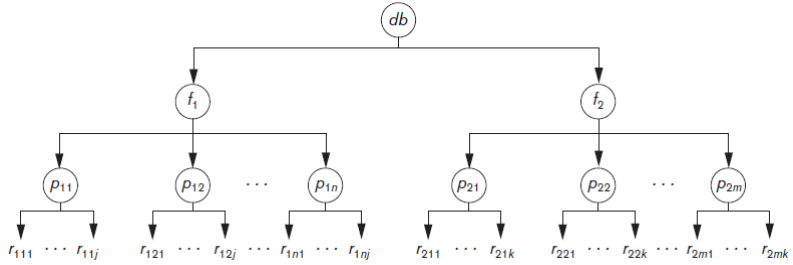
Database Systems

Transactions, Concurrency Control and Recovery (Continued - 2)

Jing Sun and Miao Qiao
The University of Auckland

Locks of Multiple Granularities

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No



Multiple granularity locking (MGL) protocol rules

- The lock compatibility must be adhere to.
- The root of the tree must be locked first.
- A node N can locked in S or IS mode by T only if the parent of N is locked in either IS or IX mode.
- A node N can be locked in X or IX or SIX mode if the parent of N is locked in either IX or SIX mode.
- A transaction T can lock a node only if T has not unlocked any node (enforce 2PL protocol).
- A transaction T can unlock a node N only if none of the descendent of N is currently locked by T.

Using Locks for Concurrency Control in Indexes

- Insertion: When new data item is inserted, it cannot be accessed until after the operation is completed
- Deletion operation on the existing data item: write lock must be obtained before deletion

Phantom revisited

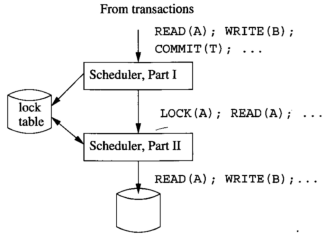
- Phantom can occur when a new record being inserted satisfies a condition that a set of records accessed by another transaction must satisfy
- The record that causes conflict is not recognized by concurrency control protocol

Locks

- Binary Locks
- Shared/Exclusive Locks
- Update Locks
- Increment Locks
- Locks with Multiple Granularity
- Locking scheduler
- Locking protocols (resolving starvation and deadlock)

Locking Scheduler

- The transactions themselves do not request locks or cannot be relied upon to do so. It is the job of the scheduler to insert lock actions into the stream of reads, writes and other actions that access data.
- Transactions do not release locks. Rather the scheduler releases the locks when the transaction manager tells it that the transaction will commit or abort.



Locking Scheduler

- Part I inserts appropriate lock actions for database access actions (choose lock mode)
- Part II executes the *sequence* of lock and database access actions passed by Part I; assume that the current operation o is on transaction T:
  - If o acquire a lock for T — look at the lock table, there are two cases:
    - if the lock cannot be grant, put the request to the lock table, notify Part I to delay T
    - grant the lock to T and update the lock table
  - If keys requested so far by T have been granted and o is a database access operations — execute
  - Commits/aborts a transaction T
    - Notify Part I. Release all locks hold by T. If any transactions are waiting for any of these locks, Part I notifies Part II
  - If notified by Part I that the lock on element X has been released, pick the next transaction T in the waiting list to lock X, notify Part I to resume T.

Locks

- Binary Locks
- Shared/Exclusive Locks
- Update Locks
- Increment Locks
- Locks with Multiple Granularity
- Locking scheduler
- Locking protocols (resolving starvation and deadlock)

Starvation and Deadlock

- Starvation
  : a transaction cannot proceed for an indefinite period of time while other transactions continue normally.
  -Solution: first-come-first-serve queue for each lock (waiting list)
- Deadlock: a set S of ≥ 2 transactions has a **deadlock** if each transaction T in S is *waiting* for some item that is locked by another transaction in S.

T_1	T_2
$sl_1(A); r_1(A);$	
	$sl_2(A); r_2(A);$
$xl_1(A)$ Denied	
	$xl_2(A)$ Denied

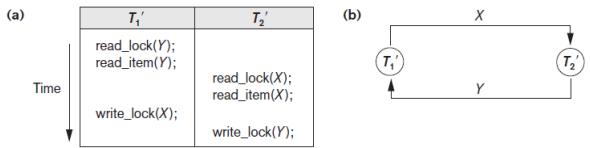
Deadlock Detection

Detect deadlock by timeout. If system waits for a transaction T longer than a predefined threshold, abort T and then restart T. Drawback: no perfect time limit.

Detect deadlock with wait-for graph. A set S of transactions has a deadlock if there is a cycle in the wait-for graph of S.

Wait-for graph

An edge from T_i to T_j if T_i is waiting for a lock currently hold by T_j.



- Victim selection: Abort one transaction T in case of a deadlock and restart T.

Deadlock Detection

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(C)$; $r_2(C)$;		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(D)$; $r_4(D)$;
5)		$l_2(A)$; **Denied**		
6)			$l_3(C)$; **Denied**	
7)				$l_4(A)$; **Denied**
8)	$l_1(B)$; **Denied**			

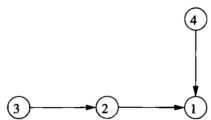


Figure: Wait-For Graph after Step (7)

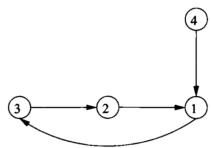


Figure: Wait-For Graph after Step (8)

Deadlock & Starvation Prevention Protocols

Transaction Timestamp

Assign, for each transaction T, a **unique** identifier $TS(T)$ related to the starting time of the transaction (a smaller $TS(T)$ means an older transaction).

There are two protocols to prevent both deadlock and starvation.

Let T be a transaction that is requesting a lock that is currently hold by transaction U.

- Wait-die: If $TS(T) < TS(U)$, let T wait for the lock; otherwise, abort T.
- Would-wait: If $TS(T) < TS(U)$, abort U; otherwise, let T to wait for the lock.

All transactions will eventually compete:

- Both wait-die and would-wait kills young transactions in a competition of resources.
- The aborted transaction will be resumed with the original timestamp, so young transactions that are killed will eventually be old.

Overview

- Serializability, Recoverability, Two phase locking
- Locks
  - Binary Locks
  - Shared/Exclusive Locks
  - Update Locks
  - Increment Locks
  - Locks with Multiple Granularity
  - Locking scheduler
  - Locking protocols (resolving starvation and deadlock)

Recovery

- Recovery Concepts
- Checkppoint
- Rollback
- Recovery Techniques Based on Deferred Update
- Recovery Techniques Based on Immediate Update

Recovery Concepts

Recovery process restores database to most recent consistent state before time of failure.

- Update strategies (Before-image BFIM & After-image AFIM):
  - In-place updating: Writes the buffer to the same original disk location, overwrites old values of any changed data items
  - Shadowing: Writes an updated buffer at a different disk location, to maintain multiple versions of data items
- Write-ahead logging (both redo and undo may be needed)
  - Ensure the BFIM and/or AFIM are recorded
  - Appropriate log entry flushed to disk
  - Necessary for (potential) UNDO operation
- UNDO-type log entries: [write_item, T, data_item, BFIM]
- REDO-type log entries: [write_item, T, data_item, AFIM]
- REDO/UNDO-type log entries: [write_item, T, data_item, BFIM, AFIM]

Recovery Concepts (cont.)

Recovery process restores database to the most recent consistent state before time of failure.

- Typical recovery strategies
  - Restore backed-up copy of database — Best in extensive damage
  - Identify any changes that may cause inconsistency — Best in noncatastrophic failure
  - Deferred update: Do not physically update the database until after transaction commits. Undo is not needed; redo may be needed
  - Immediate update: Database may be updated by some operations of a transaction before it reaches commit point
- Undo and redo operations required to be idempotent
  - Executing operations multiple times equivalent to executing just once
  - Entire recovery process should be idempotent

Recovery Concepts (cont.)

Recovery process restores database to most recent consistent state before time of failure.

- Steal/no-steal and force/no-force Specify rules that govern when a page from the database cache can be written to disk
  - No-steal approach: Cache buffer page updated by a transaction cannot be written to disk before the transaction commits
  - Steal approach: Recovery protocol allows writing an updated buffer before the transaction commits
- Force approach: All pages updated by a transaction are immediately written to disk before the transaction commits
- Typical database systems employ a steal/no-force strategy
  - Avoids need for very large buffer space
  - Reduces disk I/O operations for heavily updated pages

Recovery

- Recovery Concepts
- Checkppoint
- Rollback
- Recovery Techniques Based on Deferred Update
- Recovery Techniques Based on Immediate Update

Take a checkpoint

- Suspend execution of all transactions temporarily
- Force-write all main memory buffers that have been modified to disk
- Write a checkpoint record to the log, and force-write the log to the disk
- Resume executing transactions

Transaction Rollback

Transaction failure after update but before commit

- Necessary to roll back the transaction
- Old data values restored using undo-type log entries
- Cascading rollback
  - Caused by dirty read
  - Almost all recovery mechanisms were designed to avoid this

Recovery

- Recovery Concepts
- Checkppoint
- Rollback
- Recovery Techniques Based on Deferred Update
- Recovery Techniques Based on Immediate Update

Recovery Techniques Based on Deferred Update

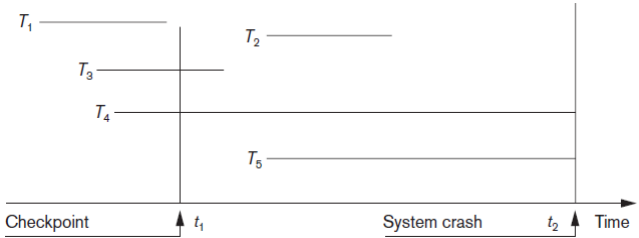
THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Deferred update protocol (no-steal): Transaction cannot change the database on disk until it reaches its commit point

- Postpone updates to the database on disk until the transaction completes successfully and reaches its commit point
- Transaction does not reach its commit point until all its REDO-type log entries are recorded in log and log buffer is force-written to disk
- All buffers changed by the transaction must be pinned until the transaction commits (no-steal policy)
- Redo-type log entries are needed
- Undo-type log entries not necessary
- Can only be used for short transactions and transactions that change few items: Buffer space an issue with longer transactions

Recovery Techniques Based on Deferred Update



Redo the transactions (T_2, T_3) that were committed after the latest checkpoint; No undo.

Recovery Techniques Based on Deferred Update

Another example:

[start_transaction, T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item, T_2, D, 25]

← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

Recovery Techniques Based on Immediate Update

THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND

SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Updated immediately protocol (steal): Transaction can change the database on disk before its commit point

- Not a requirement that every update be immediate
- UNDO-type log entries must be stored
- Recovery algorithms
  - UNDO (steal/force) (self-study)
  - UNDO/REDO (steal/no-force strategy)

Procedure RIU_M (UNDO/REDO with checkpoints).

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
2. Undo all the write_item operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the write_item operations of the *committed* transactions from the log, in the order in which they were written into the log, using the REDO procedure defined earlier.

Procedure UNDO (WRITE_OP). Undoing a write_item operation write_op consists of examining its log entry [write_item, T, X, old_value, new_value] and setting the value of item X in the database to old_value, which is the before image (BFIM). Undoing a number of write_item operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

Summary

- Locks
- Serializability, Recoverability, Two phase locking
- Deadlock and Starvation
- Recovery
  - Recovery Concepts
  - Checkppoint
  - Rollback
  - Recovery Techniques Based on Deferred Update
  - Recovery Techniques Based on Immediate Update

Thank you for your attention!

Any questions?