January 1989

Replicated Data and Partition Failures

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Replicated Data and Partition Failures

Abstract
In a distributed database system, data is often replicated to improve performance and availability. By storing copies of shared data on processors where it is frequently accessed, the need for expensive, remote read accesses is decreased. By storing copies of critical data on processors with independent failure modes, the probability that at least one copy of the data will be accessible increases. In theory, data replication makes it possible to provide arbitrarily high data availability.

In practice, realizing the benefits of data replication is difficult since the *correctness* of data must be maintained. One important aspect of correctness with replicated data is *mutual* consistency: all copies of the same logical data-item must agree on exactly one "current value" for the data-item. Furthermore, this value should "make sense" in terms of the transactions executed on copies of the data-item. When communication fails between sites containing copies of the same logical data-item, mutual consistency between copies becomes complicated to ensure. The most disruptive of these communication failures are *partition failures*, which fragment the network into isolated subnetworks called partitions. Unless partition failures are detected and recognized by all affected processors, independent and uncoordinated updates may be applied to different copies of the data, thereby compromising the correctness of data. Consider, for example, an Airline Reservation System implemented by a distributed database which splits into two partitions when the communication network fails. If, at the time of the failure, all the nodes have one seat remaining for PAN AM 537, reservations could be made in both partitions. This would violate correctness: 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.)

The design of a replicated data management algorithm tolerating partition failures (or *partition processing strategy*) is a notoriously hard problem. Typically, the cause or extent of a partition failure cannot be discerned by the processors themselves. At best, a processor may be able to identify the other processors in its partition; but, for the processors outside of its partition, it will not be able to distinguish between the case where those processors are simply isolated from it and the case where those processors are down. In addition, slow responses can cause the network to appear partitioned even when it is not, further complicating the design of a fault-tolerant algorithm.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-02.

REPLICATED DATA AND PARTITION FAILURES

Susan B. Davidson

MS-CIS-89-02

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

January 1989

Acknowledgements: This research was supported in part by NSF grants MCS-8219196-CER, IRI84-10413-AO2, U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027 and by a grant from AT&T's Telecommunications Program.

Replicated Data and Partition Failures

Susan B. Davidson*
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

January 5, 1989

1 Introduction.

In a distributed database system, data is often replicated to improve performance and availability. By storing copies of shared data on processors where it is frequently accessed, the need for expensive, remote read accesses is decreased. By storing copies of critical data on processors with independent failure modes, the probability that at least one copy of the data will be accessible increases. In theory, data replication makes it possible to provide arbitrarily high data availability.

In practice, realizing the benefits of data replication is difficult since the *correctness* of data must be maintained. One important aspect of correctness with replicated data is *mutual consistency*: all copies of the same logical data-item must agree on exactly one "current value" for the data-item. Furthermore, this value should "make sense" in terms of the transactions executed on copies of the data-item. When communication fails between sites containing copies of the same logical data-item, mutual consistency between copies becomes complicated to ensure. The most disruptive of these communication failures are *partition failures*, which fragment the network into isolated subnetworks called *partitions*. Unless partition failures are detected and recognized by all affected processors, independent and uncoordinated updates may be applied to different copies of the data, thereby compromising the correctness of data. Consider, for example, an Airline Reservation System implemented by a distributed database which splits into two partitions when the communication network fails. If, at the time of the failure, all the nodes have one seat remaining for PAN AM 537, reservations could be made in both partitions. This would violate correctness: 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.)

The design of a replicated data management algorithm tolerating partition failures (or *partition-processing strategy*) is a notoriously hard problem. Typically, the cause or extent of a partition failure cannot be discerned by the processors themselves. At best, a processor may be able to identify the other processors in its partition; but, for the processors outside of its partition, it will not be able to distinguish between the case where those processors are simply isolated from it and

*Parts of this were reprinted with permission from "Consistency in Partitioned Networks" by Susan B. Davidson, Hector Garcia-Molina and Dale Skeen, *ACM Computing Surveys* 17:3 (Sept. 1985) pp. 341-370. This research was supported in part by a grant from AT&T's Telecommunications Program at the University of Pennsylvania, and by grants NSF MCS 8219196-CER.

the case where those processors are down. In addition, slow responses can cause the network to appear partitioned even when it is not, further complicating the design of a fault-tolerant algorithm.

However, the problems associated with maintaining correct operation in a partitioned distributed database system are not limited to the problems associated with the correctness of data. Due to the expense and complexity of maintaining replicated data, most distributed database systems limit the amount of replication to a few copies. Since data is not replicated at every site in the network, it is possible to pose *queries*[1] during network partitioning for which not all data are available. Ideally in such a situation, the system should attempt to provide some level of service by providing *as good an answer as possible.* For example, if a doctor at a hospital wanted to query a distributed Blood Bank database to find how many pints of blood were available throughout the system, and only Blood Bank A were accessible due to a partition failure, a useful answer would be "Blood Bank A has 20 pints, but there may be more at other (currently inaccessible) sites."

In this chapter, we will first discuss the tradeoffs involved in designing a partition-processing strategy. A formal notion of correctness in a replicated database system ("one-copy serializability") will then be given, along with an overview of several "quorum-based" partition-processing strategies. We will then shift our attention away from the problem of *updating* during partition failures to the problem of answering *queries*, and present methods of providing partial answers in the face of unavailable data.

Although the discussion on updating transactions 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, and other applications using transactions as their underlying model of processing.

2 Correctness Versus Availability

When designing a system that will operate when it is partitioned, the competing goals of availability (the system's normal function 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; but this severely compromises availability. In applications where partitions either occur frequently or occur when access to the data is imperative, this solution is not acceptable. For example, in the Airline Reservation System it may be too expensive to have a high connectivity network and partitions may occasionally occur. Many transactions are executed each second (TWA's centralized reservations system estimates 170 transactions per second at peak time [GS]), 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.

[1]Queries are transactions that do *not* perform updates, also called *read-only* transactions.

In some applications, such "incorrect" results may be acceptable in light of the higher availability achieved: when partitions are reconnected, the problems may be corrected by executing transactions missed by a partition, and by choosing certain transactions to "undo." If the chosen transactions have had no real-world effects, they can be undone using standard database recovery methods. If, on the other hand, they have had real-world effects, then appropriate *compensating transactions* must be run, transactions which not only restore the values of the changed database items but also issue real-world actions to nullify the effects of the chosen transactions (*e.g.*, by canceling certain reservations and sending messages to affected users). Alternatively, *correcting transactions* can be run, transforming the database from an incorrect state to a correct state without undoing the effects of any previous transactions. For instance, in a banking application, the correcting transaction for overdrawing a checking account during a partitioning would apply an overdraft charge. Of course, in some applications incorrect results are either unacceptable or incorrectable. For example, it may not be possible to undo or correct a transaction that effectively hands $1,000,000 to a customer.

Since it is clearly impossible to satisfy both goals simultaneously, one or both must be relaxed to some extent. Several partition processing strategies have been suggested that either *relax correctness*, or rely on compensating or correcting transactions to *regain consistency* once the partition is repaired [Dav,PPR*,GAB*,LB,KG,GK]. Other partition processing strategies have been suggested that *pre-analyze* transactions or use *type-specific* information to increase availability while guaranteeing correctness [Her,GK,BGR*,Wri]. Since most of these techniques require extensive knowledge about what the information in the database represents, how applications manipulate the information, and how much undoing/correcting/compensating inconsistencies will cost, we will limit our discussion of partition-processing strategies to a class that *guarantees* correctness, and does so in a *syntactic* manner (*i.e.*, no semantic understanding of the database is required). A more complete survey of these techniques can be found in [DGS], and an analysis of the limitations on availability for strategies that guarantee correctness can be found in [COK].

3 The Notion of Correctness

What does correct processing mean in a database system? Informally, a database is correct if it correctly describes the external objects and processes that it is intended to model. In theory, such a vague notion of correctness could be formalized by a set of static constraints on objects and their attributes, and a set of dynamic constraints on how objects can interact and evolve. In practice, a complete specification of the constraints governing even a small database is impractical (besides, even if it were practical, enforcing the constraints would not be). Consequently, database systems use a less ambitious, very general notion of correctness based on the order of transaction execution, *one-copy serializability*, and on a small set of static data constraints known as *integrity constraints*.

In this section, we examine the notion of correctness, beginning informally with examples illustrating incorrect behavior, followed by a more formal definition of correctness in the traditional database system. When referring to the state of the database, we use the terms "correct" and "consistent" interchangeably.

3.1 Anomalies

Consider a banking database that contains a checking account and a savings account for a certain customer, with a copy of each account stored at sites A and B. Suppose a communication failure

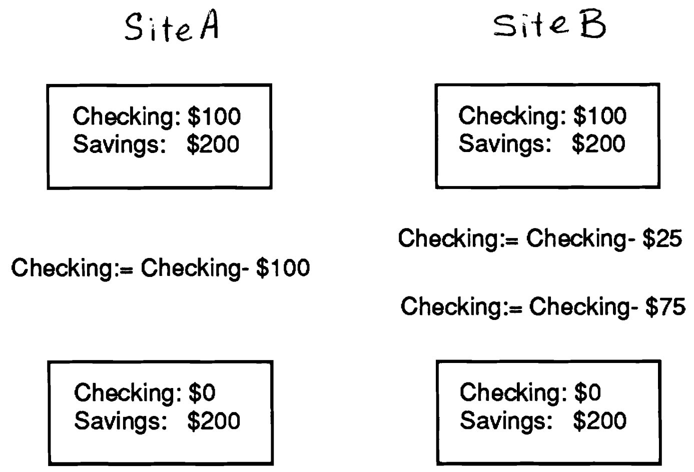

Figure 1: An anomaly resulting from concurrent write operations on the same data item in separate partitions.

isolates 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). 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 a checking account containing only \$100. The anomaly is caused by conflicting write operations issued in parallel by transactions executing in different partitions.

An interesting aspect of this example is that in the resulting database all copies are mutually consistent, *i.e.*, all copies of a data-item contain the same value.[2] Thus, although it is commonly used as the correctness criterion for replicated file systems and information databases, such as telephone directories, mutual consistency is not a sufficient condition for correctness in a transaction-oriented database system. 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 on the same database is illustrated in Figure 2.This figure shows the result of executing a checking withdrawal of \$200 at site A, and a savings withdrawal of \$200 at site B. Here, we assume that the semantics of the checking withdrawal allow the account to be overdrawn as long as the overdraft is covered by funds in the savings account (*i.e.*, $checking + savings \geq 0$). The semantics of the savings withdrawal are similar.

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 conflicting writes (none existed since the transactions updated different accounts), but instead because accounts are allowed to be read in one partition and updated in another.

Concurrent reads and writes in different partitions are not the only sources of inconsistencies in a partitioned system, 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

[2]This is the narrowest interpretation of several uses of the term "mutual consistency" that appear in the literature. Some authors use mutual consistency synonymously with one-copy equivalence (defined in the next section).

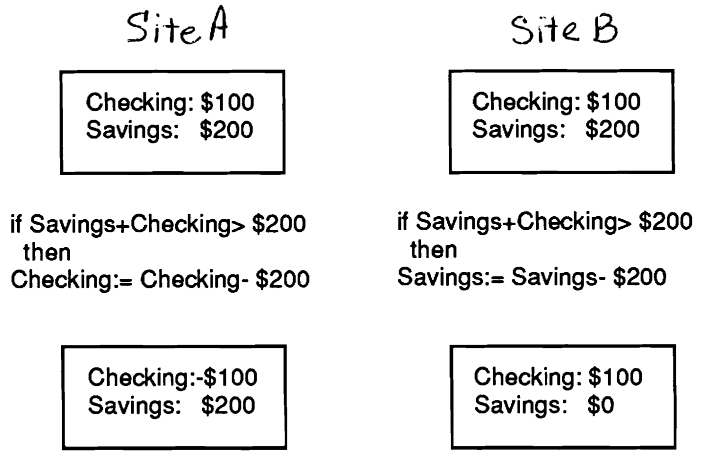

Figure 2: An anomaly resulting from concurrent read and write operations in different partitions.

be violated. However, the above are typical anomalies that can occur if conflicting transactions are executed in different partitions.

3.2 One-Copy Serializability

A database is a set of *logical data-items* that support the basic operations *read* and *write*. The granularity of these items is not important: they could be records, files, relations, etc. The *state* of the database is an assignment of values to the logical data-items. For brevity, logical data-items are subsequently called data-items or, more simply, items.

A *transaction* is a program that issues read and write operations on the data-items, and either terminates successfully (*commits*) or fails (*aborts*). 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 *read-only transaction* (or *query*) neither issues write requests nor has external effects. Transactions are assumed to be correct, that is *a transaction, when executed alone, transforms an initially correct database state into another correct state* [TGGL].

Transactions interact with one another indirectly by reading and writing the same data-items. Two operations on the same item are said to *conflict* if at least one of them is a write. Conflicts are often labeled either *read-write*, *write-read*, or *write-write* depending on the types of data operations involved and their order of execution [BGa]. Conflicting operations are significant because their order of execution affects the final database state.

A generally accepted notion of correctness for a database system is that it executes 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. The execution of each transaction is "all or nothing": either all of the transaction's 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*.) This property is often referred to as *atomic commitment*.

2. The execution of several transactions concurrently produces the same database state as some

serial execution of the same transactions. The execution is then said to be *serializable.*

The first property is established by the commit and recovery algorithms of the database system (*e.g.*, logging techniques [GMB*,Ver]); the second, by the concurrency control algorithm. For now, we will concentrate on the concurrency control aspects of transaction processing and assume that site and transaction failures are tolerated correctly.

EXAMPLE: As an example of how concurrent execution can produce incorrect results in a *centralized* database, consider the concurrent activity at site B in Figure 1. Here, two transactions, T_1 and T_2, concurrently withdraw money from a checking account, which we will represent as x. A withdrawal transaction T_i could be represented by the following operations:

$$T_i : r_i(x), w_i(x),$$

where r_i represents a read operation and w_i a write operation of the new value of x by transaction T_i. For transaction T_1, $w_1(x)$ writes a value of x that is \$25 less than the value read by $r_1(x)$; for transaction T_2, $w_2(x)$ writes a value of x that is \$75 less than the value read by $r_2(x)$. The following sequence of operations (or *history*) represents an execution that is intuitively incorrect:

$$H = r_1(x), r_2(x), w_1(x), w_2(x).$$

Although the customer withdrew a total of \$100 from the account, H only reflects the execution of *one* withdrawal transaction. Since both transactions read the same initial value of x (\$100), the final value of x, written by T_2, is \$25. H is not serializable; there is also no serial execution of T_1 and T_2 that results in a final value of \$25 for x. For example, the serial execution of T_1 before T_2 would be:

$$H_s = r_1(x), w_1(x), r_2(x), w_2(x).$$

Since T_2 reads the value of x produced by T_1 (\$75), the final value written by T_2 is \$0. This would also be the final value for x if T_2 were executed before T_1.

One concurrency control technique that would have avoided the previous anomaly is *strict two-phase locking*[EGLT]: Before executing a write operation, the transaction must acquire an exclusive lock on the data-item; before executing a read operation, the transaction must acquire at least a shared lock (if the transaction will later write the data-item, an exclusive lock may be obtained). Exclusive locks are said to *conflict* with other shared or exclusive locks on the same data-item; however, shared locks only conflict with exclusive locks on the same data-item. Under strict two-phase locking, conflicting locks may not be granted, and any lock that is granted is held until the transaction terminates. If this technique had been used in H, T_1 would have acquired an exclusive lock on x before reading it, thus preventing T_2 from reading x until T_1 had written its new value for x. The result would have been the serial execution H_s.

Atomic transaction execution (the concurrent execution of transactions is serializable) together with the assumption that transactions are correct (a transaction executed alone transforms an initially correct database state into another correct state) imply by induction that the execution of any set of transactions transforms an initially correct database state into a new, correct state. While atomic execution is not always necessary to preserve correctness, most real database systems implement it as their sole criterion of correctness. This is because atomic execution is simple (it corresponds to users' intuitive model that transactions are processed sequentially) and can be enforced by very general mechanisms that determine the order of conflicting data operations (such as strict two-phase locking, shown in the previous example). These mechanisms are independent of both the semantics of the data being stored and of the transactions manipulating it.

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 accounts must be nonnegative) to elaborate constraints that relate the values of many data-items. In systems enforcing integrity constraints, a transaction is allowed only if its execution is atomic and its results satisfy the integrity constraints. To simplify the discussion, we will assume that integrity constraints are checked as part of the normal processing of a transaction (*e.g.*, the withdrawal transaction fails if the checking account balance becomes negative).

Notice that we have not specified whether we were discussing a centralized or a distributed database system; it has not been necessary to do so since the definitions, the properties of transaction processing, and the correctness criteria are the same in both. Of course, the algorithms for achieving correct transaction processing differ markedly between the two types of implementations.

In a *replicated database*, the value of each logical item x is stored in one or more *physical data-items*, which are referred to as the *copies* of x. Each read and write operation issued by a transaction on some logical data-item must be mapped by the database system to corresponding operations on physical copies. To be correct, the mapping must ensure that *the concurrent execution of transactions on replicated data is equivalent to a serial execution on nonreplicated data*, a property known as *one-copy serializability*. The logic that is responsible for performing this mapping is called the *replica control algorithm*.

EXAMPLE: Continuing with the previous banking example, consider the situation in Figure 1. Here, transactions T_1 and T_2 execute at site B, while transaction T_3 executes at site A. If the concurrency control used at each site is strict two-phase locking, we know that the *local* execution will be serializable. However, the global execution may be incorrect due to an incorrect replica control algorithm. If we adopt a "read-one, write-one" replica control (the *local copy* of a data-item is read and updated), we get the execution in Figure 1. Letting the copies of x at A and B be x_A and x_B respectively, the withdrawal transactions become the following sequences of operations on physical copies:

$$T_3 : r_3(x_A), w_3(x_A),$$

$$T_1 : r_1(x_B), w_1(x_B).$$

$$T_2 : r_2(x_B), w_2(x_B).$$

As shown in Figure 1, site A executes T_3 while site B executes T_1 followed by T_2. While mutual consistency is preserved ($x_A = x_B = \$200$), the result is incorrect since only \$100 was withdrawn from the logical data-item x. The execution is *not* one-copy serializable since the execution of T_1, T_2 and T_3 in the distributed system does not reflect a serial execution of the transactions on the logical data-item x.

However, if the replica control algorithm used in were "read-one, write-all", this anomaly would have been avoided: A transaction must read one copy of a data-item (usually, the nearest copy), but must update *all* copies. In this case, the withdrawal transactions become the following sequences of operations on physical copies:

$$T_3 : r_3(x_A), w_3(x_A), w_3(x_B)$$

$$T_1 : r_1(x_B), w_1(x_A), w_1(x_B)$$

$$T_2 : r_2(x_B), w_2(x_A), w_2(x_B).$$

Since each site uses strict two-phase locking as its concurrency control, if T_3 and T_1 both read the original value of x, deadlock will occur when they try to update the remote copies of x: T_3 holds an exclusive lock on x_A and cannot release the lock until it acquires an exclusive lock on x_B and completes, while T_1 holds an exclusive lock on x_B and cannot release the lock until it acquires an exclusive lock on x_A and completes. However, the following execution sequences at sites A and B respectively would avoid deadlock and are one-copy serializable:

$$\begin{array}{lllll} H_A = & r_3(x_A), w_3(x_A), & & w_1(x_A), & w_2(x_A) \\ H_B = & & w_3(x_A), r_1(x_B), w_1(x_B), & r_2(x_B), w_2(x_B) & \end{array}$$

Note that the joint execution of H_A and H_B corresponds to executing T_3, T_1 and then T_2 using the logical data-item x.

Similar reasoning will lead the reader to conclude that the anomaly shown in Figure 2 would also be avoided using "read-one, write-all" replica control together with strict two-phase locking.

As a correctness criterion, one-copy serializability is attractive for the same reasons that (normal) serializability is: it is intuitive, and can be enforced using general-purpose mechanisms that are independent of the semantics of the database and of the transactions executed.

3.3 Partitioned Operation

Let us now consider transaction processing in a partitioned network, where the communication connectivity of the system is broken by failures or by anticipated communication shutdowns. To keep the exposition simple, let us assume that the network is "cleanly" partitioned (that is, any two sites in the same partition can communicate and any two sites in different partitions cannot communicate), the database is *completely replicated* (a copy of every item is at every site throughout the system), and one-copy serializability is the correctness criterion.

While the system is partitioned, each partition must determine which transactions it can execute without violating the correctness criteria. Actually, this can be thought of as two problems:

1. each partition must maintain correctness within the part of the database stored at the sites comprising the partition, and

2. each partition must make sure that its actions do not conflict with the actions of other partitions, so that the database is correct across all partitions.

If we assume that each site in the network is capable of detecting partition failures, then correctness *within* a partition can be maintained by adapting one of the standard replica control algorithms for nonpartitioned systems. For example, the sites in a partition can implement a write operation on a logical object by writing all available copies in the partition ("read-one, write-all-available").

This, along with a standard concurrency control protocol, ensures one-copy serializability in the partition.

The really difficult problem is ensuring one-copy serializability *across* partitions: it is not sufficient to run a replica control algorithm that is correct in each partition to ensure that overall transaction execution is one-copy serializable.

EXAMPLE: Continuing with the banking example, suppose that a partition failure occurs before T_3 is executed at site A and T_1 and T_2 are executed at site B. If a "read-one, write-all-available" replica control strategy were used, the resulting execution would be the same as the "read-one, write-one" strategy in the previous section. Although the execution of T_1, T_2 and T_3 in their respective partitions is trivially one-copy serializable, conflicting operations occurred in different partitions, and the joint execution of both partitions is *not* one copy serializable.

In addition to solving the problem of global correctness, a partition processing strategy must solve two problems of a different sort. First, when the partitioning occurs, the database is faced with the problem of atomically committing ongoing transactions. The complication is that the sites executing the transaction may find themselves in different partitions, and thus unable to communicate a decision regarding whether to complete the transaction (commit) or to undo it (abort). In many cases, it is impossible to make a decision within each partition that is consistent across partitions, and the transaction is forced to *wait* until the failure is repaired. In this case, the transaction is said to be *blocked*. Blocking is clearly undesirable since the availability of data is reduced; for example, locks on data-items cannot be released until the transaction terminates. Unfortunately, while there are methods of *reducing* the likelihood of blocking, *there are no nonblocking commit protocols for network partitions* [Ske]. Note that the problem of atomic commitment in multiple partitions does not arise for a transaction submitted after the partitioning occurs (such a transaction will be executed in only one partition), and that this problem arises in any partitioned database system whether it is replicated or not.

Second, when partitions are reconnected, mutual consistency between copies in different partitions must be reestablished. That is, the updates made to a logical data object in one partition must be propagated to its copies in the other partitions. Conceptually, this problem can be solved in a straightforward manner by extra bookkeeping whenever the system partitions. For example, each update applied in a partition can be logged, and this log can be sent to other partitions upon reconnection. (Such a log may be integrated with the "recovery log" that is already kept by many systems.) In practice, an *efficient* solution to this problem is likely to be intricate and to depend on the normal recovery mechanisms employed in the database system. For this reason, we do not discuss it further.

3.4 Modeling Partitioned Behavior

To model the conflict between transactions in partitioned systems, we will use a *precedence graph* [Dav].[3] A precedence graph models the necessary ordering between transactions, and is used to check serializability across partitions. They are adapted from serialization graphs, which are used to

[3]A more complete modeling of partitioned behavior is replicated data serialization graphs [BHG]; however, precedence graphs are sufficient for this discussion.

check serializability within a site [Pap]. 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, see [Ull].)

The transactions executed in each partition group during the failure are represented by a serial history of transactions, their readsets and writesets. Such a history must exist since, by assumption, transaction execution within a partition is serializable. For partition i, For partition i, let $T_{i1}, T_{i2}, ..., T_{in}$ be the set of transactions, in serialization order, executed in i.

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 conflicts between transactions *in the same partition* with precedence edges. A *precedence edge* $(T_{ij} \longrightarrow T_{ik})$ represents the fact that T_{ij} wrote a copy that was later read by T_{ik} (write-read conflict), or that T_{ij} read a copy that was later changed by T_{ik} (read-write conflict). Since we are assuming that the readset of a transaction contains its writeset, write-write conflicts are subsumed by the write-read conflicts. 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, and that any equivalent execution must maintain this order. 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. This is modeled by interference edges. An *interference edge* $(T_{ij} \longrightarrow T_{lk},\ i \neq l)$ indicates that T_{ij} read an item that is written by T_{lk} in another partition $(READSET(T_{ij}) \cap WRITESET(T_{lk}) \neq \emptyset)$. The meaning of an 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, and indicates that any equivalent execution must maintain this order. (A write-write conflict manifests as a pair of read-write conflicts since each transaction's readset contains its writeset.)

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 in Figure 3, 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.) Note that the precedence graph contains the cycle

$$T_{11} \longrightarrow T_{12} \longrightarrow T_{13} \longrightarrow T_{21} \longrightarrow T_{22} \longrightarrow T_{11}$$

Intuitively, cycles in the precedence graph are bad: if T_{ij} and T_{kl} are in a cycle then the database reflects the results of T_{ij} executing before T_{kl} and of T_{kl} executing before T_{ij}, 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* [Dav]. 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. Thus, the combined execution within the two groups is *one-copy serializable.*

In the previous example, since a cycle resulted in the precedence graph, the combined execution

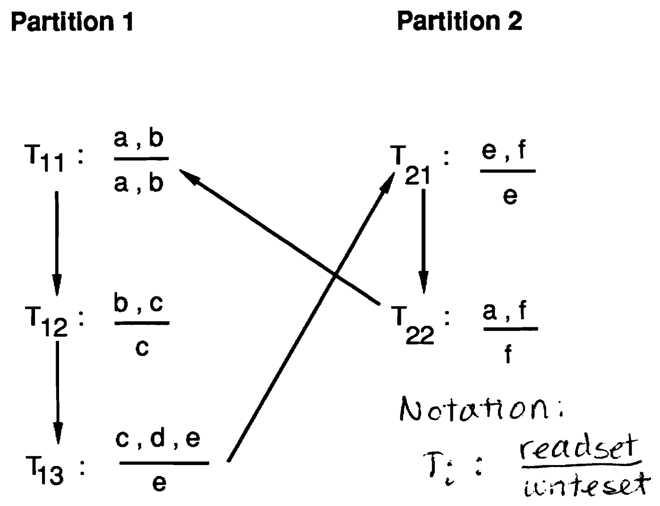

Figure 3: Conflict between transactions executed in different partitions indicated by cycle in precedence graph.

within the two partition groups is *not* one-copy serializable. In the next section, we will discuss *quorum based* partition processing strategies that guarantee acyclic precedence graphs.

4 Quorum-Based Approaches

Quorum-based partition processing strategies attempt to increase the availability of data while guaranteeing one-copy serializability by ***adjusting the number of copies that must be accessed to successfully read and write within a partition.*** For example, in "read-one, write-all" replica control, data-items cannot be written by transactions in either partition after a single failure since at least one copy becomes inaccessible in both groups.[4] However, if a replica control strategy required that only *some* copies be accessed, write operations could be performed in a group that contained *enough* copies after a partition failure.

Quorum-based approaches also model the varying "importance" of different copies of data-items by assigning each copy some number of *votes.* A replica control strategy then uses the total number of votes assigned to a data-item to dictate a read-quorum r and write-quorum w; that is, it dictates how many votes must be "collected" to read and write a data-item. If access is granted to a copy that has a vote of n, the transaction collects n votes from that copy. For example, to model the fact that a customer withdraws money from site A more frequently than site B, more votes could be assigned to x_A than to x_B.

In order to guarantee one-copy serializability, quorums must satisfy two constraints [Gif]:

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

2. $w > v/2$.

[4]Recall that we are assuming that data-items are completely replicated

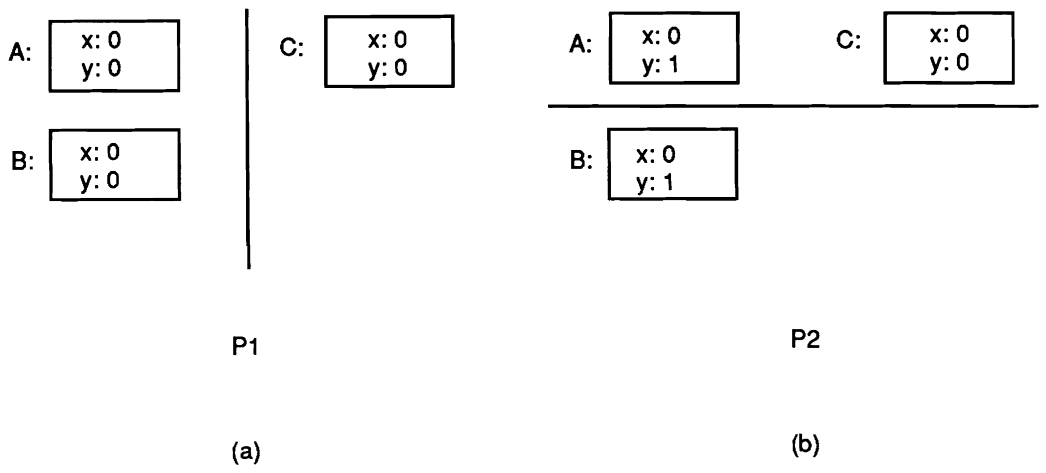

Figure 4: Correct transaction processing during partitioning using voting.

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. The most recent copy can be identified by *version numbers*; the copy with the highest version number is the copy read.

In a partitioned system, the first constraint guarantees that an item cannot be read in one partition and written in another; the second constraint ensures that two writes cannot occur in two different partitions on the same data-item. Hence, no interference edges can appear in the precedence graph of any execution, and one-copy serializability is guaranteed by the fact that correctness is maintained within each partition.

EXAMPLE: Suppose that sites A, B and C all contain copies of items x and y, and that a partition P_1 occurs, isolating A and B from C. Initially, $x = y = 0$, each site has 1 vote for each of x and y, and $r = w = 2$ for both x and y (see Figure 4(a)).

During the partitioning, transaction T_1 wishes to update y based on values read for x and y. Although it cannot be executed at C since it cannot obtain a read quorum for x, or read and write quorums for y, it can be executed at A, and the new value $y = 1$ is propagated to B (see Figure 4(b)).

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

Note that read-accessibility can be given a high priority by choosing r small; if $r < v/2$, 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. Note also that the algorithm does not distinguish between communication failures, site failures, or just slow response.

4.1 Assigning Votes

Different choices of vote assignments and quorums yield different "flavors" of partition processing strategies with different performance characteristics. For example, if all v votes are assigned to one copy, x_p, and $r = q = v$, then a "primary copy" strategy is emulated [AD,Sto]: x_p is responsible for all the read and write activity on x in the system. During a partition failure, only the partition group containing x_p can process transactions accessing x. Unfortunately, if the site containing the primary copy of a data-item fails and site failure cannot be distinguished from network partitioning, the data-item becomes inaccessible everywhere. As another example, if every copy is given a single vote, and $r = w = \lfloor \frac{v}{2} \rfloor + 1$, we have a simple "majority consensus" algorithm [Tho]: in the event of a partition failure, a partition containing a majority of sites can process transactions. In this scheme, the data-item may become inaccessible everywhere if the network fragments so that no group contains a majority of the sites; however, since partition failures are assumed to be "infrequent catastrophes", this is an unlikely occurrence.

Although the "majority consensus" approach might seem natural, there are cases in which it does not perform well [GB]. For example, consider a system with data-item x replicated at four sites A, B, C and D. Each copy is given a single vote, and $r = w = 3$. The set of groups of nodes that could execute transactions against x would be:

$$S = \{\{A,B,C\},\{A,B,D\},\{A,C,D\},\{B,C,D\}\}.$$

However, if we assigned x_A a vote of 2, and other copies a single vote, the majority is still 3 ($r = w = 3$), we get a *better* vote assignment. There are *more* groups of nodes that can execute transactions against x in the event of a network partition:

$$R = \{\{A,B\},\{A,C\},\{A,D\},\{B,C,D\}.$$

Note that every group of nodes that can operate under S can operate under R, but not *vice versa*. It is therefore important to carefully consider the vote assignments and failure characteristics of the network to choose the *best* assignment for a given application.

Another "quorum-like" partition processing strategy that appears to be similar to this simple form of voting is *coteries* [Lam]. In this approach, groups of nodes are selected that may perform the read and write operations for each data item. Each pair of groups must have a node in common to guarantee mutual exclusion. For example, R (given above) is a coterie; if read and write operations are only allowed to be performed in partitions that are a superset of one of the groups in R, one-copy serializability is guaranteed. Surprisingly, it turns out that coteries are more powerful than vote assignments [GB]: *there are sets of groups for which there exists no vote assignments.* However, since voting is easier to implement, most systems do not use coteries.

A weakness of *static* vote and quorum assignments is that reading an item can be expensive. Furthermore, it is *unnecessarily* expensive when there are no failures [ES,BGb]. In the next subsections, we will discuss ways of reducing this overhead by *dynamically* adjusting the read- and write-quorums.

4.2 Failure-Mode Quorums

Requiring a readset quorum significantly degrades performance when there are no failures, but is *necessary* to guarantee correctness *when there are failures*. Thus, an enhancement of the "static" voting strategy is to allow transactions to run in two modes, normal and failure. When in *normal mode*, transaction T reads one copy of each data-item in its readset, and updates all copies in its writeset. If some copy cannot be updated, T becomes "aware" of a *missing update*, and must run in *failure mode*, in which quorums must be obtained for each data-item in the readset and writeset. This "missing update information" is then passed along to all following transactions that need the information, *i.e.*, all transactions in the precedence graph of future execution connected to T by a path of precedence edges originating at T. These transactions also become aware of missing updates, and must run in failure mode. Since T cannot see the future and does not know what later transactions will be affected, a level of indirection is used: missing update information is posted at sites along with a description of what transactions need the information. When the failure is repaired, the missing update information will eventually be posted at the sites that "caused" the missing updates, *i.e.*, those that did not receive the updates. The updates can then be applied, and postings removed from other sites throughout the system.

The algorithm hinges on the ability to recognize "missing writes", and to propagate the information to later transactions so that cycles in the precedence graph of committed transactions are avoided. Note that certain transactions may be able to execute without restriction even if there are partition failures present in the system; there is no harm in allowing read-only transactions to "run in the past" during a failure, *i.e.*, read an old value of a data-item, as long as no cycles result in the precedence graph of committed transactions. This ability to run in the past allows a site that has become isolated from the rest of the network to execute read-only transactions even if updates are being performed on remote copies of the data-items stored at that site.

EXAMPLE: Suppose that there are four sites in the system A, B, C and D. Sites A, B and C contain copies of data-item x; site B, C and D contain copies of data-item y. Now suppose a failure occurs, isolating sites A and B from site C and D; transactions T_1, T_2, T_3 are initiated at site A (in that order), while transaction T_4 is initiated at D. The readsets, writesets and precedence graph are depicted in Figure 5. (The graph shown is of *uncommitted* transactions since cycles in the graph of *committed* transactions will obviously be avoided.)

T_1 is unaware of the failure, since it can obtain a copy of x and y at A; it can happily run in the past. T_2 becomes aware of the failure when it is unsuccessful at updating the copy of x at C; it is allowed to commit, however, since it can receive a quorum for each data-item in its read and write sets (assuming that each copy has a weight of 1). T_2 is then required to pass all of its missing update information to transactions that are incoming nodes for outgoing edges from T_2, such as T_3 in this example. If T_3 were to successfully commit, it would also be required to pass on the missing update information. However, in this example, T_3 is not allowed to commit; since it is aware of missing updates, it is required to obtain a quorum for data-items in its readset, which it cannot for y (its group only contains the copy y_A). Transaction T_4 would also not be allowed to commit since although it can obtain a quorum for y, it finds that it cannot update the copy of y at B, and must then run in failure mode. Since it cannot obtain a quorum for x, it cannot complete successfully. Thus, in this example (as well in all others), there are no cycles in the graph of *committed* transactions. Note that the restriction that T_2 and T_4 be rerun in failure mode is

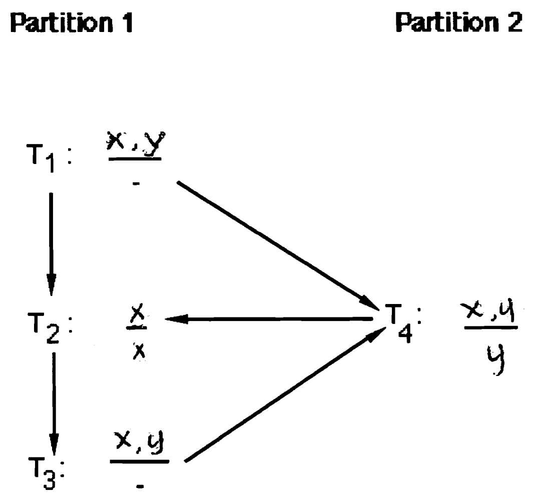

Figure 5: Potential conflict between transactions in different partitions is avoided by requiring transactions aware of missing updates to collect read and write quorums.

necessary: suppose that T_2 and T_4 both read x and y, but T_2 updated x while T_4 updated y. If they both executed in normal mode and did not switch to failure mode when they became aware of missing updates, a cycle would result in the graph of committed transactions.

Using different techniques, this scheme can be generalized to *multiple* levels of quorums. That is, "normal" transaction processing is level 1, and levels 2, 3.... correspond to various vote assignments that optimize availability for different failure modes [Her]. It can also be adapted to use *abstract data type* information for increased availability [Her].

4.3 "One-Copy" Reads

A disadvantage of the previous "failure-mode" quorums is that a read quorum must still be accessed in the event of a partition failure. Virtual partitions [ESC,ET] ensure that a transaction never has to access more than *one copy* for a read operation.

In this partition processing strategy, each site S maintains a "view" (or *virtual partition*) of what sites it believes it can communicate with, called $view(S)$. To be able to process a read (or write) operation on item x, $view(S)$ must contain a read- (or write-) quorum for x. To read x, a transaction executing at S accesses the nearest copy in its view; $w_i(x)$ is translated to writing every copy in $view(S)$. Note that determining whether a read- and write- quorum are available can be determined using local information (*i.e.*, by consulting its view).

Since views only *approximate* the actual state of the network, site S may discover that $view(S)$ is out of date. This may happen, for example, when a transaction executing at S is unable to update a copy in $view(S)$, or when a transaction originating at a site that is not in $view(S)$ attempts to write some copy at S (in which case the write will be rejected). S must then abort all active transactions originating at S and initiate a *creation protocol* to update its view. This protocol

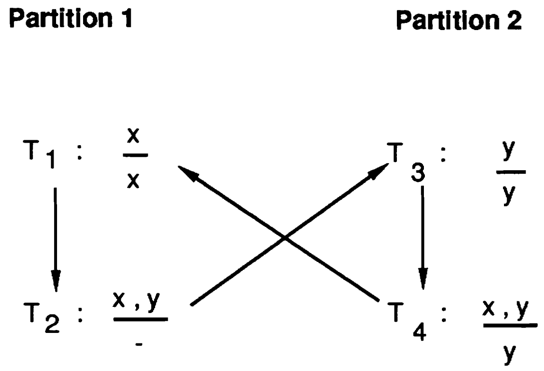

Figure 6: Conflict involving queries.

ensures that all sites in the new view agree on the view, and that all copies of data-items for which there is a read quorum in the new view are up-to-date.[5]

To motivate the correctness of this approach, recall that the basic property of read- and write-quorums is that they are mutually exclusive: If one partition has a read-quorum, then no other partition can have a write-quorum. If site S *believes* it has a write-quorum but in fact does not, the fact that $view(S)$ is incorrect will be discovered at the time a "write-all" is attempted. If site S incorrectly believes it has a read-quorum, however, it may read an *old* value and not discover the mistake until an inaccessible update is performed in its view. Fortunately, these "incorrect" read operations can safely be "run in the past", as in the previous scheme.

5 Querying in the Face of Partitions

In the previous section, we concentrated on the problem of updates, and how to maintain the correctness of data in the face of partition failures. The observant reader will have noticed that even read-only transactions or *queries* for which *all data is available* may be forced to wait if one-copy-serializability is the correctness criteria. For example, consider a database containing two data-items, x and y, that is completely replicated over two sites, A and B. A partition failure occurs so that A and B can no longer communicate. During the failure, transactions T_1 and T_2 execute at site A: T_1 updates x; T_2 then reads the values of x and y. Meanwhile, transactions T_3 and T_4 execute at site B: T_3 updates y; T_4 then reads the values of x and y. Note that the query T_2 requires T_1 to be serialized before T_3 in an equivalent global schedule, while the query T_4 requires T_3 to be serialized before T_1, a contradiction (see Figure 6).

Although such anomalies are theoretically disturbing, in practice queries are frequently allowed to execute without restriction despite the fact that they may see an inconsistent database state. In this section, we will therefore ignore the problem of updates and serializability. That is, we will assume that the database is *static* during the failure, and concentrate on providing approximate answers to queries in the face of network partitions. Furthermore, we will limit our attention to

[5]The creation protocol given in [ESC,ET] tolerates additional partition failures occurring during its execution.

systems in which data is stored as simple tables or *relations*, although the results are applicable to a variety of other types of systems.

5.1 Approximating Queries

Due to the expense and complexity of maintaining replicated data, most distributed database systems limit the amount of replication. Data is fragmented to mirror the usage patterns in the database, and partially replicated to minimize cost and provide fault-tolerance [CP]. Since data is not completely replicated, it is possible to pose queries during network partitioning for which not all data are available.

Queries which require access to unavailable data are typically either delayed until the data becomes available (the partition is repaired), or are aborted with some explanation to the user. If the query is aborted, the user may decide to rephrase the query using only data that is available within the partition. However, this requires a knowledge of system-level issues, such as what nodes are within the same partition group and where data is stored. This not only violates the notion of *transparency of data*, but may be impossible to determine if the state of the network is changing. What is needed is an automatic method of providing an *approximate answer* to the original query, using the available data. The approximate answer should be *monotonic* in the sense that any fact which is said to be true remains true as data structures become available, and any fact which is said to be false remains false. In other words, if we call the answer that would be given if the network were not partitioned the *complete answer*, we require that anything the approximation says to be true be true in the complete answer, and anything the approximation says to be false be false in the complete answer.[6]

To illustrate what we mean by monotonic computation, we will digress from databases for the moment and consider the bisection method for obtaining the root of a function, f. For simplicity, f is assumed to have exactly one root, r. At any point in computation, r is represented by an interval, (a, b), where $a \leq r \leq b$. The computation successively halves the interval: If $f(a) \cdot f(\frac{a+b}{2}) \leq 0$, the new interval becomes $(a, \frac{a+b}{2})$; otherwise the interval becomes $(\frac{a+b}{2}, b)$. Assuming that the initial estimate of r is correct, at any later point in computation the interval is correct: The root is possibly any of the points between a and b, and definitely not any of the points outside of that interval. Furthermore, the answer improves as computation progresses.

Since the answer to a query in a database is a set of tuples, an obvious analogy of an "interval" in the database realm is a *bounding pair* (A, B), where A is a "superset" of the answer T and B is a "subset" of T. A is called a *complete approximation* of T: Every tuple in A is *possibly* an approximation of one or more tuples in T, and anything not approximated by something in A is *definitely not* in T. B is called a *consistent approximation*: Every tuple in B is *definitely* an approximation of one or more tuples in T.

More precisely, we say that a tuple x *approximates* another tuple y (written $x \sqsubseteq y$) if every field (or *attribute*) in x is contained in y and agrees on the value; however, y may contain additional fields as well. For example, if x and y are given as

$$x = [Name \Rightarrow 'John\ Doe'],\ y = [Name \Rightarrow 'John\ Doe'; Age \Rightarrow 21],$$

[6]Recall that we are assuming a *static* database for now, and are ignoring any updates that may be performed in other partitions.

then $x \sqsubseteq y$ since x is not defined on *Age* but agrees on the *Name* value. However, if

$$u = [Name \Rightarrow 'John\ Adams']$$

then neither x nor y are related to u since they disagree on *Name*. Finally, if

$$v = [Name \Rightarrow 'John\ Doe'; CITY \Rightarrow 'Philadelphia'],$$

then $x \sqsubseteq v$, but y and v are unrelated. Extending this ordering to sets, if A, B and T are *sets of tuples*, we say that A is a *complete approximation* of T if for every t in T there exists an a in A such that $a \sqsubseteq t$. B is a *consistent approximation* of T if for every b in B there exists a t in T such that $b \sqsubseteq t$. If A is a complete approximation for T and B is a consistent approximation for the same set T, then (A, B) is a *bounding pair* for T.

Using this ordering of approximation, the *smallest tuple* is [ ]: [ ] approximates every other tuple since no attributes are known. The set containing no tuples (empty set) is the *smallest consistent approximation* for any set T (trivially); and the set containing the smallest tuple, {[ ]}, is the *smallest complete approximation* for any set T since every tuple in T is approximated by [ ].

Note that if a bounding pair (A, B) approximates T, and A, B and T are all *stored* within the distributed database, the statement "(A, B) approximates T" is an integrity constraint, and expresses *implicit redundancy*. Since A is a complete approximation of T, every tuple in T must agree with some tuple in A on common attributes. Since B is a consistent approximation of T, every tuple in B must agree with some tuple in T on common attributes. It is due to this implicit redundancy that we will be able to provide useful, approximate answers in the presence of unavailable data.

In the past five years, attempts to "marry" knowledge-base systems and database systems have provided powerful logic-based languages for expressing a semantic understanding of data in addition to expressing queries [GMN]. In the next section, we will give an example of how a "rule based" system [Ull] could be used to give an approximate answer in the face of partition failures.[7] It should be emphasized that providing approximate answers is a current area of research, and that such techniques are not currently being used in the commercial world.

5.2 Example of an Approximating Query

Suppose we have five tables of information distributed throughout the system: one for teaching fellows, one for graduate students, one for university employees and one each for all the teachers of all sections of the classes CS4 and CS5. (Please see Figures 7, 8 and 9 for the particular example tables we will use.) These tables have the following column entries:

- *TF*- (Teaching Fellows) *Name*, *Salary* and *TPhone*.
- *GS* - (Graduate Students) *Name*, *Degree* and *Phone*
- *UE* - (University Employee) *Name* and *Salary*
- *CS*4 - (Teachers of CS4) *Name*, *Section* and *Office*
- *CS*5 - (Teachers of CS5) *Name*, *Section* and *TPhone*.

Name	*Salary*	*TPhone*
Liza	11K	4093
Joe	11K	8599
Ella	10K	8789
Burt	7K	1423

Figure 7: The teaching fellows (TF) relation.

Name	*Degree*	*Phone*
Joe	PhD	1324
Mary	PhD	3241
Burt	MS	1423
Ella	PhD	4123
Nancy	MS	3214
Chuck	MS	1342
Liza	PhD	1432

Name	*Salary*
Karen	35K
Liza	11K
Steve	60K
Rose	71K
Joe	11K
Ella	10K
Paul	13K
Burt	7K
Edward	40K
Mary	10K

Figure 8: The graduate students (GS), and university employees (UE) relations.

Name	*CS4Section*	*Office*
Joe	2	023
Burt	1	126

Name	*CS5Section*	*TPhone*
Ella	2	8789
Burt	1	1423

Figure 9: The $CS4$ and $CS5$ Relations.

Name	*Salary*	*TPhone*
—	—	—

Name	*Salary*	*TPhone*

Figure 10: The initial bounding pair, (TF_A, TF_B).

For now, we will assume that the tables are all correct and complete (*e.g.*, all university employees are listed in the UE table and everyone listed in UE is a university employee), and that the names listed are unique (*e.g.*, our database does not have more than one person named Burt).

In addition to the tabular data, semantic relationships between the tables have been specified as *rules* of the form

$$p \longrightarrow q.$$

Such a rule can be interpreted as: "Whenever a pattern is found that matches p, then there must be a pattern that matches q". In this example, "patterns" can be thought of as tuples in named relations. The notation "-" for an attribute indicates that the value is unimportant; attributes that must match in value are given the same name variable (*e.g.*, "*Name*" in the first rule). The semantic relationships for this example are:

(a) $TF(Name, Salary, -) \longrightarrow UE(Name, Salary)$: Every teaching fellow is a university employee.

(b) $TF(Name, -, -) \longrightarrow GS(Name, -, -)$: Every teaching fellow is a graduate student.

(c) $CS4(Name, -, -) \longrightarrow TF(Name, -, -)$: Only teaching fellows teach CS4.

(d) $CS5(Name, -, TPhone) \longrightarrow TF(Name, -, TPhone)$: Only teaching fellows teach CS5. Note that the phone number that appears in CS5 is the phone number that appears in TF.

In the absence of any partition failures, these rules act as ***integrity constraints.*** If the system had an integrity subsystem, the appropriate rule(s) would be evaluated at the end of any transaction that might violate the rule (see [Bla,BC] for a discussion of efficiently monitoring integrity constraints). If a violation was detected, the transaction would be aborted. Unfortunately, due to the overhead introduced, few systems implement integrity subsystems of this complexity.

Now suppose that a query to retrieve TF is submitted; unfortunately, a partition failure has occurred and TF is no longer available. We therefore want the system to automatically determine an approximation of TF using the remaining information (GS, UE, $CS4$ and $CS5$). We will now mimic how the system might approximate TF using these rules:

1. Create an initial bounding pair of tables, (TF_A, TF_B). TF_A is the smallest complete approximation of TF, $\{[\,]\}$ (*i.e.*, the table with one entry consisting of null values everywhere), and TF_B the smallest consistent approximation of TF, the empty set (see Figure 10).

[7]For complete definitions of these concepts, along with proofs of correctness, see [BDWa,BDWb].

Name	*Salary*	*TPhone*
Liza	11K	—
Joe	11K	—
Ella	10K	—
Burt	7K	—
Mary	10K	—

Figure 11: The complete approximation, TF_A, after comparison with UE and GS.

Name	*Salary*	*TPhone*
Joe	—	—
Ella	—	8789
Burt	—	1423

Figure 12: The consistent approximation, TF_B, after comparison with $CS4$ and $CS5$.

2. Rule (a) says that every teaching fellow must be a university employee. A restatement of this is that it is only possible for TF to contain tuples that approximate something in UE. We can therefore replace TF_A with the *Name* and *Salary* values of all tuples in the UE table, placing null values for the *TPhone* column.

3. Rule (b) says that every teaching fellow must also be a graduate student. Using similar reasoning, we can therefore cross off any entry in TF_A whose name does not appear in the GS table, *i.e.*, Karen, Steve, Rose, Paul and Edward. Figure 11 represents TF_A at this point.

4. Insert into TF_B the *Name* fields of tuples in $CS4$ with nulls everywhere else, since teachers of CS4 are definitely teaching fellows by Rule (c).

5. Using Rule (d), repeat the previous step for the $CS5$ table, filling in the appropriate *TPhone* value as well as *Name*. If a tuple with the same name already appears in TF_B, just fill in the *TPhone* value (as with Burt, for example). Figure 12 represents TF_B at this point.

6. We now notice that since *Names* are unique, the only possible *Salary* value for tuples in TF_B are those found in the corresponding tuples of TF_A.[8] We can therefore improve the approximations in TF_B by filling in the *Salary* fields. Likewise, we can improve the approximations in TF_A by filling in the appropriate *TPhone* values from TF_B.

At this point, we have done all we can to approximate teaching fellows using the available data, and the final bounding pair is as shown in Figure 13. Referring to Figure 7, note that TF_A is indeed a complete approximation of TF since every name in TF appears in TF_A. Note that in TF_A, the *TPhone* is sometimes undefined; a tuple in TF_A is only an *approximation* of a tuple in TF. TF_B

[8]This is also due to the fact that any other value for *Salary* would violate Rule (a).

Name	*Salary*	*TPhone*
Liza	11K	—
Joe	11K	—
Ella	10K	8789
Burt	7K	1423
Mary	10K	—

Name	*Salary*	*TPhone*
Joe	11K	—
Ella	10K	8789
Burt	7K	1423

Figure 13: The final bounding pair, (TF_A, TF_B).

is also a consistent approximation of TF since every tuple in TF_B is in TF. Again, in TF_B the *TPhone* is sometimes undefined since tuples in TF_B only *approximate* those in TF.

During this process, the system might also identify *anomalies* in the database. For example, if an entry for a student with name "Rose" were added to the CS4 table, the system would detect an anomaly in step (5) outlined above: There is no entry in the complete approximation TF_A for Rose; Rose was crossed off TF_A in step (2) since she is not in the table of graduate students (see Figure 11). This is a violation of the semantic relationships: an entry for a student with name "Rose" should either appear in GS, or CS4 should not contain an entry for a student with name "Rose". This would be caught at the time the incorrect deletion or insertion were made if an integrity subsystem were being used.

It should also be noted that TF is correctly approximated at any point in this computation. For example, if only GS and UE are available, TF is approximated by the bounding pair (A, B) where A is the table in Figure 11, and B is the empty relation. If only $CS4$ and $CS5$ were available, TF is approximated by the the bounding pair with A the relation consisting of one tuple with nulls everywhere (see Figure 10), and B as in Figure 12.

6 Conclusions

Although partition failures may occur infrequently, their effects on distributed systems can be devastating: the availability of data may be severely restricted, and its correctness threatened. In this chapter, we have defined "correct" transaction processing, and have discussed the problems and tradeoffs involved in maintaining correctness. We then presented a class of partition-processing strategies that maintain correctness during partitioning. They have the additional advantage of working when network partitions are not "clean", and do not distinguish between communication failures, site failures, or slow responses. Thus, correctness is maintained even when the cause and extent of partitioning cannot be accurately determined.

Due to the overhead of maintaining the correctness of replicated data, most systems limit the number of copies to at most two or three. During network partitioning, queries may therefore be rejected if the data they access is unavailable. However, data is often "implicitly" replicated; integrity constraints define relationships that must exist between different data-items. Using these relationships, it is possible to construct "approximate" answers to queries even when the data needed is unavailable. While the research in this area is still young, initial results are promising.

References

[AD] P.A. Alsberg and Day. A Principle for Resilient Sharing of Distributed Resources. In *Proc. 2nd International Conf. on Software Engineering*, pages 627–644, 1976.

[BC] P. Buneman and E.K. Clemons. Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems*, 4(3):368–382, September 1979.

[BDWa] P. Buneman, S. Davidson, and A. Watters. A Semantics for Complex Objects and Approximate Queries. *JCSS*, To appear.

[BDWb] P. Buneman, S. Davidson, and A. Watters. Querying Independent Databases. *Information Sciences: An International Journal*, To appear.

[BGa] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *Computing Surveys*, 13(2):185–221, June 1981.

[BGb] P.A. Bernstein and N. Goodman. The Failure and Recovery Problem for Replicated Databases. In *Proc. 2nd ACM Symp. on Princ. of Distributed Computing*, pages 114–122, August 1983.

[BGR*] B. Blaustein, H. Garcia, D.R. Ries, R.M. Chilenskas, and C.W. Kaufman. Maintaining Replicated Databases Even in the Presence of Network Partitions. In *Proc. IEEE 16th Electrical and Aereospace Systems Conf.*, pages 353–360, Sept. 1983.

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

[Bla] B. Blaustein. *Enforcing Database Assertions: Techniques and Applications*. PhD thesis, Harvard University, 1981.

[COK] B. Coan, B. Oki, and E. Kolodner. Limitations on Databse Availability when Networks Partition. In *Proc. 5'th ACM SIGACT-SIGOPS Sym. on Principles of Distributed Computing*, pages 187–194, 1986.

[CP] S. Ceri and G. Pelegatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, New York, 1984.

[Dav] S.B. Davidson. Optimism and Consistency in Partitioned Distributed Database Systems. *ACM Trans. on Database Systems*, 9(3):456–481, Sept. 1984.

[DGS] S.B. Davidson, H. Garcia, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.

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

[ES] D.L. Eager and K.C. Sevcik. Achieving Robustness in Distributed Database Systems. *ACM Trans. on Database Systems*, 8(3):354–381, Sept. 1983.

[ESC] A. El Abbadi, D. Skeen, and F. Cristian. An Efficient, Fault-tolerant Algorithm for Replicated Data Management. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 215–229, March 1985.

[ET] A. El Abbadi and S. Toueg. Availability in Partitioned Replicated Databases. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 240–251, March 1986.

[GAB*] H. Garcia, T. Allen, B. Blaustein, R.M. Chilenskas, and D.R. Ries. Data-Patch: Integrating Inconsistent Copies of a Database After a Partition. In *Proc. Third IEEE Symposium on Reliability in Dist. Software and Database Systems*, pages 38–48, Oct. 1983.

[GB] H. Garcia and D. Barbara. How to Assign Votes is a Distributed System. *Journal of the ACM*, 32(4):841–860, October 1985.

[Gif] D.K. Gifford. Weighted Voting for Replicated Data. In *Proc. of the 7th Symposium on Operating Systems Principles*, pages 150–162, Dec. 1979.

[GK] H. Garcia and B. Kogan. Achieving High Availability in Distributed Databases. *IEEE Trans. on Software Eng.*, 14(7):886–896, July 1988.

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

[GMN] H. Gallaire, J. Minker, and J. Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[GS] D.K. Gifford and A. Spector. The TWA Reservation System. *Comm. of the ACM*, 27(7):650–665, July 1984.

[Her] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986. Also CMU-CS-84-164B.

[KG] B. Kogan and H. Garcia. Update Propagation in Bakunin Data Networks. In *Proc. 6'th ACM SIGACT-SIGOPS Sym. on Principles of Distributed Computing*, pages 13–26, 1987.

[Lam] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LB] N. Lynch and B. Blaustein. Correctness Conditions for Highly Available Replicated Databases. In *Proc. 5'th ACM SIGACT-SIGOPS Sym. on Principles of Distributed Computing*, pages 11–28, 1986.

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

[PPR*] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Trans. on Software Engineering*, 9(3), May 1983.

[Ske] D. Skeen. On Network Partitioning. In *IEEE COMPSAC*, pages 454–455, Nov. 1982.

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

[TGGL] I.L. Traiger, J.N. Gray, C.A. Galtieri, and B.G. Lindsay. Transactions and Consi stency in Distributed Database Systems. *ACM Transactions on Database Systems*, 7(3):323–342, September 1982.

[Tho] R.H. Thomas. A Majority Consensus Approach to Concurrency Control. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[Ull] J.D. Ullman. *Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, 1988.

[Ver] J.S.M. Verhofstad. Recovery Techniques for Database Systems. *ACM Computing Surveys*, 10(2):167–196, 1978.

[Wri] D.D. Wright. *Managing Distributed Databases in Partitioned Networks.* PhD thesis, Cornell University, Dept. of Computer Science, Sept. 1983.