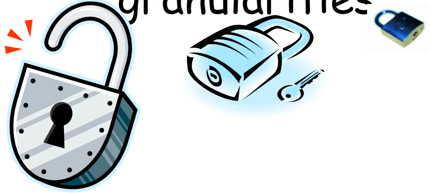
CS 2550 / Spring 2006
Principles of Database Systems

11 – Timestamp Locking and Multiversion CC

Alexandros Labrinidis
University of Pittsburgh

LOCKING
under multiple granularities



Granularity of Locks

- Locking granularity is the size of the data item being locked.

  Example:
  - page
  - file
  - tuple (record)
  - field in a tuple
  - a particular field of all tuples (column)
- The granularity of locks is unimportant w.r.t. *correctness*, but it is important w.r.t. *performance*.

Granularity And Atomicity Of Reads And Writes

Assume that
- Read/Write is done by blocks
- Locking granularity is record, and
- Block b contains three records r1, r2, r3.

Granularity And Atomicity Of Reads And Writes

Database	T_1	T_2
b: $r_1\, r_2\, r_3$		
b: 0 0 0	$rl(r_1)$	
	b′= r(b) [b′:000]	
	$r_1 \leftarrow 8$ [b′:800]	
		$rl(r_2)$
	$wl(r_1)$	b′= r(b) [b′:000]
b: 8 0 0	w(b, b′)	
		$r_2 \leftarrow 6$ [b′:060]
		$wl(r_2)$
b: 0 6 0		
		w(b, b′)

Granularity And Atomicity Of Reads And Writes

- The granularity of locking must be at least as coarse as the granularity of the atomic read and write.

  OR
  - Place another lock on block while read or write is performed; release it when operation completes (not according to 2PL rule).
  - Use Multi-Granularity Locking.

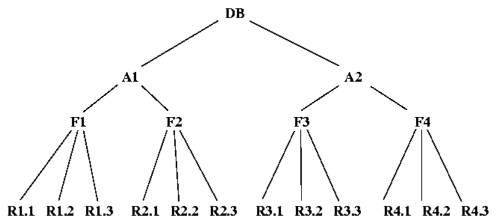
Multi-Granularity Locking

- Define a hierarchy of granules where lower level granules are finer:

Database
|
Areas
|
Files
|
Records

Multi-Granularity Locking

- An instance of this hierarchy might be:



Explicit, Implicit, And Intention Locks

- A lock on a granule *x*, **explicitly** locks *x*, and **implicitly** all its descendants in the same mode.

- If T_i wants to lock a record, say R1.1, all R1.1's ancestors must be checked for a lock; R1.1 may be implicitly locked.
  - If implicit locking is not available, a transaction T_i that locks coarse granules should also lock all descendants.
  - This defeats the purpose of introducing multiple granules!
    Why ?

Explicit, Implicit, And Intention Locks

- An **intention** lock on an item *x* means that a transaction performs some operation on a descendant of *x*.
  - What is the need for intention locks ?

- The operation may be determined by the type (mode of the intention lock:
  - irl (intention to read lock)
  - iwl (intention to write lock)
  - riwl (read intention to write lock)

Multi-Granularity 2PL Protocol

	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

Multi-Granularity 2PL Protocol

Growing Phase (top down manner)

- The root of hierarchy must be locked first.
- To set rl(*x*) or irl(x), T_i must have an irl or iwl on x's parent.
- To set wl(x) or iwl(x), T_i must have an iwl on x's parent.
- To read (write) x, T_i must have an rl (wl) on x or one of its ancestors (i.e., must be implicitly or explicitly locked).

Shrinking Phase (bottom up manner)

- T_i cannot release a lock on x if it holds a lock on any of x's children.
- Once T_i unlocks at item, it cannot request another lock on any item.

Implementing MGL

- To rl(x) (or wl(x)), we must first irl (or iwl) all of x's ancestors

  Who does this ?

  Who knows the granularity hierarchy in a system ?
  - How about the Lock Manager ?
  - How about application programmers ?
- Scheduler?
  - It predicts the need for coarse granularity locks based on the transaction's recent behavior
  - it uses lock escalation.
- In the system, where queries are compiled, the compiler may also generate coarse grain requests.

Implementing MGL

- To rl(x) (or wl(x)), we must first irl (or iwl) all of x's ancestors

  Who does this ?

  Who knows the granularity hierarchy in a system ?
  - How about the Lock Manager ?

    The LM has no idea of granules, etc.
  - How about application programmers ?

    They do not bother with lock/unlock operations even for a single item.
- A scheduler sends the appropriate lock requests to the LM. It predicts the need for coarse granularity locks based on the transaction's recent behavior using escalation.
- In the system, where queries are compiled, the compiler may also generate coarse grain requests.

Lock Escalation

- Transactions start locking at fine granularity.
-
- When the number of lock requests exceeds a threshold, the scheduler (or TM) may do one of the following:
  - Escalate the granularity of the transaction's lock requests.
    - Escalating lock requests from level l_k to level l_{k-1} implies a lock conversion on level l_{k-1}.
  - Restart the transaction, this time setting coarser grain locks.

Lock Escalation

		Old Lock				
		ir	iw	r	riw	w
Requested Lock	ir	ir	iw	r	riw	w
	iw	iw	iw	riw	riw	w
	r	r	riw	r	riw	w
	riw	riw	riw	riw	riw	w
	w	w	w	w	w	w

$$\text{Strength} : w > riw > r \sim iw > ir$$

Timestamp Ordering

Timestamp Ordering

- The basic idea:
  - Each transaction T_i has a timestamp $ts(T_i)$.
  - If the scheduler receives an operation by T_i
    - and it has already processed a conflicting operation by T_j
    - and $ts(T_i) < ts(T_j)$
    - then T_i is *aborted*.
  - When a transaction aborts, it must restart with a new (i.e. *larger*) timestamp.

Max Read/Write Timestamps

- To decide whether an operation is in timestamp order, we associate two values with each data item *x*.
  - **max-*rts(x)*:**

    the max *ts* of transactions that performed a Read on *x*.

    If ts(Ti) = max-*rts(x)* then Ti is the youngest transaction that has read X successfully
  - **max-*wts(x)*:**

    the max *ts* of transactions that performed a Write on *x*.

    If ts(Ti) = max-*wts(x)* then Ti is the youngest transaction that has written X successfully

---

Read/Write in Basic TO

- $Read_i(x)$

```
if $ts(T_i)$ < max-$wts(x)$ then
        Abort $T_i$
else
    send $R_i(x)$ to DM;
    max-$rts(x)$ = max(max-$rts(x)$, $ts(T_i)$)
endif;
```

- $Write_i(x)$

```
if $ts(T_i)$ < max-$rts(x)$ or $ts(T_i)$ < max-$wts(x)$ then
    Abort $T_i$
Else
    send $W_i(x)$ to DM;
    max-$wts(x)$ = $ts(T_i)$
endif
```

Timestamp Table

- These rules assume that each operation runs to completion before the next one is submitted to DM.
- For example,
  $S{:}W_1(x)R_2(x)$, with $ts(T_1) < ts(T_2)$
  is a legal TO schedule.
- However, when the the scheduler sends $R_2(x)$ to DM, it must know that $W_1(x)$ is finished.
- Thus, we need
  - *r-in-progress(x)*: number of transactions reading *x*
  - *w-in-progress(x)*: number of transactions writing *x* (0 or 1)
  - *waiting-list(x)*: transactions waiting to access *x*.

Timestamp Table

- This information is stored in the *timestamp table*.

data item	max-rts	max-wts	r-in-progress	w-in-progress	waiting-list
x	10	4	2	0	w_{12}
y	11	12	0	1	r_{20}, w_{21}

Implementing Basic TO Rules

```
Read_i(x)
if ts(T_i) < max-wts(x) then
    Abort T_i                              Must also consider waiting list
else if w-in-progress(x) = 0 then
    send R_i(x) to DM
    max-rts(x) = max(max-rts(x), ts(T_i))
    r-in-progress(x) = r-in-progress(x) + 1
else
    insert R_i to waiting-list(x) in timestamp order
end if
```

Implementing Basic TO Rules

```
Write_i(X)
 if ts(T_i) < max-rts(x) or ts(T_i) < max-wts(x) then
    Abort T_i
else if r-in-progress (x) = 0 and w-in-progress(x) =
0
then                                       Must also consider waiting list
    send W_i(x) to DM
    max-wts(x) = ts(Ti)
    w-in-progress(x) = 1
else
    insert W_i to writing-list(x) in timestamp order
end if
```

Example

	Admission		*Scheduling to DM*		
	max-rts	max-wts	r-in-progress	w-in-progress	waiting-list
Initially	0	0	0	0	-
$R_1(x)$	1	0	1	0	-
$R_3(x)$	3	0	2	0	-
$W_2(x)$	Abort T_2 (because $ts(T_2)$ <max-rts)				
$W_7(x)$	3	0	2	0	W_7
$R_6(x)$	6	0	3	0	W_7
ack($R_1(x)$)	6	0	2	0	W_7
ack($R_3(x)$)	6	0	1	0	W_7

Example

	max-rts	max-wts	r-in-progress	w-in-progress	waiting-list
$R_8(x)$	6	0	1	0	W_7, R_8
ack($R_6(x)$)	6 6	0 7	0 0	0 1	W_7, R_8 R_8
$R_5(x)$	Abort T_5 (because $ts(T_5)$ <max-wts)				
$W_4(x)$	Abort T_4 (because $ts(T_4)$ <max-rts and max-wts)				
$R_9(x)$	6	7	0	1	R_8, R_9
ack($W_7(x)$)	6 9	7 7	0 2	0 0	R_8, R_9 -

Basic TO and Recovery

- Basic TO is not strict or ACA
  - does not prohibit overwriting of uncommitted data.
  - We must somehow delay $W_i(x)$ if x was previously written by T_j until T_j terminates.
  - If we do not want cascading aborts we must also delay read operations on uncommitted data.
- Solution
  - The scheduler sets *w-in-progress* to 1 when a T_i starts the write operation on some x.
    It resets *w-in-progress* to 0 when T_i terminates and not when T_i finishes writing on x.

Thomas' Write Rule

- Consider transactions T_1, T_2, and T_3 where $ts(T_i) = i$. Assume the scheduler has already processed the following sequence of operations:
  $W_1(x)W_3(x)$
- According to basic TO, if the scheduler receives $W_2(x)$, T_2 should abort.
- TWR says ...
  - No problem, simply ignore T_2's write operation;
    - send an *ack* that $W_2(x)$ is successfully performed.
  - What matters is that the last write operation on x was performed by the transaction with the maximum *ts*.

Read Operations and TWR

- Assume transactions T_1, T_2, T_3, T_4, and T_5 and that the scheduler has already received these operations:
  $W_1(x)R_3(x)W_5(x)$
- If the scheduler receives $W_4(x)$, could this operation be ignored?
  - Yes. It is like executing: $W_1(x)R_3(x)W_4(x)W_5(x)$
- If the scheduler receives $W_2(x)$, could this operation be ignored?
  - No. The correct schedule would be:
    $W_1(x)W_2(x)R_3(x)W_5(x)$
    but that's impossible, because T_3 already read the write of T_1. So $W_2(x)$ should be rejected.

TO With TWR

- **$Write_i(x)$:**

```
if $ts(Ti) <$ max-$rts(x)$ then
        abort $Ti$
    else if $ts(Ti) <$ max-$wts(x)$ then
        ignore $W_i(x)$ (i.e., assume it is done)
    else if $w$-$in$-$progress(x) = 0$ and $r$-$in$-$progress(x) = 0$ then
        send $W_i(x)$ to DM
        max-$wts(x) = ts(Ti)$
        $w$-$in$-$progress(x) = 1$
    else
        insert $W_i$ to waiting-list$(x)$ in timestamp order
    end if
```

- **$Read_i(x)$:** Same as in Basic TO

Timestamp Table Management

- To process an operation on *x*, we need timestamp information for *x* (for every *x*). Thus, the timestamp table may become too long.

- The solution can be based on the following idea:
  - The scheduler can delete all x for which it can be sure that it will not receive operations on *x* from a transaction whose *ts* is less than max-*wts(x)*.
  - Two solutions
    - Based on the *ts* of the oldest active transaction.
    - Based on timeout.

Based on the Oldest Transaction

- The scheduler keeps the timestamp of the oldest active transaction T_{oldest}
  - When the table becomes too long, the scheduler removes all x for which
    max-$rts(x) < ts(T_{oldest})$ and max-$wts(x) < ts(T_{oldest})$

  - In this case, we are certain that no transaction should abort when it tries to access a data item which is not in the table.

Timeout

- Assume TM uses a real time clock to generate timestamps. Then at a given time *t*, we are *almost* sure that no transaction is active in the system with a timestamp less then $t-\delta$.
- The scheduler periodically does the following:
  - It sets ts_{min} to be $t-\delta$.
  - It removes from the timestamp table all *x* for which max-*rts* and max-*wts* are less than ts_{min}.
  - It marks the table with ts_{min}.

Timeout

- Now, to process some operation on *x*, the scheduler must proceed as follows:
  - if *x* exists in the table proceed as usual.
  - if *x* is not in the table and $ts(Ti) \geq ts_{min}$ add x to the table and proceed as usual.
  - if *x* is not in the table and $ts(Ti) < ts_{min}$ abort *Ti*.

TO Versus 2PL

In the following, assume that $ts(Ti) = i$.

- In 2PL, a transaction is never aborted because it submitted an operation too late; it simply waits.
- *Example*: the scheduler receives the following requests

  $\boldsymbol{R_2(x)C_2W_1(x)C_1}$

  - In TO, T_1 must abort T_1 submits $W_1(x)$ too late.
  - In 2PL, it is a legal sequence of operations.

TO Versus 2PL

- In 2PL, a transaction *Ti* does not unlock an item *x* until after it has locked all data items it wants to access. Meanwhile *x* is unavailable to other transactions.
- *Example*: The scheduler receives the following requests

  $R_1(x)W_2(x)C_2R_1(y)C_1$

  - In 2PL, T_2 can not write lock x until T_1 unlocks *x* (after $R_1(y)$).
  - In TO, it is a legal sequence of operations.
- Deadlock can not arise in TO.
- Starvation?

Multi-version Concurrency Control

Multiversion Concurrency Control

- Assume the following sequence of events.

  $\boldsymbol{W_0(x)\ C_0 W_2\ (x)\ R_1(x)\ C_2 C_1}$

- This sequence CANNOT be produced by a strict 2PL, or Timestamp-Ordering, because
  - Strict 2PL
    - T_1 can not read lock x until after C_2 .
  - TO
    - Since $ts(T_1) < ts(T_2)$, T_1 should abort when it tries to $R_1(X)$.

Multiversion Concurrency Control

An Idea !!

- If we had kept the old version of x when $W_2\ (x)$, then we could avoid having to delay T_1 in (2PL) or abort T_1 (in TO) by having T_1 read the before image of x
- Disadvantages?
  - Complexity
  - Storage space

Basic Idea

- The DM keeps a list of versions for each x.
  - Version x_i means the version of x produced by a Write on x by transaction T_i.
- When the scheduler receives a $W_i\ (x)$, it sends a $W_i(x_i)$ to DM. Each Write(x) produces a new version of x.
- When the scheduler receives a $R_i\ (x)$, it must decide when to send the operation to DM *and* which version of x to read. A Read operation to the DM will be of the form $R_i(x_j)$.
- If a transaction T is aborted, any version it created is destroyed.

Basic Idea

- Example: Assume the scheduler receives:

  $W_0\ (x)\ C_0W_2(x)\ R_1(x)\ C_2C_1$

  The scheduler sends to DM the following operations:

  $W_0\ (x_0)\ C_0W_2\ (x_2)\ R_1(x_0)\ C_2C_1$

- The above is a legal schedule in both types of schedulers: strict 2PL, TO.

Visibility of Versions

- Versions are under the absolute control of the scheduler and data manager.
  - Users (transactions) still reference data items as usual not by versions.
  - In applications where versions of *x* do exist, each version of *x* must be considered as an individual item.
- ***One-copy serializability*** *(1SR)* is the correctness criterion for Multiversion Concurrency Control.
  - 1SR requires that transaction executions are equivalent to a serial execution of those transactions on a *one-copy* database.

Alternatives for Storing Multiple Versions

Storing multiple versions

- Horizontal Redundancy
  - Extend database schema horizontally
    - Extra "instances" of fields that change
    - 2VNL (2VNL/k)
- Vertical Redundancy
  - Extend database schema verticaly
    - Extra tuples with modified fields
    - MVNL
- [additional material on the web page]

Multi-version Timestamp Ordering

Multiversion Timestamp Ordering

- Each transaction T_i has a unique timestamp $ts(T_i)$.
- Each version of ***x*** is labeled with the timestamp of the transaction that wrote *x*.
- The scheduler translates operations on data items into operations on versions of these data items.

Scheduling Operations

- R_i(x)
  - Find x_k, the version of *x*, where T_k has the largest timestamp less than or equal to $ts(T_i)$.
  - Send $R_i(x_k)$ to DM.
    - Therefore, a Read operation is never delayed or rejected.
- W_i(x)
  - If an operation $R_j(x_k)$, where $ts(T_k) <= ts(T_i) <= ts(T_j)$, has already been processed then reject $W_i(x)$, and restart T_i.
  - Otherwise, send $W_i(x_i)$ to DM.
    - Write operations may abort

Scheduling Operations

- C_i
  - Delay C_i until all transactions that wrote versions read by T_i commit (to ensure recoverability).
  - If one of those transactions aborts, abort T_i too.
    - Thus, a *read-only* transaction may be aborted.
- Can we avoid cascading aborts altogether by using the *write-in-progress* bit?

Deleting Old Versions

- The scheduler must delete versions from the oldest to the newest.
  - Keep the smallest timestamp, ts_{min}, of all currently active transactions (i.e., the timestamp of the oldest active transaction).
    - When the oldest transaction T_i terminates, find the most recent x_k such that
    - $k \leq ts(T_i)$, and
    - x_k is not the most recent version of x.
    - Delete all committed x_j for which $j < k$.

Deleting Old Versions

- Example: Assume

  Versions: $x_1, x_4, x_5, x_8, x_{12}, x_{20}$
  Active transactions: $T_6, T_{10}, T_{12}, T_{14}$
  If T_{10} commits which version should be deleted?
  if T_6 commits which version should be deleted?

- Alternatively, delete periodically all versions older than some number.
  - If the scheduler receives $R_i(x_j)$ and x_j has been deleted, it aborts T_i.

Revisiting 2PL

Two Version 2PL (2V2PL)

- The DM keeps one or two versions of each data item *x*.
- When a T_i wants to write *x*, it sets a *wl(x)* and it creates a new version of *x*, x_i.
- The *wl(x)* prohibits other transactions from writing *x*.
- When T_i commits, the x_i version of x becomes *x*'s unique version (the before image of x may now be deleted).
- Readers are allowed to place a *rl* on the a write locked x and they read the previous version of *x* (the before image).

  Therefore, a Read operation is performed on committed updates only (no cascading aborts).

Commit

- To delete the before image of x_i when T_i commits, we need to know that no other transaction reads *x*.
- We introduced a third lock, *commit lock*. The compatibility matrix is

	rl	wl	cl
rl	y	y	n
wl	y	n	n
cl	n	n	n

Commit

- When the scheduler receives the Commit(T_i),
  - It tries to convert the *wl(x)* on all x updated by T_i to *cl*.
  - Since *rl* and *cl* are not compatible, the scheduler delays the commit of T_i until no transaction reads *x*.
  - It then sends C_i to DM.
  - When *ack*(C_i) is received from DM, it removes the commit or read lock from all *x*'s locked by T_i.
  - It sends C_i to TM.

Read/Write Operations

- *Write*$_i$*(x)*
  - If there is a *wl* or *cl* on *x*, place W_i in *waiting-list(x)*.
  - If T_i already owns a *wl* on *x*, send $W_i(x_i)$ to DM.
  - In any other case (x is unlocked or read locked), set a wl_i(x) and send $W_i(x_i)$ to DM. Data item x remains unaffected.
- *Read*$_i$*(x)*
  - If there is a *cl* on *x*, place R_i in *waiting-list(x)*.
  - If T_i already owns a *wl* on x then send $R_i(x_i)$ to DM.
  - In any other case (i.e., x is unlocked or write locked by another transaction), set rl_i and send R_i(x) to DM.

Discussion

- The 2V2PL is recoverable and avoids cascading aborts.
- Deadlocks are possible for one more reason

  T_1 tries to convert its *rl* on x to *wl*
  T_2 tries to convert its *wl* on x to *cl*

  Nothing special here; use any deadlock detection or prevention technique.
- Usually, in 2V2PL, it takes less time to commit a transaction than to execute it.

  Therefore, commit locks delay Reads less than 2PL's write locks.