|  |
| --- |
| http://www.wpi.edu/~vernescu/logo.png |
| Assignment 1 |
| CS 542: Database Management Systems |
|  |
| **Tyler Carroll, Tom Strott and James Silvia** |
| **2/15/2015** |

Contents

[Table of Figures 1](#_Toc412058774)

[1. Introduction 2](#_Toc412058775)

[2. Assumptions and Limitations 2](#_Toc412058776)

[3. Client/Server Model 3](#_Toc412058777)

[4. Relation Class 4](#_Toc412058778)

[5. Isolation Manager 5](#_Toc412058779)

[6. Database Manager 5](#_Toc412058780)

[6.1 Database Table 6](#_Toc412058781)

[6.2 Memory Manager 7](#_Toc412058782)

[6.3 Memory Map 7](#_Toc412058783)

[7. Testing Suite 8](#_Toc412058784)

# Table of Figures

[Figure 1: Database Server Flow 3](#_Toc411772632)

# 1. Introduction

For project 1 our team has implemented a value store database. Source code for this project can be found at <https://github.com/jamesesilvia/CS542>. The value store consists of three interfaces. Interface one stores data under the given key provided by the user. Interface two retrieves the data at a given key. Interface three removes the data at a given key. The interfaces can be seen below:

1. void Put(int key, byte[] data);
2. byte[] Get(int key);
3. void Remove(int key);

Our system can support large sizes of the byte value array, up to 1GB and focuses on correct operation in a multithreaded environment where multiple users can interact with the database at the same time. The following sections will describe our design as well as our testing suite which is included to show the functionality of our project.

# 2. Assumptions and Limitations

The Put command is used to put data in empty keys. A user cannot put data in an in-use key. A Remove command is needed to reuse a key.

Also, data sent from client to server can be up to 1kB typed out, but the size can be larger if it is contained by a file. Note that we only officially support up to 1GB data transfers, but larger files may work. The database only expands up to 2GB.

Reliability was prioritized over performance, and the 1GB transfer time suffers as a result.

Finally, we decided to isolate transactions based on serialization.

# 3. Client/Server Model

Our value store database runs in a client/server model that use socket communication to transfer information. The client is a command line interface for a user that accepts commands “put”, “get”, and “remove” and it also gathers all required information for each command. Once all information is gathered, the client informs the server of the intended users action and waits for a response.

The database represents the server and is made up of several threads. This server is in charge of handling users requests, managing the information in the database, all while maintaining the integrity of the data stored in the database. Databases often receive multiple requests in an extremely short period of time and must prevent these accesses from overwriting or damaging any information.

The first server thread initializes the server and waits for new client connections. On each client connection a new thread will be spawned in the server to handle the user’s requests to put, get, or remove information from the database. These “client handlers” are responsible for determining the intent of the client and forwarding that information to another piece of the server. These threads can be linked to the database through a Relation. The client handler server requests have a 1:1 mapping to the information in the database (one Relation for one Table).

The manager that handles all requests from the client handlers is referred to as the Isolation Manager and is a single thread on the server. It protects the integrity of the database by acting as a traffic cop of user requests. Once the traffic cop allows information to be accessed it initiates a request to the database manager.

This can be summed up in the diagram below. Further details will be explained shortly. Note in this case we only have one table in our database so there is only one isolation manager and one relation.

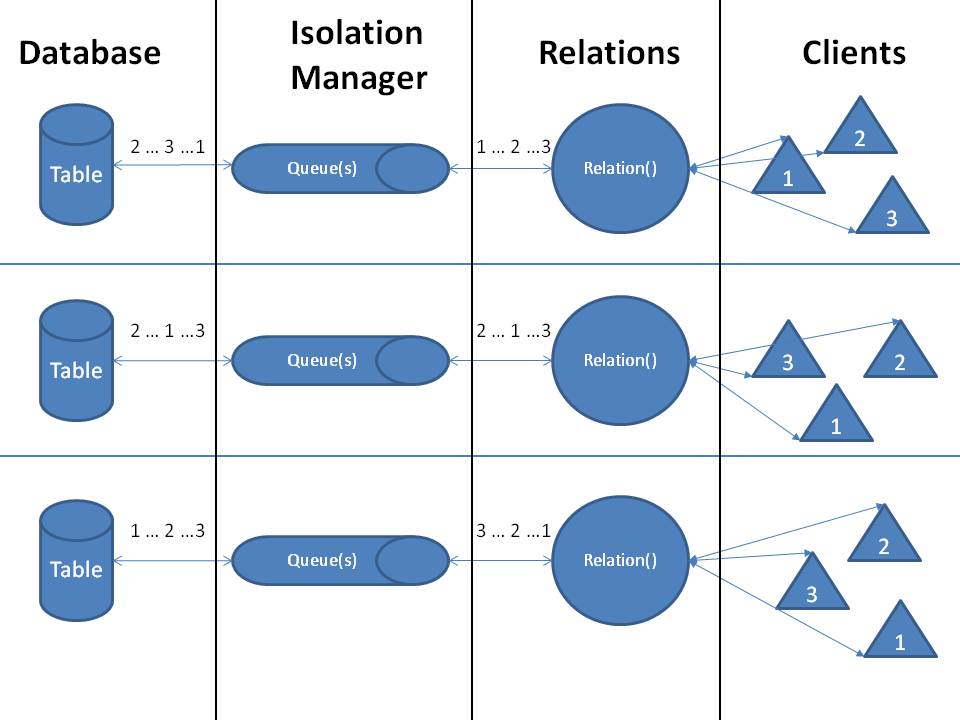


Figure : Database Server Flow

# 4. Relation Class

The relation class contains the interfaces used to access the database. There is a single relation for each table that is represented in the database server. For this specific project, there is one Relation.

Once the server has handled the user’s request, it hands off the request to the Relation call of either put, get, and remove. Because there are multiple clients, and multiple client handler threads on the server, much of the data in the Relation class must be locked using mutexes to protect the processes from stomping all over each other.

The Relation class contains two queues, service\_queue and done\_queue, and is shared across all client handling threads. The queues allow proper request handling when the server is heavily loaded. The Relation class is in charge of placing new requests on to the service queue, removing the same request once complete from the done queue, and forwarding the server response to the correct client.

This abstracted block can ONLY put on to the service queue, and remove from the done queue.

# 5. Isolation Manager

The isolation manager is it’s own thread and is included in the Relation class. But can be abstracted out for this document. The isolation manager removes requests to be serviced, determines the action, and forwards it on to the database manager. Once the database manager has handled the request, the isolation manager puts the request on the done queue for the Relation to return the information to the client.

This abstracted block can ONLY remove from the service queue, and put on to the done queue.

Figure 1 does a good job of describing the full path of the server.

# 6. Database Manager

The database manager is in charge of managing the location of information in the database. The manager has several functionalities, each of which will be described below. The functionalities it has are the database table, memory manager and the memory map.

## 6.1 Database Table

The database table keeps track of all the indices in the database. The database table is a linked list that is ordered by offset in memory. Each member of the list is a structure that contains information about a given index. For each index the structure keeps track of their index number, offset in memory, size in bytes, which fragment of the index this chunk of memory is and has a pointer to the next fragment of that index. The pointers to the next fragment allow the server to quickly find all the fragments for a given index without having to iterate through the entire list multiple times.

Eventually after the user performs several adds and deletes the memory will start to become fragmented. To keep track of this, each index entry in the database table will have a pointer to the next fragment for that index. For example if an index contains 1000 bytes but there are only two empty spaces in memory of 500 bytes each, the database table can show the index starts at location 1 and is 500 bytes long then has a pointer to location 2 which is at a different offset and has another 500 bytes. If a user wants to delete an index then the database table only needs to remove that index and now the memory manager will consider that memory to be available. The database table will also be responsible for reporting to the isolation manager if a given index exists or not.

The database table must be persistent as well or the user will not know what is actually in the database. A separate comma delimited text file gets maintained by the server which replicates all the information in the database table. Every time a change is made to the database table, the change is replicated in the text file backup. On server initialization we load the text file and reform our database table including relinking all the fragments of each index back together with separate pointers.

## 6.2 Memory Manager

The memory manager is responsible for finding open spaces in memory for the isolation manager. The memory manager will scan the database table to be able to find the open areas in memory and then report them to the isolation manager. The memory manager attempts to minimize fragmentation by first looking for a contiguous block of the desired size before going back and returning the next largest block of memory. If that block of memory is not big enough to fit the entire write then the server keeps asking the memory manager for the next largest chunk of memory and writing the information into it until the server completes the write.

Before finding an open space in memory for a write, we check to see if there are enough total bytes open for the write. We don’t begin the write if there are not enough total bytes to start with. If there are not enough total bytes for the write before we start then we expand the database in chunks of 1kB until there is enough room for the write. Then the memory manager finds an open space for the write and completes the operation.

## 6.3 Memory Map

Our database will be contained within several data files. Each table in the database will have its own file to simplify the memory management. However, for project one there will only be one table so only one data file will exist. On the server we will use mmap() to map the entire file into memory. That way we can access individual addresses in the file. The database table will list the exact address and length of the indices in the memory map. The memory map simply provides access to the file. Each write done to the mapped memory will be immediately written down to the actual file.

When we need to expand the database because we have run out of space we first unmap the database from memory, expand the file by adding more space to the end, then remap the database back into memory. The database starts at a size of 1MB and expands in 1kB chunks until it reaches a maximum size of 2GB.

# 7. Testing Suite

There are nine test cases along with three setup shell scripts that should be run first. Optionally, there is also a teardown shell script to delete all of the unnecessary text files. We used Expect for our test automation. It is a scripting tool that waits until a certain known string is displayed in the console and responds accordingly with the next command.

All testing was done on the CCC machines via Putty. Each test case passed, although the 1 GB data transfer takes an exceptionally long time (hours). Table 1 below depicts each test case, a brief description of it, how to use it, and the expected results. See the README file and test scripts in the test directory on Github for additional details.

Table : Test Cases

|  |  |  |  |
| --- | --- | --- | --- |
| **TC#** | **Description** | **Usage** | **Expected Results** |
| 1 | Tests what happens when the Put command is successful | Run TC1PutSuccess.exp | The Put succeeds |
| 2 | Tests what happens when the Put command is unsuccessful | Run TC2PutFailure.exp | The Put fails safely |
| 3 | Tests what happens when the Get command is successful | Run TC3GetSuccess.exp | The Get succeeds |
| 4 | Tests what happens when the Get command is unsuccessful | Run TC4GetFailure.exp | The Get fails safely |
| 5 | Tests what happens when the Remove command is both successful and unsuccessful | Run TC5Remove.exp | After the initial Remove that ensures the key is empty, which is allowed to either succeed or fail, the following occurs in order:   * Put succeeds * Get succeeds * Remove succeeds * Get fails safely * Remove fails safely |
| 6 | Tests the Put command from file, to allow for larger data sizes | Run CreateSmallInput.sh if you haven’t already, then run TC6InputFile.exp | Client Get yields “This is data.” |
| 7 | Tests the Put and Get command from file, to allow for larger data stores and retrieves | Run CreateSmallInput.sh if you haven’t already, then run TC7InputFileOutputFile.exp | BasicInputFile.txt and BasicOutputFile.txt both contain “This is data.” |
| 8 | Tests the Put and Get command with a 1GB file with a random amount of data | Run CreateGiantInput.sh if you haven’t already, then run TC8GiantFile.exp | GiantInputFile.txt and GiantOutputFile.txt are roughly 1GB with matching data |
| 9 | Tests scenarios when the server is receiving commands from three different clients at the same time | Run CreateMediumInput.sh if you haven’t already, then run TC9Client1.exp, TC9Client2.exp, and TC9Client3.exp concurrently | Determine if server/client output match, and the data is handled properly based on the order that the commands are received by the server. |
| 10 | Tests whether the database persists even when all clients and servers are cancelled | This is a manual test.   1. Start server on Putty session 2. Start client on Putty session 3. Remove data at key 10 if it exists 4. Put data “This is data.” at key 10 5. Close both Putty sessions 6. Start new client and server Putty sessions 7. Get data at key 10 | “This is data.” retrieved from key 10. |