report lab3DbSys

BOUCHY Elliot MAYAUD Victor

January 2024

1 Introduction

This lab task is an advanced continuation of our exploration of database management systems, focusing specifically on improving SimpleDB functionality. Unlike the fundamental aspects discussed in the previous lab, this task entrusts us with the implementation of a set of operators for SimpleDB. These operators will facilitate table changes, such as inserting and deleting records, while allowing selections, joins, and aggregates.

2 Our different implementations

2.1 Predicate

The `Predicate` class in the supplied code is designed to facilitate tuple comparisons in the context of the SimpleDB database. It encapsulates elements such as the field number (`_field`), the comparison operation (`_op`), and the field value (`_operand`). Thanks to its constructor and access methods like `getField`, `getOp`, and `getOperand`, the class allows the creation of predicates for specific comparisons. The `filter` method performs the actual comparison between a given tuple and the criteria defined by the predicate.

2.2 JoinPredicate

The JoinPredicate class in SimpleDB is designed for tuple field comparisons in the Join operator. It includes attributes like field1, field2, and op (e.g., GREATER THAN). The constructor sets up the comparison, and the filter method applies it to tuples. Access methods provide field indices and the operator. Overall, JoinPredicate encapsulates conditions for comparing tuple fields in Join operations.

2.3 Filter

The Filter class in SimpleDB serves as a selection operator, applying a given predicate to filter tuples from a child operator. Its methods manage predicate access, tuple description, operator open/close/rewind, and child operator retrieval. The fetchNext method iterates through child tuples, applying the predicate and returning satisfying tuples. The class employs a while loop and returns null if no filtered tuples remain. In essence, the Filter class facilitates relational selection, allowing the application of selection conditions in a relational database context.

2.4 Join

The Join class implements the relational join operation between two operators. It accepts two child operators and a join predicate. The class's methods access the join predicate, the names of the join fields, the description of the resulting tuples, open, close and rewind the operator, and retrieve and define the child operators.The fetchNext method implements the next read using a while loop and two nested loops. It traverses the tuples of the two child operators, applying the join predicate, and returning tuples that satisfy the condition. The joinTuples method is used to merge the matching tuples.

2.5 IntegerAggregator

In the code section of our SimpleDB project, we have a constructor and two main methods. The constructor takes several inputs for proper grouping and aggregation, including the group-by field index (gbfield), its type (gbfieldtype), the aggregate field index (afield), and the aggregation operation (what). The mergeTupleIntoGroup method handles the merging and aggregation of tuples based on these constructor parameters. A notable addition is the AggregateInfo class, used for storing aggregate values and counts, which is particularly useful for AVG calculations. Finally, the iterator method creates an OpIterator to iterate over the aggregated results. Depending on whether grouping is used, each tuple in the iterator represents either a pair (group value, aggregate value) or a single (aggregate value).

2.6 StringAggregator

This code is very similar to the IntegerAggregator code except we manipulate only string so we considere only the COUNT operator.

2.7 Aggregate

In the "Aggregate.java" class of our SimpleDB project, we implement an aggregation operator for computing SQL aggregates like sum, average, max, and min, with support for single-column grouping. The constructor initializes an IntegerAggregator or StringAggregator based on the aggregate field's type. It supports key methods for group and aggregate field information and operations. The open() method sets up the aggregation process, and fetchNext() retrieves aggregated results. The class also includes methods for output tuple description and iterator management, ensuring seamless aggregation functionality within SimpleDB.

2.8 HeapPage

The class HeapPage represents a page in a heap file and implements the Page interface used by the BufferPool. It contains methods for creating a page from disk data, retrieving the number of tuples on the page, computing the header size, and generating a byte array representing the page's contents for serialization. Additionally, it provides methods for inserting and deleting tuples, marking the page as dirty, checking if it's dirty, and managing slots.

2.9 HeapFile

The class HeapFile in the SimpleDB package represents a file that stores tuples in no particular order. It works closely with HeapPage and implements the DbFile interface. The constructor initializes the file and tuple descriptor. Methods include returning the file backing the HeapFile, generating a unique ID based on the file's absolute name, getting the tuple descriptor, reading and writing pages, determining the number of pages, and inserting or deleting tuples.

2.10 BufferPool

insertTuple: This method inserts a tuple into the buffer pool, updates the associated pages, and marks them as "dirty" (modified). It uses the catalog manager to obtain the database file associated with the table, then inserts the tuple into this file. Modified pages are marked as "dirty" and cached.

deleteTuple: This method deletes a tuple from the buffer pool by acquiring write locks on the relevant pages and marking them as "dirty". It also uses the catalog manager to obtain the database file associated with the table, then deletes the tuple from this file. Modified pages are marked as "dirty" and cached.

2.11 Insert

The Insert class in SimpleDB is designed for inserting tuples into a specific table within a database. It initializes with a transaction identifier, a child operator providing tuples, and the target table ID. The class ensures data format consistency between the input tuples and the target table. Its core functionality lies in the fetchNext method, where it iterates over tuples from the child operator, inserting them into the table via the BufferPool. This process also involves tracking the count of inserted tuples, returning this count as a single-field tuple. The class also includes lifecycle management methods (open, close, rewind) for initializing and resetting the operator, along with methods to manage its child operator (getChildren, setChildren). Overall, the Insert class is crucial for maintaining data integrity and executing insert operations in the SimpleDB system.

2.12 Delete

The Delete class in SimpleDB functions similarly to the Insert class, but instead of adding tuples, it is responsible for removing them from the specified table in a database. Initialized with a transaction identifier and a child operator, the class reads tuples from the child and deletes them from their respective tables. The deletion process is managed through the BufferPool, ensuring that changes are consistently applied to the database. A key method in this class is fetchNext, which iterates over the tuples provided by the child operator, removing each from the database. It keeps a count of the number of tuples deleted, ultimately returning this count in a single-field tuple. This count provides feedback on the number of tuples successfully deleted during the operation.

3 Difficulties

We spent about two hours understanding the structure of what was asked of us, as well as the role of each method, and then about ten hours implementing and testing them. We delved into comprehending the intricate structure of the various classes involved in the lab. Understanding how these classes interacted with each other posed a significant challenge. And understand the role of certain classes.

4 Conclusion

In this lab, we delved into the complexities of database management systems, focusing on implementing operations such as table modifications (insert and delete records), selections, joins, and aggregates. A key highlight involved addressing the management of the buffer pool, tackling the challenge of handling references to more pages than can fit in memory over the database's lifetime through the design of an eviction policy.