Introduction to Database Systems
CSE 444

Lecture 14

Transactions: Best Practices

(part 2)

Today's Outline

1. The ARIES recovery method (part 2)
2. Snapshot isolation

- Reading: M. J. Franklin. "Concurrency Control and Recovery". Posted on class website

ARIES Overview

- Undo/redo log with lots of clever details
- Physiological logging
- Each log entry has unique *Log Sequence Number*, LSN

Aries Data Structures

- Each page on disk has **pageLSN**:
  = LSN of the last log entry for that page
- Transaction table: each entry has **lastLSN**
  = LSN of the last log entry for that transaction
  Transaction table tracks all active transactions
- Dirty page table: each entry has **recoveryLSN**
  = LSN of earliest log entry that made it dirty
  Dirty page table tracks all dirty pages

Checkpoints

- Write into the log
  - Contents of transactions table
  - Contents of dirty page table
- Very fast ! No waiting, no END CKPT
- But, effectiveness is limited by dirty pages
  - There is a background process that periodically sends dirty pages to disk

ARIES Recovery in Three Steps

- **Analysis pass**
  - Figure out what was going on at time of crash
  - List of dirty pages and running transactions
- **Redo pass (repeating history principle)**
  - Redo all operations, even for transactions that will not commit
  - Get back state at the moment of the crash
- **Undo pass**
  - Remove effects of all uncommitted transactions
  - Log changes during undo in case of another crash during undo

ARIES Method Illustration

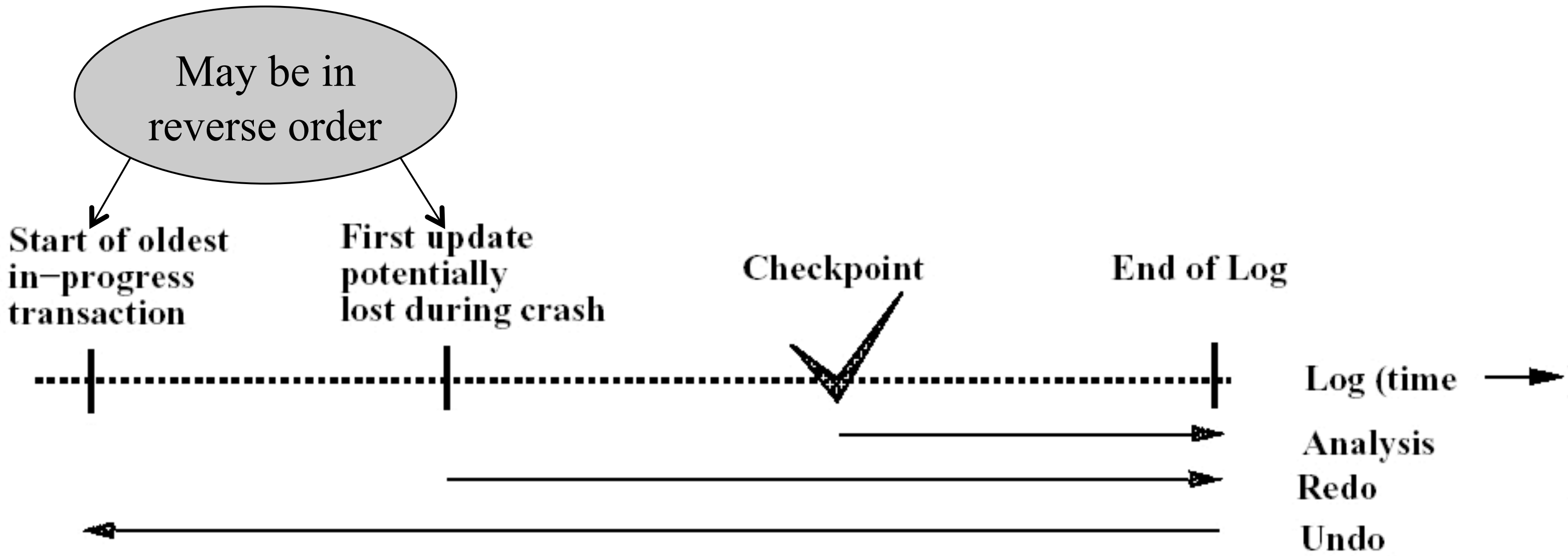

Figure 3: The Three Passes of ARIES Restart

[Franklin97]

Analysis Phase

- Goal
  - Determine point in log where to start REDO
  - Determine set of dirty pages when crashed
    - Conservative estimate of dirty pages
  - Identify active transactions when crashed

- Approach
  - Rebuild transactions table and dirty pages table
  - Start from the latest checkpoint
  - Scan the log, and update the two tables accordingly
  - Find oldest recoveryLSN (firstLSN) in dirty pages tables

Redo Phase

- Goal: redo all updates since firstLSN
- For each log record
  – If affected page is not in the Dirty Page Table then **do not update**
  – If affected page is in the Dirty Page Table but recoveryLSN > LSN of record, then **no update**
  – Else need to read the page from disk; if pageLSN > LSN, then **no update**
  – Otherwise perform update

Undo Phase

- Goal: undo effects of aborted transactions
- Identifies all loser transactions in trans. table
- Scan log backwards
  - Undo all operations of loser transactions
  - Undo each operation unconditionally
  - All ops. logged with compensation log records (CLR)
  - Never undo a CLR
    - Look-up the UndoNextLSN and continue from there

Handling Crashes during Undo

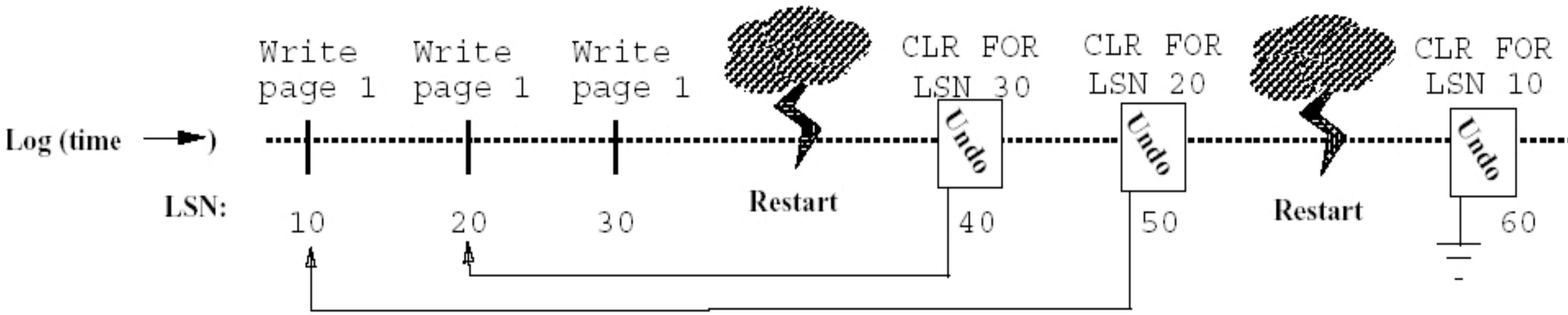

Figure 4: The Use of CLRs for UNDO

[Franklin97]

Today’s Outline

1. The ARIES recovery method (part 2)
2. Snapshot isolation

- Reading: M. J. Franklin. “Concurrency Control and Recovery”. Posted on class website

Snapshot Isolation

- A type of multiversion concurrency control algorithm
- Provides yet another level of isolation

- Very efficient, and very popular
  - Oracle, PostgreSQL, SQL Server 2005

- Prevents many classical anomalies BUT…
- Not serializable (!), yet ORACLE and PostgreSQL use it even for SERIALIZABLE transactions!

Snapshot Isolation Rules

- Each transactions receives a timestamp TS(T)
- Transaction T sees snapshot at time TS(T) of the database
- When T commits, updated pages are written to disk
- Write/write conflicts resolved by “first committer wins” rule
- Read/write conflicts are ignored

Snapshot Isolation (Details)

- Multiversion concurrency control:
  - Versions of X: X_{t1}, X_{t2}, X_{t3}, . . .
- When T reads X, return $X_{TS(T)}$.
- When T writes X: if other transaction updated X, abort
  - Not faithful to “first committer” rule, because the other transaction U might have committed after T. But once we abort T, U becomes the first committer ☺

What Works and What Not

- No dirty reads (Why ?)
- No inconsistent reads (Why ?)
  - A: Each transaction reads a consistent snapshot

- No lost updates (“first committer wins”)

- Moreover: no reads are ever delayed

- However: read-write conflicts not caught !

Write Skew

```
T1:
  READ(X);
  if X >= 50
      then Y = -50; WRITE(Y)
  COMMIT
```

```
T2:
  READ(Y);
  if Y >= 50
      then X = -50; WRITE(X)
  COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Starting with X=50,Y=50, we end with X=-50, Y=-50.
Non-serializable !!!

Write Skews Can Be Serious

- Acidicland had two viceroys, Delta and Rho
- Budget had two registers: taXes, and spendYng
- They had high taxes and low spending…

```
Delta:
  READ(taXes);
  if taXes = ‘High’
       then { spendYng = ‘Raise’;
                 WRITE(spendYng) }
  COMMIT
```

```
Rho:
  READ(spendYng);
  if spendYng = ‘Low’
       then {taXes = ‘Cut’;
                 WRITE(taXes) }
  COMMIT
```

… and they ran a deficit ever since.

Questions/Discussions

- How does snapshot isolation (SI) compare to repeatable reads and serializable?
  - A: SI avoids most but not all phantoms (e.g., write skew)
- Note: Oracle & PostgreSQL implement it even for isolation level SERIALIZABLE
- How can we enforce serializability at the app. level ?
  - A: Use dummy writes for all reads to create write-write conflicts