CS213 · Principles of Database(H)

Project 1

Bai Junyan / 12212108

October 15, 2023

Contents

1 Introduction

This project aims to investigate the differences between database systems (DBMS) and the Java file system, as well as how to simulate DBMS functionality. A DBMS is a system specifically designed for storing, retrieving, and managing large volumes of data. Data is stored in tabular form within a database and is typically accessed and managed using SQL (Structured Query Language). In this project, we primarily focus on relational databases.

In contrast, the Java file system is implemented through programming languages such as C++ and Java, for reading and writing data in text files or binary files. Database systems and file systems exhibit significant differences in various aspects, including data storage, query performance, security, and scalability. The main objective of this project is to compare the query performance of both systems to gain a better understanding of why databases are preferred when dealing with substantial data volumes.

2 Project Requirements and Design

In this project, the relevant operations for both the database system and the file system are implemented separately in Datagrip and IDEA using SQL and the Java programming language.

2.1 Basic Requirements and Their Design

- Data Import: Import a large volume of data files into both the database and the Java file system.
- Basic Retrieval Comparison: Compare basic retrieval methods using keywords in the database and the Java file system. Subsequently, I will draw inspiration from the underlying principles of databases to optimize retrieval methods in the Java file system.
- Data Update Operation: Replace parts of names containing ”TO” with ”TTOO” in both the database and the Java file system. Initially, use the most basic methods in Java, and later, optimize the process.

2.2 Other Design Considerations

In addition to meeting the basic requirements, I delved into the underlying principles of databases, gaining insights into the fundamental reasons behind their exceptional performance. Subsequently, I dedicated efforts to simulate the core principles of databases in Java, aiming to enhance performance in aspects like file I/O and retrieval. I conducted an extensive comparison between these optimized methods and the initial basic approaches.

- Creating a HashMap Index Table for Retrieval Optimization: To optimize the retrieval process, we create a HashMap index table for the file. This index table maps IDs to HashMap keys, and each line’s specific information is concatenated into a single string, which serves as the corresponding value for the key in the HashMap.
- Simulating a Database’s Primary Key: Utilizing the created HashMap, we establish a virtual primary key for the file and compare query speeds with and without a primary key.
- Data Deletion Operations: A comparison of basic methods for deleting specific rows in both the database and the Java file system.
- Simulating a Database’s Foreign Key: Utilizing the created HashMap, we perform checks before each deletion operation on the file, enhancing the security of data in the Java file system.
- Join Operation Comparison: A comparison of basic methods for joining two tables in both the database and the Java file system.
- Multithreading and Parallel Processing: Creating a thread pool in Java to simulate the database’s multithreaded parallel processing. When reading files, divide large files into multiple regions, with each region handled by a separate thread to expedite the process. For concurrent queries, employ asynchronous processing, distributing query tasks to multiple threads.
- Optimizing Update, Delete, and Query Operations with HashMap Index: By leveraging the created index table for fast querying, we optimize update, delete, query, and join operations. We compare the performance against the basic methods.
- Real-Time Updating of HashMap Index Table Using File Watcher: By creating a file watcher, we capture updates and deletions of files, automatically updating the corresponding index table accordingly.
- Simulating Database Index Principles Using B+ Trees and Brin Index: We compare the query speed of B+ tree and Brin indexes to that of the HashMap index, exploring how these index structures mimic database indexing principles.
- Optimizing Java File Join Operations with Sorting-Merge Algorithms: First, perform internal sorting of the file using Heapsort. Then, optimize join operations in Java using the sorting-merge algorithm

commonly used in database engines. Compare this approach to using HashMap indexing and basic methods to assess its performance.

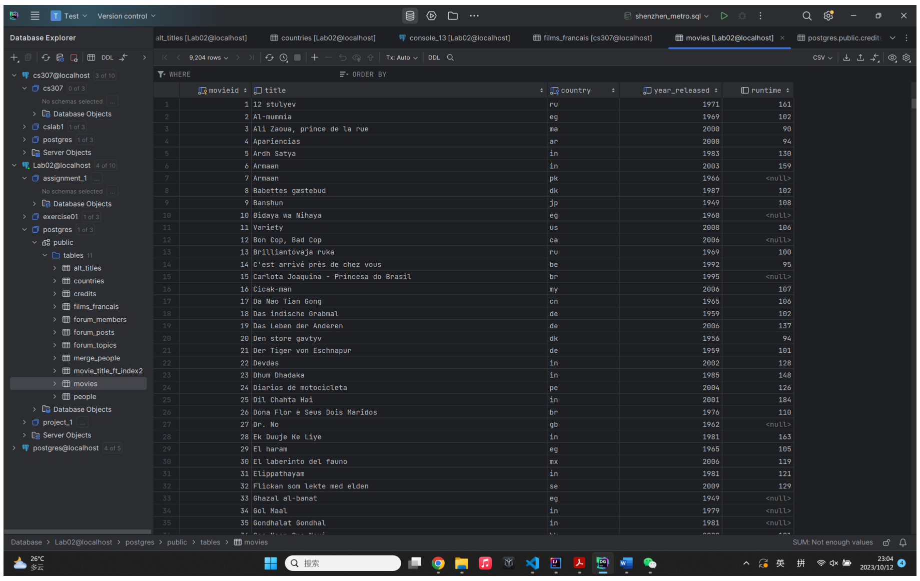
3 Implementation of the Project

For comparing the performance of database and file system operations, after each execution of the same method, obtain the code's execution time through Datagrip's output information panel. In the Java client, wrap the code execution portion using **System.nanoTime()** and print the runtime.

3.1 Data Import

The data used for this project comes from the **filmdb.sql** file obtained from Blackboard, which creates multiple tables related to movies, and the data in it exceeds **10,000** records, meeting the project requirements.

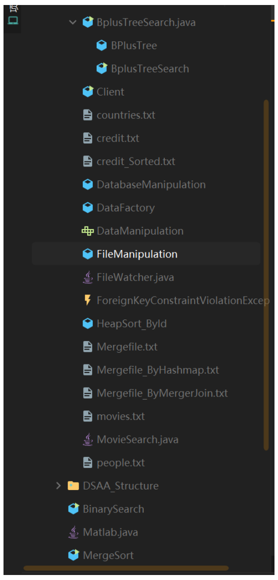
In this project, I used the PostgreSQL database engine, and all database operations were implemented in **DataGrip**. To import data into the database, I drag the data file into **DataGrip** and configure it to run in the desired database.



	movieid	title	country	year_released	runtime
1	1	12 stulyev	ru	1971	161
2	2	Al-mummia	eg	1969	102
3	3	Ali Zaoua, prince de la rue	ma	2000	90
4	4	Apariencias	ar	2000	94
5	5	Ardh Satya	in	1983	130
6	6	Armaan	in	2003	159
7	7	Armaan	pk	1966	<null>
8	8	Babettes gæstebud	dk	1987	102
9	9	Banshun	jp	1949	108
10	10	Bidaya wa Nihaya	eg	1960	<null>
11	11	Variety	us	2008	106
12	12	Bon Cop, Bad Cop	ca	2006	<null>
13	13	Brilliantovaja ruka	ru	1969	100
14	14	C'est arrivé près de chez vous	be	1992	95
15	15	Carlota Joaquina - Princesa do Brasil	br	1995	<null>
16	16	Cicak-man	my	2006	107
17	17	Da Nao Tian Gong	cn	1965	106
18	18	Das indische Grabmal	de	1959	102
19	19	Das Leben der Anderen	de	2006	137
20	20	Den store gavtyv	dk	1956	94
21	21	Der Tiger von Eschnapur	de	1959	101
22	22	Devdas	in	2002	128
23	23	Dhum Dhadaka	in	1985	148
24	24	Diarios de motocicleta	pe	2004	126
25	25	Dil Chahta Hai	in	2001	184
26	26	Dona Flor e Seus Dois Maridos	br	1976	110
27	27	Dr. No	gb	1962	<null>
28	28	Ek Duuje Ke Liye	in	1981	163
29	29	El haram	eg	1965	105
30	30	El laberinto del fauno	mx	2006	119
31	31	Elippathayam	in	1981	121
32	32	Flickan som lekte med elden	se	2009	129
33	33	Ghazal al-banat	eg	1949	<null>
34	34	Gol Maal	in	1979	<null>
35	35	Gondhalat Gondhal	in	1981	<null>

Figure 1: Importing Data into the Database

For the Java project management, I used **IDEA**. To import movie data, I just need to drag the pre-created text files to the project's root directory.



```
BplusTreeSearch.java
    BPlusTree
    BplusTreeSearch
Client
countries.txt
credit.txt
credit_Sorted.txt
DatabaseManipulation
DataFactory
DataManipulation
FileManipulation
FileWatcher.java
ForeignKeyConstraintViolationExcep
HeapSort_ById
Mergefile.txt
Mergefile_ByHashmap.txt
Mergefile_ByMergerJoin.txt
movies.txt
MovieSearch.java
people.txt
DSAA_Structure
BinarySearch
Matlab.java
MergeSort
```

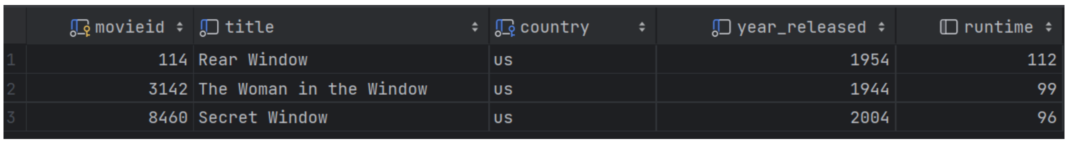
Figure 2: Importing Data into IDEA

3.2 Basic Retrieval Comparison

To query by the key field of the movie title in the database, the SQL command is as follows:

```sql
Select * from movies where title Like '%Window%';
```

This piece of code uses the **Select...Where** statement to retrieve all movie information from the "movies" table where the title contains the word "Window." The result is as follows, **the execution time is 27 milliseconds**:



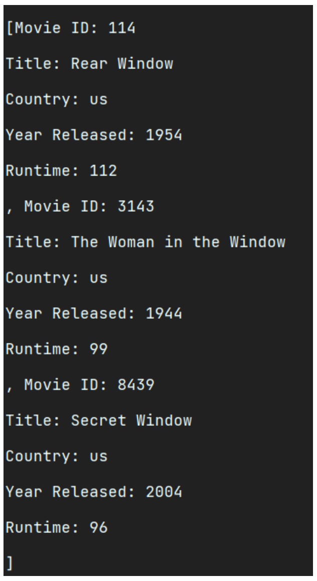
	movieid	title	country	year_released	runtime
1	114	Rear Window	us	1954	112
2	3142	The Woman in the Window	us	1944	99
3	8460	Secret Window	us	2004	96

Figure 3: Results of the basic Database Retrieval.

When implementing this retrieval function using the Java file system, the basic approach is to read the file into memory, iterate through the file line by line, perform conditional checks for each line, and finally print all lines that meet the criteria. The **pseudocode** is as follows:

```
    function findMoviesByTitle(title):
matchingMovies = empty list

for each line in movies.txt file:
    split the line into an array using semicolon as the delimiter
    titleIndex = 1
    if the title is contained in splitArray[titleIndex]:
        id = parse integer from splitArray[0]
        movieTitle = get non-null value from splitArray[1]
        country = get non-null value from splitArray[2]
        yearReleased = parse integer from splitArray[3]
        runtime = parse integer from splitArray[4]

        movieInfo = create movie info string
        add movieInfo to matchingMovies list

    return matchingMovies as a string
```

This section primarily utilizes the **split** method of strings to split a string using ";" and store each segment of information in the **splitArray** array. When filtering the title field, this array is used. The printed results match those obtained from a database query, **and the execution time is 63ms**:



```
[Movie ID: 114
Title: Rear Window
Country: us
Year Released: 1954
Runtime: 112
, Movie ID: 3143
Title: The Woman in the Window
Country: us
Year Released: 1944
Runtime: 99
, Movie ID: 8439
Title: Secret Window
Country: us
Year Released: 2004
Runtime: 96
]
```

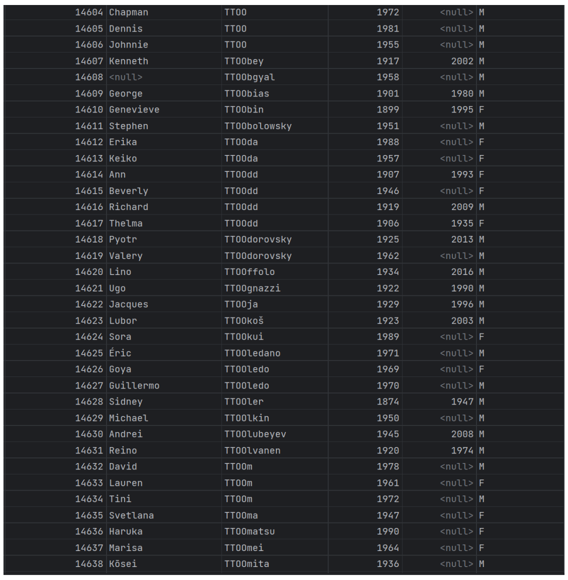
Figure 4: Data Retrieval By Java

3.3 Data Update Operations

I directly use the **"Update...SET...Where"** statement in the database to perform update operations. The code is as follows:

```
        UPDATE people
        SET first_name = replace(first_name,'To','TTOO')
        WHERE first_name ILIKE '%To'
```

This code segment utilizes the **"replace"** method to replace the specified 'To' field, **with an execution time of 11ms**.



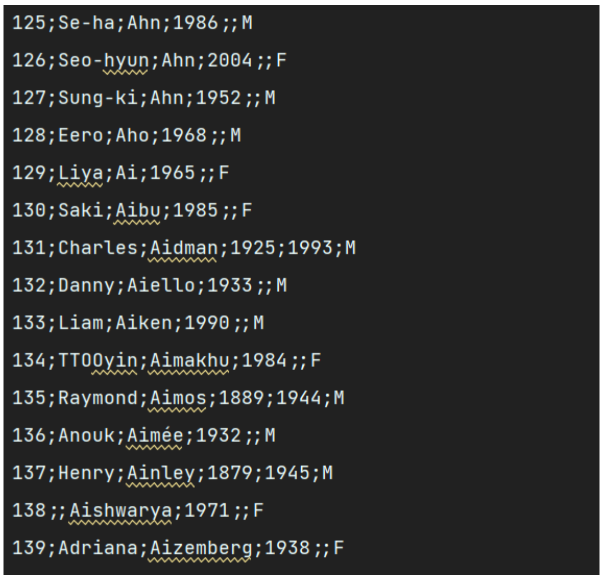
14604	Chapman	TTOO	1972	<null>	M
14605	Dennis	TTOO	1981	<null>	M
14606	Johnnie	TTOO	1955	<null>	M
14607	Kenneth	TTOObey	1917	2002	M
14608	<null>	TTOObgyal	1958	<null>	M
14609	George	TTOObias	1901	1980	M
14610	Genevieve	TTOObin	1899	1995	F
14611	Stephen	TTOObolowsky	1951	<null>	M
14612	Erika	TTOOda	1988	<null>	F
14613	Keiko	TTOOda	1957	<null>	F
14614	Ann	TTOOdd	1907	1993	F
14615	Beverly	TTOOdd	1946	<null>	F
14616	Richard	TTOOdd	1919	2009	M
14617	Thelma	TTOOdd	1906	1935	F
14618	Pyotr	TTOOdorovsky	1925	2013	M
14619	Valery	TTOOdorovsky	1962	<null>	M
14620	Lino	TTOOffolo	1934	2016	M
14621	Ugo	TTOOgnazzi	1922	1990	M
14622	Jacques	TTOOja	1929	1996	M
14623	Lubor	TTOOkoš	1923	2003	M
14624	Sora	TTOOkui	1989	<null>	F
14625	Éric	TTOOledano	1971	<null>	M
14626	Goya	TTOOledo	1969	<null>	F
14627	Guillermo	TTOOledo	1970	<null>	M
14628	Sidney	TTOOler	1874	1947	M
14629	Michael	TTOOlkin	1950	<null>	M
14630	Andrei	TTOOlubeyev	1945	2008	M
14631	Reino	TTOOlvanen	1920	1974	M
14632	David	TTOOm	1978	<null>	M
14633	Lauren	TTOOm	1961	<null>	F
14634	Tini	TTOOm	1972	<null>	M
14635	Svetlana	TTOOma	1947	<null>	F
14636	Haruka	TTOOmatsu	1990	<null>	F
14637	Marisa	TTOOmei	1964	<null>	F
14638	Kōsei	TTOOmita	1936	<null>	M

Figure 5: The Updated ”people” Table

In the file system, the most basic approach is similar, where read the file into memory, traverse through it line by line, and perform a line-by-line comparison to filter data with names containing ”TO,” replacing it with ”TTOO.” After replacing a line, it's written back into the original file to overwrite it. It's worth noting that nested comparisons can lead to **poor performance**, and further optimization using indexing will be explored to enhance the update operation's efficiency. Below is the **pseudocode** of the basic method:

```
    Function updatePeopleName(BeforeReplacement, AfterReplacement):
filePath = ”src\\Database\\people.txt”
lines = ReadFromFile(filePath)
modifiedLines = []

for line in lines :
    if line contains BeforeReplacement:
        modifiedLine = Replace ”To” with ”TTOO” in line
        Append modifiedLine to modifiedLines
    else Append modifiedLine to modifiedLines

if modifiedLines is not empty:
    Write modifiedLines to the file at filePath
    Load index from ”src\\Database\\people.txt” # Automatically update the index
    Print ”File␣updated␣successfully.”
else :
    Print ”No␣such␣match.”
```

In this code segment, the built-in string **replace** method is also utilized for easy substring replacement. The code execution time is 43ms.



```
125;Se-ha;Ahn;1986;;M
126;Seo-hyun;Ahn;2004;;F
127;Sung-ki;Ahn;1952;;M
128;Eero;Aho;1968;;M
129;Liya;Ai;1965;;F
130;Saki;Aibu;1985;;F
131;Charles;Aidman;1925;1993;M
132;Danny;Aiello;1933;;M
133;Liam;Aiken;1990;;M
134;TTOOyin;Aimakhu;1984;;F
135;Raymond;Aimos;1889;1944;M
136;Anouk;Aimée;1932;;M
137;Henry;Ainley;1879;1945;M
138;;Aishwarya;1971;;F
139;Adriana;Aizemberg;1938;;F
```

Figure 6: The Updated "people" File

Because there will be several subsequent operations that will make use of file reading and writing methods, I have encapsulated two methods, **readFromFile** and **writeToFile**, for convenient future use.

3.4 Creating Hashmap Indexes to Optimize the Retrieval Process

The **retrieval operation** is one of the fundamental data operations upon which most other operations are built. After studying the principles of database retrieval, I gained an understanding of the concept of index tables. I learned that creating index tables can significantly enhance retrieval efficiency. Similarly, optimizing this step is crucial in the context of Java. For each of the three files imported into Java, namely **movies.txt, credit.txt, and person.txt**, I created a **hashmap index table** for each file. In this process, I mapped the IDs to keys in the hashmap and concatenated the specific information for each line into a single string, which was then mapped to a value. The brief code implementation is as follows(Using the creation of a hash index for **movie.txt** as an example):

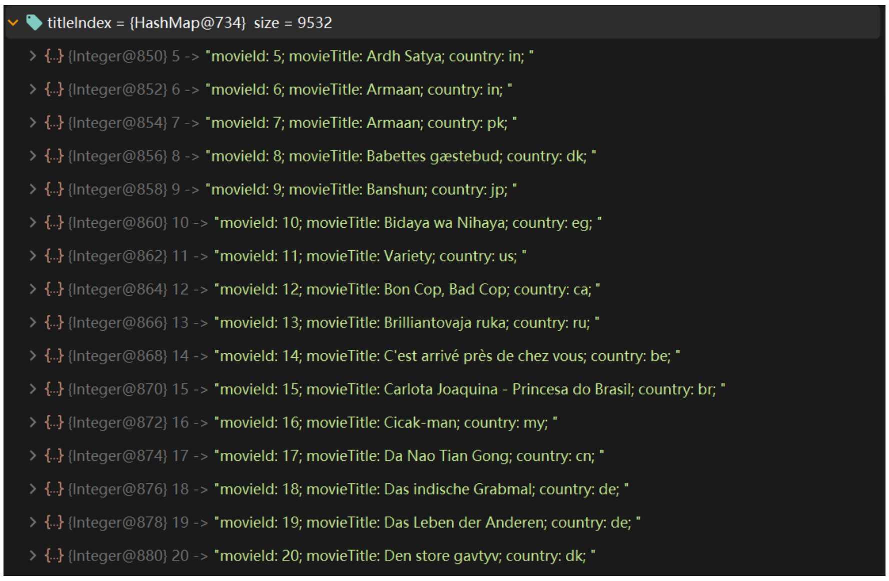
```
function loadIndexFromFile(indexpath):
 open indexpath as a file
 for each line in the file :
     split the line by ';'
     extract the id from the first part
     extract the movieTitle and country from the second and third parts
     concatenate id, movieTitle, and country into a value string
     add the id as the key and value as the associated data to the titleIndex
 close the file
end function
```

In this code, each line's **movieId** and the concatenated information **value** are stored in the HashMap **titleIndex**. To make it convenient for other methods to directly access the created index table, I've treated the created index table as an attribute of the **FileManipulation** class. In the **constructor method**, I directly call the method to create a hash index to ensure that the index table is automatically created **before** any operations are performed.

```
   public class FileManipulation implements DataManipulation {
 private ExecutorService threadPool;
 Map<Integer, String> titleIndex = new HashMap<>();
 Map<Integer, String> creditIndex = new HashMap<>();
 Map<String, String> countryIndex = new HashMap<>();
```

```
    public FileManipulation() {
        threadPool = Executors.newFixedThreadPool(10);
        loadIndexFromFile("src\\Database\\movies.txt");
        loadCreditFromfile("src\\Database\\credit.txt");
        loadCountryFromFile("src\\Database\\countries.txt");
    }
```

The created Hashmap **titleIndex** is as follow:



```
⌄ titleIndex = {HashMap@734}  size = 9532
  > {...} {Integer@850} 5 -> "movieId: 5; movieTitle: Ardh Satya; country: in; "
  > {...} {Integer@852} 6 -> "movieId: 6; movieTitle: Armaan; country: in; "
  > {...} {Integer@854} 7 -> "movieId: 7; movieTitle: Armaan; country: pk; "
  > {...} {Integer@856} 8 -> "movieId: 8; movieTitle: Babettes gæstebud; country: dk; "
  > {...} {Integer@858} 9 -> "movieId: 9; movieTitle: Banshun; country: jp; "
  > {...} {Integer@860} 10 -> "movieId: 10; movieTitle: Bidaya wa Nihaya; country: eg; "
  > {...} {Integer@862} 11 -> "movieId: 11; movieTitle: Variety; country: us; "
  > {...} {Integer@864} 12 -> "movieId: 12; movieTitle: Bon Cop, Bad Cop; country: ca; "
  > {...} {Integer@866} 13 -> "movieId: 13; movieTitle: Brilliantovaja ruka; country: ru; "
  > {...} {Integer@868} 14 -> "movieId: 14; movieTitle: C'est arrivé près de chez vous; country: be; "
  > {...} {Integer@870} 15 -> "movieId: 15; movieTitle: Carlota Joaquina - Princesa do Brasil; country: br; "
  > {...} {Integer@872} 16 -> "movieId: 16; movieTitle: Cicak-man; country: my; "
  > {...} {Integer@874} 17 -> "movieId: 17; movieTitle: Da Nao Tian Gong; country: cn; "
  > {...} {Integer@876} 18 -> "movieId: 18; movieTitle: Das indische Grabmal; country: de; "
  > {...} {Integer@878} 19 -> "movieId: 19; movieTitle: Das Leben der Anderen; country: de; "
  > {...} {Integer@880} 20 -> "movieId: 20; movieTitle: Den store gavtyv; country: dk; "
```

Figure 7: Hashmap **titleIndex**

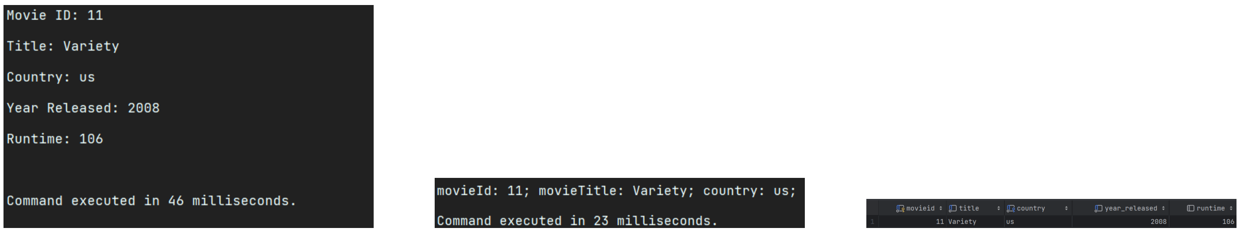
3.5 Simulating Primary Key In Database

The created HashMap inherently provides the functionality of a **primary key**, where each row's key serves as the primary key for database-like queries. It adheres to the basic properties of primary keys, such as **"Not null" and "Unique."** To compare the retrieval speed between having a primary key and not having one, I've created two new methods: **findMovieById** and **findMovieByIdHashmap**.

The code is as follows (the **findMovieById** method is similar to **findMovieByTitle** and is not displayed here):

```
        public void findMovieById_Hashmap(int id) {
        threadPool.submit(() -> {
            long start = System.nanoTime();
            String information = titleIndex.get(id);
            long end = System.nanoTime();
            long executionTimeInMilliseconds = (end - start);
            System.out.println("Command␣executed␣in␣"
            + executionTimeInMilliseconds + "␣nanoseconds.");
            System.out.println(information);
        });
```

In this code section, direct calls to the Hashmap's get method are used to retrieve information corresponding to the **primary key**. The code execution time is **23ms**, significantly shorter compared to the **43ms** execution time of **findMovieById**. However, there is still a notable difference when compared to the **3ms** execution time in the database.



```
Movie ID: 11
Title: Variety
Country: us
Year Released: 2008
Runtime: 106

Command executed in 46 milliseconds.
```

```
movieId: 11; movieTitle: Variety; country: us;
Command executed in 23 milliseconds.
```

```
  movieid  title    country  year_released  runtime
1      11  Variety  us                2008      106
```

(a) Without Primary Key (b) With Primary Key (c) Database output

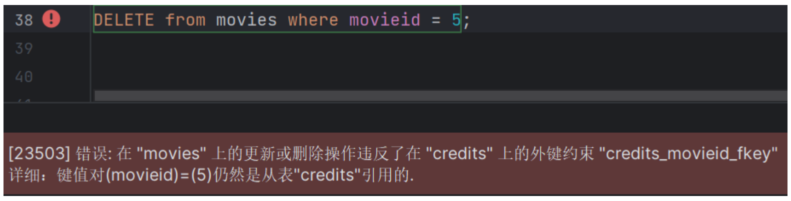
Figure 8: Simulating Database Primary Keys

3.6 Data Deletion Operations

In a database, deletion operations can be accomplished using the SQL statement **Delete...From...Where**:

```sql
DELETE from movies where movieid = 5;
```

However, upon running the code, it was discovered that deletion couldn't be performed due to **foreign key constraints**:



```
DELETE from movies where movieid = 5;
```

```
[23503] 错误: 在 "movies" 上的更新或删除操作违反了在 "credits" 上的外键约束 "credits_movieid_fkey"
详细：键值对(movieid)=(5)仍然是从表"credits"引用的.
```

Figure 9: Failed Attempts to Delete In Database

Using Java for data deletion through file read and write operations, the basic idea is as follows: first, create an empty table to store the data, then read and traverse the file. By using conditional statements, if the condition is met, the data is not added to the table; if the condition is not met, it is stored in the table. Finally, convert this table into the file format and write it back to the original file for replacement. The **pseudocode** is as follows:

```
Pseudocode deleteMovie(id):
Start a new thread to execute:
    filePath = "src\\Database\\movies.txt"
    If it's possible to delete the movie with id:
        Read the content of the file at filePath into lines
        For each line in lines:
            line = current line in lines
            splitArray = split line using a semicolon as the delimiter
            If splitArray[0] is equal to id:
                Remove the current line from lines
                Exit the loop
        Write lines back to the file at filePath
        Load the index table from the file at "src\\Database\\movies.txt"
        Print "Index table updated"
    Else:
        Throw a ForeignKeyConstraintViolationException
```

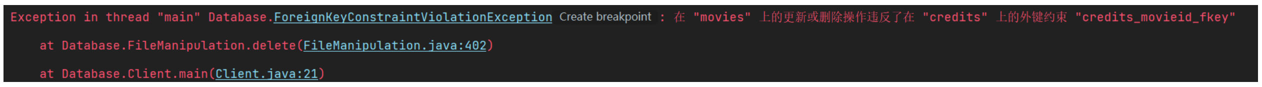
After executing the code, it was observed that Java allowed data to be deleted **without any restrictions**, highlighting a significant disparity in **data security** between the two systems. In the next section, I will use Java to simulate **foreign key constraints** in the database to enhance data security.

3.7 Simulate Database Foreign Key Constraints

During the simulation of data deletion, the Java file system does not adhere to the **foreign key constraints** present in the database. This allows for easy manipulation of file content, resulting in very low security. To simulate this functionality, I utilized the index tables of **'movie.txt'** and **'credit.txt'** to establish a **'constraint effect'** between these two files. The specific code is as follows:

```
        public boolean canDeleteMovie(int movieId) {
            return !creditIndex.containsKey(movieId);
        }
    public class ForeignKeyConstraintViolationException extends RuntimeException {
        public ForeignKeyConstraintViolationException(String message) {
        super(message);
        }
    }
        If it's possible to delete the movie with id:
            ..........
        Else:
            Throw a ForeignKeyConstraintViolationException
```

I created the **'canDeleteMovie'** method, which internally uses the **'contains'** method of a HashMap to determine if there is a constraint relationship between IDs in two tables. I also customized an exception class, **'ForeignKeyConstraintViolationException'**. During the deletion operation, when a constraint violation occurs, it will return the error message **'Violation of foreign key constraint!'.**



```
Exception in thread "main" Database.ForeignKeyConstraintViolationException Create breakpoint : 在 "movies" 上的更新或删除操作违反了在 "credits" 上的外键约束 "credits_movieid_fkey"
    at Database.FileManipulation.delete(FileManipulation.java:402)
    at Database.Client.main(Client.java:21)
```

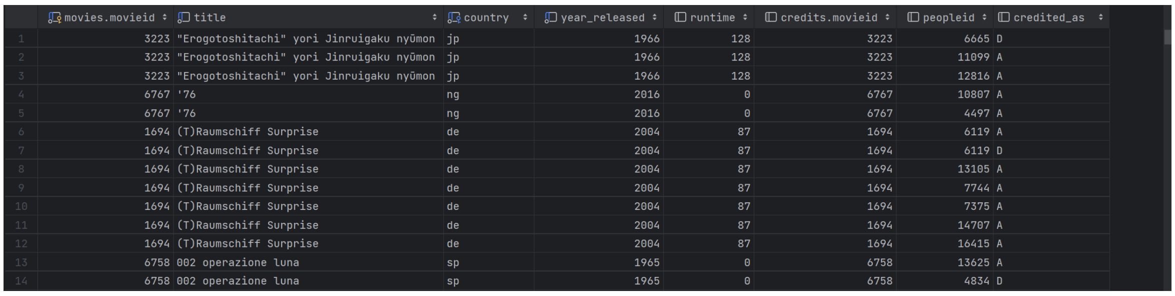
Figure 10: Failed Attempts to Delete In Java

3.8 Comparison of Join Operations

Join operations enable the merging of multiple tables into a single table using a common key. In a database, this operation can be performed using SQL statements like **'Join...On'**:

```
    Select * from movies Join credits on movies.movieid = credits.movieid
```

This command joins the **'movies'** and **'credits'** tables using the common 'id' and produces the following result, **the execution time is 11 milliseconds**:



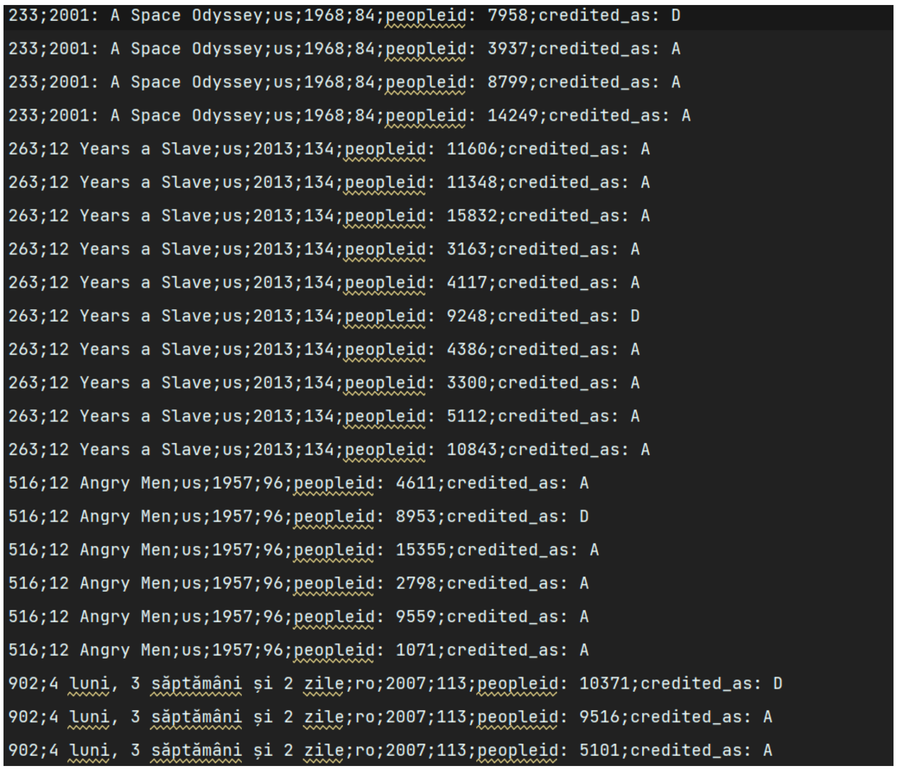
	movies.movieid	title	country	year_released	runtime	credits.movieid	peopleid	credited_as
1	3223	"Erogotoshitachi" yori Jinruigaku nyūmon	jp	1966	128	3223	6665	D
2	3223	"Erogotoshitachi" yori Jinruigaku nyūmon	jp	1966	128	3223	11099	A
3	3223	"Erogotoshitachi" yori Jinruigaku nyūmon	jp	1966	128	3223	12816	A
4	6767	'76	ng	2016	0	6767	10807	A
5	6767	'76	ng	2016	0	6767	4497	A
6	1694	(T)Raumschiff Surprise	de	2004	87	1694	6119	A
7	1694	(T)Raumschiff Surprise	de	2004	87	1694	6119	D
8	1694	(T)Raumschiff Surprise	de	2004	87	1694	13105	A
9	1694	(T)Raumschiff Surprise	de	2004	87	1694	7744	A
10	1694	(T)Raumschiff Surprise	de	2004	87	1694	7375	A
11	1694	(T)Raumschiff Surprise	de	2004	87	1694	14707	A
12	1694	(T)Raumschiff Surprise	de	2004	87	1694	16415	A
13	6758	002 operazione luna	sp	1965	0	6758	13625	A
14	6758	002 operazione luna	sp	1965	0	6758	4834	D

Figure 11: Join Operation In Database

In a file system, to achieve this operation using Java for file read and write, the basic idea is to use two nested loops to iterate over the two files that need to be merged. Within the inner loop, perform key-value condition checks and merge the rows that meet the criteria, then write them to a new file. Here's the **pseudocode** for this process:

```
Create a file 'Mergefile.txt'
Open 'movies.txt' for reading as 'moviesReader'
Open 'credit.txt' for reading as 'creditsReader'
Open 'Mergefile.txt' for writing as 'mergedWriter'
Skip the first line in 'moviesReader'
While there are lines in 'moviesReader':
 Read a line from 'moviesReader' into 'moviesLine'
    Split 'moviesLine' by ';'
    Extract 'movieId' from the first element of the split result
    Skip the first line in 'creditsReader'
        While there are lines in 'creditsReader':
        Read a line from 'creditsReader' into 'creditsLine'
        Split 'creditsLine' by ';'
        Extract 'creditMovieId' from the first element of the split result
        If 'movieId' matches 'creditMovieId':
        Merge the data from 'moviesLine' and 'creditsData'
        Write the merged data to 'mergedWriter'
Close 'creditsReader'
Reopen 'credit.txt' and assign to 'creditsReader'
```

Nested loops for two files have relatively low performance, **with an execution time of 48ms**. The merged result is as follows:



```
233;2001: A Space Odyssey;us;1968;84;peopleid: 7958;credited_as: D
233;2001: A Space Odyssey;us;1968;84;peopleid: 3937;credited_as: A
233;2001: A Space Odyssey;us;1968;84;peopleid: 8799;credited_as: A
233;2001: A Space Odyssey;us;1968;84;peopleid: 14249;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 11606;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 11348;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 15832;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 3163;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 4117;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 9248;credited_as: D
263;12 Years a Slave;us;2013;134;peopleid: 4386;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 3300;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 5112;credited_as: A
263;12 Years a Slave;us;2013;134;peopleid: 10843;credited_as: A
516;12 Angry Men;us;1957;96;peopleid: 4611;credited_as: A
516;12 Angry Men;us;1957;96;peopleid: 8953;credited_as: D
516;12 Angry Men;us;1957;96;peopleid: 15355;credited_as: A
516;12 Angry Men;us;1957;96;peopleid: 2798;credited_as: A
516;12 Angry Men;us;1957;96;peopleid: 9559;credited_as: A
516;12 Angry Men;us;1957;96;peopleid: 1071;credited_as: A
902;4 luni, 3 săptămâni și 2 zile;ro;2007;113;peopleid: 10371;credited_as: D
902;4 luni, 3 săptămâni și 2 zile;ro;2007;113;peopleid: 9516;credited_as: A
902;4 luni, 3 săptămâni și 2 zile;ro;2007;113;peopleid: 5101;credited_as: A
```

Figure 12: Join Operation In Java

3.9 Multithreaded Parallel Processing

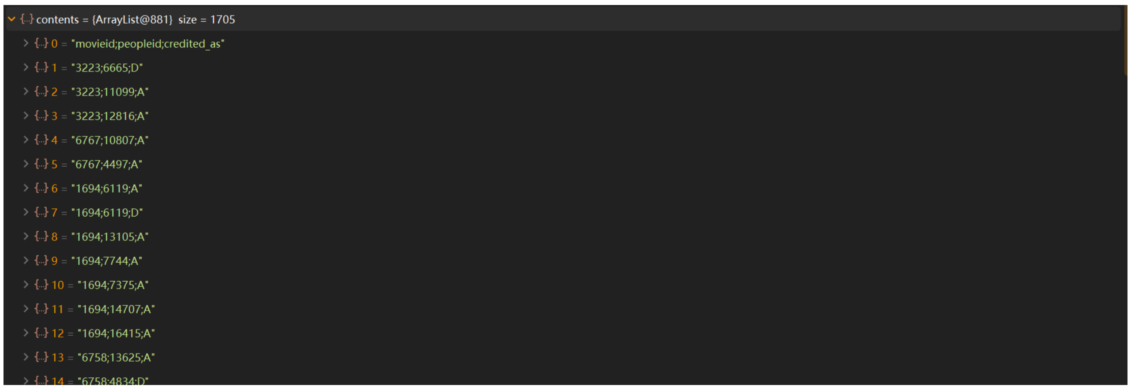
- **Multi-threaded file reading**:

  In Java, the basic idea for multi-threaded reading of large files is to **divide** the file into multiple segments, with each thread responsible for reading one of these segments. Afterward, the data read by each thread is merged and processed. My implementation code is as follows:

```
        public static List<String> multiThreadRead
        (String filePath, int threadCount) throws Exception {
    List<String> contents = new ArrayList<>();
    ExecutorService pool = Executors.newFixedThreadPool(threadCount);
    long fileSize = new File(filePath).length();
    int chunkSize = (int) Math.ceil(fileSize * 1.0 / threadCount);
    List<Future<List<String>>> futures = new ArrayList<>();
    for (int i = 0; i < threadCount; i++) {
        int startPos = i * chunkSize;
        Future<List<String>> future =
        pool.submit(new ReadFileTask(filePath, startPos, chunkSize));
        futures.add(future);
    }
    for (Future<List<String>> future : futures) {
        contents.addAll(future.get());
    }
    pool.shutdown();
    return contents;
}
static class ReadFileTask implements Callable<List<String>> {
    private String filePath;
    private int startPos;
    private int chunkSize;
    public ReadFileTask(String filePath, int startPos, int chunkSize) {
        this.filePath = filePath;
        this.startPos = startPos;
        this.chunkSize = chunkSize;
    }
    @Override
    public List<String> call() throws Exception {
        List<String> lines = new ArrayList<>();
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(startPos);
        String line = null;
        while ((line = file.readLine()) != null) {
            lines.add(line);
        }
        file.close();
        return lines;
    }
```

In this code snippet, I primarily encapsulated the **"multiThreadRead"** method. It starts by accepting the file path and the number of threads to return a list of strings containing the text content. Next, it creates a thread pool to enable concurrent reading of the file. It utilizes a list of **Future** objects for asynchronous processing to accommodate the reading tasks of each thread. Finally, after waiting for all threads to finish, the results are merged into the **"contents"** list.

For the method associated with each thread, I created an inner class named **"ReadFileTask."** It leverages the **RandomAccessFile** to achieve random access to the file and uses the call method to read the text content line by line, parts of the results are as follows:



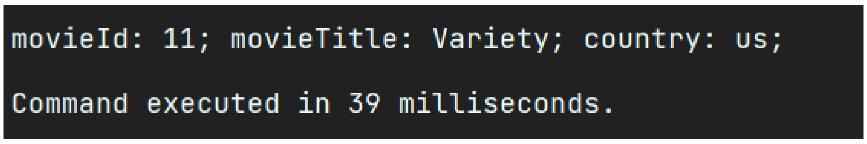
```
{...} contents = {ArrayList@881}  size = 1705
  {...} 0 = "movieid;peopleid;credited_as"
  {...} 1 = "3223;6665;D"
  {...} 2 = "3223;11099;A"
  {...} 3 = "3223;12816;A"
  {...} 4 = "6767;10807;A"
  {...} 5 = "6767;4497;A"
  {...} 6 = "1694;6119;A"
  {...} 7 = "1694;6119;D"
  {...} 8 = "1694;13105;A"
  {...} 9 = "1694;7744;A"
  {...} 10 = "1694;7375;A"
  {...} 11 = "1694;14707;A"
  {...} 12 = "1694;16415;A"
  {...} 13 = "6758;13625;A"
  {...} 14 = "6758;4834;D"
```

Figure 13: Multi-threaded File Reading

I added this method to the retrieval logic of the **"findMovieById"** method, modifying the file reading logic. The main code is as follows:

```
List<String> lines = FileUtils.multiThreadRead("src\\Database\\movies.txt", 4);
        for (String line : lines) {
            String[] splitArray = line.split(";");
            int movieId = Integer.parseInt(splitArray[0].trim());
            if (movieId == id) {
                StringBuilder movieInfo = new StringBuilder();
                movieInfo.append("Movie ID: ").append(id).append("\n");
                long endTime = System.nanoTime();
                System.out.println("Time: " + (endTime - startTime));
                return movieInfo.toString();
```

The result is as follow, of which the execution time is **39 milliseconds**, which is way faster than the original method.



```
movieId: 11; movieTitle: Variety; country: us;

Command executed in 39 milliseconds.
```

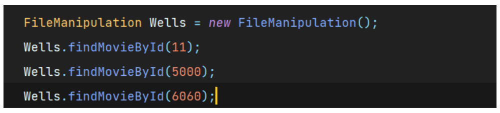
Figure 14: Result Of Multi-threaded File Reading

- **Multi-Threaded Queries:**

  When multiple query operations need to be executed, using a **single-threaded** sequential approach can be slow. In a database, multi-threaded querying is an important method to improve database retrieval performance. To implement concurrent queries in Java, we simply need to create a thread pool, assign each query task to a thread, and execute them concurrently. My main implementation code is as follows:

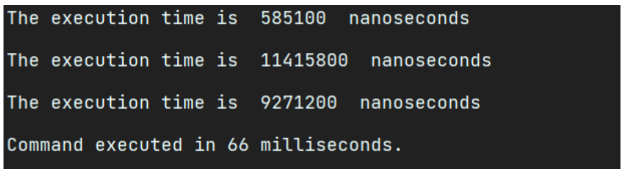
```
public FileManipulation{
threadPool = Executors.newFixedThreadPool(10);
}
private synchronized ExecutorService getThreadPool() {
    if (threadPool == null) {
        threadPool = Executors.newFixedThreadPool(10);
    }
    return threadPool;
}
```

In the **FileManipulation class constructor**, I created a thread pool and initialized the threads using a lazy-loading method. Finally, we only need to add the thread pool to each query method, the rusult is as follows:



```
FileManipulation Wells = new FileManipulation();
Wells.findMovieById(11);
Wells.findMovieById(5000);
Wells.findMovieById(6060);
```

(a) Commands



```
The execution time is  585100  nanoseconds

The execution time is  11415800  nanoseconds

The execution time is  9271200  nanoseconds

Command executed in 66 milliseconds.
```

(b) Result Of MultiThreaded Queries

Figure 15: MultiThreaded Queries

3.10 Optimizing Basic Operations Using Hash Index

The basic idea for performance optimization using hash indexing is to skip the step of reading files directly and instead utilize **automatically created index tables** to perform various fundamental operations.

Using the three created index tables, we can directly query a row that satisfies specific conditions by utilizing the **index-to-key** relationship in the Hashmap and applying **constraint statements** to retrieve the data that meets the criteria. I used this concept to rewrite the **findMovieById** method, and the implementation is as follows:

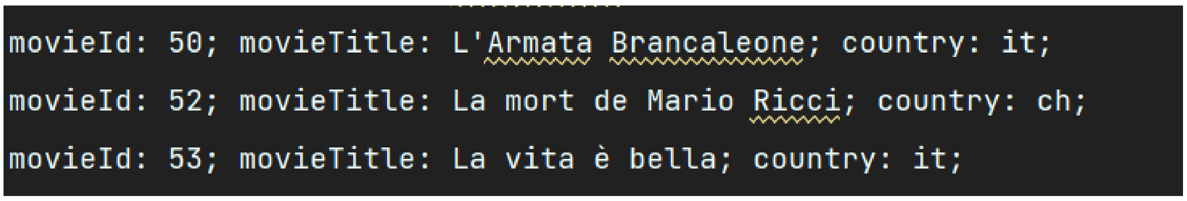
```
public void findMovieById_Hashmap(int id) {
threadPool.submit(() -> {
    long start = System.nanoTime();
    String information = titleIndex.get(id);
    long end = System.nanoTime();
    long executionTimeInMilliseconds = (end - start);
    System.out.println("Command␣executed␣in␣"
    + executionTimeInMilliseconds + "␣nanoseconds.");
    System.out.println(information);
});
```

The optimized query time is **23ms**, which outperforms the regular method's **46ms**.

For **update** and **delete** operations, you only need to perform the operations using the index table, and finally write the index table back to the original file to overwrite it, as demonstrated in the following code (using **delete operation** as an example):

```
     public void delete_ByHashmap(int id) throws IOException {
    BufferedWriter bufferedWriter =
    new BufferedWriter(new FileWriter("src\\Database\\movies.txt"));
    bufferedWriter.flush();
    if(canDeleteMovie(id)){
        titleIndex.remove(id);
    }
    else {
        throw new ForeignKeyConstraintViolationException
    }
    for (Map.Entry<Integer, String> entry : titleIndex.entrySet()) {
        bufferedWriter.write(entry.getValue());
        bufferedWriter.newLine();
    }
    bufferedWriter.close();
}
```

This code utilizes the **”remove”** method from Hashmap to perform deletion operations directly on the index table, while adhering to foreign key constraints. The execution time is **43ms**, which is **30ms faster** than the basic delete operation.



```
movieId: 50; movieTitle: L'Armata Brancaleone; country: it;
movieId: 52; movieTitle: La mort de Mario Ricci; country: ch;
movieId: 53; movieTitle: La vita è bella; country: it;
```

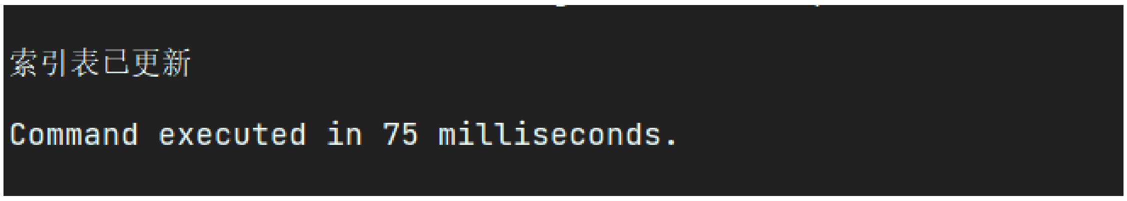
Figure 16: Delete Operation Using Hashmap In Java

It’s important to note that I’ve set up a **file monitor** to watch the files in the project directory. When the file monitor detects file updates, a **new** index table needs to be regenerated. This means that updates, deletions, and other operations will also affect the index table accordingly.

```
class FileWatcher {

    public static void startFileWatcher() {
        Path movies = Paths.get("src\\Database\\movies.txt");
        Path credit = Paths.get("src\\Database\\credit.txt");
        Path countries = Paths.get("src\\Database\\countries.txt");

        try {
            WatchService watchService = FileSystems.getDefault().newWatchService();

            movies.getParent().register(watchService, ENTRY_MODIFY);
            credit.getParent().register(watchService, ENTRY_MODIFY);
            countries.getParent().register(watchService, ENTRY_MODIFY);

            while (true) {
                WatchKey key = watchService.take();

                for (WatchEvent<?> event : key.pollEvents()) {
                    if (event.kind() == ENTRY_MODIFY) {
                        Path changedFile = (Path) event.context();
                        if (changedFile.equals(movies.getFileName())) {
                            loadIndexFromFile(movies.toString());
                        } else if (changedFile.equals(credit.getFileName())) {
                            loadCreditFromfile(credit.toString());
                        } else if (changedFile.equals(countries.getFileName())) {
                            loadCountryFromFile(countries.toString());
                        }
                        System.out.println("File modified: " + changedFile);
                        System.exit(0);
                    }
                }

                boolean valid = key.reset();
                if (!valid) {
                    break;
                }
            }
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

In this piece of code, I have created a watcher service, **'WatcherService'**, within the **'FileWatcher'** class to monitor three files: **'movie.txt'**, **'credit.txt'**, and **'countries.txt'**. I use the **'register'** method to register the directories to watch and the types of events to monitor with the **'WatchService'**. In this case, I'm only monitoring the **'ENTRYMODIFY'** event, which signifies changes to the file content.

The three directories, **movies'**, **'credit'**, and **'countries'**, are each registered for the **'ENTRYMODIFY'** event. Finally, when a file is modified, you can call the corresponding index table generation methods like **'loadIndexFromFile'**, **'loadCreditFromFile'**, and **'loadCountryFromFile'**, based on the actual situation. The result is as follow:



```
索引表已更新

Command executed in 75 milliseconds.
```

Figure 17: Watcher Service

3.11 Simulating Database Indexes Using Alternative Data Structures

Apart from the extensively used hash indexes mentioned earlier, databases employ various other indexing methods implemented through different data structures. I have chosen the more typical **B+ tree** index and the **Brin** index (block-level index). The main distinction lies in the fact that the latter uses information from each data block or each continuous set of data blocks as the index.

- **B+ Tree**

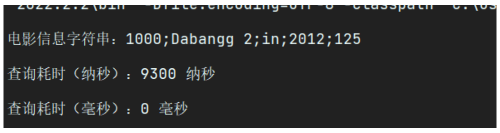
  B+ Tree is a type of multi-way search tree commonly used in databases for range queries, whereas hash indexes are only suitable for equality queries. Compared to B-trees, B+ trees can **automatically** rebalance and optimize query speed. To implement a B+ tree, the main tasks include defining the node data structure, determining the order of the tree based on the data volume, implementing node splitting and merging, as well as insertion, deletion, and other functions to maintain tree balance. Finally, the access functions need to be implemented. Below is a simplified **pseudocode** of my implementation:

```
         BPlusTree {
    Node {
        keys: String []
        strings : String []
        children : Node[]
        n: int

        Node(order: int)
    }
    root: Node
    order: int
    BPlusTree(order: int)\\I set the order to 10000
    insert (key: String, value: String)
    insert (node: Node, key: String, value: String)
    search(key: String): String
    search(node: Node, key: String): String
}
BplusTreeSearch {
    main(args: String [])
    readMovieDataFromFile(filePath: String): Map<String, String>
}
```

In this code segment, I've implemented the automatic updating and balancing of the B+ tree and defined insertion and query methods. The **insertion** method is crucial for ensuring B+ tree node updates. Here's a simplified **pseudocode** of the insertion method:

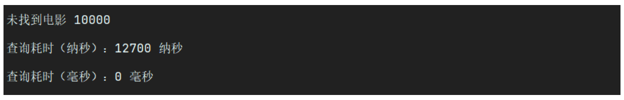
```
    Function Insert(node, key, value)
    If node Is Null Then
        Create a new node and add the key and value.
        Return the new node
    End If
    Find the insertion position (i).
    If node is full Then
        Split the node and move keys and values to a new node.
        Determine whether to insert into the left or right node based on the position (i).
        Promote the middle key to the parent node.
        Return the modified node.
    Else
        Insert the key and value at the appropriate position.
        Return the modified node.
    End If
End Function
```

The result obtained through the B+ tree query is as follows, with an execution time of **0ms.**, which is superfast.



```
电影信息字符串：1000;Dabangg 2;in;2012;125
查询耗时（纳秒）：9300 纳秒
查询耗时（毫秒）：0 毫秒
```

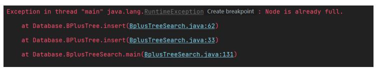
(a) Successful Search of B+ Tree



```
未找到电影 10000
查询耗时（纳秒）：12700 纳秒
查询耗时（毫秒）：0 毫秒
```

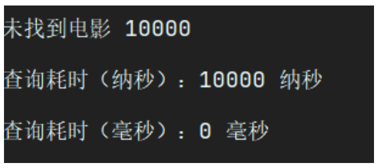
(b) Failed Search of B+ Tree

It's worth noting that the speed of queries through a B+ tree is significantly affected by the manually set **tree order**. A smaller order may result in insufficient nodes and an inability to query the desired positions, while a larger order may lead to excessive memory consumption and a rapid decline in query speed. Therefore, determining the appropriate tree order requires **testing** to find the optimal balance.



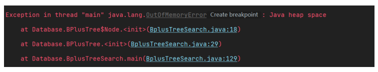
```
Exception in thread "main" java.lang.RuntimeException Create breakpoint : Node is already full.
    at Database.BPlusTree.insert(BplusTreeSearch.java:62)
    at Database.BPlusTree.insert(BplusTreeSearch.java:33)
    at Database.BplusTreeSearch.main(BplusTreeSearch.java:131)
```

(a) Order = 9000



```
未找到电影 10000
查询耗时（纳秒）：10000 纳秒
查询耗时（毫秒）：0 毫秒
```

(b) Order = 1000000



```
Exception in thread "main" java.lang.OutOfMemoryError Create breakpoint : Java heap space
    at Database.BPlusTree$Node.<init>(BplusTreeSearch.java:18)
    at Database.BPlusTree.<init>(BplusTreeSearch.java:29)
    at Database.BplusTreeSearch.main(BplusTreeSearch.java:129)
```

(c) Order = 10000000

Figure 19: Outputs Of Different Orders Of B+ Tree

- **Block Range Index**

A **Brin** index is a block-level index primarily used for handling range queries in large data tables. What sets it apart from **Hashmaps** and **B+ trees** is that it divides the data table into **blocks**, typically continuous blocks, with each block having an index entry that represents the **characteristics** of the data within that block.

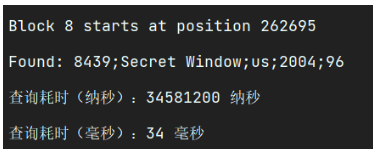
To implement a Brin index in Java, the first step is to **partition** the data into blocks, with each block containing a set of data records. Next, a **data digest** is generated to mark the **characteristics** of each block. Finally, I use data structures like **Hashmap** to create index entries for easy querying. Below is a simplified version of my implementation code:

```
    public static Map<Integer, Long> createBlockLevelIndex {
        Map<Integer, Long> blockIndex = new HashMap<>();
        int blockNumber = 1;
        long currentPosition = 0;
        int lineCount = 0;
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                lineCount++;
                if (lineCount > blockSize) {
                    blockIndex.put(blockNumber, currentPosition);
                    lineCount = 1;
                    blockNumber++;
                }
                currentPosition += line.length() + 1;
            }
        }
        blockIndex.put(blockNumber, currentPosition);
        return blockIndex;
```

This code divides the read file into blocks of a specified size and stores the starting position of each block in the **blockIndex**.

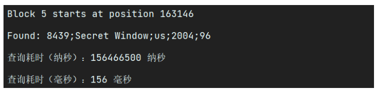
```
public static String retrieveDataFromBlock(String filePath, long blockStart, String query){
        try (RandomAccessFile file = new RandomAccessFile(filePath, "r")) {
            file.seek(blockStart);
            String line;
            while ((line = file.readLine()) != null) {
                if (line.contains(query)) {
                    return line;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

This piece of code first uses **RandomAccessFile** to open the file for random access in read-only mode (”r”). It then locates the file pointer to the starting position of the given block by calling **file.seek(blockStart)**. Afterward, it reads the file line by line, filtering for lines that contain the string **”query”**, and finally returns the results.The result is as follows, the **execution time** is fastest when **BlockNumber** is set to 8, which is **17 milliseconds**:



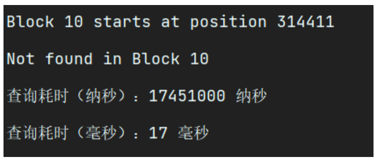
```
Block 8 starts at position 262695
Found: 8439;Secret Window;us;2004;96
查询耗时（纳秒）：34581200 纳秒
查询耗时（毫秒）：34 毫秒
```

(a) BlockNumber = 8



```
Block 5 starts at position 163146
Found: 8439;Secret Window;us;2004;96
查询耗时（纳秒）：156466500 纳秒
查询耗时（毫秒）：156 毫秒
```

(b) BlockNumber = 5



```
Block 10 starts at position 314411
Not found in Block 10
查询耗时（纳秒）：17451000 纳秒
查询耗时（毫秒）：17 毫秒
```

(c) BlockNumber = 10

Figure 20: Outputs Of Different BlockNumbers Of Brin Index

When I set the **block number** to a **lower** value, it increases the amount of data that needs to be queried, resulting in slower query speeds. On the other hand, when I set the block number to a **higher** value, it can cause the query to miss the specific data I need to retrieve, leading to query failures. Therefore, setting the block number for block-level queries is **crucial**.

3.12 Optimizing the Join Operation Using a Merge-Sort Algorithm

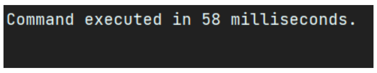
The **merge-join** algorithm is widely used in database join operations. Its fundamental concept involves sorting the files to be joined based on their associated key values. Subsequently, by individually selecting the next element from each input dataset and **merging** them into the output dataset, it yields a final, sorted result. Differing from the commonly used merge sort algorithm for sorting, I have implemented a more efficient **heap sort** algorithm. The **pseudocode** for the implementation is as follows:

```
    function BuildHeapSortAndSort(filePath, delimiter, columnCount):
    data = ReadDataFromFile(filePath, delimiter, columnCount)
    heapSort(data)
    return data

function ReadDataFromFile(filePath, delimiter, columnCount):
    data = []
    OpenAndReadFile(filePath, delimiter, columnCount, data)
    return data

function OpenAndReadFile(filePath, delimiter, columnCount, data):
    Open the file at 'filePath' for reading
    Read the first line and discard it
    While there are lines in the file :
        Parse the record from the line using 'delimiter' and 'columnCount'
        Add the parsed record to 'data'

function ParseRecord(line, delimiter, columnCount):
    Split 'line' into 'parts' using 'delimiter'
    Extract 'id' from 'parts'
    Extract 'movieTitle' and 'country' from 'parts'
    Create a 'value' from 'id'
    Create 'result' by formatting 'id', 'movieTitle', and 'country'
    Return [value, result]

function heapSort(data):
    n = Length of 'data'
    BuildMaxHeap(data, n)
    For 'i' from n-1 to 0:
        Swap 'data[0]' and 'data[i]'
        heapify(data, i, 0)

function BuildMaxHeap(data, n):
    For 'i' from n/2 - 1 down to 0:
        heapify(data, n, i)

function heapify(data, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    If 'l' < 'n' and 'data[l][0]' > 'data[largest][0]':
        Set 'largest' to 'l'
    If 'r' < 'n' and 'data[r][0]' > 'data[largest][0]':
        Set 'largest' to 'r'
    If 'largest' != 'i':
        Swap 'data[i]' and 'data[largest]'
```

Once the sorting is complete, the **join** operation between two files can be performed. The fundamental concept involves reading both tables separately. The table with the smaller key value reads the next row until the key values match, at which point the two rows are **merged**. The main code for this process is as follows:

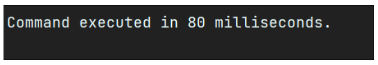
```
    while(line1 != null && line2 != null) {
                String[] split1 = line1.split(";");
                String[] split2 = line2.split(";");
                if(Integer.parseInt(split1[0]) < Integer.parseInt(split2[0])) {
                    line1 = reader1.readLine();
                } else if(Integer.parseInt(split1[0]) == Integer.parseInt(split2[0])) {
                    mergedData.add(line1.concat(line2));
                    line1 = reader1.readLine();
                    line2 = reader2.readLine();
                } else {
                    line2 = reader2.readLine();
                }
            }
            while(line1 != null) {
                mergedData.add(line1);
                line1 = reader1.readLine();
            }
            while(line2 != null) {
                mergedData.add(line2);
                line2 = reader2.readLine();
            }
```

When calling this method in the main function to perform the join operation, the execution time is **58ms**. In contrast, using regular methods and the hash index method for the join operation resulted in execution times of **80ms** and **39ms**, respectively.



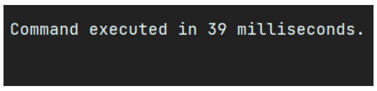
```
Command executed in 58 milliseconds.
```

(a) Join By Merge-Join



```
Command executed in 80 milliseconds.
```

(b) Join By Common Method



```
Command executed in 39 milliseconds.
```

(c) Join By Hash-Join

Figure 21: Output Of Different Methods Join Operations

4 Evaluations Of Different Functions

4.1 Evaluations Of Data Retrieval

In this project, I used retrieval methods with and without **indexes** in Java. Among the indexed methods, I employed three different types of indexes: **hash index**, **B+ index**, and **Brin index**. When executing the same query command, the retrieval method without an index took the longest time, which was **63ms**, as it required traversing every line in the file. Its time complexity is $O(n)$, and the space complexity is $O(1)$. This retrieval method's performance is significantly low when dealing with **large datasets**. The retrieval times using **hash index**, **B+ index**, and **Brin index** were **43ms**, **25ms**, and **28ms**, respectively. It's evident that the B+ index provides the **best performance** for this type of data retrieval.

As for the **hash index**, its space complexity is $O(n)$. Looking up key-value pairs in a HashMap has a time complexity of $O(1)$, and loading the HashMap has a time complexity of $O(n)$. Since we are comparing **retrieval times**, the retrieval time complexity of the hash index is $O(1)$. This leads to a significant performance improvement compared to directly traversing the file.

In the case of the **B+ index**, it needs to store all data, resulting in a space complexity of $O(n)$. **Insertion** operations have a time complexity of $O(logn)$, and retrieval time complexity is $O(logn)$. Therefore, the overall time complexity is $O(logn)$. This approach also offers a substantial performance enhancement compared to a straightforward traversal algorithm.

The **Brin** index, when building data blocks, has a space complexity of $O(n/m)$, where n is the total number of lines in the file, and m is the block size. In practice, m can be considered as a constant, so it's effectively $O(n)$. The time complexity for building **block-level** indexes is $O(n)$ because it requires a full file traversal. Searching within a specified block has a time complexity of $O(m)$, where m is the number of lines in a block. In this context, it's set to **1000**, which is considered a constant. File lookup time complexity is $O(1)$ because it directly locates the file position through seek. So, the overall time complexity is: $O(n + 1 + m) = O(n)$ However, since we can ignore the time taken to build **block-level** indexes, the effective time complexity is significantly lower than that of a traversal algorithm, resulting in significantly improved performance.

A direct horizontal comparison between these three index types doesn't provide significant reference value. After **conducting research**, it is found that in PostgreSQL databases, these three index types are typically used in combination for different retrieval scenarios:

- Hash Index:
  - Use Cases: Hash indexes are suitable for equality searches, meaning they are only effective when a key value is matched precisely.
  - Advantages: Hash indexes provide very fast lookup speeds because they use a hash function to map key values to index positions, resulting in nearly O(1) time complexity for lookups.
  - Limitations: Hash indexes do not support range queries or sorting operations as they do not store data in key order.
- B+ Tree Index:
  - Use Cases: B+ tree indexes are versatile and appropriate for various query operations, including equality searches, range queries, and sorting.
  - Advantages: B+ tree indexes offer efficient range query and sorting performance while supporting multiple query operations.
  - Limitations: Query performance of B+ tree indexes is related to the depth of the tree, so for large datasets, maintaining the index may be necessary to preserve performance.
- BRIN Index (Block Range INdexes):
  - Use Cases: BRIN indexes are suitable for range queries in large datasets, especially time-series data or other ordered data.

- Advantages: BRIN indexes do not require storing an index entry for every data row. Instead, they use block summary information, making them suitable for large datasets and reducing disk space usage.
- Limitations: BRIN indexes are not suitable for equality searches and are only used for range queries. Additionally, for unevenly distributed data, BRIN indexes may experience reduced query performance.

So, in the simple retrieval tasks of this project, a **hash index** should be the optimal choice. In the subsequent optimization steps, I primarily used a **hash index**. **The execution time for the same retrieval in DataGrip is 27ms**, which is a significant improvement compared to regular file reading. Further optimizing the database queries can enhance retrieval performance, a capability that regular Java file reading lacks.

It's worth noting that when using the hash index, I simulated the functionality of a primary key in a database. In comparison to situations without this constraint, the retrieval time improved by 20ms. This demonstrates the importance of establishing primary keys in a database.

4.2 Data Security Assessment

In Java, when simulating database operations for data updates and deletions, I encountered issues related to **data security**. In a typical database, there are constraints like **foreign keys** and **primary keys** to ensure data security, which the file system lacks by default. To address this, I used a hash index and a file watcher to simulate a basic form of foreign key constraints in a file-based system. I also updated the **index tables** in real-time after update and delete operations. However, this is just the **tip of the iceberg** when it comes to implementing methods for ensuring data security in a database.

To further optimize my approach, I believe I can consider the following aspects:

- Data Backup: Implement a regular data backup mechanism to create copies of the data files. This helps prevent data loss or corruption.
- Data Recovery Mechanism: Develop a data recovery mechanism to handle situations where data becomes corrupted or inconsistent. This could involve repairing damaged data files or rolling back to a previous data state.
- Logging: Implement transaction logging to track all data modification operations. This allows for transaction rollback or replay in case of errors, ensuring data consistency.
- Access Control: Utilize the file system's access control mechanisms to restrict access to data files. Ensure that only authorized users or processes can modify the data files. A basic principle is to separate the front-end from the back-end, hide methods, and enforce constraints within setter functions.

4.3 Evaluations Of Join Operations

In the database, the command to connect two specified tables takes **11ms** to execute. However, when using a traversal method in Java, it requires **80ms**, which is **7 times slower**. This is because it involves two full table scans of the files, resulting in higher time complexity. For each row in the "movies" table, reading the "credits" file has a time complexity of $O(m)$, where m is the number of rows in the "credits" file, resulting in a complexity of $O(nm)$. The space complexity mainly involves storing the "merged" file, which is $O(n+m)$ but can be approximated to $O(n)$.

After optimizing the algorithm in Java by using **hash indexing** and the **sorted-merge algorithm**, the execution times are **39ms** and **58ms**, respectively.

- Hash Index:

  Loading the "movies" data into a HashMap has a time complexity of $O(n)$, where n is the number of rows in the "movies" table. Loading the "credits" data into a HashMap has a time complexity of $O(m)$, where m is the number of rows in the "credits" table. Iterating through the "movies" HashMap and looking up the corresponding data in the "credits" HashMap has a time complexity of $O(1)$. Writing to the "merged" file has a time complexity of $O(n + m)$.

So, the **total** time complexity is: $O(n) + O(m) + O(n) + O(n + m) = O(n + m)$

The **space complexity** mainly involves the space used by the two HashMaps, which is $O(n + m)$. As you can see, by storing "movies" and "credits" in HashMaps, the time complexity for joining is reduced to $O(n+m)$ instead of the Cartesian product $O(n*m)$. This significantly improves performance, especially with **large datasets**.

- Sort-Merge Algorithm:

  Using heap sort to sort a single file by its key values has a time complexity of $O(nlogn)$. Reading each of the two files has a time complexity of $O(n)$ and $O(m)$, where n and m are the data sizes of the two files, respectively. Merging two sorted sequences has a time complexity of $O(n + m)$. Writing to the output file has a time complexity of $O(n + m)$.

  Therefore, the total time complexity is: $O(n) + O(m) + O(n + m) + O(n + m) = O(n + m)$

  The space complexity is mainly determined by the **mergedData** list used to store the sorted results and is O(n+m).

As can be seen, the optimized algorithm has a significant advantage over the straightforward iteration and reading methods. However, the performance of a database engine is still **superior** to these two methods, demonstrating that database engines utilize various optimization techniques. This is why we choose powerful **database engines** for data management.

4.4 Multithreaded Method Assessment

Multiple-row queries in a database will definitely leverage the principles of **multithreading** to enhance performance. Prior to multithreading, concurrent queries took **147ms**, while after implementing multithreading, the time reduced to **66ms**, resulting in a significant performance improvement. Based on my code implementation, I believe there are additional **improvements** that can be made:

- **Asynchronous Processing:**

  Submit blocking or time-consuming operations to a thread pool, allowing the main thread to continue executing other tasks, thus enhancing program responsiveness.

- **Cache Warm-Up:**

  Preload or compute potentially required results using multiple threads during program startup to accelerate subsequent access to cached data.

- **Data Chunking:**

  Divide large data sets into chunks and process each data chunk concurrently using multiple threads to reduce the workload of each thread and improve efficiency.

- **Task Decomposition:**

  Split tasks into multiple sub-tasks and process these sub-tasks with multiple threads to reduce overall execution time.

- **Controlling Shared Resources:**

  Use thread-safe data structures or locking mechanisms to avoid contention or conflicts among multiple threads for shared resources.

4.5 Conclusion

This project conducted a comprehensive comparison between database systems and Java file systems for various data operations like retrieval, updates, deletes and joins.

I implemented several **optimizations** in Java to simulate core database features and principles, which included creating indexes like **HashMaps**, implementing **primary keys**, **foreign keys**, **multithreading**, and using algorithms like **Merge-join** and **Heapsort**.

The evaluations showed that database systems clearly outperform plain Java file I/O in areas like query performance, scalability and security. My simulating database features in Java led to significant improvements in retrieval, join and concurrent query speeds. However, Java still lacks behind an optimized database engine.

I believe the project provided me with useful insights into the internal workings of database systems. It demonstrated the importance of elements like indexes, constraints, parallel processing and optimized algorithms that enable fast data processing on large datasets.

Applying database principles improved the performance of Java file processing. However, a full-fledged database system is still the **preferred choice** for serious data management due to better performance, scalability and resilience.

That's why it's a must for me to study database!

5 Appendix

1. This project is based on the framework uploaded by Pro.Yu on Blackboard, but all the remaining code is added by myself.
2. I searched for some database problems and Java code implementation resources, and below is the list:
   (a) https://blog.csdn.net/qq_34018603/article/details/103079281
   (b) https://blog.csdn.net/le_17_4_6/article/details/118699111
   (c) https://blog.csdn.net/weixin_34318956/article/details/90536190?
   (d) https://blog.csdn.net/hitits/article/details/90293142?
   (e) https://download.csdn.net/download/weixin_42133753/18527248?