Programming Assignment #1 Part 3
CAS CS 460: Introduction to Database Systems
Due Date and Time: 11/10, 23:59 on gradescope

In this assignment, you will write a set of operators for SimpleDB to implement table modifications (e.g., insert and delete records), selections, joins, and aggregates. These will build on top of the foundation that you wrote in PA 1 to provide you with a database system that can perform simple queries over multiple tables.

Additionally, you will also utilize the buffer eviction code you have developed in previous assignment. You do not need to implement transactions or locking now.

The remainder of this document gives some suggestions about how to start coding, describes a set of exercises to help you work through the assignment, and discusses how to hand in your code. This assignment requires you to write a fair amount of code, since you have a midterm to take, we encourage you to start early!

1. Getting started

1.1. Adding skeleton code for assignment 3

You should begin with the code you submitted for part 1 and 2 (if you did not submit code, or your solution didn't work properly, contact us to discuss options). We have provided you with extra test cases and .java files for this assignment that are not in the original code distribution you received, download them here. Again, the unit tests we provide are to help guide your implementation along, but they are not intended to be comprehensive or to establish correctness.
You will need to add these new files to your release. The easiest way to do this is to copy and paste all of them in their corresponding folders.

1.2 Implementation hints

As before, we encourage you to read through this entire document to get a feel for the high-level design of SimpleDB before you write code.

We suggest exercises along this document to guide your implementation, but you may find that a different order

makes more sense for you. As before, we will grade your assignment by looking at your code and verifying that you have passed the test for the ant targets testand systemtest. See Section 3.4 for a complete discussion of grading and list of the tests you will need to pass.

Here's a rough outline of one way you might proceed with your SimpleDB implementation; more details on the steps in this outline, including exercises, are given in Section 2 below.

- Implement the operators Filter and Join and verify that their corresponding tests work. The Javadoc comments for these operators contain details about how they should work. We have given you implementations of Project and OrderBy which may help you understand how other operators work.
- Implement IntegerAggregator and StringAggregator. Here, you will write the logic that actually computes an aggregate over a particular field across multiple groups in a sequence of input tuples. Use integer division for computing the average, since SimpleDB only supports integers. StringAggegator only needs to support the COUNT aggregate, since the other operations do not make sense for strings.
- Implement the Aggregate operator. As with other operators, aggregates implement the DbIterator interface so that they can be placed in SimpleDB query plans. Note that the output of an Aggregateoperator is an aggregate value of an entire group for each call to next(), and that the aggregate constructor takes the aggregation and grouping fields.
- Implement the methods related to tuple insertion, deletion, and page eviction in BufferPool. You do not need to worry about transactions at this point.
- Implement the Insert and Delete operators. Like all operators, Insert and Delete implement DbIterator, accepting a stream of tuples to insert or delete and outputting a single tuple with an integer field that indicates the number of tuples inserted or deleted. These operators will need to call the appropriate methods in BufferPool that actually modify the pages on disk. Check that the tests for inserting and deleting tuples work properly.
- Note that SimpleDB does not implement any kind of consistency or integrity checking, so it is possible to insert duplicate records into a file and there is no way to enforce primary or foreign key constraints.

At this point you should be able to pass all of the tests in the ant systemtest target, which is the goal of this assignment.

Finally, you might notice that the iterators in this assignment extend the Operator class instead of implementing the DbIterator interface. Because the implementation of next/hasNext is often repetitive, annoying, and error-prone, Operator implements this logic generically, and only requires that you implement a simpler readNext. Feel free to use this style of implementation, or just implement the DbIterator interface if you prefer. To implement the DbIterator interface, remove extends Operator from iterator classes, and in its place put

implements DbIterator.

2. SimpleDB Architecture and Implementation Guide

2.1. Filter and Join

Recall that SimpleDB DbIterator classes implement the operations of the relational algebra. You will now implement two operators that will enable you to perform queries that are slightly more interesting than a table scan.

- *Filter*: This operator only returns tuples that satisfy a Predicate that is specified as part of its constructor. Hence, it filters out any tuples that do not match the predicate.
- *Join*: This operator joins tuples from its two children according to a JoinPredicatethat is passed in as part of its constructor. We require a simple nested loops join implementation and a hash join implementation respectively, but you may explore more interesting join implementations.
- Describe your implementation in your writeup.

Exercise 1. Implement the skeleton methods in:

- src/simpledb/Predicate.java
- src/simpledb/JoinPredicate.java
- src/simpledb/Filter.javasrc/simpledb/Join.java
- src/simpledb/HashEquiJoin.java

At this point, your code should pass the unit tests in PredicateTest, JoinPredicateTest, FilterTest, JoinTest, and HashEquiJoinTest. Furthermore, you should be able to pass the system tests FilterTest, JoinTest and HashEquiJoinTest.

2.2. Aggregates

An additional SimpleDB operator implements basic SQL aggregates with a GROUP BY clause. You should implement the five SQL aggregates (COUNT, SUM, AVG, MIN, MAX) and support grouping. You only need to support aggregates over a single field and grouping by a single field.

In order to calculate aggregates, we use an Aggregator interface which merges a new tuple into the existing

calculation of an aggregate. The Aggregator is told during construction what operation it should use for aggregation. Subsequently, the client code should call Aggregator.mergeTupleIntoGroup() for every tuple in the child iterator. After all tuples have been merged, the client can retrieve a DbIterator of aggregation results. Each tuple in the result is a pair of the form (groupValue, aggregateValue), unless the value of the group by field was Aggregator.NO_GROUPING, in which case the result is a single tuple of the form (aggregateValue).

Note that this implementation requires space linear in the number of distinct groups. For the purposes of this assignment, you do not need to worry about the situation where the number of groups exceeds available memory.

Exercise 2. Implement the skeleton methods in:

- src/simpledb/IntegerAggregator.java
- src/simpledb/StringAggregator.java
- src/simpledb/Aggregate.java

At this point, your code should pass the unit tests IntegerAggregatorTest, StringAggregatorTest, and AggregateTest. Furthermore, you should be able to pass the AggregateTest system test.

2.3. HeapFile Mutability

Now, we will begin to implement methods to support modifying tables. We begin at the level of individual pages and files. There are two main sets of operations: adding tuples and removing tuples.

Removing tuples: To remove a tuple, you will need to implement deleteTuple. Tuples contain RecordIDs which allow you to find the page they reside on, so this should be as simple as locating the page a tuple belongs to and modifying the headers of the page appropriately.

Adding tuples: The insertTuple method in HeapFile.java is responsible for adding a tuple to a heap file. To add a new tuple to a HeapFile, you will have to find a page with an empty slot. If no such pages exist in the HeapFile, you need to create a new page and append it to the physical file on disk. You will need to ensure that the RecordID in the tuple is updated correctly.

Exercise 3. Implement the remaining skeleton methods in:

- src/simpledb/HeapPage.java

- src/simpledb/HeapFile.java

To implement HeapPage, you will need to modify the header bitmap for methods such as insertTuple() and deleteTuple(). You may find that the getNumEmptySlots() and isSlotUsed() methods we asked you to implement in PA 1 serve as useful abstractions. Note that there is a markSlotUsed method provided as an abstraction to modify the filled or cleared status of a tuple in the page header.

Note that it is important that the HeapFile.insertTuple() and HeapFile.deleteTuple() methods access pages using the BufferPool.getPage() method; otherwise, your implementation of transactions in the next assignment will not work properly.

Implement the following skeleton methods in src/simpledb/BufferPool.java:

- insertTuple()
- deleteTuple()

These methods should call the appropriate methods in the HeapFile that belong to the table being modified (this extra level of indirection is needed to support other types of files like indices in the future).

At this point, your code should pass the unit tests in HeapPageWriteTest and HeapFileWriteTest. We have not provided additional unit tests for HeapFile.deleteTuple()or BufferPool.

2.4. Insertion and deletion

Now that you have written all of the HeapFile machinery to add and remove tuples, you will implement the Insert and Deleteoperators.

For plans that implement insertand deletequeries, the top-most operator is a special Insertor Deleteoperator that modifies the pages on disk. These operators return the number of affected tuples. This is implemented by returning a single tuple with one integer field, containing the count.

- *Insert*: This operator adds the tuples it reads from its child operator to the tableid specified in its constructor. It should use the BufferPool.insertTuple()method to do this.
- *Delete*: This operator deletes the tuples it reads from its child operator from the tableid specified in its constructor. It should use the BufferPool.deleteTuple()method to do this.

Exercise 4. Implement the skeleton methods in:

- src/simpledb/Insert.java
- src/simpledb/Delete.java

At this point, your code should pass the unit tests in InsertTest. We have not provided unit tests for Delete. Furthermore, you should be able to pass the InsertTest and DeleteTest system tests.

3. Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made. If you used something other than a nested-loops join, describe the tradeoffs of the algorithm you chose.
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe how long you spent on the assignment, and whether there was anything you found particularly difficult or confusing.

3.1. Collaboration

Please indicate clearly who you worked with, if anyone, on your writeup.

3.2. Submitting your assignment

Please only submit source code files on **gradescope**! Do **not** submit .class files or anything else. If you worked in a group make a **GROUP** submission. Also do not forget to mention the names of the people that collaborated in the write-up!

3.3. Academic Honesty

Assignments must be completed by you or your group. Representing the work of another person as your own is expressly forbidden. This includes "borrowing", "stealing", copying programs/solutions or parts of them from others. We will use an automated plagiarism checker. Cheating will not be tolerated under any circumstances. See the CAS Academic Conduct Code, in particular regarding plagiarism and cheating on exams. A student suspected to violate this code will be reported to the Academic Conduct Committee, and if found culpable, the

student will receive a grade of "F" for the course.
We hope you enjoy hacking on this assignment!