Mike Calder, Nathan Longnecker
CS542 - Database Management Systems

Project 2 - Index Mechanism

We implemented an index mechanism. The program exposes a simple interface allowing users to put, get, and remove key-value pairs from the index. The keys and values are both strings. In the index mechanism, the key represents the logical address of the associated entry in the relation. The value represents the data value for attribute that is being indexed. The system was designed to follow the ACID paradigm, so each feature we included aimed towards fulfilling Atomicity, Consistency, Isolation, or Durability.

Using our index mechanism program

We wrote our index mechanism in Java. The program exposes the following interface:

`void Put(string key, string data_value);` adds the index entry.

`string Get(string data_value);` retrieves the key given the index.

`void Remove(string key);` deletes the index.

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

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

Algorithm used

We used the linear hashing algorithm that was discussed in class. We begin with four empty buckets (all of which have a maximum size of four indices) and start the round robin number at 0. The round robin order therefore begins at three, and we increment that whenever we have cycled through all round robin numbers less than 1 << (order - 1). Whenever we add to a bucket that is filled to capacity, we add an overflow bucket to put the index in, and then we split the bucket at the index of the round robin number.

Assumptions made

We assumed that the index was stored in memory, but we also recognised that it needed to be persistent. To solve this, every time a change is made to the index, it is written to the disk. This way, the index mechanism may be restored in the case of a system failure.

We assumed that multiple users could be accessing the index mechanism at once, so we used locks to prevent two threads from being in critical regions at the same time.

We assumed that the program could crash during execution, so we write all bucket updates to the disk and keep a backup while doing so at all times.

We assumed that Java’s built-in “hashCode” function for strings produces a reasonably even distribution. In the case it doesn’t, calls to that method could easily be replaced with another hash function.

We assumed that the whole index was intended to be kept in memory, so we load the full list of buckets from the disks when recovering from an old index mechanism file.

Design

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

Atomicity
Each action is treated as though it is not started or fully completed. The program uses locks to ensure that other threads in the program will not interrupt each other, and writes the index to the disk after each action so that only a consistent state is recorded.

Consistency
The index mechanism uses locks to ensure that the index is always in a consistent state. The program is only written at the end of each action, so it only persists when the index is in a consistent state. This ensures that in the event that the program terminates during the action, it will be able to restore itself to a consistent state when it starts up again.

Isolation
The program ensures that each action runs in isolation relative to each other action. It uses locks on the buckets to ensure that the buckets are only being modified by one action at a time.

Durability
Actions completed in the system persist because they are written to the disk. Additionally, the recovery system ensures that once an action is completed, it will be saved even if the database crashes after the execution of the action.

How it works

Creation
When each instance of the IndexMechanismImpl is created, the user may specify the directory that the index mechanism 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 an IndexMechanismImpl, which causes the index to be restored to the state of the previous index mechanism.

Persistence
Our program implements a logger to ensure that every addition/deletion is stored on the disk and is recoverable in the case of a program crash or end of use. When any action has been completed, the list of buckets used by the program is written to a file via an Object Writer. The file contains the full index mechanism so that a new instance of IndexMechanismImpl can recover the previously existing buckets when created again.

Synchronization
The index mechanism uses the Object containing its buckets to lock down the index any time its contents are being altered or the round robin number/order is being changed. This ensures that multiple users can access the index mechanism and not overwrite each other or cause race conditions during use.

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.