Concurrency Control

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.

- Principles:
  - We want to interleave the execution of transactions for performance reasons
    - E.g., execute operations of another transaction when the first one starts doing I/O.
  - However, we want the results of interleaved executions to be equivalent to non-interleaved execution for correctness
    - We need to be able to reason about the execution order of transactions.
Potential Anomalies Due to Concurrent Execution

- **Lost updates**
  - The effects of some transactions are not reflected in the database.
  - Transaction $T_2$ reading uncommitted changes to data made by transaction $T_1$.
    - Write-Read conflicts
  - Transaction $T_2$ overwriting uncommitted changes of transaction $T_1$.
    - Write-Write conflicts

- **Inconsistent retrievals (unrepeatable reads)**
  - A transaction, if it reads the same data item more than once, should always read the same value.
  - Transaction $T_2$ modifies data that is being accessed by transaction $T_1$.
    - Read-Write conflicts
Execution Schedule (or History)

- An order in which the operations of a set of transactions are executed.
- A schedule (history) can be defined as a partial order over the operations of a set of transactions.

\[ H_1 = W_2(x) R_1(x) R_3(x) W_1(x) C_1 W_2(y) R_3(y) R_2(z) C_2 R_3(z) C_3 \]
A complete schedule $SC(T)$ over a set of transactions $T=\{T_1, \ldots, T_n\}$ is a partial order $SC(T)=\{\Sigma_T, <_T\}$ where

1. $\Sigma_T = \bigcup_i \Sigma_i$, for $i = 1, 2, \ldots, n$

2. $<_T \supseteq \bigcup_i <_i$, for $i = 1, 2, \ldots, n$

3. For any two conflicting operations $o_{ij}, o_{kl} \in \Sigma_T$, either $o_{ij} <_T o_{kl}$ or $o_{kl} <_T o_{ij}$

(Remember: $o_{ij}$ is an operation of transaction $T_i$)
Complete Schedule – Example

Given three transactions

\( T_1: \) Read(\( x \))
Write(\( x \))
Commit

\( T_2: \) Write(\( x \))
Write(\( y \))
Read(\( z \))
Commit

\( T_3: \) Read(\( x \))
Read(\( y \))
Read(\( z \))
Commit

A possible complete schedule is given as the DAG

```
R_1(\( x \)) \rightarrow W_2(\( x \)) \rightarrow R_3(\( x \))
```
```
R_1(\( x \)) \rightarrow W_1(\( x \)) \rightarrow W_2(\( y \)) \rightarrow R_3(\( y \))
```
```
R_1(\( x \)) \rightarrow W_1(\( x \)) \rightarrow W_2(\( y \)) \rightarrow R_3(\( z \)) \rightarrow R_3(\( z \))
```
```
C_1 \rightarrow R_2(\( z \)) \rightarrow R_3(\( z \))
```
```
C_2 \rightarrow C_3
```

9-5
Schedule Definition

A schedule is a prefix of a complete schedule such that only some of the operations and only some of the ordering relationships are included.

\[ \begin{align*}
T_1: & \quad \text{Read}(x) \\
& \quad \text{Write}(x) \\
& \quad \text{Commit} \\
\end{align*} \quad \begin{align*}
T_2: & \quad \text{Write}(x) \\
& \quad \text{Write}(y) \\
& \quad \text{Read}(z) \\
& \quad \text{Commit} \\
\end{align*} \quad \begin{align*}
T_3: & \quad \text{Read}(x) \\
& \quad \text{Read}(y) \\
& \quad \text{Read}(z) \\
& \quad \text{Commit} \\
\end{align*} \]

\[ \begin{align*}
R_1(x) \quad W_2(x) \quad R_3(x) \\
W_1(x) \quad W_2(y) \quad R_3(y) \quad R_1(x) \quad W_2(x) \quad R_3(x) \\
C_1 \quad R_2(z) \quad R_3(z) \quad W_2(y) \quad R_3(y) \\
C_2 \quad C_3 \quad R_2(z) \quad R_3(z) \\
\end{align*} \]
Serial Schedule

- All the actions of a transaction occur consecutively.
- No interleaving of transaction operations.
- If each transaction is consistent (obeys integrity rules), then the database is guaranteed to be consistent at the end of executing a serial schedule.

\[ H_s = W_2(x) W_2(y) R_2(z) C_2 R_1(x) W_1(x) C_1 R_3(x) R_3(y) R_3(z) C_3 \]

\[ T_2 \rightarrow T_1 \rightarrow T_3 \]
Serializable Schedule

- Transactions execute concurrently, but the net effect of the resulting schedule upon the database is equivalent to some serial schedule.

- Equivalent with respect to what?
  - Conflict equivalence: the relative order of execution of the conflicting operations belonging to committed transactions in two schedules are the same.
  - Conflicting operations: two incompatible operations (e.g., Read and Write) conflict if they both access the same data item.
    - Incompatible operations of each transaction is assumed to conflict; do not change their execution orders.
    - If two operations from two different transactions conflict, the corresponding transactions are also said to conflict.
Serializable Schedule

\begin{align*}
T_1: & \quad \text{Read}(x) \quad \text{Write}(x) \quad \text{Commit} \\
T_2: & \quad \text{Write}(x) \quad \text{Write}(y) \quad \text{Read}(z) \quad \text{Commit} \\
T_3: & \quad \text{Read}(x) \quad \text{Read}(y) \quad \text{Read}(z) \quad \text{Commit}
\end{align*}

The following are not conflict equivalent

\begin{align*}
H_s &= \text{W_2}(x) \text{ W_2}(y) \text{ R_2}(z) \text{ C_2} \text{ R_1}(x) \text{ W_1}(x) \text{ C_1} \text{ R_3(x)} \text{ R_3(y)} \text{ R_3(z)} \text{ C_3} \\
H_1 &= \text{W_2}(x) \text{ R_1(x)} \text{ R_3(x)} \text{ W_1(x)} \text{ C_1} \text{ W_2(y)} \text{ R_3(y)} \text{ R_2(z)} \text{ C_2} \text{ R_3(z)} \text{ C_3}
\end{align*}

The following are conflict equivalent; therefore \(H_2\) is \textit{serializable}.

\begin{align*}
H_s &= \text{W_2}(x) \text{ W_2}(y) \text{ R_2(z)} \text{ C_2} \text{ R_1(x)} \text{ W_1(x)} \text{ C_1} \text{ R_3(x)} \text{ R_3(y)} \text{ R_3(z)} \text{ C_3} \\
H_2 &= \text{W_2}(x) \text{ R_1(x)} \text{ W_1(x)} \text{ C_1} \text{ R_3(x)} \text{ W_2(y)} \text{ R_3(y)} \text{ R_2(z)} \text{ C_2} \text{ R_3(z)} \text{ C_3}
\end{align*}
Serializability Graph

- Serializability graph $SG_H = \{V, E\}$ for schedule $H$:
  - $V = \{T \mid T$ is a committed transaction in $H\}$
  - $E = \{T_i \rightarrow T_j \text{ if } o_{ij} \in T_i \text{ and } o_{kl} \in T_k \text{ conflict and } o_{ij} <_H o_{kl}\}$

- Theorem: Schedule $H$ is serializable iff $SG_H$ does not contain any cycles.
Concurrence Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
  - Timestamp Ordering (TO)
- Optimistic
Locking-Based Algorithms

- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (rl) [also called shared lock] or write lock (wl) [also called exclusive lock].
- Read locks and write locks conflict (because Read and Write operations are incompatible)

<table>
<thead>
<tr>
<th></th>
<th>rl</th>
<th>wl</th>
</tr>
</thead>
<tbody>
<tr>
<td>rl</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>wl</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

- Locking works nicely to allow concurrent processing of transactions.
Two-Phase Locking (2PL)

1. A Transaction locks an object before using it.
2. When an object is locked by another transaction, the requesting transaction must wait.
3. When a transaction releases a lock, it may not request another lock.
Strict 2PL

Hold locks until the end.

No. of locks

Obtain lock
Release lock

BEGIN END

period of data item use

Transaction duration
Timestamp Ordering

1. Transaction \((T_i)\) is assigned a globally unique timestamp \(ts(T_i)\).
2. Transaction manager attaches the timestamp to all operations issued by the transaction.
3. Each data item is assigned a write timestamp \((wts)\) and a read timestamp \((rts)\):
   - \(rts(x)\) = largest timestamp of any read on \(x\)
   - \(wts(x)\) = largest timestamp of any write on \(x\)
4. Conflicting operations are resolved by timestamp order.

Basic T/O:

For \(R_i(x)\):
- \(\text{if } ts(T_i) < wts(x)\)
  - \(\text{then reject } R_i(x)\)
  - \(\text{else } \{ \text{accept } R_i(x) \}\)
  - \(rts(x) \leftarrow ts(T_i)\)

For \(W_i(x)\):
- \(\text{if } ts(T_i) < rts(x) \text{ or } ts(T_i) < wts(x)\)
  - \(\text{then reject } W_i(x)\)
  - \(\text{else } \{ \text{accept } W_i(x) \}\)
  - \(wts(x) \leftarrow ts(T_i)\)
Multiversion Timestamp Ordering

- Do not modify the values in the database, create new values.

- A $R_i(x)$ is translated into a read on one version of $x$.
  - Find a version of $x$ (say $x_v$) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$.

- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that

$$ts(T_i) < ts(x_r) < ts(T_j)$$

$W_i(x)$: Some other transaction created $x_r$

$R_j(x_r)$
Optimistic Concurrency Control Algorithms

Pessimistic execution

Validate Read Compute Write

Optimistic execution

Read Compute Validate Write
Optimistic CC Validation Test

1. If all transactions $T_k$ where $ts(T_k) < ts(T_i)$ have completed their write phase before $T_i$ has started its read phase, then validation succeeds.

   - Transaction executions in serial order

   $T_k | R | V | W |

   $T_i | R | V | W |
Optimistic CC Validation Test

If there is any transaction $T_k$ such that $ts(T_k)<ts(T_i)$ and which completes its write phase while $T_i$ is in its read phase, then validation succeeds if $WS(T_k) \cap RS(T_i) = \emptyset$

- Read and write phases overlap, but $T_i$ does not read data items written by $T_k$

\[
\begin{array}{c|c|c|c|c}
T_k & R & V & W \\
\mid & & & & \\
T_i & R & V & W \\
\end{array}
\]
Optimistic CC Validation Test

3. If there is any transaction $T_k$ such that $ts(T_k) < ts(T_i)$ and which completes its read phase before $T_i$ completes its read phase, then validation succeeds if $WS(T_k) \cap RS(T_i) = \emptyset$ and $WS(T_k) \cap WS(T_i) = \emptyset$

- They overlap, but don't access any common data items.

```
<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>V</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_k$</td>
<td>R</td>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>$T_i$</td>
<td>R</td>
<td>V</td>
<td>W</td>
</tr>
</tbody>
</table>
```
Deadlock

- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- Wait-for graph
  - If transaction $T_i$ waits for another transaction $T_j$ to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.
Deadlock Management

- Prevention
  - Guaranteeing that deadlocks can never occur in the first place. Check transaction when it is initiated. Requires no run time support.

- Avoidance
  - Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support.

- Detection and Recovery
  - Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.
Deadlock Prevention

■ All resources that may be needed by a transaction must be predeclared.
  - The system must guarantee that none of the resources will be needed by an ongoing transaction.
  - Resources must only be reserved, but not necessarily allocated a priori
  - Unsuitable in database environment
  - Suitable for systems that have no provisions for undoing processes.

■ Evaluation:
  - Reduced concurrency due to pre-allocation
  - Evaluating whether an allocation is safe leads to added overhead.
  - Difficult to determine (partial order)
  + No transaction rollback or restart is caused.
Deadlock Avoidance

- Transactions are not required to request resources a priori.
- Transactions are allowed to proceed unless a requested resource is unavailable.
- In case of conflict, transactions may be allowed to wait for a fixed time interval.
- Order the data items and always request locks in that order.
- More attractive than prevention in a database environment.
**Deadlock Avoidance – Wait-Die & Wound-Wait Algorithms**

**WAIT-DIE Rule:** If $T_i$ requests a lock on a data item which is already locked by $T_j$, then $T_i$ is permitted to wait iff $ts(T_i) < ts(T_j)$. If $ts(T_i) > ts(T_j)$, then $T_i$ is aborted and restarted with the same timestamp.

- if $ts(T_i) < ts(T_j)$ then $T_i$ waits
- else $T_i$ dies
- non-preemptive: $T_i$ never preempts $T_j$

**WOUND-WAIT Rule:** If $T_i$ requests a lock on a data item which is already locked by $T_j$, then $T_i$ is permitted to wait iff $ts(T_i) > ts(T_j)$. If $ts(T_i) < ts(T_j)$, then $T_j$ is aborted and the lock is granted to $T_i$.

- if $ts(T_i) < ts(T_j)$ then $T_j$ is wounded
- else $T_i$ waits
- preemptive: $T_i$ preempts $T_j$ if it is younger
Deadlock Detection

- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.