Assignment 3: Failure Recovery

Assigned: Monday, May 13th, 2019
Due: Thursday, May 30th, 2019 @ 11:59PM (Midnight)
Starter Code: https://github.com/stanford-futuredata/cs245-as3-public
Link to class Gradescope: https://www.gradescope.com/courses/43752

Overview

In our lectures on Failure Recovery, we have discussed techniques for ensuring data consistency and fault tolerance in databases. In this assignment, you will use write ahead redo logging combined with persistent storage to implement failure recovery for a simple key-value store. The result will be a database with atomic transactions that can resume after a crash while guaranteeing external consistency. You will also implement log truncation to reduce the amount of overhead required for recovery after a crash.

We have implemented the write-ahead log as well as a persistent storage engine for you, and you should not modify these implementations. We have also started you off with a transaction manager that ensures transactions are atomic. Your modifications in this assignment will add durability to the transaction manager.

For the purpose of testing your solution, we will simulate persistence by keeping the "persisted" data in the log and the storage engine in-memory. Unpersisted data and the transaction manager are both destroyed during a simulated crash. The test code will reinitialize the transaction manager you implement, which should restore any committed writes from what data was persisted.

Setup

Software Dependencies

- This assignment is once again in Java. Java 8 should be sufficient, the starter code should open readily in your IDE of choice such as IntelliJ or Eclipse.
- To run tests locally, you will need JUnit 4.

Part I (70% of your grade): Implementing durable transactions

For this part, you will be exclusively modifying `TransactionManager.java` (in `src/cs245/as3`) in order to implement fault tolerance. The starting code we have provided exposes the following API, and you should preserve this API and all its properties:

- `start(long txID)`
    - Indicates the start of a new transaction. We will guarantee that txID always increases (even across crashes)

- `read(long txID, long key)`
    - Returns the latest committed value for a key by any transaction.
- `write(long txID, long key, byte[] value)`
    - Indicates a write to the database. Note that such writes should not be visible to `read()` calls until the transaction making the write commits. For simplicity, we will not make reads to this same key from txID itself after we make a write to the key.
- `commit(long txID)`
    - Commits a transaction, and makes its writes visible to subsequent read operations.
- `abort(long txID)`
    - Aborts a transaction.
- `writePersisted(long key, long tag, byte[] value)`
    - You can ignore this call for now; we will describe it in Part 2 of the assignment.

Additionally, you must implement one method, which is the initialization and recovery API for TransactionManager:

- `initAndRecover(LogManager lm, StorageManager sm)`
    - Initializes your transaction manager with the provided log manager and storage manager. We will call this method during testing with our own instantiations of LogManager and StorageManager. We guarantee that this method will be called before any other calls to the methods of TransactionManager.

As a first step, we recommend going through TransactionManager and understanding how it currently implements atomic transactions. Note that reads should always return the latest-committed value for a particular key and writes are always committed atomically, assuming there is no crash. You should **not attempt to validate read or writesets during commit** to provide a different transactional isolation level from the the TransactionManager we provide. If you run the JUnit tests, all the transaction tests should succeed but any of the Recovery-related tests should fail.

During part 1 of this assignment, you are adding durability to the transactions provided by TransactionManager. This means that transactions that have successfully committed should remain committed after a crash. You will achieve this by having your TransactionManager implementation call methods in the LogManager and StorageManager classes (which we provide.)

The LogManager and StorageManager classes have different persistence guarantees. The log can only be modified by appending **log records**. Log records are limited size byte arrays with a format that you will design. Appending a log record is atomic and blocks until the append is persisted. The storage manager in contrast persists changes asynchronously, where writes to the same destination are persisted in-order, but writes to different destinations can be persisted out of order. The interface of these are as follows:

- LogManager
    - `appendLogRecord(byte[] record)`

        - Atomically appends and persists record to the end of the log (implying that all previous appends have succeeded).
        - Returns the log length prior to the append.
        - Log records have a maximum length of 128 bytes
    - `getLogEndOffset()`
        - Returns the log offset after the last append.
    - `readLogRecord(int position, int size)`
        - Returns the slice of the log [position, position+size). Throws an `ArrayIndexOutOfBoundsException` if any index in the range is past the end of the log or truncated.
    - `setLogTruncationOffset(int offset)`
        - Durably stores the offset as the current log truncation offset and truncates (deletes) the log up to that point. You can ignore this until part 2 of this assignment.
    - `getLogTruncationOffset()`
        - Returns the current log truncation offset. You can ignore this until part 2 of this assignment.
- StorageManager
    - `queueWrite(long key, long tag, byte[] value)`
        - You should use **log offsets** as your tags, but they are not necessary until part 2 of this assignment.
        - Writes to **different** keys may be persisted out of order with respect to the order of calls to `queueWrite`, but writes to the **same** key will be persisted in order with respect to `queueWrite`. Conceptually, you can think of this as each key having a separate queue from which writes are drained in order, but with no guarantees on ordering between queues.
        - The storage manager will invoke the `writePersisted()` callback on your transaction manager whenever each write is persisted, in order of their persistence. You can ignore this until part 2 of this assignment.
    - `Map<Long, TaggedValue> readStoredTable()`
        - Returns the mapping of keys to value persisted before the last crash. Returns an empty map for the first initialization of the database.

You should consider how you can use the interface of the LogManager and the StorageManager to store writes. You will need to design a serialization format for storing writes in the form of log records. We recommend **buffering writes until commit**, i.e. only writing changes to the LogManager and StorageManager during commit. We also recommend using **redo logging**, that is, include the newly written value in your log record format (in contrast to undo logging, where the prior value is written to the log.) Because appends have a maximum length, you will need to consider how to split a transactions' writes across multiple records (see the other important details about workloads below - you should never need to split an individual `write()` call across records, for example.)

Tip: Use these classes to serialize objects into byte arrays.

- ByteBuffer
  https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html
- ByteArrayOutputStream
  https://docs.oracle.com/javase/8/docs/api/java/io/ByteArrayOutputStream.html

Tip: Implement a custom Record class that can be serialized to `byte[]` and deserialized from `byte[]`.

Finally, you will implement recovery in the `init()` method. At this point all of the recovery correctness tests should succeed, but the recovery performance tests might not.

Important details:

- Data format
  - Keys are longs
  - Log offsets are ints
  - Values are byte arrays.
- Test workloads properties:
  - The total number of write() calls across all transactions in a workload will be at most 1 million. And, each test will start at most 1 million transactions.
  - Each transaction will involve at most 1000 calls to `write()`, and the length of each written value will be at most 100 bytes.
  - The maximum number of concurrent open transactions at any time is 10.
  - We will not make reads to this same key within a transaction with the same txID after we make a write to that key.
  - We limit the log offset to 1Gb, which you should not reach on any of the workloads assuming your record format is not too wasteful.

On the **first** call to `init` within a test, we guarantee that the log and database will be empty. In subsequent calls to `init` after a crash, we guarantee that the log passed to your TransactionManager will be your log, unmodified, from before the crash, and that the database will be in some state consistent with that log. (In short, you should feel free to use custom serialization formats and metadata for the log values, which you will be able to parse at recovery time.)

All methods of LogManager and StorageManager may crash by throwing a RuntimeException. These simulate hard-stop crash failures, such as a power-outage. You **should not catch these** in your implementation of TransactionManager and instead simply allow the test code to handle the exception. You can assume that after any crashes, there will be no further calls to the TransactionManager that had the failed request. Rather, the test code will always construct a fresh instance of TransactionManager, and then enter recovery, before calling any methods of TransactionManager.

You also **must not store state in static members** of TransactionManager that might be used for recovery. The spirit of the assignment is that a fresh TransactionManager comes up after a crash and

resumes transaction processing using only the data persisted in the LogManager and StorageManager. Solutions that don't match this spirit will be penalized.

Part 2 (30% of your grade): Truncating the log and fast recovery

In this part of the assignment, you will truncate the log to reduce the cost of recovery. The StorageManager will call `writePersisted` whenever a queued write is persisted. You will need to implement some way to track which committed writes are not yet stored durably by the StorageManager. From this information you should be able to derive a **safe log truncation point**, one that does not go past the point of any write in the log that has not yet been persisted to the StorageManager. You should then periodically (i.e. in your implementation of `writePersisted`) call `setLogTruncationPoint` on the log manager to truncate the log. On recovery, you should resume recovery from the last log truncation point. Your goal is to minimize the number of times you need to read from the log in order to recover.

At this point, the `recoveryPerformance` test should succeed as well. This test will check that the log is being truncated during operation and that recovery uses a small number of read I/Os to the log.

Submission Instructions

See `README.md`