>>

Query Performance Tuning

^ >>

❖ Query Performance Tuning

What to do if the DBMS takes "too long" to answer some queries?

Improving performance may involve any/all of:

- making applications using the DB run faster
- lowering response time of queries/transactions
- improving overall transaction throughput

Remembering that, to some extent ...

- the query optimiser removes choices from DB developers
- by making its own decision on the optimal execution plan

<< ^ >>

❖ Query Performance Tuning (cont)

Tuning requires us to consider the following:

- which queries and transactions will be used?
  (e.g. check balance for payment, display recent transaction history)
- how frequently does each query/transaction occur?
  (e.g. 80% withdrawals; 1% deposits; 19% balance check)
- are there time constraints on queries/transactions?
  (e.g. EFTPOS payments must be approved within 7 seconds)
- are there uniqueness constraints on any attributes?
  (define indexes on attributes to speed up insertion uniqueness check)
- how frequently do updates occur?
  (indexes slow down updates, because must update table *and* index)

<< ^ >>

❖ Query Performance Tuning (cont)

Performance can be considered at two times:

- *during* schema design
  - typically towards the end of schema design process
  - requires schema transformations such as denormalisation
- *outside* schema design
  - typically after application has been deployed/used
  - requires adding/modifying data structures such as indexes

Difficult to predict what query optimiser will do, so ...

- implement queries using methods which *should* be efficient
- observe execution behaviour and modify query accordingly

<< ^ >>

❖ PostgreSQL Query Tuning

PostgreSQL provides the **`explain`** statement to

- give a representation of the query execution plan
- with information that may help to tune query performance

Usage:

```
EXPLAIN [ANALYZE] Query
```

Without **`ANALYZE`**, **`EXPLAIN`** shows plan with estimated costs.

With **`ANALYZE`**, **`EXPLAIN`** executes query and prints real costs.

Note that runtimes may show considerable variation due to buffering.

❖ EXPLAIN examples

Using the following database ...

```
CourseEnrolments(student, course, mark, grade, ...)
Courses(id, subject, semester, homepage)
People(id, family, given, title, name, ..., birthday)
ProgramEnrolments(id, student, semester, program, wam, ...)
Students(id, stype)
Subjects(id, code, name, longname, uoc, offeredby, ...)
```

with a view defined as

```
create view EnrolmentCounts as
 select s.code, c.semester, count(e.student) as nstudes
   from Courses c join Subjects s on c.subject=s.id
        join Course_enrolments e on e.course = c.id
  group by s.code, c.semester;
```

<< ^ >>

❖ EXPLAIN examples (cont)

Some database statistics:

```
         tab_name          | n_records
---------------------------+-----------
 courseenrolments          |    503120
 courses                   |     71288
 people                    |     36497
 programenrolments         |    161110
 students                  |     31048
 subjects                  |     18799
```

<< ^ >>

❖ EXPLAIN examples (cont)

Example: Select on non-indexed attribute

```
uni=# explain
uni=# select * from Students where stype='local';
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on students
              (cost=0.00..562.01 rows=23544 width=9)
   Filter: ((stype)::text = 'local'::text)
```

where

- **`Seq Scan`** = operation (plan node)
- **`cost=`***StartUpCost*`..`*TotalCost*
- **`rows=`***NumberOfResultTuples*
- **`width=`***SizeOfTuple* (# bytes)

<< ^ >>

❖ EXPLAIN examples (cont)

More notes on **`explain`** output:

- each major entry corresponds to a plan node
  - e.g. **`Seq Scan`**, **`Index Scan`**, **`Hash Join`**, **`Merge Join`**, ...
- some nodes include additional qualifying information
  - e.g. **`Filter`**, **`Index Cond`**, **`Hash Cond`**, **`Buckets`**, ...
- **`cost`** values in **`explain`** are estimates (notional units)
- **`explain analyze`** also includes actual time costs (ms)
- costs of parent nodes include costs of all children
- estimates of #results based on sample of data

<< ^ >>

❖ EXPLAIN examples (cont)

Example: Select on non-indexed attribute with actual costs

```
uni=# explain analyze
uni=# select * from Students where stype='local';
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on students
             (cost=0.00..562.01 rows=23544 width=9)
             (actual time=0.052..5.792 rows=23551 loops=1)
   Filter: ((stype)::text = 'local'::text)
   Rows Removed by Filter: 7810
 Planning time: 0.075 ms
 Execution time: 6.978 ms
```

<< ^ >>

❖ EXPLAIN examples (cont)

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=100250;
                          QUERY PLAN
-----------------------------------------------------------
 Index Scan using student_pkey on student
             (cost=0.00..8.27 rows=1 width=9)
             (actual time=0.049..0.049 rows=0 loops=1)
   Index Cond: (id = 100250)
 Planning Time: 0.274 ms
 Execution Time: 0.109 ms
```

<< ^ >>

❖ EXPLAIN examples (cont)

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=1216988;
                          QUERY PLAN
-----------------------------------------------------------
 Index Scan using students_pkey on students
             (cost=0.29..8.30 rows=1 width=9)
             (actual time=0.011..0.012 rows=1 loops=1)
   Index Cond: (id = 1216988)
 Planning time: 0.273 ms
 Execution time: 0.115 ms
```

<< ^ >>

❖ EXPLAIN examples (cont)

Example: Join on a primary key (indexed) attribute (2016)

```
uni=# explain analyze
uni-# select s.id,p.name
uni-# from Students s, People p where s.id=p.id;
                        QUERY PLAN
-----------------------------------------------------------
Hash Join (cost=988.58..3112.76 rows=31048 width=19)
          (actual time=11.504..39.478 rows=31048 loops=1)
  Hash Cond: (p.id = s.id)
  -> Seq Scan on people p
        (cost=0.00..989.97 rows=36497 width=19)
        (actual time=0.016..8.312 rows=36497 loops=1)
  -> Hash (cost=478.48..478.48 rows=31048 width=4)
          (actual time=10.532..10.532 rows=31048 loops=1)
          Buckets: 4096  Batches: 2  Memory Usage: 548kB
       ->  Seq Scan on students s
              (cost=0.00..478.48 rows=31048 width=4)
              (actual time=0.005..4.630 rows=31048 loops=1)
 Planning Time: 0.691 ms
 Execution Time: 44.842 ms
```

COMP9315 21T1 ♢ Performance Tuning ♢ [12/13]

<< ^

❖ EXPLAIN examples (cont)

Example: Join on a primary key (indexed) attribute (2018)

```
uni=# explain analyze
uni-# select s.id,p.name
uni-# from Students s, People p where s.id=p.id;
                        QUERY PLAN
-----------------------------------------------------------
Merge Join  (cost=0.58..2829.25 rows=31361 width=18)
            (actual time=0.044..25.883 rows=31361 loops=1)
  Merge Cond: (s.id = p.id)
  ->  Index Only Scan using students_pkey on students s
            (cost=0.29..995.70 rows=31361 width=4)
            (actual time=0.033..6.195 rows=31361 loops=1)
        Heap Fetches: 31361
  ->  Index Scan using people_pkey on people p
            (cost=0.29..2434.49 rows=55767 width=18)
            (actual time=0.006..6.662 rows=31361 loops=1)
Planning time: 0.259 ms
Execution time: 27.327 ms
```

COMP9315 21T1 ♢ Performance Tuning ♢ [13/13]

Produced: 6 Apr 2021