ABSTRACT. Dozens of articles have been published describing "new" concurrency control algorithms for distributed database systems. All of these algorithms can be derived and understood using a few basic concepts. We show how to decompose the concurrency control problem into several subproblems, each of which has just a few known solutions. By appropriately combining known solutions to the subproblems, we show that all published concurrency control algorithms and many new ones can be constructed. The glue that binds the subproblems and solutions together is a mathematical theory known as serializability theory.

This paper does not assume previous knowledge of distributed database concurrency control algorithms, and is suitable for both the uninitiated and the cognoscente.

1. INTRODUCTION

A distributed database system (DDBS) is a database system (DBS) that provides commands to read and write data that is stored at multiple sites of a network. If users access a DDBS concurrently, they may interfere with each other by attempting to read and/or write the same data. Concurrency control is the activity of preventing such behavior.

Dozens of algorithms that solve the DDBS concurrency control problem have been published (see [BGl] and the references). Unfortunately, many of these algorithms are so complex that only an expert can understand them.

To remedy this situation, we have developed a simple framework for understanding concurrency control algorithms. The framework decomposes the problem into subproblems and gives basic techniques for solving each subproblem. To understand a published algorithm, one first identifies the technique used for each subproblem and then checks that the techniques are appropriately combined. The framework can also be used to develop new algorithms by combining existing techniques in new ways.

The paper has 10 sections. Sections 2 and 3 set the stage by describing a simple DDBS architecture and sketching the framework in terms of the architecture. The framework itself appears in Sections 4-8. Section 9 uses the framework to explain several published algorithms. Section 10 is the conclusion.

This paper refines an earlier survey of concurrency control algorithms [BGl]. The earlier paper includes many technical details that are omitted here. We urge the interested reader to consult [BGl] for more details.

2. DISTRIBUTED DBS ARCHITECTURE

We use a simple model of DDBS structure and behavior. The model highlights those aspects of a DDBS that are important for understanding concurrency control, while hiding details that don't affect concurrency control.

A database consists of a set of data items, denoted \{\ldots ,x,y,z\}. In practice, a data item can be a file, record, page, etc. But for the purposes of this paper, it's best to think of a data item as a simple variable. For now, assume each data item is stored at exactly one site.

Users access data items by issuing Read and Write operations. \texttt{Read} \texttt{(x)} returns the current value of \texttt{x}. \texttt{Write}\texttt{(x,new value)} updates the current value of \texttt{x} to \texttt{new-value}.

Users interact with the DDBS by executing programs called transactions. A transaction only interacts with the outside world by issuing Reads and Writes to the DDBS or by doing terminal I/O. We assume that every transaction is a complete and correct computation; each transaction, if

*This work was supported by the National Science Foundation, grant number MCS79-07762, by the Office for Naval Research, contract number N00014-80-C-647, and by Rome Air Development Center, Contract number F30602-81-C-0028.
executed alone on an initially consistent database, would terminate, produce correct results, and leave the database consistent.

Each site of a DDBS runs one or more of the following software modules (see Figures 1 and 2): a transaction manager (TM), a data manager (DM), or a scheduler. Transactions talk to TMs; TMs talk to schedulers; schedulers talk among themselves and also talk to DMs; and DMs manage the data.

---

**Figure 1. DDBS Architecture**

Transaction

```
Begin
Read(x)
Write(y)
End
```

Figure 2. Processing Operations

Each transaction issues all of its Reads and Writes to a single TM. A transaction also issues a Begin operation to its TM when it starts executing and an End when it's finished.

The TM forwards each Read and Write to a scheduler. (Which scheduler depends on the concurrency control algorithm; usually, the scheduler is at the same site as the data being read or written. In some algorithms, Begins and Ends are also sent to schedulers.)

The scheduler controls the order in which DM's process Reads and Writes. When a scheduler receives a Read or Write operation, it can either output the operation right away (usually to a DM, sometimes to another scheduler), delay the operation by holding it for later action, or reject the operation. A rejection causes the system to abort the transaction that issued the operation; every Write processed on behalf of the transaction is undone (restoring the old value of the data item), and every transaction that read a value written by the aborted transaction is also aborted. This phenomenon of one abort triggering other aborts is called cascading aborts. (It is usually avoided in commercial DBS's by not allowing a transaction to read another transaction's output until the DBS is certain that the latter transaction will not abort. In this paper, we will not try to prevent cascading aborts.) This paper does not discuss techniques for implementing abort. See [GMBL, HS, LS].

The DM executes each Read and Write it receives. For Read, the DM locks in its local database and returns the requested value. For Write, the DM modifies its local database and returns an acknowledgment. The DM sends the returned value or acknowledgment to the scheduler, which relays it back to the TM, which relays it back to the transaction.

DM's do not necessarily execute operations first-come-first-served. If a DM receives a Read(x) and a Write(x) at about the same time, the DM is free to execute these operations in either order. If the order matters (as it probably does in this case), it is the scheduler's responsibility to enforce the order. This is done by using a handshaking communication discipline between schedulers and DM's (see Figure 3): if the scheduler wants Read(x) to be executed before Write(x), it sends Read(x) to the DM, waits for the DM's response, and then sends Write(x). Thus the scheduler doesn't even send Write(x) to the DM until it knows Read(x) was executed. Of course, when the execution order doesn't matter, the scheduler can send operations without the handshake.

Handshaking is also used between other modules when execution order is important.

To execute Read(x) on behalf of transaction 1 followed by Write(x) on behalf of transaction 2

```
Scheduler
  send Read(x)
  receive Read(x)
  execute Read(x)
  send value

  receive value
  send Write(x)
  receive Write(x)
  execute Write(x)
  send ack
```

Figure 3. Handshaking
3. THE FRAMEWORK

The DDBS modules most important to concurrency control are schedulers. A concurrency control algorithm consists of some number of schedulers, running some type of scheduling algorithm, in a centralized or distributed fashion. In addition, the concurrency control algorithm must handle "replicated data" somehow. (TM's often handle this problem.)

To understand a concurrency control algorithm using our framework one determines

(i) the type of scheduling algorithm used (discussed in Sections 5 and 8),
(ii) the location of the scheduler(s), i.e. centralized vs. distributed (Section 6), and
(iii) how replicated data is handled (Section 7).

The framework also includes rules that tell when a concurrency control algorithm is correct. These rules give precise conditions under which a DDBS produces correct executions. These rules, called serializability theory, are discussed in the next section.

4. SERIALIZABILITY THEORY

Serializability theory is a collection of mathematical rules that tell whether a concurrency control algorithm works correctly [BSW,Casa,EGLT,Papa,PBR,SLR]. Serializability theory does its job by looking at the executions allowed by the concurrency control algorithm. The theory gives a precise condition under which an execution is correct. A concurrency control algorithm is then judged to be correct if all of its executions are correct.

4.1 Logs

Serializability theory models executions by a construct called a log. A log identifies the Read and Write operations executed on behalf of each transaction, and tells the order in which those operations were executed. Following Lamport, we allow an execution order to be a partial order [Lamp].

A transaction log represents an allowable execution of a single transaction. Formally, a transaction log is a partially ordered set (poset) $T_i = (X_i, \leq_i)$ where $X_i$ is the set of Reads and Writes issued by (an execution of) transaction $i$, and $\leq_i$ tells the order in which those operations must be executed. We write transaction logs as diagrams.

$$T_i = \begin{array}{l} \xymatrix{ r_1[x] \ar[r] & w_1[x] \ar[l] \} \end{array} \quad \begin{array}{l} \xymatrix{ r_1[z] \ar[r] & w_1[x] \ar[l] \} \end{array} \quad \begin{array}{l} \xymatrix{ r_1[x] \ar[r] & r_1[z] \ar[l] & w_1[x] \ar[l] \} \end{array} \quad \begin{array}{l} \xymatrix{ r_1[x] \ar[r] & r_1[z] \ar[l] & w_1[x] \ar[l] \} \end{array} \end{array}$$

$T_i$ represents a transaction that reads $x$ and $z$ in parallel, and then writes $x$. (Presumably, the value written depends on the values read.) We use $r_i(x)$ (resp., $w_i(x)$) to denote a read (resp., write) on $x$ issued by $T_i$. To keep this notation unambiguous, we assume that no transaction reads or writes a data item more than once.

Let $T = \{T_0, \ldots, T_n\}$ be a set of transaction logs. A DDBS log (or simply a log) over $T$ represents an execution of $T_0, \ldots, T_n$. Formally, a log over $T$ is a poset $L = (U, \prec)$ where

1. $U = \bigcup_{i=0}^n X_i$, and
2. $\prec = \bigcup_{i=0}^n \leq_i$.

Condition (1) states that the DDBS executed all, and only, the operations submitted by $T_0, \ldots, T_n$. Condition (2) states that the DDBS honored all operation orderings stipulated by the transactions.

The following are all possible logs over the example transaction log $T_1$ from above.

1. $\xymatrix{ r_1[x] \ar[r] & w_1[x] \ar[l] }$
2. $\xymatrix{ r_1[z] \ar[r] & w_1[x] \ar[l] }$
3. $\xymatrix{ r_1[x] \ar[r] & r_1[z] \ar[l] & w_1[x] \ar[l] }$
4. $\xymatrix{ r_1[x] \ar[r] & r_1[z] \ar[l] & w_1[x] \ar[l] }$

Notice that the DDBS is not required to process $\text{Read}(x)$ and $\text{Read}(z)$ in parallel, even though $T_1$ allows this parallelism. However, the DDBS is not allowed to reverse or eliminate any ordering stipulated by $T_1$. The following is not a log over $T_1$

because it reverses the order in which $T_1$ reads and writes $x$.

There is one further constraint on the form of logs. Two operations conflict if they operate on the same data item and (at least) one of them is a Write. To ensure that logs represent unique computations, we require that all pairs of conflicting operations be ordered. This constraint applies to transaction logs as well as DDBS logs.

Given transaction logs

$$\begin{array}{l} w_0[x] \end{array} \quad \begin{array}{l} w_0[y] \end{array} \quad \begin{array}{l} w_0[z] \end{array} \quad \begin{array}{l} r_1[x] \ar[r] & w_1[x] \ar[l] \end{array} \quad \begin{array}{l} r_1[z] \ar[r] & w_1[x] \ar[l] \end{array} \end{array}$$
T₂ = r₂[x] → w₂[y]  
T₃ = r₃[z] → w₃[y]

the following is a log over \{T₀, T₁, T₂, T₃\}.

\[
\begin{align*}
L₁ = w₀[y] &\rightarrow r₁[x] \rightarrow w₁[x] \\
&\rightarrow r₂[x] \rightarrow w₂[y] \\
&\rightarrow r₃[z] \rightarrow w₃[y] \\
&\rightarrow w₄[z] \\
\end{align*}
\]

(Note that orderings implied by transitivity are usually not drawn. E.g. w₀[y] < w₃[y] is not drawn in the diagram, although it follows from w₀[y] < w₂[y] and w₂[y] < w₃[y].)

4.2 Log Equivalence

Let L be a log over some set T, and suppose wᵢ[x] and rⱼ[x] are operations in L. We say rⱼ[x] reads-from wᵢ[x] if wᵢ[x] < rⱼ[x] and no wₖ[x] falls between rᵢ[x] and wⱼ[x]. In this log

w₀[x] → r₁[x] → w₂[y] → r₄[x] → w₃[z]

r₁[x] reads-from w₀[x], and r₃[x] and r₄[x] read-from w₂[y]. We call w₁[x] a final-write in L if no wᵢ[x] follows it. In this log

w₀[x] → w₁[x] → w₂[y] → r₂[x]

w₁[x] and w₂[y] are final-writes.

Intuitively, two logs over T are equivalent if they represent the same computation. Formally, two logs over T are equivalent if

1. each Read reads-from the same Write in both logs, and
2. they have the same final-writes.

Condition (1) ensures that each transaction reads the same values from the database in each log. Condition (2) ensures that the same transaction writes the final value of a given data item in both logs.

The following log L₂ is equivalent to log L₁ of Section 4.2.


(When we write a log as a sequence, e.g. L₂, we mean that the log is totally ordered: each operation precedes the next one and all subsequent ones in the sequence. Thus, in L₂, w₀[x] < w₁[y] < w₂[y] < r₂[x] . . . .)

4.3 Serializable Logs

A serial log is a total order on L such that for every pair of transactions Tᵢ and Tⱼ, either all of Tᵢ's operations precede all of Tⱼ's, or vice versa (e.g., L₂ in Section 4.2). A serial log represents an execution in which there is no concurrency whatsoever; each transaction executes from beginning to end without interference from the next transaction begins. From the point of view of concurrency control, therefore, every serial log represents an obviously correct execution.

What other logs represent correct executions? From the point of view of concurrency control, a correct execution is one in which concurrency is invisible. That is, an execution is correct if it is equivalent to an execution in which there is no concurrency. Serial logs represent the latter executions, and so a correct log is any log equivalent to a serial log. Such logs are termed serializable (SR). Log L₁ of Sec. 4.1 is SR, because it is equivalent to serial log L₂ of Sec. 4.2. Therefore L₁ is a correct log.

Serializability theory is the study of serializable logs.

4.4 The Serializability Theorem

This section presents the main theorem of serializability theory. Later sections rely on this theorem to analyze concurrency control algorithms. This theorem uses a graph derived from a log, called a serialization graph.

Suppose L is a log over {T₀,...,Tₙ}. The serialization graph for L, SG(L), is a directed graph whose nodes are T₀,...,Tₙ, and whose edges are all Tᵢ → Tⱼ such that, for some x, either (i) rᵢ[x] < wᵢ[x], or (ii) wᵢ[x] < rⱼ[x], or (iii) wᵢ[x] < wⱼ[x]. The serialization graphs for example log L₁ is

\[
SG(L₁) = T₀ \rightarrow T₃ \rightarrow T₂
\]

Edge T₀ → T₁ is present because w₀[x] < r₁[x]. Edge T₂ → T₁ is caused by r₂[x] < w₁[x]. Edge T₃ → T₂ arises from w₂[y] < w₃[y]. And so forth.

SERIALIZABILITY THEOREM. If SG(L) is acyclic then L is SR.

For example, since SG(L₁) is acyclic, L₁ is SR.
We can also use the Serializability Theorem to determine if a scheduler produces SR logs. First, we characterize the logs produced by the scheduler. Then we prove that every such log has an acyclic SG [BSW, Papa].

Some concurrency control algorithms schedule read-write conflicts separately from write-write conflicts. It is easier to analyze such algorithms using a restatement of the Serializability theorem. Define the read-write serialization graph for \( L \), \( SG_{rw}(L) \), as follows: \( SG_{rw}(L) \) has nodes \( T_0, \ldots, T_n \) and edges \( T_i + T_j \) such that, for some \( x \), either (i) \( r_i(x) < w_j(x) \), or (ii) \( w_i(x) < r_j(x) \). In other words, \( SG_{rw}(L) \) is like \( SG(L) \) except we don't care about write-write conflicts. The write-write serialization graph for \( L \), \( SG_{ww}(L) \), is defined analogously: the nodes are \( T_0, \ldots, T_n \) and the edges are \( T_i + T_j \) such that, for some \( x \), \( w_i(x) < w_j(x) \).

\[
SG_{rw}(L) = T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \\
SG_{ww}(L) = T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n
\]

Of course, \( SG(L) = SG_{rw}(L) \cup SG_{ww}(L) \).

**RESTATED SERIALIZABILITY THEOREM [BG1].** If the following four conditions hold, then \( L \) is SR

(i) \( SG_{rw}(L) \) is acyclic.

(ii) \( SG_{ww}(L) \) is acyclic.

(iii) For all \( T_i \) and \( T_j \), if \( T_i \) precedes \( T_j \) in \( SG_{rw}(L) \), then either \( T_i \) precedes \( T_j \) in \( SG_{ww}(L) \), or there is no path between them in \( SG_{ww}(L) \).

(iv) For all \( T_i \) and \( T_j \), if \( T_i \) precedes \( T_j \) in \( SG_{ww}(L) \), then either \( T_i \) precedes \( T_j \) in \( SG_{rw}(L) \), or there is no path between them in \( SG_{rw}(L) \).

Conditions (1)-(4) are just another way of saying that \( SG(L) \) is acyclic. The conditions allow us to analyze the correctness of read-write (rw) scheduling almost independently of write-write (ww) scheduling.

5. SCHEDULERS

There are four types of schedulers for producing SR executions: two-phase locking, timestamp ordering, serialization graph checking and certifiers. Each type of scheduler can be used to schedule rw conflicts, ww conflicts, or both. This section describes each type of scheduler assuming it is used for both kinds of conflict. Ways of combining scheduler types (e.g., two-phase locking for rw conflicts and timestamp ordering for ww conflicts) are described in section 9. This section also assumes that the scheduler runs at a single site, see Figure 4; Section 6 lifts this restriction.

**5.1 Two-Phase Locking.**

A two-phase locking (2PL) scheduler is defined by three rules [BGLT]:

i. Before outputting \( r_i(x) \) (resp. \( w_i(x) \)), set a read-lock (resp. write-lock) for \( T_i \) on \( x \). The lock must be held (at least) until the operation is executed by the appropriate DM. (Handshaking can be used to guarantee that locks are held long enough.)

ii. Different transactions cannot simultaneously hold "conflicting" locks. Two locks conflict if they are on the same data item and (at least) one is a write-lock. If rw and ww scheduling is done separately, the definition of "conflict" is modified. For rw scheduling, two locks on the same data item conflict if exactly one is a write-lock; i.e., write-locks don't conflict with each other. For ww scheduling, both locks must be write-locks.

iii. After releasing a lock, a transaction cannot obtain any more locks.

Rule (iii) causes locks to be obtained in a two-phase manner. During its growing phase, a transaction obtains locks without releasing any. By releasing a lock, the transaction enters its shrinking phase during which it can only release locks. Rule (iii) is usually implemented by holding all of a transaction's locks until it terminates.

**2PL THEOREM.** A 2PL scheduler only produces SR logs.

**Proof Sketch.** Consider a log \( L \) produced by a 2PL scheduler. If \( T_i + T_j \) is in \( SG(L) \), then \( T_i \) released some lock before \( T_j \) obtained that lock. If there's a nonempty path in \( SG(L) \) from \( T_i \) to \( T_j \) (i.e., a cycle) then, by...
Due to rule (ii), an operation received by a scheduler may be delayed because another transaction already owns a conflicting lock. Such blocking situations can lead to deadlock. For example, suppose $r_1[x]$ and $r_2[y]$ set read-locks, and then the scheduler receives $w_1[y]$ and $w_2[x]$. The scheduler cannot set the write-lock needed by $w_1[y]$ because $T_2$ holds a read-lock on $y$, nor can it set the write-lock for $w_2[x]$ because $T_1$ holds a read-lock on $x$. And, neither $T_1$ nor $T_2$ can release its read-lock before getting the needed write-lock because of rule (iii). Hence, we have a deadlock: $T_1$ is waiting for $T_2$ which is waiting for $T_1$.

Deadlocks can be characterized by a waits-for graph $\langle Halt, KC \rangle$, a directed graph whose nodes represent transactions and whose edges represent waiting relationships: edge $T_i \rightarrow T_j$ means $T_i$ is waiting for a lock owned by $T_j$. A deadlock exists if and only if (iff) the waits-for graph has a cycle. E.g., in the above example the waits-for graph is $T_1 \rightarrow T_2 \rightarrow T_1$.

A popular way of handling deadlock is to maintain the waits-for graph and periodically search it for cycles. (See [Chap. 4, AHU] for cycle detection algorithms.) When a deadlock is detected, one of the transactions on the cycle is aborted and restarted, thereby breaking the deadlock.

5.2 Timestamp Ordering

In timestamp ordering (T/O) each transaction is assigned a globally unique timestamp by its TM. (See [Mus, Thom] for how this is done.) The TM attaches the timestamp to all operations issued by the transaction. A T/O scheduler is defined by a single rule: Output all pairs of conflicting operations in timestamp order. Make sure conflicting operations are executed by DMs in the order they were output. (Handshaking can be used to make sure of this.) As for 2PL, the definition of "conflicting operation" is modified, if $rw$ and $ww$ scheduling are done separately.

**T/O Theorem.** A T/O scheduler only produces SR logs.

**Proof Sketch.** Since every pair of conflicting operations is in timestamp order, each edge $T_i \rightarrow T_j$ in SG has $TS(i) < TS(j)$ (where $TS(i)$ is the timestamp of $T_i$). Thus, SG cannot have any cycles. So, by the Serializability Theorem, the log produced by the scheduler is SR.

Several varieties of T/O schedulers have been proposed. We only sketch these variations here. Full details appear in [BG1].

A basic T/O scheduler outputs operations in essentially first-come-first-served order, as long as the T/O scheduling rule holds. When the scheduler receives $r_1[x]$ it does the following.

- If $TS(i) < \text{largetest timestamp of any Write on } x$ yet "accepted"
  - then reject $r_1[x]$,
- Else "accept" $r_1[x]$ and output it as soon as all Writes on $x$ with smaller timestamp have been acknowledged by the DM.

When the scheduler receives $w_i[x]$ it behaves as follows.

- If $TS(i) < \text{largest timestamp of any Read or Write on } x$ yet "accepted"
  - then reject $w_i[x]$,
- Else "accept" $w_i[x]$ and output it as soon as all Reads and Writes on $x$ with smaller timestamp have been acknowledged by the DM.

A conservative T/O scheduler avoids rejections by delaying operations instead. An operation is delayed until the scheduler is sure that outputting it will cause no future operations to be rejected. Conservative T/O requires that each scheduler receive Reads and Writes from each TM in timestamp order. To output any operation, the scheduler must have an operation from each TM in its "input queue." The scheduler then "accepts" the operation with smallest timestamp. "Accept" means remove the operation from the input queue, and output it as soon as all conflicting operations with smaller timestamp have been acknowledged by the DM. Variations on conservative T/O are discussed in [BG1,BSR].

Basic T/O and conservative T/O are endpoints of a spectrum. Basic T/O delays operations very little, but tends to reject many operations. Conservative T/O never rejects operations, but tends to delay them a lot. One can imagine T/O schedulers between these extremes. To our knowledge, no one has yet proposed such a scheduler.

Thomas’ write rule (TWR) is a technique that reduces delay and rejection [Thom]. TWR can only be used to schedule Writes, and needs to be combined with basic or conservative T/O to yield a complete scheduler. If we’re only interested in $ww$ scheduling, TWR is simple. When the scheduler receives $w_i[x]$ it does the following.
if \( TS(i) < \) largest timestamp of any Write on x yet "accepted"
then "pretend" to execute \( w_i[x] \) -- i.e., send an acknowledgement back to the TM, but don't send the Write to the DM
else "accept" \( w_i[x] \) and process it as usual.

The basic T/O TWR combination works like this. Reads are processed exactly as in basic T/O. But when the scheduler receives a \( w_i[x] \), it combines the basic T/O rule and TWR as follows.

if \( TS(i) < \) largest timestamp \( w_i[x] \) scheduling of any Read on x yet "accepted"
then reject \( w_i[x] \)
else if \( TS(i) < \) largest timestamp \( w_i[x] \) scheduling yet "accepted"
then "pretend" to execute \( w_i[x] \)
else "accept" the \( w_i[x] \) and output it as soon as all operations on x with smaller timestamp has been acknowledged by the DM.

The conservative T/O-TWR combination is described in [BG1].

5.3 Serialization Graph Checking

This type of scheduler works by explicitly building a serialization graph, SG, and checking it for cycles. Like basic T/O, an SG checking scheduler never delays an operation (except for handshaking reasons). Rejection is the only action used to avoid incorrect logs.

An SG checking scheduler is defined by the following rules.

i. when transaction \( T_i \) Begins, add node \( T_i \) to SG.

ii. When a Read or Write from \( T_i \) is received, add all edges \( T_j \rightarrow T_i \) such that \( T_j \) is a node of SG, and the scheduler has already output a conflicting operation from \( T_j \). As for the previous schedulers, the definition of "conflicting operation" is modified if rw and ww conflicts are scheduled separately.

iii. If after step (ii) SG is still acyclic, output the operation. Make sure that conflicting operations are executed by DM's in the order they were output. (Handshaking can be used for this.)

iv. If after (ii) SG has become cyclic, reject the operation. Delete node \( T_i \) and all edges \( T_j \rightarrow T_i \) or \( T_j \rightarrow T_i \) from SG. (SG is now acyclic again.)

SG CHECKER THEOREM. An SG checking scheduler only produces SR logs.

Proof Sketch. Every log produced by the scheduler has an acyclic SG. So, by the Serializability Theorem, every log is SR.

One technical problem with SG checkers is that a transaction must remain in SG even after it has terminated. A transaction can only be deleted from SG when it is a source node of the graph, i.e., when it has no incoming edges. See [Casa] for a discussion of this problem and techniques for efficiently encoding information about terminated transactions that remain in SG.

5.4 Certifiers

The term "certifier" refers to a scheduling philosophy, not a specific scheduling rule. A certifier is a scheduler that makes its decisions on a per-transaction basis. When a certifier receives an operation, it internally stores information about the operation and outputs it as soon as all earlier conflicting operations have been acknowledged. When a transaction ends, its TM sends the End operation to the certifier. At this point, the certifier checks its stored information to see if the transaction executed serializably. If it did, the certifier certifies the transaction, allowing it to terminate; otherwise, the certifier aborts the transaction.

All of the earlier schedulers can be adapted to work as certifiers. Here is an SG checking certifier. When the certifier receives an operation, it adds a node and some edges to SG as explained in the previous section. The certifier does not check for cycles at this time. When a transaction, \( T_i \), ends, the certifier checks SG for cycles. If \( T_i \) does not lie on a cycle, it is certified; otherwise it is aborted.

SG CERTIFIER THEOREM. An SG checking certifier only produces SR logs.

Proof Sketch. Consider any "completed" log produced by the certifier. Completed means that all uncertified transactions are aborted and removed from the log. (As always, any transaction that read data written by an aborted transaction is also aborted; this may include some certified transaction.) The completed log has an acyclic serialization graph. So by the Serializability Theorem, the log is SR.

Here is a 2PL certifier [Thom,KR]. Define a transaction to be active from the time the certifier receives its first operation until the certifier processes its End. The certifier stores two sets for each active transaction \( T_i \):

- \( T_i \)'s readset, \( RS(i) = \{x|\text{the certifier has output } r_i[x]\} \)
- \( T_i \)'s writset, \( WS(i) = \{x|\text{the certifier has output } w_i[x]\} \).

Proceedings of the Eighth International Conference on Very Large Data Bases

Mexico City, September, 1982
The certifier updates these sets as it receives operations. When the certifier receives End, it runs the following test.

Let \( RS(\text{active}) = \bigcup \{ RS(j), \text{such that } T_j \text{ is active, but } j \neq i \} \)

\( WS(\text{active}) = \bigcup \{ WS(j), \text{such that } T_j \text{ is active, but } j \neq i \} \)

\[
\text{if } \quad RS(i) \cap WS(\text{active}) = \emptyset, \text{ and } \quad WS(i) \cap (RS(\text{active}) \cup WS(\text{active})) = \emptyset \\
\text{then certify } T_i \\
\text{else abort } T_i
\]

This amounts to pretending that transactions hold imaginary locks on their readsets and write-sets. When transaction \( T_i \) ends, the certifier sees if \( T_i \)'s imaginary locks conflict with the imaginary locks held by other active transactions. If there is no conflict, \( T_i \) is certified; else \( T_i \) is aborted.

2PL CERTIFIER THEOREM. A 2PL certifier only produces SR logs.

Proof Sketch. Consider a completed log \( L \) produced by the certifier. If \( T_i \rightarrow T_j \) is in \( SG(L) \), then since both \( T_i \) and \( T_j \) were certified, the certifier processed End\_i before End\_j. If there's a nonempty path in \( SG(L) \) from \( T_i \) to \( T_j \) (i.e., a cycle) then, by transitivity, the certifier processed End\_i before End\_j. This is absurd. So, \( SG(L) \) is acyclic, and by the Serializability Theorem, \( L \) is SR.

T/O certifiers are also possible. To our knowledge, no one has proposed this algorithm yet.

Certifiers can also be built that check for serializable executions during transactions' executions, not just at the end. The extreme version of this idea is to check for serializability on every operation. At this extreme, the certifier reduces to a "normal" scheduler.

6. SCHEDULER LOCATION

The schedulers of Section 5 can be modified to work in a distributed manner. Instead of one scheduler for the whole system, we now assume one scheduler per DM (refer back to Figure 1). The scheduler normally runs at the same site as the DM, and schedules all operations that the DM executes.

The new issue in this setting is that the distributed schedulers must cooperate to attain the scheduling rules of Section 5.

The main problem caused by distributing schedulers is the maintenance of global data structures. Distributed 2PL schedulers need a global waits-for graph. Distributed T/O schedulers need a global SG. In distributed T/O scheduling, no global data structures are needed; each scheduler can make its scheduling decisions using local copies of R-TS\_x and W-TS\_x for each \( x \) at its DM. Distributed certifiers generally manifest the same problems as their corresponding schedulers.

6.1 Distributed Two-Phase Locking

Refer to the 2PL scheduling rules of Section 5.1. Rules (i) and (ii) are "local." The scheduler for data item \( x \) schedules all operations on \( x \). Hence this scheduler can set all locks on \( x \). Rule (iii) requires a small amount of inter-scheduler cooperation: no scheduler can obtain a lock for transaction \( T_i \) after any scheduler releases a lock for \( T_j \). This can be done by handshaking between TMs and schedulers. When \( T_i \) ends, its TM waits until all of \( T_i \)'s reads and writes are acknowledged. At this point the TM knows that all of \( T_i \)'s locks are set, and it's safe to release locks. The TM forwards End\_i to the schedulers which then release \( T_i \)'s locks.

One problem with distributed 2PL is that multi-site deadlocks are possible. Suppose \( x \) and \( y \) are stored at sites A and B, respectively. Suppose \( r_i[x] \) is processed at A, setting a read-lock on \( x \) for \( T_i \) at A; and \( r_j[y] \) is processed at B, setting a read-lock on \( y \) for \( T_j \) at B. If \( w_i[x] \) and \( w_i[y] \) are now issued, a deadlock will result; \( T_j \) will be waiting for \( T_i \) to release its lock on \( x \) at A and \( T_i \) will be waiting for \( T_j \) to release its lock on \( y \) at B. Unfortunately, the deadlock isn't apparent by looking at site A or B alone. Only by taking the union of the waits-for graphs at both sites does the deadlock cycle materialize.

See [MM,Ston,GilSh,Lomet 1-4,RSL] for solutions to this problem.

6.2 Distributed Timestamp Ordering

T/O schedulers are easy to distribute, because the T/O scheduling rule of Section 5.2 is inherently local. Consider a basic T/O scheduler for data item \( x \). To process an operation on \( x \), the scheduler only needs to know if a conflicting operation with larger timestamp has been accepted. Since the scheduler handles all operations on \( x \), it can make this decision itself.

6.3 Distributed Serialization Graph Checking

SG checkers are harder to distribute than the other scheduler because the serialization graph, SG, is inherently global. A transaction that accesses data at a single site can become involved in a cycle spanning many sites. See [Casa] for a discussion of this problem.
6.4 Distributed Certifiers

Distributed certifiers have a synchronization requirement a little like rule (iii) of 2PL: Ti’s TM must not send Endi to any certifier, until all of Ti’s Reads and Writes have been acknowledged. I.e., we must not try to certify Ti at any site until we are ready to certify Ti at all sites.

Beyond this, each distributed certifier behaves like the corresponding scheduler. A distributed 2PL certifier needs little interscheduler cooperation (beyond the previous paragraph). The certifier at each site keeps track of the data items at its site read or written by active transactions. When the certifier at site A receives Endi, it sees if any active transaction conflicts with Ti at site A. If not, Ti is certified at site A. If Ti is certified at all sites at which it accessed data, then it is "really" certified; else Ti is aborted.

A distributed SG certifier shares the problems of distributed SG schedulers: the certifier needs to check for cycles in a global graph every time a transaction ends.

6.5 Other Architectures

Centralized and distributed scheduling are endpoints of a spectrum. One can imagine hybrid architectures with multiple DM’s per scheduler. See Figure 5. This architecture adds no technical issues beyond those already discussed.

Hierarchical scheduler architectures are also possible. See Figure 6. To our knowledge, one has studied this approach yet.

7. DATA REPLICATION

In a replicated database, each logical data item, x, can have many physical copies, denoted \( \{x_1, \ldots, x_m\} \), which are resident at different DM’s. Transactions issue Reads and Writes on logical data items. TM’s translate those operations into Reads and Writes on physical data. The effect, as seen by each transaction, must be as if there were only one copy of each data item.

There is a simple way to obtain this effect. Each TM translates \( r_i[x] \) into \( r_i[x_j] \) for some copy \( x_j \) of \( x \) and \( w_i[x] \) into \( \{w_i[x_j] | \) all copies \( x_j \) of \( x \} \). If the scheduler(s) is SR, the effect is just like a nonreplicated database. To see this, consider a serial log equivalent to the SR log that executed. Since each transaction writes into all copies of each logical data item, each \( r_i[x_j] \) reads from the 'latest' transaction preceding it that wrote into any copy of \( x \). But this is exactly what would have happened had there been only one copy of \( x \). (For a more rigorous explanation, see [ABC].) Consider this example.

Each \( w_i[x] \) translates \( r_i[x] \) into \( r_i[x_j] \) for some copy \( x_j \) of \( x \) and \( w_i[x] \) into \( \{w_i[x_j] | \) all copies \( x_j \) of \( x \} \).

Figure 5. Hybrid Architecture

Figure 6. Hierarchical Architecture
We call this the do nothing approach to replication--just write into all copies of each data item and use an SR scheduler.

Two other approaches to replication have been suggested. In the primary copy approach, a copy of each x, say x_p, is designated its primary copy [Stan]. Each TM translates r_i[x] into r_i[x_p] for some copy x_p, as before.

 Writes are translated differently, though. The TM translates w_i[x] into a single Write, w_i[x_p], on the primary copy. When the primary copy's scheduler outputs w_i[x_p], it also issues Writes on the other copies of x (i.e., w_i[x_1],...,w_i[x_m]). See Figure 7. These Writes are processed by the schedulers for x_1,...,x_m in the usual way. For example, in 2PL, the scheduler for x_1 must get a write-lock on x_1 for T_1 before outputting w_i[x_1]. The primary copy's scheduler may be centralized (in which case the technique is called primary site [AD]), or distributed with the primary copy's DM.

In the voting approach to replication, TM's again distribute Writes to all copies of each data item [Thom]. Assume we are using distributed schedulers. When a scheduler is ready to output w_i[x_1], it sends a vote of yes to the vote collector for x; it does not output w_i[x_1] at this time. When the vote collector receives yes votes from a majority of schedulers, it tells all schedulers to output their Writes. (Each scheduler may need to update its local data structures before outputting w_i[x_1], e.g., set a write-lock on x_1.) Assume each scheduler is correct (i.e., produces an acyclic SG). Then, since every pair of conflicting operations was voted yes by some correct scheduler (both operations got a majority of yes's), the SG must be acyclic and the result is correct.

The principal benefit of voting is fault tolerance; it works correctly as long as a majority of sites holding a copy of x are running. See [Thom, Giff] for details.

5. MULTIVERSION DATA

Let us return to a database system model where each logical data item is stored at one TM.

In a multiversion database each Write, w_i[x], produces a new copy (or version) of x, denoted x_i. Thus, the value of x is a set of versions. For each Read, r_i[x], the scheduler selects one of the versions of x to be read.

Since Writes don't overwrite each other, and since Reads can read any version, the scheduler has more flexibility in controlling the effective order of Reads and Writes.

Although the database has multiple versions, users expect their transactions to behave as if there were just one copy of each data item. Serial logs don't behave this way. For example,

w_0[x][x_0][x_1][w_1[x_1]][r_2[x_2]][w_2[y_2]]

is a serial log, but its behavior cannot be reproduced with only one copy of x. We must therefore restrict the set of allowable serial logs.

A serial log is 1-copy serial (or l-serial) if each r_i[x] reads from the last transaction preceding it that wrote into any version of x. The above log is not l-serial, because r_2 reads x from w_0, but w_0[x_0]<w_1[x_1]<r_2[x_2]. A log is 1-serializable (l-SR) if it's equivalent to a l-serial log. 1-serializability is our correctness criterion for multiversion database systems.

All multiversion concurrency control algorithms (that we know of) totally order the versions of each data item in some simple way. A version order, <<, for L is an order relation over versions such that, for each x, << totally orders the versions of x.

Given a version order <<, we define the multiversion SG w.r.t. L and << (denoted MVSG(L,<<)) as SG(L) with the following edges

\[
\begin{align*}
&MVSG(L,<<) \\
&= SG(L) \\
&\text{where } E = \{ (x_i, x_j) : x_i << x_j \}.
\end{align*}
\]
added. For each \( ri[x^i] \) and \( wi[x^i] \) in \( L \), if \( x^k << x^j \) then include \( T_k \rightarrow T_j \); else include \( T_i \rightarrow T_k \).

**Multiversion Theorem [BG3].** A multiversion log is 1-SR iff there exists a version order << such that \( MVSG(L,<<) \) is acyclic.

This theorem enables us to prove multiversion concurrency control algorithms to be correct. We must argue that for every log \( L \) produced by the algorithm, \( MVSG(L,<<) \) is acyclic for some <<.

The types of multiversion schedulers that have been proposed fall into two classes that approximately correspond to timestamping and locking.

8.1 Multiversion Timestamping

Multiversion concurrency control was first introduced by Reed in his multiversion timestamping method [Reed]. In Reed's algorithm, each transaction has a unique timestamp. Each Read and Write carries the timestamp of the transaction that wrote it. The version order is defined by \( x^k << x^j \) if \( TS(i) < TS(j) \).

Operations are processed first-come-first-served. However, the version selection rules ensure that the overall effect is as if operations were processed in timestamp order. To process \( ri[x] \), the scheduler (or DM) returns the version of \( x \) with largest timestamp \( <TS(i) \). To process \( wi[x] \), version \( x^1 \) is created, unless some \( ri[x] \) has already been processed with \( TS(j) < TS(i) < TS(k) \). If this condition holds, the Write is rejected.

An analysis of \( MVSG(L,<<) \) for any \( L \) produced by this method shows that every edge \( T_i \rightarrow T_j \) is consistent with the order in which transactions were certified. Since certification is an atomic event, the certification order is a total order. Thus, \( MVSG(L,<<) \) is acyclic, and so \( L \) is 1-SR.

8.2 Multiversion Locking

In multiversion locking, the Writes on each data item, \( x \), must be ordered. We define \( x^1 << x^j \) if \( wi[x^1] < wi[x^j] \). Each version is in the certified or uncertified state. When a version is first written, it is uncertified. Each Read, \( r_i[x] \), reads either the last (wrt <<) certified version of \( x \) or any uncertified version of \( x \). When a transaction finishes executing, the database system attempts to certify it. To certify \( T_i \), three conditions must hold:

- **C1.** For each \( ri[x^i] \), \( x^i \) is certified.
- **C2.** For each \( wi[x^i] \), all \( x^j << x^i \) are certified.
- **C3.** For each \( wi[x^i] \) and each \( x^j << x^i \), all transactions that read \( x^j \) have been certified.

These conditions must be tested atomically. When they hold, \( T_i \) is declared to be certified and all versions it wrote are (atomically) certified.

An analysis of \( MVSG(L,<<) \) for any \( L \) produced by this method shows that every edge \( T_i \rightarrow T_j \) is in timestamp order. Thus, \( MVSG(L,<<) \) is acyclic, and so \( L \) is 1-SR.

Two details of the algorithm require some discussion. First, the algorithm can deadlock. For example, in this log:

\[ w_0[x^0]r_1[x^0]r_2[x^0]w_1[x^1]w_2[x^2] \]

\( T_1 \) and \( T_2 \) are deadlocked due to certification condition C3. As in 2PL, deadlocks can be detected by cycle detection on a waits-for graph whose edges include \( T_i \rightarrow T_j \) such that \( T_i \) is waiting for \( T_j \) to become certified (so that \( T_i \) will satisfy C1-C3).

Second, C1-C3 can be tested atomically without using a critical section. Once C1 or C2 is satisfied for some \( ri[x] \) or \( wi[x] \), no future event can falsify it. When C3 becomes true for some \( wi[x^1] \), we "lock" \( x^1 \) so that no future reads can read versions that precede \( x^1 \). This allows C1-C3 to be checked one data item at a time. Of course, the waits-for graph must be extended to account for these new version locks.

Two similar multiversion locking algorithms have been proposed which allow at most one uncertified version of each data item. In Stearns' and Rosenkrantz's method [SR], the waits-for graph is avoided by using a timestamp-based deadlock avoidance scheme. In Bayer et al.'s method [HRH, BEHR], a waits-for graph is used to help prevent deadlocks. This algorithms consults the waits-for graph before selecting a version to read, and always select a version that creates no cycles.

Multiversion locking algorithms in which queries (read-only transactions) are given special treatment are described in [Dubol], [BG4].
9. COMBINING THE TECHNIQUES

The techniques described in Sections 4–8 can be combined in almost all possible ways. The three basic scheduling techniques (2PL, T/O, SG checking) can be used in scheduler mode or certifier mode. This gives six basic concurrency control techniques. Each technique can be used for rw or ww scheduling or both \((6^2 = 36)\). Schedulers can be centralized or distributed \((36 \times 2 = 72)\), and replicated data can be handled in three ways (Do Nothing, Primary Copy, Voting) \((72 \times 3 = 216)\). Then, one can use multiversions or not \((216 \times 2 = 432)\). By considering the multifacous variations of each technique, the number of distinct algorithms is in the thousands.

To illustrate our framework, we describe some of these algorithms that have already appeared in the literature.

The distributed locking algorithm proposed for System R* uses a 2PL scheduler for rw and ww synchronization. The schedulers are distributed at the DM's. Replication is handled by the do nothing approach.

Distributed INGRES uses a similar locking algorithm \([Ston]\). The main difference is that distributed INGRES uses primary copy for replication.

Many researchers have proposed algorithms that use conservative T/O for all scheduling \([SM,LeTa,KWTh,CB]\). They typically distribute the schedulers at DM's and take the do nothing approach to replication.

SDD-1 uses conservative T/O for rw scheduling and Thomas' write rule for ww scheduling. The algorithm has distributed schedulers and takes the do nothing approach to replication. SDD-1 also uses conflict graph analysis, a technique for preanalyzing transactions to determine which run-time conflicts need not be synchronized.

A method using 2PL for rw scheduling and Thomas' write rule for ww scheduling is described in \([BGL]\). Distributed schedulers and the do nothing approach to replication were suggested. To ensure that the locking order is consistent with the timestamp order, one can use a Lamport clock: each message is timestamped with the local clock time when it was sent; if a site receives a message with a timestamp, TS, greater than its local clock time, the site pushes its clock ahead to TS. After a transaction obtains all of its locks, it is assigned a timestamp using the TM's local Lamport clock.

Thomas' majority consensus algorithm was one of the first distributed concurrency control algorithms. It uses a 2PL certifier for rw scheduling and Thomas' write rule for ww scheduling. Schedulers are distributed and voting is used for replication. Each transaction is assigned a timestamp from a Lamport clock when it is certified. This ensures that the certification order (produced by rw scheduling) is consistent with the timestamp order used for ww scheduling.

Each of these algorithms is quite complex. A complete treatment of each would be lengthy. Yet by understanding the basic techniques and how they can be correctly combined, we can explain the essentials of each algorithm in a few sentences.

10. PERFORMANCE

Given that thousands of concurrency control algorithms are conceivable, which one is best for each type of application? Every concurrency control algorithm delays and/or aborts some transactions, when conflicting operations are submitted concurrently. The question is: which algorithms increase overall transaction response time the least?

Although there have been several performance studies of some of these algorithms, the results are still inconclusive \([GS,GM1,GM2,Lee,Lin,LN,MN1,MN2,Ries1,Ries2]\). There is some evidence that 2PL schedulers perform well at low to moderate intensity of conflicting operations. However, we know of no quantitative results that tell when 2PL thrashes due to too many deadlocks. There are similar gaps in our understanding of the performance of other types of schedulers. More analysis is badly needed to help us learn how to predict which concurrency control algorithms will perform well for the applications and systems we will encounter in practice.

ACKNOWLEDGMENT. We would like to thank Renate D'Arcangelo for her expert editorial work in the preparation of the manuscript.

REFERENCES


Lin, W.K. "Concurrency Control in a Multiple Copy Distributed Data Base System," Proc. 4th Berkeley Conf. on Distributed Data Management and Computer Networks, August 1979.


