts. In the Optimistic Protocol, conflicts are mainly caused by updates to the same data-item. By splitting data-items and recombining at recovery, this can be avoided. In Data-Patch, integration becomes easier since the value for *SEATS* can simply be computed without canceling reservations.

5. ATOMIC COMMITMENT

A transaction on a distributed database typically executes at several sites. In order to ensure the "all or nothing" property of the transaction, the executing sites must unanimously agree to commit or to abort the transaction. Until now we have assumed that this agreement, known as atomic commitment, can be achieved in a partitioned system. Let us now examine how reasonable this assumption is.

Viewed abstractly, in a commitment protocol each participant first votes to "accept" or "reject" the transaction based on its ability to process the transaction and then decides whether to commit or abort based on the voting. Commitment normally requires unanimous acceptance[3] Of course, all decisions must agree.

The *two-phase commit protocol* [GRAY78] is a straightforward implementation of the above. In the first phase, a designated participant, the coordinator, solicits the votes from its cohorts. In the second phase, it decides based on the votes and then sends the decision to all participants. In the course of the protocol, each participant voting "accept" goes though three distinct states: an *uncommitted* state where it has not voted, an *in-doubt* state where it has voted but does not know the result of the voting, and a *decision* state where it knows the commit/abort decision. (A participant voting "reject" does not occupy the in-doubt state since it knows the eventual outcome.)

Consider the consequences of a partitioning occurring during the execution of the two-phase commit protocol. In each partition the participants, acting together, will attempt to decide the

[3]Some protocols for fully replicated databases require only acceptance by a majority.

outcome based on their states. If the partition contains the coordinator, a decided participant, or an uncommitted participant, a consistent decision can be reached (in the case of an uncommitted participant, abort will be chosen). However, a partition containing only in-doubt participants and lacking the coordinator can not safely decide: the participants can not commit since they do not know the outcome of the voting, and they can not abort since they may contradict the decision of the coordinator. Hence, these sites must wait until reconnection before deciding, and the protocol (and associated transaction) is said to be *blocked* at those sites.

Given that the two-phase commit protocol occasionally blocks, the interesting question then is: are there any nonblocking protocols for partitionings? The answer is no: even under the most favorable, realistic partitioning assumptions, there exists no nonblocking protocols [SKEE82b]. The situation is even worse if sites can fail during a partitioning; in this case there is no protocol that guarantees that even a single site will be able to decide.

Since it is impossible to eliminate blocking, it is desirable to minimize it. Several protocols have been proposed that, under appropriate partitioning assumptions, block less than the two-phase commit protocol. One protocol, the *decentralized two-phase commit protocol,* reduces the likelihood of blocking by decreasing the time a site spends in the in-doubt state [SKEE82c]. This is accomplished by having the participants send their votes directly to each other, bypassing the coordinator. Another protocol, the *quorum commit protocol,* reduces the probability that a large partition (one consisting of many participants) will be blocked in the event of a partitioning, by introducing extra phases [SKEE82a, SKEE82b]. Its principal advantage is that it is also resilient to site failures and (nonpartitioning) communication failures. However, both protocols have drawbacks. Although the decentralized protocol decreases the probability that a partitioning will occur while sites are in the in-doubt state, it increases the expected number of blocked sites if a partitioning should occur. The quorum protocol actually increases the chance that some site will be blocked in the event of a partitioning (although the expected number of blocked sites decreases).

How the partition strategies of the previous two sections treat blocked transactions depends on whether the strategy is pessimistic or optimistic. In a pessimistic strategy, the data items at undecided sites must be rendered inaccessible until reconnection. In an optimistic strategy more flexibility is possible. A partition can tentatively commit or abort a blocked transaction. If its decision is inconsistent with other decisions, it can resolve in the same way that it resolves other inconsistencies, by rolling back the offending transaction and all dependent transactions. Since rolling back is fairly expensive, a tentative decision should be made only if it has a high probability of being correct.

6. DISCUSSION

6.1. Guidelines for the Selection of a Strategy.

Given an application, how should one choose a partition strategy? Caution should be used in answering this question. A criticism that has been levied at research in the distributed database area in general [MOHA80] is that solutions are commonly viewed in isolation from other problems. In fact, different mechanisms may be so highly intertwined that changes proposed in one area affect many other parts of the system. In particular, with partition strategies one must pay attention to the concurrency control mechanism being used. For example, the use of a voting or token-passing concurrency control algorithm may dictate a corresponding partition algorithm unless one worries about restructuring the vote or reassigning tokens (see also DAVI82 for a discussion of the relationship between the optimistic protocol and common forms of concurrency control). In addition, very little attention has been given to the performance of proposed mechanisms. In some cases this is because it is difficult to construct an appropriate model; in others it is because the mechanism is highly application dependent.

With these cautions in mind, we group the factors that influence the choice of a strategy into three areas:

Environment. Included here are the properties of the network and the nature of the partitionings. An important consideration is whether partitionings are caused by failures or are the result of anticipated events. In the latter case, complete information about the characteristics of the partitioning, including duration and network topology, may be known, and this can be exploited in some strategies (in particular, class conflict analysis).

However, most systems partition because of failures, and in this case the robustness of the strategy may be an important factor. For example, a primary site strategy would be a poor choice if sites failures can not be distinguished from communication failures. Also, class conflict analysis (as presented) can not be used if communication failures do not always result in clearly delineated partitions.

Regardless of the cause of a partitioning, its duration is an important factor in choosing between optimistic and pessimistic strategies.

Workload. Two important workload characteristics are average transaction length and transaction variance. Optimistic policies work better when transactions are short and variance small.

Another important workload factor is locality of reference: Do updates to given data-items tend to occur at a certain site? If so, a primary site strategy will not prohibit many transactions and availability will still be good. The backout rate in the optimistic protocol will also be reduced, but the transactions will still have to be tested for conflict.

Application Specific. These factors fall into two groups. The first are requirements placed by the application on transaction processing. Two important questions are:

(1) Can transaction processing be temporarily halted for recovery purposes? If not, a pessimistic approach should be adopted which merely requires the forwarding of updates to merge the databases.

(2) Can transaction processing be limited in parts of the database, or is availability a premium? If the latter, a more optimistic approach should be used.

The second group include semantic considerations. Relevant questions here are:

(1) Can transactions be backed out? That is, do they have an inverse? If the latter, either conflict should be avoided totally, or the divergent databases should be patched up using compensating actions if necessary to achieve correctness.

(2) Is serializability a concern, or is a more procedural definition of "correctness" in the final database state acceptable? If not, a Data-Patch approach can be used.

(3) Should a partitioned system be expected to behave exactly as an unpartitioned system? For example, even if serializability is the "normal" correctness criterion, under extenuating circumstances (such as partition failures) a more lenient definition could be used.

6.2. Future Directions

Partitioned operation is still very much an active research area. We comment briefly on three interesting research directions.

One obvious deficiency in our current knowledge of partition strategies is the lack of any performance data on how well they work. Few strategies have been implemented and none tested on a representative application. Clearly, more experience with the proposed strategies is needed before we can understand the performance tradeoffs between them.

Another important area of research is the adaptation of these strategies to accommodate more general processing models, in particular, nested transactions (and the related concept of multilevel atomicity [LYNCH83]). Nested transactions arise in general purpose distributed programming environments such as ARGUS [LISK83].

Finally, the use of semantics in partitioned strategies has been only scantily explored. One interesting direction is to assume that data items are instances of abstract data types and transactions are instances of operations on those types. Type-specific partition strategies can be derived from the formal properties of the types. (This is an extension of the notion of type-specific concurrency control proposed in [SCSP83].)

References

[ALDA76] Alsberg, P.A., and Day, J.D. "A Principle for Resilient Sharing of Distributed Resources," *Proc. 2nd International Conference on Software Engineering,* 1976.

[APWI84] Apers, P.M., and Wiederhold, G., "Transaction Classification to Survive a Network Partition", unpublished manuscript, Computer Science Department, Stanford University, July 1984.

[BEGO81] Bernstein, P.A., and Goodman, N., "Concurrency Control in Distributed Database Systems," *Computing Surveys 13,* 2 (June, 1981).

[BEGO83] Bernstein, P.A., and Goodman, N., "The Failure and Recovery Problem for Replicated Databases," *Proc. 2nd ACM Symp. on Princ. of Distributed Computing,* Montreal, Quebec, August 1983, 114-122.

[BSR80] Bernstein, P.A., Shipman, D.W., and Rothnie, Jr., J.B., "Concurrency Control in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems 5,* 1 (March 1980), 18-51.

[BGRCK83] Blaustein, B.T., Garcia-Molina, H., Ries, D.R., Chilenskas, R.M., and Kaufman, C.W., "Maintaining Replicated Databases Even in the Presence of Network Partitions," *EASCON,* 1983.

[BLAU81] Blaustein, B.T. *Enforcing Database Assertions: Techniques and Applications,* TR-21-81 Aiken Computation Laboratory, Harvard University, 1981.

[DAVI82] Davidson, S.B., *An Optimistic Protocol for Partitioned Distributed Database Systems,* Princeton University, Dept. of EECS, October 1982.

[EGLT76] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM 19,* 11 (November 1976), 624-633.

[FIMI82] Fischer, M.J., and Michael, A., "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* May 1982, 70-75.

[GAWE82] Garcia-Molina, H., and Weiderhold, G., "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems 7* 2, (June 1982), 209-234.

[GARC83] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems 8,* 2 (June 1983).

[GABCR82] Garcia-Molina, H., Allen, T., Blaustein, B., Chilenskas, R.M., and Ries, D.R., "Data-Patch: Integrating Inconsistent Copies of a Database After a Partition," ????

[GMBLL81] Gray, J.N., McJones, P., Blasgen, M., Lindsay, B., Lorie, L., Price, T., Putzulo, F., and Traiger, I., "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys 13,* 2, (June 1981) 223-242.

[GSCDFR83] Goodman, N., Skeen, D., Chan, A., Dayal, U., Fox, S., and Ries, D., "A Recovery Algorithm for a Distributed Database System," *Proc. 2nd ACM Symp. on Princ. of Database Systems,* Atlanta, Georgia, March 1983, 8-15.

[GIFF79] Gifford, D.K. "Weighted Voting for Replicated Data," *Proc. of the 7th Symposium on Operating Systems Principles,* Dec. 1979.

[HASH80] Hammer, M.M., and Shipman, D.W., "The Reliability Mechanisms of SDD-1: A System for Distributed Databases", Computer Corporation of America Tech. Report CCA-80-04 (Jan. 1980) (a shorter version appears in: *ACM Transactions on Database Systems 5,* 4, 431-466).

[KOHL81] Kohler, W., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys 13,* 2, (June 1981), 149-184.

[KURO81] Kung, H.T., and Robinson, J.T., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems 6,* 2 (June 1982), 213-226.

[LIPS79] Lipski, W., "On Semantic Issues Connected With Incomplete Information Databases", *ACM Transactions on Database Systems 4,* 3 (Sept. 1979).

[LISC83] Liskov, B., and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions Programming Languages and Systems 5,* 3 (July 1983), 381-404.

[LYNCH83] Lynch, N.A., "Multilevel Atomicity— A New Correctness Criterion for Distributed Databases," *ACM Transactions on Database Systems 8,* 4 (December 1983), 484-502.

[MIWI82] Minoura, T., and Wiederhold, G, "Resilient Extended True-Copy Token Scheme for a Distributed Database System", *IEEE Transactions on Software Engineering SE-8,* 3 (May 1982), 173-189.

[MOHA80] Mohan, C., "Distributed Data Base Management: Some Thoughts and Analyses," *ACM Proceeding 1980,* 399-410.

[PAPA79] Papadimitriou, C.H., "The Serializability of Concurrent Database Updates," *J. ACM 26,* 4, (October 1979) 631-653.

[PPR81] Parker, D.S., Popek, G.P., Rudisin, G., et al., "Detection of Mutual Inconsistency in Distributed Systems," *Proc. of the 5th Berkeley Workshop on Distributed Data Management and Computer Networks,* February, 1982.

[PARA82] Parker, D.S., Ramos, R.A., "A Distributed File System Architecture Supporting High Availability," *Proc. of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks,* Pacific Grove, California, February 1982.

[POPE81] Popek, G., et al., "Locus: A Network Transparent, High Reliability Distributed System," *Proc. 8th SOSP,* December 1981, 169-177.

[ROGO77] Rothnie, J.B., and Goodman, N., "A Survey of Research and Development in Distributed Database Management," *Proc. 3rd VLDB,* Tokyo (Oct. 1977), 48-61.

[SCSP83] Schwartz, P., and Spector, A., "Synchronizing Shared Abstract Types," Tech. Report CMU-CS-83-163, Carnegie-Mellon University, Computer Science Dept., Pittsburgh, Pennsylvania, November 1983.

[SKEE82a] Skeen, D., "A Quorum-Based Commit Protocol," *Proc. of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks,* Pacific Grove, California, February 1982.

[SKEE82b] Skeen, D., *Crash Recovery in a Distributed Database System* (Ph.D. Thesis), ERL Memo M82/45, University of California, Berkeley, May 1982.

[SKEE82c] Skeen, D., "On Network Partitioning," IEEE COMPSAC, Nov. 1982.

[SKST83] Skeen, D., and Stonebraker, M., "A Formal Model of Crash Recovery in a Distributed System," *IEEE Transactions on Software Engineering SE-9,* 3 (May 1983), 219-228.

[SKWR84] Skeen, D., and Wright, D., "Increasing Availability in Partitioned Networks," *Proc. 3rd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* April 1984, 290-299.

[TGGL82] Traiger, I.L., Gray, J.N., Galtieri, C.A., and Lindsay, B.G., "Transactions and Consistency in Distributed Database Systems," *Transactions on Database Systems 7,* 3 (September 1982), 323-342.

[STON79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies in Distributed INGRES," *IEEE Transactions on Software Engineering SE-5,* 3 (May 1979), 188-194.

[THOM78] Thomas, R.H., "A Solution to the Concurrency Control Problem for Multiple Copy Data Bases," *IEEE COMPCON* Spring 1978.