Assignment 3
Query Optimization

1 Introduction

The PostgreSQL query optimizer is based on bottom-up enumeration of plans. At each level, possible plans are constructed and the best plans are kept. A plan `A` is considered better than a plan `B` if the cost of the former is less than the latter and the order of the output tuples of plan `A` is at least as "strong" as the order of the tuples in plan `B`'s result.[1] That means that there is a possibility of keeping a plan with a higher cost whenever it has a unique ordering of resulting tuples that cannot be guaranteed by alternative plans.

In `DB2`, a set of "interesting orders" is constructed based on the query properties, such as the desired order of the output, the grouping attributes, DISTINCT attributes and the attributes that appear in join conditions. These interesting orders may be beneficial to query evaluation and hence the optimizer attempts to maintain them at all planning levels whenever possible. PostgreSQL adopts an alternative approach, which is to retrain interesting ordering that are available and to enforce non-existent orders only when needed. For example, before performing a merge join or grouping, the optimizers inserts an explicit sort operator whenever the input relation(s) do not have the ordering required by the merge join or grouping operator. A drawback of this approach is that optimization opportunities could be missed because some plans are not enumerated by the optimizer. For example, consider the following query that joins two relations, `Emp` and `Dept` and returns the output ordered on the attribute `Dept.deptname`.

```
SELECT *
FROM Emp E, Dept D
WHERE E.dno=D.dno
ORDER BY D.deptname;
```

The plan depicted in Figure 1 cannot be generated by the PostgreSQL optimizer. The reason is that nested loop join operator does not require any ordering of the input relations, and thus no sort operators are inserted before the join. At later stage in the plan, the optimizer will enforces the required output tuple ordering (on `deptname`) by inserting a sort operator above the nested loop operator. However, if many employees match each department, this plan may be much more expensive than the plan in Figure 1.

In this assignment, you are required to modify the current implementation of the PostgreSQL query optimizer to eagerly exploit the interesting orders in query planning. This will allow PostgreSQL to consider plans such as the one shown in Figure 1. In general, PostgreSQL could potentially eagerly generate interesting orders at all levels of join enumeration. However, to simplify this assignment, you are only required to make PostgreSQL do so during first level of enumeration, i.e., during the enumeration of *single-relation* sub-plans. Furthermore, you are only required to make PostgreSQL do so for single-relation plans that use sequential scans. For such plans, you must modify the PostgreSQL optimizer to generate sequential scan plans that produce all interesting orders. PostgreSQL can generate the required orders by adding sort operators above the sequential scans.

2 Sort Orders in PostgreSQL

During the planning process, the optimizer builds "Path" trees representing the different ways of executing a query. The cheapest Path that respects the query's required tuple ordering is selected and converted into a Plan to pass to the executor. Each Path records the order of its output in a structure named "PathKeys". The PathKeys data structure represents what is known about the sort order of a particular Path.

[1] We say that plan `A`'s sort order is stronger than plan `B`'s if `B`'s list of sort attributes is a prefix of `A`'s list of sort attributes. For example, if `A` is sorted on `Latitude,Longitude` and `B` is sorted on `Latitude` or is not sorted at all, then `A`'s sort order is stronger than `B`'s.

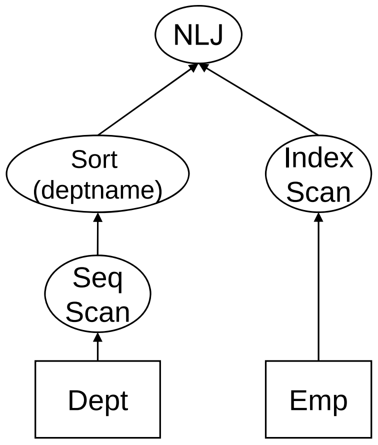

Figure 1: A plan that cannot be generated by the PostgreSQL optimizer.

A Pathkey is a List of Lists of PathKeyItem nodes that represent the sort order of the result generated by the Path. The nth sublist represents the nth sort key of the result. Each sublist contains all equivalent keys across all relations in the query. If we scan the `WHERE` clause for equijoin clauses during planner startup, we can construct lists of equivalent Pathkey items for the query. There could be more than two items per equivalence set; for example, `WHERE A.X = B.Y AND B.Y = C.Z AND D.R = E.S` creates the equivalence sets { `A.X B.Y C.Z` } and { `D.R E.S` }. Any Pathkey item that belongs to an equivalence set implies that all the other items in its set apply to the relation too. In fact, the canonical representation of a Pathkey sublist is a pointer directly to the relevant equivalence set, which is kept in a list of Pathkey equivalence sets for the query. Then Pathkey sublist comparison reduces to pointer-equality checking.

When constructing a path, the query optimizer enforces ordering of the tuples when the current operator requires its input(s) to be sorted. For example, when constructing a merge join path, the optimizer checks whether the input relations are properly ordered or not and appends a sort operator to the unordered input(s).

2.1 Relevant Files

Here, we describe files, structures and functions that may need to be changed or could be useful for the purposes of this assignment.

- `src/backend/optimizer/path/allpaths.c`: This file include routines necessary to find possible search paths for processing a query. One function of special interest is named `set_plain_rel_pathlist`. This function creates all possible sequential scan and index scan paths for a certain base relation. You will need to modify this function to create as many paths as needed to cover interesting orders.
- `src/backend/optimizer/path/costsize.c` This file includes the query operator cost functions. You will have to redefine the cost of scan paths to include the cost of any following sorting operation.
- `src/backend/optimizer/util/pathnode.c` This file contains procedures to construct path nodes of any type. You may need to modify the function responsible for creating access paths to include any necessary information.
- `src/backend/optimizer/util/var.c` This file defines all functions necessary to handle the `Var` structure. This structure represents a column (attribute) in a relation. You may need to do column equality checking while considering interesting orders.
- `src/backend/optimizer/path/pathkeys.c` This file includes all routines to handle Pathkeys, e.g., Pathkeys comparisons and construction.

- `src/backend/optimizer/plan/createplan.c` This file contains all of the functions for converting paths into plan trees to be passed to the executer. You have to override the creation of access path nodes to create necessary sort nodes whenever necessary.
- `src/backend/optimizer/plan/planmain.c` This files contains the main code for planning a basic join operation, shorn of features like subselects, inheritance, aggregates, grouping, and so on. It passes the created plan to the main planner procedures in `planner.c` to add these nodes.
- `src/include/optimizer/` This directory contains all header files that define data structures and functions prototypes.
- `src/backend/optimizer/README` A general overview of query optimization in PostgreSQL , including a discussion of `PathKeys`

Not all of the files listed above will have to be modified. The purposes of this list is to help you limit the scope of the source code that you will need to work with for this assignment.

2.2 Useful Hints

- The scope of static functions is restricted to the files in which they are declared. While debugging static functions, local variables cannot be inspected.
- Pathkey comparisons are done by checking pointer equality. Be careful not to create new Pathkeys of your own, otherwise they will not be detected as useful orders at higher levels.
- You may use the function `NodeToString` to convert the contents of a `Node` structure into a string. It can also be used with any subtype (e.g. `Var, PathKeys, List`). Furthermore, you may use the function `print_path` to print a Path to the default output.

3 Problem Statement

In this assignment, you are required to exploit interesting orders while creating sequential scan access paths to base relations. This assignment can be divided into subtasks as follows.

1. Extract interesting orders from query (specifically from the `PlannerInfo` structure). Interesting orders include attributes in the `Order By` clause and the `Group By` clause. They also include equality predicates which can be used in merge-joining relations. (20%)
2. Once you have all interesting orders, you should modify sequential scan access paths so that they include a description of the orders that the optimizer should consider imposing on them. In other words, you must make it possible to record explicit sorting requirements for each sequential scan access path. (50%)
3. Cost estimation for access paths has to be modified to add any additional costs from sorting. (10%)
4. Once the optimal Path has been found, it has to be converted into a Plan so that it can be passed to the executor. At this point, you have to insert sorting operators based on the desired ordering of each sequential scan path. (20%)

3.1 Query Planning Output

Before enumerating access paths, you are expected to print all interesting orders that are extracted from the query. The output should also include the constructed paths, their Pathkeys, and the estimated cost. Output should be printed at the client side. There is no strict output format that you have to follow, however, the output should be well-organized and readable.

4 Deliverables

All modified files should be submitted electronically. Only `.h` and `.c` files will be accepted. Assuming that the current directory contains all modified files, you can use the following command to submit them:

```
submit cs448 a3 .
```

Make sure to clearly indicate those parts of the files that you have modified by inserting comments before and after the changes. Also, include comments to describe your modifications. All comments should be preceded by `'CS448:'`. If you need to include any other comments or general implementation notes, you can place them in a `README` file and submit it as well.