Mike Calder, Nathan Longnecker
CS542 - Database Management Systems

Project 1 - Values Store

We implemented a values store program. The program exposes a simple interface allowing users to put, get, and remove key-value pairs from the system. The keys are integers, and the values are an array of bytes. The system was designed to follow the ACID paradigm, so each feature we included aimed towards fulfilling Atomicity, Consistency, Isolation, or Durability.

Using our values store program

We wrote our value store program in Java. The program exposes the following interface:

`void Put(int key, byte[] data);` stores data under the given key,

`byte[] Get(int key);` retrieves the data and

`void Remove(int key);` deletes the key.

This interface is visible in our code in the interface IValueStore. The realization of this interface is the ValueStoreImpl class. The system is designed to allow users to create as many instances of the ValueStoreImpl class as they would like.

To use the system, simply create a main method that creates one or more instances of the ValueStoreImpl and calls the methods exposed by the interface. A sample main method is provided in the code we submitted.

Design

Our design was mostly based around Atomicity, Consistency, Isolation, and Durability. By fulfilling these four paradigms, we ensure that our key-value store meets the standards set by a database.

Atomicity
Each action is treated as though it is not started or fully completed. The program uses read/write locks to ensure that other threads in the program will not interrupt each other, and logs the status of each action so that if something interrupts the program then the action can be completed later.

Consistency
The key-value store begins each transaction in a consistent state and ends each action in a consistent state. The program implements a logger to ensure that in the event that the

program terminates during the execution, it will be able to restore itself to a consistent state when it starts up again.

Isolation
The program ensures that each transaction runs in isolation relative to each other transaction. It uses locks on each file to ensure that each file is only being modified by one transaction at a time.

Durability
Actions completed against the system persist because they are written to the disk. Additionally, the recovery log ensures that once an action is committed, it will occur even if the database crashes during the execution of the transaction.

How it works

Creation
When each instance of the ValueStoreImpl is created, the user may specify the directory that the value store is located in. It is recommended to use an empty directory, but you may use any directory that does not contain any directories in it. Many operating systems allow files and directories with the same name in the same folder, which could cause problems for our implementation so it is therefore not allowed. You may also reuse a database directory that was previously the folder for a ValueStoreImpl, making it have the same keys/values the old one did.

Storage
The key-value store stores each key-value pair as a file. The key is the name of the file, and the value is the contents of the file. All files are stored in a flat directory. Using separate files allows us to keep operations on separate key-value pairs isolated while allowing both to occur simultaneously. It also allows us to support a database much larger than the system's memory, because the entire set of key-value pairs does not need to be in memory simultaneously.

Lock Manager
Every ValueStoreImpl has its own "LockManager", which keeps a read/write lock for every file the values store when there is a transaction occurring that involves it. This makes it so that multiple users can be reading a value for the same key at the same time, but if one user is writing to the key then no other user's can read/write to/from it until it is done. This is done on a per-key basis, so the values store is still highly performant. We also prevent memory overhead when storing the locks by deleting any ones that are not currently in use, making the maximum number of locks stored at a time equal to the number of users currently executing transactions on the values store.

Logger

Our program implements a logger to ensure that every transaction that is started will be completed and leave the program in a consistent state.

When an action is started, it first notifies the logger that the action is being started. The logger then records what action is about to be started, and then flushes the log to the disk. The program can then begin the action. Once the action has been completed, it notifies the logger, and the logger notes that that action has completed by removing it from the log. Once an action has been completed, there is no evidence of that action remaining in the log. Therefore, we know that when we start up, any transactions remaining in the log were unfinished. We can then redo the action to ensure that the database will always be in a consistent state.

Testing

We have created a number of JUnit 4 tests to test parts of the application as well as to stress test the application when there are several users and threads attempting to access the data simultaneously. These tests are visible in the test folder of our github repository.