|  |
| --- |
| http://www.wpi.edu/~vernescu/logo.png |
| Assignment 2 |
| CS 542: Database Management Systems |
|  |
| **Tyler Carroll, Tom Strott and James Silvia** |
| **3/3/2015** |

Contents

[Table of Figures 1](#_Toc413238413)

[1. Introduction 2](#_Toc413238414)

[2. Assumptions and Limitations 2](#_Toc413238415)

[3. Client/Server Model 3](#_Toc413238416)

[4. Relation Class 4](#_Toc413238417)

[5. Isolation Manager 5](#_Toc413238418)

[6. Database Manager 6](#_Toc413238419)

[6.1 Database Table 6](#_Toc413238420)

[6.2 Memory Manager 6](#_Toc413238421)

[6.3 Memory Map 7](#_Toc413238422)

[7. B+ Trees 8](#_Toc413238423)

[8. Testing Suite 8](#_Toc413238424)

[References 11](#_Toc413238425)

# Table of Figures

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

# 1. Introduction

For project 2 our team has implemented an indexing mechanism on top of last project’s value store database. Source code for this project can be found at <https://github.com/jamesesilvia/CS542>. The indexing mechanism consists of four interfaces. Interface one stores a data record, which consists of a related pair of number and string data provided by the user with a server-assigned key. Interface two retrieves all data records with string\_data matching the specified string\_data . Interface three retrieves all data records with number\_data matching the specified number\_data. Interface four removes a data record. The interfaces can be seen below:

1. void Put(int number\_data, string string\_data);
2. string Get(string string\_data);
3. int Get (int number\_data);
4. void Remove(int key);

Our system can support multiple records having the same string\_data, number\_data, or both while maintaining a unique key 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 server maintains the unique identifiers as new records are added to the database. There is no reason for the user to specify these identifiers when the purpose of the put command is to put meaningful data in the database. Furthermore, the put command an associated number and string into the database at the same time. The team modeled this as a population and a city name as defined in the project three description.

Once again, the isolation level is serialized transactions.

# 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\_index\_by\_name”, “get\_index\_by\_population” and “remove” and it also gathers all required information for each command. Once all information is gathered, the client informs the server of the user’s intended 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 project, much like the last one, we only have one table in our database so there is only one isolation manager and one relation.

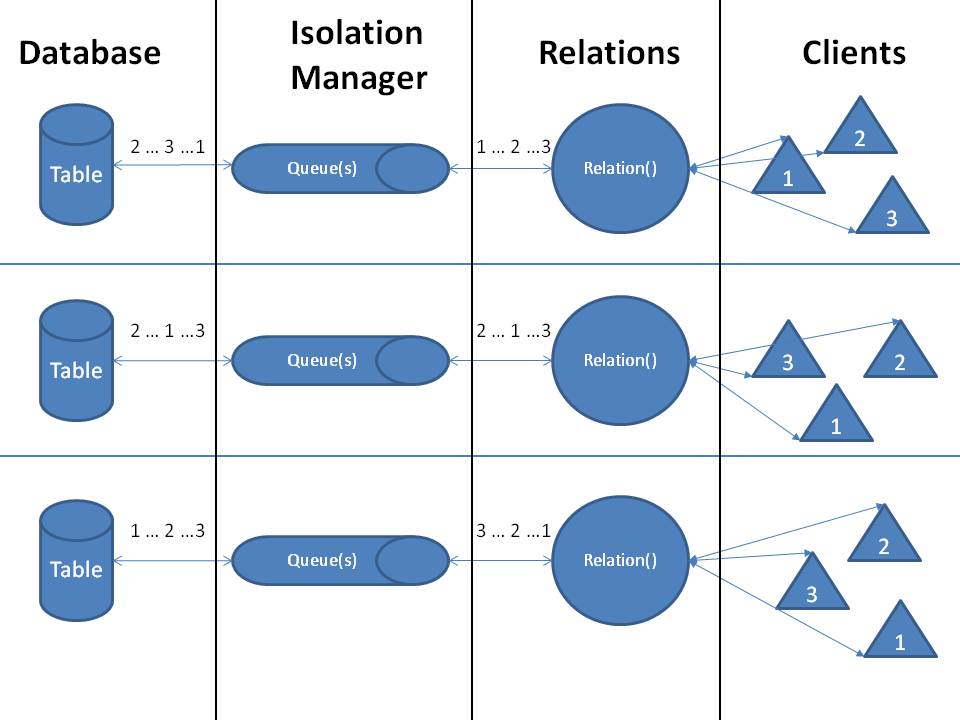


Figure 1: Database Server Flow

# 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 exclusively put on to the service queue and remove from the done queue.

# 5. Isolation Manager

The isolation manager is its own thread and is included in the Relation class. However, it 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 exclusively 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.

This has been drastically simplified for Project 2.

## 6.1 Database Table

The database table keeps track of all the indices in the database. The database table is a list that is ordered by ID, where the ID is unique. The ID cannot be reused unless a table entry has been removed from the database entirely. The unique ID links a table entry to a specific, physical storage location in the table’s database.

## 6.2 Memory Manager

All data that is stored in the database fits inside of a predetermined container. For this project, the container is made up of the unique ID discussed above and the corresponding value and string. For this application, it can be described as an ID, a population value, and a location string. The container will not shrink or grow based on what it is stored because it is a \_\_packed\_\_ structure and the sizes are limited on input. This simplifies the relationship between the unique ID and its corresponding physical location in storage.

The memory manager is responsible for physically placing the above container in to the database, therefore securing that the data can be recovered. If for any reason the database needs to be expanded (size increase) it handles that as well. The isolation manager will request to the memory manager the request of storing, retrieving, or removing information stored in the database.

The unique ID (UID) that information is stored keyed as links the physical storage location simply by the size of the container previously discussed. A quick example…

* Table Entry UID: 0. Storage Offset: 0
* Table Entry UID: 1. Storage Offset: 109
* Table Entry UID: 2. Storage Offset: 218

If a request is received to place more information in to the database, the memory manager will determine the next available UID, and store the information in that location. If a request to remove is received, the entry is cleared from the database table; which now opens the physical location and UID for a new value. Similarly on a get, the memory manager will extract the relevant information.

B+ Tree’s have been implemented as a searching algorithm to determine what information is stored inside of the database. Without giving too much information, they maintain lists of each table in a specific order for improved database access. They direct the memory manager to the correct UID. More information is available in Section 7.

## 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 offsets based on UID’s in the file Each write done to the mapped memory will be immediately written down to the actual file.

The database starts at a size of 1MB and expands in 1kB chunks until it reaches a maximum size of 2GB.

# 7. B+ Trees

B+ trees were implemented to maintain the physical storage location of a given index based on the data that is contained at the location. They are extremely efficient at storing data in block levels (our container). Due to the level of complexity associated with B+ Trees, a licensed, open sourced example has been modified to fit our application1. The modifications are as follows:

1. Conversion to C++ class from C code
2. Multiple data entries per key support
3. Creation of a separate class of B+ Tree that uses strings as the key and creates an alphabetically sorted B+ tree.

Within our application, there is a 1:1 relation between the number of B+ Tree’s and the data that we want to index. For example, Project 2 supports table entries that are made up of a population and location. There will be a B+ tree that is indexing on the integer population value, and a separate B+ tree that indexes on the string location value.

# 8. Testing Suite

There are eight test cases. Between each test case, the database text file should be deleted. This can be done either manually or with a provided shell script in the test directory. Note that the server maintains a database.dat file while the server is running, so this shell script must be used while the database is not running. Put is never tested individually, but the Get tests prove that Put must also be working because we start with a fresh database for every test. The final test case ensures that data continues to persist after the system is killed and restarted, and it is imperative that the database is not deleted between running the two scripts. It is not fully automated.

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 except for TC7, although a less rigorous manual multiclient operation does work. Table 1 below depicts each test case, a brief description of it, how to use it, and an overview of the expected results. See the test scripts and README file in the test directory on Github for additional details.

Table 1: Test Cases

|  |  |  |  |
| --- | --- | --- | --- |
| **TC#** | **Description** | **Usage** | **Expected Results** |
| 1 | Tests various scenarios when the Get\_index\_by\_population command is successful | Delete build/database.txt, then run TC1PutGetByPop.exp | Put succeeds,  Get\_index\_by\_population succeeds when there is one entry with that population, Get\_index\_by\_population succeeds when there are multiple entries with that population |
| 2 | Tests various scenarios when the Get\_index\_by\_name command is successful | Delete build/database.txt, then run TC1PutGetByName.exp | Put succeeds,  Get\_index\_by\_name succeeds when there is one entry with that name, Get\_index\_by\_name succeeds when there are multiple entries with that name |
| 3 | Tests what happens when the both Get commands are unsuccessful | Delete build/database.txt, then run TC3GetFail.exp | Both Gets fail safely |
| 4 | Tests various scenarios when the Remove command is successful | Delete build/database.txt, then run TC4Remove.exp | Put succeeds, the following Get succeeds, the following Remove succeeds, the following Get fails because the data was removed, the series of Puts succeeds, the two Removes succeed, and the Get yields seven results with the three specifically deleted cities missing |
| 5 | Tests what happens when the Remove command is unsuccessful | Delete build/database.txt, then run TC5RemoveFail.exp | Remove fails safely |
| 6 | Tests 1000 rapid Put commands and then Gets them all | Delete build/database.txt, then run TC6PutManyRecords.exp | All 1000 Put commands succeed and the Get returns all 1000 results |
| 7 | Tests 1000 rapid Put commands and then Gets them all from two clients simultaneously | Delete build/database.txt, then run TC6PutManyRecords.exp and TC7PutManyRecords.exp from two clients simultaneously | All 2000 Put commands succeed and the second client’s Get returns all 2000 results |
| 8 | Regression: Tests whether the database persists even when all clients and servers are killed | This is a mostly manual test.   1. Delete database.txt 2. Start server on Putty session 3. Start client on Putty session 4. Run TC1PutGetByPop.exp 5. Close client and server 6. Start client and server 7. RunTC8PutGetByPop.exp | After TC8PutGetByPop.exp is run, we expect double of the data we expected for TC1PutGetByPop.exp at each step |

# References

1. GPL Licensed B+ Tree Source Code. Accessed: 3/1/2015.

<http://www.amittai.com/prose/bpt.c>