Final concurrency topics, Crash Recovery

Fall 2008

Slide content adapted from Ramakrishnan & Gehrke

Lock-Based Concurrency Control

- DBMS must ensure
  - only serializable, recoverable schedules are allowed
  - No actions of committed trans lost while undoing aborted trans
- Lock - associated with some object
  - shared or exclusive
- Locking protocol - set of rules to be followed by each transaction to ensure good properties.

Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- 2PL ensures **conflict serializability**
  - Transactions can be ordered by their end of growing phase; this is serializable order.

Strict Two Phase Locking (Strict 2PL)

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object
  - Strict 2PL ensures **conflict serializable** and **cascadeless** schedules

What should we lock?

T1

SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.rating = 8

T2

UPDATE Sailors(Name, Rating, Age) VALUES ("Joe", 8, 33)

- T1 **S**-lock on Sailors; T2 **X**-lock on Sailors
- T1 **S**-lock on all rows with rating=8; T2 X-lock on Joe's tuple.

Phantom

- T1: “Find oldest sailor for each of the rating levels 1 and 2”
  - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
- T2: “Insert new sailor. rating=1, age=96”
- T2: “Deletes oldest sailor with rating = 2 (and, say, age = 80), and commits
- T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).

The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!
- Strict 2PL will not assure serializability

The Phantom Problem

- Phantom problem: A transaction retrieves a collection of tuples and sees different results, even though it did not modify the tuples itself.
  - Conceptually: must lock all *possible* rows.
  - Can lock entire table.
  - Better, use index locking.

Specify isolation level

- General rules of thumb w.r.t. isolation:
  - Fully serializable isolation is more expensive than "no isolation"
    - We can't do as many things concurrently (or we have to undo them frequently)
- For performance, we generally want to specify the most relaxed isolation level that's acceptable for our application.
  - Note that we're "slightly" violating a correctness constraint to get performance!

Specifying isolation level in SQL

SET TRANSACTION [READ WRITE | READ ONLY]
ISOLATION LEVEL [LEVEL];

LEVEL = SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITED

$\downarrow$ Less isolation

The default isolation level is SERIALIZABLE

Locks sets of objects, avoids phantoms

REPEATABLE READ

- T reads only changes made by committed transactions
- No value read/written by T is changed by another transaction until T completes.

- Phantoms possible: inserts of qualifying tuples not avoided.

Locks only individual objects

READ COMMITTED

- T reads only changes made by committed transactions
- No value ~~read~~/written by T is changed by another transaction until T completes.
- Value read by T may be modified while T in progress.
- Phantoms possible.

X locks on written objects, held to end
S locks on read objects, but released immediately.

READ UNCOMMITTED

- Greatest exposure to other transactions
- Dirty reads possible
- Can't make changes: must be READ ONLY
- Does not obtain shared locks before reading
  - Thus no locks ever requested.

Summary of Isolation Levels

Level	Dirty Read	Unrepeatable Read	Phantoms
READ UN-COMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Recovery

Review: The ACID properties

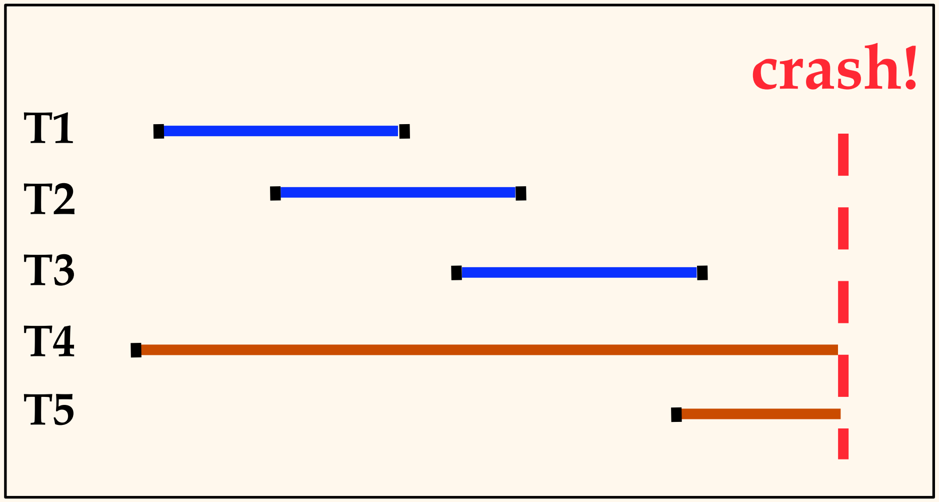
- **A** tomicity: All actions in the Xact happen, or none happen.
- **C** onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one Xact is isolated from that of other Xacts.
- **D** urability: If a Xact commits, its effects persist.

- The **Recovery Manager** guarantees **Atomicity** & **Durability**.

Types of failure

- Transaction failure
  - partially-executed transaction cannot commit
  - ➡ changes must be removed: ROLLBACK
- System failure
  - volatile memory lost
  - ➡ updates of committed Xact persist
  - ➡ updates of aborted or partial Xacts removed
- Media failure
  - corrupted storage media
  - ➡ database brought up-to-date using backup

Motivation



- Desired Behavior after system restarts:
  - T1, T2 & T3 should be **durable**.
  - T4 & T5 should be **aborted** (effects not seen).

Undo and Redo

- UNDO:
  - removing effects of incomplete or aborted transaction (for atomicity)
- REDO:
  - re-instating effects of committed transactions (for durability)
- The work the recovery subsystem must do to support UNDO and REDO depends on **key policies** of the buffer manager.

Handling the Buffer Pool

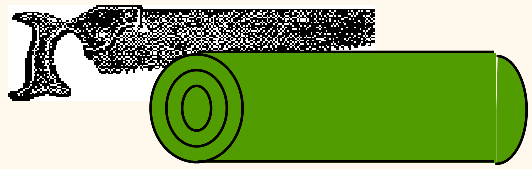
- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommited Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

More on Steal and Force

- **STEAL** (why enforcing Atomicity is hard)
  - *To steal frame F:* Current page in F (say P) is written to disk; some Xact holds lock on P.
    - What if the Xact with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P).
- **NO FORCE** (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time,to support REDOing modifications.

Basic Idea: Logging



- Record REDO and UNDO information, for every update, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- Log: An ordered list of REDO/UNDO actions
  - Log record contains:

    Before image (for UNDO), After image (for REDO)
  - and additional control info (which we'll see soon).

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  - ① Must force the log record for an update *before* the corresponding data page is overwritten on disk.
  - ② Must write all log records for a Xact *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

- Exactly how is logging and recovery done?
  - We'll study the ARIES algorithms.

Log Records

LogRecord fields:

prevLSN
XID
type
pageID
length
offset
before-image
after-image

update records only: pageID, length, offset, before-image, after-image

Possible log record types:

- **Update**
- **Commit**
- **Abort**
- **End** (signifies end of commit or abort)
- Compensation Log Records (CLRs)
  - for UNDO actions

Other Log-Related State

- Transaction Table:
  - One entry per active Xact.
  - Contains XID, status (running/commited/aborted), and lastLSN.
- Dirty Page Table:
  - One entry per dirty page in buffer pool.
  - Contains recLSN -- the LSN of the log record which ***first*** caused the page to be dirty.

Log and Transaction table

pageID	recLSN
p500	
p600	
p505	

Dirty page table

transID	lastLSN
T1000	
T2000	

Transaction table

Oldest

prevLSN	transID	type	pageID	length	offset	before	after
	T1000	update	p500	3	21	ABC	DEF
	T2000	update	p600	3	41	HIJ	KLM
	T2000	update	p500	3	20	GDE	QRS
	T1000	update	p505	3	21	TUV	WXY

Latest

Log

Checkpointing

- Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:
  - begin_checkpoint record: Indicates when chkpt began.
  - end_checkpoint record: Contains current *Xact table* and *dirty page table*. This is a `fuzzy checkpoint':
    - Other Xacts continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
    - No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page. (So it's a good idea to periodically flush dirty pages to disk!)
  - Store LSN of chkpt record in a safe place (*master* record).

The Big Picture: What's Stored Where

LogRecords

prevLSN
XID
type
pageID
length
offset
before-image
after-image

Data pages

each
with a
pageLSN

master record

Xact Table

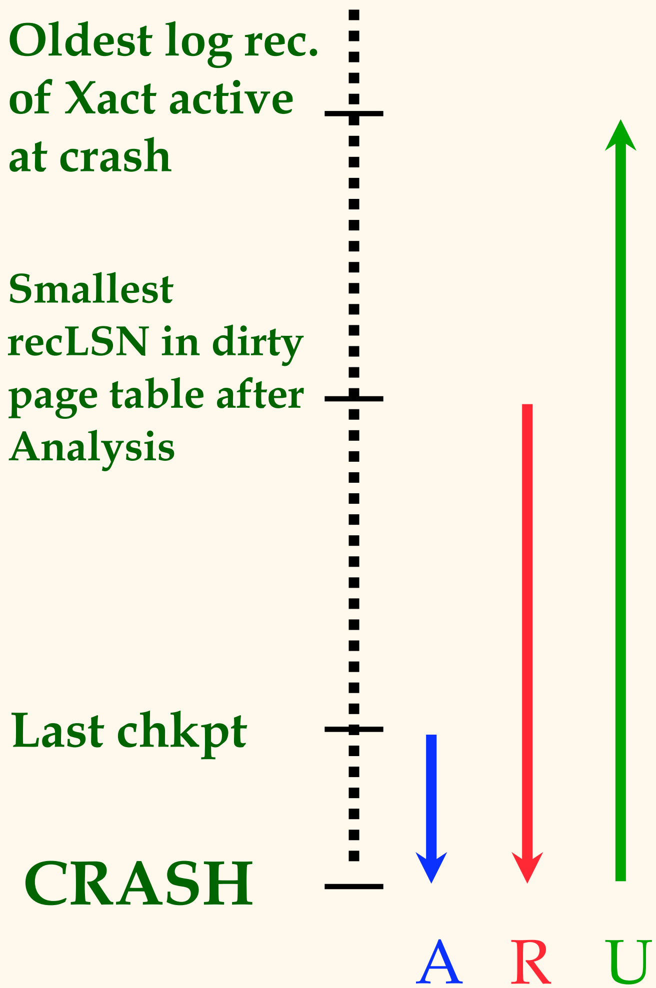
lastLSN
status

Dirty Page Table

recLSN

flushedLSN

Crash Recovery: Big Picture



- Start from a checkpoint (found via master record).
- Three phases. Need to:
  - Analysis: Figure out which Xacts committed since checkpoint, which failed.
  - REDO ***all*** actions.
    - (repeat history)
  - UNDO effects of failed Xacts.

Recovery: The Analysis Phase

- Reconstruct state of most recent checkpoint.
- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table.
  - Other records: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
  - Update record: If P not in Dirty Page Table,
    - Add P to D.P.T., set its recLSN=LSN.

Recovery: The REDO Phase

- We *repeat History* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted Xacts!), redo CLRs.
- Scan forward from log rec containing smallest recLSN in D.P.T. For each CLR or update log rec LSN, REDO the action unless:
  1. Affected page is not in the Dirty Page Table, or
  2. Affected page is in D.P.T., but has recLSN > LSN, or
  3. pageLSN (in DB) ≥ LSN.
- To REDO an action:
  - Reapply logged action.
  - Set pageLSN to LSN. No additional logging!

Recovery: The UNDO Phase

ToUndo={ *l* | *l* a lastLSN of a "loser" Xact}

Repeat:

- Choose largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL
  - Write an End record for this Xact.
- If this LSN is a CLR, and undonextLSN != NULL
  - Add undonextLSN to ToUndo
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

Summary of Logging/Recovery

- Recovery Manager guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.

Summary, Cont.

- Checkpointing: A quick way to limit the amount of log to scan on recovery.
- Recovery works in 3 phases:
  - Analysis: Forward from checkpoint.
  - Redo: Forward from oldest recLSN.
  - Undo: Backward from end to first LSN of oldest Xact alive at crash.
- Upon Undo, write CLRs.
- Redo “repeats history”: Simplifies the logic!