Lecture #12: Logging Protocols

15-721 Advanced Database Systems (Spring 2018)
`http://15721.courses.cs.cmu.edu/spring2018/`
Carnegie Mellon University
Prof. Andy Pavlo

1 Database Recovery

Database recovery algorithms are techniques to ensure database **consistency**, transaction **atomicity**, and **durability** despite failures. Recovery Algorithms have two parts:

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

1.1 Logging Schemes

Approach #1 – Physical Logging:

- Record the changes made to a specific record in the database.
- Example: Store the original value and the after value for an attribute that is changed by a query.

Approach #2 – Logical Logging:

- Record the high-level operations executed by transactions.
- Example: The `UPDATE`, `DELETE` and `INSERT` queries invoked by a transaction.

Logical logging writes less data in each log record than physical logging. Difficult to implement recovery with logical logging if you have concurrent transactions. Its hard to determine which parts of the database may have been modified by a query before crash and it takes longer to recover because you have to re-execute every transaction over again.

2 Disk-Oriented Logging and Recovery

The "gold standard" for physical logging and recovery in a disk-oriented DBMS is **ARIES** (Algorithms for Recovery and Isolation Exploiting Semantics) [3]. ARIES was invented by IBM Research in the early 1990s for **IBM DB2**. It relies on STEAL and NO-FORCE buffer pool management policies.

2.1 Main Ideas

Write-Ahead Logging:

- Any change is recorded in log on stable storage before the database change is written to disk.
- Each log record is assigned a unique identifier (LSN).

Repeating History During Redo:

- On restart, retrace actions and restore database to exact state before crash.

Logging Changes During Undo:

- Record undo actions to log to ensure action is not repeated in the event of repeated failures.

2.2 Recovery Phases

Phase 1: Analysis

- Read the WAL to identify dirty pages in the buffer pool and active transactions at the time of the crash.

Phase 2: Redo

- Repeat all actions starting from an appropriate point in the log.
- Log redo operations in case of crash during recovery.

Phase 3: Undo

- Reverse the actions of transactions that did not commit before the crash.

2.3 Log Sequence Numbers

Every log record has a globally unique **log sequence numbers** (LSN) that is used to determine the serial order of those records. The DBMS keeps track of various LSNs in both volatile and non-volatile storage to determine the order of almost everything.

2.4 Optimizations

Group Commit:

- Batch together log records from multiple transactions and flush them together with a single `fsync`. Amortizes the cost of I/O over several transactions.
- Logs are flushed either after a timeout or when the buffer gets full.
- Originally developed in **IBM IMS Fastpath** in the 1980s.

Early Lock Release:

- A transaction's locks can be released before its commit record is written to disk as long as it does not return results to the client before becoming durable.
- Other transactions can read data updated by a **pre-committed** transaction become dependent on it and also have to wait for their predecessor's log records to reach disk.

3 In-Memory Database Recovery

Recovery in an in-memory system is slightly easier than in a disk-oriented system because the DBMS does not have to worry about tracking dirty pages in case of a crash during recovery. The DBMS also does not need to store undo records.

It is still stymied by the slow sync time of non-volatile storage. This is why early papers (1980s) on recovery for in-memory DBMSs assume that there is non-volatile memory [1].

4 SiloR – Physical Logging

SiloR is an extended version of the Silo DBMS that supports logging and checkpoints [5]. It uses the epoch based OCC that we discussed previously [4]. It achieves high performance by parallelizing all aspects of logging, checkpointing, and recovery.

4.1 Logging Protocol

- The DBMS assumes that there is one storage device per CPU socket. Each socket has one logger thread dedicated to writing data to that device.

- Worker threads are grouped per CPU socket. As the worker executes a transaction, it creates new log records that contain the values that were written to the database (i.e., REDO).
- Each logger thread maintains a pool of log buffers that are given to its worker threads When a worker's buffer is full, it gives it back to the logger thread to flush to disk and attempts to acquire a new one. If there are no available buffers, then the worker thread stalls.

4.2 Log Files

The logger threads write buffers out to files.

- After 100 epochs, it creates a new file.
- The old file is renamed with a marker indicating the max epoch of records that it contains.

Log Record Format:

- Id of the transaction that modified the record (`TID`).
- A set of value log triples (Table, Key, Value).
- The value can be a list of attribute + value pairs.

4.3 Persistent Epoch

A special logger thread keeps track of the current persistent epoch (`pepoch`). Special log file that maintains the high epoch that is durable across all the loggers. Transactions that executed in epoch e can only release their results when the `pepoch` is durable to non-volatile storage.

4.4 Recovery Protocol

Phase 1: Load Last Checkpoint

- Install the contents of the last checkpoint that was saved into the database.
- All indexes have to be rebuilt.

Phase 2: Replay Log

- Process logs in reverse order to reconcile the latest version of each tuple.
- First Check the `pepoch` file to determine the most recent persistent epoch. Any log record from after the **pepoch** is ignored.
- Log files are then processed from newest to oldest:
  - Value logging is able to be replayed in any order.
  - For each log record, the thread checks to see whether the tuple already exists. If it does not, then it is created with the value. If it does, then the tuple's value is overwritten only if the log `TID` is newer than the tuple's `TID`.

5 VoltDB – Logical Logging

VoltDB uses a variant of logical logging called *command logging* [2]. The DBMS only records the stored procedure invocation:

- Store procedure name.
- Input parameters.
- Additional safety checks to make sure that the DBMS executes the correct version of the stored procedure.

The key issue with command logging is that if the log contains multi-node transactions, then if one node goes down and there are no more replicas, then the entire DBMS has to restart.

Command logging also only works if the DBMS uses a *deterministic* concurrency control scheme. For a given state of the database, the execution of a serial schedule will always put the database in the same new state if the order of transactions is defined before they start executing and the transaction logic is deterministic.

Executing a deterministic transaction on the multiple copies of the same database in the same order provides strongly consistent replicates. This means that DBMS does not need to use an atomic commit protocol (e.g., two-phase commit) to make sure that all of the nodes are in the same state.

5.1 Logging Protocol

- The DBMS logs the transaction command **before** it starts executing once a transaction has been assigned its serial order.
- The node with the transaction's base partition is responsible for writing the log record:
  - Remote partitions do not log anything.
  - Replica nodes have to log just like their master.

5.2 Recovery Protocol

- The DBMS loads in the last complete checkpoint from disk.
- Nodes then re-execute all of the transactions in the log that arrived after the checkpoint started (oldest to newest).
- The amount of time elapsed since the last checkpoint in the log (roughly) determines how long recovery will take. Transactions that are aborted first still have to be executed unless there are hints in the log.

References

[1] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *SIGMOD '87 Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 104–117, Dec 1987. URL `https://dl.acm.org/citation.cfm?id=38730`.

[2] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, March 2014. doi: 10.1109/ICDE.2014.6816685.

[3] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aaries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Datab. Syst*, 17(1):94–162, MArch 1992. URL `https://dl.acm.org/citation.cfm?id=128770`.

[4] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, November 2013. URL `https://dl.acm.org/citation.cfm?id=2522713`.

[5] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI'14 Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 465–477, Oct 2014. URL `https://dl.acm.org/citation.cfm?id=2685085`.