Chapter 14: Concurrency Control

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

Lock-Based Protocols

- Two lock modes can be set on each item:

  lock-X (exclusive lock)
  lock-S (shared lock)

- Locks can be set only by adhering to a compatibility matrix

	S	X
S	true	false
X	false	false

- This matrix is interpreted as follows:

  A transaction may set a lock on an item if this lock is compatible with locks already held on the item by other transactions

Lock-Based Protocols (Cont.)

- The compatibility matrix allows any number of transactions to 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.
- There are privileges associated with locks.
  - A transaction holding a **lock-X** may issue a write or a read access request on the item.
  - A transaction holding a **lock-S** may issue a read access request on the item.

The Two-Phase Locking Protocol

- 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

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. On the other hand, it shares with the first protocol the reliance on the application programmer to insert the various locking instructions

Lock-Conversion Based Protocol

A transaction T_i issues the standard read/write instruction.

- 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;
```

Lock-Conversion Based Protocol (Cont.)

- write(D) is processed as:

  if T_i 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_i has a **lock-S** on D
          then
            upgrade it to **lock-X**
          else
            grant it **lock-X**
        write(D)
      end;

- All locks are released after commit or abort

Graph-Based Protocols

- Impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, ..., d_h\}$ 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 only lock allowed in a *tree protocol* is **lock-X**.
  1. The first lock by T_i may be on any data item.
  2. Subsequently, a data item 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.

Tree-Locking Protocol

- 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
  - increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
- However,in the tree-locking protocol, a transaction may have to lock data items that it does not access.
  - increased locking overhead
  - additional waiting time
  - potential decrease in concurrency

Timestamp-Based Protocols

- *Timestamp-ordering* scheme – determine serializability order by selecting an ordering among transactions in advance.
- Each transaction is issued a timestamp when it enters the system. Timestamps are drawn from an increasing sequence of integers.
- The protocol manages concurrent execution so that it will be equivalent to a predetermined serial execution.
  - The serial execution is defined by the increasing order of timestamps.
  - If two transactions conflict in the history, then the one with the lower timestamp accessed the item first.

Timestamp-Based Protocols (Cont.)

- In order to assure such behavior, the protocol maintains for each data item Q two timestamp values:
  - **W-timestamp**(Q) [also denoted W-TS(Q)]

    The maximal timestamp of a transaction that wrote the item
  - **R-timestamp**(Q) [also denoted R-TS(Q)]

    The maximal timestamp of any transaction that read the item
- If a transaction wants to access an item in a way that will not create a prohibited conflict, the access is allowed and a synchronization value is updated.
- If the conflict is prohibited, the transaction is aborted.

The Timestamp-Ordering Protocol

- Let TS(T_i) be the timestamp of the executing transaction.
- The operation read(Q) is handled as:

```
if TS(T_i) ≥ W-TS(Q)
   then
      begin
         read(Q)
         R-TS(Q) ← max (TS(T_i), R-TS(Q))
      end
   else
      abort;
```

The Timestamp-Ordering Protocol (Cont.)

- The operation write(Q) is handled as:

if TS(T_i) $\geq$ R-TS(Q) **and** TS(T_i) $\geq$ W-TS(Q)
    then
        begin
            write(Q)
            W-TS(Q) $\leftarrow$ TS(T_i)
        end
else
    begin
        if TS(T_i) $\geq$ R-TS(Q) **and** TS(T_i) $<$ W-TS(Q)
            then ignore write(Q) request
            else abort
    end.

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- If $\mathrm{TS}(T_i) \geq \text{R-TS}(Q)$ and $\mathrm{TS}(T_i) < \text{W-TS}(Q)$ then it is safe to ignore the write because:
  - A newer value for Q is already written.
  - Any read request for T_j with $\mathrm{TS}(T_j) < \text{W-TS}(Q)$ will fail.
  - Therefore it is not possible to have a transaction that needs to read the value T_i is attempting to write and that does not abort.

Cascading Rollback

- Problem
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort
  - But then any transaction that has read a data item written by T_j must abort
  - This leads to a chain of rollbacks called *cascading rollback*
- Solution
  - 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

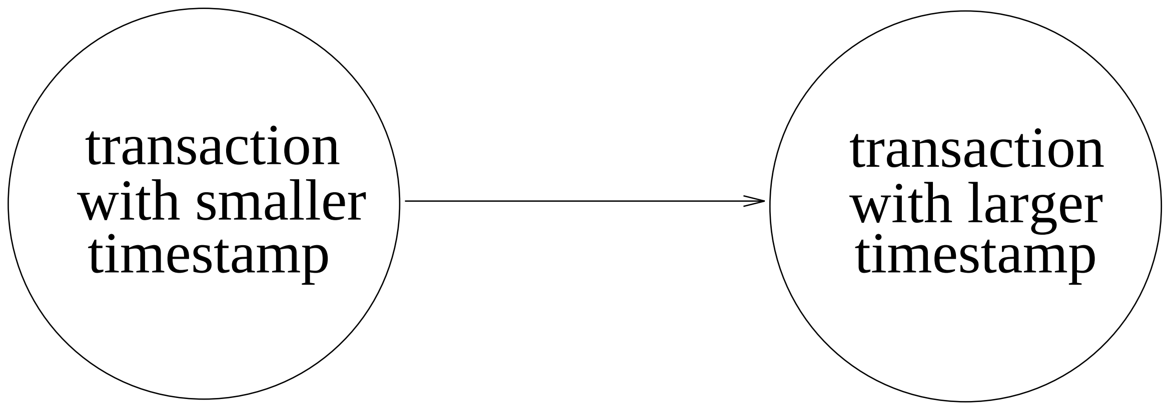
Example Use of the Protocol

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)				
			write(Z)	
			abort	
				write(X)
				write(Z)

Correctness of the Protocol

The protocol's correctness is guaranteed by the fact that all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

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.

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - **start**(T_i): the time when T_i started its execution
  - **validation**(T_i): the time when T_i entered its validation phase
  - **finish**(T_i): the time when T_i finished its write phase
- Serializability order is determined by the timestamp ordering technique using the value of the timestamp **validation**(T_i).
  - $\text{TS}(T_i) =$ **validation**(T_i).
  - if $\text{TS}(T_i) < \text{TS}(T_j)$, then any schedule must be equivalent to a serial schedule where T_i is before T_j.

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:

- **finish**$(T_i) <$ **start**(T_j) Ensures no overlapped execution.
- **start**$(T_j) <$ **finish**$(T_i) <$ **validation**(T_j) The set of data items written by T_i does not intersect with the set of data items read by T_j. Ensures that:
  - T_j does not read partially updated data generated by T_i.
  - the writes of T_i and T_j do not overlap.
  - the writes of T_j occur after the reads of T_i are all done.

Then the test succeeds. Otherwise, the validation test fails and T_j is aborted.

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.
- Locks are acquired in root-to-leaf order, while locks are released in leaf-to-root order.
- Granularity of data
  - Fine:

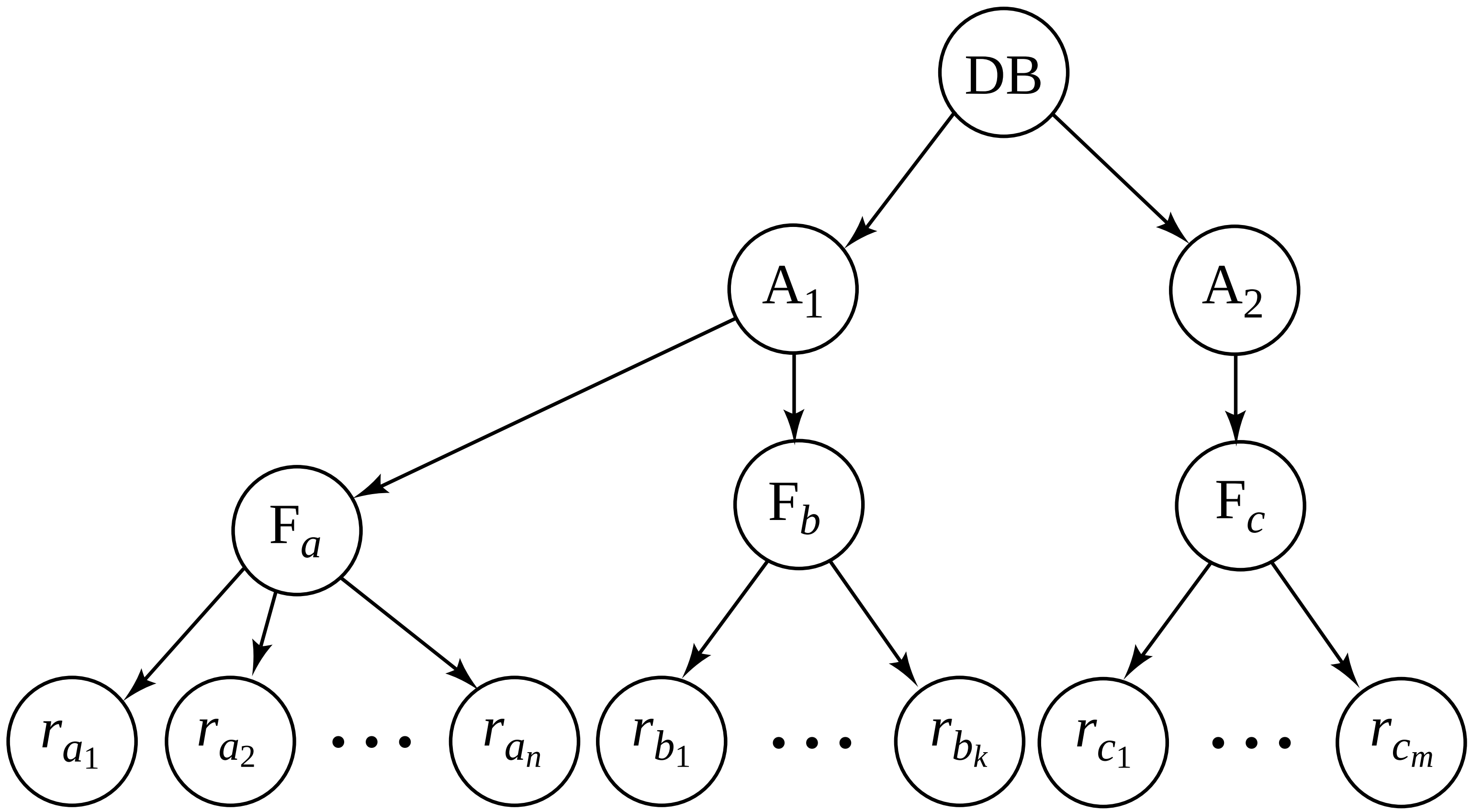
    high concurrency

    high overhead
  - Coarse:

    low overhead

    low concurrency

Example of Granularity Hierarchy



Intention Lock Modes

In addition to S and X lock modes, there are three additional lock modes

- *intention-shared* (IS): explicit locking at a lower level of the tree but only with shared locks.
- *intention-exclusive* (IX): 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.

Compatibility Matrix

The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

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.

Multiversion Schemes

- Maintain multiple versions of each data item
- Each write results in the creation of a new version
- Each version Q_k contains three data fields:
  - content – the value of version Q_k.
  - W-TS(Q_k) – timestamp of the transaction that created (wrote) version Q_k
  - R-TS(Q_k) – largest timestamp of a transaction that successfully read version Q_k

Multiversion Timestamp Ordering

- Let TS(T_i) denote the timestamp of transaction T_i
- Operation read(Q):
  - Let T_i read the version Q_k such that W-TS(Q_k) is the largest timestamp less than TS(T_i)
- Operation write(Q):
  - Let Q_k be the version Q_k such that W-TS(Q_k) is the largest timestamp less than TS(T_i)

```
if TS(T_i) ≥ R-TS(Q_k)
  then
    begin
      allow write
      (create new version)
    end
  else abort T_i
```

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions; *update transactions* hold all locks up to the end of the transaction.

- Timestamp is a counter (**ts_counter**) that is incremented during commit processing.

- Assign a timestamp to read-only transactions by reading the current value of **ts_counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

Multiversion Two-Phase Locking (Cont.)

- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts_counter** + 1
  - T_i increments **ts_counter** by 1
- Read-only transactions that start after T_i increments **ts_counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts_counter** will see the value before the updates by T_i.
- Read-only transactions never need to wait for locks.

Deadlock Handling

- Consider the following two transactions:

T_1:	write(X)	T_2:	write(Y)
	write(Y)		write(X)

- Schedule with deadlock

T_1	T_2
lock-X on X	
write(X)	
	lock-X on Y
	write(Y)
	wait for
	lock-X on X
wait for	
lock-X on Y	

Deadlock Prevention

- *Deadlock prevention* protocol ensures that the system will *never* enter a deadlock state.
- Require that each transaction locks all its data items before it begins execution.
- Impose partial ordering of all data items and require that a transaction can lock a data item only in the order specified by the partial order.

Deadlock Prevention (Cont.)

Two deadlock prevention schemes:

- **wait-die** scheme — nonpreemptive
  - older transaction must wait for younger one to release its data item
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - older transaction never waits for younger transaction
  - may be fewer rollbacks

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 the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm.

Deadlock Recovery

- When deadlock detected
  - Select “victim” – roll back those transactions that will incur the 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.
  - Abort victim
- Starvation – always picking the same transaction as a victim. Include the number of rollbacks in the cost factor.

Insert and Delete Operations

- 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 can lead to the *phantom phenomenon*, in which an insertion conflicts with a query even though the two transactions may access no tuple in common.
- The index-locking technique solves the phantom phenomenon problem by requiring locks on certain index buckets. These locks ensure that all conflicting transactions conflict on a "real" data item rather than on a phantom.

Concurrency in Index Structures

- Special features of index structures allow alternative transaction management techniques.
  - An index contains no unique data and can be reconstructed from the database itself.
  - Acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
- Lookup
- Insertion and deletion
- Split
- Coalescence