Lab 2

DBsys

Elia Faure-Rolland
Tuomas Rahkola
Jonas Thalmeier

26/11/2024

EURECOM
Sophia Antipolis

Design Overview

In this lab, we implemented core modules for the SimpleDB system, focusing on managing heap files and sequential scanning. Key design choices included:

- **HeapFile and HeapPage Structure**: The heap file structure was designed to store pages sequentially on disk. Each page contains a header (bitmask) and fixed-size tuple slots. Header bits indicate whether corresponding tuple slots are occupied. Also in the HeapFile class we fixed the DEFAULT_PAGE_SIZE to 4096, in order to avoid calling BufferPool methods when reading from the disk.
- **Tuple and TupleDesc Management**: Tuples were defined with schemas provided by `TupleDesc`, which describes field names and types. These components enable consistent tuple manipulation across the database.
- **Buffer Pool for Page Management**: The `BufferPool` class caches pages, minimizing disk access. It retrieves pages using the `getPage` method, which checks for cached pages or reads from disk if unavailable.
- **Iterative Design in SeqScan**: The `SeqScan` operator was implemented to sequentially iterate over tuples in the database using the `DbFileIterator`.

Key Problems Encountered

Four major challenges arose during the implementation:

1. Hashing Function in `HeapPageId`

The initial implementation of the `hashCode` method in `HeapPageId` used a naive approach by concatenating table and page IDs as strings and parsing them into integers. This caused an **integer overflow**, leading to unexpected behavior when `HeapPageId` was used as a hash key. To resolve this, we adopted Java's built-in `Objects.hash` method:

```
return Objects.hash(tableId, pgNo);
```

This approach ensures a reliable hash code without risking overflow, leveraging robust hashing provided by the Java standard library.

2. Header Size Calculation in `HeapPage`

The `getHeaderSize` method initially used integer division:

```
return (int) Math.ceil(this.getNumTuples() / 8);
```

This caused an error in calculating the number of bytes required for the page header due to the truncation of integer division. The issue was corrected by casting the divisor to `double`, ensuring proper division:

```
return (int) Math.ceil(this.getNumTuples() / 8.0);
```

This adjustment ensures accurate header size computation, especially for pages with a small number of tuples.

3. Equals method in `HeapPageId`

The `Equals` method in HeapPageId originally used:

```
return this.tableId == other.tableId && this.pgNo == other.pgNo;
```

Because this did not correctly compare the two, it caused the getPage() method to not be able to find previous pages from cache. The issue was fixed by changing the `== to .equals()`, that compares the two correctly:

```
return this.tableId.equals(other.tableId) && this.pgNo.equals(other.pgNo);
```

The failure of getPage was noticed on the ScanTest.testCache test, which failed due to bufferpool, getting full.

4. Issue with `next()` Method in `HeapFile` Iterator

The original implementation of the `next()` method in the `HeapFile` iterator encountered an issue due to multiple calls to check for the next tuple within the method. This redundancy led to the page index being incremented twice under specific conditions, causing the iterator to skip pages unexpectedly. Surprisingly all the tests worked anyhow.

To prevent problems in the next labs, we simplified the method to ensure that the check for the next tuple occurs only once. This change prevents redundant increments and ensures proper tracking of the current page and tuple. As a result, the iterator now retrieves tuples sequentially and reliably without skipping pages.

Running the First Query

After resolving all encountered issues and bugs, the scan query code provided in the assignment was implemented as a class named `ScanQuery`, saved in a Java file of the same name within the `simpledb` directory. For testing purposes, a .txt file containing 100 rows and 4 columns of integers ranging from 1 to 10 was created. This text file was converted into a .dat file using the provided command. The query code was then executed, and the output successfully matched the input data.

Conclusions

The project emphasized the importance of careful consideration in method implementation, especially when relying on fundamental operations like hashing and division. Addressing these issues improved both correctness and robustness in handling database pages. Overall we spent about 20-30 hours combined in the span of 3 weeks. Most of it was spent implementing the code, but significant part was also spent on bug fixing. It was also hard to pinpoint the reason for the errors using the provided tests due to the recursive nature of the database.