Distributed Database Systems

Article · January 2002

CITATIONS
13

READS
7,633

1 author:

M. Tamer Özsu
University of Waterloo
398 PUBLICATIONS **11,199** CITATIONS

SEE PROFILE

DISTRIBUTED DATABASE SYSTEMS

M. Tamer Özsu
University of Waterloo
Department of Computer Science
Waterloo, Ontario Canada N2L 3G1
{tozsu@db.uwaterloo.ca}

Outline

In this article, we discuss the fundamentals of distributed DBMS technology. We address the data distribution and architectural design issues as well as the algorithms that need to be implemented to provide the basic DBMS functions such as query processing, concurrency control, reliability, and replication control.

Glossary

Atomicity: The property of transaction processing whereby either all the operations of a transaction are executed or none of them are (all-or-nothing).

Client/server architecture: A distributed/parallel DBMS architecture where a set of client machines with limited functionality access a set of servers which manage data.

Concurrency control algorithm: Algorithms that synchronize the operations of concurrent transactions that execute on a shared database.

Distributed database management system: A database management system that manages a database that is distributed across the nodes of a computer network and makes this distribution transparent to the users.

Deadlock: An occurrence where each transaction in a set of transactions circularly waits on locks that are held by other transactions in the set.

Durability: The property of transaction processing whereby the effects of successfully completed (i.e., committed) transactions endure subsequent failures.

Isolation: The property of transaction execution which states that the effects of one transaction on the database are isolated from other transactions until the first completes its execution.

Locking: A method of concurrency control where locks are placed on database units (e.g., pages) on behalf of transactions that attempt to access them.

Logging protocol: The protocol which records, in a separate location, the changes that a transaction makes to the database before the change is actually made.

One copy equivalence: Replica control policy which asserts that the values of all copies of a logical data item should be identical when the transaction that updates that item terminates.

Query optimization: The process by which the ``best'' execution strategy for a given query is found from among a set of alternatives.

Query processing: The process by which a declarative query is translated into low-level data manipulation operations.

Quorum-based voting algorithm: A replica control protocol where transactions collect votes to read and write copies of data items. They are permitted to read or write data items if they can collect a quorum of votes.

Read-Once/Write-All protocol: The replica control protocol which maps each logical read operation to a read on one of the physical copies and maps a logical write operation to a write on all of the physical copies.

Serializability: The concurrency control correctness criterion which requires that the concurrent execution of a set of transactions should be equivalent to the effect of some serial execution of those transactions.

Termination protocol: A protocol by which individual sites can decide how to terminate a particular transaction when they cannot communicate with other sites where the transaction executes.

Transaction: A unit of consistent and atomic execution against the database.

Transparency: Extension of data independence to distributed systems by hiding the distribution, fragmentation and replication of data from the users.

Two-phase commit: An atomic commitment protocol which ensures that a transaction is terminated the same way at every site where it executes. The name comes from the fact that two rounds of messages are exchanged during this process.

Two-phase locking: A locking algorithm where transactions are not allowed to request new locks once they release a previously held lock.

1. Introduction

The maturation of database management system (DBMS) technology has coincided with significant developments in computer network and distributed computing technologies. The end result is the emergence of distributed database management systems. These systems have started to become the dominant data management tools for highly data-intensive applications. Many DBMS vendors have incorporated some degree of distribution into their products.

A *distributed database* (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network. A *distributed database management system* (distributed DBMS) is the software system that permits the management of the distributed database and makes the distribution transparent to the users. The term "distributed database system" (DDBS) is typically used to refer to the combination of DDB and the distributed DBMS. These definitions point to two identifying architectural principles. The first is that the system consists of a (possibly empty) set of *query sites* and a non-empty set of *data sites*. The data sites have data storage capability while the query sites do not. The latter only run the user interface routines in order to facilitate the data access at data sites. The second is that each site (query or data) is assumed to logically consist of a single, independent computer. Therefore, each site has its own primary and secondary storage, runs its own operating system (which may be the same or different at different sites), and has the capability to execute applications on its own. A computer network, rather than a multiprocessor configuration, interconnects the sites. The important point here is the emphasis on loose interconnection between processors that have their own operating systems and operate independently.

2. Data Distribution Alternatives

A distributed database is physically distributed across the data sites by *fragmenting* and *replicating* the data. Given a relational database schema, fragmentation subdivides each relation into horizontal or vertical partitions. *Horizontal fragmentation* of a relation is accomplished by a selection operation that places each tuple of the relation in a different partition based on a fragmentation predicate (e.g., an `Employee` relation may be fragmented according to the location of the employees). *Vertical fragmentation* divides a relation into a number of fragments by projecting over its attributes (e.g., the `Employee` relation may be fragmented such that the `Emp_number`, `Emp_name` and `Address` information is in one fragment, and `Emp_number`, `Salary` and `Manager` information is in another fragment). Fragmentation is desirable because it enables the placement of data in close proximity to its place of use, thus potentially reducing transmission cost, and it reduces the size of relations that are involved in user queries. Based on the user access patterns, each of the fragments may also be *replicated*. This is preferable when the same data are accessed from applications that run at a number of sites. In this case, it may be more cost-effective to duplicate the data at a number of sites rather than continuously moving it between them. Figure 1 depicts a data distribution where `Employee`, `Project` and `Assignment` relations are fragmented, replicated and distributed across multiple sites of a distributed database.

Figure 1 goes here

Figure 1. A Fragmented, Replicated, and Distributed Database Example

3. Architectural Alternatives

There are many possible alternatives for architecting a distributed DBMS. The simplest is the *client/server architecture*, where a number of client machines access a single database server. The simplest client/server systems involve a single server that is accessed by a number of clients (these can be called *multiple-client/single-server*). In this case, the database management problems are considerably simplified since the database is stored on a single server. The pertinent issues relate to the management of client buffers and the caching of data and (possibly) locks. The data management is done *centrally* at the single server. A more distributed, and more flexible, architecture is the *multiple-client/multiple-server* architecture where the database is distributed across multiple servers that have to communicate with each other in responding to user queries and in executing transactions. Each client machine has a "home" server to which it directs user requests. The communication of the servers among themselves is transparent to the users. Most current database management systems implement one or the other type of the client-server architectures. A truly distributed DBMS does not distinguish between client and server machines. Ideally, each site can perform the functionality of a client and a server. Such architectures, called *peer-to-peer*, require sophisticated protocols to manage data that is distributed across multiple sites. The complexity of required software has delayed the offering of peer-to-peer distributed DBMS products.

If the distributed database systems at various sites are autonomous and (possibly) exhibit some form of heterogeneity, they are usually referred to as *multidatabase systems* or *federated database systems*. If the data and DBMS functionality distribution is accomplished on a multiprocessor computer, then it is referred to as a *parallel database system*.

These are different than a distributed database system where the logical integration among distributed data is tighter than is the case with multidatabase systems or federated database systems, but the physical control is looser than that in parallel DBMSs. In this article, we do not consider multidatabase systems or parallel database systems.

4. Overview of Technical Issues

A distributed DBMS has to provide the same functionality that its centralized counterparts provide, such as support for declarative user queries and their optimization, transactional access to the database involving concurrency control and reliability, enforcement of integrity constraints and others. In the remaining sections we discuss some of these functions; in this section we provide a brief overview.

Query processing deals with designing algorithms that analyze queries and convert them into a series of data manipulation operations. Besides the methodological issues, an important aspect of query processing is *query optimization*. The problem is how to decide on a strategy for executing each query over the network in the most cost-effective way, however cost is defined. The factors to be considered are the distribution of data, communication costs, and lack of sufficient locally available information. The objective is to optimize where the inherent parallelism of the distributed system is used to improve the performance of executing the query, subject to the above-mentioned constraints. The problem is NP-hard in nature, and the approaches are usually heuristic.

User accesses to shared databases are formulated as *transactions*, which are units of execution that satisfy four properties: *atomicity*, *consistency*, *isolation*, and *durability* – jointly known as the ACID properties. Atomicity means that a transaction is an atomic

unit and either the effects of all of its actions are reflected in the database, or none of them are. Consistency generally refers to the correctness of the individual transactions; i.e., that a transaction does not violate any of the integrity constraints that have been defined over the database. Isolation addresses the concurrent execution of transactions and specifies that actions of concurrent transactions do not impact each other. Finally, durability concerns the persistence of database changes in the face of failures. ACID properties are enforced by means of concurrency control algorithms and reliability protocols.

Concurrency control involves the synchronization of accesses to the distributed database, such that the integrity of the database is maintained. The concurrency control problem in a distributed context is somewhat different than in a centralized framework. One not only has to worry about the integrity of a single database, but also about the consistency of multiple copies of the database. The condition that requires all the values of multiple copies of every data item to converge to the same value is called *mutual consistency*.

Reliability protocols deal with the termination of transactions, in particular, their behavior in the face of failures. In addition to the typical failure types (i.e., transaction failures and system failures), distributed DBMSs have to account for communication (network) failures as well. The implication of communication failures is that, when a failure occurs and various sites become either inoperable or inaccessible, the databases at the operational sites remain consistent and up to date. This complicates the picture, as the actions of these sites have to be eventually reconciled with those of failed ones. Therefore, recovery protocols coordinate the termination of transactions so that they terminate uniformly (i.e., they either abort or they commit) at all the sites where they execute. Fur-

thermore, when the computer system or network recovers from the failure, the distributed DBMS should be able to recover and bring the databases at the failed sites up-to-date. This may be especially difficult in the case of network partitioning, where the sites are divided into two or more groups with no communication among them.

Distributed databases are typically replicated; that is, a number of the data items reside at more than one site. Replication improves performance (since data access can be localized) and availability (since the failure of a site does not make a data item inaccessible). However, management of replicated data requires that the values of multiple copies of a data item are the same. This is called the *one copy equivalence* property. Distributed DBMSs that allow replicated data implement replication protocols to enforce one copy equivalence.

5. Distributed Query Optimization

Query processing is the process by which a declarative query is translated into low-level data manipulation operations. SQL is the standard query language that is supported in current DBMSs. Query optimization refers to the process by which the "best" execution strategy for a given query is found from among a set of alternatives.

In distributed DBMSs, the process typically involves four steps (Figure 2): *query decomposition*, *data localization*, *global optimization*, and *local optimization*. Query decomposition takes an SQL query and translates it into one expressed in relational algebra. In the process, the query is analyzed semantically so that incorrect queries are detected and rejected as early as possible, and correct queries are simplified. Simplification involves the elimination of redundant predicates that may be introduced as a result of query

modification to deal with views, security enforcement and semantic integrity control. The simplified query is then restructured as an algebraic query.

The initial algebraic query generated by the query decomposition step is input to the second step: data localization. The initial algebraic query is specified on global relations irrespective of their fragmentation or distribution. The main role of data localization is to localize the query's data using data distribution information. In this step, the fragments that are involved in the query are determined and the query is transformed into one that operates on fragments rather than global relations. As indicated earlier, fragmentation is defined through fragmentation rules that can be expressed as relational operations (horizontal fragmentation by selection, vertical fragmentation by projection). A distributed relation can be reconstructed by applying the inverse of the fragmentation rules. This is called a *localization program*. The localization program for a horizontally (vertically) fragmented query is the union (join) of the fragments. Thus, during the data localization step each global relation is first replaced by its localization program, and then the resulting fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition step. As in the decomposition step, the final fragment query is generally far from optimal; the process has only eliminated "bad" algebraic queries.

Figure 2 goes in here

Figure 2. Distributed Query Processing Methodology

For a given SQL query, there is more than one possible algebraic query. Some of these algebraic queries are "better"' than others. The quality of an algebraic query is de-

fined in terms of expected performance. The process of query optimization involves taking the initial algebraic query and, using algebraic transformation rules, transforming it into other algebraic queries until the "best" one is found. The "best" algebraic query is determined according to a cost function that calculates the cost of executing the query according to that algebraic specification. In a distributed setting, the process involves global optimization to handle operations that involve data from multiple sites (e.g., join) followed by local optimization for further optimizing operations that will be performed at a given site.

The input to the third step, global optimization, is a fragment query, that is, an algebraic query on fragments. The goal of query optimization is to find an execution strategy for the query that is close to optimal. Remember that finding the optimal solution is computationally intractable. An execution strategy for a distributed query can be described with *relational algebra operations* and *communication primitives* (send/receive operations) for transferring data between sites. The previous layers have already optimized the query – for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as cardinalities. In addition, communication operations are not yet specified. By permuting the order of operations within one fragment query, many equivalent query execution plans may be found. Query optimization consists of finding the "best" one among candidate plans examined by the optimizer[1].

[1] The difference between an optimal plan and the best plan is that the optimizer does not, because of computational intractability, examine all of the possible plans.

The final step, local optimization, takes a part of the global query (called a subquery) that will run at a particular site and optimizes it further. This step is very similar to query optimization in centralized DBMSs. Thus, it is at this stage that local information about data storage, such as indexes, etc, are used to determine the best execution strategy for that subquery.

The query optimizer is usually modeled as consisting of three components: a search space, a cost model, and a search strategy. The *search space* is the set of alternative execution plans to represent the input query. These plans are equivalent, in the sense that they yield the same result but they differ on the execution order of operations and the way these operations are implemented. The *cost model* predicts the cost of a given execution plan. To be accurate, the cost model must have accurate knowledge about the parallel execution environment. The *search strategy* explores the search space and selects the best plan. It defines which plans are examined and in which order.

In a distributed environment, the cost function, often defined in terms of time units, refers to computing resources such as disk space, disk I/Os, buffer space, CPU cost, communication cost, and so on. Generally, it is a weighted combination of I/O, CPU, and communication costs. Nevertheless, a typical simplification made by distributed DBMSs is to consider communication cost as the most significant factor. This is valid for wide area networks, where the limited bandwidth makes communication much more costly than it is in local processing. To select the ordering of operations it is necessary to predict execution costs of alternative candidate orderings. Determining execution costs before query execution (i.e., static optimization) is based on fragment statistics and the formulas for estimating the cardinalities of results of relational operations. Thus the optimi-

zation decisions depend on the available statistics on fragments. An important aspect of query optimization is *join ordering*, since permutations of the joins within the query may lead to improvements of several orders of magnitude. One basic technique for optimizing a sequence of distributed join operations is through use of the semijoin operator. The main value of the semijoin in a distributed system is to reduce the size of the join operands and thus the communication cost. However, more recent techniques, which consider local processing costs as well as communication costs, do not use semijoins because they might increase local processing costs. The output of the query optimization layer is an optimized algebraic query with communication operations included on fragments.

6. Distributed Concurrency Control

Whenever multiple users access (read and write) a shared database, these accesses need to be synchronized to ensure database consistency. The synchronization is achieved by means of *concurrency control algorithms* that enforce a correctness criterion such as *serializability*. User accesses are encapsulated as transactions, whose operations at the lowest level are a set of read and write operations to the database. Concurrency control algorithms enforce the *isolation* property of transaction execution, which states that the effects of one transaction on the database are isolated from other transactions until the first completes its execution.

The most popular concurrency control algorithms are *locking*-based. In such schemes, a lock, in either shared or exclusive mode, is placed on some unit of storage (usually a page) whenever a transaction attempts to access it. These locks are placed according to lock compatibility rules such that *read-write*, *write-read*, and *write-write* con-

flicts are avoided. It is a well known theorem that if lock actions on behalf of concurrent transactions obey a simple rule, then it is possible to ensure the serializability of these transactions: "No lock on behalf of a transaction should be set once a lock previously held by the transaction is released." This is known as *two-phase locking*, since transactions go through a growing phase when they obtain locks and a shrinking phase when they release locks. In general, releasing of locks prior to the end of a transaction is problematic. Thus, most of the locking-based concurrency control algorithms are *strict* in that they hold on to their locks until the end of the transaction.

In distributed DBMSs, the challenge is to extend both the serializability argument and the concurrency control algorithms to the distributed execution environment. In these systems, the operations of a given transaction may execute at multiple sites where they access data. In such a case, the serializability argument is more difficult to specify and enforce. The complication is due to the fact that the serialization order of the same set of transactions may be different at different sites. Therefore, the execution of a set of distributed transactions is serializable if and only if

1. the execution of the set of transactions at each site is serializable, and

2. the serialization orders of these transactions at all these sites are identical.

Distributed concurrency control algorithms enforce this notion of *global serializability*. In locking-based algorithms there are three alternative ways of enforcing global serializability: centralized locking, primary copy locking, and distributed locking algorithm.

In *centralized locking*, there is a single lock table for the entire distributed database. This lock table is placed, at one of the sites, under the control of a single lock manager. The lock manager is responsible for setting and releasing locks on behalf of transactions. Since all locks are managed at one site, this is similar to centralized concurrency control and it is straightforward to enforce the global serializability rule. These algorithms are simple to implement, but suffer from two problems. The central site may become a bottleneck, both because of the amount of work it is expected to perform and because of the traffic that is generated around it; and the system may be less reliable since the failure or inaccessibility of the central site would cause system unavailability. *Primary copy locking* is a concurrency control algorithm that is useful in replicated databases where there may be multiple copies of a data item stored at different sites. One of the copies is designated as a primary copy and it is this copy that has to be locked in order to access that item. All the sites know the set of primary copies for each data item in the distributed system, and the lock requests on behalf of transactions are directed to the appropriate primary copy. If the distributed database is not replicated, copy locking degenerates into a distributed locking algorithm.

In *distributed* (or *decentralized*) *locking*, the lock management duty is shared by all the sites in the system. The execution of a transaction involves the participation and coordination of lock managers at more than one site. Locks are obtained at each site where the transaction accesses a data item. Distributed locking algorithms do not have the overhead of centralized locking ones. However, both the communication overhead to obtain all the locks, and the complexity of the algorithm are greater.

One side effect of all locking-based concurrency control algorithms is that they cause *deadlocks*. The detection and management of deadlocks in a distributed system is difficult. Nevertheless, the relative simplicity and better performance of locking algorithms make them more popular than alternatives such as *timestamp-based algorithms* or *optimistic concurrency control*.

7. Distributed Reliability Protocols

Two properties of transactions are maintained by reliability protocols: *atomicity* and *durability*. Atomicity requires that either all the operations of a transaction are executed or none of them are (all-or-nothing property). Thus, the set of operations contained in a transaction is treated as one atomic unit. Atomicity is maintained in the face of failures. Durability requires that the effects of successfully completed (i.e., committed) transactions endure subsequent failures.

The underlying issue addressed by reliability protocols is how the DBMS can continue to function properly in the face of various types of failures. In a distributed DBMS, four types of failures are possible *transaction failures*, *site* (*system*) *failures*, *media* (*disk*) *failures* and *communication failures*. Transactions can fail for a number of reasons: due to an error in the transaction caused by input data or by an error in the transaction code, or the detection of a present or potential deadlock. The usual approach to take in cases of transaction failure is to abort the transaction, resetting the database to its state prior to the start of the database.

Site (or system) failures are due to a hardware failure (eg, processor, main memory, power supply) or a software failure (bugs in system code). The effect of system failures is the loss of main memory contents. Therefore, any updates to the parts of the database that are in the main memory buffers (also called *volatile database*) are lost as a result of system failures. However, the database that is stored in secondary storage (also called *stable database*) is safe and correct. To achieve this, DBMSs typically employ *logging protocols*, such as Write-Ahead Logging, which record changes to the database in system logs and move these log records and the volatile database pages to stable storage at appropriate times. From the perspective of distributed transaction execution, site failures are important since the failed sites cannot participate in the execution of any transaction.

Media failures refer to the failure of secondary storage devices that store the stable database. Typically, these failures are addressed by introducing redundancy of storage devices and maintaining archival copies of the database. Media failures are frequently treated as problems local to one site and therefore are not specifically addressed in the reliability mechanisms of distributed DBMSs.

The three types of failures described above are common to both centralized and distributed DBMSs. Communication failures, on the other hand, are unique to distributed systems. There are a number of types of communication failures. The most common ones are errors in the messages, improperly ordered messages, lost (or undelivered) messages, and line failures. Generally, the first two of these are considered to be the responsibility of the computer network protocols and are not addressed by the distributed DBMS. The last two, on the other hand, have an impact on the distributed DBMS protocols and, therefore, need to be considered in the design of these protocols. If one site is

expecting a message from another site and this message never arrives, this may be because (a) the message is lost, (b) the line(s) connecting the two sites may be broken, or (c) the site that is supposed to send the message may have failed. Thus, it is not always possible to distinguish between site failures and communication failures. The waiting site simply timeouts and has to assume that the other site is incommunicado. Distributed DBMS protocols have to deal with this uncertainty. One drastic result of line failures may be network partitioning in which the sites form groups where communication within each group is possible but communication across groups is not. This is difficult to deal with in the sense that it may not be possible to make the database available for access while at the same time guaranteeing its consistency.

The enforcement of atomicity and durability requires the implementation of *atomic commitment protocols* and *distributed recovery protocols.* The most popular atomic commitment protocol is *two-phase commit*. The recoverability protocols are built on top of the local recovery protocols, which are dependent upon the supported mode of interaction (of the DBMS) with the operating system.

Two-phase commit (2PC) is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent (i.e., all sites terminate the transaction in the same manner). If all the sites agree to commit a transaction then all the actions of the distributed transaction take effect; if one of the sites declines to commit the operations at that site, then all of the other sites are required to abort the transaction. Thus, the fundamental 2PC rule states:

1. If even one site rejects to commit (which means it votes to abort) the transaction, the distributed transaction has to be aborted at each site where it executes; and

2. If all the sites vote to commit the transaction, the distributed transaction is committed at each site where it executes.

The simple execution of the 2PC protocol is as follows (Figure 3). There is a *coordinator* process at the site where the distributed transaction originates, and *participant* processes at all the other sites where the transaction executes. Initially, the coordinator sends a "prepare" message to all the participants each of which independently determines whether or not it can commit the transaction at that site. Those that can commit send back a "vote-commit" message while those who are not able to commit send back a "vote-abort" message. Once a participant registers its vote, it cannot change it. The coordinator collects these messages and determines the fate of the transaction according to the 2PC rule. If the decision is to commit, the coordinator sends a "global-commit" message to all the participants; if the decision is to abort, it sends a "global-abort" message to those participants who had earlier voted to commit the transaction. No message needs to be sent to those participants who had originally voted to abort since they can assume, according to the 2PC rule, that the transaction is going to be eventually globally aborted. This is known as the "unilateral abort" option of the participants.

Figure 3 goes in here

Figure 3. 2PC Protocol Actions

There are two rounds of message exchanges between the coordinator and the participants; hence the name 2PC protocol. There are a number of variations of 2PC, such as the linear 2PC and distributed 2PC, that have not found much favor among distributed DBMS vendors. Two important variants of 2PC are the *presumed abort 2PC* and *presumed commit 2PC*. These are important because they reduce the message and I/O overhead of the protocols. Presumed abort protocol is included in the X/Open XA standard and has been adopted as part of the ISO standard for Open Distributed Processing.

One important characteristic of 2PC protocol is its *blocking* nature. Failures can occur during the commit process. As discussed above, the only way to detect these failures is by means of a time-out of the process waiting for a message. When this happens, the process (coordinator or participant) that timeouts follows a *termination protocol* to determine what to do with the transaction that was in the middle of the commit process. A non-blocking commit protocol is one whose termination protocol can determine what to do with a transaction in case of failures under any circumstance. In the case of 2PC, if a site failure occurs at the coordinator site and one participant site while the coordinator is collecting votes from the participants, the remaining participants cannot determine the fate of the transaction among themselves, and they have to remain blocked until the coordinator or the failed participant recovers. During this period, the locks that are held by the transaction cannot be released, which reduces the availability of the database. There have been attempts to devise non-blocking commit protocols (e.g., three-phase commit) but the high overhead of these protocols has precluded their adoption.

The inverse of termination is recovery. When a failed site recovers from the failure, what actions does it have to take to recover the database at that site to a consistent state?

This is the domain of *distributed recovery protocols*. If each site can look at its own log and decide what to do with the transaction, then the recovery protocol is said to be *independent*. For example, if the coordinator fails after it sends the "prepare" command and while waiting for the responses from the participants, upon recovery, it can determine from its log where it was in the process and can restart the commit process for the transaction from the beginning by sending the "prepare" message one more time. If the participants had already terminated the transaction, they can inform the coordinator. If they were blocked, they can now resend their earlier votes and resume the commit process. However, this is not always possible and the failed site has to ask others for the fate of the transaction.

8. Replication Protocols

In replicated distributed databases, each logical data item has a number of physical instances. For example, the salary of an employee (*logical data item*) may be stored at three sites (*physical copies*). The issue in this type of a database system is to maintain some notion of consistency among the copies. The most discussed consistency criterion is *one copy equivalence*, which asserts that the values of all copies of a logical data item should be identical when the transaction that updates it terminates.

If replication transparency is maintained, transactions will issue read and write operations on a logical data item x. The replica control protocol is responsible for mapping operations on x to operations on physical copies of x (x_1, …, x_n). A typical replica control protocol that enforces one copy equivalence is known as *Read-Once/Write-All* (ROWA) protocol. ROWA maps each read on x [Read(x)] to a read on one of the physical copies x_i

[Read(x_i)]. The copy that is read is insignificant from the perspective of the replica control protocol and may be determined by performance considerations. On the other hand, each write on logical data item x is mapped to a set of writes on all copies of x.

ROWA protocol is simple and straightforward, but it requires that all copies of all logical data items that are updated by a transaction be accessible for the transaction to terminate. Failure of one site may block a transaction, reducing database availability.

A number of alternative algorithms have been proposed which reduce the requirement that all copies of a logical data item be updated before the transaction can terminate. They relax ROWA by mapping each write to only a subset of the physical copies. The *majority consensus algorithm* is one such algorithm which terminates a transaction as long as a majority of the copies can be updated. Thus, all the copies of a logical data item may not be updated to the new value when the transaction terminates.

This idea of possibly updating only a subset of the copies, but nevertheless successfully terminating the transaction, has formed the basis of quorum-based voting for replica control protocols. The majority consensus algorithm can be viewed from a slightly different perspective: It assigns equal votes to each copy and a transaction that updates that logical data item can successfully complete as long as it has a majority of the votes. Based on this idea, an early *quorum-based voting algorithm* assigns a (possibly unequal) vote to each copy of a replicated data item. Each operation then has to obtain a *read quorum* (V_r) or a *write quorum* (V_w) to read or write a data item, respectively. If a given data item has a total of V votes, the quorums have to obey the following rules:

1. $V_r + V_w > V$ (a data item is not read and written by two transactions concurrently, avoiding the read-write conflict);

2. $V_w > V_r /2$ (two write operations from two transactions cannot occur concurrently on the same data item; avoiding write-write conflict).

The difficulty with this approach is that transactions are required to obtain a quorum even to read data. This significantly and unnecessarily slows down read access to the database. An alternative quorum-based voting protocol that overcomes this serious performance drawback has also been proposed. However, this protocol makes unrealistic assumptions about the underlying communication system. It requires that all sites detect failures that change the network's topology instantaneously, and that each site has a view of the network consisting of all the sites with which it can communicate. In general, communication networks cannot guarantee to meet these requirements. The single copy equivalence replica control protocols are generally considered to be restrictive in terms of the availability they provide. Voting-based protocols, on the other hand, are considered too complicated with high overheads. Therefore, these techniques are not used in current distributed DBMS products. More flexible replication schemes have been investigated where the type of consistency between copies is under user control. A number of *replication servers* have been developed or are being developed with this principle. Unfortunately, there is no clear theory that can be used to reason about the consistency of a replicated database when the more relaxed replication policies are used.

Bibliography

Bernstein, P. A. and Newcomer, E. 1997. *Principles of Transaction Processing for the Systems Professional*. Morgan Kaufmann, San Mateo, California.

Gray, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California.

Helal, A. A. , Heddaya, A. A. and Bhargava, B. B. 1997. *Replication Techniques in Distributed Systems*, Kluwer Academic Publishers, Boston, MA.

Kumar, V. (ed.). 1996. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, Englewood Cliffs, NJ.

Özsu, M.T. and Valduriez, P. 1999. *Principles of Distributed Database Systems, 2nd edition*, Prentice-Hall, Englewood Cliffs, NJ.

Sheth, A. and Larson, J. September 1990. "Federated Databases: Architectures and Integration", *ACM Computing Surveys,* Vol. 22, No. 3, pages 183-236.

Yu, C. and Meng, W. 1998. *Principles of Query Processing for Advanced Database Applictions*, Morgan Kaufmann, San Francisco, California.