Approaches to concurrency control in distributed data base systems*

by PHILIP A. BERNSTEIN and NATHAN GOODMAN
Harvard University
Cambridge, Massachusetts

INTRODUCTION

Whenever multiple users or programs access a data base concurrently, the problem of concurrency control arises. The problem is to synchronize concurrent interactions so that each reads consistent data from the data base, writes consistent data, and is ultimately processed to completion. In a distributed data base this problem is exacerbated because a concurrency control mechanism at one site cannot instantaneously know about interactions at other sites. No fewer than 30 papers on this topic have appeared to date. Our purpose is to survey this literature, concentrating on three approaches—locking, majority consensus, and SDD-1 protocols—which together subsume the bulk of the literature.*

Distributed concurrency control is complex and our treatment is, of necessity, sketchy. We urge the interested reader to consult the source materials listed in the bibliography.

BACKGROUND

Preliminary definitions

A distributed data base management system (abbr. DDBMS) is a collection of sites interconnected by a network. Each site is a computer running a local (i.e. non-distributed) DBMS, and the network is any computer-to-computer communication system. We assume that sites are widely dispersed geographically, so the network must employ long-distance communication media. Consequently, inter-site communication is qualitatively slower and more costly than intra-site computation.

We define a data base to be a collection of data items. In practice, a data item may be a field, a record, a file, etc. This “level of granularity” is important, but does not impact concurrency control and so we leave it unspecified.

Each data item may be stored at any site in the system, and moreover each may be stored redundantly at several sites. Redundant data improves the robustness and performance of a DDBMS and must be supported by general purpose systems. Unfortunately, it is also a major source of complexity. A stored copy of a data item is called a stored data item. Though it is impossible for all stored copies of a data item to be identical at every instant of time, it is essential that all “converge” to the same final value. We use the term logical data item when the distinction between “data item” and “stored data item” requires emphasis.

Users interact with the DDBMS by entering transactions, by which we mean a program or on-line query that accesses the data base. Transactions have two important properties in our model—(1) We assume they represent complete and correct computations: i.e. each transaction, if executed alone on an initially consistent data base, would terminate, output correct results, and leave the data base consistent. (2) We assume transactions obtain data from the data base by issuing Read commands to the DDBMS, and modify data by issuing Write commands. The arguments to these commands are logical data items and it is the responsibility of the DDBMS (a) to choose one stored copy of each data item for Reads, and (b) to update all stored copies of each data item for Writes. We model a transaction as a sequence of Read and Write operations paying no attention to its internal computations.

The read-set of a transaction is the set of logical data items it reads, and its write-set is the set of logical data items it writes. Two transactions conflict if the write-set of one intersects the read-set or write-set of the other. Similarly, two operations conflict if one is a Write and they operate on the same data. It is a fundamental theorem of concurrency control that two transactions require synchronization only if they conflict. (The converse need not be true, as we shall see in the fifth section).

Serializability

A log is a sequence of Reads and Writes. A log is serial if the Reads and Writes for each transaction are contiguous (see Figure 1). Such a log represents an execution in which no transactions execute concurrently. Since we assume each

* This work was supported by the National Science Foundation under Grant MCS-77-05314 and by the Advanced Research Projects Agency of the Department of Defense, contract number N00039-78-G-0020.
** References on these approaches are listed in the bibliography by topic. We will limit our use of in-text references in the interest of readability.
transaction preserves consistency if executed alone, a serial sequence of transactions also preserves consistency. A log is serializable (abbr. SR) if it is "equivalent" to a serial log, meaning that for all initial data base states it produces the same output and the same final data base state as some serial log. Since serial logs preserve consistency, and SR logs are equivalent to serial logs, SR logs preserve consistency as well.

In a DDBMS, each site processes a different log. We define a distributed log (abbr. dlog) to be a set of logs, one per site. A serial dlog is a dlog in which each component log is serial and reflects the same total ordering of transactions (i.e., all transactions are in the same relative order in all logs in which they appear). A dlog is serializable if it is equivalent to a serial dlog.

Serializability has been adopted almost universally as the correctness criterion for DBMS concurrency control; all the approaches we describe follow this convention. Alternate correctness criteria are discussed in References 20 and 22.

Other aspects of concurrency control

In addition to ensuring serializability, a concurrency controller must guarantee termination; it must operate robustly; and it must operate efficiently.

A transaction may fail to terminate for one of three reasons—(1) Deadlock may occur, i.e. two or more operations might be forced to wait for each other. (2) Some operation may be indefinitely postponed by an unexpected conspiracy of events. Or (3) Cyclic restart might be experienced, meaning that the transaction repeatedly reaches a blocked state and is aborted and restarted. Every concurrency control approach is susceptible to some combination of these problems.

With respect to robustness, all approaches face essentially identical problems. We discuss this issue in the sixth section.

The efficiency of a distributed concurrency controller is determined principally by how much inter-site communication it requires. Typically, message delays in long distance networks are tenths of seconds, and network capacity is the scarcest system resource. In analyzing the performance of a controller, then, it is reasonable to study its communication behavior, and ignore other aspects. We compare the performance of various approaches in the Conclusion section.

DISTRIBUTED LOCKING ALGORITHMS

Locking is the most widely used concurrency control technique. We describe locking first in the centralized DBMS context and then present several extensions for distributed systems.

Centralized locking

Locking synchronizes transactions by explicitly detecting and preventing conflicts. When a transaction issues a Read or Write command, the DBMS attempts to "set a lock" on the desired data item; the lock is "granted" only if no other transaction holds a conflicting lock. If the lock is not granted, the requesting transaction waits until the lock is available and can be granted.

Since the DBMS processes all Read and Write commands from every transaction, it can automatically generate lock requests for each command. This is important because it allows programmers to ignore concurrency control issues when writing their transactions.

Eswaran et al. prove that locking is sufficient to ensure serializability provided no transaction requests new locks after releasing a lock. This usually amounts to having transactions hold all locks until they finish execution.

Since transactions are made to wait for unavailable locks, the possibility of deadlock exists (see Figure 2). Deadlocks can be detected by maintaining a deadlock graph in the DBMS. The nodes of the graph represent transactions and the arcs represent the "waiting-for" relationship; an arc is drawn from transaction \( T_i \) to transaction \( T_j \) if \( T_i \) is waiting for a lock held by \( T_j \).
Approaches to Concurrency Control in Distributed Data Base Systems

Primary site locking

Primary site locking is a simple extension of centralized locking. One site of a DDBMS is designated to be the 'primary site' and it manages all synchronization. When a transaction wishes to access data at any site, a lock is requested from the primary site. The primary site processes lock requests exactly as described in the previous section, the only difference being that lock requests come in over the network. Similarly, issues of termination are handled by the primary site exactly as in centralized locking.

Although locks are centralized at the primary site, the data base is, of course, distributed. Once a transaction is granted a lock, it may access data at whatever site has a copy. It is important that if a transaction updates a data item that has many stored copies all copies are actually updated before the lock is released; otherwise another transaction can read a copy of the data item before the first update propagated there. It is also important that read-only transactions follow the locking discipline, or else they could read inconsistent data (see Figure 4). This point is often overlooked in discussions of distributed locking, yet is important because most applications predominantly consist of read-only transactions.

The principal drawback of primary site locking is that the primary site tends to be a bottleneck—the capacity of the primary site to process locks bounds the capacity of the entire distributed system.

Primary copy locking

Primary copy locking is an extension of primary site locking that eliminates the primary site bottleneck. For each logical data item, one copy is designated the "primary copy"; when a transaction wishes to access a data item, it locks the primary copy. Since the primary copies of different data items may be stored at different sites, no single site is primary in any sense. This eliminates the bottleneck, but introduces a new problem—deadlock detection.

To test for deadlock, all sites with some primary copy must participate. For example, Figure 5 illustrates a deadlock involving two sites which cannot be recognized locally by either site. The solution is to designate one site of the DDBMS as the "deadlock detector"; periodically each other site sends it a list of newly granted or released locks, and newly pending requests. The deadlock detector then operates as in the centralized case.

As with primary site locking, if a transaction writes into a data item, all copies must be updated before the lock is

`Transactions
T_1: Write(x); Write(y);
end
T_2: Read(x); Read(y);
end`

Order in which operations executed

Database

Figure 4—Read-only transactions must lock, too.
Transactions

\[ T_1: \text{Read}(x); \quad T_2: \text{Read}(y); \quad T_3: \text{Read}(z); \]
\[ \text{Write}(y); \quad \text{Write}(z); \quad \text{Write}(x) \]

end \hspace{1cm} \text{end} \hspace{1cm} \text{end}

Order in which locks are requested at each site

\begin{itemize}
  \item site 1
    \begin{itemize}
      \item lock x for R_1
      \item lock x for W_3
    \end{itemize}
  \item site 2
    \begin{itemize}
      \item lock y for R_2
      \item lock y for W_1
    \end{itemize}
  \item site 3
    \begin{itemize}
      \item lock z for R_3
      \item lock z for W_2
    \end{itemize}
\end{itemize}

None of the \textit{*}ed locks can be granted, hence the system is in deadlock. But deadlock graphs at each site are acyclic:

\begin{itemize}
  \item site 1
    \begin{itemize}
      \item T_3 \rightarrow T_1
    \end{itemize}
  \item site 2
    \begin{itemize}
      \item T_1 \rightarrow T_2
    \end{itemize}
  \item site 3
    \begin{itemize}
      \item T_2 \rightarrow T_3
    \end{itemize}
\end{itemize}

Figure 5—Multi-site deadlock.

Conflict-driven restarts

An interesting variant of primary copy locking has been described by Rosenkrantz, Lewis and Stearns.\(^\text{30}\) The mechanism, which we call conflict-driven restart, uses a model of transaction execution in which each transaction is active at only one site at a time and moves from site to site during its execution. When a transaction wishes to access a data item at a site, it tests whether it conflicts with a previous access made by an in-progress transaction. If it does conflict, one of three actions is possible—it waits, it is restarted, or the other transaction is restarted. Since testing for conflict is equivalent to asking whether a data item is \textit{locked}, this approach is essentially a locking mechanism.\(^\text{12}\)

The analysis of conflict-driven restart yields interesting observations about termination problems. If the system responds to conflict by making the requesting transaction wait, deadlock is possible. To avoid deadlock Rosenkrantz et al. propose two mechanisms that substitute restarts for waiting. Both mechanisms require that transactions be assigned unique "timestamps" when they are submitted. Intuitively, timestamps correspond to the time a transaction was submitted, and have two important properties—timestamps assigned at any particular site must strictly increase with time, and timestamps assigned at different sites must be different. Timestamps are used to resolve conflicts as follows. In one mechanism, called the \textit{Wait-Die System}, the requesting transaction waits if it has a smaller timestamp (i.e. is older); else it is restarted. In the second mechanism, called the \textit{Wound-Wait System}, the requesting transaction waits if it has a larger timestamp (i.e., is younger); else the transaction it conflicts with is restarted.

Rosenkrantz et al. prove that both mechanisms avoid cyclic restart, but the details of their behavior is quite different. In a Wound-Wait system, an old transaction may be restarted many times, whereas in a Wait-Die system old transactions are never restarted. It is suggested that Wound-Wait produces fewer overall restarts, but the justification is more intuitive than analytic.

MAJORITY CONSENSUS ALGORITHM

The majority consensus algorithm of R. Thomas\(^\text{37}\) was one of the first distributed concurrency control mechanisms proposed. Many of Thomas's ideas have found their way into more recent designs.

The majority consensus algorithm as presented by Thomas assumes a fully redundant data base, meaning that every site has a stored copy of every logical data item in the data base. A transaction executes at one site. Its Read commands access stored data at its site, and do so \textit{without} locking or any other synchronization. Whenever the transaction issues a Write command, the name of the data item being updated and its new value are recorded in an \textit{update list}: the data base itself is \textit{not} modified at this time. When the transaction completes, the update list is sent to all sites and each site votes on it. If a majority of the sites vote "Yes," the transaction is \textit{accepted}, and the updates are installed at all sites. Otherwise the transaction is restarted. The heart of the algorithm is the rules that determine how each site votes.

A site votes "Yes" on transaction \(T\) if

1. The data items read by \(T\) have not been modified since \(T\) read them (the algorithm requires that a data item must be read before it can be written).
2. \(T\) does not conflict with any transaction \(T'\) that is \textit{pending} at the site (\(T'\) is \textit{pending} if the site has voted "Yes" but \(T'\) has not yet been accepted or rejected system-wide).

From the collection of the Computer History Museum (www.computerhistory.org)
One way to meet Condition 1 is to use locking; but the majority consensus algorithm uses a timestamping technique instead.

Transactions are assigned timestamps as in "conflict driven restart," and each stored data item is tagged with the timestamp of the most recent transaction that has updated it. Also, update lists are augmented to include the name of each data item read by the transaction and its timestamp. Now, when a site receives an update list it can compare timestamps to determine whether Condition 1 holds. Since augmented updated lists specify transactions’ read-sets and write-sets, Condition 2 is easily checked as well.

If Condition 1 is not satisfied, the site "veto's" the transaction and it is restarted. If (1) is satisfied but (2) is not, the site cannot vote on this transaction until the pending one is resolved. Since different sites receive update lists in different orders, they vote in different orders and deadlock could result. To avoid deadlock, the site votes “No” if (1) holds, (2) does not hold, and the transaction has a larger timestamp (i.e. is younger) than the pending one. If a majority of sites vote "No," the transaction is restarted.

The voting rules ensure that two conflicting transactions are both accepted only if one has read the other’s output. Since both transactions received a majority of “Yes” votes, some site, say $S$, must have voted “Yes” on both transactions. Since they conflict, $S$ must have installed one before voting on the other; this guarantees that the second read the first one’s output, for otherwise $S$ would not have voted “Yes.” This is sufficient to guarantee serializability.

THE SDD-1 APPROACH

The SDD-1 DDBMS employs a qualitatively different approach to concurrency control. Each of the preceding methods synchronizes all conflicting Reads and Writes. However, not all conflicts can violate serializability (see Figure 6). SDD-1 exploits this fact by means of two mechanisms—conflict graph analysis, and timestamp-based protocols.

Conflict graph analysis

Conflict graph analysis is a technique for determining which conflicts require synchronization. The method begins with the definition of transaction classes. A transaction class is defined by a read-set and write-set. A transaction is a member of a class if the transaction’s read-set and write-set are contained in the class’s read-set and write-set (respectively). Associated with each class is a transaction module (abbr. TM), a software DBMS component that serially processes transactions from that class. Since transactions in a single class run serially, only transactions in different classes can "interfere." Hence, only inter-class conflicts need be considered.

Due to the way classes are defined, transactions in different classes can conflict only if their corresponding classes conflict. Class conflicts are modelled by an undirected conflict graph whose nodes represent class read-sets and write-sets, and whose edges represent conflicts. (There is also an edge between the read-set and write-set of each individual class. See Figure 7.) The important property of a conflict graph is that transactions that do not lie on a cycle are always serializable and do not need synchronization. Only transactions that lie on cycles require synchronization.

In a conflict graph system, the conflict graph is constructed and analyzed statically when the data base and classes are defined. Classes that do not lie on cycles are noted; the TMs corresponding to these classes are 'told' not to Class definitions:

- $C_1$: read-set = \{x,y\}, write-set = \{z\}
- $C_2$: read-set = \{x\}, write-set = \{y\}

Confict graph

\[
\begin{align*}
C_1' & \text{'s read-set } \{x,y\} & C_2' & \text{'s read-set } \{x\} \\
C_1' & \text{'s write-set } \{z\} & C_2' & \text{'s write-set } \{y\}
\end{align*}
\]

Note: Transactions $T_1$ and $T_2$ in figure 6 are in classes $C_1$ and $C_2$ resp. Since the conflict graph is acyclic, their conflict cannot violate serializability (see text).

Figure 7—Conflict graph.
to use synchronization when executing transactions. The remaining TMs must synchronize their transactions.

Locking is one correct way to synchronize transactions that lie on cycles. If all transactions on cycles use locking, all executions are serializable. However, other synchronization mechanisms are possible.

**Timestamp-based protocols**

SDD-1 uses timestamp-based synchronization protocols in place of locking. While the details of these protocols are too involved for this paper, their basic structure can be sketched.

Each edge from one class’s read-set to another’s write-set represents a conflict that must be synchronized if the edge lies in a cycle. This synchronization occurs during the processing of Read commands. Suppose the read-set of class i conflicts with the write-set of class j, and suppose T_i is a transaction in class i. To process a Read for T_i at site S, the concurrency controller waits until S has processed all Writes for transactions in class j that are ‘older’ than T_i,*** but no Writes for transactions in class j that are ‘younger’ than T_i. This has the same effect as locking the data shared by i’s read-set and j’s write-set.

The advantage of timestamp-based protocols lies in the wide range of protocols that can be used. There is a special protocol for read-only transactions which is more efficient than locking. There is a special protocol for infrequently run transactions that places a heavier synchronization burden on these transactions while reducing the synchronization required for common transactions. In addition, all protocols use timestamps to resolve conflicts, so deadlock is prevented without the overhead of a detection algorithm.

The correctness of conflict graph analysis and the SDD-1 protocols is proved in Reference 41.

**ROBUSTNESS**

Component failures are inevitable in a DDBMS, and any practical concurrency controller must operate correctly despite them. Problems of three types arise—(1) A failed site may hold information needed to synchronize in-progress transactions. (2) A failed site may hold stored copies of data items being updated by a transaction. (3) A transaction that is updating data at several sites may fail after performing some updates but not all of them. No mechanism yet developed attains 100 percent robustness and it is believed that no such mechanism is possible. Given this apparent limitation, one cannot prove that a concurrency controller is, or is not, robust: all one can do is express the level of robustness it attains.

*** SDD-1 assigns unique timestamps to transactions in the same manner as majority consensus.

**Loss of synchronization information**

When a site holding synchronization information fails, there are two options. One is to abort all in-progress transactions that depend on the information. If transactions are short and failures occur infrequently, this simple approach is satisfactory. The alternative is to maintain redundant copies of synchronization information. Techniques for managing these redundant copies have been proposed by Alsuberg and Day and Menasce et al. The techniques are presented in the context of primary site or primary copy locking, but could be adapted for other approaches.

The Alsuberg and Day technique employs a back-up for each primary copy. When transaction T_i wishes to access data item x, it requests a lock against the primary copy as described in the section on Primary Copy Locking. If the concurrency controller decides to grant the lock, it forwards this information to the back-up, which records the lock in memory. Only when the lock is safely recorded at the back-up is transaction T_i permitted to access x. If the site containing the primary copy fails, the back-up can immediately take over and a new back-up selected. This scheme offers 100-percent protection against single-site failures, but of course is susceptible to multi-site failures. Protection against multiple failures can be improved by using multiple back-ups, although Alsuberg et al. argue that one back-up is sufficient for most applications.

Menasce et al. propose a similar mechanism designed for multiple back-ups. The heart of their approach is a communication procedure for ensuring that all locks are received by all back-ups, and a procedure for reconstructing consistent lock tables following site failures.

**Non-availability of stored data items**

Suppose a transaction issues a Write command against a logical data item x, and some stored copy of x is unavailable. Since the DDBMS must update all stored copies of x, we have a problem. The DDBMS could delay the Write until all stored copies were simultaneously available, but this might never happen. Or it could abort the transaction, but then the availability of a data item would decrease as more copies of it were maintained. The solution is to buffer Write operations against non-available sites and to perform them when the failed site recovers. By buffering the Writes at multiple sites, increased protection against multiple failures can be achieved. This technique is sometimes called spooling.

**Transaction failures**

If a transaction fails before completion, a serious concurrency control problem is created—every Write performed by the transaction must be backed out to avoid leaving partial results in the database. The usual technique for doing this is called two-phase commit.
While a transaction executes, all Writes it performs are placed in temporary files and not the permanent data base. When the transaction completes, it issues Commit messages to each site holding temporary files, whereupon the temporary file is merged into the permanent data base. If the transaction fails before sending the first Commit, no updates are installed. If it fails after sending some but not all Commits, the sites holding temporary files can recognize the situation and can consult the other sites. If any site did receive a Commit, all sites will perform the update.

This technique achieves 100-percent protection against failures of the transaction alone, but is not fully robust with respect to multi-site failures. Hammer and Shipman describe mechanisms for improving the multi-site robustness of two phase commit.

CONCLUSION

We have presented several approaches to distributed concurrency control, and the obvious question is, "Which one is best?" We have no clear answer to this question, but a comparison of the methods may be helpful.

First, all methods presented here are correct—they all guarantee serializable executions. Second, the methods offer slightly different degrees of concurrency—conflict-driven restart and majority consensus offer slightly less concurrency than conventional locking; conflict graph analysis combined with locking offers slightly more concurrency than conventional locking; and conflict graph analysis coupled with SDD-1 timestamp protocols offers an "incomparable" degree of concurrency, meaning it allows some executions the other techniques prohibit, while prohibiting some executions the others allow. Termination issues are best understood in the context of locking, and locking is the only technique for which termination can be proved. Majority consensus is susceptible to cyclic restart, and conflict graph analysis coupled with SDD-1 protocols can lead to indefinite postponement; in practice, however, the probability of non-termination can be made acceptably small. With respect to robustness, all approaches share the same problems, and the same techniques. So the approaches compare almost identically on these four issues, at least.

The remaining area of comparison is performance. As explained in the section on Other Aspects of Concurrency Control, the key determinant of performance is communications behavior. Unfortunately, few quantitative performance results are available and we shall limit ourselves to basic observations.

Primary site locking requires inter-site communication whenever a lock is requested or released. In principle, primary copy locking could require the same amount of communication, but in a well configured system we would expect to do better. The reason is locality of reference—in many distributed applications, the majority of transactions access data local to the site at which they run. If that data is the primary copy, all lock requests can be processed locally. Of course, this advantage cannot be realized for data items that are heavily accessed from multiple sites.

The performance of the SDD-1 technique similarly depends on application-specific factors. If many transactions run in classes that do not require synchronization, the system will require few synchronization messages. Since the class definitions are tunable, classes can (in principle) be designed so that frequently-executed classes do not lie on cycles. However, if most transactions run in classes that do require synchronization, the communication overhead involved will be comparable to locking.

The performance of majority consensus is comparable to primary site locking with these differences—all locks are in effect requested in a single message; in return for this savings, though, multiple restarts may have to be endured.

The material we have presented on each approach is a bare outline. We have left out many important details and variations, and we urge the interested reader to consult source materials directly.

ACKNOWLEDGMENTS

We would like to thank Clarice Louis for editorial assistance in preparing this paper. We also thank S. Kimbleton and R. Munetz for prodding us to produce this tutorial survey.

BIBLIOGRAPHY

Note: References are listed by topic, and many do not appear in the text.

General

Concurrency control


Distributed locking


34. Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," Proc. 3rd Int. Workshop on Distributed Databases and Computer Networks, August 1978.


See also Reference 4.

Majority consensus


SDD-I


Other


Robustness

