Chapter 12 : Concurrency Control
Chapter 12: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes:
  1. *shared (S)* mode. Data item can only be read. S-lock is requested using `lock-S` instruction.
  2. *exclusive (X)* mode. Data item can be both read as well as written. X-lock is requested using `lock-X` instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
- Example of a transaction performing locking:
  \[ T_2: \text{lock-}S(A); \]
  \[ \text{read} (A); \]
  \[ \text{unlock}(A); \]
  \[ \text{lock-}S(B); \]
  \[ \text{read} (B); \]
  \[ \text{unlock}(B); \]
  \[ \text{display}(A+B) \]
- Locking as above is not sufficient to guarantee serializability — if \( A \) and \( B \) get updated in-between the read of \( A \) and \( B \), the displayed sum would be wrong.
- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.
Pitfalls of Lock-Based Protocols

- Consider the partial schedule

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-x($B$)</td>
<td>lock-S($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$B := B - 50$</td>
<td>lock-S($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>

- Neither $T_3$ nor $T_4$ can make progress — executing **lock-S($B$)** causes $T_4$ to wait for $T_3$ to release its lock on $B$, while executing **lock-X($A$)** causes $T_3$ to wait for $T_4$ to release its lock on $A$.

- Such a situation is called a **deadlock**.
  - To handle a deadlock one of $T_3$ or $T_4$ must be rolled back and its locks released.
The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

**Starvation** is also possible if concurrency control manager is badly designed. For example:

- A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
- The same transaction is repeatedly wait.

Concurrency control manager can be designed to prevent starvation.
The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).
The Two-Phase Locking Protocol (Cont.)

- Two-phase locking does not ensure freedom from deadlocks.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
Two-phase locking does not ensure freedom from cascading rollbacks

<table>
<thead>
<tr>
<th>$T_5$</th>
<th>$T_6$</th>
<th>$T_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-$X(A)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock-$S(B)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock-$X(A)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock($A$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock-$S(A)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read($A$)</td>
</tr>
</tbody>
</table>
The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
  Given a transaction $T_i$ that does not follow two-phase locking, we can find a transaction $T_j$ that uses two-phase locking, and a schedule for $T_i$ and $T_j$ that is not conflict serializable.

两段锁协议是保证冲突可串行化调度的充分条件
Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)

- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.
Automatic Acquisition of Locks

• A transaction $T_i$ issues the standard read/write instruction, without explicit locking calls.
• The operation $\text{read}(D)$ is processed as:

```plaintext
if $T_i$ has a lock on $D$
then
    read($D$)
else begin
    if necessary wait until no other transaction has a $\text{lock-X}$ on $D$
    grant $T_i$ a $\text{lock-S}$ on $D$;
    read($D$)
end
```
• **write**(*D*) is processed as:

  if *T*<sub>i</sub> has a **lock-X** on *D*
  then
    write(*D*)
  else begin
    if necessary wait until no other trans. has any lock on *D*,
    if *T*<sub>i</sub> has a **lock-S** on *D*
    then
      upgrade lock on *D* to **lock-X**
    else
      grant *T*<sub>i</sub> a **lock-X** on *D*
    write(*D*)
  end;

• All locks are released after commit or abort
Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking.
- Impose a partial ordering $\rightarrow$ on the set $D = \{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both $d_i$ and $d_j$ must access $d_i$ before accessing $d_j$.
  - Implies that the set $D$ may now be viewed as a directed acyclic graph, called a database graph.
- The tree-protocol is a simple kind of graph protocol.
Tree Protocol

1. Only exclusive locks are allowed.
2. The first lock by $T_i$ may be on any data item. Subsequently, a data $Q$ can be locked by $T_i$ only if the parent of $Q$ is currently locked by $T_i$.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by $T_i$ cannot subsequently be relocked by $T_i$. 
Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.
## Serializable Schedule Under the Tree Protocol

<table>
<thead>
<tr>
<th>$T_{10}$</th>
<th>$T_{11}$</th>
<th>$T_{12}$</th>
<th>$T_{13}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X($B$)</td>
<td>lock-X($D$)</td>
<td>lock-X($B$)</td>
<td>lock-X($D$)</td>
</tr>
<tr>
<td>lock-X($E$)</td>
<td>lock-X($H$)</td>
<td>lock-X($B$)</td>
<td>lock-X($H$)</td>
</tr>
<tr>
<td>lock-X($D$)</td>
<td>unlock($D$)</td>
<td>lock-X($E$)</td>
<td>unlock($D$)</td>
</tr>
<tr>
<td>unlock($B$)</td>
<td>unlock($H$)</td>
<td>unlock($E$)</td>
<td>unlock($D$)</td>
</tr>
<tr>
<td>unlock($E$)</td>
<td>unlock($H$)</td>
<td>unlock($H$)</td>
<td>unlock($H$)</td>
</tr>
<tr>
<td>lock-X($D$)</td>
<td>unlock($E$)</td>
<td>unlock($E$)</td>
<td>unlock($H$)</td>
</tr>
<tr>
<td>unlock($G$)</td>
<td>unlock($B$)</td>
<td>unlock($B$)</td>
<td>unlock($H$)</td>
</tr>
</tbody>
</table>
Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.

- Can be represented graphically as a tree (but don't confuse with tree-locking protocol).

- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency
The levels, starting from the coarsest (top) level are

- database
- area
- file
- record
Intention Lock Modes

• In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  • *intention-shared* (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  • *intention-exclusive* (IX): indicates explicit locking at a lower level with exclusive or shared locks
  • *shared and intention-exclusive* (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
  • intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.
Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>S IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S IX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>IS</td>
<td>IX</td>
<td>S</td>
<td>SIX</td>
<td>X</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>IS</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>IX</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>SIX</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
Multiple Granularity Locking Scheme

- Transaction $T_i$ can lock a node $Q$, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node $Q$ can be locked by $T_i$ in S or IS mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or IS mode.
  4. A node $Q$ can be locked by $T_i$ in X, SIX, or IX mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or SIX mode.
  5. $T_i$ can lock a node only if it has not previously unlocked any node (that is, $T_i$ is two-phase).
  6. $T_i$ can unlock a node $Q$ only if none of the children of $Q$ are currently locked by $T_i$.

- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
Other Approaches to Concurrency Control
Each transaction is issued a timestamp when it enters the system. If an old transaction $T_i$ has time-stamp $TS(T_i)$, a new transaction $T_j$ is assigned time-stamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.

The protocol manages concurrent execution such that the time-stamps determine the serializability order.

In order to assure such behavior, the protocol maintains for each data $Q$ two timestamp values:

- **W-timestamp**($Q$) is the largest time-stamp of any transaction that executed **write**($Q$) successfully.
- **R-timestamp**($Q$) is the largest time-stamp of any transaction that executed **read**($Q$) successfully.
 timestamp ordering protocol ensures that any conflicting \textbf{read} and \textbf{write} operations are executed in timestamp order.

Suppose a transaction $T_i$ issues a \textbf{read}(Q)

1. If $TS(T_i) \leq W$-timestamp$(Q)$, then $T_i$ needs to read a value of $Q$ that was already overwritten.
   
   \begin{itemize}
   \item Hence, the \textbf{read} operation is rejected, and $T_i$ is rolled back.
   \end{itemize}

2. If $TS(T_i) \geq W$-timestamp$(Q)$, then the \textbf{read} operation is executed, and $R$-timestamp$(Q)$ is set to $\text{max}(R$-timestamp$(Q), TS(T_i))$. 
Suppose that transaction $T_i$ issues write($Q$).

1. If $TS(T_i) < R$-timestamp($Q$), then the value of $Q$ that $T_i$ is producing was needed previously, and the system assumed that that value would never be produced.
   - Hence, the write operation is rejected, and $T_i$ is rolled back.

2. If $TS(T_i) < W$-timestamp($Q$), then $T_i$ is attempting to write an obsolete value of $Q$.
   - Hence, this write operation is rejected, and $T_i$ is rolled back.

3. Otherwise, the write operation is executed, and $W$-timestamp($Q$) is set to $TS(T_i)$. 
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$) write($Z$)</td>
<td>read($X$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read($X$) abort</td>
<td></td>
<td>write($Z$) abort</td>
<td></td>
<td>read($Z$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write($Y$) write($Z$)</td>
<td>write($Z$)</td>
</tr>
</tbody>
</table>
Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

![Diagram showing precedence graph with arcs from transaction with smaller timestamp to transaction with larger timestamp]

Thus, there will be no cycles in the precedence graph.

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

- But the schedule may not be cascade-free, and may not even be recoverable.
Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose $T_i$ aborts, but $T_j$ has read a data item written by $T_i$
  - Then $T_j$ must abort; if $T_j$ had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by $T_j$ must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks

- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp

- Solution 2: Limited form of locking: wait for data to be committed before reading it

- Solution 3: Use commit dependencies to ensure recoverability
Thomas’ Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.

- When $T_i$ attempts to write data item $Q$, if $TS(T_i) < W$-timestamp($Q$), then $T_i$ is attempting to write an obsolete value of \{Q\}.
  - Rather than rolling back $T_i$ as the timestamp ordering protocol would have done, this \{write\} operation can be ignored.

- Otherwise this protocol is the same as the timestamp ordering protocol.

- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.
Validation-Based Protocol

- Execution of transaction \( T_i \) is done in three phases.

1. **Read and execution phase**: Transaction \( T_i \) writes only to temporary local variables.

2. **Validation phase**: Transaction \( T_i \) performs a "validation test" to determine if local variables can be written without violating serializability.

3. **Write phase**: If \( T_i \) is validated, the updates are applied to the database; otherwise, \( T_i \) is rolled back.

- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially.
    - I.e., only one transaction executes validation/write at a time.
  - Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation.
Validation-Based Protocol (Cont.)

• Each transaction $T_i$ has 3 timestamps
  • $\text{Start}(T_i)$: the time when $T_i$ started its execution
  • $\text{Validation}(T_i)$: the time when $T_i$ entered its validation phase
  • $\text{Finish}(T_i)$: the time when $T_i$ finished its write phase

• Serializability order is determined by timestamp given at validation time, to increase concurrency.
  • Thus $\text{TS}(T_i)$ is given the value of $\text{Validation}(T_i)$.

• This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  • because the serializability order is not pre-decided, and
  • relatively few transactions will have to be rolled back.
Validation Test for Transaction $T_j$

- If for all $T_i$ with $TS(T_i) < TS(T_j)$ either one of the following condition holds:
  - $\text{finish}(T_i) < \text{start}(T_j)$
  - $\text{start}(T_j) < \text{finish}(T_i) < \text{validation}(T_j)$ and the set of data items written by $T_i$ does not intersect with the set of data items read by $T_j$.

  then validation succeeds and $T_j$ can be committed. Otherwise, validation fails and $T_j$ is aborted.

- Justification: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of $T_j$ do not affect reads of $T_i$ since they occur after $T_i$ has finished its reads.
  - the writes of $T_i$ do not affect reads of $T_j$ since $T_j$ does not read any item written by $T_i$. 

Example of schedule produced using validation

<table>
<thead>
<tr>
<th>$T_{14}$</th>
<th>$T_{15}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read(B)</code></td>
<td><code>read(B)</code></td>
</tr>
<tr>
<td><code>read(A)</code></td>
<td><code>B := B - 50</code></td>
</tr>
<tr>
<td><code>(validate)</code></td>
<td><code>read(A)</code></td>
</tr>
<tr>
<td><code>display (A+B)</code></td>
<td><code>A := A + 50</code></td>
</tr>
</tbody>
</table>

`(validate)`
`write (B)`
`write (A)`
Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful write results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a read\((Q)\) operation is issued, select an appropriate version of \(Q\) based on the timestamp of the transaction, and return the value of the selected version.
- reads never have to wait as an appropriate version is returned immediately.
Each data item $Q$ has a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$. Each version $Q_k$ contains three data fields:

- **Content** -- the value of version $Q_k$.
- **W-timestamp** $(Q_k)$ -- timestamp of the transaction that created (wrote) version $Q_k$.
- **R-timestamp** $(Q_k)$ -- largest timestamp of a transaction that successfully read version $Q_k$.

When a transaction $T_i$ creates a new version $Q_k$ of $Q$, $Q_k$'s W-timestamp and R-timestamp are initialized to $TS(T_i)$.

R-timestamp of $Q_k$ is updated whenever a transaction $T_j$ reads $Q_k$, and $TS(T_j) > R$-timestamp$(Q_k)$. 

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction $T_i$ issues a **read**(Q) or **write**(Q) operation. Let $Q_k$ denote the version of Q whose write timestamp is the largest write timestamp less than or equal to $TS(T_i)$.
  1. If transaction $T_i$ issues a **read**(Q), then the value returned is the content of version $Q_k$.
  2. If transaction $T_i$ issues a **write**(Q)
     1. if $TS(T_i) < R$-timestamp($Q_k$), then transaction $T_i$ is rolled back.
     2. if $TS(T_i) = W$-timestamp($Q_k$), the contents of $Q_k$ are overwritten
     3. else a new version of Q is created.

- Observe that
  - Reads always succeed
  - A write by $T_i$ is rejected if some other transaction $T_j$ that (in the serialization order defined by the timestamp values) should read $T_i$'s write, has already read a version created by a transaction older than $T_i$.

- Protocol guarantees serializability
Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful *write* results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a counter `ts-counter` that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading the current value of `ts-counter` before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.
When an update transaction wants to read a data item:
- it obtains a shared lock on it, and reads the latest version.

When it wants to write an item
- it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to $\infty$.

When update transaction $T_i$ completes, commit processing occurs:
- $T_i$ sets timestamp on the versions it has created to $\text{ts-counter} + 1$
- $T_i$ increments $\text{ts-counter}$ by 1

Read-only transactions that start after $T_i$ increments $\text{ts-counter}$ will see the values updated by $T_i$.

Read-only transactions that start before $T_i$ increments the $\text{ts-counter}$ will see the value before the updates by $T_i$.

Only serializable schedules are produced.
MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
Deadlock Handling

- Consider the following two transactions:

  \[ T_1: \text{write}(X) \quad T_2: \text{write}(Y) \]
  \[ \text{write}(Y) \quad \text{write}(X) \]

- Schedule with deadlock

<table>
<thead>
<tr>
<th></th>
<th>(T_1)</th>
<th>(T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-(X) on (X)</td>
<td>write ((X))</td>
<td>lock-(X) on (Y)</td>
</tr>
<tr>
<td>write ((X))</td>
<td></td>
<td>write ((X))</td>
</tr>
<tr>
<td>wait for lock-(X) on (Y)</td>
<td></td>
<td>wait for lock-(X) on (X)</td>
</tr>
</tbody>
</table>
Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item

- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.
Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

- **Timeout-Based Schemes:**
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.
Deadlock Detection

• Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V, E)$,
  • $V$ is a set of vertices (all the transactions in the system)
  • $E$ is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.

• If $T_i \rightarrow T_j$ is in $E$, then there is a directed edge from $T_i$ to $T_j$, implying that $T_i$ is waiting for $T_j$ to release a data item.

• When $T_i$ requests a data item currently being held by $T_j$, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when $T_j$ is no longer holding a data item needed by $T_i$.

• The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.
Deadlock Detection (Cont.)

Wait-for graph without a cycle

Wait-for graph with a cycle
Deadlock Recovery

• When deadlock is detected:
  • Some transaction will have to be rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  • Rollback -- determine how far to roll back transaction
    • Total rollback: Abort the transaction and then restart it.
    • More effective to roll back transaction only as far as necessary to break deadlock.
  • Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation
Insert and Delete Operations

- If two-phase locking is used:
  - A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple.
- Insertions and deletions can lead to the phantom phenomenon.
  - A transaction that scans a relation
    - (e.g., find sum of balances of all accounts in Perryridge)
  and a transaction that inserts a tuple in the relation
    - (e.g., insert a new account at Perryridge)
  (conceptually) conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result
  - E.g. the scan transaction does not see the new account, but reads some other tuple written by the update transaction.
Insert and Delete Operations (Cont.)

The usual anomalies

• dirty read
• lost update
• non-repeatable read
The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information.

- The information should be locked.

One solution:

- Associate a data item with the relation, to represent the information about what tuples the relation contains.
- Transactions scanning the relation acquire a shared lock in the data item,
- Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)

Above protocol provides very low concurrency for insertions/deletions.

Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.
Index Locking Protocol

- Index locking protocol:
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation.
  - A transaction $T_i$ that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode.
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g., for a range query, no tuple in a leaf is in the range).
  - A transaction $T_i$ that inserts, updates or deletes a tuple $t_i$ in a relation $r$.
    - must update all indices to $r$.
    - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete.
  - The rules of the two-phase locking protocol must be observed.
  - Guarantees that phantom phenomenon won’t occur.
Weak Levels of Consistency

- **Degree-two consistency**: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]

- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency
Weak Levels of Consistency in SQL

• SQL allows non-serializable executions
  • **Serializable**: is the default
  • **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    • However, the phantom phenomenon need not be prevented
      • T1 may see some records inserted by T2, but may not see others inserted by T2
  • **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  • **Read uncommitted**: allows even uncommitted data to be read

• In many database systems, read committed is the default consistency level
  • has to be explicitly changed to serializable when required
    • set isolation level serializable