|  |
| --- |
| http://www.wpi.edu/~vernescu/logo.png |
| Assignment 4 |
| CS 542: Database Management Systems |
|  |
| **Tyler Carroll, Tom Strott and James Silvia** |
| **4/8/2015** |

Contents

[Table of Figures 1](#_Toc416801717)

[1. Introduction 3](#_Toc416801718)

[2. Assumptions and Limitations 3](#_Toc416801719)

[3. Client/Server Model 4](#_Toc416801720)

[3.1 Database Initialization 6](#_Toc416801721)

[4. Relation Class 7](#_Toc416801722)

[4.1 Open 8](#_Toc416801723)

[4.2 GetNext 8](#_Toc416801724)

[4.3 Close 8](#_Toc416801725)

[5. Logger Class 9](#_Toc416801726)

[5.1 Start Commit 9](#_Toc416801727)

[5.2 Update Entry 9](#_Toc416801728)

[5.3 End Commit 9](#_Toc416801729)

[6. Isolation Manager 10](#_Toc416801730)

[6.1 Population Changes 10](#_Toc416801731)

[6.2 Restore Log 10](#_Toc416801732)

[7. Memory Manager Class 11](#_Toc416801733)

[7.1 Database Table 11](#_Toc416801734)

[7.2 Memory Manager 11](#_Toc416801735)

[7.3 Memory Map 13](#_Toc416801736)

[8. B+ Trees 13](#_Toc416801737)

[9. Test 14](#_Toc416801738)

# Table of Figures

[Figure 1: Database Server Flow 5](#_Toc416771142)

# 1. Introduction

For project 4 our team has implemented an undo/redo logger on top of last project’s query executioner with B+ tree indexing. Source code for this project can be found at <https://github.com/jamesesilvia/CS542>. The undo/redo logger consists of two interfaces. The first interface named “Update” increases the population of all records in the database by a specific percentage that accepts an integer value as a percentage. The second interface named “Import” allows the user to specify a log file that the database will parse and apply all changes in the log file to the current database. Import does not need arguments because the log files have names given by the database based on the table they are associated with.

1. Update(int percentage);
2. Import();

The logging has been implemented by a new Logger class that gives the database the ability to start a transaction, add changes to the database and then issue a commit at the end. We are able to update a database by importing another log file and parsing it, then performing each change specific in the log file.

Most of the work in Project 4 has been adding the Logger class and providing the functionality to increase the population values of all entries in both the city and country tables in the database.

# 2. Assumptions and Limitations

The schema used for this project was from the world project that is used for MySQL training. We ported this file to SQL before getting started but no data was impacted by the port. The changes mostly consisted of removing quotes around database objects and converting the MySQL enums to a SQL CHECK statement with a default value.

Our application will load all data entries included in the city and country table in to two separate databases on initialization. The data remains persistent on application exit. We do not allow users to add, remove, or edit any of the data stored in the database directly but we do give the users the ability to increase or decrease the values of all populations of a given integer percentage between 1% and 10%. We assumed that both tables should be impacted by this change, and that as city population changes by a certain percentage, the overall country population also changes by the same percentage. We also expect that the schema files used for initialization and the database files are not tampered with during execution. Our code assumes that each entry in the city table has a country code, and each country code only appears in one country table entry.

We provided the ability to increase or decrease the population by varying percentages to show the modularity and power of our application. We left in the query execution engine so the user could see the effects of the query at varying percentages with different population changes.

# 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 “query”, “update-population”, “restore-log” and “quit”. “Query” will prompt for a percentage value from 0 to 100, while “update-population” will prompt for a percentage value from 1 to 10 to increase or decrease the population of each city and country. The user may input negative integers to decrease the populations. “Restore-log” assumes that the log files have not been moved since they were created. If “restore-log” is used on a different machine, it is necessary to put the log files in the proper directory location.

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.

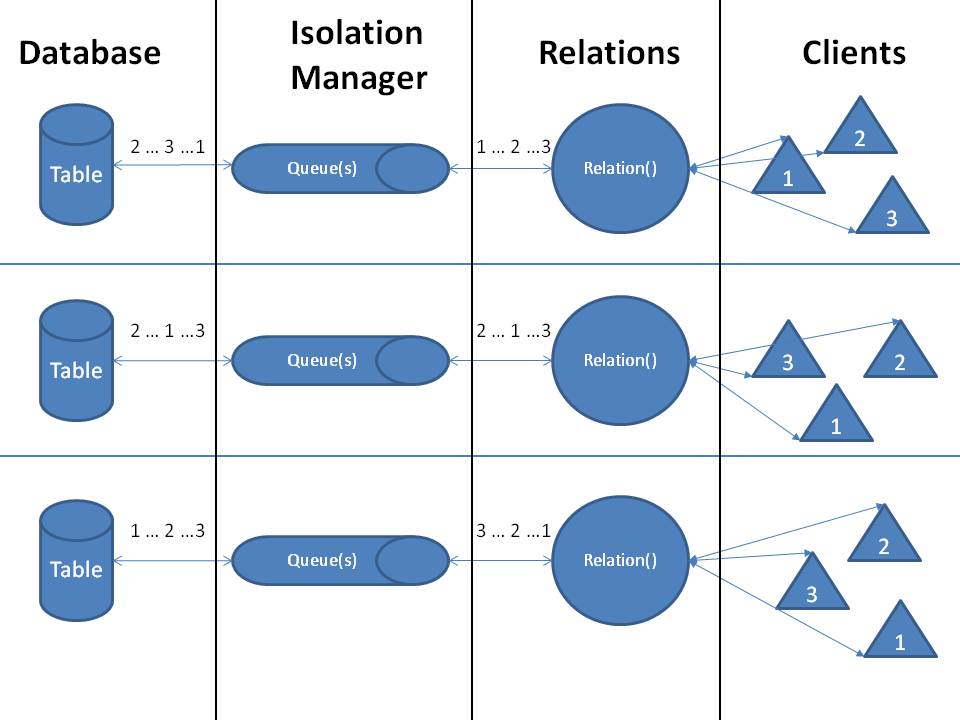


Figure 1: Database Server Flow

## 3.1 Database Initialization

On first execution of the database server two “.csv” files are parsed that contain the information from the MySQL database. Each table entry is stored in the database and indexed in our B-Tree through the Relation class. This information is persistent on each server shutdown/startup; but if either database file is removed; the initialization will take place again. It can be lengthy because we are loading thousands of rows in to our database.

If the database has already been loaded, the only initialization required on startup is re-building the B+ Trees to gain access to the information stored in the database. The initialization makes use of Open, and Close in the Relation. More details in the sections that follow…

# 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 are two instances of a Relation.

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.

For project 3 we altered how the Relation class works and how it interacts with the Memory Manager class and we carried these changes onto Project 4. In previous projects the Memory Manager class had a single global instance that could be accessed from anywhere because there was only one table in the database. Now that we have multiple tables and there exists both a Relation object and Memory Manager object for each table, we decided to get rid of the global Memory Manager object. We added a Memory Manager object to the Relation class so each Relation object would have its own Memory Manager object that it can use to interact directly with its own table. This also allowed us to let the Relation class have more control over the Memory Manager. We added three new functions to the Relation class for this project, open, getNext and close, which utilize the extra control that the Memory Manager object provides. These functions will be explained in the next sections.

## 4.1 Open

Open is a new function added to the Relation class in order to allow the Relation to choose when the table is mapped into memory. Previously the entire database was mapped into memory at all times on server initialization and kept in memory until the server was shut down. Now that a Memory Manager object is a member of the Relation class, the Relation has control when to map and unmap the table it has control over. The open function allows the Relation to load the table into memory before any accesses occur in order to save memory.

## 4.2 GetNext

In previous projects all accesses to a table were done to access a specific tuple. All reads either asked for a tuple at a specific index or a tuple with a specific indexed value. In order to perform a query and a join we needed a way to be able to get every tuple one at a time. The getNext function allows the Relation object to retrieve one tuple from its table at a time until it runs out of tuples. This functionality allowed us to perform a join by retrieving tuples one at a time and performing the necessary comparisons to see if they should be joined or not. Before getNext can be used the table needs to be loaded into memory using open. Once the join completes the query can use close to unload the table from memory.

## 4.3 Close

The close function performs the opposite action as the open function. Close allows the relation to unmap the table from memory whenever an access has completed. The addition of open and close give the database control of which tables exist in memory at a given time and ensure that tables that are currently not being used do not exist in memory. These functions decrease the average memory use of the server.

# 5. Logger Class

The logger class is made up of three functions that provide the interface to start, update, and end a transaction. Each relation class has their own logger member such that any update to a specific table is logged only with the changes for that table. Each relation logger writes to a ‘.log’ file located at the same level as the database itself.

## 5.1 Start Commit

When the server receives a request to update the population table, this begins a new transaction. The start commit function will write to the table’s log ‘START <TRANS. ID>. Every transaction needs to be book-ended by a START and an END.

## 5.2 Update Entry

Each item in the table that is updated will also be logged. The information stored in the log file is the transaction ID, the database entry ID, the old value, and the new value. It is important to store the old and new values in case the transaction is never committed, or an error occurs.

## 5.3 End Commit

Once all items have been updated, the relation informs the logger that the commit is complete, which then writes ‘END <TRANS. ID> to the log file. This ends the transaction and solidifies that the database change will be persistent. However, if for some reason the transaction fails, or the update is not committed, any changes will be reverted.

# 6. 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.

## 6.1 Population Changes

When making a percentage population increase or decrease to the database, that change needs to be applied across all tuples in both the city and country tables. The user would not want to increase the populations of only the cities or countries, they would need that change to apply across both tables. The user specifies which percentage increase or decrease they wish to occur in the population. The isolation manager opens both the city and country tables then retrieves each tuple one at a time. Next the isolation manager reads population from that tuple, performs the population increase or decrease calculation, then rewrites the new value into memory for that tuple using the memory manager class. Once all tuples are updated from both the city and country tables then the isolation manager closes those tables.

## 6.2 Restore Log

When a restore log command is issued to the server, all log files linked to each Relation will be parsed. While parsing, the isolation manager will update the corresponding database entry and also maintain a list of work completed. The entire log file will be parsed and all transactions will be applied to the tables to ensure proper migration.

If for any reason, a transaction was not completed correctly, the isolation manager will revert all changes related to that specific transaction, and then continue to the next transaction ID. This ensures that all updates are migrated from one database to another.

# 7. Memory Manager Class

The Memory Manager class 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. All of these functions are performed by the Memory Manager class. The database manager has one Memory Manager object per table. In previous projects there was only one table so a single global Memory Manager object existed. In project 4 the Memory Manager object remains in the Relation class for greater database control.

## 7.1 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.

## 7.2 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, a location string, and a country code. 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: 113
* Table Entry UID: 2. Storage Offset: 226

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.

## 7.3 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. In project 3 we added two tables so there will be two separate files, one for each table. We kept these changes for project 4. 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.

Instead of mapping the entire file into memory on server initialization, the Memory Manager gives the Relation class the functionality to map and unmap the table to and from memory whenever it wishes.

# 8. 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. In Project 4 the group used B+ Tree’s to index the country code attribute of each tuple. By indexing this attribute we were able to perform our join much more efficiently. The query involved checking each tuple in the City table and comparing the tuple to the matching tuple in the Country table. The country code is an attribute in both tables. This means for each tuple in the City table we were able to easily find the matching tuple in the Country table by searching for the particular country code using our B+ Tree instead of iterating the entire Country table. We kept the B+ Tree as a part of the database because we carried over the Query command from Project 3 into Project 4 to show how the queries change as the populations are increased or decreased. The B+ trees are not used in the “update-population” command or the “restore-log” command. The “update-population” command iterates all tuples by index and makes the necessary changes while the “restore-log” command retrieves the necessary tuple it needs to change by unique index then rewrites the change.

# 9. Test

There are four test cases associated with this project. The team assumes that the query command was already proven to work, however some manual regression testing was done to make sure the results are as expected because the query command is used to verify whether the other commands work. The first three test cases are automated and can be found in the test directory, and the fourth test case is a manual test case.

The first test case tests the update-population command with a two percent population increase. It assumes that no modification has been made to the dataset. It is an automated test that updates the population and makes sure the query at 40% displays the population values at 1.02 times their initial values.

The second test case tests the restore-log command. It is intended to be run after the first test case. Before the test case is run, it assumes that the server data gets deleted and needs to be reinitialized. It is important that the log files are not deleted at this time. After that, restart the server and run the test script. It expects the same result as the first test case, but it achieves that result by using the restore-log command instead of the update-population command.

The third test case tests the restore-log command when logs do not exist. It assumes that log files either have never been created or have been deleted. It is also an automated test. It expects that the server will notify the client that log files could not be found and then allows the user to continue operating the client.

The fourth test case is a manual test. The test steps are listed below.

1. First, from the project 4 directory do a “make clean” followed by a “make”.
2. After that, start a server instance from that directory with the command “./build/server”.
3. Allow the server to initialize its data, and then after the server is listening for connections, connect a client.
4. Use a “query” command at 40% and write down the 19 population values that result from the query.
5. Send an “update-population” command to increase the population by 10%.
6. Before this operation completes, select the server instance of putty and press Ctrl+C to kill the process.
7. Restart the server and connect the client to it.
8. Use the “restore-log” command.
9. Use another “query” command at 40% and verify that the population values are the same as the last query.

This test suite verifies that update-population works as expected and that restore-log will behave properly whether the log needs to undo an operation, redo an operation, or does not exist.

References

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

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