DATABASE MANAGEMENT SYSTEM: SIMPLE-DB (2) REPORT

Group 3: Hamza Jad Al Aoun and Meriem Driss

1 Summary

This second lab extends SimpleDB's functionality. The tasks involve designing eviction policies, implementing filter and join operators, supporting SQL aggregates, enabling heap file mutability, and developing insertion and deletion operators.

2 Filter & Join

The `Predicate` class in SimpleDB is responsible for managing tuple comparisons based on specified conditions. It supports operations such as `EQUALS`, `GREATER_THAN`, `LESS_THAN`, etc. The constructor initializes the field number, operation (`Op`), and operand. The `filter()` method assesses whether a given tuple satisfies the predicate conditions and offers a human-readable representation through `toString()`. The `Join` operator orchestrates the relational join operation, merging tuples from two child operators based on a given `JoinPredicate`. It ensures accurate tuple combination, managing the concatenation process. `JoinPredicate` is pivotal for `Join`, comparing fields in two tuples and specifying conditions for their combination.

`Filter` introduces relational select capabilities, applying a predicate to filter tuples from a child operator. It iterates over tuples, selectively returning those meeting specified conditions, with key components including the predicate, tuple descriptor, and lifecycle methods (`open()`, `close()`, `rewind()`). This functionality facilitates targeted tuple retrieval based on user-defined conditions.

3 Aggregates

The `IntegerAggregator` class enables the computation of SQL aggregates, supporting operations like COUNT, SUM, AVG, MIN, and MAX, along with GROUP BY functionality. Instantiated with parameters such as `gbfield`, `gbfieldtype`, `afield`, and `what`, it utilizes `grpAggResMap` and `grpCounterMap` for efficient storage of aggregate results and tuple counts per group. The `mergeTupleIntoGroup` method handles tuple integration into aggregate calculations, considering both grouping and non-grouping scenarios. For grouping, the method updates the group's aggregate results and tuple counts based on the specified aggregation operation. The `iterator` method constructs an `OpIterator` over group aggregate results, adhering to the aggregation operation and grouping conditions. Efficient data storage using HashMaps ensures constant-time access, optimizing performance.

The `StringAggregator` class in SimpleDB parallels the `IntegerAggregator`, supporting basic SQL aggregates for a set of `StringFields`. It differs in handling string-based fields and exclusively supports the COUNT operation, as specified in its constructor.

In `Aggregate` class, `Aggregate` operator uses the Aggregators implemented above. It groups tuples from the child iterator and performs aggregate calculations. The operator's iterator utilizes the Aggregator's iterator to acquire the results of a Group By query. The constructor dynamically initializes either an `IntegerAggregator` or `StringAggregator` based on the type of the aggregation field. The `getTupleDesc()` method constructs the output tuple description, taking into account grouping conditions.

4 HeapFile Mutability

The `insertTuple` method in `HeapPage` performs a linear search through the page's header, utilizing `isSlotUsed`, to find a free space. It then inserts the tuple into that slot, updating the header accordingly. The `deleteTuple` method extracts the `RecordId` from the tuple to be deleted, removes it from the `tuples` array, and updates the header.The `insertTuple` method in `HeapFile` linearly searches through the file for a page with a free slot. If none is found, it creates a new page, inserts the tuple into that page using `HeapPage`'s `insertTuple`, and appends it to the file. The `deleteTuple` method determines the `PageId` from the tuple to be deleted and uses `HeapPage`'s `deleteTuple` function to delete the tuple.

In `Bufferpool` with the `insertTuple` omethod tuples are being inserted into the database file by iterating through the pages. Pages affected by the insertion are marked as dirty to ensure data consistency. On the other hand, the `deleteTuple` method removes tuples from the file and marks the corresponding pages as dirty. both use methods implemented above for the insertion and deletion of pages and files.

5 Insertion & Deletion

In `Insert.java` and `Delete.java` being operators on the top level beside there constructors, we have the `fetchNext` method in both as a core method that iterates through the child iterator, utilizing the `BufferPool` methods for efficient insertions and deletions. A `flag` variable is used ensuring that the method is executed only once successfully. also a `counter` is used to keeps track of the number of items inserted or deleted during the iteration. The result is a single tuple, containing one integer field that represents the count of affected tuples.

6 Page Eviction

The chosen eviction policy is a basic random eviction approach. When the buffer pool surpasses the limit, the `evictPage()` method randomly selects a page from the pool for eviction. The randomness introduces an element of unpredictability, providing simplicity while effectively managing memory.

The `flushPage()` method uses the `writePage()` method, ensuring that the page data is written to disk. Additionally, the `discardPage()` method is implemented to remove a page from the buffer pool without persisting it on disk. The `evictPage()` method starts by collecting all page IDs from the buffer pool. It randomly selects a page ID and proceeds to flush it to disk `flushPage()`, and then discards it from the pool using `discardPage()`. The random selection is achieved by generating a random index based on the size of the page ID collection. This process continues until the buffer pool size is within the specified limit (`numPages`).

The design decision to implement a random eviction policy was driven by the need for simplicity and efficiency. The tradeoff involves a lack of determinism in the eviction process, but for the purposes of this lab, simplicity and ease of implementation were prioritized.

7 Query Walkthrough & Conclusion

For the first query, we added the jointestexample.java and we tried to run the system on two files having common entries in F1. The result is in the annex page. For the second query, we ran the parser of our Database passing to it a new table stored in data.txt. The query was easily executed in the parser showing the content of this table. As shown in the figure in the annex page, it was our first query, so we had a transaction id = 0.

To conclude, we spent about five sessions during the holidays, each lasting three hours. Thanks to the first lab, our familiarity with Java significantly increased, resulting in a faster and more efficient development process in this lab. Despite the difficulties encountered in the lab, it served as a valuable learning experience that contributed to a deeper understanding of the data structures discussed in class.

8 Annex Page

```
┌──(kali㉿ IbrahimAsus)-[~/Downloads/DataBase/Lab1-starter-code]
└─$ java -classpath dist/simpledb.jar simpledb.jointestexample
10      11      12      1320    11      12
14      13      15      140     13      135

┌──(kali㉿ IbrahimAsus)-[~/Downloads/DataBase/Lab1-starter-code]
```

```
┌──(kali㉿ IbrahimAsus)-[~/Downloads/hamza/DataBase/Lab1-starter-code]
└─$ java -jar dist/simpledb.jar parser catalog.txt
Added table : data with schema f1(INT_TYPE), f2(INT_TYPE),
Computing table stats.
Done.
SimpleDB> select d.f1, d.f2 from data d;
Started a new transaction tid = 0
Added scan of table d
Added select list field d.f1
Added select list field d.f2
The query plan is:
  π(d.f1,d.f2),card:0
  |
scan(data d)

d.f1    d.f2
------------------
1       10
2       20
3       30
4       40
5       50

 5 rows.
Transaction 0 committed.
----------------
0.14 seconds

SimpleDB> _
```