Concurrency Control– Timestamp Ordering

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.

Multiversion Timestamp Ordering

- 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)

- The multiversion timestamp scheme presented next ensures serializability.
- 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), and if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back. Otherwise, if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten, otherwise a new version of Q is created.

- 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.

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.

Multiversion Two-Phase Locking (Cont.)

- 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 ∞.
- 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.
- Only serializable schedules are produced.

Deadlock Handling

- Consider the following two transactions:

  T_1: write (X)
  write(Y)

  T_2: write(Y)
  write(X)

- Schedule with deadlock

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

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.

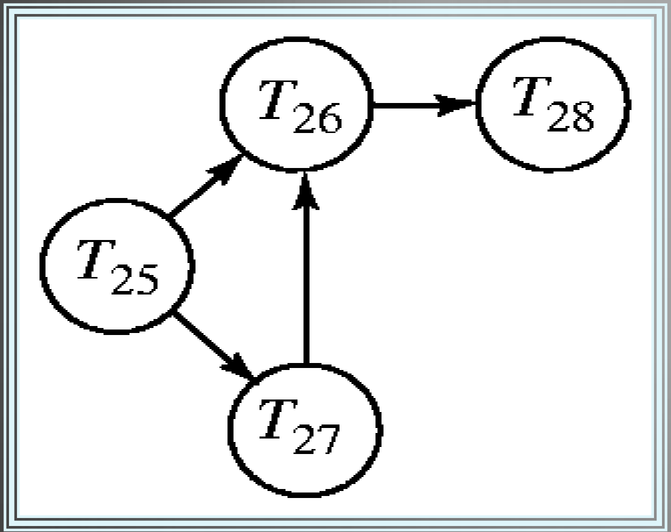
Deadlock prevention (Cont.)

- 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.

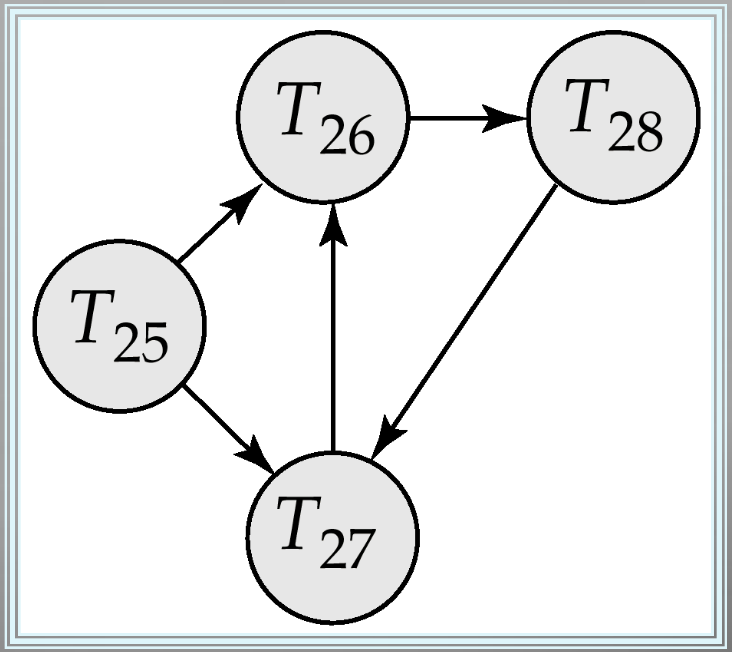
Deadlock Detection

- 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 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 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 all accounts in Perryridge) and a transaction that inserts a tuple in the relation (e.g., insert a new account at Perryridge) may conflict in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result: the scan transaction may not see the new account, yet may be serialized before the insert transaction.

Insert and Delete Operations (Cont.)

- 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

- Every relation must have at least one index. Access to a relation must be made only through one of the indices on the relation.
- A transaction T_i that performs a lookup must lock all the index buckets that it accesses, in S-mode.
- A transaction T_i may not insert a tuple t_i into a relation r without updating all indices to r.
- T_i must perform a lookup on every index to find all index buckets that could have possibly contained a pointer to tuple t_i, had it existed already, and obtain locks in X-mode on all these index buckets. T_i must also obtain locks in X-mode on all index buckets that it modifies.
- The rules of the two-phase locking protocol must be observed.

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

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 leads to low concurrency. Two-phase locking on an index may result in transactions executing practically one-at-a-time.
- 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.
- There are index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.

Concurrency in Index Structures (Cont.)

- Example of index concurrency protocol:
- Use **crabbing** instead of two-phase locking on the nodes of the B^+-tree, as follows. During search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node.
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks. Better protocols are available; see Section 16.9 for one such protocol, the B-link tree protocol