DATABASE RECOVERY

University of Waterloo

List of Slides

1
2 Goals and Setting
3 Failures and What to do
4 Atomicity and Durability
5 Recovery and Buffer Management
6 Approaches to Recovery
7 Log-Based Approaches
8 Example of a LOG
9 Write-Ahead Logging (WAL)
10 Reads and Writes
11 Committing a Transaction
12 Rolling back A Transaction
13 Recovery Using the Log
14 Crash/Failure Recovery
15 Recovery (cont.)
16 Summary

Goals and Setting

Two goals:

- allow transactions to be

  committed
  (with a guarantee that the effects are permanent) or
  aborted
  (with a guarantee that the effects disappear)

- allow the database to be **recovered** to a consistent state in case on HW/power/. . . failure.

Input: a schedule of operations produced by TM.

$\Rightarrow$ including commit and abort requests

Output: a schedule of reads/writes/**forced writes**.

Failures and What to do

Transactions can simplify recovery from failures.

- **System Failure:**
  - ⇒ the database server is halted abruptly
  - ⇒ processing of in-progress SQL command(s) is halted abruptly
  - ⇒ connections to application programs (clients) are broken.
  - ⇒ contents of memory buffers are lost
  - ⇒ database files are not damaged.
- **Media Failure:**
  - ⇒ one or more database files become damaged or inaccessible
  - ⇒ a media failure may cause a system failure, or possibly an orderly system shutdown

Atomicity and Durability

After a failure occurs:

- Transactions that were **active** when the failure occurs will be aborted automatically.
- Transactions that had **committed** before the failure will be durable, i.e., changes they made the the database will not be lost as a result of the failure.

A failure cannot cause a transaction to be partially-executed.

Recovery and Buffer Management

- force every (committed) write on disk?

  $\Rightarrow$ poor response time
  $\Rightarrow$ commit is a bottleneck
  $\Rightarrow$ BUT guarantees durability

- never force uncommitted writes (no steal)?

  $\Rightarrow$ what if a transaction aborts?
  $\Rightarrow$ what if we run out of buffers?

Approaches to Recovery

Two essential approaches:

1. Shadowing

   $\Rightarrow$ copy-on-write and merge-on-commit approach
   $\Rightarrow$ poor clustering
   $\Rightarrow$ used in system R, but not in modern systems

2. Logging

   $\Rightarrow$ use of LOG (separate disk) to avoid forced writes
   $\Rightarrow$ good utilization of buffers
   $\Rightarrow$ preserves original clusters

Log-Based Approaches

A log is a read/**append only** data structure (a file).

When transactions are running, **log records** are appended to the log. Log records contain several types of information:

- **UNDO information:** old versions of objects that have been modified by a transaction. UNDO information can be used to undo database changes made by a transaction that aborts.
- **REDO information:** new versions of objects that have been modified by a transaction. REDO records can be used to redo the work done by a transaction that commits.
- **BEGIN/COMMIT/ABORT** records are recorded whenever a transaction begins, commits, or aborts.

Example of a LOG

log head	$\rightarrow$	T_0,begin
(oldest part)		T_0,X,99,100
		T_1,begin
		T_1,Y,199,200
		T_2,begin
		T_2,Z,51,50
		T_1,M,1000,10
		T_1,commit
		T_3,begin
		T_2,abort
		T_3,Y,200,50
		T_4,begin
(newest part)		T_4,M,10,100
log tail	$\rightarrow$	T_3,commit

Write-Ahead Logging (WAL)

How do we make sure the LOG is consistent with the main database?

- the WAL protocol requires:
  1. **UNDO rule:** a **log record** for an update is written to disk **before** the corresponding data page is written to disk.
  2. **REDO rule: all log records** for a transaction are written to disk before **commit**.

  $\Rightarrow$ (1) guarantees *Atomicity*
  $\Rightarrow$ (2) guarantees *Durability*

- ARIES Algorithm(s)

  $\Rightarrow$ assumes cascadeless schedules and
  $\Rightarrow$ only single copy of any data item in the cache

Reads and Writes

- Read(T_i, x):
  - $\Rightarrow$ get the value v of x (read it)
  - $\Rightarrow$ return it to the upper manager
- Write(T_i, x, v):
  - $\Rightarrow$ add T_i to the set of active transactions
  - $\Rightarrow$ get the old value v' of x (read it)
  - $\Rightarrow$ LOG $[T_i, x, v', v]$
  - $\Rightarrow$ write v to x (it is in the cache!)
  - $\Rightarrow$ acknowledge the operation

Committing a Transaction

- Commit(T_i)
  - $\Rightarrow$ LOG $[T_i, \text{commit}]$
  - $\Rightarrow$ flush all log records
  - $\Rightarrow$ acknowledge commit
  - $\Rightarrow$ LOG $[T_i, \text{end}]$

The sequence of operations is **crucial**

$\Rightarrow$ T_i is considered committed when its
commit gets written to the log.
(this guarantees that all updates are also in the LOG)

$\Rightarrow$ if this doesn't happen, it is aborted.

Rolling back A Transaction

- Abort(T_i)
  - $\Rightarrow$ scan LOG backwards for data items updated by T_i:
    - $*$ read x
    - $*$ restore the old value of x
  - $\Rightarrow$ LOG $[T_i, \text{abort}]$
  - $\Rightarrow$ acknowledge abort
  - $\Rightarrow$ LOG $[T_i, \text{end}]$

Note that this only restores the values in the cache!

Recovery Using the Log

recover from a system failure:

The database server can use the log to determine which transactions were active when the failure occurred, and to undo their database updates. Also, it may use the log to recreate the committed updates that may have been lost.

recover from a media failure:

The database server can use the log to determine which transactions committed since the most recent backup, and to redo their database updates.

abort a single transaction:

The database server can use the log to undo any database updates made by the aborted transaction.

Crash/Failure Recovery

When system accidentally crashes:

1. Let $R = \{\}$ and $U = \{\}$
2. scan LOG to find all **active** and **committed** transactions
3. scan LOG **backwards** for records $[T_i, x, v', v]$ and for each such record such that $x \notin R \cup U$:
   - read x
   - if T_i was committed then

     $\Rightarrow$ write v into x and
     $\Rightarrow R := R \cup \{x\}$
   - if T_i was active (but not committed) then

     $\Rightarrow$ write v' into x and
     $\Rightarrow U := U \cup \{x\}$
4. when done, write abort and end records for all transactions that were active but not committed
5. restart completed

Recovery (cont.)

- Restart operation is **idempotent**

  $\Rightarrow$ if the system crashes during restart, we don't mind

- Restart has to scan the whole log (potentially)

  $\Rightarrow$ ARIES LOGs what pages were flushed when
  $\Rightarrow$ the UNDO/REDO executed in two phases
  $\Rightarrow$ this allows to analyse the log only for relevant info

- Logs can grow very big over time:

  $\Rightarrow$ Checkpointing (interleaved with transactions!)

Summary

- the DBMS must guarantee durability and atomicity

  $\Rightarrow$ responsibility of the **recovery manager**

- synchronous writing is too inefficient

  $\Rightarrow$ replaced by synchronous writes to LOG
  $\Rightarrow$ much cheaper (append only)

- the penalties

  $\Rightarrow$ more complex writes (2 ops instead of 1)
  $\Rightarrow$ need to follow WAL
  $\Rightarrow$ during restart we need to scan the LOG