CSI 508. Database Systems I – Fall 2018

Programming Assignment III

The total grade for this assignment is 100 points. The deadline for this assignment is **11:59 PM, December 18, 2018**. *Submissions after this deadline will not be accepted.* Students are required to enter the UAlbany Blackboard system and then upload a .zip file (in the form of [first name]_[last name].zip) that contains the Eclipse project directory and a short document describing:

- any missing or incomplete elements of the code
- any changes made to the original API
- the amount of time spent for this assignment
- suggestions or comments if any

In this programming assignment, you need to implement two relational operators (selection and aggregation). You first need to run Eclipse on your machine and import the "`query_processor`" project (see Appendix A). Please generate a Javadoc API document and then take a look at the document as well as the source code to familiarize yourself with this assignment. For this assignment, we have provided you with a set of classes (see particularly `ProjectionOperator` which will help you understand how a relational operator can be implemented as well as `SelectionOperator`, `AggregageOperator`, and `Aggregator` that you need to complete by adding more code). Your code will be graded by running a set of unit tests and then examining your code. Passing unit tests does not necessarily guarantee that your implementation is correct and efficient. Please strive to write correct and efficient code. If you have questions, please contact the TA(s) or the instructor. The remainder of this document describes the components that you need to implement.

Part 1. Selection (60 points)

In this part, you need to complete the code in `SelectionOperator.java`. For this, it might be helpful to understand the implementation of `ProjectionOperator`. Each `SelectionOperator` outputs, given a series of `Tuple`s from another operator, the `Tuple`s that satisfy a predicate specified as part of its constructor. In other words, it filters out all `Tuple`s that do not match its predicate. Each `SelectionOperator` operator has its own `ExpressionEvaluator` which can evaluate an expression (i.e., the predicate) for each input `Tuple`. Given `ExpressionEvaluator evaluator` and an input `Tuple t`, `(evaluator.evaluate(t) == Boolean.TRUE)` indicates that `t` satisfies the predicate that `evaluator` uses (i.e., the `SelectionOperator` must output `t`). The `SelectionOperator` can also be viewed as an iterator over all `Tuple`s output by itself. Your implementation should not keep all of the output `Tuple`s in the memory since it may be infeasible (i.e., too many `Tuple`s to fit into the memory) in practical situations. Instead, the `SelectionOperator` (which is also viewed as an iterator) needs to find, whenever the `hasMext()` and `next()` methods are called, the next input `Tuple` that satisfies the predicate (on-demand, pull-based pipeline; see Section 12.7.2 in the textbook).

The constructor/methods to complete are as follows:

- `SelectionOperator(Operator input, String predicate)`: constructs a `SelectionOperator`. This constructor needs to create an `ExpressionEvaluator` for evaluating the predicate on each input `Tuple` and may do some additional work (depending on your implementation) to support the `hasNext()` and `next()` methods.

- `outputSchema()`: returns the output schema of the `SelectionOperator`. This output schema is the same as the input schema of the `SelectionOperator`. Consider using a method provided by a super-type of `SelectionOperator` to get the input schema of the `SelectionOperator`.

- `hasNext()`: determines whether or not the `SelectionOperator` has the next output `Tuple`.

- `next()`: returns the next output `Tuple`.

- `rewind()`: rewinds the operator in order to retrieve all output `Tuple`s again from the first output `Tuple`.

When all of the above methods are implemented correctly, `SelectionOperatorTest` will produce the following output:

```
input schema: {ID=java.lang.Integer, Location=java.lang.Integer, Celsius=java.lang.Double}
input tuples:
[0, 0, 0.0]
[1, 0, 1.0]
[2, 0, 2.0]
[3, 0, 3.0]
[4, 1, 4.0]
[5, 1, 5.0]
[6, 1, 6.0]
[7, 1, 7.0]
[8, 2, 8.0]
[9, 2, 9.0]
[10, 2, 10.0]
[11, 2, 11.0]

predicate: Celsius < 5
output tuples:
[0, 0, 0.0]
[1, 0, 1.0]
[2, 0, 2.0]
[3, 0, 3.0]
[4, 1, 4.0]

predicate: Celsius > 5
output tuples:
[6, 1, 6.0]
[7, 1, 7.0]
[8, 2, 8.0]
[9, 2, 9.0]
[10, 2, 10.0]
[11, 2, 11.0]
```

In the above example, there are 12 input `Tuple`s all with three attributes (`ID`, `Location`, and `Celsius`). The first `SelectionOperator` produces 4 output `Tuple`s based on its predicate (`Celsius < 5`). The second `SelectionOperator` produces 6 output `Tuple`s based on its predicate (`Celsius > 5`).

Part 2. Aggregation (40 points)

In this part, you need to complete `AggregateOperator.java` and `Aggregator.java`. An `Aggregator` groups all `Tuples` (by user-specified attributes) from an input `Operator` and outputs, for each group of `Tuples`, a `Tuple` that represents/summarizes that group of `Tuples` (e.g., for each location code, show the minimum and maximum temperatures in Celsius) . An `AggregateOperator` is an operator that uses an `Aggregator` and supports the basic iterator capabilities (`hasNext()` and `next()` methods).

The methods/constructors to complete are as follows:

- `createOutputSchema(AggregateFunction[] aggregateFunctions)` in `AggregageOperator.java`: constructs and then returns the output schema of the `AggregageOperator`. The output schema consists of the grouping attributes (specified at the time of constructing `AggregageOperator`) and additional attributes (one for each `AggregateFunction` specified at the time of constructing `AggregageOperator`) for storing the aggregate values from the `AggregateFunctions` (e.g., one grouping attribute `Location` and another attribute for storing the minimum temperature in Celsius). This method needs to construct a new `RelationSchema` which requires an array storing the names of the attributes to include in the `RelationSchema` and the types of these attributes. For each grouping attribute to include in the `RelationSchema`, the name and type of that grouping attribute can be obtained from the input schema of the `AggregageOperator` (e.g., if `Location` is a grouping attribute, the name and type of that attribute can be obtained from the input scheme). On the other hand, for an `AggregateFunction f`, the name and type of the attribute from `f`, can be obtained by calling `f.toString()` and `f.valueType()`, respectively. For example, the name and type of the attribute for storing the result of applying `Maximum` to `Celsius` can be obtained by calling `toString()` and `valueType()` on that `Maximum` function.

- `Aggregator(Operator input, RelationSchema outputSchema, String[] groupingAttributeNames, Class<?>[] aggregateFunctionTypes, String[] aggregationAttributeNames)` in `Aggregator.java`: constructs an `Aggregator`. Given an input tuple `t`, the `Aggregator` needs to extract the values of the grouping attributes (e.g., `Location` value 0) and then finds the `AggregateFunctions` for that combination of grouping values (e.g., `Minimum` and `Maximum` that have been applied to the `Celsius` attribute from all `Tuples` whose `Location` value is 0). Then, the `Aggregator` needs to update all of these `AggregateFunctions` based on tuple `t` (e.g., update the minimum value if the `Celsiums` value of `t` is smaller than the previous minimum). The above implementation approach 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.

- `iterator()` in `Aggregator.java`: returns an iterator over the output `Tuples` of the `Aggregator`. The details of these output `Tuples` are explained above (refer to the output schema of the `AggregageOperator`).

When all of the above methods are implemented correctly, `AggregateOperatorTest` will produce the following output:

```
input schema: {ID=java.lang.Integer, Location=java.lang.Integer, Celsius=java.lang.Double}
input tuples:
[0, 0, 0.0]
[1, 0, 1.0]
[2, 0, 2.0]
[3, 0, 3.0]
[4, 1, 4.0]
```

```
[5, 1, 5.0]
[6, 1, 6.0]
[7, 1, 7.0]
[8, 2, 8.0]
[9, 2, 9.0]
[10, 2, 10.0]
[11, 2, 11.0]

grouping attributes: [Location]
aggregate functions: [Minimum(Celsius), Maximum(Celsius), Average(Celsius)]
output schema: {Location=java.lang.Integer, Minimum(Celsius)=java.lang.Double,
Maximum(Celsius)=java.lang.Double, Average(Celsius)=java.lang.Double}
[0, 0.0, 3.0, 1.5]
[1, 4.0, 7.0, 5.5]
[2, 8.0, 11.0, 9.5]
```

In the above example, an `AggregatorOperator` groups 12 input `Tuple`s by only one attribute (`Location`) and then, for each group of `Tuple`s, applies three `AggregateFunctions` (`Minimum`, `Maximum`, and `Average`) to the `Celsius` attribute. It then produces, for each `Location` value (i.e., 0, 1, 2), one tuple summarizing the corresponding group of `Tuple`s: the tuple contains the grouping attribute value (i.e., the `Location` value), and the three aggregate values from the three `AggregateFunction`s (`Minimum`, `Maximum`, and `Average`).

Appendix A. Importing a Java Project

1. Start Eclipse. If Eclipse runs for the first time, it asks the user to choose the workspace location. You may use the default location.

2. In the menu bar, choose "File" and then "Import". Next, select "General" and "Existing Projects into Workspace". Then, click the "Browse" button and select the "`query_processor.zip`" file contained in this assignment package.

3. Once the project is imported, you can choose `ProjectionOperatorTest.java`, `SelectionOperatorTest.jav` or `AggregateOperatorTest.java` and then run the program.