CS422 Principles of Database Systems
Failure Recovery

Chengyu Sun
California State University, Los Angeles

ACID Properties of DB Transaction

- Atomicity
- Consistency
- Isolation
- Durability

Failure Recovery

- Ensure atomicity and durability despite system failures

```
start transaction;
select balance from accounts where id=1;
update accounts set balance=balance-100
       where id=1;
```

System crash →

```
update accounts set balance=balance+100
       where id=2;
commit;
```

System crash →

Failure Model

- System crash
  - CPU halts
  - Data in memory is lost
  - *Data on disk is OK*
- Everything else

Logging

- Log
  - A sequence of *log records*
  - Append only

What Do We Log

Transaction → Log

```
start transaction;
select balance
       from accounts
       where id=1;
update accounts
       set balance=balance-100
       where id=1;
update accounts
       set balance=balance+100
       where id=2;
commit;
```

??

Log Records in SimpleDB

Record Type — Transaction #

```
<START, 27>
<SETINT, 27, accounts.tbl, 0, 38, 1000>
<SETINT, 27, accounts.tbl, 2, 64, 10>
<COMMIT, 27>
```

File Name — Block # — Position — Old Value

General Notation for Log Records

- <START, T>
- <UPATE, T, X, v_x, v_x' >
- <COMMIT, T>
- <ABORT, T>

Recover from System Crash

- Remove changes made by uncommitted transactions – Undo
- Reapply changes made by committed transactions – Redo

Recover with Undo Only

- Prerequisite: all changes made by *committed* transactions have been saved to disk

Example: Create Undo Logging Records

Transaction		Log
Start Transaction;	→	<START, T>
Write(X, v_x')	→	<UPDATE, T, X, v_x>
Write(Y, v_y')	→	<UPDATE, T, Y, v_y>
Commit;		<COMMIT, T>

About Logging

- Undo logging records do not need to store the new values
  - Why??
- The key of logging is to decide when to flush to disk
  - The changes made by the transaction
  - The log records

Example: Flushing for Undo Recovery

- Order the actions, including `Flush(X)` and `Flush(log)`, into a sequence that allows Undo Recovery

Transaction	Log
Start Transaction;	<START, T>
Write(X, v_x')	<UPDATE, T, X, v_x>
Write(Y, v_y')	<UPDATE, T, Y, v_y>
Commit;	<COMMIT, T>

About the Actions

Write(X, v_x')	Update X in memory (i.e. buffer)
Flush(X)	Flush the buffer page that contains X.
<UPDATE, T, X, v_x>	Create a log record in memory – log records have their own buffer page.
Flush(log)	Flush the log buffer page. Note that *all* log records in the log buffer will be flushed to disk.

Order Flush(X) and Flush(<UPDATE,X>) for Undo

- Consider an incomplete transaction
  - (a) Both X and <UPDATE,X> are written to disk
  - (b) X is written to disk but not <UPDATE,X>
  - (c) <UPDATE,X> is written to disk but not X
  - (d) Neither is written to disk

Write-Ahead Logging

- A modified buffer can be written to disk only *after* all of its update log records have been written to disk

Implement Write-Ahead Logging

- Each log record has a unique id called *log sequence number* (LSN)
- Each buffer page keeps the LSN of the log record corresponding to the latest change
- Before a buffer page is flushed, notify the log manager to flush the log up to the buffer's LSN

Order Flush(<COMMIT,T>) for Undo

- <COMMIT,T> cannot be written to disk before new value of X is written to disk
- Commit statement cannot return before <COMMIT,T> is written to disk

Undo Logging

- Write <UPDATE,T,X,v_x> to disk *before* writing new value of X to disk
- Write <COMMIT,T> *after* writing all new values to disk
- COMMIT returns *after* writing <COMMIT,T> to disk

Undo Recovery

- Scan the log
  - *Forward or backward??*
- <COMMIT,T>: add T to a list of committed transactions
- <UPDATE,T,X,v_x>: if T is not in the lists of committed transactions, restore X's value to v_x

Undo Logging and Recovery Example

- Consider two transactions T_1 and T_2
  - T_1 updates X and Y
  - T_2 updates Z
- Show a possible sequence of undo logging
- Discuss possible crushes and recoveries

About Undo Recovery

- No need to keep the new value
- Scan the log once for recovery
- Idempotent – recovery processes can be run multiple times with the same result
- COMMIT must wait until all changes are flushed

Recover with Redo Only

- Prerequisite: *none* of the changes made by *uncommitted* transactions have been saved to disk

Example: Flushing for Redo Recovery

- Order the actions, including `Flush(X)` and `Flush(<log>)`, into a sequence that allows Redo Recovery

Transaction	Log
Start Transaction;	<START, T>
Write(X, v_x')	<UPDATE, T, X, v_x'>
Write(Y, v_y')	<UPDATE, T, Y, v_y'>
Commit;	<COMMIT, T>

Redo Logging

- Write <UPDATE,T,X,v_x'> and <COMMIT,T> to disk *before* writing *any* new value of the transaction to disk
- COMMIT returns *after* writing <COMMIT,T> to disk

Redo Recovery

- Scan the log to create a list of committed transactions
- Scan the log again to replay the updates of the committed transactions
  - *Forward or backward??*

About Redo Recovery

- COMMIT can return after all log records are flushed – transactions complete faster than using Undo-only
  - *Why??*
- A transaction must keep all the blocks it needs pinned until the transaction completes – increases buffer contention

Combine Undo and Redo – Undo/Redo Logging

- Write <UPDATE,T,X,v_x,v_x'> to disk *before* writing new value of X to disk
- COMMIT returns *after* writing <COMMIT,T> to disk

Undo/Redo Recovery

- Stage 1: undo recovery
- Stage 2: redo recovery

Advantages of Undo/Redo

- Vs. Undo??
- Vs. Redo??

Checkpoint

- Log can get very large
- An *Undo/Redo* recovery algorithm can stop scanning the log if it knows
  - All the remaining records are for completed transactions
  - All the changes made by these transactions have been written to disk

Quiescent Checkpointing

- Stop accepting new transactions
- Wait for all existing transactions to finish
- Flush all dirty buffer pages
- Create a <CHECKPOINT> log record
- Flush the log
- Start accepting new transactions

Nonquiescent Checkpointing

- Stop accepting new transactions
- Let $T_1,\ldots,T_k$ be the currently running transactions
- Flush all modified buffers
- Write the record <NQCKPT, $T_1,\ldots,T_k$> to the log
- Start accepting new transactions

Quiescent vs. Nonquiescent

Quiescent	Nonquiescent
<START, 0>	<START, 0>
…	…
<START, 1>	<START, 1>
…	…
<COMMIT, 0>	<NQCHPT, 0, 1>
…	<START, 2>
<COMMIT, 1>	…
<CHPT>	<COMMIT, 0>
<START, 2>	…
…	<COMMIT, 1>
	…

Example: Nonquiescent Checkpoint

- Using Undo/Redo Recovery

<START, 0>
<WRITE, 0, A, v_a, v_a'>
<START, 1>
<START, 2>
<COMMIT, 1>
<WRITE, 2, B, v_b, v_b'>
<NQCKPT, 0, 2>
<WRITE, 0, C, v_c, v_c'>
<COMMIT, 0>
<START, 3>
<WRITE, 2, D, v_d, v_d'>
<WRITE, 3, E, v_e, v_e'>

About Nonquiescent Checkpointing

- Do not need to wait for existing transactions to complete
- Recovery algorithm may stop at
  - <NQCKPT> if all $\{T_1,\ldots,T_k\}$ committed, or
  - <START> of the earliest *uncommitted* transaction in $\{T_1,\ldots,T_k\}$
- *But why do we need to stop accepting new transactions??*

Readings

- Textbook
  - Chapter 13.1-13.3
  - Chapter 14.1-14.3
- SimpleDB source code
  - simpledb.log
  - simpledb.tx
  - simpledb.txt.recovery