Database Systems- Project

Lab 2: SimpleDB Operators

Chirag Shivakumar 1004996, Devanshi Joshi 1005162, Suthar Prachi Jayesh 1005421

# Design Decisions made during the Project

## Exercise 1- Filter and Join

### This exercise shows how to create Filter and Join operators in SimpleDB OpIterator classes to perform sophisticated queries. The join condition and fields being joined are represented by getter and setter functions in the class. The passage also provides an exercise to apply these solutions and pass exams.

### predicate.java

This code defines the Predicate class which compares tuples to a specified field value using a specified operation. It also provides methods for getting the field number, operator, operand, and for filtering a tuple based on the predicate.

### joinPredcate.java

This code defines a predicate used by the Join operator to compare the fields of two tuples and return true if they satisfy the predicate. It includes methods to get the two fields being compared and the operator used to compare them.

### filter.java

This code filters tuples based on a predicate and returns only the tuples that pass the filter.

### join.java

## In a database system, the code implements a Join operator that takes two relations and a predicate, joins the tuples that fulfill the join requirement, and returns the result as a new tuple.

## Exercise 2- Aggregates

The SimpleDB operator implements the five SQL aggregates (COUNT, SUM, AVG, MIN, MAX) and grouping utilizing an Aggregator interface to handle basic SQL aggregates with a GROUP BY clause. The Aggregator inserts a new tuple into an existing aggregate calculation, and the client code can retrieve an OpIterator of aggregation results.

### Aggregate.java

The IntegerAggregator and StringAggregator classes implement the Aggregator interface and offer methods for integrating a new tuple into an aggregate's existing calculation. They are built with an aggregate operation type (such as COUNT or AVERAGE) and a field index that the aggregate should be performed on.

### IntegerAggregator.java (Explained more in non-Trivial parts of code)

This class constructor accepts the 0-based index of the tuple's group-by field, or NO GROUPING if there is no grouping, the type of the group by field, the 0-based index of the tuple's aggregate field, and the aggregation operator.

The mergeTupleIntoGroup method adds a new tuple to the aggregate, grouping it as specified in the constructor. The method reads the tuple and retrieves the current group-by field and the aggregate field before performing the aggregate operation (COUNT, SUM, AVG, MIN, or MAX). The outcomes are saved in a HashMap.

The iterator method generates an OpIterator from the aggregate results of a group. If there is no grouping, a single tuple with the aggregate value is generated. Otherwise, a pair tuple containing the group-by field and the aggregate value is produced.

### StringAggregator.java

The simpledb.execution package's StringAggregator class may compute aggregate values (currently only COUNT) over a set of StringFields. It requires numerous parameters, including the group-by field index and the aggregate field index. It uses the mergeTupleIntoGroup function to group tuples supplied in the constructor and returns the result as an OpIterator over group aggregate values. If a group is utilized, the OpIterator returns a pair of groupVal and aggregateVal, else it returns a single aggregateVal. The StringAggregator class's iterator method provides an OpIterator through the group aggregate results and methods for opening, closing, rewinding, and retrieving the TupleDesc.

## Exercise 3- Heap File Mutability

### HeapPage.java

HeapPage is a Java class file that implements the Page interface used by BufferPool to hold data for one page of HeapFiles and corresponds to SimpleDB's HeapPage. It has a distinct HeapPageId, a TransactionId, an array of Tuples, a byte array for the header, and a TupleDesc. The class includes methods for creating a HeapPage from a byte array of data, counting the number of tuples on a page, calculating the size of a page's header, and retrieving the original state of a page. Moreover, HeapPage includes methods for setting and getting the page's HeapPageId as well as for constructing a byte array that encodes the page's data.

### HeapFile.java

HeapFile, stores tuples without regard to order. The file is made up of a number of fixed-size pages. The program's code contains methods for initializing the file, retrieving the file's ID and tuple descriptor, reading and writing pages, adding and removing tuples, and returning an iterator over all of the tuples.

## Exercise 4- Insertion and Deletion

### Insert.java

This operator is used to insert tuples from a child operator into a designated table is implemented in the provided code. It verifies that the child's TupleDesc corresponds to the table that the tuples are to be inserted into. The insert operation does not look for duplicate tuples before inserting; instead, it travels through the BufferPool.

### Delete.java

This is Delete operator uses the buffer pool to read tuples from its child operator and delete them from the database to which they belong. The number of deleted records is contained in a 1-field tuple that the operator returns.

## Exercise 5- Page Eviction

### BufferPool.java

The BufferPool class has methods to control the acquisition and release of locks on pages as well as the reading and writing of pages into memory from disk. It is invoked by access methods to retrieve pages, and it fetches the right pages from the right place. To speed up subsequent requests, the BufferPool keeps frequently used pages in memory cache.

# Non-Trivial parts of code

### Exercise 2- Implementation of the Integer Aggregate Class

The IntegerAggregator class is intended to execute aggregate actions on a collection of IntFields, such as calculating the count, total, average, min, or max. It can handle both grouping and non-grouping scenarios, which means it can aggregate either the complete set of IntFields or subsets of IntFields based on a grouping criteria. This makes the code non-trivial because it takes careful consideration of numerous edge circumstances and inputs to achieve accurate and efficient aggregation results.

1. This code must support both grouping and non-grouping cases. There is only one aggregate value to compute if there is no grouping. If there is grouping, there is only one aggregate value for each group. Depending on whether grouping is used or not, the code handles both circumstances by constructing distinct sorts of tuples and tuple descriptors.
2. The Code must handle many types of aggregating operators and apply the proper logic for each operator. The COUNT operator, for example, must keep track of the number of occurrences of each group, whereas the AVG operator must keep track of both the count and sum of each group in order to compute the average.
3. The code covers a variety of edge circumstances, such as null values, empty groups, and errors thrown during execution.

# Justify any changes made to the API- None

# Any missing or incomplete elements in your code- None