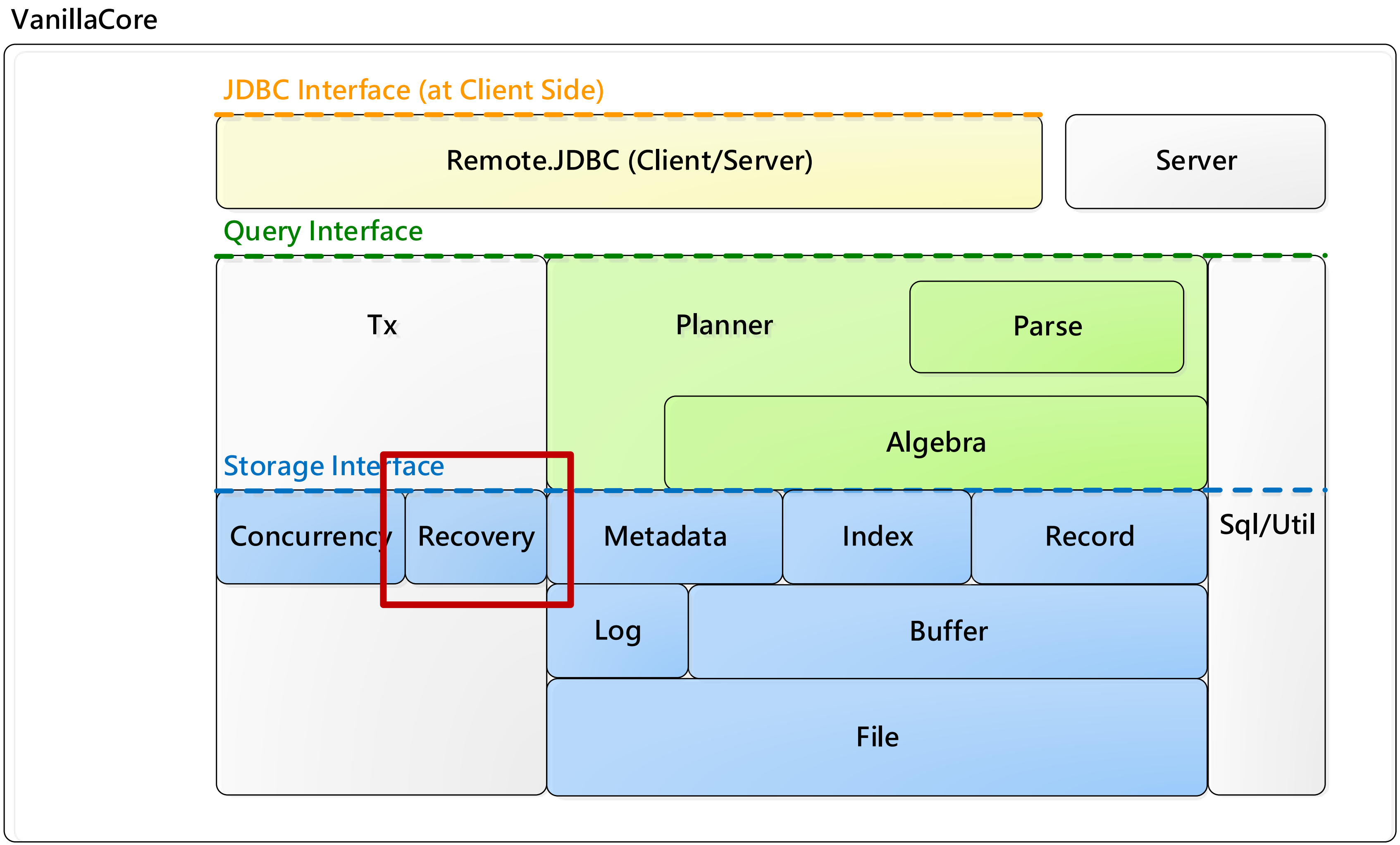
Transaction Management Part II: Recovery

vanilladb.org

Today's Topic: Recovery Mgr



Failure in a DBMS

- Types:
  - Disk crash, power outage, software error, disaster (e.g., a fire), etc.
- In this lecture, we consider only:
  - ***Transaction hangs***
    - Logical hangs: e.g., data not found, overflow, bad input
    - System hangs: e.g., deadlock
  - ***System hangs/crashes***
    - Hardware error, or a bug in software that hangs the DBMS

Assumptions about Failure

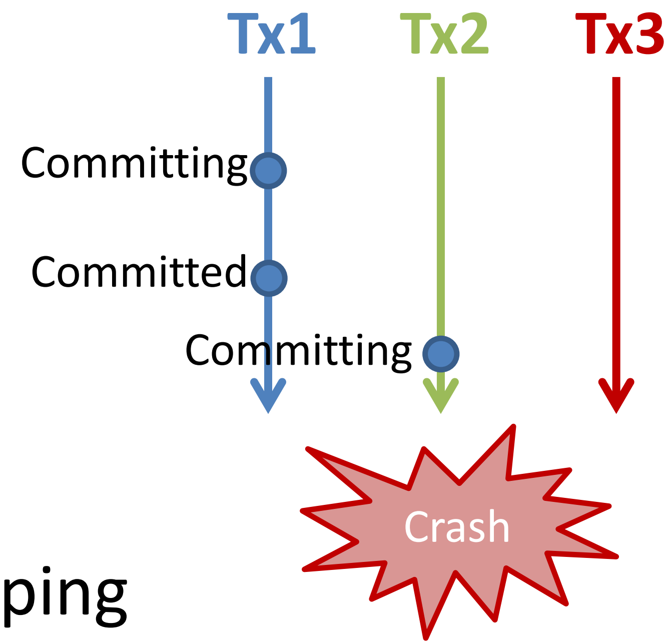
- Contents in nonvolatile storage are ***not corrupted***
  - E.g., via file-system journaling
- No Byzantine failure (zombies)
- Other types of failure will be dealt with in other ways
  - E.g., via replication, quorums, etc.

Review: Naïve A and D

- D given buffers?
- Flush all dirty buffers of a tx ***before*** committing the tx (and returning to the DBMS client)

Review: Naïve A and D

- What if system crashes and then recovers?
- To ensure A, DBMS needs to rollback uncommitted txs (2 and 3) at sart-up
  – Why 3? flushes due to swapping
- Problems:
  – How to determine which txs to rollback?
  – How to rollback all actions made by a tx?



Review: Naïve A and D

- Idea: ***Write-Ahead-Logging*** (***WAL***)
  - Record a ***log*** of each modification made by a tx
    - E.g., <SETVAL, <TX>, <BLK>, <OFFSET>, <VAL_TYPE>, <OLD_VAL> >
    - ***In memory*** to save I/Os
  - To commit a tx,
    1. Write all associated logs to a log file ***before*** flushing a buffer
    2. After flushing, write a <COMMIT, <TX>> log to the log file
  - To swap a dirty buffer (in BufferMgr)
    - All logs must be flushed ***before*** flushing a buffer

Review: Naïve A and D

- Which txs to rollback?
  - Observation: txs with COMMIT logs must have flushed all their dirty blocks
  - Ans: those without COMMIT logs in the log file
- How to rollback a tx?
  - Observation: each action on the disk:
  1. With log and block
  2. With log, but without block
  3. Without log and block
  - Ans: simply ***undo*** actions that are logged to disk, flush all affected blocks, and then writes a <ROLLBACK, <TX>> log
  - Applicable to self-rollback made by a tx

Review: Naïve A and D

- Assumption of WAL: each block-write either succeeds or fails entirely on a disk, despite power failure
  - I.e., no corrupted log block after crash
  - Modern disks usually store enough power to finish the ongoing sector-write upon power-off
  - Valid if block size == sector size or a journaling file system (e.g., EXT3/4, NTFS) is used
    - Block/physical vs. metadata/logical journals

Review: Caching Logs

- Like user blocks, the blocks of the log file are cached
  – Each tx operation is logged ***into memory***
  – To avoid excessive I/Os
- Log blocks are flushed only on either
  – Tx commit, or
  – Flushing of data buffer

System Components related to Recovery

- The ***log manager*** manages the caching for logs
  - Does not understand the semantic of logs
- The ***buffer manager*** ensures WAL for each flushed data buffer
- The ***recovery manager*** ensures A and D by deciding:
  - What to log (semantically)
  - When to flush log tail and buffers
  - How to rollback a tx
  - How to recover a DB from crash

Actions of Recovery Manager

1. Actions during normal tx processing:
   - Adds ***log records*** to cache
   - ***Flushes*** log tail and buffers at the right time (e.g., COMMIT)
   - ***Rolls back*** txs
     - By undoing changes made by each tx
   - On behalf of normal txs
2. Actions after system re-start (from a failure):
   - ***Recovers*** the database to a consistent state
     - By undoing changes made by all incomplete tx
   - In a dedicated ***recovery tx*** (before all normal txs start)

Txn B:

```
Write y = 10;
Read x;
If (x>=4)
    Write x=x+1;
else
    Rollback;
Commit;
```

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

Log Records

- In order to be able to roll back a transaction, the recovery manager saves information in the log
- Recovery manager add a ***log record*** to the log cache each time a loggable activity occurs
  - Start
  - Commit
  - Rollback
  - Update record
  - Checkpoint

Log Records

Txn 27:
start;
getVal(blk0, 46);
setVal(blk1, 58, “abc”);
commit;

- The log records of txn 27:

```
<START, 27>
<SETVAL, 27, student.tbl, 1, 58, ‘kay’, ‘abc’>
<COMMIT, 27>
```

block Id

old value

offset

- In general, multiple txns will be writing to the log concurrently, and so the log records for a given txn will be dispersed throughout the log

```
<START, 27>
<ROLLBACK, 23>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, ‘kay’, ‘abc’>
<COMMIT, 27>
...
```

Why COMMIT/ROLLBACK Logs?

- Used to identify incomplete txs during recovery
- Incomplete txs?
  - E.g., those without COMMIT/ROLLBACK logs on disk
  - To be discussed later

Flushing COMMIT

- When committing a tx, the COMMIT log must be flushed ***before*** returning to the user
  - Why?

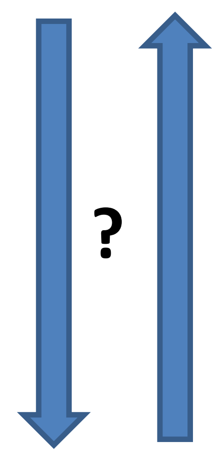
```
public void onTxCommit(Transaction tx) {
    VanillaDb.bufferMgr().flushAll(txNum);
    long lsn = new CommitRecord(txNum).writeToLog();
    VanillaDb.logMgr().flush(lsn);
}
```

- What if the system returns to the client but crashes before writing a commit log?
  - The recovery manager will treat it as an incomplete tx and undo all its changes
  - Dangers durability

Rollback

- The recovery manger can use the log to roll back a tx by ***undoing*** all tx’s modifications
- How to undo txn 27?

```
...
<START, 27>
<ROLLBACK, 23>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, ‘kay’, ‘abc’>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
...
```



Rollback

- Undo txn 27

...

<SETVAL, 23, dept.tbl, 10, 0, 15, 35>

<START, 27>

ensures the correctness of multiple modifications

<SETVAL, 27, dept.tbl, 2, 40, **15**, 9>

<ROLLBACK, 23>

<START, 28>

restores old values

<SETVAL, 28, dept.tbl, 23, 0, 1, 5>

<SETVAL, 27, student.tbl, 1, 58, **'kay'**, 'abc'>

<SETVAL, 27, dept.tbl, 2, 40, **9**, 25>

<START, 29>

undo starts from log tail

<ROLLBACK, 27>

The log records of *T* are more likely to be at the end of log file

Rollback

- The algorithm for rolling back txn *T*
  1. Set the current record to be the most recent log record
  2. Do until the current record is the start record for *T*:
     a) If the current record is an update record for *T*, then write back the old value
     b) Move to the previous record in the log
  3. Flush all dirty buffers made by *T*
  4. Append a rollback record to the log file
  5. Return

Codes for Rollback

```
public void onTxRollback(Transaction tx) {
     doRollback();
     VanillaDb.bufferMgr().flushAll(txNum);
     long lsn = new RollbackRecord(txNum).writeToLog();
     VanillaDb.logMgr().flush(lsn);
}

private void doRollback() {
     Iterator<LogRecord> iter = new LogRecordIterator();
     while (iter.hasNext()) {
          LogRecord rec = iter.next();
          if (rec.txNumber() == txNum) {
               if (rec.op() == OP_START)
                    return;
               rec.undo(txNum);
          }
     }
}
```

- Notice that all dirty buffers are flushed (to be explained later)

Working with Locks

- When a tx *T* that is rolling back, recovery manager requires the DBMS to prevent any access (by other txs) to the data modified by *T*
  - Otherwise, undoing an operation of *T* may override later modifications
- Can easily be enforced by, for example, S2PL

Working with Memory Managers

- ***No*** tx should be able to modify the buffer when that buffer, and its logs, are being flushed; ***and vise versa***
- How?
- For each block, pinning and flushing contend for a short-term X lock, called ***latch***

Latching on Blocks

- To modify a block:
  1. Acquire the latch of that block
  2. Log the update (in memory, done by LogMgr)
  3. Perform the change
  4. Release the latch
- To flush a buffer containing a block:
  1. Acquire the latch of that block (after pin())
  2. Flush corresponding log records
  3. Flush buffer
  4. Release the latch
- Latches have ***nothing*** to do with
  - Locks in S2PL
  - pinning/unpinning in BufferMgr (more like mid-term S locks)

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

Recovery

- When the DMBS restart (from crash), the recovery manager is responsible to restore the database
  - All incomplete txs should be ***rolled back***
- How to identify incomplete txs?

Incomplete Txs (1)

- Recall that when committing/rolling back a tx, the CIMMIT/ROLLBACK log must be flushed before returning to the user

```
public void onTxCommit(Transaction tx) {
     VanillaDb.bufferMgr().flushAll(txNum);
     long lsn = new CommitRecord(txNum).writeToLog();
     VanillaDb.logMgr().flush(lsn);
}


public void onTxRollback(Transaction tx) {
     doRollback();
     VanillaDb.bufferMgr().flushAll(txNum);
     long lsn = new RollbackRecord(txNum).writeToLog();
     VanillaDb.logMgr().flush(lsn);
}
```

Incomplete Txs (2)

- Definition: txs without COMMIT or ROLLBACK records in the log file on disk
- Could be in any of following states when crash happens:
  1. Active
  2. Committing (but not completed yet)
  3. Rolling back

Undo-only Recovery Algorithm

1. For each log record (reading backwards from the end):
   a) If the current record is a commit record then:
      Add that transaction to the list of committed transactions.
   b) If the current record is a rollback record then:
      Add that transaction to the list of rolled-back transactions.
   c) If the current record is an update record and that transaction is not on the committed or rollback list, then:
      Restore the old value at the specified location.

Undo-only Recovery Algorithm

```
public void recover() { // called on start-up
    doRecover();
    VanillaDb.bufferMgr().flushAll(txNum);
    long lsn = new CheckpointRecord().writeToLog();
    VanillaDb.logMgr().flush(lsn);
}

private void doRecover() {
    Collection<Long> finishedTxs = new ArrayList<Long>();
    Iterator<LogRecord> iter = new LogRecordIterator();
    while (iter.hasNext()) {
        LogRecord rec = iter.next();
        if (rec.op() == OP_CHECKPOINT)
            return;
        if (rec.op() == OP_COMMIT || rec.op() == OP_ROLLBACK)
            finishedTxs.add(rec.txNumber());
        else if (!finishedTxs.contains(rec.txNumber()))
            rec.undo(txNum);
    }
}
```

- Flushing and checkpointing will be explained later

Working with Other System Components

- No special requirement since the recovery tx is the ***only*** tx in system at startup
  - Normal txs start only ***after*** the recovery tx finishes

The above RecoveryMgr will make system unacceptably slow!

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

Why Slow?

- Slow commit
  - Flushes: undo logs, dirty blocks, and then COMMIT log
- Slow rollback
  - Flushes: dirty blocks and ROLLBACK log
- Slow recovery
  - Recovery manager need to scan the entire log file (backward from tail) every time

Force vs. No-Force

- Force approach
  - When committing tx, all modifications need to be written to disk ***before*** returning to user
- When client committing a txn
  1. Flush the logs till the LSN of the last modification
  2. Flush dirty pages
  3. Write a COMMIT record to log file on disk
  4. Return

Force vs. No-Force

- Do we really need to flush all dirty blocks when committing a tx?
- Why not just writing logs?
  - No flushing data blocks → faster commit
- But we need ***redo***!
  - Committed txs may not be reflected to disk
  - Buffer state in memory need to be reconstructed

Undo-Redo Recovery

- Undo and redo

older

Beginning of log

new value

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
<START, 27>
<COMMIT, 23>
<SETVAL, 27, dept.tbl, 2, 40, 15, 9>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, 4, 5>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
<START, 29>
<SETVAL, 29, emp.tbl, 1, 0, 1, 9>
<ROLLBACK, 27>
```

newer

Undo-Redo Recovery

- Undo and redo

Completed Txn: 27

older

Beginning of log

Undo

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
<START, 27>
<COMMIT, 23>
<SETVAL, 27, dept.tbl, 2, 40, 15, 9>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, 4, 5>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
<START, 29>
<SETVAL, 29, emp.tbl, 1, 0, 1, 9>
<ROLLBACK, 27>
```

undo txn 29

newer

Undo-Redo Recovery

- Undo and redo

Completed Txn: 27

older

Beginning of log

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
<START, 27>
<COMMIT, 23>
<SETVAL, 27, dept.tbl, 2, 40, 15, 9>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, 4, 5>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
<START, 29>
<SETVAL, 29, emp.tbl, 1, 0, 1, 9>
<ROLLBACK, 27>
```

newer

Undo

undo txn 28

Undo-Redo Recovery

- Undo and redo

Completed Txn: 27, 23

older

Beginning of log

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
<START, 27>
<COMMIT, 23>
<SETVAL, 27, dept.tbl, 2, 40, 15, 9>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, 4, 5>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
<START, 29>
<SETVAL, 29, emp.tbl, 1, 0, 1, 9>
<ROLLBACK, 27>
```

newer

Undo

Undo-Redo Recovery

- Undo and redo

Completed Txn:
27, 23

older

Beginning of log

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
<START, 27>
<COMMIT, 23>
<SETVAL, 27, dept.tbl, 2, 40, 15, 9>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, 4, 5>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
<START, 29>
<SETVAL, 29, emp.tbl, 1, 0, 1, 9>
<ROLLBACK, 27>
```

newer

Undo

Redo

Undo-Redo Recovery

- Undo and redo

Completed Txn: 27, 23

older

Beginning of log

Undo

Redo

redo

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
<START, 27>
<COMMIT, 23>
<SETVAL, 27, dept.tbl, 2, 40, 15, 9>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, 4, 5>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
<START, 29>
<SETVAL, 29, emp.tbl, 1, 0, 1, 9>
<ROLLBACK, 27>
```

newer

Undo-Redo Recovery

- Undo and redo

Completed Txn: 27, 23

older

Beginning of log

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
<START, 27>
<COMMIT, 23>
<SETVAL, 27, dept.tbl, 2, 40, 15, 9>
<START, 28>
<SETVAL, 28, dept.tbl, 23, 0, 1, 5>
<SETVAL, 27, student.tbl, 1, 58, 4, 5>
<SETVAL, 27, dept.tbl, 2, 40, 9, 25>
<START, 29>
<SETVAL, 29, emp.tbl, 1, 0, 1, 9>
<ROLLBACK, 27>
```

newer

Undo

Redo

redo

The Undo-Redo Recovery Algorithm V1

// The undo stage

1. For each log record (reading backwards from the end):
   a) If the current record is a commit record then:
      Add that transaction to the list of committed transactions.
   b) If the current record is a rollback record then:
      Add that transaction to the list of rolled-back transactions.
   c) If the current record is an update record and that transaction is not on the committed or rollback list, then:
      Restore the old value at the specified location.

// The redo stage

2. For each log record (reading forwards from the beginning):
   If the current record is an update record and that transaction is on the committed list, then:
      Restore the new value at the specified location.

Figure 14-6
The undo-redo algorithm for recovering a database

From Database Design and Implementation by Edward Sciore, chapter 14.

Physical Logging

- This algorithm does not consider the actual content stored in the disk
  - Depending on swapping state in buffer manager, some actions may be unnecessary or redundant
- Actions need to be undone/redone ***following the exact order in the log file***

Can We Make Rollback Faster Too?

- Recall that when rolling back a tx, we flush dirty pages and write a rollback log

```
public void onTxRollback(Transaction tx) {
     doRollback();
     VanillaDb.bufferMgr().flushAll(txNum);
     long lsn = new RollbackRecord(txNum).writeToLog();
     VanillaDb.logMgr().flush(lsn);
}

private void doRollback() {
     Iterator<LogRecord> iter = new LogRecordIterator();
     while (iter.hasNext()) {
          LogRecord rec = iter.next();
          if (rec.txNumber() == txNum) {
               if (rec.op() == OP_START)
                    return;
               rec.undo(txNum);
          }
     }
}
```

Slow Rollback

```
public void onTxRollback(Transaction tx) {
	doRollback();
	VanillaDb.bufferMgr().flushAll(txNum);
	long lsn = new RollbackRecord(txNum).writeToLog();
	VanillaDb.logMgr().flush(lsn);
}

private void doRollback() {
	Iterator<LogRecord> iter = new LogRecordIterator();
	while (iter.hasNext()) {
		LogRecord rec = iter.next();
		if (rec.txNumber() == txNum) {
			if (rec.op() == OP_START)
				return;
			rec.undo(txNum);
		}
	}
}
```

- Why flushing dirty buffers?
  - So the recovery tx can skip txs that have been rolled back
- Is it necessary to flush the rollback log record before return?
  - No durability issue, losing rollback record just results in rollback again

Fast Rollback

- No-force:
  - Do ***not*** flush dirty pages during rollback
  - In addition, there's ***no*** need to keep the ROLLBACK record in cache at all!
- Aborted txs will be rolled back again during startup recovery
  - No harm to C: undo operations are ***idempotent*** (i.e., rolling back a tx several times makes no difference than rolling back once)

The Undo-Redo Recovery Algorithm V2

// The undo stage

1. For each log record (reading backwards from the end):

   a) If the current record is a commit record then:

      Add that transaction to the list of committed transactions.

   No (b). All txs not in the committed list are un-done (maybe again)

   b) If the current record is a rollback record then:

      Add that transaction to the list of rolled-back transactions.

   c) If the current record is an update record and that transaction is not on the committed or rollback list, then:

      Restore the old value at the specified location.

// The redo stage

2. For each log record (reading forwards from the beginning):

   If the current record is an update record and that transaction is on the committed list, then:

   Restore the new value at the specified location.

Figure 14-6
The undo-redo algorithm for recovering a database

From Database Design and Implementation by Edward Sciore, chapter 14.

Undo or Redo Phase First?

- Does not matter for the recovery algorithm V1
- But matters for V2!
  - Undo phase ***must precede*** the redo phase
  - Otherwise, C may be damaged due to aborted txs
  - E. g.,

```
<START, 23>
<SETVAL, 23, dept.tbl, 10, 0, 15, 35>
// T23 rolls back (not logged) and release locks
<START, 27>
<SETVAL, 27, dept.tbl, 10, 0, 15, 40>
<COMMIT, 27>
```

  - Rolling back T23 erases the modification made by T27

Undo-Only vs. Undo-Redo Recovery

- Pros of undo-only:
  - Faster recovery
  - No redo logs
- Cons of undo-only:
  - Slower commit/rollback
- Which one?
  - Commercial DBMSs usually choose no-force approach + undo-redo recovry

Steal vs. No Steal

- Can the changes be flushed back to disk before txn commits?
  - Buffer manager replaces the modified page for other transaction's need
  - ***Steal*** approach
- If we can prevent buffers of a uncommitted tx from being flushed, we don't need undo!
  - How? Pin all the modified buffers until tx ends
  - Redo-only recovery

No redo, no undo with force + no steal?

Redo-Only Recovery and Beyond

- No-steal is not practical
- Dirty pages still need to be flushed before commits
  - To ensure durability
- How about crash during flushing?

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

What if system crashes again during recovery?

Should we log the undos/redos?

Idempotent Recovery

- No!
- The rollbacks/recovery need not be undone as long as they are ***idempotent***
  - The database will be the same even if the rollbacks/recovery execute several times
- For each modification done by undo/redo, the recovery manager passes -1 as the LSN number to the buffer manager
  - See `SetValueRecord.undo()`

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

Checkpointing

- As the system keeps processing requests, the log file may become very large
  - Running recovery process is time consuming
  - Can we just read a portion of the log?

- A ***checkpoint*** is like a consistent snapshot of the DBMS state
  - All earlier log records were written by “completed” txns
  - Those txns’ modifications have been flushed to disk
- During recovery, the recovery manager can ignore all the log records before a checkpoint

Quiescent Checkpointing

1. Stop accepting new transactions
2. Wait for existing transactions to finish
3. Flush all modified buffers
4. Append a quiescent checkpoint record to the log and flush it to disk
5. Start accepting new transactions

Quiescent Checkpointing

```
<START, 0>
<SETINT, 0, student.tbl, 0, 38, 2004, 2005>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
      //The quiescent checkpoint procedure starts here
<SETSTRING, 0, student.tbl, 0, 46, amy, aimee>
<COMMIT, 0>
      //tx 3 wants to start here, but must wait
<SETINT, 2, junk, 66, 8, 0, 116>
<COMMIT, 2>
<CHECKPOINT>
<START, 3>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Undo Redo

Figure 14-10
A log using quiescent checkpointing

Quiescent Checkpointing is Slow

- Quiescent checkpointing is simple but may make the system unavailable ***for too long*** during checkpointing process

Root Cause of Unavailability

1. Stop accepting new transactions
2. Wait for existing transactions to finish
3. Flush all modified buffers *May be very long!*
4. Append a quiescent checkpoint record to the log and flush it to disk
5. Start accepting new transactions

Can we shorten the quiescent period?

Nonquiescent Checkpointing

1. Stop accepting new transactions
2. Let $T_1, \dots, T_k$ be the currently running transactions
3. Flush all modified buffers
4. Write the record $<\texttt{NQCKPT}, T_1, \dots, T_k>$ and flush it to disk
5. Start accepting new transactions

Recovery with Nonquiescent Checkpointing

- Txs not in checkpoint log are flushed thus can be neglected

Redo

```
<START, 0>
<SETINT, 0, student.tbl, 0, 38, 2004, 2005>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
<NOCKPT, 0, 2>   Only tx2 needs to be undone
<SETSTRING, 0, student.tbl, 0, 46, amy, aimee>
<COMMIT, 0>   Tx0 has been committed
<START, 3>
<SETINT, 2, junk, 66, 8, 0, 116>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Undo

Figure 14-12
A log using nonquiescent checkpointing

Working with Memory Managers

- ***No*** tx should be able to
  1. append the log, and
  2. modify the buffer

  between steps 3 and 4
- How?
- The checkpoint tx obtains
  1. latch of log file, and
  2. latches of all blocks in BufferMgr

  before step 3
- Then release them after step 4

When to Checkpoint?

- By taking checkpoints periodically, the recovery process can become more efficient
- When is a good time to checkpoint?
  - During system startup (after the recovery has completed and before any txn has started)

```
public void recover() { // called on start-up
    doRecover();
    VanillaDb.bufferMgr().flushAll(txNum);
    long lsn = new CheckpointRecord().writeToLog();
    VanillaDb.logMgr().flush(lsn);
}
```

  - Execution time with ***low workload*** (e.g., midnight)

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

Early Lock Release

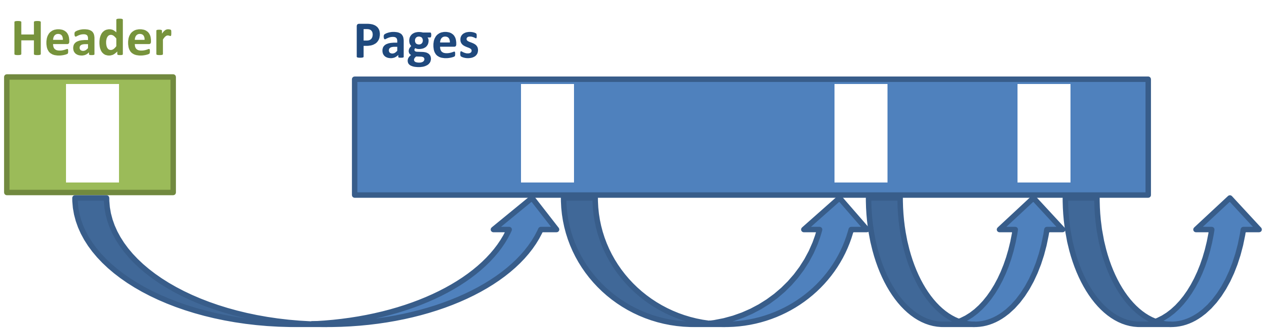
- Recall that there are usually meta-structures in a DBMS
  - E.g., `FileHeaderPage` in a `RecordFile`
  - Indices
- Poor performance if they are locked in strict manner
  - E.g., S2PL on `FileHeaderPage` serializes all insertions and deletions
- Locks on meta-structures are usually ***released early***

Logical Operations

- ***Logical insertions*** to a `RecordFile`:
  - Acquire locks of `FileHeaderPage` and target object (`RecordPage` or a record) in order
  - Perform insertion
  - ***Release*** the lock of `FileHeaderPage` (but not the object)
- Other examples: insertions to an index
  - Following a lock-crabbing protocol
- Better I
- No harm to C
- Needs special care to ensure A and D

Problems of Logical Operations

- Suppose
  1. *T1* inserts a record *A* to a table/file
     - `FileHeaderPage` and a `RecordPage` modified
  2. *T2* inserts another record *B* to the same table
     - Same `FileHeaderPage` and another `RecordPage` modified
  3. *T1* aborts
- If the physical undo record is used to rollback *T1*, *B* will be lost!



Undoing Logical Operations

- How to rollback *T1*?
  - By executing a logical deletion of record *A*
- Logical operations need to be undone ***logically***

Rolling Back a Transaction

- What if *T1* aborts in the middle of a logical operation?
- Log each physical operation performed during a logical operation
- So partial logical operation can be undone, by undoing the physical operations

older

Beginning of log

```
<START, T1>
<SETVAL, T1, RC, 15, 35>
<OPBEGIN, T1, OP1> // insert a record
<SETVAL, T1, H, 100, 105>
<SETVAL, T1, RA, 0, 700>
<OPEND, T1, OP1, delete RA>
... // other tx can access H (early lock release)
```

Identifier can be LSN

newer

Rolling Back a Transaction

older

Beginning of log

```
<START, T1>
<SETVAL, T1, RC, 15, 35>
<OPBEGIN, T1, OP1> // insert a record
<SETVAL, T1, H, 100, 105>
<SETVAL, T1, RA, 0, 700>
<OPEND, T1, OP1, delete RA>
. // other tx can access H
```

Logical undo information

T1 aborts

- Undo *OP1* using physical logs if it is not completed yet
  - ***Locks of physical objects are not released*** so nothing can go wrong
- *OP1* must be undone logically once it is complete
  - Some locks may be released early (e.g., that of *H*)
  - ***Must acquire the locks of physical objects again during logical undo***

Undo an Undo

- What if system crashes when *T1* is undoing a logical undo?
  - The “undo” need to be undone, but how?
- The undo is itself an logical operation
- Why not log all the physical operations of such an undo?
  - The logical undo can be undone now
  - Then at recovery time, logically undo the target logical operation again

Undo an Undo

older

Beginning of log

```
<START, T1>
<SETVAL, T1, RC, 15, 35>
<OPBEGIN, T1, OP1> // insert a record
<SETVAL, T1, H, 100, 105>
<SETVAL, T1, RA, 0, 700>
<OPEND, T1, OP1, delete RA>
...
<SETVAL, T1, H, 123, 100>
<SETVAL, T1, RA, 700, 0>
<OPABORT, T1, OP1>
```

Some locks are released

T1 aborts

Released locks are acquired again

newer

- Be prepared for crashes

Crashes

- Two goals of restart recovery:
  - Rolling back incomplete txs
  - Reconstruct memory state
- Handled by UNDO and REDO phase respectively
- Undo-redo recovery algorithm does ***not*** work anymore!
- Why?
- Since locks may be released early, physical logs may ***depend*** on each other
- Undoing/redoing physical logs must be carried out in the order they happened to ensure C

Example

Beginning of log

```
<START, T1>
<SETVAL, T1, RC, 15, 35>
<OPBEGIN, T1, OP1> // insert a record
<SETVAL, T1, H, 100, 105>
<SETVAL, T1, RA, 0, 700>
<OPEND, T1, OP1, delete RA>
...
// T2 inserts another record (changing H),
// makes some physical changes, and then commits
...
<SETVAL, T1, H, 123, 100>
<SETVAL, T1, RA, 700, 0>
<OPABORT, T1, OP1>
```

T1 aborts

Crash

- To carry out the last two physical ops (i.e., “undo of undo”)
  - *T2* needs to be redone physically ***first***
- Redoing *T2* requires *T1* to be redone ***partially***, even if *T1* will be rolled back eventually

Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

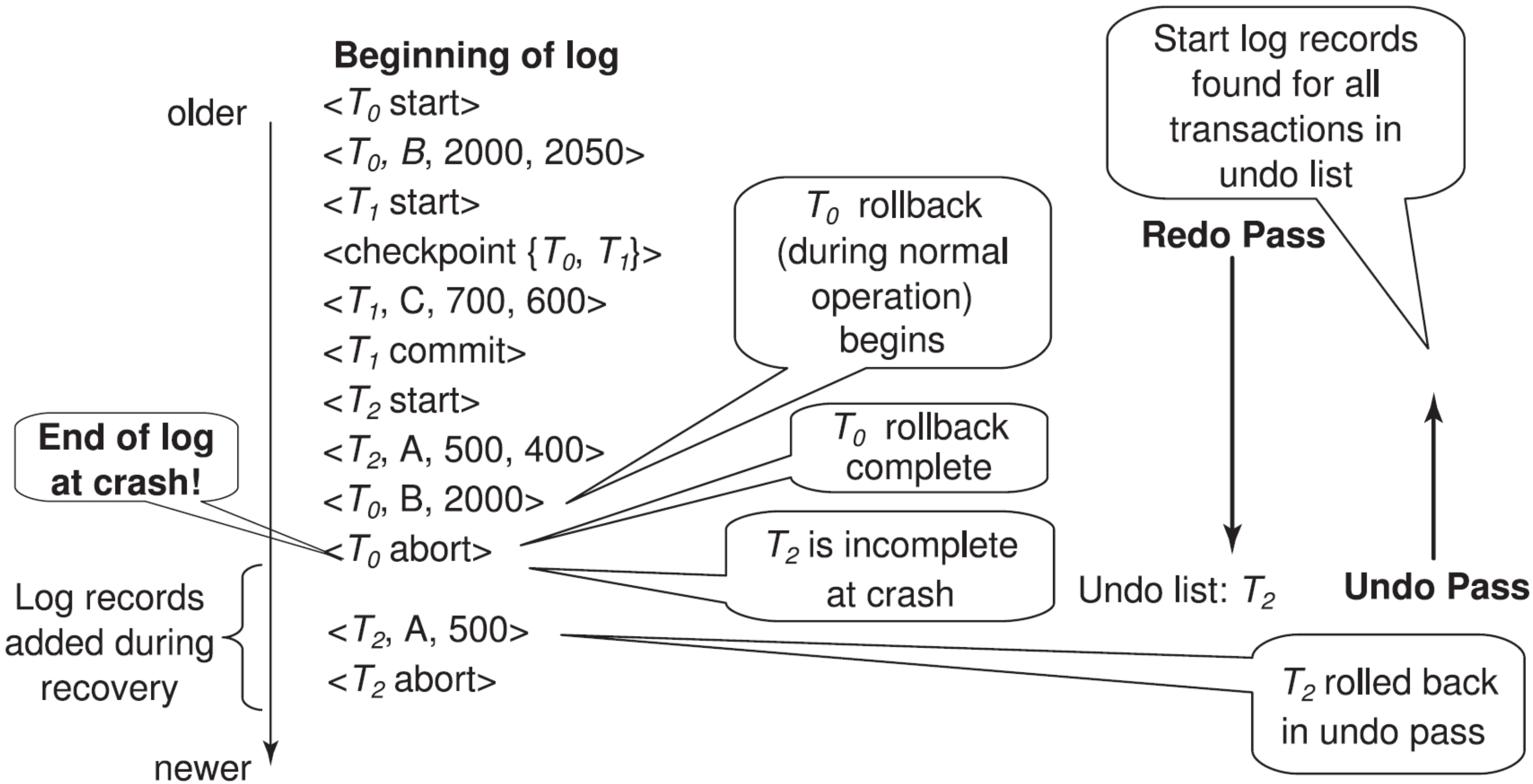
Recovery by Repeating History

- Idea:
  1. Repeat history: replay all dependent physical operations (from the last checkpoint) ***following the exact order they happened***
     - So the memory state can be reconstructed correctly
  2. ***Resume*** rolling back all incomplete txs
     - Logically for each completed logical operation
- This leads to the state-of-the-art recovery algorithm, ***ARIES***
- Steps 1/2 are called REDO/UNDO phase in ARIES
  - Very different from REDO/UNDO phase in previous sections

Compensation Logs

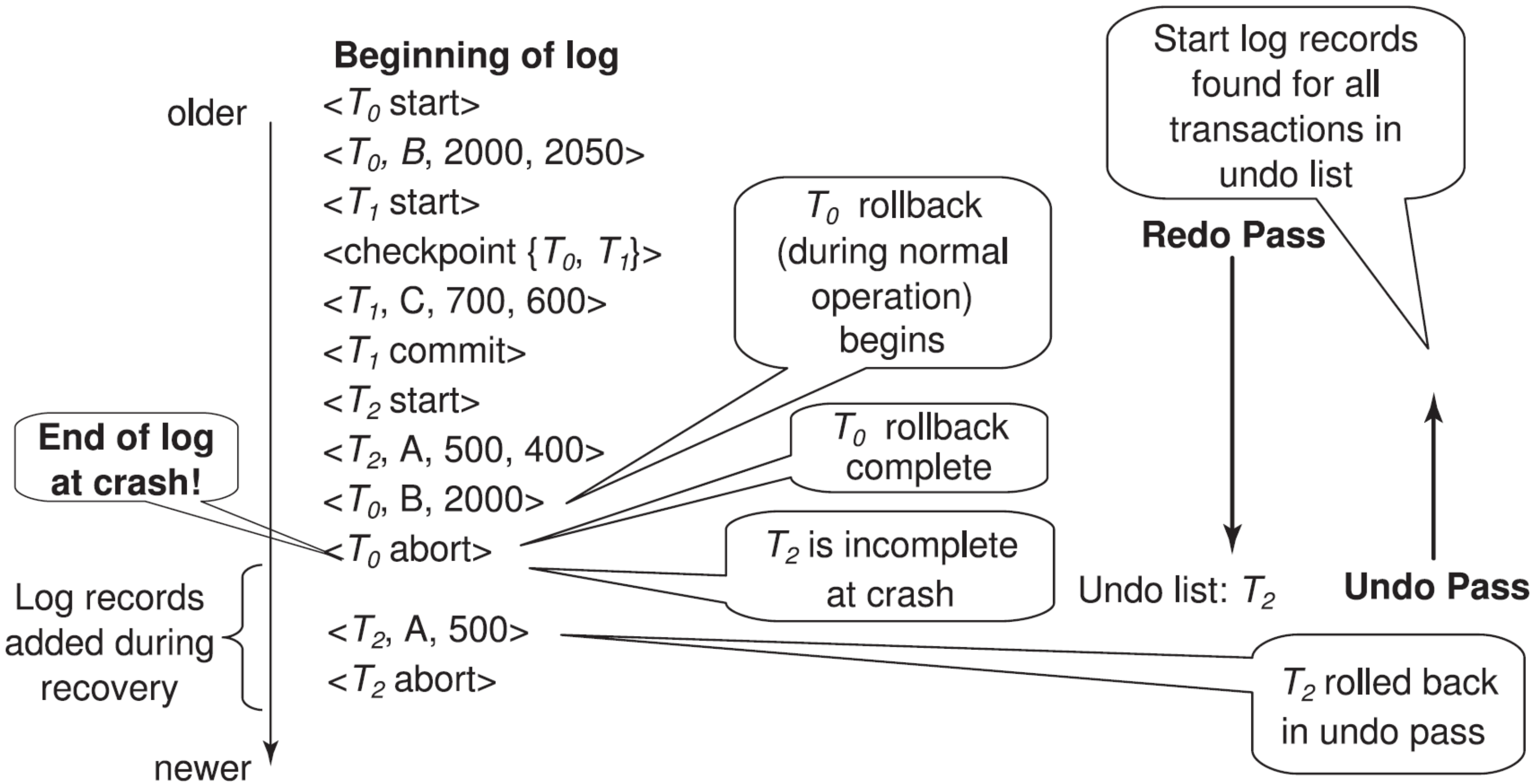
- Replaying history includes replaying previous undos
  - There may be previous ***undos for some physical ops*** (due to, e.g., tx rollbacks or crashes)
  - Need to be replayed too! But not logged currently
- How to replay history in a single phase (log scan)?
- When undoing a physical op, append an redo log, called ***compensation log***, for such undo in LogMgr
- Then , during recovery, RecoveryMgr can simply replay history by redoing ***both*** physical and compensation logs
  - In the order they appear in the log file (***from checkpoint to tail***)

REDO-UNDO Recovery Algorithm V1



- Assuming no logical ops
- Incomplete txs are identified during the REDO phase and kept into a undo list

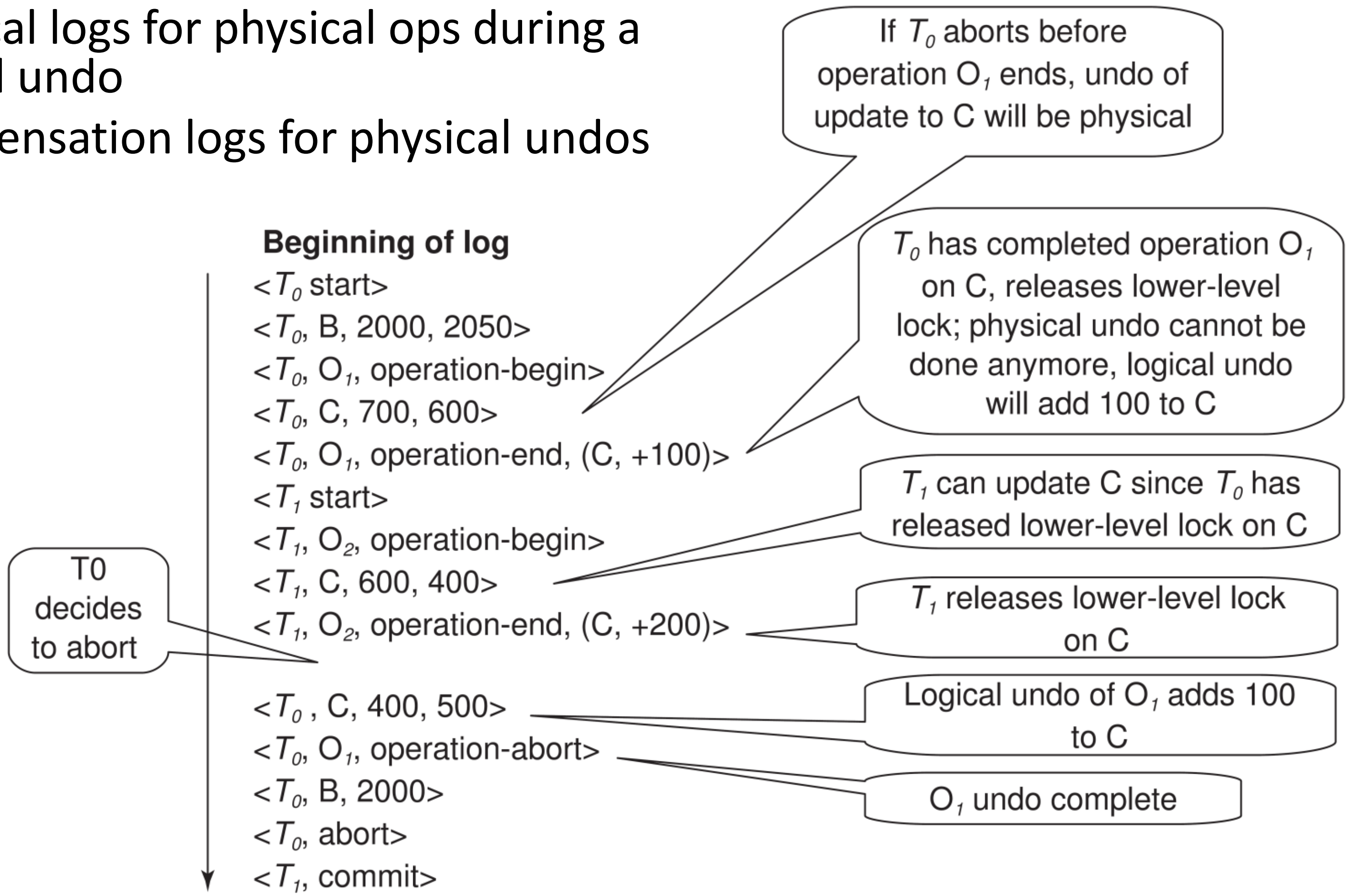
REDO-UNDO Recovery Algorithm V1



- Can handle repeated crashes during recovery
  - Although some redos and undos may be unnecessary

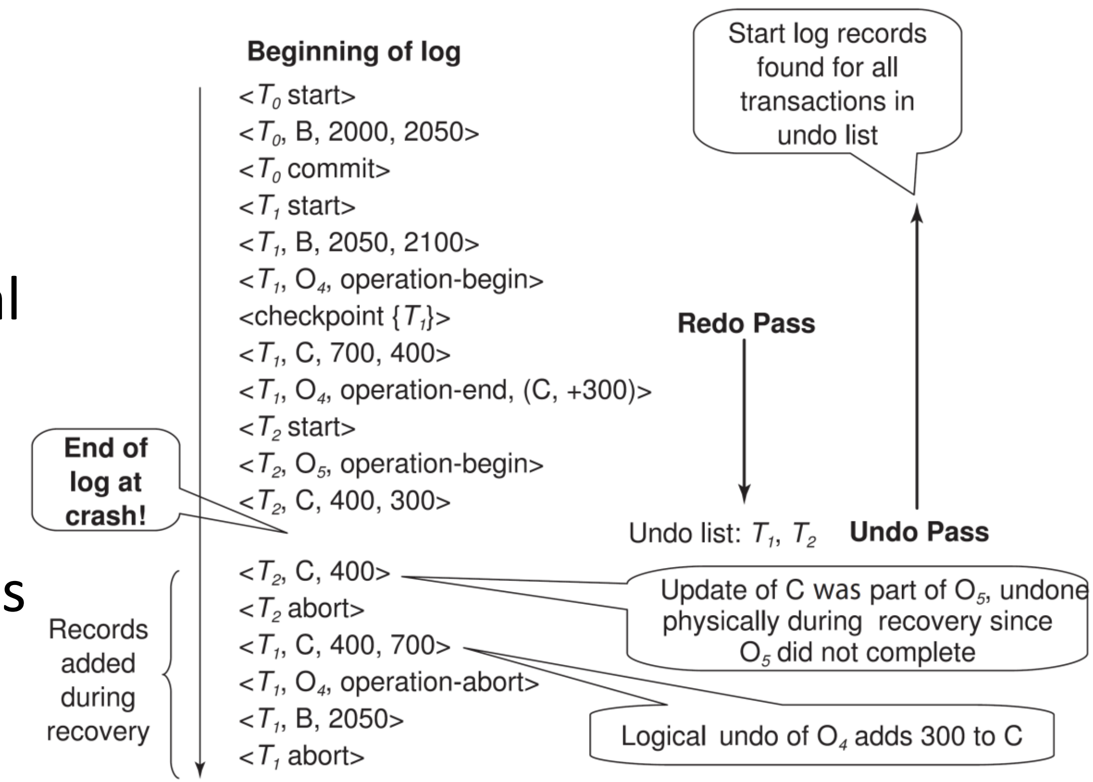
Supporting Logical OPs

- Keep logging (even during UNDO phase):
  - Physical logs for physical ops during a logical undo
  - Compensation logs for physical undos



REDO-UNDO Recovery Algorithm V2

- REDO: repeat history
  - Reply ***both*** physical and compensation logs
- UNDO:
  - Physically for physical and incomplete logical ops
  - Logically for completed logical ops
  - Skip all aborted logical ops, as undoing a logical op is ***not*** idempotent anymore



Non-Idempotent Logical OPs

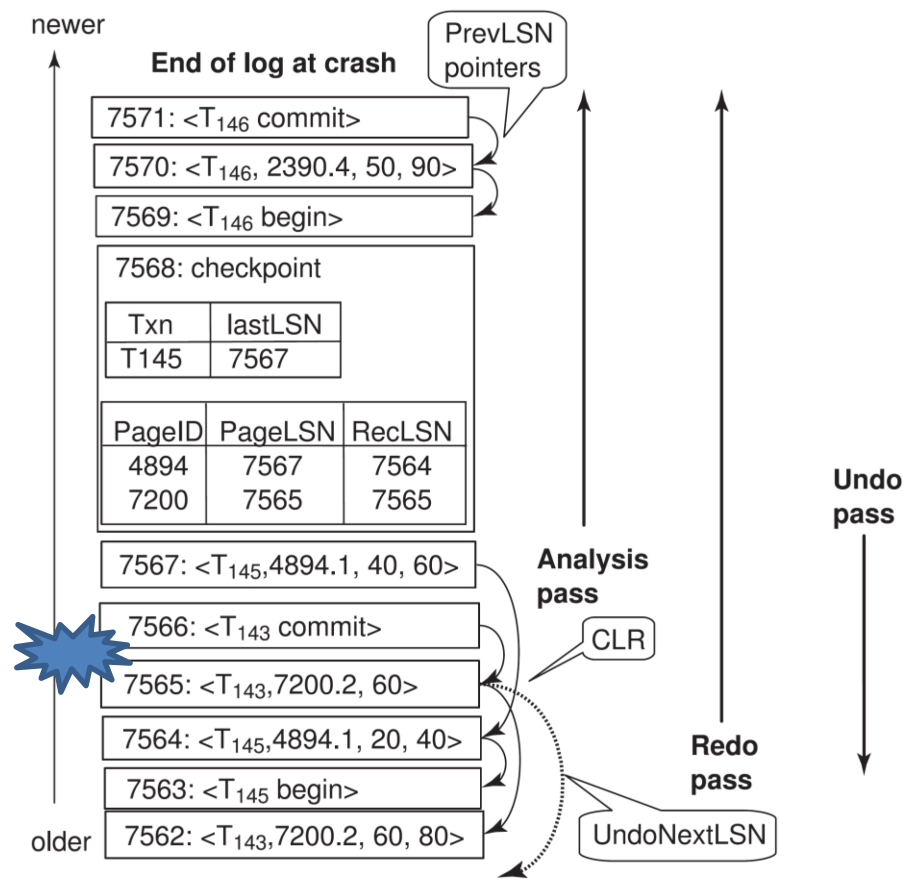
- Note that logical operations, and their logical undos, are ***not*** idempotent
- Completed logical ops and logical undos are repeated ***using physical logs***
  - In REDO phase
  - “history” grows
- So, UNDO phase must skip completed logical undos
  - When rolling back a tx, we, upon finding a record <OPABORT, Ti, Oj>, need to skip all preceding records (including OPEND record for Oj) until <OPBEGIN, Ti, Oj>
  - An operation-abort log record would be found only if a tx that is being rolled back had been partially rolled back earlier

Resume Rollbacks

- How to resume rolling back all incomplete txs in UNDO phase?
- For each incomplete tx:
  - Completed logical undos must be skipped (discussed earilier)
- In addition, completed physical undos can be skipped
  - Optional; just for better performance

Optimization: the PrevLSN and UndoNextLSN pointers

- Logging:
  - Each physical log keeps the PrevLSN
  - Each compensation log keeps the UndoNextLSN
- RecoveryMgr
  - Remembers the last pointer value of each tx in the undo list
  - The next LSN to process during UNDO phase is the max of the pointer values
- Tx rollback can be resumed



Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

Problems of Physical Logging

- Physical logs will be huge!
- For example, if the system wants to sort records in a file, all ops will be logged
  - Common when maintaining the indices
- How to save the number of physical logs?

Physiological logging

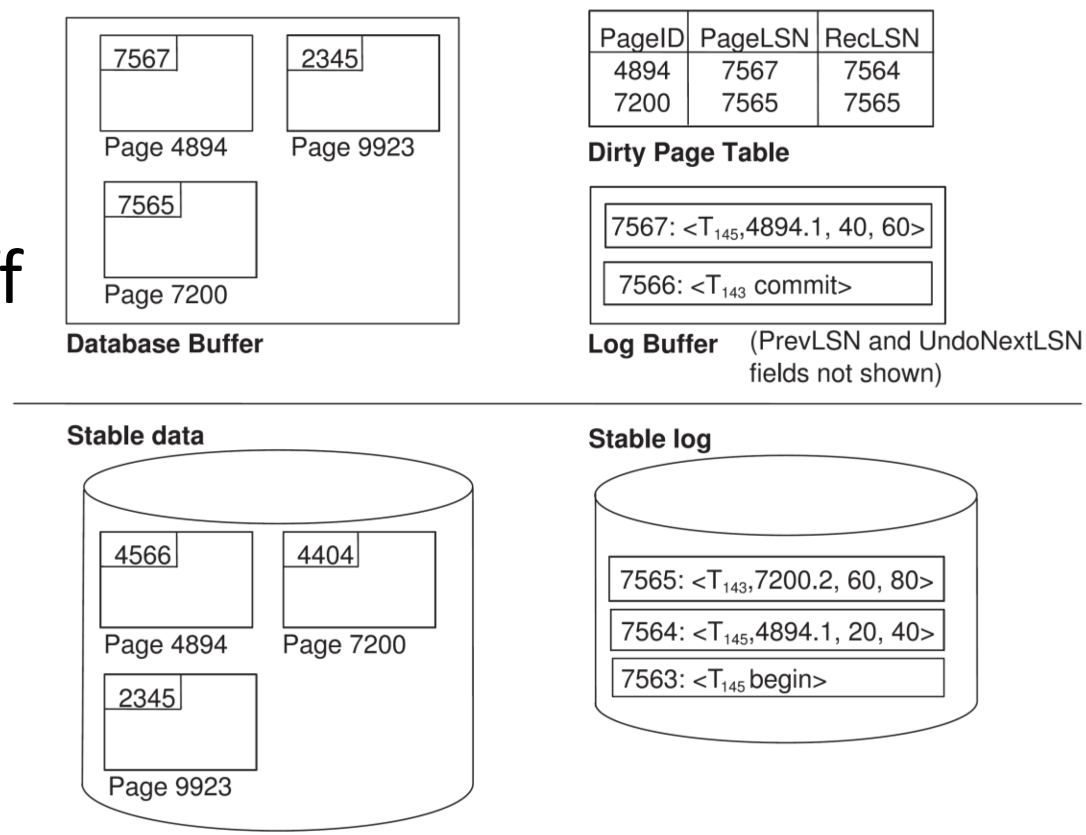
- Observe that, during a sorting op, all physical ops to the same block will be written to disk in just one flush
- Why not log all these physical ops as one logical op?
  - As long as this logical op can be undone logically
- Called ***physiological logs***, in that
  - Physical across blocks
  - Logical within each block
- Significantly save the cost of physical logging
- But complicates recovery algorithm further
  - As REDOs are ***not*** idempotent anymore

REDO-UNDO Recovery Algorithm V3

- During UNDO, threat each physiological op as ***physical***
  - Write compensation log that is also a physiological op
- During REDO, ***skip*** all physiological ops and their compensations that have been replayed previously
  - How?

Avoiding Repeated Replay

- Keep a PageLSN for each block
- Replay a physiological log iff its LSN is larger than the PageLSN of the target block
- Further optimized in ARIES



Outline

- Physical logging:
  - Logs and rollback
  - UNDO-only recovery
  - UNDO-REDO recovery
  - Failures during recovery
  - Checkpointing
- Logical logging:
  - Early lock release and logical UNDOs
  - Repeating history
- Physiological logging
- RecoveryMgr in VanillaCore

The VanillaDB Recovery Manager

- Log granularity: values
- Implements ***ARIES*** recovery algorithm
  - Steal and non-force
  - Physiological logs
  - No optimizations
- Non-quiescent checkpointing (periodically)
- Related package
  - `storage.tx.recovery`
- Public class
  - `RecoveryMgr`
  - Each transaction has its own recovery manager

References

- Database Design and Implementation, chapter 14. Edward Sciore.
- Database management System 3/e, chapter 16. Ramakrishnan Gehrke.
- Database system concepts 6/e, chapter 15, 16. Silberschatz.
- Hellerstein, J. M., Stonebraker, M., and Hamilton, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2, 2007

You Have Assignment!

Assignment: ARIES Optimization

- The current implementation of ARIES in VanillaDB only focused on correctness
- Checkpointing and recovery might be slow
- Basically, you can do anything to make whole system faster during normal operations or recovery
- But the correctness still needs to be hold
  - We will provide test cases to ensure this

Assignment: ARIES Optimization

- For example, our checkpointing is very slow
  - VanillaDB creates a checkpoint by flushing all buffers to disks
- We can make checkpointing faster, but it needs some additional information:
  - Fuzzy checkpointing
  - Dirty page table
  - Transaction table
- Read this paper, or get more information in TA's class