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 incorrect result
  - We are interested in concurrent schedules that are equivalent to serial schedules. These are referred to as serializable schedules.

Transaction Schedule

Consistent - performs correctly when executed in isolation starting in a consistent database state
- Preserves database consistency
- Moves database to a new state that corresponds to new real-world state

Schedule

Arriving schedule (merge of transaction schedules)
Schedule in which requests are serviced
To database

Database server
Schedule

- Representation 1:
  \[ T_1: p_1, p_2, p_3, p_4 \]
  \[ T_2: p_1, p_2 \]
  \[ \text{time} \rightarrow \]

- Representation 2:
  \[ p_{1,1}, p_{1,2}, p_{2,1}, p_{1,3}, p_{2,2}, p_{1,4} \]
  \[ \text{time} \rightarrow \]

Concurrency Control

- Transforms arriving 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 servicing a request

The Inconsistent Analysis Problem

- Occurs when a transaction reads several values from a database while a second transaction updates some of them.

<table>
<thead>
<tr>
<th>T1</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum=0</td>
<td>$100</td>
<td>$50</td>
<td>$25</td>
<td>0</td>
</tr>
<tr>
<td>R(A)</td>
<td>$100</td>
<td>$50</td>
<td>$25</td>
<td>0</td>
</tr>
<tr>
<td>sum=sum+A</td>
<td>$100</td>
<td>$50</td>
<td>$25</td>
<td>100</td>
</tr>
<tr>
<td>A=A-10</td>
<td>$100</td>
<td>$50</td>
<td>$25</td>
<td>100</td>
</tr>
<tr>
<td>R(B)</td>
<td>$90</td>
<td>$50</td>
<td>$25</td>
<td>150</td>
</tr>
<tr>
<td>sum=sum+B</td>
<td>$90</td>
<td>$50</td>
<td>$25</td>
<td>150</td>
</tr>
<tr>
<td>C=C+10</td>
<td>$90</td>
<td>$50</td>
<td>$35</td>
<td>150</td>
</tr>
<tr>
<td>W(C)</td>
<td>$90</td>
<td>$50</td>
<td>$35</td>
<td>150</td>
</tr>
<tr>
<td>R(C)</td>
<td>$90</td>
<td>$50</td>
<td>$35</td>
<td>185</td>
</tr>
<tr>
<td>sum=sum+C</td>
<td>$90</td>
<td>$50</td>
<td>$35</td>
<td>185</td>
</tr>
</tbody>
</table>

Should be 175

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 return the same results and leave the database in the same final state when executed in either order.
Commutativity of 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

- \( p_1 \) commutes with \( p_2 \) if
  - They operate on different data items
    - \( w_1(x) \) commutes with \( w_2(y) \) and \( r_2(y) \)
    - Both are reads
      - \( r_1(x) \) commutes with \( r_2(x) \)
  - Operations that do not commute conflict
    - \( w_1(x) \) conflicts with \( w_2(x) \)
    - \( w_1(x) \) conflicts with \( r_2(x) \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Read(x)</th>
<th>Write(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(x)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Write(x)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Equivalence of Schedules

- An interchange of adjacent operations of different transactions in a schedule creates an equivalent schedule if the operations commute
  \[
  S_1 : S_{1,1}, p_{i,j}, p_{k,l}, S_{1,2} \quad \text{where} \quad i \neq k
  \]
  \[
  S_2 : S_{1,1}, p_{k,l}, p_{i,j}, S_{1,2}
  \]
- Equivalence is transitive: If \( S_1 \) is equivalent to \( S_2 \) (by a series of such interchanges), and \( S_2 \) is equivalent to \( S_3 \), then \( S_1 \) is equivalent to \( S_3 \)

Example of Equivalence

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

- \( S_1 \) 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 \( T_2, T_1 \)
Example of Equivalence

\[ T_1: \text{begin transaction} \]
\[ \text{read } (x, X); \]
\[ X = X + 4; \]
\[ \text{write } (x, X); \]
\[ \text{commit}; \]

\[ T_2: \text{begin transaction} \]
\[ \text{read } (x, Y); \]
\[ \text{write } (y, Y); \]
\[ \text{commit}; \]

Interchange commuting operations

\[ x = 1, y = 3 \]
\[ r_1(x) \quad r_2(x) \quad w_2(y) \quad w_1(x) \]

Interchange conflicting operations

\[ x = 1, y = 3 \]
\[ r_1(x) \quad r_2(x) \quad w_2(y) \quad w_1(x) \]

\[ T_2 \]
\[ T_1 \]

\[ x = 5, y = 1 \]
\[ r_1(x) \quad r_2(x) \quad w_2(y) \quad w_1(x) \]

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 any application if it is a serializable schedule of consistent transactions
- The schedule:
  \[ r_1(x) \quad r_2(y) \quad w_2(x) \quad w_1(y) \]
  is not serializable

Serializable

- **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

  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

  Only if: Commutative interchanges do not reorder conflicting operations

Isolation Levels

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

- **Definition** - Two schedules, $S_1$ and $S_2$, of the same set of operations are *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

Result - A schedule is serializable if it is conflict equivalent to a serial schedule

\[ r_i(x) \ x \ w_j(y) \ r_j(x) \ w_j(y) \ r_2(y) \ x \ r_2(y) \]

- If in $S$ transactions $T_1$ and $T_2$ have several pairs of conflicting operations ($p_{1,1}$ conflicts with $p_{2,1}$ and $p_{1,2}$ conflicts with $p_{2,2}$) then $p_{1,1}$ must precede $p_{2,1}$ and $p_{1,2}$ must precede $p_{2,2}$ (or vice versa) in order for $S$ to be serializable.

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 $T_i$ to node $T_j$ if $T_i$ has an operation $p_{i,k}$ that conflicts with an operation $p_{j,r}$ of $T_j$ and $p_{i,k}$ precedes $p_{j,r}$ in $S$
- **Theorem** - A schedule is conflict serializable if and only if its serialization graph has no cycles
Example

\[ S: \ldots p_{1,r}, \ldots, p_{2,r}, \ldots \]

\[ * \]

\begin{align*}
T_1 & \rightarrow T_3 & T_2 & \rightarrow T_4 & T_5 & \rightarrow T_6 & T_7 \\
T_1 & \rightarrow T_5 & & T_2 & \rightarrow T_4 & & T_6 & \rightarrow T_7 \\
T_1 & \rightarrow T_3 & & T_2 & \rightarrow T_4 & & T_5 & \rightarrow T_6
\end{align*}

\(|\text{Conflict (*)}|\)

S is serializable in order: \(T_1, T_2, T_3, T_4, T_5, T_6, T_7\)

S is not serializable due to cycle: \(T_2, T_6, T_7, T_2\)

Intuition: Serializability and Nonserializability

- Consider the nonserializable schedule:
  \(r_1(x) \ w_2(x) \ r_2(y) \ w_1(y)\)
  \(T_1 \rightarrow T_2\)

- Two ways to think about it:
  - Because of the read and write 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 it, \(T_1\) must follow \(T_2\) in any ordering --- clearly an impossibility.

Recoverability: Schedules with Aborted Transactions

\(T_1: \ r(x) \ w(y) \ \text{commit}\)
\(T_2: \ w(x) \ \text{abort}\)

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

\(T_1: \ r(x) \ w(y) \ \text{req_commit} \ \text{abort}\)
\(T_2: \ w(x) \ \text{abort}\)

Cascaded Abort

- Recoverable schedules solve abort problem but allow \text{cascaded abort}: abort of one transaction forces abort of another.
  \(T_1: \ r(y) \ w(z) \ \text{abort}\)
  \(T_2: \ r(x) \ w(y) \ \text{abort}\)
  \(T_3: \ w(x) \ \text{abort}\)

- Better solution: prohibit dirty reads.
Dirty Write

- **Dirty write**: A transaction writes a data item written by an active transaction
- Dirty write complicates rollback:
  
  no rollback necessary

  \[ \text{T}_1: \ w(x) \quad \text{abort} \]

  \[ \text{T}_2: \ w(x) \quad \text{abort} \]

  what value of x should be restored?

Strict Schedules

- **Strict schedule**: Dirty writes and dirty reads are prohibited
- Strict and serializable are two different properties
  
  - Strict, non-serializable schedule:
    \[ r_1(x) \ w_2(x) \ w_2(y) \ r_2(y) \ r_1(y) \ c_1 \ c_2 \]
  
  - Serializable, non-strict schedule:
    \[ w_2(x) \ r_1(x) \ w_2(y) \ r_1(y) \ c_1 \ c_2 \]

Concurrency Control

Arriving schedule
(from transactions) \[ \rightarrow \] Concurrency Control \[ \rightarrow \] Serializable schedule
(to processing engine)

- 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**
  
  - 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 likely not discuss this)
  
  - A write stores new value in the transaction’s intentions list (does not update database)
  
  - A read copies value from database or transaction’s intentions list
  
  - Commit uses intentions list to durably update database
  
  - Abort discards intentions list
Immediate vs. Deferred Update

Immediate Update

- Database
- Transaction T
- Read/write

Deferred Update

- Database
- Transaction T
- Read/write
- Commit
- T’s intentions list

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
  - The earlier it can aborted the better

Deadlock

- **Problem**: Controls that cause transactions to wait can cause deadlocks
  
  \[ w_1(x) w_2(y) \text{ request } r_1(y) \text{ request } r_2(x) \]

- **Solution**: Abort a 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 Prevention

- Assign priorities based on timestamps (i.e. The oldest transaction has higher priority).
- Assume T_i wants a lock that T_j holds. Two policies are possible:
  - Wait-Die: If T_i has higher priority, T_i allowed to wait for T_j; otherwise (T_i younger) T_i aborts
  - Wound-wait: If T_i has higher priority, T_j aborts; otherwise (T_i younger) T_i waits
- If a transaction re-starts, make sure it has its original timestamp

Deadlock and Timeouts

- A simple approach to deadlock prevention (and pseudo detection) is based on lock timeouts
- After requesting a lock on a locked data object, a transaction waits, but if the lock is not granted within a period (timeout), a deadlock is assumed and the waiting transaction is aborted and re-started.
- Very simple practical solution adopted by many DBMSs.

Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Deadlock exists if there is a cycle in the graph.
- Periodically check for cycles in the waits-for graph.

Deadlock Detection (Continued)

Example:

T1: S(A), R(A), S(B)
T2: X(B), W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)

T1 → T2
T2 → T3
T3 → T3
T4 → T1
T1 → T2
T2 → T3
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 access.

Locking

- Request for read lock granted if no transaction currently holds write lock on 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

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

- All locks held by a transaction are released when the transaction completes (commits or aborts).

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
  - $W(x)$ contains an entry for each pending request
  - 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, $L_i$, with each transaction, $T_i$.
  - $L_i$ links $T_i$’s elements in all lock and wait sets
  - Used to release locks on termination

Two-Phase Locking

- Transaction does not release a lock until it has all the locks it will ever require.
- Transaction, $T$, has a locking phase followed by an unlocking phase
  - Growing phase
  - Shrinking phase
  - Objects are used
  - In strict 2PL all locks are released at one before the transaction commits
- Guarantees serializability when locking is done manually

Theorem: A concurrency control that uses two-phase locking produces only serializable schedules.

- Proof: 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 precedes $T_2$’s first unlock.
  - If they do not access common data items, then all operations commute and $S$ is serializable.
  - Suppose they do. For each common item $x$, all of $T_1$’s accesses to $x$ precede all of $T_2$’s. If this were not the case, $T_2$’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.
  - Thus $S$ is serializable.
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)

T1: \( l(x) \ r(x) \ l(y) \ w(y) \ u(y) \) abort
T2: \( l(y) \ r(y) \ l(z) \ w(z) \ u(z) \ u(y) \) commit

Two-Phase Locking

A two-phase locking control that holds write locks until commit produces strict serializable schedules

A strict two-phase locking control holds all locks until commit and 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

Lock Granularity

- Table locking (**coarse**)
  - Lock entire table when a row is accessed.
- Row (tuple) locking (**fine**)
  - Lock only the row that is accessed.
- Page locking (compromise)
  - When a row is accessed, lock the containing page
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)

Associated with each database item, $x$, are two timestamps:
- $wt(x)$, the largest timestamp of any transaction that has written $x$,
- $rt(x)$, 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

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)

If $T$ requests to write $x$:
- **W1**: If $TS(T) < rt(x)$, then $T$ is too old; abort $T$
- **W2**: If $rt(x) < TS(T) < wt(x)$, then no transaction that read $x$ should have read the value $T$ is attempting to write and no transaction will read that value (R1)
  - if $x$ is committed, grant the request but do not do the write
  - if $x$ is not committed, $T$ waits to see if newer value will commit.
    - If it does, discard $T$’s write, else perform it
- **W3**: If $wt(x), rt(x) < TS(T)$, then if $x$ is committed, grant the request and assign $TS(T)$ to $wt(x)$, else $T$ waits
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 $TS(T_1)$

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

$t_0$ $t_1$ $t_2$ $t_3$ $t_4$

$t_1$: (R2) $TS(T_1) > wt(y)$; assign $TS(T_1)$ to $rt(y)$

$t_2$: (W3) $TS(T_2) > rt(y), wt(y)$; assign $TS(T_2)$ to $wt(y)$

$t_3$: (W3) $TS(T_2) > rt(x), wt(x)$; assign $TS(T_2)$ to $wt(x)$

$t_4$: (W2) $rt(x) < TS(T_1) < wt(x)$; grant request, but don’t do the write

Timestamp-Ordered Concurrency Control

- Control accepts schedules that are not conflict equivalent to any serial schedule and would not be accepted by a two-phase locking control
  - Previous example equivalent to $T_1, T_2$
- But additional space required in database for storing timestamps and time for managing timestamps
  - Reading a data item now implies writing back a new value of its timestamp

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