CMU SCS

Carnegie Mellon Univ.
Dept. of Computer Science
15-415 - Database Applications

Alternative Concurrency
Control Methods (R&G ch. 17)

CMU SCS

Outline

- serializability; 2PL; deadlocks
- Locking granularity
- Tree locking protocols
- Phantoms & predicate locking
- Optimistic CC
- Timestamp based methods
- Multiversion CC

very popular – used in all commercial systems

CMU SCS

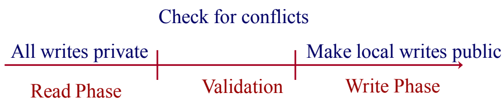
Optimistic CC (Kung&Robinson)

- Assumption: conflicts are rare
- Optimize for the no-conflict case.

CMU SCS

Optimistic CC (Kung&Robinson)

- All transactions consist of three phases
  - **Read:** all writes are to **private** storage.
  - **Validation:** check for no conflicts
  - **Write**: flush 'writes' (or abort!)



CMU SCS

Why Might this Make Sense?

CMU SCS

Why Might this Make Sense?

- All transactions are readers
- Many transactions,
  - each accessing/modifying few tuples
  - from many tuples
  - Low probability of conflict, so again locking is wasted

Validation Phase

- Goal: guarantee only serializable schedules
- Intuitively: at validation, Tj checks its ‘elders’ for RW and WW conflicts
- and makes sure that all conflicts go one way (from elder to younger)

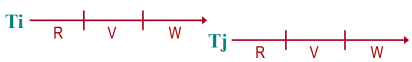
Validation Phase

Specifically:

- Assign each transaction a TN (transaction number)
- Require TN order to be the serialization order
- If TN(Ti) < TN(Tj) ⇒ **ONE** of the following must hold:

Validation Phase (1)

1. Ti completes W before Tj starts R

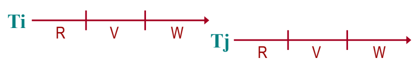


Correctness

1. Ti completes W before Tj starts R

ok W-R
ok W-W

ok R-W

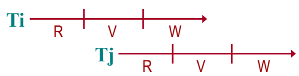


Correctness

- In fact, this is a true serial execution

Validation Phase (2)

2. WS(Ti) ∩ RS(Tj) = ∅ **and**
   Ti completes W before Tj starts W



CMU SCS

Correctness

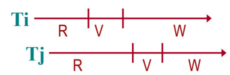
2. WS(Ti) ∩ RS(Tj) = ∅ **and** Ti completes W before Tj starts W

no W-R
ok W-W
ok R-W

CMU SCS

Validation Phase (3)

3. WS(Ti) ∩ RS(Tj) = ∅ **and**
WS(Ti) ∩ WS(Tj) = ∅ **and**
Ti completes its R before Tj completes its R

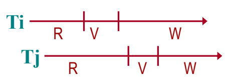


CMU SCS

Correctness:

3. WS(Ti) ∩ RS(Tj) = ∅ **and**
WS(Ti) ∩ WS(Tj) = ∅ **and**
Ti completes its R before Tj completes its R

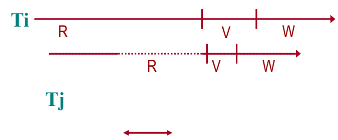
no W-R
no W-W
ok R-W



CMU SCS

Observations

- When to better assign TN's?
- at beginning of read phase: Tj has to wait...

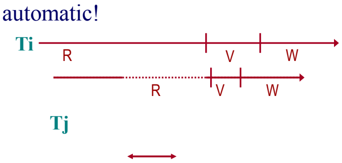


Tj has to wait for W(Ti)

CMU SCS

Observations

- When to better assign TN's?
- at beginning of **validation** phase:
  - Tj can start
  - condition (3): automatic!



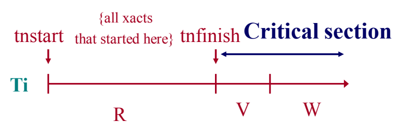
Tj has to wait for W(Ti)

CMU SCS

A Serial Validation Technique

Goal: to ensure conditions 1 and/or 2 above.

- Requires that write phases be done serially
- Validation + Write: in a 'critical section'



CMU SCS

Serial Validation Algorithm

1. Record ***start_tn*** when Xact starts (to identify active Xacts later)
2. Obtain the Xact's real Transaction Number (TN) at the start of validation phase
3. Record read set and write set while running and write into local copy
4. Do validation and write phase inside a critical section

Faloutsos SCS 15-415 #19

CMU SCS

Opt CC vs. Locking

Locking:	Optimistic cc
• order is of first lock;	• order is of *TN(i)*
• wait	• abort
• on deadlock, abort	• on starvation, lock

Faloutsos SCS 15-415 #20

CMU SCS

Conclusions

- Analysis [Agrawal, Carey, Livny, '87]:
  - locking performs well
- All vendors use locking
- Optimistic cc: promising when resource utilization is low.

Faloutsos SCS 15-415 #21

CMU SCS

Outline

- serializability; 2PL; deadlocks
- Locking granularity
- Tree locking protocols
- Phantoms & predicate locking
- Optimistic CC
- ➡ Timestamp based methods
- Multiversion CC

Faloutsos SCS 15-415 #22

CMU SCS

Timestamp based

Motivation:

- can we avoid locks
- AND also avoid the 'critical section' of optimistic CC?

Faloutsos SCS 15-415 #23

CMU SCS

Timestamp based

Main idea

- each xact goes ahead reading and writing
- if it tries to access an object 'from the future', it aborts

(Resembles 'optimistic cc', but writes go directly on the db)

Faloutsos SCS 15-415 #24

CMU SCS

Timestamp CC:

- each xact gets a timestamp (TS)
- each object has
  - a read-timestamp (RTS) (latest xact that read it)
  - and a write-timestamp (WTS) (latest xact that wrote it)

Faloutsos SCS 15-415 #25

CMU SCS

Timestamp CC

- If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart the offending Xact.
- Specifically:

Faloutsos SCS 15-415 #26

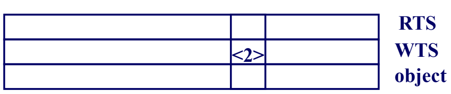
CMU SCS

On 'reads':

time

T1:<1> .... R(O) ....
T2:<2> .... W(O) ....

O



Faloutsos SCS 15-415 #27

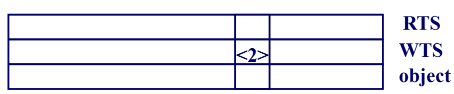
CMU SCS

On 'reads':

time

T1:<1> .... R(O) T1 ABORTS!
T2:<2> .... W(O) ....

O



Faloutsos SCS 15-415 #28

CMU SCS

Timestamp CC – Reads:

- If TS(T) < WTS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it (with same TS? why?)
- Else
  - Allow T to read O.
  - Update RTS(O) to max(RTS(O), TS(T))

Faloutsos SCS 15-415 #29

CMU SCS

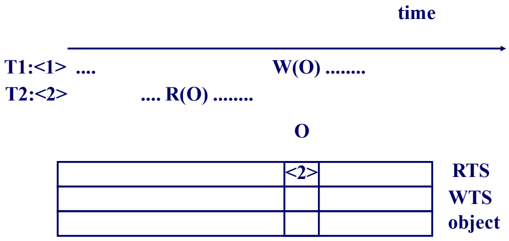
Timestamp CC - Reads

Notice: Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.

Faloutsos SCS 15-415 #30

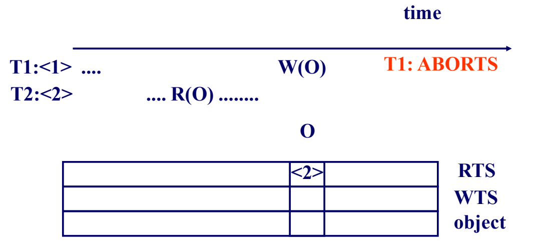
CMU SCS

On ‘writes’: RW conflict



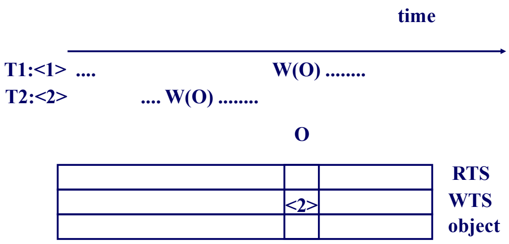
CMU SCS

On ‘writes’: RW conflict



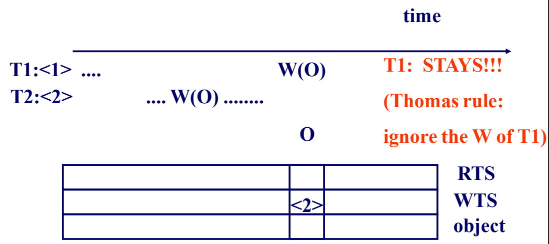
CMU SCS

On ‘writes’: WW conflict



CMU SCS

On ‘writes’: WW conflict



CMU SCS

Timestamp CC: Writes

- If TS(T) < RTS(O), abort and restart T.
- If TS(T) < WTS(O), violates timestamp order of T w.r.t. writer of O.
  - **Thomas Write Rule :** ignore W op, and continue with T
- Else, allow T to write O.
  - and update the WTS(O)

CMU SCS

Digging deeper:

- How about recoverability (ie, cascading aborts?)
- Can they appear, under timestamp CC?

T1	T2
W(A)	
	R(A)
	W(A)
	Commit
…	
Abort	

BAD

CMU SCS

Digging deeper:

- How about recoverability (ie, cascading aborts?)
- Can they appear, under timestamp CC?
- Yes!

T1	T2
W(A)	
	R(A)
	W(A)
	Commit
…	
Abort	

BAD

Faloutsos SCS 15-415 #37

CMU SCS

Timestamp CC and Recoverability

Recoverable schedule: xacts commit only after (and if) all xacts whose changes they read commit

- Unrecoverable schedules are allowed by Timestamp CC !
- (Explain why?)

T1	T2
W(A)	
	R(A)
	W(A)
	Commit
…	
Abort	

BAD

Faloutsos SCS 15-415 #38

CMU SCS

Timestamp CC and Recoverability

- Timestamp CC can be modified, to give recoverable schedules – how?

Faloutsos SCS 15-415 #39

CMU SCS

Timestamp CC and Recoverability

- Timestamp CC can be modified, to give recoverable schedules – how?
- A:
  - Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.

Similar to writers holding X locks until commit, (but not =2PL).

Faloutsos SCS 15-415 #40

CMU SCS

Outline

- serializability; 2PL; deadlocks
- Locking granularity
- Tree locking protocols
- Phantoms & predicate locking
- Optimistic CC
- Timestamp based methods
- Multiversion CC

Faloutsos SCS 15-415 #41

CMU SCS

Multiversion CC

- Readers need NO LOCKS!
  - How would you do it?

Faloutsos SCS 15-415 #42

CMU SCS

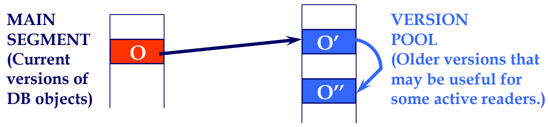
Multiversion CC

- Readers need NO LOCKS!
  - keep a history of all attribute values
  - give each reader the appropriate version
  - (abort the belated writers)

CMU SCS

Multiversion Timestamp CC

- **Idea:** Let writers make a "new" copy while readers use an appropriate "old" copy:



- Readers are always allowed to proceed.
  - But may be blocked until writer commits.

CMU SCS

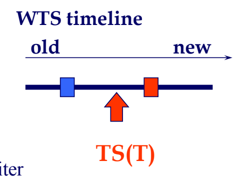
Multiversion CC (Contd.)

- Each Xact is classified as Reader or Writer.
  - Writer *may* write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.
- Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are "too old to be of interest".

CMU SCS

Reader Xact

- Find **newest version** with WTS < TS(T).
- Reader Xacts are never restarted.
  - However, might block until writer of the appropriate version commits.



CMU SCS

Writer Xact

- try to insert/append a new version
- abort if there is a reader 'from the future', that read an older version
- Specifically:

CMU SCS

Writer

time

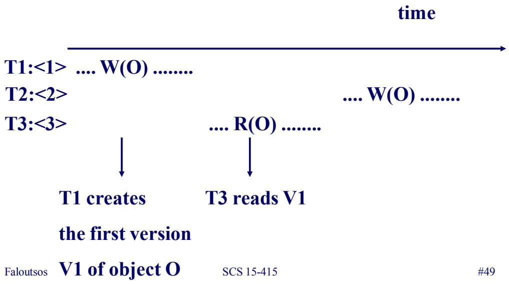
T1:<1> .... W(O) ....

T2:<2> .... W(O) ....

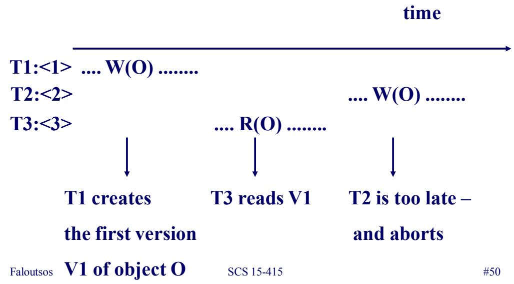
T3:<3> .... R(O) ....

T1 creates the first version V1 of object O

Writer

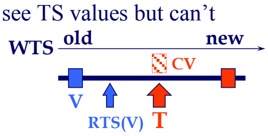


Writer



Writer Xact

- To read an object, follows reader protocol.
- To write an object:
  - Finds **newest version V** s.t. WTS < TS(T).
  - If RTS(V) < TS(T), T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T). (Write is buffered until T commits; other Xacts can see TS values but can't read version CV.)
  - Else, reject write.



Summary – optimistic CC

- Optimistic CC (using a posteriori "validation") aims to minimize CC overheads in an "optimistic'' environment in which reads are common and writes are rare.
- Optimistic CC has its own overheads however; most real systems use locking.

Summary – timestamp based

- Timestamp CC allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability requires ability to block Xacts, which is similar to locking.

Summary - multiversion

- read-only Xacts are never restarted; they can always read a suitable older version.
- Has additional overhead of version maintenance.
  - Oracle uses a flavor of multiversion CC

CMU SCS

Overall summary of CC

- Most commercial systems use
  - locking
  - with wait-for graphs for deadlock detection
  - multiple granularity locking (table, page, row)

Faloutsos SCS 15-415 #55