Introduction to Database Systems
CSE 444

Lectures 13-14

Transactions: Best Practices

Today's Outline

1. User interface:
    1. Read-only transactions
    2. Weak isolation levels
    3. Transaction implementation in commercial DBMSs
2. The ARIES recovery method
3. Snapshot Isolation

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

READ-ONLY Transactions

```
Client 1: START TRANSACTION
          INSERT INTO SmallProduct(name, price)
                    SELECT pname, price
                    FROM Product
                    WHERE price <= 0.99

          DELETE  FROM Product
                    WHERE price <=0.99
          COMMIT

Client 2: SET TRANSACTION READ ONLY
          START TRANSACTION
          SELECT count(*)
          FROM Product

          SELECT count(*)
          FROM SmallProduct
          COMMIT
```

Can help DBMS improve performance

Isolation Levels in SQL

1. "Dirty reads"

   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. "Committed reads"

   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. "Repeatable reads"

   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions

   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

Choosing Isolation Level

- Trade-off: efficiency vs correctness
- DBMSs give user choice of level

Always read DBMS docs!

Beware!!

- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly ACID

1. Isolation Level: Dirty Reads

Implementation using locks:

- “Long duration” WRITE locks
  - A.k.a Strict Two Phase Locking (you knew that !)
- Do not use READ locks
  - Read-only transactions are never delayed

Possible problems: dirty and inconsistent reads

2. Isolation Level: Read Committed

Implementation using locks:

- “Long duration” WRITE locks
- “Short duration” READ locks
  - Only acquire lock while reading (not 2PL)

- Possible problems: unrepeatable reads
  - When reading same element twice,
  - may get two different values

2. Read Committed in Java

In the handout: Lecture13.java - Transaction 1:

```
db.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
db.setAutoCommit(false);
readAccount();
Thread.sleep(5000);
readAccount();
db.commit();
```

Can see a different value

In the handout: Lecture13.java – Transaction 2:

```
db.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
db.setAutoCommit(false);
writeAccount();
db.commit();
```

3. Isolation Level: Repeatable Read

Implementation using locks:

- "Long duration" READ and WRITE locks
  - Full Strict Two Phase Locking
- This is not serializable yet !!!

3. Repeatable Read in Java

In the handout: Lecture13.java - Transaction 1:

```
db.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
readAccount();
Thread.sleep(5000);
readAccount();
db.commit();
```

Now sees the same value

In the handout: Lecture13.java – Transaction 2:

```
db.setTransactionIsolation(Connection. TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
writeAccount();
db.commit();
```

3. Repeatable Read in Java

In the handout: Lecture13.java – Transaction 3:

```
db.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
countAccounts();
Thread.sleep(5000);
countAccounts();
db.commit();
```

Can see a different count

In the handout: Lecture13.java – Transaction 4:

```
db.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
insertAccount();
db.commit();
```

Note: In PostgreSQL will still see the same count.

The Phantom Problem

"Phantom" = tuple visible only during some part of the transaction

```
T1:
   select count(*) from R where price>20
   . . . .
   . . . .
   . . . .
   . . . .
   select count(*) from R where price>20
```

```
T2:
   . . . .
   . . . .
   insert into R(name,price)
           values('Gizmo', 50)
    . . . .
```

$R_1(X), R_1(Y), R_1(Z), W_2(New), R_1(X), R_1(Y), R_1(Z), R_1(New)$

The schedule is conflict-serializable, yet we get different counts !

The Phantom Problem

- The problem is in the way we model transactions:
  - Fixed set of elements
- This model fails to capture insertions, because these *create* new elements
- No easy solutions:
  - Need “predicate locking” but how to implement it?
  - Sol1: Lock on the entire relation R (or chunks)
  - Sol2: If there is an index on ‘price’, lock the index nodes

4. Serializable in Java

In the handout: Lecture13.java – Transaction 3:

```
db.setTransactionIsolation(Connection. TRANSACTION_SERIALIZABLE);
db.setAutoCommit(false);
countAccounts();
Thread.sleep(5000);
countAccounts();
db.commit();
```

Now should see same count

In the handout: Lecture13.java – Transaction 4:

```
db.setTransactionIsolation(Connection. TRANSACTION_SERIALIZABLE);
db.setAutoCommit(false);
insertAccount();
db.commit();
```

Commercial Systems

- DB2: Strict 2PL
- SQL Server:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- PostgreSQL:
  - Multiversion concurrency control
- Oracle
  - Multiversion concurrency control

Today's Outline

1. User interface
2. The ARIES recovery method
3. 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

Granularity in ARIES

- Physical logging for REDO (element=one page)
- Logical logging for UNDO (element=one record)
- Result: logs logical operations within a page
- This is called *physiological logging*
- Why this choice?
  - Must do physical REDO since cannot guarantee that db is in an action-consistent state after crash
  - Must do logical undo because ARIES will only undo loser transactions (this also facilitates ROLLBACKs)

The LSN

- Each log entry receives a unique *Log Sequence Number*, LSN
  - The LSN is written in the log entry
  - Entries belonging to the same transaction are chained in the log via **prevLSN**
  - LSN's help us find the end of a circular log file:

After crash, log file = (22, 23, 24, 25, 26, 18, 19, 20, 21)
Where is the end of the log ? 18

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

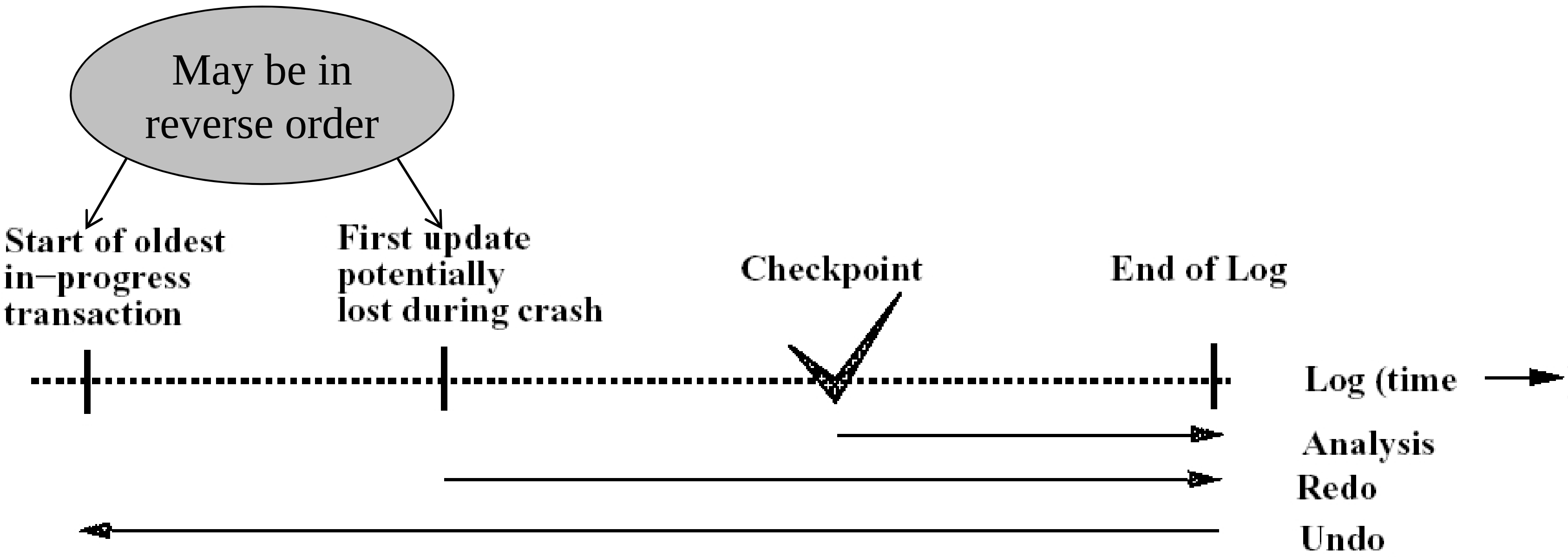

Figure 3: The Three Passes of ARIES Restart

[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

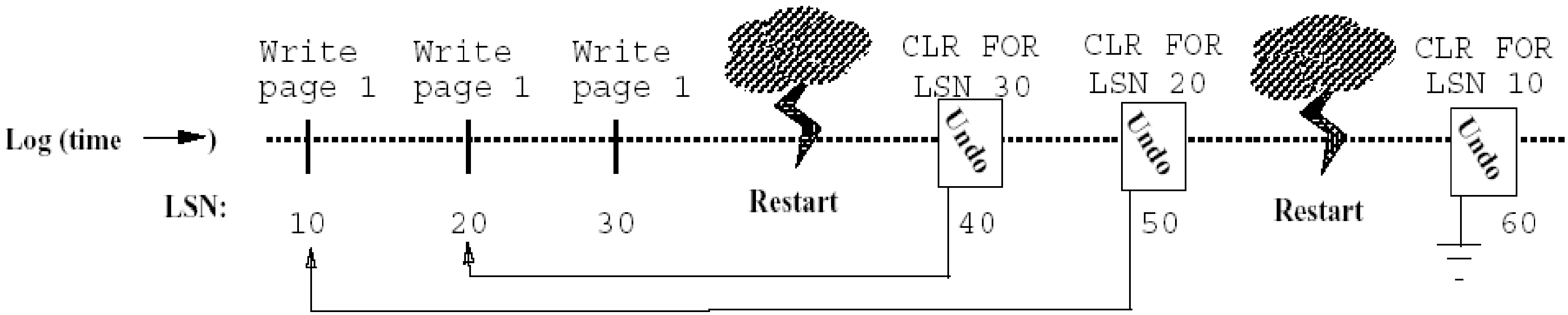

Figure 4: The Use of CLRs for UNDO

[Franklin97]

Today's Outline

1. User interface
2. The ARIES recovery method
3. Snapshot Isolation

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