CS448/648 Database Systems Implementation 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 `A` is less than the cost of `B` and the order of the output tuples of plan `A` is at least as "strong" as the order of the output tuples of plan `B`.[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, and the attributes that appear in join conditions. These interesting orders may be beneficial to query evaluation and hence the optimizer attempts to maintain all interesting orders at all planning levels. PostgreSQL adopts an alternative approach, which is to retain only those interesting ordering that are available, and to enforce other orders only when needed. For example, before performing a merge join or grouping, the optimizer 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 query and corresponding query plan show in Figure 1. That plan 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. Yet this plan *might* be a good plan for executing the query, particularly if many employees match each department.

In this assignment, you are required to modify the PostgreSQL query optimizer so that it will generate interesting orders more aggressively during query planning. This will allow PostgreSQL to consider plans such as the one shown in Figure 1. Specifically, your objective is to ensure that when PostgreSQL generates plans for single-relation subqueries, it will generate and consider plans for *all* interesting orders for that single-relation subquery. PostgreSQL can generate different types of single-relation query plans, e.g., plans using indexes or plans using sequential scans. To simplify the assignment, *you are only required to generate all interesting orders for single-relation plans that use sequential scans.* PostgreSQL can generate the required orders by adding sort operators above the sequential scan operators in these plans.

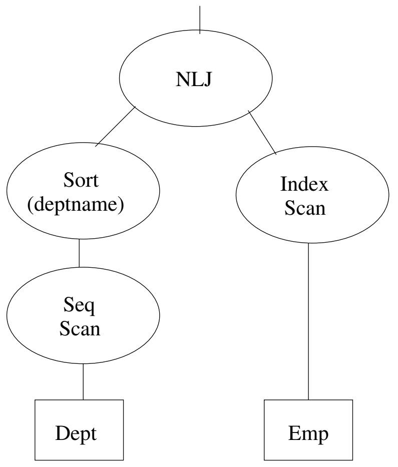
In general, PostgreSQL could aggressively generate interesting orders at all optimization levels, i.e., for single-relation plans, for two-relation plans, for three-relation plans, and so on. For this assignment, you are only required to do so for single-relation plans (and only those using sequential 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 that PostgreSQL refers to as the PathKeys for that Path. The PathKeys are represented as a list of sub-lists of PathKeyItem nodes. The nth sublist represents the nth sort key of the result. Each sub-list identifies a set of equivalent relational attributes. For example, consider the following query:

```
select e.ename
from emp e, dept d, manages m
where d.dno = m.dno and m.eno=e.eno
order by e.ename,e.eno
```

[1]Plan `A`'s sort order is stronger than plan `B`'s if `B`'s sort attributes are a prefix of `A`'s 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.



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

Figure 1: A SQL Query and Impossible Query Plan

PostgreSQL would represent the required output sort order for this query using PathKeys like this:

```
( (e.ename) , (e.eno,m.eno) )
```

This indicates that `e.ename` is the primary sort key, and that both `e.eno` and `m.eno` are the secondary sort keys. The secondary sort key includes two attributes because PostgreSQL knows, from one of the query's join conditions, that the values of those two attributes will always be the same in each output tuple, i.e., the two attributes are equivalent.

2.1 Relevant Files

Here, we describe key files that are relevant to this assignment. You may not need to modify all of these files. However, *any changes that you make must be limited to these files*, as they are the only files that may be submitted.

- `src/backend/optimizer/path/allpaths.c`:
  This file includes routines necessary to find possible 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 sorting operation that is added to enforce an interesting order..
- `src/include/optimizer/cost.h`:
  Contains function signatures and related definitions for the functions implemented in `costsize.c`.
- `src/backend/optimizer/util/pathnode.c`:
  This file contains procedures to construct Paths. You may wish to modify the function responsible for creating access paths to include any necessary information.
- `src/include/optimizer/pathnode.h`:
  Contains function signatures for the functions implemented in `pathnode.c`.
- `src/include/nodes/pg_list.h`:
  Functions and macros for manipulating lists.
- `src/include/nodes/nodes.h`:
  Functions for manipulating and testing `Nodes`, which are found in `PathKeys`.

- `src/include/nodes/relation.h`:
  Defines important structures like `PlannerInfo` and `Path`.
- `src/include/nodes/print.h`:
  Defines useful functions for printing.
- `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 will need to change the way that sequential scan plans are created to ensure that sorting operations are added if required by the path.
- `src/backend/optimizer/README`:
  A general overview of query optimization in PostgreSQL , including a discussion of `PathKey`s

2.2 Useful Hints

- Start this assignment with a fresh copy of the PostgreSQL server source code, to avoid any chance of bugs from your work on earlier assignments affecting your work on this one. This assignment does not depend in any way on your work from the earlier assignments.
- If you are using the same PostgreSQL data directory (`pgdb`) that you used for Assignment 2, be sure to re-enable non-hash joins for this assignment by editing the PostgreSQL server configuration file.
- 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.
- `allpaths.c` file includes useful functions for printing the output required by this assignment, e.g., `print_path`. Other useful printing functions, such as `print_pathkeys`, can be used by including `print.h`.

3 Assignment Details

This assignment can be divided into subtasks as follows.

1. PostgreSQL identifies extracts PathKeys from the `Group By` and `Order By` query clauses, and also identifies attributes the appear in join conditions. Using this information, you will need to identify the set of interesting orders for each single-relation sub-query that is processed by the PostgreSQL query optimizer.
2. Modify sequential scan access paths so that they can include a description of the required sort order. In other words, you must make it possible to record explicit sorting requirements for each sequential scan access path.
3. Change the way that the optimizer generates Paths for single-relation sequential scan queries so that it will generate sequential scan paths that produce each interesting order, in addition to the paths that it was already generating.
4. Change the way that the optimizer estimates the cost of single-relation sequential scan Paths so that the estimates include the cost of the final sort, for Paths that require one.
5. 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 a sorting operator into the single-relation sequential scan plans if the optimal Path is expected to produce output in an interesting order.

```
CS448 **** Interesting Order from Order By clause: ((e.ename), (m.eno, e.eno))
CS448 **** Interesting Order from Group By clause: ()
CS448 **** Interesting Orders from Join predicates: ((m.eno, e.eno), (d.dno, m.dno))
Possible Paths for Relation 1:
SeqScan(1) rows=20 cost=0.00..1.20
SeqScan(1) rows=20 cost=1.63..1.68
  pathkeys: ((e.ename), (m.eno, e.eno))
SeqScan(1) rows=20 cost=1.63..1.68
  pathkeys: ((m.eno, e.eno))
Possible Paths for Relation 2:
SeqScan(2) rows=5 cost=0.00..1.05
SeqScan(2) rows=5 cost=1.11..1.12
  pathkeys: ((d.dno, m.dno))
Possible Paths for Relation 3:
SeqScan(3) rows=5 cost=0.00..1.05
SeqScan(3) rows=5 cost=1.11..1.12
  pathkeys: ((m.eno, e.eno))
SeqScan(3) rows=5 cost=1.11..1.12
  pathkeys: ((d.dno, m.dno))
```

Figure 2: Example of Required Output Format

3.1 Required Debugging Output

You must modify the PostgreSQL optimizer so that it produces the following output for each query that it optimizes:

- the complete set of interesting orders, if any, arising from Order By clauses, Group By clauses, and join conditions.
- for each relation in the query, the complete list of sequential scan Paths generated by the optimizer for that relation. For each each Path, the output should include costing information (output cardinality estimate and cost estimates) and the pathkeys describing the output order produced by that Path.

For example, for the example query shown in Section 2, the format of your output should be like that shown in Figure 2. The complete set of interesting orders for the full query is listed first, followed by the sequential scan Paths for each input relation. For example, the output in Figure 2 shows that the optimizer considers three Paths for Relation 1 (`emp`), one that produces unordered output, one that produces output sorted by `ename,eno`, and one that produces output sorted by `eno`. The interesting orders can be printed in the format shown in Figure 2 by using the `print_pathkeys` function. Path information can be printed using the `print_path` function.

4 Deliverables

Modified files should be submitted using `submit`. *Only the .c and .h files listed in Section 2.1 will be accepted.* Assuming that the current directory contains all modified files, you can use the following command to submit them:

```
submit cs448 a3 .
```

5 Extra Credit

For extra credit (worth 10% of the value of the assignment), design a SQL join query for which the PostgreSQL optimizer will use one of your newly-created sorted single-relation sequential scan paths as part of its chosen, optimal plan for the query. Your query should go against that simple three table database that was published as a test case for Assignment 2.

To obtain the extra credit, submit an additional file, called `extra.sql`, which contains a single `psql EXPLAIN` of the query you have designed. We will use `psql` to execute the command and inspect the resulting query plan and server output to confirm that it does in fact use a sorted sequential scan path.

Before testing your query, we will run the `psql ANALYZE` command to ensure that the PostgreSQL server's database statistics are up to date. You should also do this when designing your extra credit query to ensure that you can test it under the same testing conditions that we will use.