Project 02: Query Plans and Evaluation

Project Overview

In this project, you will implement a simple query planner that translates a wide range of parsed SQL expressions into query plans that can be executed, complete the implementation of a nested-loop join plan node, and create automated tests for your join support.

Before Starting

NanoDB assignments frequently rely on the correctness of earlier assignments. Your Project 01 should pass all its tests – if you were not able to finish (including with an extension), talk to Prof. Ordentlich ASAP. Additionally, we've distributed some updates and additional files to your repository. If you don't have the file `doc/02-joins-design.md`, make sure that your local changes are committed, then `git pull --rebase` (or merge).

Plan Nodes

As discussed in lecture, NanoDB's query evaluator follows the iterator model and uses query plans made up of plan nodes, where each node requests tuples from its children. You can refer to **`PlanNode`** and other classes in the `plannodes` package for documentation on specific methods we mention here.

A query plan is a binary tree of `PlanNode`s, with `leftChild` and `rightChild`. Most data about a `PlanNode` is set via the constructor. As with relational algebra operations, every plan node specifies the schema of its output tuples. These values, along with other statistics, must be computed once an entire query plan is assembled; this is done by calling the `prepare()` method on the root plan node. This operation will recurse down the plan-tree so that all nodes have a chance to prepare themselves. The query evaluator expects that all plans have already been prepared before it starts evaluating!

Query Planner

NanoDB is currently unable to execute anything except the simplest SQL statements, because it has no way of generating more complex query plans from a SQL statement. NanoDB supports different planners being plugged into the database via the **`Planner`** interface. The two most relevant methods in the interface are:

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

This constructs a query plan (consisting of a single `SelectNode`) from a query of the form "`SELECT * FROM t WHERE ...`". It's used by `UPDATE` and `DELETE` statements to support the optional `WHERE` clause.

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

This constructs a query plan (consisting of a single `SelectNode`) from a standard parsed SQL query. Each `SELECT` statement is parsed into a **`SelectClause`** as the root of the AST, which contains all components of the statement. This includes the list of values in the `SELECT` clause, the tree of join expressions in the `FROM` clause, and the predicate in the `WHERE` clause. You can find the `Clause` classes in the `queryast` package.

`enclosingSelects` is the list of enclosing queries if this is a nested subquery. **In this project, it will be empty or `null`, since we're not handling subqueries.**

You can see example implementations of these functions in **`queryeval.SimplestPlanner`**. In particular, the `makePlan` function performs some basic error-checking, then dispatches to `makeSimpleSelect`, which makes sure to call `selectNode.prepare()` before returning.

Setting the Planner

You can configure NanoDB to use a specific planner as the default with the nanodb.plannerClass property. You can also set it by updating the DEFAULT_PLANNER_CLASS constant in the **server.properties.ServerProperties** class.

Adding Planner Features

Right now, SimplestPlanner is only able to pick tuples from a single table, and nothing else. Your task is to add support for many more SQL features in **queryeval.SimplePlanner**. This query planner is **not** focused on being optimal, so we can build a plan by following the conceptual SQL evaluation order discussed in class. That is, your makePlan will eventually follow this high-level approach:

```
PlanNode plan = null;

if (FROM-clause is present)
    plan = generate_FROM_clause_plan();

if (WHERE-clause is present)
    plan = add_WHERE_clause_to(plan);

if (GROUP-BY-clause and/or HAVING-clause is present)
    plan = handle_grouping_and_aggregation(plan);

if (ORDER-BY-clause is present)
    plan = add_ORDER_BY_clause_to(plan);

// There's always a SELECT clause of some sort!
plan = add_SELECT_clause_to(plan);

plan.prepare();
```

Some clauses are more complex to translate to a query plan than others, so we'll build this up piece by piece.

Task 1: Basis

Update SimplePlanner#makePlan to support the following SQL features:

- FROM clauses with one table
- FROM clauses that are null (e.g. SELECT 3 + 2 AS five;)
- WHERE clauses
- ORDER BY clauses
- SELECT clause

Some notes and guidelines:

- Throughout the project, you should generate the minimal plan necessary. For example, don't generate a ProjectNode if the SELECT clause specifies *.
- Your implementation does not have to support aliases within the ORDER BY clause. This allows us to put ORDER BY before SELECT.
- See the PlanUtils#addPredicateToPlan method, and the SortNode class.

To make NanoDB use SimplePlanner, make sure to update the DEFAULT_PLANNER_CLASS in ServerProperties.

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. For example, for a table `CREATE TABLE t (a INTEGER, b VARCHAR(20) );` we can use `EXPLAIN SELECT b FROM t WHERE a < 5;`

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

Use `make run` to start the NanoDB CLI. Then run the above two commands. If your Task 1 implementation is correct it should match the above output. The "cost is unknown" is because we haven't yet implemented plan costing or statistics collection – next project!

Join Trees

Task 2: Join Trees

Update `SimplePlanner#makePlan` to support the following SQL features:

- `FROM` clauses involving two or more tables being joined together
- Subqueries in the `FROM` clause

Some notes and guidelines:

- Refer to the documentation for the `FromClause` class.
- To implement this, you should start by moving your one-table `FROM` clause handler to a helper function, then recursively looking at the children of the `FROM` clause. Add `NestedLoopJoinNode`s to your plan where appropriate.
- Make sure to check `isRenamed` and apply a `RenameNode` when necessary.
- Handle (two-table) `NATURAL JOINs` and `USING` by projecting out duplicate column names – see `FromClause#getComputedSelectValues`.

You'll implement the internals of the `NestedLoopJoinNode` in a later task, though you're welcome to jump forward to it before finishing the planner.

As some minor checks for Task 2, first generate the following tables in the NanoDB CLI.

```
CMD> CREATE TABLE t1 (a INTEGER, b INTEGER);
CMD> CREATE TABLE t2 (a INTEGER, c INTEGER);
```

Next, run the following queries. You should see similar outputs.

```
CMD> EXPLAIN SELECT * FROM t1 AS x JOIN t2 AS y ON x.a = y.a;
Explain Plan:
    NestedLoop[joinType: INNER, pred:  x.a == y.a] cost is unknown
        Rename[resultTableName=x] cost is unknown
            FileScan[table:  t1] cost is unknown
        Rename[resultTableName=y] cost is unknown
            FileScan[table:  t2] cost is unknown

CMD> EXPLAIN SELECT * FROM t1 JOIN t2 USING (a);
Explain Plan:
    Project[values:  [t1.a AS a, t1.b, t2.c]] cost is unknown
        NestedLoop[joinType: INNER, pred:  t1.a == t2.a] cost is unknown
```

```
            FileScan[table:  t1] cost is unknown
            FileScan[table:  t2] cost is unknown

CMD> EXPLAIN SELECT * FROM (SELECT * FROM t1) AS s JOIN t2 ON s.a = t2.a;
Explain Plan:
    NestedLoop[joinType: INNER, pred:  s.a == t2.a] cost is unknown
        Rename[resultTableName=s] cost is unknown
            FileScan[table:  t1] cost is unknown
        FileScan[table:  t2] cost is unknown
```

Grouping and Aggregation

As discussed in class, grouping and aggregation is challenging to translate from SQL to relational algebra; because the `SELECT` clause mixes grouping/aggregation and projection, and the `HAVING` clause mixes selection and aggregation. These expressions must be scanned for aggregates so that the grouping plan node can be initialized correctly, and so that we remove aggregates and replace them with placeholder variables.

Scanning and/or Transforming Expressions

Since various parts of NanoDB want to traverse expressions, we implement the `visitor pattern` to handle traversing and transforming expression trees. In the `expressions` package, the **`Expression`** base class has a `traverse()` method that traverses the entire hierarchy, and takes an **`ExpressionProcessor`** implementation.

The processor has an `enter(Expression)` method that is called as each expression node is visited. Then the node's children are recursively traversed; then the `leave(Expression)` method is called, again with the node as an argument. (That is, `enter()` is called in preorder and `leave()` is called in postorder.) Note that `leave()` returns an `Expression`, which allows the processor to mutate the tree by returning a different node than it was called with.

An example `ExpressionProcessor` implementation is the `SymbolFinder` class nested inside `Expression`, used by `Expression#getAllSymbols()` (though it doesn't modify the tree).

Task 3: `AggregateFunctionExtractor`

Complete the implementation of the `AggregateFunctionExtractor`, which will identify aggregate functions in expressions, map each one to an auto-generated name, and update the expression to use the new name.

Some notes:

- See the `aggregates` argument in the `HashedGroupAggregateNode` constructor.
- You should feel free to modify the class as necessary, including adding instance variables or helper methods.
- If an aggregate appears multiple times, don't create multiple names or store multiple `FunctionCalls`.
- Nested aggregates (e.g. `MAX(AVG(x))`) are errors; throw `InvalidSQLException` if encountered.

Identifying and Replacing Aggregate Functions

Since the `traverse()` function is recursive, the result needs to be used after the top-level call. As an example, when you're iterating over `SelectClause` expressions for aggregates, you'll want to write code 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);
}
```

Function calls are a kind of expression, represented by the **`FunctionCall`** class in the `expressions` package. You can use `FunctionCall#getFunction()` to retrieve the specific function, and see if it's an aggregate function, like so:

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

If you want to replace an aggregate function call with a column reference, use the `ColumnValue` expression class.

Task 4: Using the `AggregateFunctionExtractor`

Back in `SimplePlanner#makePlan()`, use your `AggregateFunctionExtractor` to rewrite the `SELECT` values and the `HAVING` clause. Then, use a `HashedGroupAggregateNode` in your plan to account for a `GROUP BY` node (and add a predicate for the `HAVING` clause if necessary).

Some notes:

- Not all expressions can contain aggregates. If the `WHERE`, `ON`, `GROUP BY`, or `FROM` clause contains an aggregate, throw an `InvalidSQLException`. You may want to modify your `AggregateFunctionExtractor` slightly to support this.

At this point, `make hw2` tests should pass.

Plan Node Lifecycle

Now that we're about to implement a `PlanNode`, we go into more details about their internals.

Before a plan node can produce any rows, its `initialize()` method *must* be invoked to set up evaluation-time resources, so that the next invocation of `getNextTuple()` will return the first tuple. Typically, this method invokes `super.initialize()` to initialize its parent class, then it initializes its own internal members, and finally invokes `initialize()` on its left and right children.

The `getNextTuple()` method is where the plan node implements its logic. On each call, it returns single tuple generated by the plan node (or `null` if the node is exhausted). Many plan nodes must call their children's `getNextTuple()` multiple times before being able to return their own next tuple. See `SelectNode` and `SimpleFilterNode` for an example of this, where retrieved tuples must be tested against the predicate. Some plan nodes must consume *all* data from their children before producing any results (e.g. sorting, hash-based grouping).

Some plan nodes may need to iterate over the tuples from their children multiple times, including the nested loop join node. Subplans can be restarted at any point by calling `initialize()`. For some plan nodes, like a file scan, resetting to the beginning is just seeking to the start of the tuple file. However, other plan nodes may not have their results on disk, and will need to recompute their results from scratch. If a node implements any external memory algorithms, `initialize()` will need to handle these properly.

Finally, when the evaluator is done executing, it calls `cleanUp()` on the root plan node, which recurses in a similar way. Most plan nodes require no clean up, but external memory algorithms do.

Nested-Loop Join

The nested-loop join algorithm is as follows: given two tables r and s, to compute $\left(r \bowtie_{\text{pred}} s\right)$, do the following:

```
for tr in r:
  for ts in s:
    if pred(tr, ts):
      add join(tr, ts) to result
```

Table r is the "outer relation", and s the "inner relation". In NanoDB, and many other database implementations, the left child is the outer relation, and the right child is the inner relation.

What makes this algorithm nontrivial to implement is the streaming (demand-driven, pull-based, or iterator) model discussed in class, where plan nodes must return the next tuple anytime a call to `getNextTuple()` is made. Because of this, the plan node must store some internal state between calls, like Java `Iterators` and `next()`.

A partial implementation of this is provided in the **`NestedLoopJoinNode`** class, which currently has the innermost if-statement and call to `joinTuple`, but the loop itself (`getTuplesToJoin`) is not implemented. The class has three fields that are used for tracking the execution state:

- `done`, to indicate when the plan node has completely consumed its inputs.
- `leftTuple` and `rightTuple`, the most recent tuple retrieved from the left or right child plan nodes, respectively.

Task 5: `NestedLoopJoinNode`

Complete the implementation of the `NestedLoopJoinNode` class to support inner joins, as well as left-outer and right-outer joins. The bulk of your work will be in the `getTuplesToJoin`, where you iterate over the rows in the left and right child nodes to update the next `leftTuple` and `rightTuple` that should be considered.

Some notes:

- When your inner relation is exhausted, you can use `initialize()` to reset it back to the start.
- Make sure that you call `unpin` when you're done with a tuple.
- To implement left-outer joins, you will need to add field(s) to the class.
- The superclass `ThetaJoinNode` provides the `swap` function, which will help with implementing right-outer joins without much additional code.
- Ignore the `TODO` comment in the `resetToLastMark` function. You do not need to complete any implementation for that.

You may notice there's no tests for (simple) `JOIN` behavior. Your final task is to verify correctness of your own join node implementation with a set of **integration tests**. In the `src/test/.../sql` directory, you'll find a simple framework for issuing queries against the NanoDB query processor, as well as test classes for some of the features you've just implemented. The constructors of these classes reference properties in the `resources/.../sql/test_sql.props` file, where each property specifies the SQL initialization code necessary to run the corresponding test cases.

Task 6: Testing Joins

In the `TestSimpleJoins` class, write tests to exercise your join support. Some suggestions for testing:

- (Inner, left-outer, right-outer) joins with an empty left table and a non-empty right table, and vice-versa
- (...) joins with an empty left table and an empty right table
- (...) joins where a given row in the left table matches with several rows in the right table, and vice-versa

Document what your tests are doing in some way (comments, function names, etc.). Your tests should pass.

`NATURAL JOIN` for Multiple Tables (Extra Credit)

NanoDB currently doesn't have (full) support for the `NATURAL JOIN` or `USING` clauses – in particular, it doesn't support joining multiple tables. Consider the following schemas and query:

```
t1 (int a, int b)
t2 (int a, int c)
t3 (int a, int d)

SELECT * FROM t1 NATURAL JOIN t2 NATURAL JOIN t3;
```

The resulting schema should be `(a, t1.b, t2.c, t3.d)`. The first join of `t1` and `t2` works fine, producing the schema `(a, t1.b, t2.c)`, where duplicate a's have been projected out. However, the second join (with `t3`) has an issue with column ambiguity: since the intermediate table is unnamed, the unqualified column reference a in the join condition `a = t3.a` (and in the projection) is ambiguous!

Like we introduced aliases for expressions, we can fix this by introducing placeholder names for intermediate tables, making the schema for the intermediate join `#R1(a, t1.b, t2.c)`. This allows the second join to be written as `#R1.a = t3.a` (as well as the project). Unfortunately, doing this is more complex than just adding a `RenameNode`, because of how the schema is computed in NanoDB.

Extra Credit: Multi-Table `NATURAL JOIN` and `USING`

Add support for using `NATURAL JOIN` and `USING` with multiple tables in NanoDB. You may **not** handle this by ignoring ambiguity in an unqualified column reference (when the table name is `null`), since this will ignore real ambiguous queries. You will almost certainly need to understand how the `FromClause` builds the join schema that the join node uses. Good luck. If you manage this, we'll be excited to talk about how.

There's some existing tests in the `TestNaturalUsingJoins`. At least these should pass; but you should almost certainly add more tests to exercise different join trees. You can add these to your test suite by removing the `!TestNaturalUsingJoins` flag from the `HW2_TEST_EXCLUDES` variable in the Makefile.

Design Document

Design Doc

Answer the questions in the file `doc/02-joins-design.md` in the document itself.

Submitting

Once you are finished, make sure your code is pushed to GitLab. Then, submit the commit hash you would like to be reviewed on Gradescope.

For more details, see the Project Submission instructions on the course website.