Implementing Isolation

Chapter 20

The Issue

• Maintaining database correctness when many transactions are accessing the database concurrently
  – Assuming each transaction maintains database correctness when executed in isolation

Isolation

• Serial execution:
  – Since each transaction is consistent and isolated from all others, schedule is guaranteed to be correct for all applications
  – Inadequate performance
    • Since system has multiple asynchronous resources and transaction uses only one at a time
• Concurrent execution:
  – Improved performance (multiprogramming)
  – Some interleavings produce correct result, others do not
  – We are interested in concurrent schedules that are equivalent to serial schedules. These are referred to as serializable schedules.

Transaction Schedule

T1: begin_transaction();

To db

server

Schedule

Schedule in which requests are serviced (to preserve isolation)

Database server

Transaction schedule (commit applies to this)

p1,1

p1,2

p1,3

p2,1

p2,2

p2,3

p2,4

local variables

T1: T2: T3:

T1: T2:

T1: T2:

p1,1 p1,2 p1,3 p1,4

p1,1 p1,2 p1,3 p1,4

p1,1 p1,2 p1,3 p1,4

p1,1 p1,2 p1,3 p1,4

time →

time →

time →
Concurreny Control

• Transforms arriving interleaved schedule into a correct interleaved schedule to be submitted to the DBMS
  – Delays servicing a request (reordering) - causes a transaction to wait
  – Refuses to service a request - causes transaction to abort
• Actions taken by concurrency control have performance costs
  – Goal is to avoid delaying or refusing to service a request

Correct Schedules

• Interleaved schedules equivalent to serial schedules are the only ones guaranteed to be correct for all applications
• Equivalence based on commutativity of operations
• Definition: Database operations \( p_1 \) and \( p_2 \) commute if, for all initial database states, they
  1. return the same results and
  2. leave the database in the same final state when executed in either order.

Conventional Operations

• Read
  – \( r(x, X) \) - copy the value of database variable \( x \) to local variable \( X \)
• Write
  – \( w(x, X) \) - copy the value of local variable \( X \) to database variable \( x \)
• We use \( r_1(x) \) and \( w_1(x) \) to mean a read or write of \( x \) by transaction \( T_1 \)

Commutativity of Read and Write Operations

• \( p_i \) commutes with \( p_j \) if
  – They operate on different data items
  • \( w_i(x) \) commutes with \( w_j(x) \) and \( r_j(y) \)
  – Both are reads
  • \( r_i(x) \) commutes with \( r_j(x) \)
• Operations that do not commute conflict
  • \( w_i(x) \) conflicts with \( w_j(x) \)
  • \( w_i(x) \) conflicts with \( r_j(x) \)

Equivalence of Schedules

• An interchange of adjacent operations of different transactions in a schedule creates an equivalent schedule if the operations commute
  \[ S_1: S_{1,1} p_i p_j S_{1,2} \quad \text{where} \quad i \neq k \]
  \[ S_2: S_{2,1} p_k p_j S_{2,2} \]
  – Each transaction computes the same results (since operations return the same values in both schedules) and hence writes the same values to the database.
  – The database is left in the same final state (since the state seen by \( S_{1,2} \) is the same in both schedules).

Equivalence of Schedules

• Equivalence is transitive: If \( S_1 \) can be derived from \( S_2 \) by a series of such interchanges, \( S_1 \) is equivalent to \( S_2 \)
Example of Equivalence

\[ S_1: r_1(x) \Rightarrow r_2(x) \Rightarrow w_2(x) \Rightarrow r_1(x) \Rightarrow w_1(y) \]
\[ S_2: r_1(x) \Rightarrow r_1(y) \Rightarrow r_2(x) \Rightarrow w_2(x) \Rightarrow w_1(y) \]
\[ S_3: r_1(x) \Rightarrow r_2(x) \Rightarrow r_1(y) \Rightarrow w_2(x) \]
\[ S_4: r_1(x) \Rightarrow r_2(x) \Rightarrow r_1(y) \Rightarrow w_1(y) \]
\[ S_5: r_1(x) \Rightarrow w_1(y) \Rightarrow r_2(x) \Rightarrow w_2(x) \]

\[ S_1 \] is equivalent to \( S_5 \)
\[ S_5 \] is the serial schedule \( T_1, T_2 \)
\[ S_1 \] is serializable
\[ S_1 \] is not equivalent to the serial schedule \( T_2, T_3 \)

Example of Equivalence

\[ T_2: \begin{array}{l}
\text{begin transaction} \\
\text{read} (x, X) ; \\
X = X+4 ; \\
\text{write} (x, X) ; \\
\text{commit} ;
\end{array} \]
\[ T_1: \begin{array}{l}
\text{begin transaction} \\
\text{read} (x, Y) ; \\
\text{write} (y, Y) ; \\
\text{commit} ;
\end{array} \]

Initial state
\[ x=1, y=3 \]
Final state
\[ x=5, y=1 \]

Interchange commuting operations
\[ x=1, y=3 \]
\[ r_1(x), r_1(y) \Rightarrow r_2(x), r_2(y) \Rightarrow w_2(x), w_1(y) \]

Interchange conflicting operations
\[ x=1, y=3 \]
\[ r_1(x) \Rightarrow r_2(x), r_1(y) \Rightarrow w_2(x), w_1(y) \]

Serializable Schedules

- \( S \) is serializable if it is equivalent to a serial schedule
- Transactions are totally isolated in a serializable schedule
- A schedule is correct for \( \text{any} \) application if it is a serializable schedule of consistent transactions
- The schedule:
  \[ r_1(x), r_1(y) \Rightarrow w_2(x), w_1(y) \]
  is not serializable

Isolation Levels

- Serializability provides a \textit{conservative} definition of correctness
  - For a particular application there might be many acceptable \textit{non}-serializable schedules
  - Requiring serializability might degrade performance
- DBMSs offer a variety of isolation levels:
  - \texttt{SERIALIZABLE} is the most stringent
  - Lower levels of isolation give better performance
    - \texttt{MIGHT} allow incorrect schedules
    - \texttt{MIGHT} be adequate for some applications

Serializable

- \textbf{Theorem} - Schedule \( S_1 \) can be derived from \( S_2 \) by a sequence of commutative interchanges if and only if conflicting operations in \( S_1 \) and \( S_2 \) are ordered in the same way
  \textit{Only If:} Commutative interchanges do not reorder conflicting operations
  \textit{If:} A sequence of commutative interchanges can be determined that takes \( S_1 \) to \( S_2 \) since conflicting operations do not have to be reordered (see text)

Conflict Equivalence

- \textbf{Definition} - Two schedules, \( S_1 \) and \( S_2 \), of the same set of operations are \textit{conflict equivalent} if conflicting operations are ordered in the same way in both
  - Or (using theorem) if one can be obtained from the other by a series of commutative interchanges
Conflict Equivalence

• **Result**: A schedule is serializable if it is conflict equivalent to a serial schedule

\[ r_1(x) \ w_2(x) \ w_1(y) \ r_2(y) \equiv r_1(x) \ w_2(x) \ w_1(y) \ r_2(y) \]

• If in S transactions T₁ and T₂ have several pairs of conflicting operations (p₁,₁ conflicts with p₂,₁ and p₁,₂ conflicts with p₂,₂) then p₁,₁ must precede p₂,₁ and p₁,₂ must precede p₂,₂ (or vice versa) in order for S to be serializable.

View Equivalence

• Two schedules of the same set of operations are **view equivalent** if:
  – Corresponding read operations in each return the same values (hence computations are the same)
  – Both schedules yield the same final database state

• Conflict equivalence implies view equivalence.

• View equivalence **does not** imply conflict equivalence.

View Equivalence

\[ T₁: \ w(y) \ w(x) \]
\[ T₂: \ r(y) \ w(x) \]
\[ T₃: \ w(x) \]

• Schedule is not conflict equivalent to a serial schedule

• Schedule has same effect as serial schedule T₁, T₂, T₃. It is view equivalent to a serial schedule and hence serializable

Conflict vs View Equivalence

• A concurrency control based on view equivalence should provide better performance than one based on conflict equivalence since less reordering is done but …

• It is difficult to implement a view equivalence concurrency control

Conflict Equivalence and Serializability

• Serializability is a conservative notion of correctness and conflict equivalence provides a conservative technique for determining serializability

• However, a concurrency control that guarantees conflict equivalence to serial schedules ensures correctness and is easily implemented

Serialization Graph of a Schedule, S

• Nodes represent transactions

• There is a directed edge from node Tᵢ to node Tⱼ if Tᵢ has an operation pᵢ,k that conflicts with an operation pⱼ,l of Tⱼ and pᵢ,k precedes pⱼ,l in S

• **Theorem**: A schedule is conflict serializable if and only if its serialization graph has no cycles
Intuition: Serializability and Nonserializability

- Consider the nonserializable schedule
  \[ r(x) \ w(x) \ r(y) \ w(y) \]
- Two ways to think about it:
  - Because of the conflicts, the operations of \( T_1 \) and \( T_2 \) cannot be interchanged to make an equivalent serial schedule
  - Because \( T_1 \) read \( x \) before \( T_2 \) wrote it, \( T_1 \) must precede \( T_2 \) in any ordering, and because \( T_1 \) wrote \( y \) after \( T_2 \) read \( y \), \( T_1 \) must follow \( T_2 \) in any ordering — clearly an impossibility

Recoverability: Schedules with Aborted Transactions

- \( T_2 \) has aborted but has had an indirect effect on the database — schedule is unrecoverable
- **Problem:** \( T_1 \) read uncommitted data - **dirty read**
- **Solution:** A concurrency control is recoverable if it does not allow \( T_1 \) to commit until all other transactions that wrote values \( T_1 \) read have committed

\[
\begin{align*}
T_1 &: \ r(x) \ w(y) \ &\text{commit} \\
T_2 &: \ w(x) \ &\text{commit} \\
&\text{abort}
\end{align*}
\]

- Better solution: prohibit dirty reads

Dirty Write

- **Dirty write:** A transaction writes a data item written by an active transaction
- Dirty write complicates rollback:

\[
\begin{align*}
T_1 &: \ w(x) \ &\text{commit} \\
T_2 &: \ w(x) \ &\text{commit} \\
&\text{abort}
\end{align*}
\]

Strict Schedules

- **Strict schedule:** Dirty writes and dirty reads are prohibited
- Strict and serializable are two different properties
  - Strict, non-serializable schedule:
    \[ r(x) \ w(x) \ r(y) \ w(y) \ c_1 \ c_2 \]
  - Serializable, non-strict schedule:
    \[ w(x) \ r(x) \ w(y) \ r(y) \ c_1 \ c_2 \]
Concurrency Control

• Concurrency control cannot see entire schedule:
  – It sees one request at a time and must decide whether to allow it to be serviced
• Strategy: Do not service a request if:
  – It violates strictness or serializability, or
  – There is a possibility that a subsequent arrival might cause a violation of serializability

Models of Concurrency Controls

• Immediate Update – (the model we have discussed)
  – A write updates a database item
  – A read copies value from a database item
  – Commit makes updates durable
  – Abort undoes updates
• Deferred Update – (we will discuss this later)
  – A write stores new value in the transaction’s intentions list (does not update the database)
  – A read copies value from the database or the transaction’s intentions list
  – Commit uses intentions list to durably update database
  – Abort discards intentions list

Immediate vs. Deferred Update

Immediate Update

Immediate Update

Deferred Update

Models of Concurrency Controls

• Pessimistic –
  – A transaction requests permission for each database (read/write) operation
  – Concurrency control can:
    • Grant the operation (submit it for execution)
    • Delay it until a subsequent event occurs (commit or abort of another transaction), or
    • Abort the transaction
  – Decisions are made conservatively so that a commit request can always be granted
    • Takes precautions even if conflicts do not occur

Models of Concurrency Controls

• Optimistic -
  – Request for database operations (read/write) are always granted
  – Request to commit might be denied
    • Transaction is aborted if it performed a non-serializable operation
  – Assumes that conflicts are not likely

Immediate-Update Pessimistic Control

• The most commonly used control
• Consider first a simple case
  – Suppose such a control allowed a transaction, T₁, to perform some operation and then, while T₁ was still active, it allowed another transaction, T₂, to perform a conflicting operation
  – The schedule would not be strict and so this situation cannot be allowed
    • But consider a bit further what might happen …
Immediate-Update Pessimistic Control

- If $T_1$ executes $op_1(x)$ and then $T_2$ executes a conflicting operation, $op_2(x)$, $T_2$ must follow $T_1$ in any equivalent serial schedule.
- **Problem:** If $T_1$ and $T_2$ later make conflicting accesses to $y$, control cannot allow ordering $op'_2(y)$, $op'_1(y)$
  - control has to use transitive closure of transaction ordering to prevent loop in serialization graph (too complicated)
- **Worse problem:**
  \[
  w_f(x) \ x \ y \text{ commit} \ x \ y \text{ request} \ x \ y
  \]
  looks good \[
  \xrightarrow{\text{disaster}}
  \]

Immediate-Update Pessimistic Control

- **Result:** Each schedule, $S$, is equivalent to a serial schedule in which transactions are ordered in the order in which they commit in $S$ (and possibly other serial schedules as well)
  - **Reason:** When a transaction commits, none of its operations conflict with those of other active transactions. Therefore it can be ordered before all active transactions.
  - **Example:** The following (non-serializable) schedule is not permitted because $T_1$ was active at the time $w_f(x)$ (which conflicts with $r_f(x)$) was requested

Immediate-Update Pessimistic Control

- **Rule:**
  - Do not grant a request that imposes an ordering among active transactions (delay the requesting transaction)
  - Grant a request that does not conflict with previously granted requests of active transactions
- Rule can be used as each request arrives
- If a transaction’s request is delayed, it is forced to wait (but the transaction is still considered active)
  - Delayed requests are reconsidered when a transaction completes (aborts or commits) since it becomes inactive

Immediate-Update Pessimistic Control

- **Commit order is useful since transactions might perform external actions visible to users**
  - After a deposit transaction commits, you expect a subsequent transaction to see the new account balance

Immediate-Update Pessimistic Control

- **Problem:** Controls that cause transactions to wait can cause deadlocks
  \[
  w_f(x) \ y \ x \ y \text{ request} \ x \ y
  \]
- **Solution:** Abort one transaction in the cycle
  - Use wait-for graph to detect cycle when a request is delayed or
  - Assume a deadlock when a transaction waits longer than some time-out period

Deadlock

- **Problem:** Controls that cause transactions to wait can cause deadlocks
  \[
  w_f(x) \ y \ x \ y \text{ request} \ x \ y
  \]
- **Solution:** Abort one transaction in the cycle
  - Use wait-for graph to detect cycle when a request is delayed or
  - Assume a deadlock when a transaction waits longer than some time-out period
## Locking Implementation of an Immediate-Update Pessimistic Control

- A transaction can read a database item if it holds a read (shared) lock on the item.
- It can read or update the item if it holds a write (exclusive) lock.
- If the transaction does not already hold the required lock, a lock request is automatically made as part of the (read or write) request.

## Locking

- Request for read lock on an item is granted if no transaction currently holds write lock on the item.
  - Cannot read an item written by an active transaction.
- Request for write lock granted if no transaction holds any lock on item.
  - Cannot write an item read/written by an active transaction.
- Transaction is delayed if request cannot be granted.

<table>
<thead>
<tr>
<th>Requested mode</th>
<th>Granted mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read, write</td>
</tr>
<tr>
<td>write</td>
<td>x, x</td>
</tr>
</tbody>
</table>

## Locking

- **Result:** A lock is not granted if the requested access conflicts with a prior access of an active transaction; instead the transaction waits. This enforces the rule:
  - Do not grant a request that imposes an ordering among active transactions (delay the requesting transaction).
- Resulting schedules are serializable and strict.

## Locking Implementation

- Associate a lock set, $L(x)$, and a wait set, $W(x)$, with each active database item, $x$.
  - $L(x)$ contains an entry for each granted lock on $x$.
  - $W(x)$ contains an entry for each pending request on $x$.
- When an entry is removed from $L(x)$ (due to transaction termination), promote (non-conflicting) entries from $W(x)$ using some scheduling policy (e.g., FCFS).
- Associate a lock list, $\mathcal{L}$, with each transaction, $T_i$.
  - $\mathcal{L}$ links $T_i$’s elements in all lock and wait sets.
  - Used to release locks on termination.
**Locking Implementation**

- **T** holds an *r* lock on *x* and waits for a *w* lock on *y*

**Manual Locking**

- Better performance possible if transactions are allowed to release locks before commit
  - Ex: release lock on item when finished accessing the item
  - \[ T_1: l(x) r(x) l(y) r(y) u(x) w(y) u(y) \]
  - \[ T_2: l(x) l(z) w(x) w(z) u(x) u(z) \]
  - However, early lock release can lead to non-serializable schedules

**Two-Phase Locking**

- Transaction does not release a lock until it has all the locks it will ever require.
- Transaction has a locking phase followed by an unlocking phase

**Two-Phase Locking**

- A schedule produced by a two-phase locking control is:
  - Equivalent to a serial schedule in which transactions are ordered by the time of their first unlock operation
  - Not necessarily recoverable (dirty reads and writes are possible)

- **Theorem**: A concurrency control that uses two phase locking produces only serializable schedules.
  - **Proof (sketch)**: Consider two transactions \( T_1 \) and \( T_2 \) in schedule \( S \) produced by a two-phase locking control and assume \( T_1 \)'s first unlock, \( t_1 \), precedes \( T_2 \)'s first unlock, \( t_2 \).
    - If they do not access common data items, then all operations commute.
    - Suppose they do. All of \( T_1 \)'s accesses to common items precede all of \( T_2 \)'s. If this were not so, \( T_2 \)'s first unlock must precede a lock request of \( T_1 \). Since both transactions are two-phase, this implies that \( T_2 \)'s first unlock precedes \( T_1 \)'s first unlock, contradicting the assumption. Hence, all conflicts between \( T_1 \) and \( T_2 \) are in the same direction.
    - It follows that the serialization graph is cycle-free since if there is a cycle \( T_1, T_2, \ldots, T_n \) then it must be the case that \( t_1 < t_2 < \ldots < t_n < t_1 \).

**Two-Phase Locking**

- A two-phase locking control that holds write locks until commit produces strict, serializable schedules
  - This is automatic locking
  - Equivalent to a serial schedule in which transactions are ordered by their commit time
  - “Strict” is used in two different ways: a control that releases read locks early guarantees strictness, but is not strict two-phase locking control
Lock Granularity

- Data item: variable, record, row, table, file
- When an item is accessed, the DBMS locks an entity that contains the item. The size of that entity determines the granularity of the lock
  - Coarse granularity (large entities locked)
    - Advantage: If transactions tend to access multiple items in the same entity, fewer lock requests need to be processed and less lock storage space required
    - Disadvantage: Concurrency is reduced since some items are unnecessarily locked
  - Fine granularity (small entities locked)
    - Advantages and disadvantages are reversed

Objects and Semantic Commutativity

- Read/write operations have little associated semantics and hence little associated commutativity.
  - Among operations on the same item, only reads commute.
- Abstract operations (for example operations on objects) have more semantics, allowing
  - More commutativity to be recognized
  - More concurrency to be achieved

Abstract Operations and Commutativity

- A concurrency control that deals with operations at an abstract level can recognize more commutativity and achieve more concurrency
- Example: operations deposit(acct,n), withdraw(acct,n) on an account object (where n is the dollar amount)

<table>
<thead>
<tr>
<th>Requested Mode</th>
<th>Granted Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit()</td>
<td>X</td>
</tr>
<tr>
<td>withdraw()</td>
<td>X</td>
</tr>
</tbody>
</table>

A Concurrency Control Based on Abstract Operations

- Concurrency control grants deposit and withdraw locks based on this table
- If one transaction has a deposit lock on an account object, another transaction can also obtain a deposit lock on the object
- Would not be possible if control viewed deposit as a read followed by a write and attempted to get read and write locks

A Concurrency Control Based on Abstract Operations

- Since T₁ and T₂ can both hold a deposit lock on the same account object their deposit operations do not delay each other
  - As a result, the schedule can contain:
    … deposit₁(acct,n) … deposit₂(acct,m) … commit₁
    or
    … deposit₁(acct,m) … deposit₂(acct,n) … commit₂
  - But the two deposit operations must be isolated from each other. Assuming bal is the account balance, the schedule
    r₁(bal) r₂(bal) w₁(bal) w₂(bal)
    cannot be allowed
### Partial vs. Total Operations

- **deposit()**, **withdraw()** are total operations: they are defined in all database states.
- **withdraw()** has two possible outcomes: **OK, NO**
- **Partial operations** are operations that are not defined in all database states.
- **withdraw()** can be decomposed into two partial operations, which cover all database states:
  - **withdrawOK()**
  - **withdrawNO()**

### Partial Operations

- Example: account object
  - **deposit()**: defined in all initial states (total)
  - **withdrawOK(acct,x)**: defined in all states in which `bal ≥ x` (partial)
  - **withdrawNO(acct,x)**: defined in all states in which `bal < x` (partial)

When a transaction submits **withdraw()**, control checks balance and converts to either **withdrawOK()** or **withdrawNO()** and acquires appropriate lock.

### Partial Operations

- Partial operations allow even more semantics to be introduced
- Insight: while **deposit()** does not commute with **withdraw()**, it does (backward) commute with **withdrawOK()**

\[
\text{withdrawOK}(a,n) \text{ deposit}(a,m) \rightarrow \text{deposit}(a,m) \text{ withdrawOK}(a,n)
\]

### Backward Commutativity

- **p** backward commutes through **q** if in all states in which the sequence **q, p** is defined, the sequence **p, q** is defined and
  - **p** and **q** return the same information in both and
  - The database is left in the same final state
- Example:
  - **deposit(a,m) backward commutes through withdrawOK(a,n)**
    - In all database states in which **withdrawOK(a,n)**, **deposit(a,m)** is defined, **deposit(a,m)**, **withdrawOK(a,n)** is also defined.
  - **withdrawOK(a,n)** does not backward commute through **deposit(a,m)**
  - **backward commute** is not symmetric

### A Concurrency Control Based on Partial Abstract Operations

<table>
<thead>
<tr>
<th>Requested Mode</th>
<th>Granted Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit()</td>
<td>deposit()</td>
</tr>
<tr>
<td>withdrawOK()</td>
<td>withdrawOK()</td>
</tr>
<tr>
<td>withdrawNO()</td>
<td>withdrawNO()</td>
</tr>
</tbody>
</table>

- Control grants **deposit, withdrawOK, and withdrawNO locks**
  - Conflict relation is
    - not symmetric
    - based on backward commutativity

### A Concurrency Control Based on Partial Abstract Operations

- **Advantage**: Increased concurrency and hence increased transaction throughput
- **Disadvantage**: Concurrency control has to access the database to determine the return value (hence the operation requested) before consulting table
- Hence (with an immediate update system) if T writes x and later aborts, physical restoration can be used.
Atomicity and Abstract Operations

• A write operation (the only conventional operation that modifies items) conflicts with all other operations on the same data

• Physical restoration (restore original value) does not work with abstract operations since two operations that modify a data item might commute
  – How do you handle the schedule: \( \ldots p_{f}(x) \ q_{f}(x) \ abort_{1} \ldots \) if both operations modify \( x \)?

• Logical restoration (with compensating operations) must be used
  – e.g., \( \text{increment}(x) \) compensates for \( \text{decrement}(x) \)

Requirements for an Operation to Have a Compensating Operation

• For an operation to have a compensating operation, it must be one-to-one
  – For each input there is a unique output
  – The parameters of the compensating operation are the same as the parameters of the operation being compensated

• \( \text{increment}(x) \) compensate \( \text{decrement}(x) \)

Logical Restoration (Compensation)

• Example:
  \[
  p_{f}(x) = \text{decrement}(x) \\
  p_{f}^{-1}(x) = \text{increment}(x) \\
  \text{compensating operation}
  \]

  \[
  \text{decrement}(x) \ \text{increment}(x) \ \text{increment}(x) = \text{increment}(x)
  \]

A Closer Look at Compensation

• We have discussed compensation before, but now we want to use it in combination with locking to guarantee serializability and atomicity

• We must define compensation more carefully

Logical Restoration (Compensation)

• Consider schedule: \( p_{f}(x) \ q_{f}(x) \ abort_{1} \)

• \( q_{f}(x) \) must (backward) commute through \( p_{f}(x) \), since the concurrency control scheduled the operation

• This is equivalent to \( q_{f}(x) \ p_{f}(x) \ abort_{1} \)

• Then \( abort_{1} \) can be implemented with a compensating operation: \( q_{f}(x) \ p_{f}(x) \ p_{f}^{-1}(x) \)

  – This is equivalent to \( q_{f}(x) \)

• Thus \( p_{f}(x) \ q_{f}(x) \ p_{f}^{-1}(x) \) is equivalent to \( q_{f}(x) \)

Undo Operations

• Not all operations have compensating operations
  – For example, \( \text{reset}(x) \), which sets \( x \) to 0, is not one-to-one and has no compensating operation

  – It does have an undo operation, \( \text{set}(x, X) \), which sets the value of \( x \) to what it was right before \( \text{reset}(x) \) was executed.
The Previous Approach Does Not Work

\( \text{reset}_1(x) \text{ reset}_2(x) \text{ set}_1(x, X_1) \)

• Since the two \( \text{reset} \)s commute, we can rewrite the schedule as

\( \text{reset}_2(x) \text{ reset}_1(x) \text{ set}_1(x, X_1) \)

• But this schedule does not undo the result of \( \text{reset}_1(x) \), because the value when \( \text{reset}_1(x) \) starts is different in the second schedule.

What to Do with Undo Operations

• One approach is to require that the operation get an exclusive lock, so that no other operation can come between an operation and its undo operation.

Another Approach

• Suppose \( p^{\text{undo}} \) commutes with \( q \). Then

\[ p q p^{\text{undo}} = p p^{\text{undo}} q \]

• Now \( p \) has the same initial value in both schedules, and thus the undo operation works correctly.

Another Approach

• Theorem

  – Serializability and recoverability is guaranteed if the condition under which an operation \( q \) does not conflict with a previously granted operation \( p \) is

    - \( q \) backward commutes through \( p \), and
    - Either \( p \) has a compensating operation, or when a \( p \) lock is held, \( p^{\text{undo}} \) backward commutes through \( q \).

Still Another Approach

• Sometimes we can decompose an operation that does not have a compensating operation into two partial operations, each of which does have a compensating operation

  - \( \text{withdraw}(x) \) does not have a compensating operation

    • Depending on the initial value of the account, it might perform the withdrawal and decrement that value by \( x \) or it might just return
    • It has an undo operation, \( \text{conditionalDeposit}(x, y) \)

  - The two partial operations, \( \text{withdrawOK}(x) \) and \( \text{withdrawNO}(x) \) are one-to-one and hence do have compensating operations.

Locking Implementation of Savepoints

• When \( T_i \) creates a savepoint, \( s \), insert a marker for \( s \) in \( T_i \)'s lock list, \( z_s \), that separates lock entries acquired before creation from those acquired after creation.

• When \( T_i \) rolls back to \( s \), release all locks preceding marker for \( s \) in \( z_s \) (in addition to undoing all updates made since savepoint creation).
Locking Implementation

- Chaining: nothing new
- Recoverable queue: Since queue is implemented by a separate server (different from DBMS), the locking discipline need not be two-phase; discipline can be designed to suit the semantics of (the abstract operations) `enqueue` and `dequeue`
  - Lock on head (tail) pointer released when dequeue (enqueue) operations complete
  - Hence not strict or isolated
  - Lock on entry that is enqueued or dequeued held to commit time

Recoverable Queue

```
begin transaction
... enqueue(y) release L
... commit release L
```

Locking Implementation of Nested Transactions

- Nested transactions satisfy:
  - Nested transactions are isolated with respect to one another
  - A parent does not execute concurrently with its children
  - A child (and its descendants) is isolated from its siblings (and their descendants)

Intuition

- A request to read `x` by subtransaction `T'` of nested transaction `T` is granted even though an ancestor of `T` holds a write lock on `x`
Locking Implementation of Nested Transactions

- A request to write \( x \) by subtransaction \( T' \) of nested transaction \( T \) is granted if:
  - No other nested transaction holds a read or write lock on \( x \)
  - All other subtransactions of \( T \) holding read or write locks on \( x \) are ancestors of \( T' \) (and hence are not executing)

Example - Switch Sections

Multilevel Transactions

- Generalization of strict two-phase locking concurrency control
  - Uses semantics of operations at each level to determine commutativity
  - Uses different concurrency control at each level

Multilevel Control

- Problem: A control assumes that the execution of operations it schedules is isolated: If \( op_1 \) and \( op_2 \) do not conflict, they can be executed concurrently and the result will be either \( op_1 \), \( op_2 \) or \( op_2 \), \( op_1 \)
  - Not true in a multilevel control where an operation is implemented as a program at the next lower level that might invoke multiple operations at the level below. Hence, concurrent operations at one level might not be totally ordered at the next
Multilevel Transactions

Dec_(s1) and Dec_(s1) commute at L_2 and hence can execute concurrently, but their implementation at L_0 is interleaved.

Guaranteeing Operation Isolation

- **Solution**: Use a concurrency control at each level
  - L_1 receives a request from L_{i+1} to execute op
  - Concurrency control at L_i, CC_i, schedules op to be executed; it assumes execution is isolated
  - op is implemented as a program, P_i, in L_i
  - P is executed as a subtransaction so that it is serializable with respect to other operations scheduled by CC_i
  - Serializability guaranteed by CC_{i-1}

A Multilevel Concurrency Control for the Example

- The control at L_2 uses TestInc and Dec locks
- The control at L_1 uses Sel and Upd locks
- The control at L_0 uses Rd and Wr locks

Timestamp-Ordered Concurrency Control

- Each transaction given a (unique) timestamp (current clock value) when initiated
- Uses the immediate update model
- Guarantees equivalent serial order based on timestamps (initiation order)
  - Control is static (as opposed to dynamic, in which the equivalent serial order is determined as the schedule progresses)

Timestamp-Ordered Concurrency Control

- Associated with each database item, x, are two timestamps:
  - \( w_t(x) \), the largest timestamp of any transaction that has written x,
  - \( r_t(x) \), the largest timestamp of any transaction that has read x,
  - and an indication of whether or not the last write to that item is from a committed transaction
Timestamp-Ordered Concurrency Control

- If T requests to read x:
  - R1: if $TS(T) < wt(x)$, then T is too old; abort T
  - R2: if $TS(T) > wt(x)$, then
    - if the value of x is committed, grant T’s read and if $TS(T) > rt(x)$ assign $TS(T)$ to $rt(x)$
    - if the value of x is not committed, T waits (to avoid a dirty read)

Example

- Assume $TS(T_1) < TS(T_2)$, at $t_0$, x and y are committed, and x’s and y’s read and write timestamps are less than $TS(T_i)$

  $T_1: r(y) w(x) commit$
  $T_2: w(y) w(x) commit$
  $t_0 \ t_1 \ t_2 \ t_3 \ t_4$

  $t_2$: (R2) $TS(T_2) > wt(y)$; assign $TS(T_2)$ to $rt(x)$
  $t_2$: (W3) $TS(T_2) > r(x), wt(y)$; assign $TS(T_2)$ to $wt(y)$
  $t_2$: (W3) $TS(T_2) > r(x), wt(x)$; assign $TS(T_2)$ to $wt(x)$
  $t_2$: (W2) $rt(x) < TS(T_2) < wt(x)$; grant request, but do not do the write

Optimistic Algorithms

- Do task under simplifying (optimistic) assumption
  - Example: Operations rarely conflict
- Check afterwards if assumption was true.
  - Example: Did a conflict occur?
- Redo task if assumption was false
  - Example: If a conflict has occurred rollback, else commit
- Performance benefit if assumption is generally true and check can be done efficiently

Optimistic Concurrency Control

- Under the optimistic assumption that conflicts do not occur, read and write requests are always granted (no locking, no overhead!)
- Since conflicts might occur:
  - Database might be corrupted if writes were immediate, hence a deferred-update model is used
  - Transaction has to be "validated" when it completes
    - If a conflict has occurred abort (but no rollback is necessary) and redo transaction
  - Approach contrasts with pessimistic control which assumes conflicts are likely, takes preventative measures (locking), and does no validation
Optimistic Concurrency Control

- Transaction has three phases:
  - **Begin transaction**
  - **Read Phase**: transaction executes: reads from database, writes to intentions list (deferred-update, no changes to database)
  - **Request commit**
  - **Validation Phase**: check whether conflicts occurred during read phase; if yes abort (discard intentions list)
  - **Commit**
  - **Write Phase**: write intentions list to database (deferred update) if validation successful
- For simplicity, we assume here that validation and write phases form a single critical section (only one transaction is in its validation/write phase at a time)

Validation

- When T1 enters validation, a check is made to see if T1 conflicted with any transaction, T2, that entered validation at an earlier time
- Check uses two sets constructed during read phase:
  - R(T1): identity of all database items T1 read
  - W(T1): identity of all database items T1 wrote

Validation

- **Case 1**: T1’s read phase started after T2 finished its validation/write phase
  - T1 follows T2 in all conflicts, hence commit T1 (T1 follows T2 in equivalent serial order)

Validation

- **Case 2**: T1’s read phase overlaps T2’s validation/write phase
  - If WS(T2) ∩ RS(T1) ≠ Φ, then abort T1
    - A read of T1 might have preceded a write of T2 – a possible violation of equivalent serial order
  - Else commit T1 (T1 follows T2 in equivalent serial order)

Validation

- **Case 3**: T1’s validation/write phase overlaps T2’s validation/write phase
  - Cannot happen since we have assumed that validation/write phases do not overlap
  - Hence, all possible overlaps of T1 and T2 have been considered
Validation

- A more practical optimistic control allows case 3 and avoids the bottleneck implied by only allowing only one transaction at a time in the validation/write phase.

- **Case 3:** T₁’s validation/write phase overlaps T₂’s validation/write phase
  - If \( WS(T_2) \cap (WS(T_1) \cup RS(T_1)) \neq \emptyset \), then abort T₁
    - A read or write of T₁ might have preceded a write of T₂ – a violation of equivalent serial order
  - Else commit T₁ (T₁ follows T₂ in equivalent serial order)

<table>
<thead>
<tr>
<th>T₁ starts</th>
<th>read phase T₁</th>
<th>valid/write phase T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>read phase T₂</td>
<td>valid/write phase T₂</td>
<td>T₁ ends</td>
</tr>
</tbody>
</table>

Optimistic Concurrency Control

- No locking (and hence no waiting) means deadlocks are not possible
- Rollback is a problem if optimistic assumption is not valid: work of entire transaction is lost
  - With two-phase locking, rollback occurs only with deadlock
  - With timestamp-ordered control, rollback is detected before transaction completes