Assignment 4: Join Optimization

In this assignment you will write a planner/optimizer that chooses an optimal join ordering using a dynamic programming algorithm. Your optimizer must generate valid plans for queries involving outer joins as well as inner joins, which increases the complexity of the planner somewhat.

Your planner should be able to handle the same queries that your Assignment 2 planner was required to handle, including queries with ORDER BY clauses, and grouping and aggregation. Of course, you can reuse your Assignment 2 code to achieve this goal. (Recall that the suggested design uses an abstract base-class so that multiple planners can share common functionality.)

Overview

Join ordering can have a profound impact on query execution performance, so most databases devote specific effort to finding an optimal join order. Most database systems use a Selinger-style join optimizer that uses dynamic programming to choose a join order, but that also keeps costlier plans that generate results in "interesting orders." For this assignment you won't worry about result ordering (we don't have any plan nodes that can take advantage of result-ordering anyway); you simply need to implement a join planner that uses dynamic programming to find the optimal order.

Although we will identify an optimal join order based on plan costs, we will also employ a heuristic to simplify planning: we will always perform selection as early as possible in plans. We can do this because NanoDB doesn't have any indexes yet, and all tables are heap files, so this will produce optimal plans. However, in general it is a bad approach since it may rule out other more optimal plans, particularly when there are indexes or other optimizations that affect when selection should be applied.

As before, plan optimization is performed on units of SELECT-FROM-WHERE blocks; if a FROM clause includes a subquery then the subquery will be optimized separately, and then the subquery's generated plan is treated as a black box.

Optimization Procedure

The general approach for an optimizer based on dynamic programming is as follows:

- Identify all "leaves" in the FROM-expression of the query. This would include base-tables and subqueries, for example.
- Create an optimal plan for each leaf identified above. Store each optimal leaf plan, along with its cost.
- For every pair of leaves, create an optimal plan that joins the pair of leaves together. Store each of these plans, along with their costs. When accessing each leaf, use the plans cached in the previous step.
- Continue this process for three leaves, etc., using the optimal plans generated in the previous steps, until all leaves are joined together into a single plan. As we go, if a more optimal plan is found to join a particular set of leaves, it will replace any previous plan that joins that set of leaves.

Although this process is conceptually straightforward, it can be somewhat involved to handle all of the details. Specifically, while join-expressions may specify the conditions to join on, we may also have conditions in the WHERE clause that we can take advantage of lower in the tree as well. Consider these three queries:

```
SELECT * FROM t1, t2 WHERE t1.a = t2.a AND t2.b > 5;
SELECT * FROM t1 JOIN t2 ON t1.a = t2.a WHERE t2.b > 5;
SELECT * FROM t1 JOIN t2 ON t1.a = t2.a AND t2.b > 5;
```

All three of these queries produce the exact same results, but the conditions appear in different places on the SQL abstract syntax tree:

- The first query performs a Cartesian product of t1 and t2, and has a WHERE-expression specifying both the join condition and an additional condition.
- The second query performs a theta-join with a condition of t1.a = t2.a, and also has a WHERE-expression specifying the additional condition.
- The third query performs a theta-join with a composite condition of t1.a = t2.a AND t2.b > 5. It has no WHERE-expression in the SQL AST.

A good optimizer should be able to take any of these queries and produce the exact same execution plan.

Approach

Generally, planners manipulate components of predicates called *conjuncts*. A conjunct is simply a condition that is ANDed together with other conjuncts to create the overall predicate. The above examples specify two conjuncts: t1.a = t2.a, and t2.b > 5. Once these conjuncts are collected out of the query, we can determine the optimal place to apply each conjunct in the final execution plan.

For a query SELECT *SelectVals*… FROM *JoinExpr* WHERE *Pw*, the SQL standard specifies that it should be translated into a relational algebra expression like this:

$$\Pi_{SelectVals}(\ \sigma_{\mathrm{Pw}}(\ JoinExpr\ )\ )$$

The join expression *JoinExpr* will include join conditions if a NATURAL join or a USING/ON clause is specified in the query. Additionally, there may be conjuncts in the WHERE-predicate *Pw* that can be used to constrain joins, or even constrain the inputs to a join. In other words, we may be able to take some or all of the conjuncts in *Pw*, and push them down into the plan we create for the *JoinExpr* to make it more efficient.

Therefore, we need to collect all of these conjuncts together in order to determine the optimal placement of each conjunct in the plan. That way we can constrain each node to generate as few records as possible, and if we have indexes or equijoin conditions, we can take advantage of them.

Equivalence Rules

When it comes to inner joins, we are quite free to rearrange the join order and place conditions wherever we want in the plan, because we have many equivalence rules for manipulating inner joins. Here is a small sampling:

- $E1 \bowtie E2 \equiv E2 \bowtie E1$
- $E1 \bowtie (E2 \bowtie E3) \equiv (E1 \bowtie E2) \bowtie E3$
- $\sigma_{\theta 1}(E1 \bowtie E2) \equiv \sigma_{\theta 1}(E1) \bowtie E2$ ($\theta 1$ only refers to attributes in E1)

As long as we are careful to only place predicates where they make sense (i.e. where the referenced columns are actually available), we won't get into any trouble.

This suggests that our planner will need to traverse the contents of the join-expression *JoinExpr*, identifying all base-tables and SELECT subqueries to be manipulated – these are the leaves of the join expression, for which we need to determine an optimal join ordering. Additionally, we will also need to collect all conjuncts contained in join-conditions in *JoinExpr*, so that we can apply each conjunct as early as possible in the plan.

Outer joins complicate this process, because there are many equivalence rules that hold for inner joins that do not hold on outer joins. For example, these general cases apply:

- E1 ⟕ E2 ≡ $\Pi_{E1,E2}$(E2 ⟖ E1) (requires a project to properly order the schemas)
- E1 ⟗ E2 ≡ $\Pi_{E1,E2}$(E2 ⟗ E1) (requires a project to properly order the schemas)
- $\sigma_{\theta1}$(E1 ⟕ E2) ≡ $\sigma_{\theta1}$(E1) ⟕ E2 (θ1 only refers to attributes in E1)
- $\sigma_{\theta2}$(E1 ⟕ E2) is <u>not</u> equivalent to E1 ⟕ $\sigma_{\theta1}$(E2) (θ2 only refers to attributes in E2)
- $\sigma_{\theta1}$(E1 ⟗ E2) is <u>not</u> equivalent to $\sigma_{\theta1}$(E1) ⟗ E2 (θ1 only refers to attributes in E1)
- (E1 ⟕ E2) ⟕ E3 is <u>not</u> equivalent to E1 ⟕ (E2 ⟕ E3)

Looking at second through fourth points, it is relatively easy to see that we can only push a conjunct down through an outer-join if the "outer" side is <u>not</u> opposite the child that the conjunct applies to. For example, in the first point above, we can push θ1 down through the left-outer join, since the right side of the join is not an "outer" side. (It should be evident that full-outer joins completely disallow pushing conjuncts down through them.)

Since outer joins generally resist reorganizing, our planner will basically leave them alone. This involves two basic strategies:

- Treat outer joins as "leaves" when scanning the join expression *JoinExpr* for the set of input-relations to determine an optimal order for.
- Be careful to only push conjuncts down through an outer join if the resulting plan will still be equivalent to the original query.

Thankfully, this doesn't complicate things too much.

Implementation

Your implementation of the dynamic-programming join planner will go into the `CostBasedJoinPlanner` class in the `queryeval` package. (Feel free to copy over code from your `SimplePlanner` to fill in missing pieces.) Note that several helper functions are provided to simplify your efforts; if you find yourself needing a particular operation, it may already exist.

You will notice that the `CostBasedJoinPlanner` class contains a nested class called `JoinComponent`. This class holds all details for a particular join-plan, including the set of conjuncts used within the plan, and the set of leaf-plans that are joined together by the join-plan. This allows us to easily build up the overall plan from the various components.

As before, the main entry-points into the planner are:

- `makePlan()` – generates a plan for a SELECT-FROM-WHERE expression
- `makeSimpleSelect()` – generates a simple "SELECT * FROM *table* WHERE *predicate*" plan

You will note that the `CostBasedJoinPlanner` includes several other operations; these will be discussed in subsequent sections. There is a third entry-point you will need to be aware of:

- `makeJoinPlan(FromClause)` – generates an optimal join-plan from a specified `FromClause`. This method takes a second argument, a collection of conjuncts from above the join-plan. The optimizer may be able to apply these conjuncts within the join-plan to produce a better plan.

The `makeJoinPlan()` method is private because it is not for external use, but there will definitely be situations where you need to recursively invoke it to handle certain situations. This method is implemented for you; it simply coordinates the phases of the planning outlined earlier.

Collecting Details from the `FromClause`

The initial detail-collection is performed by the `collectDetails(FromClause, ...)` method in the join-planner. **You must implement this method, and also write a Javadoc header for it.**

The detail-collection phase must collect both the set of conjuncts and the list of leaf `FromClauses` from the specified `FromClause` argument. Here are a few notes to help in your implementation:

- This method accumulates its results into a couple of collections specified as parameters. This should make it easy for `collectDetails()` to recursively invoke itself on child clauses.
- As mentioned before, this method should consider base-tables, subqueries, and outer-joins to be leaves. The `FromClause` class also has a method `isOuterJoin()` which should help with this implementation.
- Only collect conjuncts from the predicates that appear on non-leaf `FromClauses` you encounter.
- A helper method `PredicateUtils.addConjuncts()` has been provided (in the `expressions` package), which can properly handle the various cases you might encounter in a predicate. If a WHERE predicate or a join condition is a Boolean AND-expression (such as our previous example of t1.a = t2.a AND t2.b > 5), then we want to break this expression apart and store each individual term as a conjunct. Any other kind of predicate is simply stored as a single conjunct. (We could be more sophisticated and perform logical analysis on our predicates, but that would be beyond the scope of the assignment. Feel free to increase the sophistication of this method on your own though, if you prefer.)

Once all leaves are identified and all conjuncts are collected, we can begin generating an optimal plan to join the leaves.

Generating Optimal Leaf-Plans

The first step in creating an optimal query plan is to devise an optimal plan for each of the leaf expressions in the `FROM` component of the query. The method responsible for generating all leaf plans is the `generateLeafJoinComponents()` method; it uses the helper `makeLeafPlan()` to generate the actual plan for each leaf. **You must implement the `makeLeafPlan()` method.**

Coming up with a basic plan for each leaf is very straightforward; we already know how to do this:

- If the leaf is a base table, use a `FileScanNode`.
- If the leaf is a derived table then recursively call the planner to get a query plan for the subquery.
- If the leaf is an outer join, generate an optimal plan for each child from-clause by recursively invoking the `makeJoinPlan()` method, then create a plan-node to perform the outer join on these two children.

  Your code for this case should be careful to pass conjuncts down to the recursive invocation of `makeJoinPlan()`, but only when the resulting plan will still be equivalent to the original query. You can make use of the two methods on `FromClause`, `hasOuterJoinOnLeft()` and `hasOuterJoinOnRight()`, to determine when to pass conjuncts to the recursive invocation.

Now, this may be a basic plan, but an "optimal" plan will also apply selections as early as possible. (Recall that this is a heuristic, and may not generate an optimal plan in the presence of indexes.) To do this, we need to see if any conjuncts can be applied to each leaf plan-node we generate. Since this can become a tedious operation, and Java is not an ideal language for this kind of thing, several helper methods are provided for you to use:

- The `PredicateUtils.findExprsUsingSchemas()` method is a very powerful helper method. It takes a collection of expressions and one or more schemas, and it pulls out all expressions that can be evaluated against the specified schemas. For example, given the schema from table t2 and the set of conjuncts (t1.a = t2.a, t2.b > 5), this method would pull out t2.b > 5 since that can be evaluated against t2's schema. The first conjunct would be excluded because it also needs t1 as well as t2.

  Note that the caller must provide a collection that the results are stored into; a newly created `HashSet<Expression>` object would be best for this. Also, note that this helper method can *optionally* remove matching expressions from the input collection; when generating leaf plans, you don't want to remove matching expressions from the input collection.

- To use the previous method, you need the schema of the leaf-plan. You can cause the leaf plan to compute its schema and cost by calling `prepare()` on the plan. Before you call `prepare()`, the plan's `getSchema()` method will return `null`; once you have prepared the plan then you will be able to retrieve the plan's schema and cost details.

  Try to call `prepare()` as little as possible, since it traverses an entire plan recursively!

- Once you have created a leaf-plan and you have the set of conjuncts that can be applied to that leaf plan, you will want to create a new predicate and apply it to that leaf-plan. (This is following the "apply selections as early as possible" heuristic.) Two methods are provided to make this very easy:

  - The `PredicateUtils.makePredicate()` helper takes a collection of zero or more conjuncts, and builds a predicate that can be applied to a plan. If the collection contains no conjuncts then the return-value is `null`. If the collection contains a single conjunct then this is returned as the predicate. Otherwise, the helper builds up a Boolean AND-expression combining all conjuncts.

  - The `PlanUtils.addPredicateToPlan()` helper takes an existing plan-node and a predicate, and it modifies the plan to include the selection predicate. If the passed-in plan-node is a select operation then the predicate will be incorporated into that plan-node's predicate. If the passed-in plan-node is some other kind of operation then a new `SimpleFilterNode` will be added above the passed-in node, with the specified predicate.

- **When you change a plan, you must call `prepare()` on it again!** This includes adding a predicate to a plan; the plan's cost and statistics must reflect the predicate you have added.

Generating an Optimal Join Plan: Approach

The final step of join optimization is to combine the *N* leaf plans into one optimal join plan. We will use dynamic programming for this step, allowing us to reuse earlier computations as we search for the answer. The algorithm will operate in a loop, starting with a collection of join-plans that combine *n* leaf-plans in an optimal way, and producing a collection of join-plans that combine *n*+1 leaf-plans in an optimal way. Initially we start with the set of leaf plans (each of which "combines" one leaf in an optimal way); the first iteration will produce plans that join 2 leaves together; the next iteration will produce plans that join 3 leaves together; and so forth.

In the final iteration of this algorithm, we should have a set of plans that combine *N*-1 leaf-plans together, and we should produce a plan that combines all *N* leaf-plans together. Since we only keep the optimal plan for each distinct set of *n* leaves being joined, we should end up with only one join-plan that optimally combines all *N* leaves.

For a specific iteration of this algorithm, we will start with a set of plans $JoinPlans_n$ that join *n* leaf-plans together, and the set of leaf-plans *LeafPlans*. To generate plans that join *n*+1 leaves together, we will simply iterate over both of these sets:

```
for $plan_n$ in $JoinPlans_n$:              # Iterate over plans that join n leaves
    for leaf in LeafPlans:                  # Iterate over the leaf plans
        if leaf already appears in $plan_n$:
            continue                        # This leaf is already joined in by the current plan

        $plan_{n+1}$ = a new join-plan that joins together $plan_n$ and leaf
        newCost = cost of new plan
        if $JoinPlans_{n+1}$ already contains a plan with all leaves in $plan_{n+1}$:
            if newCost is cheaper than cost of current “best plan” in $JoinPlans_{n+1}$:
                # $plan_{n+1}$ is the new “best plan” that combines this set of leaf-plans!
                replace current plan with new plan in $JoinPlans_{n+1}$
        else:
            # $plan_{n+1}$ is the first plan that combines this set of leaf-plans
            add $plan_{n+1}$ to $JoinPlans_{n+1}$
```

Reviewing the above algorithm, it is again clear that we should store additional details with each join-plan we generate:

- The join-plan itself (obvious). Plans know their own estimated costs, so we can easily find a plan’s cost from the plan itself.
- The set of leaf plan-nodes combined by the join-plan. This is essential, because it is how we identify whether two different plans join together the same set of leaves.
- The set of conjuncts applied within the join-plan. This is again helpful to determine which conjuncts have not yet been applied in the plan, so that we can ensure they are not left out.

Generating $plan_{n+1}$ from $plan_n$ and *leaf*

This process is generally straightforward, but there are two important details about generating a new join-plan. First, we will constrain our join optimizer to only consider *left-deep* plans. In other words, leaf nodes should always appear on the *right* of a theta-join operation; a sub-plan involving other joins will always automatically end up on the left of a theta-join. (For this assignment we don’t care if a nested subquery ends up as the right sub-plan of a join. Hopefully the estimated cost will make it unappealing.) This makes plan-generation simpler, because we don’t need to generate two candidates from $plan_n$ and *leaf*; we simply join them in that order, and that is our new plan.

The second important detail is that we must again follow our heuristic of applying selection as early as possible, so if there are conjuncts we can apply on the theta-join in the new join plan, we want to go ahead and apply them there. This requires a bit of effort:

$$SubplanConjuncts = LeftChildConjuncts \cup RightChildConjuncts$$
$$UnusedConjuncts = AllConjuncts - SubplanConjuncts$$

This is why we want to store the conjuncts used by each plan we generate, so that we can easily determine what conjuncts remain to be applied when generating a new plan.

Of this set of *UnusedConjuncts*, we must determine which of those conjuncts should be applied to the theta-join. Once that is determined, we can easily compute the set of conjuncts used by the new plan, and we can store this with the new plan.

Storing the Optimal Plan

As is given in the pseudocode earlier, we must determine if a plan combining the specified leaf-nodes has already been stored. If so, we must compare the costs of the two plans, and if the new plans is cheaper than the current "best plan" then the new plan should replace the old plan. Or, if there is no plan for this combination of leaf nodes, we always store the new plan.

Generating an Optimal Join Plan: Implementation

In NanoDB, all `Expression` classes and all `PlanNode` classes implement the `hashCode()` and `equals()` methods, allowing them to be used with `HashSets` and `HashMaps`. For example, the set of leaves joined by a plan are collected within a `HashSet<PlanNode>` object, and it is very easy to perform set-membership tests with such a collection. Additionally, the hash-set itself can be used as a key into a mapping; specifically, we can map a set of leaf-plans to the optimal join-plan that combines those leaves.

Similarly, conjuncts can be manipulated with `HashSet<Expression>` objects. `HashSets` implement several methods for set operations: the `addAll()` method can be used to compute a set-union, the `removeAll()` method can be used to compute set-difference, and the `retainAll()` method can be used to compute the set-intersection. (Note that all of these methods change the object they are called on.) The `HashSet` constructor can also take another collection as an argument; the collection's contents are copied into the `HashSet`.

The optimal join plan is generated by the `generateOptimalJoin()` method; an outline of the solution is provided in this method to steer you in the right direction. There isn't much else to say about this method that hasn't already been said, except that the implementation relies on two collections that look like this:

```
HashMap<HashSet<PlanNode>, JoinComponent> joinPlans
```

This mapping is used to store the optimal plan for joining together *n* leaf-plans. The key for this mapping is the set of leaf-nodes joined together by each plan; the value is an object that holds the plan itself, the conjuncts applied by the plan, and the set of leaf-plans joined together by the plan.

Because all plan-nodes implement the `hashCode()` and `equals()` methods, two `HashSets` of plan-nodes will hash and compare equal if they contain the same plans. Therefore, if you generate a new plan that joins together the same leaves, you can use the set of leaves to look up any existing plan for those leaves, and compare an existing plan's cost to the new plan's cost.

Some other notes:

- Note that `HashMap` provides several useful views of the mapping. The `keySet()` method exposes the keys in the mapping; the `values()` method provides a collection of all values.
- Make sure you only join in each leaf-plan once. This should be easy to do.
- The `findExprsUsingSchemas()` helper will again be very useful to determine what conjuncts to apply. Find the set of conjuncts unused by either sub-plan, and see which of those conjuncts can be applied to the join. Note that you can call this helper method with multiple schemas, so you should be able to pass in both child-plans' schemas to find the set of unused conjuncts. (See the Javadocs on this method for an example.)

- Make sure that when you compute the set of unused conjuncts, you don't change the set of all conjuncts, or either child-plan's set of conjuncts! Be careful to avoid subtle side-effect bugs.
- As before, when you generate a new plan joining two plans together, you must call `prepare()` on the new plan to actually compute its schema and cost. If you have written your code correctly, you shouldn't have to call `prepare()` on either child-plan in this method, since this will have already been done earlier. Also, you should only have to call `prepare()` on the new plan once.

Testing Your Work

Once you have completed your join planner, you should start with very simple tests to make sure everything is working properly. However, you must first tell NanoDB use the new planner. All commands that require a planner use the `PlannerFactory` class to get one. (This class is in the `edu.caltech.nanodb.queryeval` package.)

The easy way to tell the `PlannerFactory` to use your planner is to change the `DEFAULT_PLANNER` constant from `SimplePlanner` to `CostBasedJoinPlanner`. (Leave the package name the same!) Once you have rebuilt your code-base, it should begin using your new planner.

(You could also edit the `nanodb` script or the `nanodb.bat` batch-file to specify the property `nanodb.planner.class`, which would be less intrusive, but setting `DEFAULT_PLANNER` and then rebuilding is the easiest way.)

Make sure you have `ANALYZE`d your tables before testing your planner. You should be able to handle very simple queries such as:

```
SELECT * FROM states;
SELECT state_id FROM states;
```

If these queries work, you might want to try something more challenging, such as this three-table join from the last assignment:

```
SELECT store_id, property_costs
FROM stores, cities, states
WHERE stores.city_id = cities.city_id AND
        cities.state_id = states.state_id AND
        state_name = 'Oregon' AND
        property_costs > 500000;
```

Regardless of what order you specify the tables or conditions, or how the conditions are specified (e.g. in ON clauses, or in the WHERE clause, etc.), you should get the exact same join plan. For example, I always get this plan:

```
Project[values:  [STORES.STORE_ID, STORES.PROPERTY_COSTS]] cost=[tuples=22.7, …, cpuCost=11866.2, blockIOs=26]
  NestedLoop[pred:  STORES.CITY_ID == CITIES.CITY_ID] cost=[…, cpuCost=11843.5, blockIOs=26]
    NestedLoop[pred:  CITIES.STATE_ID == STATES.STATE_ID] cost=[…, cpuCost=298.0, blockIOs=2]
      FileScan[table:  STATES, pred:  STATES.STATE_NAME == 'Oregon'] cost=[…, cpuCost=44.0, blockIOs=1]
      FileScan[table:  CITIES] cost=[tuples=254.0, tupSize=23.8, cpuCost=254.0, blockIOs=1]
    FileScan[table:  STORES, pred:  STORES.PROPERTY_COSTS > 500000] cost=[…, cpuCost=2000.0, blockIOs=4]
```

Submitting Your Assignment

When you are finished with the assignment, tag it with a `hw4` tag as usual, and then push all of your changes to your team's repository.