# Chapter 15 : Concurrency Control

**Database System Concepts, 6th Ed.**

# Chapter 15: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures

# 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. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

# Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

| | S | X |
|---|---|---|
| S | true | false |
| X | false | false |

- 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.

# Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  $T_2$: **lock-S**(*A*);
  **read** (*A*);
  **unlock**(*A*);
  **lock-S**(*B*);
  **read** (*B*);
  **unlock**(*B*);
  **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

| $T_3$ | $T_4$ |
|---|---|
| lock-x ($B$) | |
| read ($B$) | |
| $B := B - 50$ | |
| write ($B$) | |
| | lock-s ($A$) |
| | read ($A$) |
| | lock-s ($B$) |
| lock-x ($A$) | |

- 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.

# Pitfalls of Lock-Based Protocols (Cont.)

- 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 rolled back due to deadlocks.
- 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.

# Quiz Time

**Quiz Q1**: Consuder the following locking schedule

| T1 |
| --- |
| lock-X(A) |
| unlock-X(A) |
| lock-S(B) |
| unlock-S(B) |

(1) the schedule is two phase
(2) the schedule is recoverable
(3) the schedule is cascade free
(4) none of the above

# 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 **read**(*D*) is processed as:

```
if Ti has a lock on D
  then
      read(D)
  else begin
      if necessary wait until no other
         transaction has a lock-X on D
      grant Ti a  lock-S on D;
      read(D)
  end
```

# Automatic Acquisition of Locks (Cont.)

- **write**(*D*) is processed as:

```
if Ti has a  lock-X on D
   then
     write(D)
  else begin
       if necessary wait until no other trans. has any lock on D,
       if Ti has a lock-S on D
            then
                upgrade lock on D  to lock-X
            else
                grant Ti a lock-X on D
            write(D)
    end;
```

- All locks are released after commit or abort

# Implementation of Locking

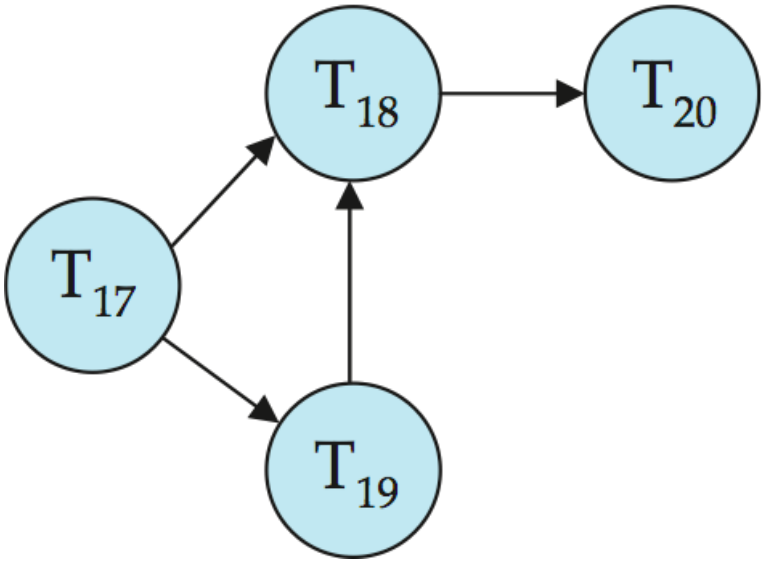
- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

# Deadlock Handling

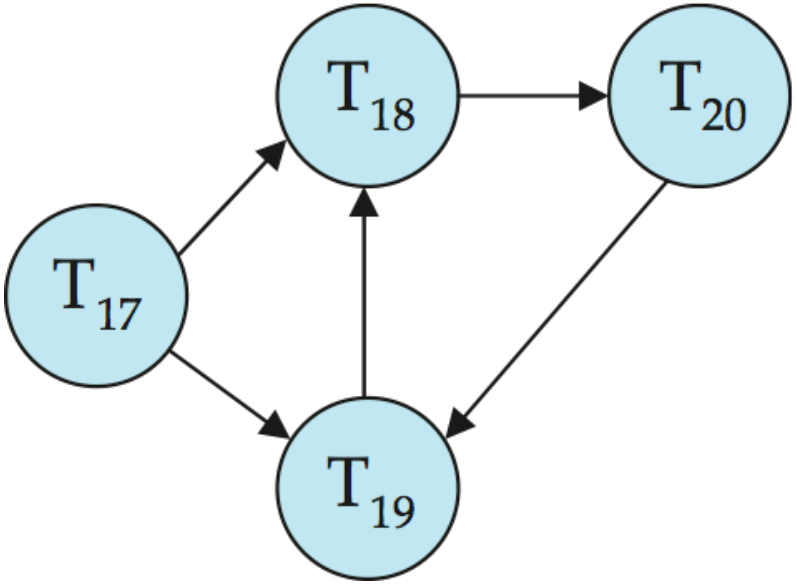
- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction 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).
  - Deadlock prevention by ordering usually ensured by careful programming of transactions

# Deadlock Detection

- ***Deadlock detection*** algorithms used to detect deadlocks




Wait-for graph without a cycle




Wait-for graph with a cycle

# Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to 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

# Quiz Time

**Quiz Q2**: Consuder the following locking schedule

| T1 | T2 |
|---|---|
| lock-S(A) | |
| | lock-S(B) |
| lock-X(B) | |
| | lock-X(B) |

(1) the schedule is not two phase (2) the schedule is deadlocked
(3) the schedule is not deadlocked (4) none of the above

# Locking Extensions

- **Multiple granularity locking:**
  - idea: instead of getting separate locks on each record
    - lock an entire page explicitly, implicitly locking all records in the page, or
    - lock an entire relation, implicitly locking all records in the relation
  - See book for details of multiple-granularity locking

# Phantom Problem

- Insertions, deletions and updates can lead to the **phantom phenomenon**.
  - A transaction that scans a relation
    - (e.g., T1: find list of students taking CS 101)

    and a transaction that inserts a tuple in the relation
    - (e.g., T2: insert a new student in CS 101)

    **(conceptually) conflict in spite of not accessing any tuple in common**.
  - If only tuple locks are used, non-serializable schedules can result
    - T1 finds students taking CS 101,
    - T2 inserts student 10101 in CS 101
    - T2 updates tot_cred of student 10101, and commits
    - T1 reads tot_cred of student 10101 (value after T2 updates it)
  - Index locking protocols used to prevent phantom phenomenon (see book for details)

# Timestamp-Based Protocols

- 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-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction $T_i$ issues a **read**($Q$)
  1. If TS($T_i$) ≤ **W**-timestamp($Q$), then $T_i$ needs to read a value of $Q$ that was already overwritten.
     - Hence, the **read** operation is rejected, and $T_i$ is rolled back.
  2. If TS($T_i$)≥ **W**-timestamp($Q$), then the **read** operation is executed, and R-timestamp($Q$) is set to **max**(R-timestamp($Q$), TS($T_i$)).

# Timestamp-Based Protocols (Cont.)

- 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($T_i$).

# Validation-Based Protocols

- 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 Protocols (Cont.)

- Validation is based on two principles
  1. Tracking what each transaction reads and writes (read set and write set)
  2. Checking for conflicts of read/write set with all concurrent transactions
     - i.e. transactions that were committed between the time the transaction started, and when it validated.
     - Concurrent transactions identified using two timestamps:
       - start time
       - validation time
- Details in book

# Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions (timestamp must correspond to commit order)
  - $Q_1$, $Q_{11}$, $Q_{45}$, …
- 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.
  - E.g. $T_{34}$ vs. $T_{50}$
- **read**s never have to wait as an appropriate version is returned immediately.

# 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

# Snapshot Isolation

- Motivation: Queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- **Solution 1**: Give logical "snapshot" of database state to read only transactions, read-write transactions use normal locking
  - Multiversion 2-phase locking
  - Works well, but how does system know a transaction is read only?
- **Solution 2**: Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
  - Problem: variety of anomalies such as lost update can result
  - Partial solution: snapshot isolation level (next slide)
    - Proposed by Berenson et al, SIGMOD 1995
    - Variants implemented in many database systems
      - E.g. Oracle, PostgreSQL, SQL Server 2005

# Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

| T1 | T2 | T3 |
|---|---|---|
| W(Y := 1)<br>Commit | | |
| | Start<br>R(X) → 0<br>R(Y)→ 1 | |
| | | W(X:=2)<br>W(Z:=3)<br>Commit |
| | R(Z) → 0<br>R(Y) → 1<br>W(X:=3)<br>Commit-Req<br>Abort | |

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

# Benefits of Snapshot Isolation

- Reading is *never* blocked,
  - and also doesn't block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- Problems with snapshot isolation (SI)
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

# Snapshot Isolation

- E.g. of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution:  x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation:  x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

# Snapshot Isolation Anomalies

- SI breaks serializability when txns modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But does occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details

# SI In Oracle and PostgreSQL

- **Warning**: Snapshot isolation used when isolation level is set to serializable, by Oracle (and in PostgreSQL versions prior to 9.1)
  - PostgreSQL's implementation of SI in versions prior to 9.1 is described in DB Concepts book, Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
  - Oracle does not support true serializable execution

# How To Enforce Serializability with SI?

- for update clause in Oracle and PostgreSQL
  - E.g.
    - **select max** (orderno) **from** orders **for update**
    - read value into local variable maxorder
    - **insert into** orders (maxorder+1, …)
  - **for update** clause treats data which is read as if it is written
    - and thus causes a conflict between a writer, and a reader which uses the for update clause
    - and also between two readers who use the for update clause even if they don't actually update the data
  - In above example, **for update** ensures two orders will not get same order number
    - and thus ensures serializability

# 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**

# Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a $B^+$-tree are irrelevant so long as we land up in the correct leaf node.

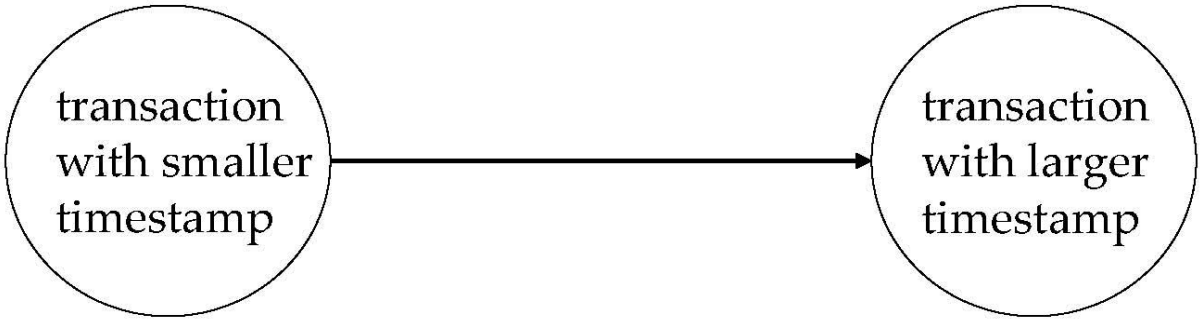
# Example Use of the Protocol

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

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| | | | | read ($X$) |
| | read ($Y$) | | | |
| read ($Y$) | | | | |
| | | write ($Y$) | | |
| | | write ($Z$) | | |
| | | | | read ($Z$) |
| | read ($Z$) | | | |
| | abort | | | |
| read ($X$) | | | | |
| | | | read ($W$) | |
| | | write ($W$) | | |
| | | abort | | |
| | | | | write ($Y$) |
| | | | | write ($Z$) |

# Correctness of Timestamp-Ordering Protocol

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




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