Lecture 21: Logging Schemes

15-445/645 Database Systems (Fall 2017)
Carnegie Mellon University
Prof. Andy Pavlo

Crash Recovery

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures. DBMS is divided into different components based on the underlying storage device. We must also classify the different types of failures that the DBMS needs to handle.

Every recovery algorithm has two parts:

1. Actions during normal transaction processing to ensure that the DBMS can recover from a failure.
2. Actions after a failure to to recover the database to a state that ensures atomicity, consistency, and durability.

The key primitives that we are going to use in a recovery algorithm are UNDO and REDO. Not all algorithms use both of these:

1. **UNDO**: The process of removing the effects of an incomplete or aborted transaction.
2. **REDO**: The process of re-instating the effects of a committed transaction for durability.

Storage Types

1. **Volatile Storage**
   (a) Data does not persist after power is cut.
   (b) Examples: DRAM, SRAM,.
2. **Non-Volatile Storage**
   (a) Data persists after losing power.
   (b) Examples: HDD, SDD.
3. **Stable Storage**
   (a) A non-existent form of non-volatile storage that survives all possible failures scenarios.
   (b) Use multiple storage devices to approximate.

Failure Classification

1. **Type #1: Transaction Failures**
   (a) **Logical Errors:** A transaction cannot complete due to some internal error condition (e.g., integrity, constraint violation).
   (b) **Internal State Errors:** The DBMS must terminate an active transaction due to an error condition (e.g. deadlock)
2. **Type #2: System Failures**
   (a) **Software Failure:** There is a problem with the DBMS implementation (e.g. uncaught divide-by-zero exception) and the system has to halt.
   (b) **Hardware Failure:** The computer hosting the DBMS crashes. We assume that non-volatile storage contents are not corrupted by system crash.
3. **Type #3: Storage Media Failure**
   (a) **Non-Repairable Hardware Failure:** A head crash or similar disk failure destroys all or parts of non-volatile storage. Destruction is assumed to be detectable. No DBMS can recover from this. Database must be restored from archived version

Buffer Pool Management Policies

Steal Policy: Whether the DBMS allows an uncommitted transaction to overwrite the most recent committed value of an object in non-volatile storage (can a transaction write uncommitted changes to disk).

1. **STEAL**: is allowed
2. **NO-STEAL**: is not allowed.

Force Policy: Whether the dbms ensures that all updates made by a transaction are reflected on non-volatile storage before the transaction is allowed to commit

1. **FORCE**: Is enforced
2. **NO-FORCE**: Is not enforced

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

Easiest System to implement: NO-STEAL + FORCE

1. The DBMS never has to undo changes of an aborted transaction because the changes were not written to disk.
2. It also never has to redo changes of a committed transaction because all the changes are guaranteed to be written to disk at committed.
3. **LIMITATION: If all of the data that a transaction needs to modify doesn't fit on memory, then that transaction cannot execute because the DBMS is not allowed to write out dirty pages to disk before the transaction commits.**

Shadow Paging

The DBMS maintains two separate copies of the database (**master, shadow**). Updates are only made in the shadow copy. When a transaction commits, atomically switch the shadow to become the new master. This is an example of a NO-STEAL + FORCE system.

Disadvantages: Copying the entire page table is expensive and the commit overhead is high.

Implementation:

1. Organize the database pages in a tree structure where the root is a single disk page.
2. There are two copies of the tree, the master and the shadow:
   (a) The root points to the master copy
   (b) Updates are applied to the shadow copy
3. To install updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow.
   (a) Before overwriting the root, none of the transactions updates are part of the disk-resident database.
   (b) After overwriting the root, all of the transactions updates are part of the disk resident database.
4. **UNDO:** Remove the shadow pages. Leave master and the DB root pointer alone
5. **REDO:** Not needed at all

Write-Ahead Logging

The DBMS records all the changes made to the database in a log file (on stable storage) before the change is made to a disk page. The log contains sufficient information to perform the necessary undo and redo actions to restore the database after a crash. This is an example of a STEAL + NO-FORCE system.

Almost every DBMS uses write-ahead logging (WAL) because it has the fastest runtime performance. But it's recovery time is slower because it has to replay the log.

Implementation:

1. 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.
2. A transaction is not considered committed until **all** its log records have been written to stable storage.
3. When the transaction starts, write a <**BEGIN**> record to the log for each transaction to mark its starting point.
4. When a transaction finishes, write a <**COMMIT**> record to the log and make sure all log records are flushed before it returns an acknowledgement to the application.
5. Each log entry contains information about the change to a single object:
   (a) Transaction ID.

   (b) Object ID.
   (c) Before Value (used for UNDO).
   (d) After Value (used for REDO).

6. Log entires to disk should be done when transaction commits. You can use group commit to batch multiple log flushes together to amortize overhead.

7. **Deferred Updates**
   (a) If we prevent the DBMS from writing dirty records to disk until the transaction commits, then we don't need to store their original values.
   (b) This wont work if the change set of a transaction is larger than the amount of memory available.
   (c) The DBMS cannot undo changes for an aborted transaction if it doesn't have the original values in the log.
   (d) Thus, this is why the DBMS needs to use the **STEAL** policy.

Checkpoints

The main problem with write-ahead logging is that the log file will grow forever. After a crash, the DBMS has to replay the entire log, which can take a long time if the log file is large. Thus, the DBMS can periodically takes a **checkpoint** where it flushes all buffers out to disk.

It is not obvious how often the DBMS should take a checkpoint. Checkpointing too often causes the runtime performance to degrade. But waiting a long time is just as bad, as recovery time increases.

Blocking Checkpoint Implementation:

1. The DBMS stops accepting new transactions and waits for all active transactions to complete.
2. Flush all log records and dirty blocks currently residing in main memory to stable storage.
3. Write a <**CHECKPOINT**> entry to the log and flush to stable storage.

Logging Schemes

1. **Physical Logging**
   (a) Record the changes made to a specific location in the database
   (b) Example: Position of a record in a page

2. **Logical Logging**
   (a) Record the high level operations executed by transactions. Not necessarily restricted to single page. Requires less data written in each log record than physical logging. Difficult to implement recovery with logical logging if you have concurrent transactions in a non-deterministic concurrency control scheme.
   (b) Example: The `UPDATE`, `DELETE`, and `INSERT` queries invoked by a transaction.

3. **Physiological logging**
   (a) Hybrid approach where log records target a single page but do not specify data organization of the page.
   (b) Most commonly used approach.

Conclusion

1. Write-Ahead Logging is most commonly used approach to handle loss of volatile storage
2. Use incremental updates (STEAL + NO-FORCE) with checkpoints
3. On recovery: undo uncommitted transactions + redo committed transactions