Exercises

Michail Kazimianec
Werner Nutt
Damiano Somenzi

Indexing and Query Execution

We consider in these exercises queries posed over the employee database that you already know from the SQL exercises. Here is a short reminder of the schema of that database. It contains the following three relations:

- `emp`(`empno`, `ename`, `job`, `mgr`, `hiredate`, `sal`, `comm`, `deptno`)
- `dept`(`deptno`, `dname`, `loc`)
- `salgrade`(`grade`, `losal`, `hisal`)

We do not assume that any primary keys are uniqueness constraints have been declared.

1. Indexes Supporting Selections Queries

Consider the table `emp` the schema of which is given above. Suppose, the following four indexes have been created:

1. `CREATE INDEX ename_idx ON emp(ename);`
2. `CREATE INDEX sal_ename_idx ON emp(sal, ename);`
3. `CREATE INDEX ename_hiredate_idx ON emp(ename, hiredate);`
4. `CREATE INDEX hiredate_sal_ename_idx ON emp(hiredate, sal, ename);`

Suppose, that a user asks the following query

```
SELECT E.ename, E.sal, E.hiredate
FROM   emp E
WHERE  E.ename = 'Smith' AND
       E.sal >= 1000 AND
       E.hiredate >= '01-JAN-2000';
```

(i) For each of the four indexes, describe how the query processor could make use of the index to answer the query.

(ii) Which of the four indexes is likely to yield the greatest benefit? Give a brief explanation.

2. Performance Measurements

In this exercise, we will check whether our theoretical analysis in the previous exercise is valid for real database instances managed by PostgreSQL. We also want to see whether the size of a the instance has an influence on query plans that PostgreSQL produces.

On the lab pages for the course you find the code of four Java classes, `RandomData`, `RandomDept`, `RandomEmp`, and `PrintRandomDeptsAndEmps` (you have to compile them in this order).

The classes can be used to create random data for tests. In those data sets, `deptno` is a key for departments, departments are located in one of 10 cities, `empno` is a key for employees, `employees` can have one of 10 jobs, only salesmen have a commission, and and the `deptno` of an employee is a foreign key for `dept`.

When you call `PrintRandomDeptsAndEmps`, you will be asked for the number of departments and the number of employees in your test data set, and for the files where each of the two sets should be written.

Those data can be loaded into tables using the PostgreSQL bulk loader. The bulk loader can be called with the `COPY` command and from within `psql` with the command `\copy`. Syntax and explanations can be found in the documentation at

`http://www.postgresql.org/docs/manuals/`.

Note that you have to be a superuser if you want to use `COPY`, while `\copy` is accessible to everyone.

Create versions `emp1`, `emp2`, etc. of the `emp` table and populate them with instances of varying size, e.g., consisting of 10^3, 10^4, 10^5, 10^6 records. Choose the number of departments by a factor of 10^2 less than the number of employees.

For each table, create indexes like the ones in Exercise 1.

(i) Find out, whether and how each of the indexes improve query execution.

(ii) Check which is the order in which the PostgreSQL optmiser prioritises the indexes.

(iii) Do the choices of the optimizer depend on the size of the tables?

(iv) Explain your observations.

Hint: Use pgAdmin for this exercise.

3. Execution of Join Queries

For each of the tables `emp1`, `emp2`, etc., create a corresponding department table `dept1`, `dept2`, etc. and populate it with the department data created together with the employee data. Drop all the indexes on the employee tables so that you can see what the system would do without any support.
We consider the query:

```
SELECT *
FROM   emp NATURAL JOIN dept;
```

(i) Find how PostgreSQL plans to execute this query over the various instances of `emp` and `dept`. (Of course, you have to submit the query with $\texttt{emp}_i$ instead of `emp` and $\texttt{dept}_j$ instead of `dept`.)

(ii) Are all the plans similar or not? Explain your findings!

One might wonder whether a (unique) index on the attribute `deptno` of `dept` may speed up the join. To find out, create a unique index `dept1_deptno_idx` on `dept1` and so on.

(iii) Do the plans change? If so, how? Can you explain your findings?

Next we want to investigate joins with additional selections on the participating relations.
We consider a join of the previous query on `emp` with `dept`, that is:

```
SELECT *
FROM   emp E NATURAL JOIN dept D
WHERE  E.ename = 'Smith' AND
       E.sal >= 1000 AND
       E.hiredate >= '01-JAN-2000';
```

(iv) Find how PostgreSQL plans to execute this query over the various instances of `emp` and `dept`.

(v) Are all the plans similar or not? Explain your findings!

Summarize your findings:

(vi) Try to explain under which conditions PostgreSQL chooses each of the join methods sort merge join, nested loops join, and hash join.

Note that the query optimiser hasn't built in any criteria for preferring one method over the other, but the preferences are consequences of the cost model.

4. Interpretation of Query Plans

We have loaded 10^5 employee records into the relation `emp3` and 10^3 department records into the relation `dept3`. We also have created indexes as described in Exercise 1.
We ask for the plan for the query

```
SELECT count(*)
FROM   emp3 e NATURAL JOIN dept3 d
WHERE  e.ename = 'Smith' AND
       e.sal > 2000 AND
       e.hiredate > '01-JAN-2000';
```

and see the following text

```
Aggregate  (cost=29.06..29.07 rows=1 width=0)
-> Hash Join  (cost=8.30..29.06 rows=1 width=0)
   Hash Cond: (d.deptno = e.deptno)
   -> Seq Scan on dept3 d  (cost=0.00..17.00 rows=1000 width=4)
   -> Hash  (cost=8.29..8.29 rows=1 width=4)
      -> Index Scan using ename3_idx on emp3 e  (cost=0.00..8.29 rows=1
         width=4)
         Index Cond: ((ename)::text = 'Smith'::text)
         Filter: ((sal > 2000) AND (hiredate > '2000-01-01'::date))
```

(i) Explain how PostgreSQL will execute this query. Draw a relational algebra tree where operations are annotated by the method used to run them. Include not only the operations mentioned but also all the selections and projections that remain implicit.

We create an index with the statement

```
CREATE INDEX dept3_deptno_idx ON dept3(deptno);
```

When we ask again for an execution plan for the query, we receive the answer

```
Aggregate  (cost=16.57..16.58 rows=1 width=0)
-> Nested Loop  (cost=0.00..16.57 rows=1 width=0)
   -> Index Scan using ename3_idx on emp3 e  (cost=0.00..8.29 rows=1 width=4)
      Index Cond: ((ename)::text = 'Smith'::text)
```

```
      Filter: ((sal > 2000) AND (hiredate > '2000-01-01'::date))
   -> Index Scan using dept3_deptno_idx on dept3 d  (cost=0.00..8.27 rows=1
      width=4)
      Index Cond: (d.deptno = e.deptno)
```

(ii) Explain what has changed in the execution plan. Draw again a relational algebra tree where operations are annotated by the method used to run them. Include also all the selections and projections that remain implicit.

Over the same database we ask for the plan for the query where the condition on `name` has been dropped.

```
SELECT *
FROM   emp3 e NATURAL JOIN dept3 d
WHERE  e.sal > 2000 AND
       e.hiredate > '01-JAN-2000';
```

and see the output

```
Hash Join  (cost=29.50..2722.09 rows=24116 width=61)
  Hash Cond: (e.deptno = d.deptno)
  -> Seq Scan on emp3 e  (cost=0.00..2361.00 rows=24116 width=43)
     Filter: ((sal > 2000) AND (hiredate > '2000-01-01'::date))
  -> Hash  (cost=17.00..17.00 rows=1000 width=22)
     -> Seq Scan on dept3 d  (cost=0.00..17.00 rows=1000 width=22)
```

(iii) Explain how PostgreSQL will execute this query. Draw a relational algebra tree as required before.

We tighten the two conditions in the `WHERE` clause so as to obtain

```
SELECT *
FROM emp3 e NATURAL JOIN dept3 d
WHERE e.sal  = 4000 AND
      e.hiredate > '01-JAN-2005';
```

The plan for that query is

```
Hash Join  (cost=94.69..1002.03 rows=340 width=61)
  Hash Cond: (e.deptno = d.deptno)"
  -> Bitmap Heap Scan on emp3 e  (cost=65.19..967.86 rows=340 width=43)
     Recheck Cond: (sal = 4000)
     Filter: (hiredate > '2005-01-01'::date)
     -> Bitmap Index Scan on sal_ename3_idx  (cost=0.00..65.10
        rows=2778 width=0)
        Index Cond: (sal = 4000)
  -> Hash  (cost=17.00..17.00 rows=1000 width=22)
     -> Seq Scan on dept3 d  (cost=0.00..17.00 rows=1000 width=22)
```

(iv) Explain how PostgreSQL will execute this query. Draw a relational algebra tree as required before.

If we replace the `AND` in the `WHERE` clause by an `OR`, the resulting query is

```
SELECT *
FROM emp3 e NATURAL JOIN dept3 d
WHERE e.sal  = 4000 OR
      e.hiredate > '01-JAN-2005';
```

The execution plan looks as follows:

```
Hash Join  (cost=409.97..1697.95 rows=14674 width=61)
  Hash Cond: (e.deptno = d.deptno)
  -> Bitmap Heap Scan on emp3 e  (cost=380.47..1466.68
     rows=14674 width=43)
     Recheck Cond: ((sal = 4000) OR (hiredate > '2005-01-01'::date))
     -> BitmapOr  (cost=380.47..380.47 rows=15014 width=0)
        -> Bitmap Index Scan on sal_ename3_idx  (cost=0.00..65.10
           rows=2778 width=0)
           Index Cond: (sal = 4000)
        -> Bitmap Index Scan on hiredate_sal_ename3_idx
           (cost=0.00..308.04 rows=12236 width=0)
           Index Cond: (hiredate > '2005-01-01'::date)
  ->  Hash (cost=17.00..17.00 rows=1000 width=22)
     ->  Seq Scan on dept3 d  (cost=0.00..17.00 rows=1000 width=22)
```

(v) Explain how PostgreSQL will execute this query.

5. Supporting Join Queries

Create yourself a table `emp3` as in Exercise 4, without declaring a primary key or uniqueness constraints. Write a query that asks:

What are the names of the employees managed by Smith?

(i) Check out the execution plan for that query and the run time.

(ii) Find out which is the best choice of indexes to speed up the execution of that query.

(iii) What is the speed-up you gain by indexing?

(iv) It is very unlikely that in your relation `emp3` there is an employee with name Smith. Look up the name of some tuple occurring in `emp3`, replace the name Smith in your query with that name, and redo the performance test. What is the speed-up gain now?

6. Queries with Negation

We want to find out whether different forms of negation have an impact on the execution plan and the run time of a query. Consider the query

```
select *
from   dept3 d
where  not exists
       (select e.deptno
        from   emp3 e
        where  e.deptno = d.deptno AND
               e.job = 'Driver');
```

(i) Give an equivalent formulation as a query using "`NOT IN`".

(ii) Compare the run time of the two queries. Which one is faster?

(iii) Analyze the query plans to explain your findings.