Database System (2210)
COSC2406
Assignment 1

Student Name: Ziqing Yan
Student Number: s3749857

Task 1: Implement a Heap File in Java

In my java code, I first create a byte array for a page and initialise it with all 0s. When a record has been read from the .csv file, the 11 required fields will be extracted from the record by the split() function of String and column index of the 11 fields in the .csv file.

My heap file is composed of fixed-length records, so I choose a reasonable maximum length for each field of String type by observation(The bytes of each field taken up: 1. personName -- 60 bytes 2. birthDate -- 16 bytes 3. birthPlace label -- 150 bytes 4. deathDate -- 16 bytes(same as birthDate) 5. field label -- 200 bytes 6. genre label -- 230 bytes 7. instrument label -- 256 bytes 8. nationality label -- 50 bytes 9. thumbnail -- 275 bytes 10. wikiPageID -- 4 bytes 11. description -- 335 bytes). For the two date fields, some values may have two dates possibly because of different historical records by ancestors. So I choose to store them by 16 bytes, the first 8 bytes is for the number of milliseconds of the first date and the second 8 bytes for the number of milliseconds between the first and second date(If there is just one date, the second 8 bytes will be all 0s). If the date value is null, -1 of long type will be converted to bytes and stored in the first 8 bytes, because no date can be converted to -1 since the -1 millisecond has a very high precision. Similarly, for wikiPageID, -1 is a representation of null value. Add the numbers above and know that each record takes up 1592 bytes. Therefore, the recommended page size of the heap file is equal to or greater than 4096. If you enter a page size less than 1592, the program will report an error message.

When the value of a field has been transformed to bytes, the bytes will be written in the page array. The page array is initialised with all 0s, so the bytes of unused space of each field will be 0s(so is the unused space of each page). Since each record is fixed-length, the starting points(index) in the page for writing the bytes of each field is also fixed.

After writing one record in the page, the next record will be repeatedly written to the end of the last record(if there is enough space in the page to store the next record). The program will record the used space of the current page. If the remaining space is smaller than the size of the record, the page will be written out to the end of the heap file from in-memory. Then, the used space will be reset to 0 and the page array will be initialised again in order to prepare for writing on the next new page. Once all records have been read and written to the heap file, the program will stop.

I have tested the program in my AWS Linux instance with the command:
java dbload -p 4096 artist.csv

Below is how the result looks like:

```
[ec2-user@ip-10-88-184-192 Assignment1]$ java dbload -p 4096 artist.csv
The number of records loaded: 96300
The number of pages used: 48150
The number of milliseconds to create the heap file:12143
```

Tips of testing the program:

1. You should enter the command to test the program like this:

java dbload -p <page size> <data file path>

2. The page size must be equal to or greater than 1592. The recommended page size of the heap file is equal to or greater than 4096 bytes.
3. The heap file will be produced in the same folder of the dbload.java file.

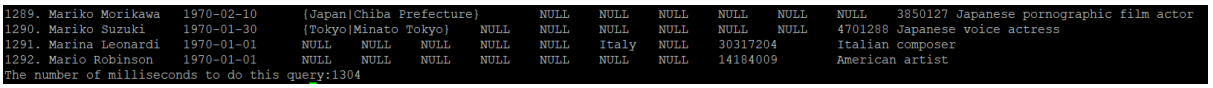
Task 2: Implement a range query on the heap file

The birthDate program firstly extracts the two dates from the command line. Then, it will parse the heap file in terms of its structure. Specifically, the bytes of a page size will be read from the heap file at a time. Each field has its own fixed length, so it can extract them at a fixed location of each record and remove 0s at the end of them.

The program first parses the birthDate field of each record and checks if the first date or the second date(if it exists) is between the start date and end date. Once the birthDate matches the range, the whole record will be printed to stdout.

I have tested the program in my AWS Linux instance using the heap file produced by Task1 with the command:
java birthDate heap.4096 19700101 19701230

Below is how the result looks like:



Tips of testing the program:

1. You should enter the command to test the program like this:

java birthDate <heap file> <start date> <end date>

2. The heap file must be the heap file produced by the program of dbload.java.

Task 3: Derby database and queries

All fields in the data set are directly related to the artists, so I choose to put all fields in one table. For simplicity, I only select 5 fields which are needed for query operations to load into Derby database. Since there is null value in the personName column and probably duplicate names, I create a new column ‘id’ as an unique identification for each record. When a record is inserted, an unique number will be created for the record.

I only delete the first four lines of the data file because they are headers. I use the command below to delete them:

sed -i '1,4d' artist.csv

After using the command **java org.apache.derby.tools.ij** to enter ij tool, I use the command **run 'task3.sql';** to execute the sql script for loading the data into Derby database(the artist.csv and task3.sql must be in the current folder). The script includes three parts. It first creates a new database with my student number and connects it. Then, it creates a table called **TASK3** which contains 6 fields, and regards id as primary key. Finally, it loads the specific fields of the data into the **TASK3** table using

CALL SYSCS_UTIL.SYSCS_IMPORT_DATA() build-in function and shows the time statistics. Below is the time statistics looks like:

```
Statement Name:
        null
Statement Text:
        CALL SYSCS_UTIL.SYSCS_IMPORT_DATA(null,'TASK3','PERSON_NAME,BIRTH_PLAC
E,GENRE,MOVEMENT,OCCUPATION','2,26,53,72,81','artist.csv',null,null,null,0)
Parse Time: 2
Bind Time: 5
Optimize Time: 0
Generate Time: 2
Compile Time: 9
Execute Time: -1649069915564
Begin Compilation Timestamp : 2022-04-04 10:58:35.546
End Compilation Timestamp : 2022-04-04 10:58:35.555
Begin Execution Timestamp : 2022-04-04 10:58:35.564
End Execution Timestamp : 2022-04-04 10:58:45.765
Statement Execution Plan Text:
null



1 row selected
```

The first query is:

```
SELECT COUNT(*) FROM TASK3
        WHERE UPPER(MOVEMENT) LIKE '%POST-IMPRESSIONISM%';
```

Result: The number of artists who are in the movement known as Post-Impressionism is 5.

First execution:

Compile Time: 77 Execute Time: 661

Second execution:

Compile Time: 77 Execute Time: 474

The second query is:

```
SELECT PERSON_NAME FROM TASK3
        WHERE UPPER(GENRE) LIKE '%PUNK ROCK%' AND
        UPPER(OCCUPATION) LIKE '%MUSICIAN%' AND
        UPPER(BIRTH_PLACE) LIKE '%AUSTRALIA%';
```

Result: Matt Doll, Anthony Mathews, Chris Cheney, Fred Negro and Guy Maddison is punk rock musician who was born in Australia.

First execution:

Compile Time: 71 Execute Time: 766

Second execution:

Compile Time: 71 Execute Time: 530

The first execute time is bigger than the second execute time for both queries. The reason probably is that the buffer pool contains the page related to the query after the first query operation. The second query only used the page in the buffer pool and didn't need to search from the disk, which can save a lot of time.