**CS 542 – DATABASE MANAGEMENT SYSTEMS**

**INTERNALS PROGRAMMING OPTION**

**Assignment 1 – Value Store**

By:

Neha Satish Mahajan

Arun Vadivel

**CONTENTS**

Contents

[**INTRODUCTION:** 3](#_Toc431425085)

[**PURPOSE OF THE PROJECT:** 3](#_Toc431425086)

[**TECHNOLOGIES INVOLVED:** 3](#_Toc431425087)

[**DATA STORAGE STRUCTURE DESIGN:** 4](#_Toc431425088)

[**PROGRAM FLOW:** 5](#_Toc431425089)

[**LIST OF CLASSES AND METHODS:** 6](#_Toc431425090)

[**INTERACTIVE GUI TO COMMUINCATE WITH THE DATABASE SYSTEM:** 8](#_Toc431425091)

[**SAMPLE OUTPUT:** 9](#_Toc431425092)

[**TEST CASES:** 9](#_Toc431425093)

[**ASSUMPTIONS:** 12](#_Toc431425094)

# **INTRODUCTION:**

A database management system (DBMS) is a collection of programs that enables storage, modification, and information extraction from a database. To understand how a database management system works, one needs to implement a database management system from the scratch. This project focusses on implementing a simple read, write and fetch operation in a database file using Java and it also includes an interactive GUI using Java Swing.

# **PURPOSE OF THE PROJECT:**

The project is focused on understanding the working of a database system during read, write and information extraction process. The database also needs to deal with fragmentations and locks during the read, write process. Implementing a database system with the mentioned challenges in mind will help the implementer to understand the internals of the working of a database management system.

# **TECHNOLOGIES INVOLVED:**

Language Implemented: Java (JDK 8)

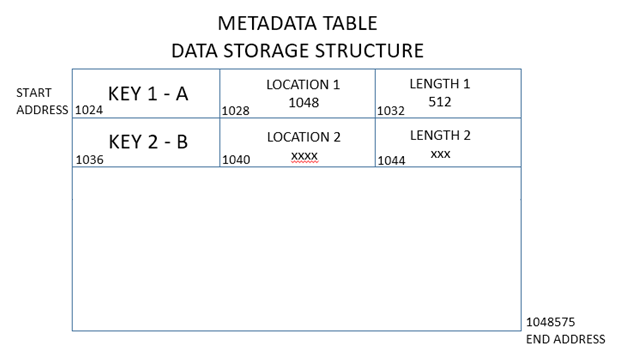
GUI toolkit: Java Swing

Platform used: Eclipse

Files:

* 1. A.txt – H.txt (9 Files)
  2. MainLaunch.class
  3. ValueStore.class
  4. Constants.java
  5. cs542.db

# **DATA STORAGE STRUCTURE DESIGN:**

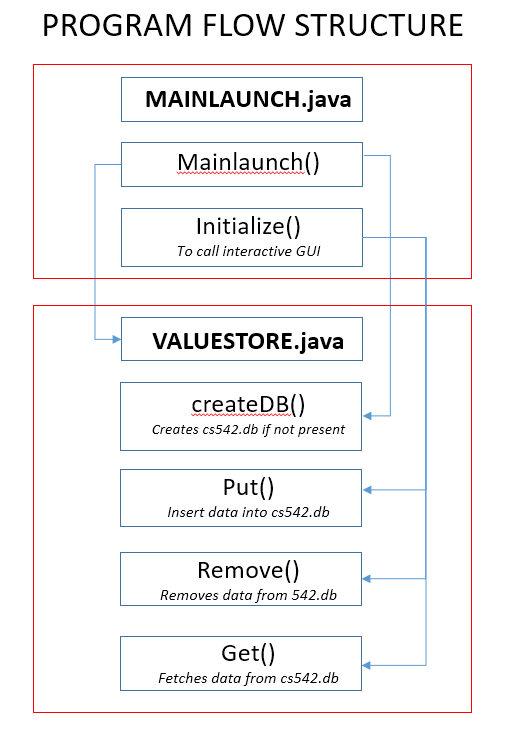


The data storage file CS542.DB has a maximum storage capacity of 5 MB. 1 MB of the total storage is allocated to the Metadata, so the file can store only 4 MB of actual data in the value filed.

The 1 MB of metadata contains the key, location and length of the data that is stored in the data section against the respective key.

Also the metadata section contains pointers which points to the free space in the metadata table and free space in the data section. There is another pointer to hold the value of the free space available in the data section.

# **PROGRAM FLOW:**



# **LIST OF CLASSES AND METHODS:**

1. **MainLaunch.java**

**public class MainLaunch**

1. public static void main(String[] args) - Launches the application
2. public MainLaunch() – calls ValueStore class and created the database file
3. private void initialize() - Initializes the contents of the frame. Displays the GUI to interact to choose the action to be performed.
4. private byte[] readFile(String filename) – Append the filename with the correct address and read the contents of the file.
5. **ValueStore.java**

**public class ValueStore**

1. public void createDB(String filepath) – Checks if the file “cs542.db” is available. If not, creates a new one.
2. public String openDB(String filepath) – Checks for the file and opens it.
3. public boolean checkEnoughSpaceInMD(int startAddress) - Checks if enough space to insert (key,position,length) in metadata table
4. public boolean checkForIfAnyDeletedRecords() - Returns true if there is atleast one deleted record in the metadata table
5. public int getMDCurrentLocation() - Returns the location for free space in the metadata table
6. public void setMDCurrentLocation(int startAddress) - Updates the location in the current metadata pointer
7. public int getTotalFreeSpace() - Returns the total free space available in the data section
8. public void setTotalFreeSpace(int freespace) - Updates the free space value in the file
9. public int getDataCurrentLocation() - Returns the location for free space in the data section
10. public void setDataCurrentLocation(int newAddress) - Updates the location in the current data pointer
11. public int getKey(int startAddress) - Returns the key value at the location
12. public void setKey(int startAddress, int key) - Writes the key in the metadata table
13. public int getLocation(int startaddress) - Returns the location for the data in the data section
14. public void setLocation(int startAddress, int dataAddress) - Writes data location in the metadata table
15. public int getLength(int startAddress) - Returns the length of the data in the data section
16. public void setLength(int startAddress, int length) - Writes data length in the metadata table
17. public byte[] getData(int startAddress, int length) - Returns the data
18. public void setData(int startAddress, byte[] data) - Writes the data in the data section
19. public void setFreeSpace(int startAddress, int freeSpaceAddress) - Set the free space location in the free space list
20. public void Put(int key, byte[] data) – Enters the data into cs542.db with respect to the key
21. public void checkForDeletedRecord(int key, byte[] data) – Checks if there is a free space to accommodate the new data; before fragmentation
22. public void handleFragmentation() – Does fragmentation on the existing data in cs542.db
23. public void checkSpaceAfterFragmentation(int key, byte[] data) - Checks if there is a free space to accommodate the new data; after fragmentation
24. public boolean openAndLockFile(String mode) – Locks the file to support concurrency validation
25. public void closeAndUnlockFile() - Unlocks the file to support concurrency validation
26. public void writeToDBFile(int position, byte[] data) – Writes data to cs542.db with respect to the key
27. public byte[] readFromDBFile(int position, int size) - Reads data from cs542.db with respect to the key
28. public byte[] Get(int key) – Fetches and displays data from cs542.db with respect to the key
29. public void Remove(int key) – Deletes the data from cs542.db with respect to the key
30. **Constants.java**

**public class Constants**

1. public static final int METADATA\_CURRENT\_POINTER = 0;

*Points to the first byte address where the current metadata pointer starts in the file. Size is of 4 bytes. The 4 bytes contains the address to the location where next free space is available in the metadata table.*

1. public static final int DATA\_CURRENT\_POINTER = 4;

*Points to the first byte address where the current data pointer starts in the file.Size is of 4 bytes. The 4 bytes contains the address to the location where next free space is available in the data section.*

1. public static final int FREESPACE\_POINTER = 8;

*Points to the first byte address where the free space pointer starts in the file.Size is of 4 bytes. The 4 bytes contains the value of total free space available in the data section.*

1. public static final int SIZE\_OF\_POINTER = 4;

*The number of bytes allocated for each of the pointers.*

1. public static final int METADATA\_TABLE\_POINTER = 1024;

*Points to the address where the metadata table starts in the file.*

1. public static final int METADATA\_TABLE\_END = 1048575;

*Points to the address where the metadata table ends in the file.*

1. public static final int DATA\_TABLE\_POINTER = 1048576;

*Points to the address where the data section starts in a file.*

1. public static final int SIZE\_OF\_KEY\_FIELD = 4;

*Byte length of key column of the metadata table. Stores the key value.*

1. public static final int SIZE\_OF\_POSITION\_FIELD = 4;

*Byte length of position column of the metadata table. Stores the location of the data in the data section corresponding to the key.*

1. public static final int SIZE\_OF\_LENGTH\_FIELD = 4;

*Byte length of length column of the metadata table. Stores the total length of data in the data section corresponding to the key.*

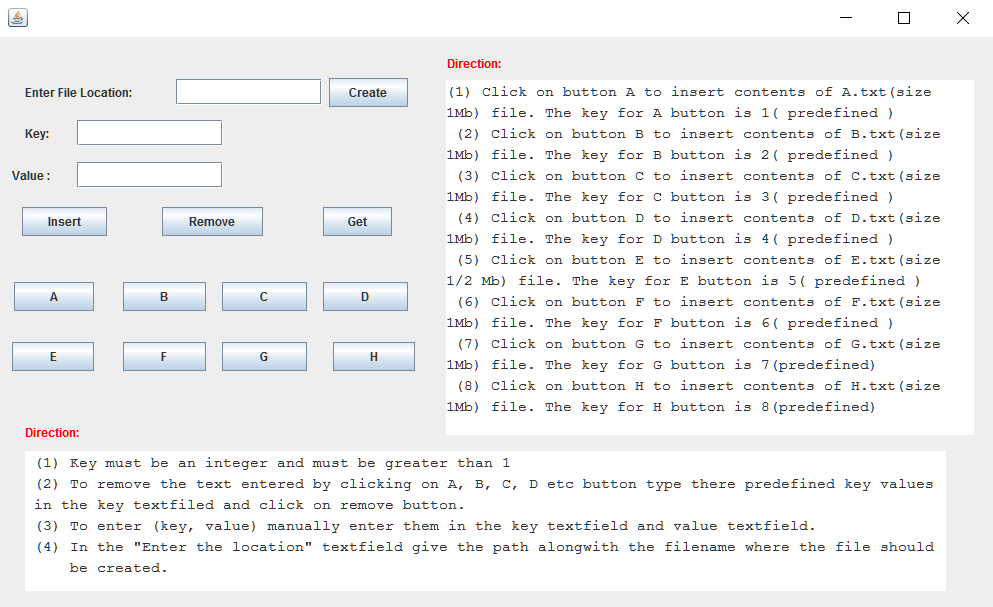
1. public static final int FILE\_LENGTH = 5242880;

*Total number of bytes allocated to the entire file (pointers + metadata + data)*

1. public static final int DELETED\_KEY\_VALUE = 0;

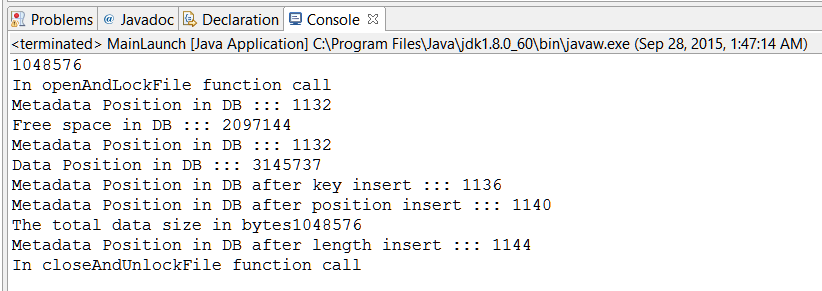
*Default value for key when it is deleted*

# **INTERACTIVE GUI TO COMMUINCATE WITH THE DATABASE SYSTEM:**



# **SAMPLE OUTPUT:**

In Eclipse:

****

# **TEST CASES:**

**Inserting into the database:**

**Concurrency Validation:**

**Test Case 1:**

One thread is doing a Get() while another caller does a Put() and replaces the data.

**Implementation:**

In the current implementation with the value store, when a caller is accessing the file to get some data, the process has locked the file and since it is the implementation of the underlying operating system, another caller who tries to Put() new data doesn’t get a lock on the file and cant proceed with the Put() operation. However once the Get() operation is finished the other caller can again try to get the lock on the file and go ahead with the Put() operation.

**Test Case 2:**

One thread is doing Remove() while another caller does a Get().

**Implementation:**

In the current implementation with the value store, when a caller is accessing the file to remove some data, the process has locked the file and since it the implementation of the underlying operating system, another caller who tries to Get() the data doesn’t get a lock on the file and cant proceed with the Get() operation.

**Durability Validation:**

**Test Case:**

What happens if, after a reboot of the machine after the data has been Put(), a caller does a Get()?

**Implementation:**

If the data and its related metadata is successfully written to the file before the reboot of the machine and another caller does a Get() then the data is accessible. However if the machine reboots while the Put() operation is still in progress then let’s see how it affects the Get() method:

Before we proceed with the test cases let us see how the data is managed into the file.

The file is divided into 2 sections: Metadata and Data.

1) When a caller does Put(key, value) the value is the first thing written to the file. To write the value in data section of the file, there is a pointer which tells the next empty location in the data section.

2) Next the key for the value is written in the metadata table.

3) The Location of the data is stored in the metadata table.

4) The length of the data is also stored in the metadata table.

5) There are two pointers: one which points to the empty location in the metadata table and another which points to the empty location in the data section.

6) The pointer for the empty location in the data part is updated when the data and metadata are written to the file.

7) The pointer to the metadata table is updated after the pointer to the data part is updated.

**Test Case 1:**

The data is written in the data section and the machine reboots.

**Implementation:**

The pointer to the metadata and data part are not updated, neither the entry for the key is made in the metadata table. So when a caller does Get(), there is no matching key in the metadata table and thus even though data is written it is not a valid data and cannot be accessed.

**Test Case 2:**

The data, key, location, length is written to the file and the machine reboots.

Implementation:

It is similar to the Test Case 1. Since the pointers are not updated the data and key both are not accessible.

**Test Case 3:**

The data, key, location, length, pointer for data part is updated and the machine reboots

Implementation:

Since the metadata table pointer is not updated the key in the metadata table is still inaccessible and thus the data for the key cannot be retrieved.

**Test Case 4:**

The data, key, location, length, pointer for data and metadata part is updated and the machine reboots

Implementation:

This is the complete implementation of the Put() operation and Get() will return the data successfully.

**Fragmentation:**

Put() 4 values, byte arrays of 1 MB each, with keys A, B, C and D. Remove key B. Put() ½ MB in size for key E. Validate that a Put() 1 MB in size for key F fails. Remove C and now validate that a Put() 1 MB in size for key G succeeds. Remove E and try Put() 1 MB in size for key H. With a naive implementation, it will fail even though there is room in cs542.db. An extra bonus point if you can modify your code such that Put("H", …) succeeds.

**Implementation:**

With the current implementation the fragmentation is handled and Put() 1 Mb for key H succeeds.

**Let us see how fragmentation is handled:**

When a caller does a Put(key, value) the key and the data are written to the file and the pointers are accordingly updated. When a caller calls Remove(some key), the tuple for that key is not deleted but it is changed to value 0 in the metadata table, which tells us that the key and the data for the key is no more valid. In case when there is not enough contiguous space, there are chunks of empty spaces in between valid data the handleFragmentation() method is called. It goes through each tuple of the metadata table and collecting only valid records in the table. It scans sequentially and updates the address space for the key as well as the data making contiguous space available at the end of the data section. So when any new Put() is called the data is appended at the end of the data section.

# **ASSUMPTIONS:**

1. A tuple size can have a maximum size of 1 MB.
2. User should enter values in the respective before inserting, deleting or fetching data.
3. Total database storage size cannot exceed 5 MB, including 1 MB of metadata storage.
4. Only one tuple can be inserted, fetch or deleted at a time.