PURDUE UNIVERSITY

CS54200: Distributed Database Systems

Two-Phase Locking
January 23, 2009
Prof. Chris Clifton

Indiana Center for Database Systems

How do we ensure Serializability

- This is the task of the scheduler.
- There are two basic techniques:
  - Locking
  - Time-Stamp Ordering
- Locking enforces serializability by ensuring that no two txns access conflicting objects in an "incorrect" order.
- Time-Stamp ordering assigns a fixed order for every pair of txns and ensures that conflicting accesses are made in that order.

12

Two Phase Locking

- Basic 2PL
- Each object has associated with it a *lock*.
- An appropriate lock must be acquired before a txn accesses the object.
- There are *2 basic types of locks*: shared (read) and exclusive (write).
- Two locks, $pl_i[x]$ and $ql_j[y]$, conflict if $x=y$ and $i<>j$; and p and q are conflicting operations.
- 2PL is defined by 3 rules

13

2 Phase Locking

1. To grant a lock, the scheduler *checks if a conflicting lock* has already been assigned, if so, *delay*, otherwise *set lock and grant* it.
2. A lock cannot be released at least until the DM acknowledges that the operation has been performed.
3. *Once the scheduler releases a lock* for a txn, it may *not subsequently acquire any more locks* (on any item) for that txn.

14

Example

- $T_1 = r_1[x]\ w_1[y]\ c_1$
- $T_2 = w_2[x]\ w_2[y]\ c_2$
- $rl_1[x]\ r_1[x]\ ru_1[x]\ wl_2[x]\ w_2[x]\ wl_2[y]\ w_2[y]\ wu_2[x]\ wu_2[y]\ c_2\ wl_1[y]\ w_1[y]\ wu_1[y]\ c_1$
- This is not SR ($r_1[x] < w_2[x]$ and $w_2[y] < w_1[y]$).
- This is <u>prevented by rule 3</u>.

15

Deadlocks

- 2PL suffers from the problem of deadlocks.
- $rl_1[x]\ r_1[x]\ wl_2[y]\ w_2[y]$ followed by TM receiving $w_2[x]$ and $w_1[y]$.
- Also due to *lock conversion*: changing a read lock to a write lock – can't release the lock.
  - Why?
  - What if two txns try to convert at the same time?

16

2PL ensures Serializability

- Add the lock and unlock operations to the notion of histories.
- Proposition 1: Let H be a history produced by a 2PL scheduler. If $o_i[x]$ is in C(H), then $ol_i[x]$ and $ou_i[x]$ are in C(H), and $ol_i[x] < o_i[x] < ou_i[x]$.
- Proposition 2: Let H be a history produced by a 2PL scheduler. If $p_i[x]$ and $q_j[x]$ (i<>j) are conflicting operation in C(H), then either $pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$.

17

Correctness of 2PL

- Proposition 3: Let H be a complete history produced by a 2PL scheduler. If $p_i[x]$ and $q_i[y]$ are in C(H), then $pl_i[x] < qu_i[y]$.
- Lemma 4: Let H be a 2PL history, and suppose $T_i \rightarrow T_j$ is in SG(H). Then, for some data item x, and some conflicting operations $p_i[x]$ and $q_j[x]$ in H, $pu_i[x] < ql_j[x]$
- Proof: trivial.

18

Correctness of 2PL

- Lemma 5: Let H be a 2PL history, and let $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ be a path in $SG(H)$, where $n > 1$. Then, for some data items x and y, and some operations $p_1[x]$ and $q_n[y]$ in H, $pu_1[x] < ql_n[y]$.
- Proof: by induction on n.
- Base Case, $n=2$. Follows from Lemma 4.
- Induction Step. Assume true for $n=k$ for $k>=2$. By the induction hypothesis, there exist data items x and z, and operations $p_l[x]$ and $o_k[z]$ in H, such that $pu_1[x] < ol_k[z]$.
- By $T_k \rightarrow T_{k+1}$ and Lemma 4, there exists y and conflicting operations $o'_k[y]$ and $q_{k+1}[y]$ in H, such that $o'u_k[y] < ql_{k+1}[y]$.

19

Correctness of 2PL

- By proposition 3, $ol_k[z] < o'u_k[y]$. Thus by transitivity, $pu_1[x] < ql_{k+1}[y]$.
- Theorem: Every 2PL history H is serializable.
- Proof: Suppose, by contradiction, that SG(H) contains a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$, where $n > 1$.
- By Lemma 5, for some data items x and y, and some operations $p_1[x]$ and $q_1[y]$ in H, $pu_1[x] < ql_1[y]$.
- This contradicts Prop 3. Thus SG(H) is acyclic.

20

Deadlocks

- 2PL suffers from deadlocks
- Timeouts
- Waits-for-graph
  - nodes are transactions
  - add edge $T_i \rightarrow T_j$ whenever T_i waits for a lock held by T_j
  - remove an edge when last blocking lock is released
  - a cycle implies a deadlock
  - all cycles need to be broken by choosing a victim txn

21

Types of Schedulers

- Schedulers can delay, reject, or immediately schedule the operations.
- Aggressive schedulers try to avoid delaying operations -- may have to abort later
- Conservative schedulers try to avoid aborting by delaying and reordering operations
- Trade-off: depends upon degree of conflict between transactions.
- Conservative schedulers try to anticipate future access of transactions.

22

Conservative 2PL

- 2PL aborts txns only because of deadlocks.
- Conservative 2PL eliminates deadlocks.
- Each txn predeclares all its operations.
- The scheduler sets all locks of a txn in one step, if it cannot (because there is some conflicting lock), the txn is put in a queue.
- When a lock is released the scheduler checks to see which txns can now acquire all their locks.
- Predeclaring *may be difficult or even impossible*.

23

Strict 2PL

- A transaction's locks are all released together after the DM acknowledges the processing of the transaction's commit or abort.
- Why?
  - To ensure a strict execution
  - Earliest time at which the scheduler is certain that no more locks will be required by the transaction. Why?

24

Timing of Lock Release

- Let H be a history produced by a strict 2PL scheduler.
- Suppose $w_i[x] < o_j[x]$.
- By rule 1 of 2PL we must have
  1. $wl_i[x] < w_i[x] < wu_i[x]$, and
  2. $ol_j[x] < o_j[x] < ou_j[x]$
- Because $wl_i[x]$ and $ol_j[x]$ conflict we must have either $wu_i[x] < ol_j[x]$ or $ou_j[x] < wl_i[x]$ (Prop. 2)

25

Timing of Lock Release

- $ou_j[x] < wl_i[x]$ with above two is impossible, so we must have: 3. $wu_i[x] < ol_j[x]$.
- Since H is produced by a strict 2PL scheduler, we must have: 4. Either $a_i < wu_i[x]$ or $c_i < wu_i[x]$
- From 2, 3, & 4: either $a_i < o_j[x]$ or $c_i < o_j[x]$, proving that H is strict.
- Note that read locks can be released upon termination.

26

Phantoms

- Consider a banking application with two files: *Accounts (number, location, balance);* and *Assets (branch, total).*
- Two txns: T_1 – checks total for some location. T_2– add an account and update total.
- Consider: Accounts at Lafayette;
  - R_1(Accounts[222],Accounts[213],Accounts[444])
  - Insert$_2$(Accounts[111],Lafayette,100)
  - R_2(Assets[Lafayette]) (reads old value)
  - W_2(Assets[Lafayette])
  - R_1(Assets[Lafayette]) – INCONSISTENT!

27

Phantoms

- This is clearly not an SR execution, however according to 2PL this is acceptable!
- The problem is that T_1 doesn't just touch the accounts 222, 213, 444 – but rather ALL accounts in the Accounts table.
- Account 111 appears in between T_1 – like a phantom.
- This is a problem of dynamic databases – where data items are created and deleted.
- How can 2PL handle this?
- The two txns interfere with each other because they both access control information.
- We require 2PL to appropriately lock such information.
- We can improve by performing index locking.

28

Multigranularity Locking

- Granularity of data is unimportant for correctness of locking, but not for performance.
- Finer granularity allows greater concurrency.
- Coarser granularity reduces locking overhead.
- We can increase flexibility by allowing multiple granularity locks.
- This is complex in general, but if we follow a simple hierarchical structure for the locks.
  - E.g. Table → Page → Record

29

Multigranularity Locking

- E.g. long transactions could lock a page, whereas short transactions could lock records.
- Must ensure that conflicts are appropriately captured: (e.g. a page cannot be read locked if any of its records is write locked)
- How can such tests be efficiently made (e.g. by not having the transaction check for locks on every record within a page)?

30

Multigranularity Locking

- Represent the relationships as *a lock type graph*.
  - Database → Area → File → Record
- A set of data items that follow this structure is called a *lock instance graph* (assume that it is a tree).
- A lock on a coarse granule *x* *explicitly* locks *x*, and *implicitly* locks all of *x*'s proper descendants.
- Each type of lock also has an associated *intention lock* type. Before locking *x*, the scheduler ensures that there are no locks on its ancestors that implicitly conflict.
- This is done by setting intention locks on the ancestors.
- Compatibility of locks and intention locks is important.

31

Example

- Before *rl[x]*, set *ir* locks on *x*'s *database*, area, and *file* ancestors (in that order).
- *irl[y]* and *wl[y]* conflict for any object *y*.
- Thus we are sure that if we get *rl[x],* then no txn can have a write lock on a parent of *x*.
- A special lock type: *riw* is defined to represent txns that read a higher granularity and also may intend to write some lower granularity objects.

32

Compatibility Matrix

	R	W	IR	IW	RIW
R	Y	N	Y	N	N
W	N	N	N	N	N
IR	Y	N	Y	Y	Y
IW	N	N	Y	Y	N
RIW	N	N	Y	N	N

33

MGL Rules

For a given lock instance graph G, that is a tree, the scheduler follows these rules:

1. If *x* is not the root of G, then to set $rl_i[x]$ or $irl_i[x]$, T_i must have an *ir* or *iw* lock on *x*'s parent.
2. If *x* is not the root of G, then to set $wl_i[x]$ or $iwl_i[x]$, T_i must have an *iw* lock on *x*'s parent.
3. To read (or write) *x*, T_i must own an *r* or *w* (or *w*) lock on some ancestor of *x*. A lock on *x* itself is an explicit lock for *x*; a lock on a proper ancestor of *x* is an implicit lock for *x*
4. A txn may not release an intention lock on a data item *x*, if it is currently holding a lock on any child of *x*.

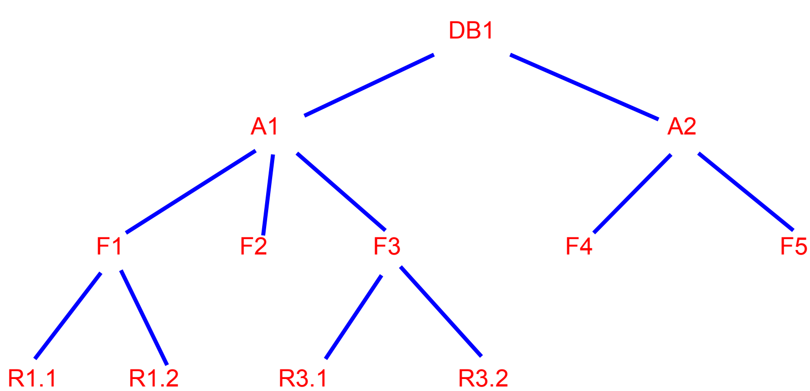
34

MGL Rules

- Rule 1, 2 ensure intention locks are acquired
- Rule 3 implies that by locking *x*, all its descendants are also locked. No need to set these locks explicitly.
- Rule 4 ensure that no lock is held without holding an intention lock on all ancestors too.

35

Example



36

Example

- T_1 wants to set $rl_1[F3]$.
- It must first set $irl_1[DB1]$, then $irl_1[A1]$, and finally $rl_1[F3]$.
- If T_2 wants to set $wl_2[R3.2]$.
- It must set $iwl_2[DB1]$, $iwl_2[A1]$, but can't get $iwl_2[F3]$.
- After T_1 releases $rl_1[F3]$, T_2 can set $iwl_2[F3]$ and $wl_2[R3.2]$.
- If T_3 tries to set $rl_3[A1]$.
- It must set $irl_3[DB1]$, but it can't get $rl_3[A1]$ until T_2 releases $iwl_2[A1]$.

37

MGL

- Correctness: The 5 rules ensure that if a txn owns an explicit or implicit lock on an object, no other txn owns a conflicting explicit or implicit lock.
- At what granularity should a txn lock? Difficult to determine in general.
- Lock Escalation: adjust the granularity dynamically. *Can lead to deadlocks*.
- Other than trees: Allow rooted dags for indexes. Modify to obtain appropriate locks on ALL parents.

38

Distributed CC

- How to handle the distributed database case?
- Data items are not located at a central site.
- For now, assume NO REPLICATION.
- Can centralize the scheduler (lock manager).
- Each site has a TM and a scheduler. This scheduler is responsible for controlling access to all items stored at this site.

40

Distributed CC

- Each TM submits operations to the appropriate scheduler. Commit and Abort operations are sent to every site where the txn operated.
- *How do we ensure that the global execution is serializable* based upon the processing of local schedulers?

41

Distributed 2PL

- 2PL easily extends to the distributed case.
- Each scheduler follows the same rules as before – if a lock can be acquired, process the operation.
- *No communication* needed – good.
- *Tricky issue: releasing locks!*
- In general would require communication.
- However, if STRICT 2PL is followed everywhere, then no communication is needed.
- Distributed, Strict 2PL is correct (*assuming that abort and commit operations are carried out atomically* – important issue that we will address later).

42

Distributed Deadlocks

- As with centralized 2PL, *distributed 2PL suffers from deadlocks*. Moreover, these can be distributed deadlocks! E.g. if *x* and *y* are at different sites.
- Solutions:
  - Timeouts
  - Deadlock Detection
  - Deadlock Prevention
- Timeouts are easy – local decision, but may be overreacting.

43

Deadlock Detection

- Again, we can use the Waits-for-Graph idea; however, we need to have a global WFG.
- Each site maintains its local WFG, and we periodically compute the global WFG.
- The global graph can be computed at
  - *Centralized* site – bottleneck
  - *Hierarchical* fashion
  - *Distributed* – add edges due to waits for non local objects.
- Phantom deadlocks – those not really present but show up due to the asynchronous nature of detection. *Can only occur due to spontaneous abortions!*

44

Deadlock Prevention

- Timestamp based – each txn is assigned a unique timestamp, in ascending order.
- When a txn T_i cannot obtain a lock because it is held by T_j, then:
  - *Wait-Die*: if $ts(T_i) < ts(T_j)$ then T_i waits else abort T_i.
  - *Wound-Wait*: if $ts(T_i) < ts(T_j)$ then abort T_j else T_i waits.
  - Aborted txn is automatically restarted.
- Upon Restart – use the SAME timestamp.
- *Both give preference to the older txn*. Note that there is no starvation in either scheme.

45