

Performance Tuning

- DB Application Performance
- Indexes
- Query Tuning
- PostgreSQL Performance Analysis
- EXPLAIN Examples

❖ DB Application Performance

In order to make DB applications efficient, it is useful to know:

- what operations on the data does the application require
(which queries, updates, inserts and how frequently is each one performed)
- how much each implementation will cost
(in terms of the amount of data transferred between memory and disk \Rightarrow time)

and then, "encourage" the DBMS to use the most efficient methods

Achieve by using indexes and avoiding certain SQL query structures

❖ DB Application Performance (cont)

Application programmer choices that affect query cost:

- how queries are expressed
 - generally join is faster than subquery
 - especially if subquery is correlated
 - filter first, then join (avoids large intermediate tables)
 - avoid applying functions in where/group-by clauses
- creating **indexes** on tables
 - index will speed-up filtering based on indexed attributes
 - indexes generally only effective for equality, gt/lt
 - mainly useful if filtering much more frequent than update

❖ DB Application Performance (cont)

Whatever you do as a DB application programmer

- the DBMS **query optimiser** will transform your query
- attempt to make it execute as efficiently as possible

You have no control over the optimisation process

- but choices you make can block certain options
- limiting the query optimiser's chance to improve

❖ DB Application Performance (cont)

Example: query to find sales people earning more than \$50K

```
select name from Employee
where salary > 50000 and
      empid in (select empid from WorksIn
                where dept = 'Sales')
```

A query evaluator might use the strategy

```
SalesEmps = (select empid from WorksIn where dept='Sales')
foreach e in Employee {
    if (e.empid in SalesEmps && e.salary > 50000)
        add e to result set
}
```

Needs to examine *all* employees, even if not in Sales

This is not a good expression of the query.

❖ DB Application Performance (cont)

A different expression of the same query:

```
select name
from   Employee join WorksIn using (empid)
where  Employee.salary > 5000 and
       WorksIn.dept = 'Sales'
```

Query evaluator might use the strategy

```
SalesEmps = (select * from WorksIn where dept='Sales')
foreach e in (Employee join SalesEmps) {
    if (e.salary > 50000)
        add e to result set
}
```

Only examines Sales employees, and uses a simpler test

This is a good expression of the query.

❖ DB Application Performance (cont)

A very poor expression of the query (correlated subquery):

```
select name from Employee e
where salary > 50000 and
      'Sales' in (select dept from WorksIn where empid=e.id)
```

A query evaluator would be forced to use the strategy:

```
foreach e in Employee {
  Depts = (select dept from WorksIn where empid=e.empid)
  if ('Sales' in Depts && e.salary > 50000)
    add e to result set
}
```

Needs to run a query for *every* employee ...

◆ Indexes

Indexes provide efficient content-based access to tuples.

Can build indexes on any (combination of) attributes.

Defining indexes:

```
CREATE INDEX name ON table ( attr1, attr2, ... )
```

*attr*_{*i*} can be an arbitrary expression (e.g. **upper(name)**).

CREATE INDEX also allows us to specify

- that the index is on **UNIQUE** values
- an access method (**USING** btree, hash, ...)

◆ Indexes (cont)

Indexes can significantly improve query costs.

Considerations in applying indexes:

- is an attribute used in frequent/expensive queries?
(note that some kinds of queries can be answered from index alone)
- should we create an index on a collection of attributes?
(yes, if the collection is used in a frequent/expensive query)
- is the table containing attribute frequently updated?
- should we use B-tree or Hash index?

```
-- use hashing for (unique) attributes in equality tests, e.g.  
select * from Employee where id = 12345  
-- use B-tree for attributes in range tests, e.g.  
select * from Employee where age > 60
```

❖ Query Tuning

Sometimes, a query can be re-phrased to affect performance:

- by helping the optimiser to make use of indexes
- by avoiding unnecessary/expensive operations

Examples which *may* prevent optimiser from using indexes:

```
select name from Employee where salary/365 > 100
    -- fix by re-phrasing condition to (salary > 36500)
select name from Employee where name like '%ith%'
select name from Employee where birthday is null
    -- above two are difficult to "fix"
select name from Employee
where dept in (select id from Dept where ...)
    -- fix by using Employee join Dept on (e.dept=d.id)
```

❖ Query Tuning (cont)

Other tricks in query tuning (effectiveness is DBMS-dependent)

- **select distinct** typically requires a sort ...
is the **distinct** really necessary? (at this stage in the query?)
- if multiple join conditions are available ...
choose join attributes that are indexed, avoid joins on strings

```
select ... Employee join Customer on (s.name = p.name)
vs
select ... Employee join Customer on (s.ssn = p.ssn)
```

- sometimes **or** prevents index from being used ...
replace the **or** condition by a union of non-**or** clauses

```
select name from Employee where Dept=1 or Dept=2
vs
(select name from Employee where Dept=1)
union
(select name from Employee where Dept=2)
```

❖ PostgreSQL Performance Analysis

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.

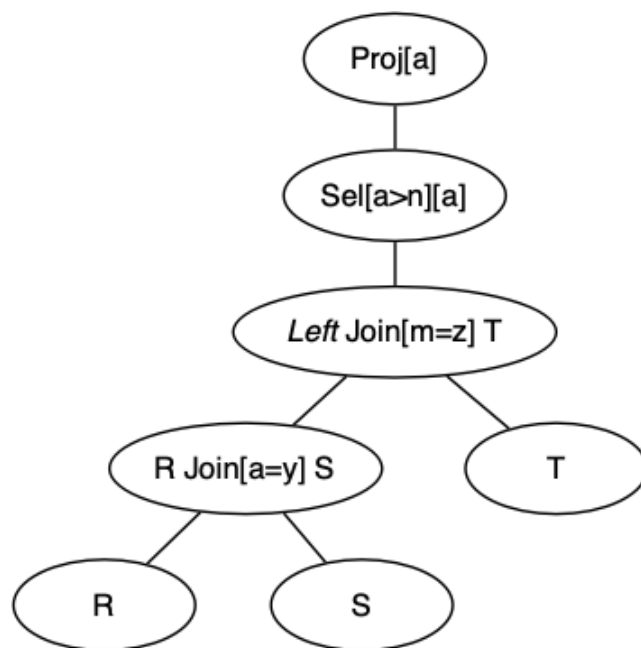
With **ANALYZE**, **EXPLAIN** executes query and prints real costs.

Note that runtimes may show considerable variation due to buffering.

If simply knowing the runtime is ok, maybe `\timing` is good enough

❖ EXPLAIN Examples

Note that PostgreSQL builds a query evaluation tree, rather than a linear plan, e.g.



```
select a
from   R
      join S on (R.a = S.y)
      join T on (T.m = S.z)
where  R.a > T.n
```

```
Tmp1 = R Join[a=y] S
Tmp2 = Tmp1 Join[m=z] T
Tmp3 = Sel[a>n](Tmp2)
Res  = Proj[a](Tmp3)
```

EXPLAIN effectively shows a pre-order traversal of the plan tree

❖ EXPLAIN Examples (cont)

Example: Select on indexed attribute

```
db=# explain analyze select * from Students where id=100250;
               QUERY PLAN
-----
Index Scan using student_pkey on student
      (cost=0.00..5.94 rows=1 width=17)
      (actual time=3.209..3.212 rows=1 loops=1)
    Index Cond: (id = 100250)
Total runtime: 3.252 ms
```

Example: Select on non-indexed attribute

```
db=# explain analyze select * from Students where stype='local';
               QUERY PLAN
-----
Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)
      (actual time=0.061..7.784 rows=2512 loops=1)
    Filter: ((stype)::text = 'local'::text)
Total runtime: 7.554 ms
```

❖ EXPLAIN Examples (cont)

Example: Join on a primary key (indexed) attribute

```
db=# explain
db=# select s.sid,p.name
db=# from Students s join People p on s.id=p.id;
```

QUERY PLAN

```
-----
Hash Join  (cost=70.33..305.86 rows=3626 width=52)
  Hash Cond: ("outer".id = "inner".id)
    -> Seq Scan on people p
          (cost=0.00..153.01 rows=3701 width=52)
    -> Hash  (cost=61.26..61.26 rows=3626 width=8)
          -> Seq Scan on student s
                (cost=0.00..61.26 rows=3626 width=8)
```

Produced: 12 Nov 2020