Lab Exercise 07
Relational Design Theory

Aims

This exercise aims to:

- explore how PostgreSQL evaluates queries
- give you practice is reading PostgreSQL query plans

Background

PostgreSQL, like any serious DBMS, has a query optimiser that attempts to map SQL statements into efficient query execution plans. SQL developers can access these query execution plans via the `EXPLAIN` statement (a PostgreSQL extension to SQL). If you prefix an SQL query statement by the keyword `EXPLAIN`, then PostgreSQL prints the query execution plan *instead of* running the query. This plan includes *estimates* from the query optimiser of the cost of each of the operations. If you prefix the query with `EXPLAIN ANALYZE`, then PostgreSQL executes the query, but does not display the results; instead it displays the query execution plan, giving both the estimated and the actual cost for each operation executed during query evaluation.

For this lab exercise, we'll use the UNSWSIS database (Similar to project 1 but not the same.). You can load it from the file:

```
/home/cs9311/web/24T2/lab/07/db.dump
```

The rest of this Lab assumes that you've got the UNSWSIS database loaded into your PostgreSQL server and called `lab09`.

Consider the following simple query to find out whether a particular student is local or international:

```
lab09=# select stype from Students where sid=2237112;
 stype
-------
 local
(1 row)
```

Recall the definition of this relation and think about how PostgreSQL might go about answering the query above:

```
create table Students (
        id        integer primary key references Person(id),
        sid       integer unique not null, -- student id
        stype     varchar(5) check (stype in ('local','intl'))
);
```

We know that PostgreSQL automatically creates a B-tree index for all primary keys, so clearly a query involving `Student.id` will make use of that index. To find out how PostgreSQL answers the query above, we can use `explain`:

```
lab09=# explain select stype from Students where sid=2237112;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using students_sid_key on students  (cost=0.00..5.94 rows=1 width=5)
   Index Cond: (sid = 2237112)
(2 rows)
```

The first thing to notice is that it's doing the selection operation via an index; this is because `Students.sid` was declared to be `unique` and PostgreSQL also creates indexes on all `unique` fields. Check the definition of the `Student` table in the `lab09` database to confirm this.

Now let's look at the `explain` output in more detail:

- `Index Scan` tells us that PostgreSQL is performing a selection operation on the `students` table using an index (B-tree) called `students_sid_key`
- the `cost` components are:
  - `0.00` : expected time (ms) until first result is available (not used in this example)
  - `5.94` : expected time (ms) until all results have been found
- `rows=1` : expected number of result tuples
- `width=5` : size (bytes) of each result tuple (exact)
- `Index Cond` ition tells us the filtering condition used to access the B-tree index
- the `(2 rows)` simply tells us how many lines there are in the query plan description; it has nothing to do with the query itself.

The above query evaluation plan gives an *estimate* of the cost. If we want to find out the actual costs, we could use `explain analyze` :

```
lab09=# explain analyze select stype from Students where sid=2237112;
                          QUERY PLAN
-------------------------------------------------------------
 Index Scan using students_sid_key on students
   (cost=0.00..5.94 rows=1 width=5)
   (actual time=0.048..0.051 rows=1 loops=1)
   Index Cond: (sid = 2237112)
 Total runtime: 0.114 ms
(3 rows)
```

The `analyze` provides us with the following additional information:

- `actual time` gives
  - `0.048` : the time (ms) to generate the first tuple
  - `0.051` : the time (ms) that the last tuple was sent

  ("Why are they different?", you say, "since only one tuple was generated". The timers used for pretty much anything on Unix are typically not accurate to the number of decimal places to which they are displayed. If PostgreSQL had used only two decimal places, it would have said 0.05 for both, and nobody would have wondered about it.)
- `rows=1` : actual number of result tuples (because it was a `unique` attribute, the estimate above was exact)
- `runtime: 0.114 ms` elapsed time until last tuple arrives

You can find more details on `EXPLAIN` in the PostgreSQL manual.

Note that I have changed the formatting from what you'll actually see in `psql` , because the lines become ridiculously wide. You'll probably need a window that occupies the entire workstation screen width in order to make sense of the analyzed evaluation plans. Note also that you will observe different actual times to what is shown here, simply because of the imprecise nature of timing under Unix.

Consider a more complex example, using a simple join, to find out the name of student, given their student id:

```
lab09=# select p.name from People p, Students s
lab09-# where p.id = s.id and s.sid = 2237112;
      name
-----------------
 Christopher Kay
(1 row)

lab09=# explain analyze
lab09-# select p.name from People p, Students s
lab09-# where p.id = s.id and s.sid = 2237112;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop
     (cost=0.00..11.94 rows=1 width=15)
     (actual time=0.074..0.079 rows=1 loops=1)
   ->  Index Scan using students_sid_key on students s
         (cost=0.00..5.94 rows=1 width=4)
         (actual time=0.040..0.042 rows=1 loops=1)
         Index Cond: (sid = 2237112)
   ->  Index Scan using people_pkey on person p
         (cost=0.00..6.00 rows=1 width=19)
         (actual time=0.019..0.022 rows=1 loops=1)
```

```
            Index Cond: (id = s.id)
    Total runtime: 0.193 ms
   (6 rows)
```

The query plan shows us that PostgreSQL uses the simple (and often regarded as inefficient) nested-loop join operation to join the two tables. It can do this, because the results of the selection on `Students.sid` consists of only one tuple. Now consider a more complex example, using a three-way join to produce the "standard representation" for each course in the UNSWSIS database

```
lab09=# explain analyze
lab09-# select s.code,t.year,t.sess
lab09-# from   Subjects s, Courses c, Terms t
lab09-# where  s.id = c.subject and c.term = t.id;
                                QUERY PLAN
-------------------------------------------------------------------------
Hash Join
(cost=679.25..1224.74 rows=10922 width=16)
(actual time=15.932..63.463 rows=10922 loops=1)
Hash Cond: (c.term = t.id)
->   Hash Join
  (cost=677.08..1072.39 rows=10922 width=13)
      (actual time=15.805..45.150 rows=10922 loops=1)
       Hash Cond: (c.subject = s.id)
      ->  Seq Scan on courses c
            (cost=0.00..163.22 rows=10922 width=8)
            (actual time=0.012..7.995 rows=10922 loops=1)
      ->  Hash
            (cost=582.59..582.59 rows=7559 width=13)
            (actual time=15.774..15.774 rows=7559 loops=1)
          Buckets: 1024  Batches: 1  Memory Usage: 266kB
          ->  Seq Scan on subjects s
                      (cost=0.00..582.59 rows=7559 width=13)
                      (actual time=0.006..7.824 rows=7559 loops=1)
 ->  Hash
      (cost=1.52..1.52 rows=52 width=11)
      (actual time=0.111..0.111 rows=52 loops=1)
       Buckets: 1024  Batches: 1  Memory Usage: 2kB
      ->  Seq Scan on terms t
            (cost=0.00..1.52 rows=52 width=11)
            (actual time=0.010..0.055 rows=52 loops=1)
 Total runtime: 70.050 ms
(12 rows)
```

You'll notice that there are two places where a `Seq Scan` (sequential scan) peration is fed into a `Hash` operation. This is PostgreSQL converting a single table into a hashed file, because it allows it to use the very efficient `Hash Join` operation in a subsequent step.

As a final example, consider the following two queries to show which UNSW people come from either Nepal or Finland:

```
lab09=# explain analyze
lab09-# select p.name from People p, Countries c
lab09-# where p.citizen = c.id and (c.name = 'Nepal' or c.name='Finland');
                          QUERY PLAN
---------------------------------------------------------------

  Hash Join
 (cost=4.89..143.17 rows=39 width=15)
 (actual time=2.629..7.344 rows=2 loops=1)
    Hash Cond: (p.citizen = c.id)
   ->  Seq Scan on people p
   (cost=0.00..124.01 rows=3701 width=19)
   (actual time=0.013..3.261 rows=3701 loops=1)
```

```
   ->  Hash
   (cost=4.87..4.87 rows=2 width=4)
   (actual time=0.107..0.107 rows=2 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on countries c  (cost=0.00..4.87 rows=2 width=4) (actual t
ime=0.044..0.100 rows=2 loops=1)
               Filter: (((name)::text = 'Nepal'::text) OR
                              ((name)::text = 'Finland'::text))
             Rows Removed by Filter: 189
 Total runtime: 7.396 ms
(9 rows)



lab09=# explain analyze
lab09-# select p.name from People p
lab09-# where p.citizen in
lab09-#         (select id from Countries where name = 'Nepal' or name='Finland');
                          QUERY PLAN
---------------------------------------------------------------

Hash Semi Join
(cost=4.89..139.74 rows=101 width=15)
(actual time=2.548..7.171 rows=2 loops=1)
    Hash Cond: (p.citizen = countries.id)
   ->  Seq Scan on people p
   (cost=0.00..124.01 rows=3701 width=19)
   (actual time=0.011..3.118 rows=3701 loops=1)
   ->  Hash  (cost=4.87..4.87 rows=2 width=4) (actual time=0.108..0.108 rows=2 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on countries
                (cost=0.00..4.87 rows=2 width=4)
                (actual time=0.042..0.101 rows=2 loops=1)
               Filter: (((name)::text = 'Nepal'::text) OR
                        ((name)::text = 'Finland'::text)
)
             Rows Removed by Filter: 189
 Total runtime: 7.223 ms
(9 rows)
```

Exercises

For each of the following queries, produce your SQL code to answer and use the PostgreSQL query analyzer to determine the query evaluation plan:

1. Find all students who scored more than 97 in a UNSW course:
2. Produce a transcript for student 2176435
3. Produce a list of all students enrolled in COMPxxx courses in 2002

For Fun

Consider the following alternative solutions for a query something like the `FullyEnrolled(1998,2002)` function:

```
-- first version (follows standard mapping of DIVISION)
select s.sid
from   Students s
where  not exists (
               (select id from Terms
                where year >= 1998 and year <= 2002 and sess in ('S1','S2'))
               except
               (select distinct c.term from CourseEnrolments e, Courses c
```

```
                    where e.student=s.id and e.course=c.id)
        )
;

-- second version (using group-by and counting)
select s.sid
from   Students s, CourseEnrolments e, Courses c, Terms t
where  e.student = s.id and e.course = c.id and c.term = t.id
        and t.year >= 1998 and t.year <= 2002 and t.sess in ('S1','S2')
group by s.sid
having count(distinct c.term) =
        (select count(id) from Terms
         where year >= 1998 and year <= 2002 and sess in ('S1','S2'))
;

-- third version (using group-by and faster counting)
select s.sid
from   Students s, CourseEnrolments e, Courses c, Terms t
where  e.student = s.id and e.course = c.id and c.term = t.id
        and t.year >= 1998 and t.year <= 2002 and t.sess in ('S1','S2')
group by s.sid
having count(distinct c.term) = 10   -- number of s1/s2 terms in 1998..2002
;
```

Run them using `explain` and compare the query plans to see if you can work out why one is so much slower than the others. If you're really brave, run `explain analyze` .

[Solution]