Chapter 22

Transaction Management (con't)

Concurrency Control with Index Structures

- Could treat each page of index as a data item and apply 2PL.
- However, as indexes will be frequently accessed, particularly higher levels, this may lead to high lock contention.
- Can make two observations about index traversal:
  - Search path starts from root and moves down to leaf nodes but search never moves back up tree. Thus, once a lower-level node has been accessed, higher-level nodes in that path will not be used again.

Concurrency Control with Index Structures

- When new index value (key and pointer) is being inserted into a leaf node, then if node is not full, insertion will not cause changes to higher-level nodes.

- Suggests only have to exclusively lock leaf node in such a case, and only exclusively lock higher-level nodes if node is full and has to be split.

Concurrency Control with Index Structures

- Thus, can derive following locking strategy:
  - For searches, obtain shared locks on nodes starting at root and proceeding downwards along required path. Release lock on node once lock has been obtained on the child node.
  - For insertions, conservative approach would be to obtain exclusive locks on all nodes as we descend tree to the leaf node to be modified.
  - For more optimistic approach, obtain shared locks on all nodes as we descend to leaf node to be modified, where obtain exclusive lock. If leaf node has to split, upgrade shared lock on parent to exclusive lock. If this node also has to split, continue to upgrade locks at next higher level.

Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to user, so DBMS should restart transaction(s).
- Three general techniques for handling deadlock:
  - Timeouts.
  - Deadlock prevention.
  - Deadlock detection and recovery.

Timeouts

- Transaction that requests lock will only wait for a system-defined period of time.
- If lock has not been granted within this period, lock request times out.
- In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

Deadlock Prevention

- DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- Could order transactions using transaction timestamps:
  - Wait-Die - only an older transaction can wait for younger one, otherwise transaction is aborted (*dies*) and restarted with same timestamp. (This is the first algorithm)

Deadlock Prevention

– **Wound-Wait - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (*wounded*).** (This is the second algorithm)
Restarted with the same timestamp

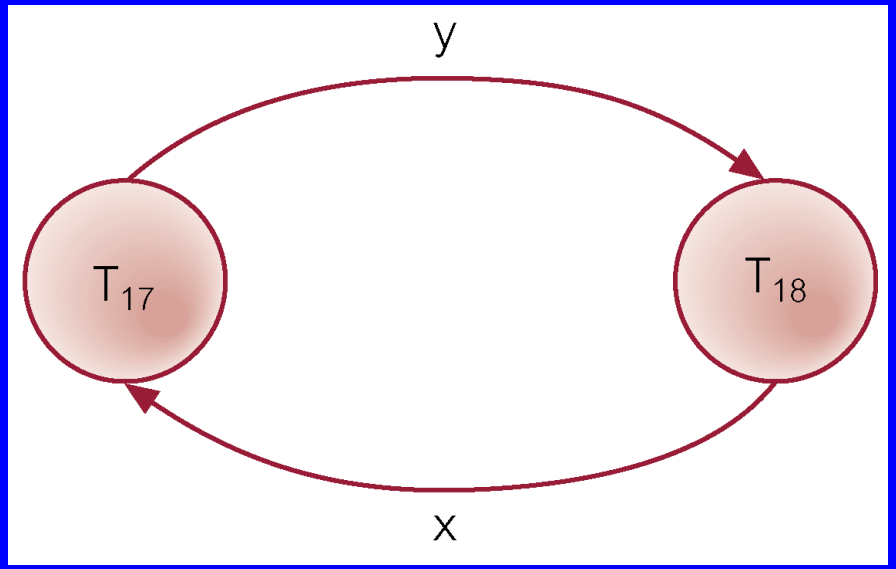
Thus for deadlock prevention, there are two different algorithms.

A variant of 2PL called conservative 2PL can be used to prevent deadlock. Obtain all locks; before start.

Deadlock Detection and Recovery

- DBMS allows deadlock to occur but recognizes it and breaks it.
- Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge T_i -> T_j, if T_i waiting to lock item locked by T_j.
- Deadlock exists if and only if WFG contains cycle.
- WFG is created at regular intervals.

Example - Wait-For-Graph (WFG)



Recovery from Deadlock Detection

- Several issues:
  - choice of deadlock victim;
  - how far to roll a transaction back;
  - avoiding starvation.

Timestamping

- Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting transaction.
- No locks so no deadlock.

Timestamping

Timestamp

A unique identifier created by DBMS that indicates relative starting time of a transaction.

- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.

Timestamping

- Read/write proceeds only if *last update on that data item* was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.
- Also timestamps for data items:
  - read-timestamp - timestamp of last transaction to read item;
  - write-timestamp - timestamp of last transaction to write item.

Timestamping - Read(x) (Transaction T issues a read(x))

- **Consider a transaction T with timestamp ts(T):**

ts(T) < write_timestamp(x)

  - **x already updated by younger (later) transaction.**
  - **Transaction must be aborted and restarted with a new timestamp.**

Otherwise, ts(T) >= write_timestamp(x), read can proceed.
read_timestamp(x) = max{ts(T), read_timestamp(x)}

Timestamping - Write(x) (Transaction T issues a write(x))

ts(T) < read_timestamp(x)

- **x already read by younger transaction.**
- **Roll back transaction and restart it using a later timestamp.** (new timestamp)

Else

//see next slide

Timestamping - Write(x)

ts(T) < write_timestamp(x)

- **x already written by younger transaction.**
- **Write can safely be ignored - *ignore obsolete write* rule.** (Also known as Thomas's write rule)

- **Otherwise, operation is accepted and executed.**

  That is,

  write(x)

  write_timestamp(x) = ts(T)

Example – Basic Timestamp Ordering

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read($\mathbf{bal_x}$)	read($\mathbf{bal_x}$)		
t_3	$\mathbf{bal_x} = \mathbf{bal_x} + 10$	$\mathbf{bal_x} = \mathbf{bal_x} + 10$		
t_4	write($\mathbf{bal_x}$)	write($\mathbf{bal_x}$)	begin_transaction	
t_5	read($\mathbf{bal_y}$)		read($\mathbf{bal_y}$)	
t_6	$\mathbf{bal_y} = \mathbf{bal_y} + 20$		$\mathbf{bal_y} = \mathbf{bal_y} + 20$	begin_transaction
t_7	read($\mathbf{bal_y}$)			read($\mathbf{bal_y}$)
t_8	write($\mathbf{bal_y}$)		write($\mathbf{bal_y}$)[+]	
t_9	$\mathbf{bal_y} = \mathbf{bal_y} + 30$			$\mathbf{bal_y} = \mathbf{bal_y} + 30$
t_{10}	write($\mathbf{bal_y}$)			write($\mathbf{bal_y}$)
t_{11}	$\mathbf{bal_z} = 100$			$\mathbf{bal_z} = 100$
t_{12}	write($\mathbf{bal_z}$)			write($\mathbf{bal_z}$)
t_{13}	$\mathbf{bal_z} = 50$	$\mathbf{bal_z} = 50$		commit
t_{14}	write($\mathbf{bal_z}$)	write($\mathbf{bal_z}$)[‡]	begin_transaction	
t_{15}	read($\mathbf{bal_y}$)	commit	read($\mathbf{bal_y}$)	
t_{16}	$\mathbf{bal_y} = \mathbf{bal_y} + 20$		$\mathbf{bal_y} = \mathbf{bal_y} + 20$	
t_{17}	write($\mathbf{bal_y}$)		write($\mathbf{bal_y}$)	
t_{18}			commit	

[+] At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.

[‡] At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Multiversion Timestamp Ordering

- Versioning of data can be used to increase concurrency.
- Basic timestamp ordering protocol assumes only one version of data item exists, and so only one transaction can access data item at a time.
- Can allow multiple transactions to read and write different versions of same data item, and ensure each transaction sees consistent set of versions for all data items it accesses.

Multiversion Timestamp Ordering

- In multiversion concurrency control, each write operation creates new version of data item while retaining old version.
- When transaction attempts to read data item, system selects one version that ensures serializability.
- Versions can be deleted once they are no longer required.

Multiversion Timestamp Ordering

- Versioning of data can be used to increase concurrency.
- Basic timestamp ordering protocol assumes only one version of data item exists, and so only one transaction can access data item at a time.
- Can allow multiple transactions to read and write different versions of same data item, and ensure each transaction sees consistent set of versions for all data items it accesses.

Multiversion Timestamp Ordering

- In multiversion concurrency control, each write operation creates new version of data item while retaining old version.
- When transaction attempts to read data item, system selects one version that ensures serializability.
- Versions can be deleted once they are no longer required.

Multiversion Timestamps

- For each data item x, the database holds n versions $x_1, x_2, \ldots, x_n$.
- For each version i, system stores:
  - the value of version x_i
  - read_timestamp(x_i) – largest timestamp of transactions that have read version x_i;
  - write_timestamp(x_i) – timestamp of transaction that created version x_i.

Multiversion Timestamp Rules

- Let ts(T) be timestamp of current transaction
- Transaction T issues a write(x):
  - if xj has largest write timestamp of x that is less than or equal to ts(T) and read_timestamp(x_j) > ts(T), transaction T restarted with new timestamp.
- Transaction T issues a read(x):
  - return the version x_j that has largest write timestamp of x that is less than or equal to ts(T).
  - set read_timestamp(x_j)= max(ts(T),read_timestamp(x_j))
  - read operation never fails

Multiversion Deletions

- Versions deleted if no longer required
- To determine if version is required:
  - find timestamp of oldest transaction in system
  - for any two versions xi and xj with write timestamps less than this oldest timestamp, delete the older version.

Optimistic Techniques

- Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.

Optimistic Techniques

- Three phases:
  - Read
  - Validation
  - Write

Optimistic Techniques - Read Phase

- Extends from start until immediately before commit.
- Transaction reads values from database and stores them in local variables. Updates are applied to a local copy of the data.

Optimistic Techniques - Validation Phase

- Follows the read phase.
- For read-only transaction, checks that data read are still current values. If no interference, transaction is committed, else aborted and restarted.
- For update transaction, checks transaction leaves database in a consistent state, with serializability maintained. (See text for details)

Optimistic Techniques - Write Phase

- Follows successful validation phase for update transactions.
- Updates made to local copy are applied to the database.

Optimistic Techniques – Validation Phase

- Each transaction T assigned timestamps:
  - start(T) at start of execution
  - validation(T) at start of validation phase
  - finish(T) at finish time
- To pass validation test, one of following true:
  - for all transaction S with earlier timestamps, finish(S) < start(T).
  - if T starts before earlier S finishes,
    - » data items written by S are not read by T
    - » start(T) < finish(S) < validation(T)

Exercises

22.29 Repeat exercise 22.25 for multiversion timestamping.

22.30 Consider the six transactions shown in the course text, Figure 22.24. There is a checkpoint at time t_c and a system crash at time t_f. Volatile storage is lost during the crash, but stable storage survives. For each of the six transactions, describe the action(s) that must be performed by the Recovery Manager, assuming the update policy of the DBMS is:

(a) Deferred Update

(b) Immediate Update

Exercises

22.31 Describe the nature of each of the following three types of transaction faults: dirty read, nonrepeatable read, phantom rows.

22.32 SQL Server has the following isolation levels: read uncommitted, read committed, repeatable read, serializable. For each isolation level, describe the type of locking used, including the lock granularity and lock duration.

Exercises

22.33 Give a detailed example of a query (or series of queries) that illustrates how *repeatable read* isolation in SQL Server does not prevent *phantom rows*.

22.34 Give a detailed example of a query (or series of queries) that illustrates how *read* committed isolation in SQL Server does not prevent *nonrepeatable reads*.