DataBase Management System: Simple-DB (1) Report

Group 3 : Hamza Jad Al Aoun and Meriem Driss

1 Summary

This lab involved implementing a simple database system in Java, that focuses on accessing stored data on disk, featuring table scanning, tuple retrieval, and buffer pool management. It consists of building a buffer pool, handling transactions, and developing iterator functionality for tuple access within a heap file structure.

2 The Database Class

The 'Database' class initializes and provides access to static components in the database system, including the catalog, buffer pool, and log files, ensuring system-wide access and management. This class was used mainly to call the bufferpool `Database.getBufferPool()` and the catalog `Database.getCatalog()` whenever needed.

Note: *The implementation of the* ***constructor*** *method, the* ***getters*** *and setters of the attributes was a routine work that was done for almost all the following classes. Additional implemented structures will be detailed below.*

3 Fields and Tuples

In developing the **TupleDesc class**, introducing a nested `TDItem` class facilitated the organization of field information. For flexibility in handling variable-length tuples, we chose to store the fields in a `Vector<TDItem>`. We also implemented an iterator as a means for traversing field `TDItem`. Additionally, the `merge()` method was designed to combine two `TupleDesc` instances, considering both field types and names. Last, the methods `equals()`, `hashCode()` and `toString()` facilitated equality comparisons, essential for future use as keys in data structures such as `HashMap`.
In the development of the **Tuple class**, we utilized an array of Fields that facilitated random access for setting and retrieving fields. To implement the iterator, we used the builtin method associated with `ArrayList` for the retrieval of an iterator `Arrays.asList(fields).iterator()`. Moreover, this class included a `RecordId` to represent the tuple's location on disk that further facilitates storage and retrieval operations.

4 Catalog

First, a **Table class** was implemented to store the table name, the file and the primary key.
The **Catalog class** takes a vector of tables `Vector<Tables>` to store the tables. This class has methods for adding new tables, ensuring that we don't have duplicates.
Also this class supports getters like `getTableId(name)`, `getDatabaseFile(tableid)` and `getPrimaryKey(tableid)`.
The `tableIdIterator()` method takes the ID's of all the tables inside our catalog, store them in a `Vector<Integer>` and return the iterator of this vector.

5 BufferPool

The implemented **BufferPool** manages a page cache using a `HashMap`, mapping PageIds to Pages for constant-time access. It handles a fixed number of pages specified by the constructor's `numPages` parameter.
`getPage()` method was very difficult and confusing. Our first approach was considering the cache as a vector of pages, but it failed some unit tests, which led us to work with HashMaps. This method retrieves pages with associated permissions, leveraging the HashMap for quick lookups and optimizing overall performance. First, if the requested page is already in the buffer pool, it returns it. Else, the method checks if there is enough space in the buffer pool to add the page. In this case, the page is read from disk through

`Database.getCatalog().getDatabaseFile(pid.getTableId()).readPage(pid)`, added to the buffer pool, and then returned. Otherwise, the method throws a DbException indicating insufficient space instead of eviction policy.

6 HeapFile Access

The **HeapPage class** serves as a manager for tuples within Heap File pages. Each Heap Page is uniquely identified by a `HeapPageId`, and contains a header that utilizes a bitmap for tracking tuple slot status using `isSlotUsed()`. To ensure tuple access we use tuple operations, like `readNextTuple()`, `insertTuple()`, and `deleteTuple()`. We also implemented `getPageData()` method, to generate a byte array representing the page content. HeapPage also required implementing a separate class `HeapPageIterator` that uses a Vector to store tuples from used slots during initialization and then iterates over non-empty tuples within a heap page.

HeapFile class having Files and TupleDesc as attributes With its constructor and getters, implements sort of high-level methods that allow the access of the tuples in a certain file. The core method of this class is the `readPage(PageId pid)` method that takes as an input the PageId. It utilizes the built-in `RandomAccessFile` class to open our file stored on the disk, reads the content of this file based on a calculated offset (depending on a previously defined page size multiplied by the page number associated with the input PageId), stores the bytes read into a buffer that will be passed to the `HeapPage((HeapPageId) pid, buffer)` constructor which will add this page to our main memory. A separate **HeapFileIterator class** was implemented. This iterator uses the `HeapPageiterator` to fetch the tuples iterators in the pages. The `open()` method starts with the first page TupleIterator, setting the page counter to 0 and the TupleIterator attribute. `hasNext()` will check if this Tupleiterator has a next, otherwise it will go to the next page having a non-null TupleIterator, upon calling this method multiple times, the TupleIterator attribute will be updated for non-null TupleIterators. `next()` retrieves the next tuple by returning `this.tupleIterator.next()`, rewind() resets the iteration, and close() terminates it by clearing the tuple iterator.

7 Operators

SeqScan class calls the underlying HeapfileIterator methods that will go down in our system, till returning iterators of our database tuples. Some simple getter methods were implemented with a `getTupleDesc()` method that returns TupleDesc with field names from the underlying HeapFile, prefixed with the tableAlias.

8 Concluding Remarks

We spent a considerable amount of time to completing the lab, totaling eight sessions, each lasting three hours.

Given our new exposure to Java, adapting to the language proved to be an obstacle. Our progress was very slow at the beginning, but we got used to it afterwards.

The main challenges revolved mainly around implementing certain methods, particularly the different iterators as well as the getPage method. Despite these challenges, we found the process beneficial as it deepened our understanding of the data structures introduced during the lectures. This hands-on experience not only reinforced theoretical concepts but also taught us Java programming.