

Project 04: Join Optimization

Project Overview

In this project, you will build on your SimplePlanner from Project 2 and your statistics from Project 3 to implement a planner/optimizer that chooses an optimal join ordering using a dynamic programming algorithm. Your planner will handle optimizing both inner and outer joins.

Background

Join ordering has an outsized effect on query execution performance, so most databases devote specific effort to optimizing it. Most systems use a Selinger-style (System R) algorithm that uses dynamic programming to choose a join order, but that also keeps costlier plans that generate results in “interesting orders”. We make the following simplifications from the original algorithm:

- **We will ignore result ordering.** NanoDB doesn't have many plan nodes that can take advantage of result ordering anyway, so we will solely focus on minimizing cost with dynamic programming.
- **We will always perform selection as early as possible.** This approach is suboptimal in general. However, NanoDB currently doesn't have indexes and all tables are heap files, so pushing down selects is always better. However, indexes or other optimizations can affect when selection should be applied.

Plan optimization is performed on single SELECT-FROM-WHERE blocks; if a FROM clause includes a subquery, then the subquery will be optimized separately and treated as a black box with respect to planning and optimization.

Algorithm

At a high level, your optimizer will take the following approach:

- Identify all “leaves” in the FROM-expression of the query, such as base tables and subqueries.
- Create an optimal plan for accessing each leaf. Store these plans, along with their cost.
- For each pair of leaves, create an optimal plan that joins the pair of leaves together (using the plans generated in the previous step). Store each of these plans, along with their costs.
- Continue for three leaves, etc., using the plans generated in the previous step until all leaves are joined together into a single plan. If a more optimal plan is found to join a particular set of leaves, discard other plans that join that set.

In addition to the above statement, there are many subtle details to consider. Specifically, while JOIN expressions may specify their conditions, we may also have join 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 queries are equivalent, but the conditions appear in different places in the SQL AST: in the JOIN, in the WHERE, or across both. A good optimizer will be able to take any of these queries and produce the same execution plan.

Predicates

Query planners typically restrict themselves to manipulating only predicates of specific forms. This works well in practice, because most real queries follow these forms. (And if they don't, they usually can't be optimized effectively...)

We will restrict our focus to conjunctive selection predicates, which are a series of conditions ANDed together. Some databases know how to deal with *disjunctions*, which are a series of conditions ORed together, but that's out of the scope of this class.

The queries on the previous page have conjunctive selection predicates: $t1.a = t2.a$ AND $t2.b > 5$. Recall that a query "SELECT SelectVals FROM JoinExpr WHERE Pw" can be translated into a relational algebra expression: $\Pi_{\text{SelectVals}}(\sigma_{Pw}(\text{JoinExpr}))$. However, conjuncts may be better placed elsewhere, i.e. a conjunct from Pw that can be used to constrain a join, or an input to a join.

To determine the optimal placement of each conjunct, we need to first traverse the expression to collect all of the conjuncts together.

Equivalence Rules

As discussed in class, inner joins are fairly robust to manipulation (commutative, associative, selection pushdown, etc.) We do need to be careful to only place predicates where they make sense (i.e. where referenced columns are available).

This suggests that our planner will need to traverse the contents of the JoinExpr to identify all base tables and SELECT subqueries – these are the leaves of the join expression, which we will find the optimal order for later. We will also need to collect the conjuncts from the JoinExpr.

NATURAL JOINS and USING must also be treated as leaves since they project out duplicate columns. This behavior is tied to a specific join boundary, so our planner should also treat these joins as leaf units.

However, outer joins are trickier to manipulate. In particular, the only valid reorganization of conjuncts is pushing down to the "outer" side (i.e. $\sigma_{\theta}(r \bowtie s) = \sigma_{\theta}(r) \bowtie s$ when θ only refers to attributes in r , and respectively for \bowtie). Our planner will treat outer joins as follows:

- Treat outer joins as "leaves" when scanning the join expression
- Be careful to follow equivalence rules when pushing conjuncts down through an outer join

Implementation

Your implementation of a planner that uses query optimization will go in the `CostBasedJoinPlanner` file, which we've added to your repo.

Refactoring

Much of the functionality from `SimplePlanner` will need to apply again in `CostBasedJoinPlanner`. To avoid duplicating code (and to help make future planners easier to write), we'll move common functionality to the abstract class `AbstractPlannerImpl`. (We use an abstract base class because some of the shared functionality involves fields, which interfaces don't support.)

Syntax-wise, this will eventually look like this:

```
public abstract class AbstractPlannerImpl implements Planner {
    // Functionality common to all planners
}

public class SimplePlanner extends AbstractPlannerImpl {
    // Code specific to the SimplePlanner
}

public class CostBasedJoinPlanner extends AbstractPlannerImpl {
    // Code specific to the cost-based planner
}
```

Task 1: Refactoring Planners

Implement the `AbstractPlannerImpl` abstract class, and update your `SimplePlanner` to use it. We've already added `makeSimpleSelect()`, but you should also consider moving the code that handles grouping and aggregation.

Not all code that may be duplicated in this project's planner should be factored out to the base class; use your judgement. Once you finish refactoring your `SimplePlanner`, your tests from Project 2 should still pass.

Join Components

`CostBasedJoinPlanner` 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 joined together.

As before, the main entry points into the planner are `makeSimpleSelect()` (same as `SimplePlanner()`) and `makePlan()` (which involves significantly more complexity). The new planner has an additional, private entry point `makeJoinPlan(fromClause, conjuncts)`, which builds an optimal join plan from a given `FromClause` and conjuncts that may be able to be applied. The method itself is implemented for you, since it just coordinates the phases of the planner and makes sure that the interesting methods have the arguments they need. Your remaining tasks are to fill in the implementations of each of the phases.

Collecting Details

The detail-collection is performed by the `collectDetails()` method in the join planner, to collect both the set of conjuncts and the list of leaf `FromClauses` from the specified `FromClause` argument.

Task 2: `collectDetails()`

Implement the `collectDetails` method according to its specification, and fill in its Javadoc. Some notes:

- The method's return type is `void`, but "returns" via argument. This makes recursive calls straightforward.
- As discussed earlier, base tables, subqueries, outer joins, and `NATURAL/USING` joins are leaves. See the [FromClause](#) documentation or your Project 2 code for details.
- Only collect predicates from predicates on non-leaf `FromClauses`.
- You will find the helper method [PredicateUtils#collectConjuncts\(\)](#) useful. (This doesn't handle everything... but it's enough for our purposes.)

Leaf Plans

Our bottom-up optimizer starts with optimal leaf plans. These plans are generated by the method `generateLeafJoinComponents()`, which calls the helper `makeLeafPlan()`.

Task 3: `makeLeafPlan()`

Implement this method to generate an optimal plan for each leaf node. You should begin by reusing your general structure for handling the `FROM` clause from Project 2.

Although we're treating outer joins as leaves, we still need to optimize their children. Recursively call `makeJoinPlan()` on each child before creating the outer join plan node. You can also use [FromClause#hasOuterJoinOnLeft\(\)](#) (or `OnRight`) to determine when conjuncts may be passed to the recursive invocation.

Similarly, for `NATURAL/USING` joins, recursively optimize both children before creating the join plan node and projecting out duplicate columns.

We provide several helpers to help with pushing conjuncts down into leaf nodes:

- [PredicateUtils#findExprsUsingSchemas\(\)](#) takes a collection of expressions and one or more schemas, and finds all expressions that can be evaluated against the specified schemas.
 - For example, given a schema for table `t2` and the conjuncts `(t1.a = t2.b, t2.b > 5)`, the method would pull out the second conjunct, but not the first (since it requires `t1` as well).
 - `dst` is best set as a new `HashSet<Expression>`. Note the optional argument; when generating leaf plans, the input collection should be left alone.
 - If a plan node hasn't had [prepare\(\)](#) called, [getSchema\(\)](#) will return `null`. Be sure to call `prepare()` as little as possible, since it traverses an entire plan recursively!
- [PredicateUtils#makePredicate\(\)](#) takes a collection of conjuncts, and builds a predicate that can be applied to a plan (or `null`).
- [PlanUtils#addPredicateToPlan\(\)](#) can be used in a similar way as in Project 2.

Finally, when you change a plan (including adding a predicate), you must call `prepare()` on it again. The plan's cost and statistics must be updated to reflect the new predicate.

Join Plans

Now that we have leaf plans, we can implement a dynamic programming algorithm to combine the optimal leaf plans into a left-deep join tree. As discussed in class, we use a simplified version of a System R (Selinger) optimizer, where we will only consider left-deep plan trees. This allows us to use the optimal ways to produce any set of n joined leaves to find the optimal ways to produce any set of $n + 1$ joined leaves.

For a specific iteration of the algorithm, it will look something like the following:

```
  ▷ JoinPlansN: map from sets of  $N$  leaves to plans that join those leaves
  ▷ JoinPlansN+1: map from sets of  $N + 1$  leaves to plans that join those leaves
  ▷ LeafPlans: set of all leaf plans
1  for planLeaves, plan $n$  in JoinPlans $N$ 
2  for leaf in LeafPlans
3  | if leaf in planLeaves
4  | | ▷ Leaf already joined by current plan
5  | | continue
6  | plan $n+1$  ← plan $n$  ⋈ leaf
7  | newCost ← cost(plan $n+1$ )
8  | newLeaves ← planLeaves ∪ {leaf}
9  | if newLeaves in JoinPlans $N+1$ 
10 | | if newCost < cost(JoinPlans $N+1$ [newLeaves])
11 | | | ▷ plan $n+1$  is the new best plan for newLeaves
12 | | | JoinPlans $N+1$ [newLeaves] ← plan $n+1$ 
13 | else
14 | | ▷ First plan that combines newLeaves
15 | | JoinPlans $N+1$ [newLeaves] ← plan $n+1$ 
```

In addition to the join plans and the sets of leaf nodes they join, we should also track the set of conjuncts applied within each join plan so that we can determine which conjuncts are left to apply to each theta join. Specifically, for a plan node $R \bowtie S$:

- The conjuncts applied to the subplan are $\text{conjuncts}(R) \cup \text{conjuncts}(S)$.
- The unused conjuncts are the set difference of all conjuncts and the subplan's conjuncts.

Of the unused conjuncts, we determine which should be applied to the theta join, then store these alongside the new plan.

Task 4: generateOptimalJoin()

Implement the dynamic programming algorithm described above. Some details:

- Use `cpuCost` as your comparison metric.
- All `Expression` and `PlanNode` classes implement `hashCode()` and `equals()`, allowing them to be used in [HashSets](#) and as keys in [HashMaps](#).
- A `HashSet` can be used as a `HashMap` key, but be very careful with accidentally invalidating the key by mutating it, or otherwise mutating things you don't intend to. While `HashSets` provide methods for set operations, they mutate the object they are called from; use the copying constructor to avoid this.
- You'll likely need to have a collection with the following type: `HashMap<HashSet<PlanNode>, JoinComponent>`. As mentioned earlier, the `JoinComponent` tracks the plan and its conjuncts and leaves.
- [PredicateUtils#findExprsUsingSchemas\(\)](#) can take multiple schemas, which will help with finding unused conjuncts.
- Remember to call `prepare()` on any newly created plan nodes. You should only have to call it once.

Putting it Together

Now that you've implemented all the pieces of join optimization, the last step is to include it in `makePlan` alongside the features you wrote in `SimplePlanner`.

Task 5: makePlan()

Using your Project 2 implementation as a base, implement `makePlan()`. The main difference is that your `FromClause` handling will instead be `makeJoinPlan()`. Some notes:

- Make sure to pass along predicates from the top-level `WHERE` clause.
- After receiving the optimal join plan, make sure to apply any unused conjuncts and any grouping, sorting, etc.

Remember that you moved some code to the common base class. You can continue refactoring here, but some code may be better to leave duplicated.

Testing

Before testing, make sure to tell NanoDB to use the new planner with the `DEFAULT_PLANNER_CLASS` constant in the `ServerProperties` class. Also make sure that your tables are `ANALYZED`.

First, verify that your planner can handle simple queries like:

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

Then, verify more complex queries, such as this three-table join from the previous 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, our reference solution always gets 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]
```

Again, the exact costs will likely vary between your implementation and ours, but the tree itself should be the same. You should also check against Project 2's tests to verify that your optimized queries still produce correct results.

Design Document

Design Doc

Answer the questions in the file `doc/04-join-optimization-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.