CS411
Database Systems

13: Logging and Recovery

Kazuhiro Minami

Outline

- Transaction
- Atomicity
  - Concurrency control
  - Recovery
- Logging
  - Redo
  - Undo
  - Redo/undo

Users and DB Programs

- End users don't see the DB directly
  - are only vaguely aware of its design
  - may be acutely aware of part of its contents
  - SQL is not a suitable end-user interface
- A single SQL query is not a sufficient unit of DB work
  - May need more than one query
  - May need to check constraints not enforced by the DBMS
  - May need to do calculations, realize “business rules”, etc.

Transaction

- DB applications are designed as a set of transactions
- Execute a number of steps in sequence
  - Those steps often modify the database
- Maintain a *state*
  - Current place in the transaction’s code being executed
  - Local variables
- Typical transaction
  - starts with data from user or from another transaction
  - includes DB reads/writes
  - ends with display of data or form, or with request to start another transaction

Atomicity

- Transactions must be "atomic"
  - Their effect is all or none
  - DB must be consistent before and after the transaction executes (not necessarily during!)
- EITHER
  - a transaction executes fully and "commits" to all the changes it makes to the DB
  - OR it must be as though that transaction never executed at all

Requirements for Atomicity

- Recovery
  - Prevent a transaction from causing inconsistent database state in the middle of its process
- Concurrency control
  - Control interactions of multiple concurrent transactions
  - Prevent multiple transactions to access the same record at the same time

A Typical Transaction

- ***User view: "Transfer money from savings to checking"***
- Program: Read savings; verify balance is adequate *, update savings balance and rewrite **; read checking; update checking balance and rewrite***.

  **DB still consistent*

  ***DB inconsistent*

  ****DB consistent again*

"Commit" and "Abort"

- A transactions which only READs expects DB to be consistent, and cannot cause it to become otherwise.
- When a transaction which does any WRITE finishes, it must either
  - COMMIT: "I'm done and the DB is consistent again" OR
  - ABORT: "I'm done but I goofed: my changes must be undone."

System failures

- Problems that cause the state of a transaction to be lost
  - Software errors, power loss, etc.
- The steps of a transaction initially occur in main memory, which is “volatile”
  - A power failure will cause the content of main memory to disappear
  - A software error may overwrite part of main memory

But DB Must Not Crash

- Can't be allowed to become inconsistent
  - A DB that's 1% inaccurate is 100% unusable.
- Can't lose data
- Can't become unavailable

A matter of life or death!

Can you name information processing systems that are more error tolerant?

Solution: use a log

- Log all database changes in a separate, nonvolatile log, coupled with recovery when necessary
  - Undo
  - Redo
  - Undo/redo
- However, the mechanisms whereby such logging can be done in a fail-safe manner are surprising intricate
  - Logs are also initially maintained in memory

Transaction Manager

- May be part of OS, a layer of middleware, or part of the DBMS
- Main duties:
  - Starts transactions
    - locate and start the right program
    - ensure timely, fair scheduling
  - Logs their activities
    - especially start/stop, writes, commits, aborts
  - Detects or avoids conflicts
  - Takes recovery actions

Elements

- Assumption: the database is composed of ***elements***
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)
- Assumption: each transaction reads/writes some elements
- A database has a *state*, which is a value for each of its elements

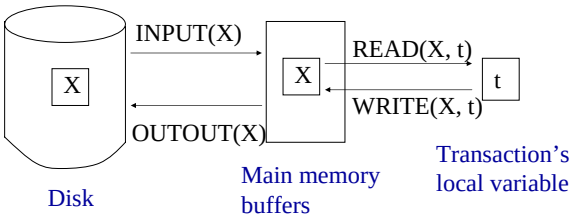
Correctness Principle

- There exists a notion of correctness for the database
  - Explicit constraints (e.g. foreign keys)
  - Implicit conditions (e.g. sum of sales = sum of invoices)
- ***Correctness principle***: if a transaction starts in a correct database state, it ends in a correct database state
- Consequence: we only need to guarantee that transactions are ***atomic***, and the database will be correct forever

Primitive Operations of Transactions

- INPUT(X)
  - read element X to memory buffer
- READ(X,t)
  - copy element X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to element X
- OUTPUT(X)
  - write element X to disk

Primitive Operations of Transactions



Example

READ(A,t); t := t*2;WRITE(A,t)
READ(B,t); t := t*2;WRITE(B,t)

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

The Log

- An append-only file containing log records
- Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transaction that committed
  - Undo other transactions that didn't commit

Undo Logging

Undo logs don't need to save after-images

Log records:

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its *old* value was v

Undo-Logging Rules

U1: If T modifies X, then <T,X,v> must be written to disk before X is written to disk

U2: If T commits, then <COMMIT T> must be written to disk only after all changes by T are written to disk

- Hence: OUTPUTs are done *early*

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Crash recovery is easy with an undo log.

1. Scan log, decide which transactions T completed.
   - ☑ <START T>….<COMMIT T>….
   - ☑ <START T>….<ABORT T>…….
   - ☒ <START T>……
2. Starting from the end of the log, undo all modifications made by incomplete transactions.

The chance of crashing during recovery is relatively high!

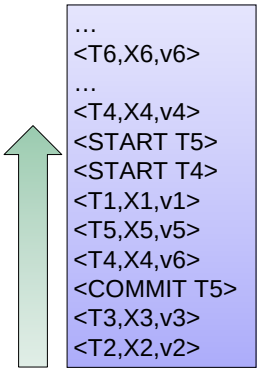
But undo recovery is idempotent: just restart it if it crashes.

Detailed algorithm for undo log recovery

From the *last* entry in the log to the first:

- <COMMIT T>: mark T as completed
- <ABORT T>: mark T as completed
- <T,X,v>: if T is not completed
  then write X=v to disk
  else ignore
- <START T>: ignore

Undo recovery practice



Which actions do we undo, in which order? What could go wrong if we undid them in a different order?

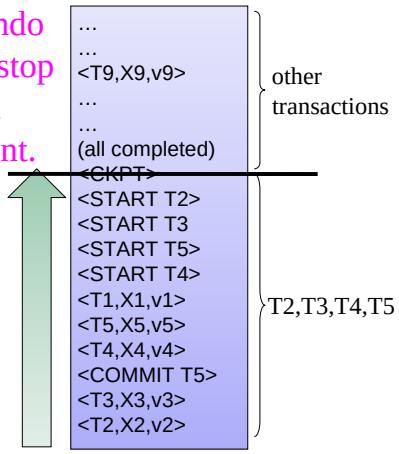
Scanning a year-long log is SLOW and businesses lose money every minute their DB is down.

Solution: checkpoint the database periodically.

Easy version:

1. Stop accepting new transactions
2. Wait until all current transactions complete
3. Flush log to disk
4. Write a <CKPT> log record, flush
5. Resume transactions

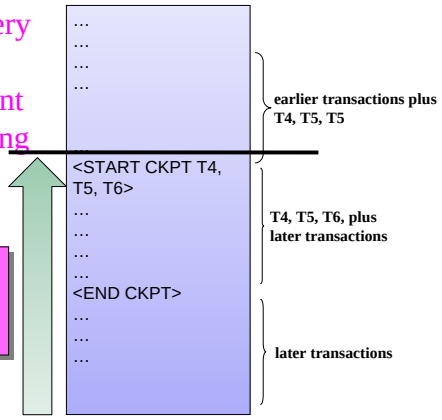
During undo recovery, stop at first checkpoint.



This "quiescent checkpointing" isn't good enough for 24/7 applications. Instead:

1. Write <START CKPT(T1,…,Tk)>, where T1,…,Tk are all active transactions
2. Continue normal operation
3. When all of T1,…,Tk have completed, write <END CKPT>

Example of undo recovery with nonquiescent checkpointing



What would go wrong if we didn't use <END CKPT> ?

Crash recovery algorithm with undo log, nonquiescent checkpoints.

1. Scan log backwards until the start of the latest *completed* checkpoint, deciding which transactions T completed.
   - ☑ <START T>….<COMMIT T>….
   - ☑ <START T>….<ABORT T>…….
   - ☑ <START CKPT {T…}>….<COMMIT T>….
   - ☑ <START CKPT {T…}>….<ABORT T>…….
   - ☒ <START T>……
2. Starting from the end of the log, undo all modifications made by incomplete transactions.

Example

<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>

Redo Logging

Redo log entries are just slightly different from undo log entries.

<START T>
<COMMIT T>
<ABORT T>
} **same as before**

<T, X, new_v>
– T has updated element X, and its **new** value is new_v

Redo logging has one rule.

R1: If T modifies X, then **both** <T, X, new_v> **and** <COMMIT T> must be written to disk before X is written to disk ("late OUTPUT")

*Implicit and reasonable assumption: log records reach disk **in order**; otherwise terrible things will happen.*

Don't have to force all those dirty data pages to disk before committing!

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Recovery is easy with an undo log.

1. Decide which transactions T completed.
   - ☑ <START T>….<COMMIT T>….
   - ☑ <START T>….<ABORT T>…….
   - ☒ <START T>……
2. Read log **from the beginning**, **redo** all updates of **committed** transactions.

The chance of crashing during recovery is relatively high!

But REDO recovery is idempotent: just restart it if it crashes.

Example of redo recovery

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>
…
…

Which actions do we redo, in which order? What could go wrong if we redid them in a different order?

Nonquiescent checkpointing is trickier with a redo log than an undo log

1. Write a <START CKPT(T1,…,Tk)> where T1,…,Tk are the active transactions
2. Flush to disk all dirty data pages of transactions committed by the time the checkpoint started, while continuing normal operation
3. After that, write <END CKPT>

dirty = written

Example of redo recovery with nonquiescent checkpointing

…
<START T1>
…
<COMMIT T1>
…
…
<START CKPT T4, T5, T6>
…
…
<END CKPT>
…
…
<START CKPT T9, T10>
…

All data written by T1 is known to be on disk

1. Look for the last <END CKPT>

2. Redo from <START T>, for committed T in {T4, T5, T6}.

3. Normal redo for committed Tns that started after this point.

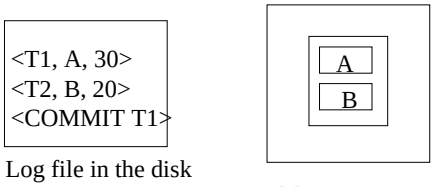
Example

<START T>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Comparison Undo/Redo

- Undo logging:
  - OUTPUT must be done early
  - Increase the number of disk I/O's
  - If <COMMIT T> is seen, T definitely has written all its data to disk (hence, don't need to undo)
- Redo logging
  - OUTPUT must be done late
  - Increase the number of buffers required by transactions
  - If <COMMIT T> is not seen, T definitely has not written any of its data to disk (hence there is not dirty data on disk)
- Would like more flexibility on when to OUTPUT: undo/redo logging (next)

What if an element is smaller than a block?



Log file in the disk

Main memory buffers

Q: Should we write the block to the disk?

Redo/undo logs save both before-images and after-images.

<START T>

<COMMIT T>

<ABORT T>

<T, X, old_v, new_v>

- T has written element X; its **old** value was old_v, and its **new** value is new_v

Undo/Redo-Logging Rule

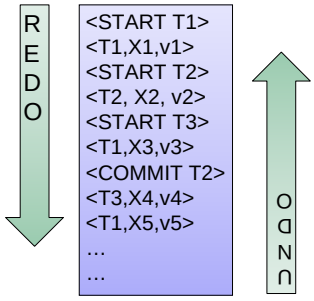
UR1: If T modifies X, then <T,X,u,v> must be written to disk before X is written to disk

Note: we are free to OUTPUT early or late (I.e. before or after <COMMIT T>)

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8,16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

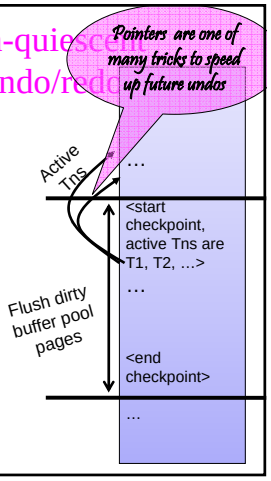
Recovery is more complex with undo/redo logging.

1. Redo all committed transactions, starting at the beginning of the log
2. Undo all incomplete transactions, starting from the end of the log



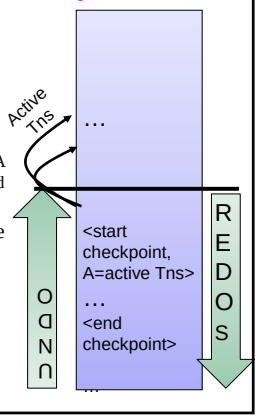
Algorithm for non-quiescent checkpoint for undo/redo

1. Write <start checkpoint, list of all active transactions> to log
2. Flush log to disk
3. Write to disk **all** dirty buffers, whether or not their transaction has committed
   (this implies some log records may need to be written to disk)
4. Write <end checkpoint> to log
5. Flush log to disk



Algorithm for undo/redo recovery with nonquiescent checkpoint

1. Backwards undo pass (end of log to start of last *completed* checkpoint)
   a. C = transactions that committed after the checkpoint started
   b. Undo actions of transactions that (are in A or started after the checkpoint started) and (are not in C)
2. Undo remaining actions by incomplete transactions
   a. Follow undo chains for transactions in (checkpoint active list) – C
3. Forward pass (start of last completed checkpoint to end of log)
   a. Redo actions of transactions in C



Examples

what to do at recovery time?

…
T1 wrote A, …
…
checkpoint start (T1 active)
…
T1 wrote B, …
…
checkpoint end
…
T1 wrote C, …
…

no <T1 commit>

 Undo T1 (undo A, B, C)

Examples

what to do at recovery time?

…
T1 wrote A, …
…
checkpoint start (T1 active)
…
T1 wrote B, …
…
checkpoint end
…
T1 wrote C, …
…
T1 commit

 Redo T1: (redo B, C)