CMU SCS
Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo
Lecture#23: Crash Recovery – Part 1
(R&G ch. 18)

Last Class

• Basic Timestamp Ordering
• Optimistic Concurrency Control
• Multi-Version Concurrency Control
• Multi-Version+2PL
• Partition-based T/O

Today’s Class

• Overview
• Write-Ahead Log
• Checkpoints
• Logging Schemes
• Shadow Paging
• Examples

Motivation

BEGIN
R(A)
W(A)
⋮
COMMIT

T1

Buffer Pool
Memory
Disk
Page

Faloutsos/Pavlo
CMU SCS 15-415/615
2

Faloutsos/Pavlo
CMU SCS 15-415/615
3

Faloutsos/Pavlo
CMU SCS 15-415/615
4
Crash Recovery

- Recovery algorithms are techniques to ensure database **consistency**, transaction **atomicity** and **durability** despite failures.
- Recovery algorithms have two parts:
  - Actions during normal txn processing to ensure that the DBMS can recover from a failure.
  - Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.

DBMS is divided into different components based on the underlying storage device.

Need to also classify the different types of failures that the DBMS needs to handle.

Storage Types

- **Volatile Storage:**
  - Data does not persist after power is cut.
  - Examples: DRAM, SRAM
- **Non-volatile Storage:**
  - Data persists after losing power.
  - Examples: HDD, SDD
- **Stable Storage:**
  - A non-existent form of non-volatile storage that survives all possible failures scenarios.

Use multiple storage devices to approximate.

Failure Classification

- Transaction Failures
- System Failures
- Storage Media Failures
### Transaction Failures

- **Logical Errors:**
  - Transaction cannot complete due to some internal error condition (e.g., integrity constraint violation).

- **Internal State Errors:**
  - DBMS must terminate an active transaction due to an error condition (e.g., deadlock).

### System Failures

- **Software Failure:**
  - Problem with the DBMS implementation (e.g., uncaught divide-by-zero exception).

- **Hardware Failure:**
  - The computer hosting the DBMS crashes (e.g., power plug gets pulled).
  - **Fail-stop Assumption:** Non-volatile storage contents are assumed to not be corrupted by system crash.

### Storage Media Failure

- **Non-Repairable Hardware Failure:**
  - A head crash or similar disk failure destroys all or part of non-volatile storage.
  - Destruction is assumed to be detectable (e.g., disk controller use checksums to detect failures).

- No DBMS can recover from this. Database must be restored from archived version.

### Problem Definition

- Primary storage location of records is on non-volatile storage, but this is much slower than volatile storage.

- Use volatile memory for faster access:
  - First copy target record into memory.
  - Perform the writes in memory.
  - Write dirty records back to disk.
**Problem Definition**

- Need to ensure:
  - The changes for any txn are durable once the DBMS has told somebody that it committed.
  - No changes are durable if the txn aborted.

**Undo vs. Redo**

- **Undo**: The process of removing the effects of an incomplete or aborted txn.
- **Redo**: The process of re-instating the effects of a committed txn for durability.

- How the DBMS supports this functionality depends on how it manages the buffer pool…

**Buffer Pool Management**

- Whether the DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage.
  - **STEAL**: Is allowed.
  - **NO-STEAL**: Is not allowed.

**Buffer Pool – Steal Policy**

- Whether the DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage.
  - **STEAL**: Is allowed.
  - **NO-STEAL**: Is not allowed.
Buffer Pool – Force Policy

• Whether the DBMS ensures that all updates made by atxn are reflected on non-volatile storage before the txnis allowed to commit:
  – **FORCE**: Is enforced.
  – **NO-FORCE**: Is not enforced.

• Force writes makes it easier to recover but results in poor runtime performance.

NO-STEAL + FORCE

• This approach is the easiest to implement:
  – Never have to undo changes of an aborted txnbecause the changes were not written to disk.
  – Never have to redo changes of a committed txnbecause all the changes are guaranteed to be written to disk at commit time.

• But this will be slow…

Today’s Class

• Overview
• Write-Ahead Log
• Checkpoints
• Logging Schemes
• Shadow Paging
• Examples
Write-Ahead Log

• Record the changes made to the database in a log before the change is made.
  – Assume that the log is on stable storage.
  – Log contains sufficient information to perform the necessary undo and redo actions to restore the database after a crash.
• Buffer Pool: STEAL + NO-FORCE

Write-Ahead Log Protocol

• All log records pertaining to an updated page are written to non-volatile storage before the page itself is allowed to be overwritten in non-volatile storage.
• A txn is not considered committed until all its log records have been written to stable storage.

Log record format:
– <txnId, objectId, beforeValue, afterValue>
  – Each transaction writes a log record first, before doing the change.
  – Write a <BEGIN> record to mark txn starting point.
• When a txn finishes, the DBMS will:
  – Write a <COMMIT> record on the log
  – Make sure that all log records are flushed before it returns an acknowledgement to application.

Write-Ahead Log – Example

The result is deemed safe to return to app.
**WAL – Implementation Details**

- When should we write log entries to disk?
  - When the transaction commits.
  - Can use **group commit** to batch multiple log flushed together to amortize overhead.

- When should we write dirty records to disk?
  - Every time the txn executes an update?
  - Once when the txn commits?

**WAL – Deferred Updates**

- Observation: If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don’t need to store their original values.

```
WAL
<T1 begin>
<T1, A, 99, 88>
<T1, B, 5, 10>
<T1 commit>
```

- Observation: If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don’t need to store their original values.

```
WAL
<T1 begin>
<T1, A, 88>
<T1, B, 10>
<T1 commit>
CRASH!
```

- This won’t work if the change set of a txn is larger than the amount of memory available.
  - Example: Update all salaries by 5%

- The DBMS cannot undo changes for an aborted txn if it doesn’t have the original values in the log.

- We need to use the **STEAL** policy.
WAL – Buffer Pool Policies

<table>
<thead>
<tr>
<th></th>
<th>NO-STEAL</th>
<th>STEAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO-FORCE</td>
<td>Fastest</td>
<td>Slowest</td>
</tr>
<tr>
<td>FORCE</td>
<td>Slowest</td>
<td>Fastest</td>
</tr>
</tbody>
</table>

Runtime Performance

Almost every DBMS uses NO-FORCE + STEAL

Today’s Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Shadow Paging
- Examples

Checkpoints

- The WAL will grow forever.
- After a crash, the DBMS has to replay the entire log which will take a long time.
- The DBMS periodically takes a **checkpoint** where it flushes all buffers out to disk.

Checkpoints

- Output onto stable storage all log records currently residing in main memory.
- Output to the disk all modified blocks.
- Write a `<CHECKPOINT>` entry to the log and flush to stable storage.
Checkpoints

- Any txn that committed before the checkpoint is ignored (T1).
- T2 + T3 did not commit before the last checkpoint.
  - Need to **redo** T2 because it committed after checkpoint.
  - Need to **undo** T3 because it did not commit before the crash.

Checkpoints – Challenges

- We have to stall all txns when take a checkpoint to ensure a consistent snapshot.
- Scanning the log to find uncommitted can take a long time.
- Not obvious how often the DBMS should take a checkpoint.

Checkpoints – Frequency

- Checkpointing too often causes the runtime performance to degrade.
  - System spends too much time flushing buffers.
- But waiting a long time is just as bad:
  - The checkpoint will be large and slow.
  - Makes recovery time much longer.

Today’s Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Shadow Paging
- Examples
Logging Schemes

- **Physical Logging**: Record the changes made to a specific location in the database.
  - Example: Position of a record in a page.
- **Logical Logging**: Record the high-level operations executed by txns.
  - Example: The `UPDATE`, `DELETE`, and `INSERT` queries invoked by a txn.

Physiological Logging

- Hybrid approach where log records target a single page but do not specify data organization of the page.
- This is the most popular approach.

Physical vs. Logical Logging

- Logical logging requires less data written in each log record than physical logging.
- Difficult to implement recovery with logical logging if you have concurrent txns.
  - Hard to determine which parts of the database may have been modified by a query before crash.
  - Also takes longer to recover because you must re-execute every txn all over again.
Shadow Paging

- Database is a tree whose root is a single disk block.
- There are two copies of the tree, the master and shadow
  - The root points to the master copy.
  - Updates are applied to the shadow copy.

Shadow Paging – Example

- Maintain two separate copies of the database (master, shadow)
- Updates are only made in the shadow copy.
- When a txn commits, atomically switch the shadow to become the new master.
- Buffer Pool: NO-STEAL + FORCE
Shadow Paging

- To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow:
  - Before overwriting the root, none of the transaction’s updates are part of the disk-resident database
  - After overwriting the root, all of the transaction’s updates are part of the disk-resident database.

Shadow Paging – Undo/Redo

- Supporting rollbacks and recovery is easy.
- **Undo:**
  - Simply remove the shadow pages. Leave the master and the DB root pointer alone.
- **Redo:**
  - Not needed at all.

Shadow Paging – Advantages

- No overhead of writing log records.
- Recovery is trivial.
Shadow Paging – Disadvantages

• Copying the entire page table is expensive:
  – Use a page table structured like a B+tree
  – No need to copy entire tree, only need to copy
    paths in the tree that lead to updated leaf nodes
• Commit overhead is high:
  – Data gets fragmented.
  – Need garbage collection.

Today’s Class

• Overview
• Write-Ahead Log
• Checkpoints
• Logging Schemes
• Shadow Paging
• Examples

Observation #1

• You can only safely write a single page to
  non-volatile storage at a time.
  – Linux Default: 4KB
• How does a DBMS make sure that large
  updates are safely written?

MySQL – Doublewrite Buffer

• When MySQL flushes dirty records from its
  buffer, it first writes them out sequentially
  to a doublewrite buffer and then fsyncs.
• If this is successful, then it can safely write
  records at their real location.
• On recovery, check whether the doublewrite
  buffer matches the record’s real location.
  – If not, then restore from doublewrite buffer.
Observation #2

- With a WAL, the DBMS has to write each update to stable storage at least twice:
  - Once in the log.
  - And again in the primary storage.
- The total amount of data per update depends on implementation (e.g., physical vs. logical).

Storing BLOBs in the Database

- Every time you change a BLOB field you have to store the before/after image in WAL.
- Don’t store large files in your database!
- Put the file on the filesystem and store a URI in the database.
- More information:
  - To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?

Log-Structured Merge Trees

- No primary storage.
- The log is the database.
  - All writes create just one log entry (fast!).
  - All reads must search the log backwards to find the last value written for the target key.
- DBMS still must periodically take a checkpoint:
  - Log compaction instead of flushing buffers.

LevelDB – LSM

- Google’s fast storage library that provides an ordered mapping of key/value pairs.
  - **MemTable**: In-memory index of log entries
  - **SSTable**: Immutable MemTables on disk.
Observation #3

- All of this has assumed that the database is stored on slow disks (HDD, SDD).
- What kind of logging should we use if the database is stored entirely in main memory?

VoltDB – Command Logging

- Even more lightweight version of logical logging based on stored procedures.
  - `<txnId, ProcedureName, Parameters>`

Command

```
UpdateAcct
UpdateUser:
UPDATE userAct
FROM votes
WHERE id = 7;
InsertHistory:
INSERT INTO history
VALUES (7, 7, 7);
run(acctId) {
  result = executeUpdateUser, acctId; 
  executeInsertHistory, acctId, value, time; 
  return (SUCCESS); 
}
```

Command vs. Physiological

- **Runtime Performance** (Higher is Better)
- **Recovery Time** (Lower is Better)

Summary

- Write-Ahead Log to handle loss of volatile storage.
- Use incremental updates (i.e., STEAL, NO-FORCE) with checkpoints.
- On recovery: undo uncommitted txns + redo committed txns.