Assignment 2: SQL Planning and Joins (100 points)

In this assignment you have three major tasks to complete:

- Implement a simple query planner that translates parsed SQL expressions into query plans that can be executed (50 points)
- Complete the implementation of a nested-loop join plan-node that supports inner and left-outer joins (20 points)
- Create some automated tests to ensure that your inner- and outer-join support works correctly (20 points)
- Submit your work along with a design document describing your efforts
- (10 points are also assigned to repository management tasks, such as whether every commit has a useful log message, log messages follow the 50/72 format, etc.)

These tasks are described in the following sections.

There are also several extra-credit tasks listed at the end of the assignment, for the ambitious.

Before You Start

If you were unable to complete a bug-free implementation of tuple deletion or tuple updating, please let us know and we will happily provide a working implementation for you to drop in.

Additionally, if your attempts to make inserts faster are buggy, feel free to revert your codebase to use the original heap-file implementation. None of the remaining assignments depend on fast tuple inserts. Your only long-term penalty will be waiting slightly longer… ☺

NanoDB Query Evaluator and Plan-Node Lifecycle

NanoDB's query evaluator uses a demand-driven pipeline. All plan-nodes must operate in the same way for this to work properly. You can refer to the API documentation for specific methods on `edu.caltech.nanodb.plans.PlanNode` (the abstract base-class of all plan-nodes), although the below description will give you an overview of how it all fits together.

When a `PlanNode` subclass is constructed, various essential details are passed to the constructor to initialize the node. All plan-nodes are allowed to have up to two child plan-nodes, available as the `leftChild` and `rightChild` fields in the `PlanNode` class. These fields are marked `protected` so that all subclasses can access them. For such plan-nodes, the child plans are usually passed to the constructor. Note that there are also plan-nodes that do not require any children, such as the file-scan node. In these cases, the constructor specifies the necessary details for the plan node.

As with relational algebra operations, every plan-node specifies the schema of its output tuples. (Every node also has an associated cost of evaluation, which we will use in the next assignment. For now you can ignore the cost information; it will likely be `null` for the time being.) These values must be computed once an entire query plan is assembled; this is done by calling the `prepare()` method on the root plan-node. Note that the query evaluator expects that all plans have already been prepared before handing them to the evaluator.

Before a plan-node can produce any rows, its `initialize()` method *must* be invoked. The `initialize()` method sets up evaluation-time resources that the plan-node requires, so that the next invocation of `getTuple()` will return the first tuple from the plan-node. (Generally, this method invokes `super.initialize()` to run parent-class initialization, then it initializes its own

internal members, and then finally it invokes `initialize()` on its left and right child-nodes, as appropriate.)

The `getTuple()` method is where the plan-node implements its logic. This method returns tuples generated by the plan-node, one by one in sequence. As long as more tuples are available, this method will return each tuple in sequence, but once there are no more tuples then `getTuple()` will start returning `null`. (It is not an error to call `getTuple()` again after it has returned `null`, but it shouldn't be necessary so it is strongly discouraged.)

Many plan-nodes must call `getTuple()` on their children multiple times before they can return the next tuple to their caller. You can look at the `SelectNode` class' `getTuple()` method for a simple example of this. Some plan nodes must even consume all data from their children before producing any results, such as in the case of sorting or hash-based grouping.

Some plan-nodes must also iterate over the tuples from their children multiple times in order to produce their results. The nested-loop join node is a perfect example of this; for each tuple produced by the left subplan, it must iterate over *all* tuples produced by the right subplan. A subplan can be restarted by calling `initialize()` on it again to reset it to the beginning of its tuple-sequence. For some plan-nodes such as the file-scan node, resetting to the beginning is a simple matter of going back to the start of the table-file. However, other plan-nodes may not store their results; such nodes must recompute their results from scratch every time the node is restarted.[1]

When the evaluator is finished executing a plan, it calls `cleanUp()` on the root plan-node. Each plan-node performs its own cleanup, and then calls `cleanUp()` on its child node(s). Most plan-nodes require no cleanup, but this is critical when a node uses external memory for computing results so that temporary files can be deleted.

NanoDB Query Planner

NanoDB is currently unable to execute anything except the simplest SQL statements, because it has no way of generating more complex execution plans from a SQL statement. This is one of your major tasks for the assignment.

NanoDB allows different planners to be plugged into the database by providing a `Planner` interface (`edu.caltech.nanodb.queryeval` package). The two most relevant methods are:

```
SelectNode makeSimpleSelect(String tableName, Expression predicate,
                            List<SelectClause> enclosingSelects)
```

> This method is used by `UPDATE` and `DELETE` statements to implement the optional `WHERE` clause. These DML statements require that the tuples produced by the plan can be used as L-values, which is how this method is currently implemented.
>
> The "simplest" planner also uses `makeSimpleSelect()` to handle "`SELECT * FROM t WHERE ...`" queries.

```
PlanNode makePlan(SelectClause selClause,
                  List<SelectClause> enclosingSelects)
```

> This method produces a query plan for a standard `SELECT...FROM...WHERE` statement. Every `SELECT` statement is parsed into a `SelectClause` that contains all components of the

[1] NanoDB doesn't implement any external-memory algorithms, but if a plan-node creates any intermediate data files when computing its results, subsequent calls to `initialize()` may need to purge or delete these intermediate files.

statement, including the list of values in the SELECT clause, the tree of join expressions in the FROM clause, and the predicate in the WHERE clause. (SelectClause, FromClause, etc., are in the queryast – "Query AST" – package.)

Note that every query plan produced by a planner implementation *must* be prepared by the planner. The query evaluator expects that the planner has completed this step.

This week you will create a planner class called SimplePlanner, starting with the SimplestPlanner implementation. Copy the SimplestPlanner.java file to SimplePlanner.java, then modify the class to be called SimplePlanner. (Don't modify SimplestPlanner.)

Next, update SimplePlanner.makePlan() so that it supports the following features:

- Basic SELECT statements that involve zero or more tables being joined together. (For example, "SELECT 3 + 2 AS five;" is a valid SQL query. The project plan-node is able to support situations where it has no child plan, and no expression references a column name.) Join predicates may appear in the WHERE clause, or in an ON clause.
- Basic joins using the "SELECT … FROM t1, t2, …" syntax, as well as joins with an ON clause. You do not need to support NATURAL joins or joins with the USING clause in this planner.
- Left- and right-outer joins using the nested-loop join node (which you will work on in the next section). You do not need to support full-outer joins.
- Subqueries in the FROM clause.
- Grouping and aggregation, where both grouping attributes and aggregates may be expressions, and where a given SELECT expression may include multiple aggregate operations. You are encouraged to use the supplied HashedGroupAggregateNode class. You will need to provide some limited validation of these expressions; details are given in a subsequent section.
- ORDER BY clauses, using the supplied edu.caltech.nanodb.plans.SortNode class.

You are <u>not</u> required to support these features in this planner (they are part of a future lab):

- Subqueries in the SELECT or WHERE clauses.
- Correlated evaluation.

Here are some additional notes and guidelines:

- The above list is rather long, but you can complete it in steps and test your work as you go. You may want to do joins last, since this is the most complicated part. Join support can also be completed in steps, e.g. handling "SELECT * FROM r, s WHERE …" first, then adding support for other join operations.

- You should try to generate the minimal plan necessary, but don't go overboard trying to achieve this. For example, only include a Project plan-node if the SELECT-clause doesn't specify a trivial project.[2] Only generate a Select plan-node if the query specifies a WHERE predicate.

- This is a <u>simple</u> query planner. Your plan doesn't need to be efficient. It just needs to produce the correct results. We will worry about efficiency in a future assignment.

[2] A "trivial project" operation is one that can be removed because it doesn't do anything. For example, the SQL statement "SELECT * FROM t WHERE a < 5" wouldn't require a project, but "SELECT b, c + 3 FROM t WHERE a < 5" would. See SelectClause.isTrivialProject().

- Note that some `FromClause` objects will include a join predicate and others will not, depending on how the SQL statement is written. For example, "`SELECT * FROM t1, t2 WHERE t1.a = t2.a`" will generate a different plan tree than "`SELECT * FROM t1 JOIN t2 ON t1.a = t2.a`", although the rows retrieved will be identical.

- When you need to pad a tuple with `NULL` values, see the `TupleLiteral(int numCols)` constructor (in the `expressions` subpackage). You may even want to cache this "all-`NULL`s" tuple in your plan node when it is needed, so that evaluating outer joins will be more efficient.

Finally, you will notice that `SimplestPlanner` (and your new `SimplePlanner` class) derives from the `AbstractPlannerImpl` class. This abstract base-class is an excellent place to put functionality that would be common to all query planners. For example, if you create helper methods to support grouping/aggregation, joins, etc., consider putting them in the abstract base-class so that you can use them in the next planner you write.

The EXPLAIN Command

Like many databases, NanoDB provides an `EXPLAIN` command that can be used to see what the database decides to do when you run a query. You can use `EXPLAIN` to display the plans generated by your planner, to see if they actually makes sense. For example, if you have a table like this:

```
CREATE TABLE t ( a INTEGER, b VARCHAR(20) );
```

You can type "`EXPLAIN SELECT b FROM t WHERE a < 5;`", and (once your planner works) NanoDB will respond with something like this:

```
Explain Plan:
    Project[values:  [T.B]] cost is unknown
        FileScan[table:  T, pred:  T.A < 5] cost is unknown
```

(The "cost is unknown" because we haven't yet implemented plan costing. Next time!)

Grouping and Aggregation

As discussed in class, grouping and aggregation is challenging to translate from SQL into relational algebra, because the `SELECT` clause mixes grouping/aggregation and projection, and the `HAVING` clause mixes selection and aggregation. The `SELECT` and `HAVING` expressions must be scanned for aggregate expressions so that the grouping plan-node can be initialized correctly, and also so that the aggregates may be removed and replaced with placeholder variables.

Scanning and Transforming Expressions

NanoDB's `expressions` package includes a mechanism for traversing and transforming expression trees. The `Expression` base-class has a `traverse()` method that performs a traversal of an entire expression hierarchy. The method takes an `ExpressionProcessor` implementation, with an `enter(Expression)` method and a `leave(Expression)` method; as each expression node is visited, the `enter()` method is called with that node as an argument, then the node's children are traversed (again with `enter()` and `leave()` being called), and then finally the `leave()` method is called again with that node as an argument.

You will also note that `leave()` returns an `Expression`; this allows the `ExpressionProcessor` implementation to mutate the expression as it is traversed. If an expression processor wants to replace the current node with some other expression, it simply returns it from `leave()`, and the original node will be replaced.

This functionality does require that you use these methods in a specific way. For example, if you were examining select-clause expressions for aggregates, you would need to do something like this:

```
for (SelectValue sv : selectValues) {
    // Skip select-values that aren't expressions
    if (!sv.isExpression())
        continue;

    Expression e = sv.getExpression().traverse(processor);
    sv.setExpression(e);
}
```

Note that the original expression is extracted, traversed, and then replaced by whatever the `traverse()` call returns.

Identifying and Replacing Aggregate Functions

Function calls are certainly a kind of expression, and they are represented by the `FunctionCall` class in the `expressions` package. You can call the `FunctionCall.getFunction()` method to retrieve the specific function that is being invoked, and see if it is an instance of an aggregate function. Something like this:

```
if (e instanceof FunctionCall) {
    FunctionCall call = (FunctionCall) e;
    Function f = call.getFunction();
    if (f instanceof AggregateFunction) {
        ...  // Do stuff
    }
}
```

Of course, if you wish to replace an aggregate function call with a column-reference, like the approach described in class, you can look at the `ColumnValue` expression class. It takes a `ColumnName` object that holds the column name, and optionally the table name. (This `ColumnName` class also represents wildcard expressions; you can read the Javadocs for the details.)

Final Guidance

A good approach may be to create an `ExpressionProcessor` implementation that identifies aggregate functions in expressions, and maps each one to an auto-generated name. Therefore the class would need some kind of mapping from auto-generated names to the corresponding aggregates they represent. This should be used to process the `SELECT` expressions and the `HAVING` expressions, for obvious reasons. But, it should also be used on the `WHERE` and `ON` clauses (if present), to ensure that they don't contain aggregates. (It would be an error for either of these clauses to contain aggregates.)

Finally, note that an aggregate function call also has an expression as its argument. It would be an error for an aggregate's argument to also contain an aggregate function call (e.g. `MAX(AVG(x))` is an invalid expression).

You should throw an `IllegalArgumentException` if you encounter any `WHERE` clauses containing aggregates, `ON` clauses containing aggregates, or aggregate function calls that contain other aggregate function calls in their arguments. Make sure the error message is suitably descriptive for the situation you encounter.

Nested-Loop Join

The nested-loop join algorithm is very easy to state. For two tables r and s, the nested-loop join algorithm does the following:

```
for t_r in r:
   for t_s in s:
      if pred(t_r, t_s):
         add join(t_r, t_s) to result
```

(Table r is frequently called the *outer* relation and s is called the *inner* relation, for obvious reasons. In NanoDB, as in many other database implementations, the left child is the outer relation and the right child is the inner relation.)

What makes this algorithm complicated to implement is using it in a demand-driven pipeline where plan-nodes must return the next tuple anytime a call to `getTuple()` is made. Because of this, the plan-node must store internal state that tracks where it is in its internal processing, so that it can pick up where it left off when `getTuple()` is called again.

A partial implementation of this join is provided in the `NestedLoopJoin` class (in the `plannodes` package). The implementation currently has the innermost if-statement and associated join-tuple operation, but the loop itself is not implemented. The class has three fields that are used for keeping track of the execution state:

- `done` is a flag that indicates when the plan-node has completely consumed its inputs. The flag is initialized to `false` in the `initialize()` method. Once `done` is `true`, the `getTuple()` method will return `null`.
- `leftTuple` is the most recent tuple retrieved from the left child-plan
- `rightTuple` is the most recent tuple retrieved from the right child-plan

(Note that there are probably cleaner ways to implement this. If you come up with something nicer then feel free to incorporate your refinements into your solution. We may even use them in a future class.)

The `getTuplesToJoin()` method updates `leftTuple` and/or `rightTuple` based on the next pair of tuples that should be considered. **This is where you must implement the nested-loop logic that allows the join to compare every possible pair of tuples.** You will need to iterate over the rows in the left and right child nodes, advancing the state of the nested loops every time `getTuplesToJoin()` is called. When you reach the end of the inner relation (i.e. when `rightChild.getTuple()` returns `null`), you can use `initialize()` to reset it back to the start.

Remember that you must also support left-outer joins with this plan node. You will need to modify the logic, and you may need to add fields to the class to properly track the execution state.

Once you have completed this node, you can use it in your planner to support the various join operations specified in the previous section.

Automated Testing

Once you have completed these tasks, it would be of value to know that they are actually correct. To this end, we can employ an automated testing framework to run tests against the database code to verify that everything works correctly. There are several different kinds of tests:

- **Unit tests** exercise the smallest testable unit of code. In most languages, this is a function or method, because it's not possible to exercise just a part of a function. Most of the tests in the

NanoDB code-base are unit tests, exercising various components of the software in isolation from each other.
- **Integration tests** exercise larger assemblies of the system together as a unit. For example, issuing a SQL query against the database and then checking the results would be an integration test, since it exercises many components of the system operating together.

In the `test/edu/caltech/test/nanodb/sql` directory is a simple framework for issuing queries against the database and comparing the results against a set of expected results. This allows us to do very basic integration testing against the NanoDB query processor. There are also a number of classes that exercise various aspects of the SQL queries you will support this week. You will note that these classes' constructors specify properties defined in the `test_sql.props` file; each property specifies the SQL initialization code that is necessary to run the corresponding test cases. (You can also issue your own initialization commands if you wish; quite a few tests do this instead.)

Create a new class `TestSimpleJoins` in this directory (you can copy one of the other classes to start with), which exercises your inner- and outer-join support and checks the results. Your testing should be reasonably complete; for example, it would be useful to verify the following:

- For all these examples, you would want to test them with inner joins, left-outer joins, and right-outer joins.
- Joins where the left table is empty and the right table is not.
- Joins where the right table is empty and the left table is not.
- Joins where both tables are empty.
- Joins where a given row in one table joins with several rows in the other table, and vice versa.

It shouldn't be too hard to construct a few small, simple input tables for these kinds of cases, and then construct the corresponding test code. (It will be tedious, but that's the price you pay for confidence that your code works.)

Don't be sloppy. Update the comments in the testing code to match what your code is doing. Also, make sure your tests pass. After you run the `ant test` target, the results will show up in the `build/testresults/nanodb-tests/index.html` file of your project. If there are test failures, you can browse to the details and then fix them. All tests should pass at the end of HW2.

Extra Credit

If you have gotten to this point and you now find yourself heartbroken that the assignment is over, you should add support for the `LIMIT` and `OFFSET` clauses by implementing a `LimitOffsetNode` and then incorporating this into your execution plan when the SQL specifies these clauses. For full credit, write unit tests exercising this functionality.

Submitting Your Work

As with the first assignment, you will tag your completed work with the `hw2` tag, and then push all of your changes to your team repository. The description of this process is duplicated from the previous assignment.

The first thing to do is tag your submission so that we can retrieve the correct version of your work:

```
git tag hw2
```

This will tag the current version of your repository with the name "`hw2`".

Once you have done this, you can push all of your changes to your CMS cluster repository:

```
git push --tags
```

The `--tags` argument will cause all tags in your local repository to be replicated into your submission repository. Once this is done, we will be able to retrieve the version of your work marked with this tag, so that we can grade it.

Note that if you decide to resubmit your work (e.g. because you found a bug that you needed to fix), you cannot reuse the same tag name again! Therefore, use a modified tag name that includes "`hw2`" in it, e.g. "`hw2-2`", so that we can still find your submission. Also, don't forget to push the new tag to your submission repository.

Finally, one team member should submit the completed Assignment 2 design document on the CS122 course Moodle.