CS 2550 / Spring 2006
Principles of Database Systems

10 – Locking

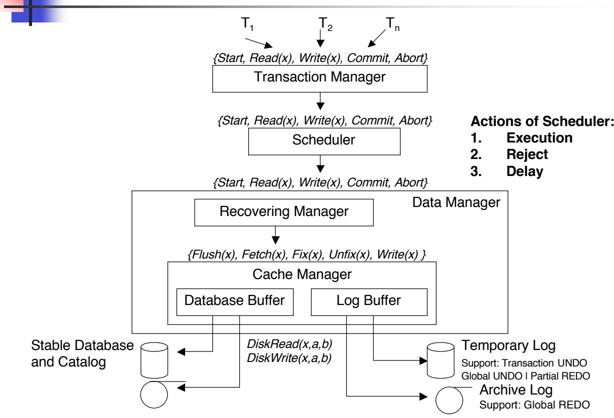
Alexandros Labrinidis
University of Pittsburgh

LOCKING

Locking

- Centralized DBMS Architecture
- Schedulers
  - Aggressive
  - Conservative
- Lock-based concurrency control
- Deadlocks
  - Detection
  - Prevention

Centralized DBMS



Aggressive Vs Conservative Schedulers

- A scheduler upon receiving an operation may
  - Execute the operation immediately, perhaps remembering the dependencies.
  - Delay the operation.
  - Reject the operation.
- A scheduler is ***aggressive*** if it avoids delaying operations thereby running the risk of rejecting them later.
  - Preferable if conflicts are rare.
- A scheduler is ***conservative*** if it deliberately delays operations thereby avoiding their (possible) subsequent rejection.
  - Attempts to anticipate future behavior of transactions.
  - Preferable if conflicts are likely.

Types of schedulers

- Almost all types of schedulers have both an aggressive and a conservative version.
- Extreme case of conservative scheduler is a serial scheduler.

Lock Based Concurrency Control

- Locking is the most common synchronization mechanism.
- A lock is associated with each data item in the database.
- A lock on *x* indicates that a transaction is performing an operation on *x*.
- Lock types
  - $rl_i(x)$: *x* is read lock by T_i (*shared* lock)
  - $wl_i(x)$: *x* is write lock by T_i (*exclusive* lock)

Lock Based Concurrency Control

- Locks *conflict* if they are associated with conflicting operations, i.e., operations that will form some dependency.

	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	No	Yes
$wl_i(x)$	Yes	Yes

- If transactions *Ti* and *Tj* request conflicting locks on data item *x* and Ti locks *x* first, then *Tj* should wait until *Ti* unlocks *x*.
  - $ru_i(x)$: remove the read lock from *x* set by T_i
  - $wu_i(x)$: remove the write lock from *x* set by T_i

Why Simple Mutual Exclusion Does Not Suffice

- Assume
  Database = { x, y }
  Initially: x = 0, y = 1
  Transactions
  T_1 : a = r(y); w(x, a) /* x ← y */
  T_2 : b = r(x); w(y, b) /* y ← x */

Why Simple Mutual Exclusion Does Not Suffice

- Consider the following schedule based on mutual exclusion

T_1	T_2	Comments
	rl(x)	granted
	b=r(x)	
	ru(x)	released
rl(y)		granted
a=r(y)		
ru(y)		released
wl(x)		granted
w(x,a)		
wu(x)		released
commit		
	wl(y)	granted
	w(y,b)	
	wu(y)	released
	commit	

Final database state: x = 1, y = 0. This history is not SR! Why not?

Basic Two Phase Locking (2PL)

- A scheduler following the 2PL protocol has two phases:
  - A *Growing* phase
  - Whenever it receives an operation $p_i(x)$ the scheduler obtains a p-lock on x ($pl_i(x)$) before executing p on the data.
  - A *Shrinking* phase
  - Once a scheduler has released a lock for a transaction,it cannot request any additional locks on any data item for this transaction.

Basic Two Phase Locking (2PL)

- Example:
  H_1: rl(x); a = r(x); wl(y); w(y, a); ru(x); wu(y);
  H_2 :rl(x); a = r(x); ru(x); wl(y); w(y, a); wu(y);
- **Theorem**: Every 2PL history H is serializable.
- Note: Eswaran, Gray, Lorie, Traiger - ``The Notions of Consistency and Predicate Locks in a Database System", *CACM*, vol. 19, no. 11 Nov. 1976, pp. 624-633

Two Phase Locking: Serializability

- Lock point
  - The point in the schedule where the transaction has obtained its final lock
  - = the end of the growing phase in 2PL
- Serializable ordering:
  - Order transactions according to their lock points
- 2PL does not guarantee freedom from deadlocks

Issues Related To Locking

- *Deadlock*
  Two or more transactions are blocked indefinitely because each holds locks on data items upon which the others are trying to perform operations, i.e., obtain locks.
- *Livelock*
  Livelock occurs when a transaction is aborted and restarted repeatedly (Cyclic Restart), e.g., because its priority is too low.
  Differs from deadlock in that it allows a transaction to execute but not to completion.
- *Starvation*
  Starvation occurs when a transaction is never allowed to run, e.g.,because there is always a transaction with a higher priority.

Conservative (Static) 2PL

- A transaction T declares *in advance* *all* data items that it might read or write.
- A transaction is executed when the scheduler obtains all the locks on the declared data items.
  - No deadlocks since there are no lock conflicts while transactions are executing.
  - Low message passing overhead between transactions and the scheduler.

Conservative (Static) 2PL

But:

- Transactions are blocked for conflicts that may never arise in an actual execution.
- Starvation is possible.
- Transactions may need to lock more data items than really need to access.
- Requires pre-processing.

Aggressive (Dynamic) 2PL

- A transaction requests locks just before it operates on a data item.
- If a transaction holds a read lock on an item x and later on it decides to update x, it can (try to) convert its read lock on x to a write lock. (This is called *lock conversion*.)
- A transaction cannot convert a write lock to a read lock. This is equivalent to releasing the write lock and obtaining a read lock.
  - Transactions only lock the data items that they really need.

Aggressive (Dynamic) 2PL

But:

- More message passing between transactions and scheduler.
- Transactions may deadlock.
- Cannot reorder operations later and hence may have to abort them.

Strict 2PL

- It is a form of aggressive (dynamic) 2PL
  - transactions request locks just before they operate on a data item.
- The growing phase ends at *commit time*.
  - no locks can be released until commit or abort time.
  - no overwriting of dirty data.
  - no overwriting of data read by active transactions.
  - no reading of dirty data.
- Is it easy to implement strict 2PL?

Deadlocks

- A *deadlock* occurs when two or more transactions are blocked indefinitely.
  - each holds locks on data items on which the other transaction(s) attempt to place a conflicting lock.
- Necessary conditions for deadlock situations.
  - mutual exclusion
  - hold and wait
  - no preemption
  - circular wait.

Deadlocks

- Examples:

(I)	2 Items	
T_1	T_2	Comments
rl(x)		granted
	rl(y)	granted
wl(y)		T_1 blocked
	wl(x)	T_2 blocked
		(deadlock)

(II)	1 Item	
T_1	T_2	Comments
rl(x)		granted
	rl(x)	granted
wl(x)		T_1 blocked
	wl(x)	T_2 blocked
		(deadlock)

- Example II involves lock conversion
- The scheduler *restarts* any transaction aborted due to deadlock.

Deadlock Detection: Timeout

- The scheduler checks periodically if a transaction has been blocked for too long.
  - In such a case, the scheduler *assumes* that the transaction is deadlocked and it aborts the transaction.
- This method may incorrectly diagnose a situation to be a deadlock.
  - The scheduler may make a mistake and abort a transaction that waits for another transaction that is taking a long time to finish.
- The *correctness* of the schedule is not affected if the scheduler makes a wrong guess.

Deadlock Detection: Timeout

- Fine tuning of the timeout period:
  Long timeout: fewer mistakes by the scheduler, but a deadlock may exist unnoticed for long periods causing long delays.
  Short timeout: quick deadlock detection, but more mistakes are possible thus aborting transactions not involved in a deadlock.
- Advantage: very simple algorithm.
- Tandem used deadlock detection based on timeout.

Deadlock Detection: Wait-for Graphs

- The scheduler maintains a *Waits-for Graph* **(WFG)** in which:
  - nodes are transactions T_i, T_j, ...
  - for edge $T_i \rightarrow T_j$ means that T_i is waiting for T_j to unlock a data item.
- The WFG is *acyclic* iff there is *no deadlock*.
- What is the relation of WFG and SG ?

Deadlock Detection: Wait-for Graphs

- Example

start T_1		**add T_1 in WFG**
start T_2		**add T_2 in WFG**
$rl_1(x)$	**yes**	
$wl_2(x)$	**no**	**$T_2 \rightarrow T_1$**
start T_3		**add T_3 in WFG**
$wl_3(x)$	**no**	**$T_3 \rightarrow T_1$**
$ru_1(x)$	**accept T_2's request**	**drop $T_2 \rightarrow T_1$**
		drop $T_3 \rightarrow T_1$
		add $T_3 \rightarrow T_2$
$wu_2(x)$	**accept T_3's request**	**drop $T_3 \rightarrow T_2$**
commit T_1		**drop T_1 from WFG**
commit T_2		**drop T_2 from WFG**
commit T_3		**drop T_3 from WFG**

- INGRES, POSTGRES, DB2 use deadlock detection based on WFG.

Victim Selection

- The scheduler runs a cycle detection algorithm in WFG every time period *t* and for every detected cycle it selects the ``best" victim to abort to break the cycle.
- What constitutes the ``best" victim ?
- Factors to consider:
  - The cost of aborting a transaction
    - all updates must be undone.
  - For how long a transaction was running.
  - How long it will take a transaction to finish.

Victim Selection

- How many deadlocks will be resolved if a particular transaction is aborted (i.e., is the transaction in more than one cycle?).
- How many times this transaction was already aborted due to deadlocks (see starvation).

In practice, deadlock cycles have a very small number of transactions and arbitrary victim selection does not affect performance.

Deadlock Prevention

- Simplest Methods:
- Predeclaration of readset and writeset. *
  - Conservative 2PL
- Whenever a T_i has to be blocked because of a conflicting lock request, the scheduler checks immediately for deadlock involving T_i.
  - a transaction may be restarted repeatedly.
  - high concurrency control overhead for each read or write lock request.

* This is known as *Deadlock Avoidance Method* in OS

Wait-Die

- Each transaction is assigned a *timestamp*, **ts(T_i)**.
- Timestamps are *totally* ordered and obtained using the
  - system clock, or
  - a counter.
- Suppose T_i can not obtain a lock on a data item because T_j holds a conflicting lock on this data item.

  If ts(T_i) < ts(T_j)
  then T_i waits
  else T_i aborts

  T_i waits if it is *older* than T_j.
  T_i aborts if it is *younger* than T_j.
- An aborted transaction restarts with its original timestamp. Why ?

Wound-Wait

- Suppose T_i requests a lock on x and T_j holds a conflicting lock on x.

  If ts(T_i) < ts(T_j)
  then T_j aborts
  else T_i waits

  T_i wounds T_j if T_i is *older* than T_j.
  T_i waits for T_j if T_i is *younger* than T_j.
- An aborted transaction restarts with its original timestamp.

Wait-Die Vs Wound-Wait

- When a transaction encounters a younger transaction:

  Wait-Die it never aborts.

  Wound-Wait it never aborts.

  =>both methods avoid starvation.
- An older transaction conflicts with a younger transaction:

  Wait-Die it waits for the younger transaction.

  Wound-Wait it wounds every transaction it encounters.

  => old transactions push their way.

Wait-Die Vs Wound-Wait

- When a younger transaction T_i restarts, T_i may encounter its older friend T_j that caused T_i to abort.
  Wait-Die T_i has to abort again.
  Wound-Wait T_i has to wait for T_j, not to abort.
- Once a transaction has locked all items it wants to access (i.e., reaches the end of the growing phase)
  Wait-Die it will never abort.
  Wound-Wait it might abort because of an older transaction.

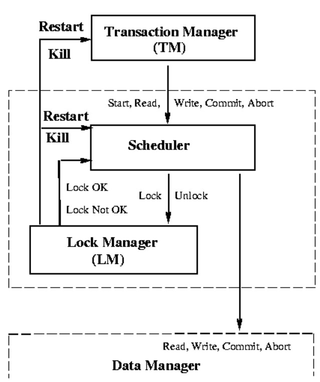
Lock Table

- Each entry in the lock table keeps information about a locked data item.

Datum	Locks Granted	Locks Requested (Blocked Transactions)
x	$<T_1, rl>, < T_2, rl>$	$<T_3, wl>, < T_5, wl>$
y	$< T_3, wl>,$	$<T_4, rl>, < T_6, wl>$
...	...	...

- Lock/Unlock operations in lock table must be very fast.
- Lock/Unlock operations are serialized.
- Abort operations must be fast.
- How do you implement the lock table ?
- Rescheduling blocked and deadlocked transactions must be fast.

Implementation of a 2PL Scheduler



Phantoms

- So far, we have considered static databases.
- What about dynamic databases that support insert and delete operations ?
- Example: consider the following EMP database

ID	NAME	PHONE	(TUPLE)
1	Mike	256-6116	a1
9	Susan	782-9392	a2
5	Alex	661-0021	a3

- Transactions T1 , T2 :

  If there is no tuple whose ID = 4 in EMP, then
  insert (4, Alex, 662-8210) in EMP;

Phantoms

- Here is a 2PL interleaved execution:

  T1 : read a1, a2, a3; no tuple has ID = 4;

  T2 : read a1, a2, a3; no tuple ID = 4;

  T1 : insert tuple a4: (4, Alex, 662-8210);

  T2 : insert tuple a5: (4, Alex, 662-8210);

How Do We Deal With Phantoms

- 2PL can deal with phantoms.
- In the previous example, T1 had to lock tuple a4 which, however, didn't exist at that time.
  - How can transactions lock phantoms ?

How Do We Deal With Phantoms

- How did T1 know that it had to read a_1, a_2, a_3?
  - It read the EOF marker.
  - It read a counter containing the number of records
  - It followed pointers.

⇒ It read some control information.

⇒ Need to lock *both* data and control information.
  - Control information such as EOF may become *hot* spots
  - index locking
  - predicate locking
  - weak locks (operations must be implemented atomically)