Lecture 20 - Query Performance Tuning - Query Functions

When to use temporary tables

1. Go from a large set of data to a small set, then apply some calculation.
2. Forcing the query planner to do a set of operations first because it is making a bad plan (try ANALYZE first and see if you can get it to make a good plan)
3. Independent data sets that then need to be aggregated.

```
SELECT CurrentInvoices.seller_no, ArchivedInvoices.sales_amt, CurrentInvoices.sales_amt
  FROM
    (
      SELECT
        seller_no,
        sum(invoice_amt) as sales_amt
      FROM invoice
      WHERE invoice_date < CURRENT_DATE
      GROUP BY
        seller_no,
        invoice_date
      ORDER BY
        seller_no,
        invoice_date
    ) as CurrentInvoices,
    (
      SELECT
        seller_no,
        sum(invoice_amt) as sales_amt
      FROM archived_invoice
      WHERE invoice_date >= date_trunc('month', CURRENT_DATE - interval '1' month)
        and invoice_date < date_trunc('month', CURRENT_DATE)
      GROUP BY
        seller_no,
        invoice_date
      ORDER BY
        seller_no,
        invoice_date
    ) as ArchivedInvoices
  WHERE CurrentInvoices.seller_no = ArchivedInvoices.seller_no
;
```

Indexes are good and bad.

Good:

1. Search's faster, sorts faster. This is the 43 min to 11 min.
2. Unique constraints. PostgreSQL requires an index for this. Oracle, DB2 can have unique constraints w/o an index. This is good for small tables - and a disaster for large ones. I don't remember if MS SQL Server or MySQL allow this.
3. Foreign Key check. Think triggers.

Bad:

1. Indexes mean that when you insert/update they have to get updates. More indexes mean slower write operations. Most d.b.'s are 90% search and 10% write. This is not true for TimeSeries data where it is 90% write and 10% search. Network database, Cassandara, Neo4j etc, are write heavy also.
2. They take space also.

Functions in where

You can use functions in the where clause. A good example is `soundex` or other data check / search functions.

```
select name
    from users
    where soundex($1) = soundex(name)
;
```

This will actually perform a full table scan and run soundex 2 times for every row. PG has no way of knowing if the function soundex is pure-functional or not!

v.s.

```
create materialized view users_soundex as
    select t1.*, soundex(name) as name_soundex
        from users as t1
;

create index users_soundex_p1 on users_soundex ( users_soundex )

select name
    from users_soundex
    where name_soundex = soundex($1)
;
```

This runs soundex(name) once for every row and saves the result during the view creation. Then once for every insert/update.

Then the query runs soundex once for every key/index value.

Side Note:

PostgreSQL can pull data directly from the index, so…

```
create materialized view users_soundex as
    select t1.name, soundex(name) as name_soundex
        from users as t1
;

create index users_soundex_p1 on users_soundex ( users_soundex, name )

select name
    from users_soundex
    where name_soundex = soundex($1)
;
```

will not access the table at all!

```
select name
    from users_soundex
    where name_soundex = soundex($1)
    order by name
;
```

will return data in sorted order for “free”.

Explain Plan

Explain plan 2 modified for printing:

```
pschlump=# \i explain-p3.sql
explain analyze
    select
        "t2"."username"
    ,    "t2"."id" as "user_id"
    ,    'root' as "user_role"
    from
        "t1_customer" as "t3" inner join "t1_user" as "t2"
            on "t3"."root_user_id" = "t2"."id"
    where
        "t3"."id" = '3f9f4a09-2d8d-414e-5afc-038ef6cec3e0'
union
    select
        "t2"."username"
    ,    "t2"."id" as "user_id"
    ,    "t4"."user_role"
    from
         "t1_customer_users" as "t4" inner join "t1_user" as "t2"
            on "t4"."user_id" = "t2"."id"
    where
        "t4"."customer_id" = '3f9f4a09-2d8d-414e-5afc-038ef6cec3e0'
order by 1 asc, 3 asc
;
                                                                  QUERY PLAN
--------------------------------------------------------------------------------
 Sort  (cost=6052.40..6052.41 rows=4 width=80) (actual time=38.323..38.324
        ^^^^^^^^^^^^^^^^^^^^^ !!!!!!
 rows=1 loops=1)
   Sort Key: t2.username, ('root'::text)
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=6052.32..6052.36 rows=4 width=80) (actual
       time=38.270..38.270 rows=1 loops=1)
        ^^^^^^^^^^^^^^^^^^
         Group Key: t2.username, t2.id, ('root'::text)
         ->  Append  (cost=136.24..6052.29 rows=4 width=80) (actual
             time=1.339..38.258 rows=1 loops=1)
               ->  Hash Join  (cost=136.24..3085.15 rows=1 width=58)
                   (actual time=1.339..38.212 rows=1 loops=1)
                     Hash Cond: (t2.id = t3.root_user_id)
                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                     ->  Seq Scan on t1_user t2  (cost=0.00..2500.02 rows=119702
                         width=26) (actual time=0.024..16.775 rows=119702
                                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^
                         loops=1)
                     ->  Hash  (cost=136.23..136.23 rows=1 width=16) (actual
                         time=1.295..1.295 rows=1 loops=1)
                                                ^^^^^^^
                           Buckets: 1024  Batches: 1  Memory Usage: 9kB
                           ->  Seq Scan on t1_customer t3  (cost=0.00..136.23
                               rows=1 width=16) (actual time=0.018..1.287
                               rows=1 loops=1)
                                 Filter: (id =
                                 '3f9f4a09-2d8d-414e-5afc-038ef6cec3e0'::uuid)
                                 Rows Removed by Filter: 5937
                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
               ->  Hash Join  (cost=18.16..2967.10 rows=3 width=58) (actual
                   time=0.044..0.044 rows=0 loops=1)
                     Hash Cond: (t2_1.id = t4.user_id)
                     ->  Seq Scan on t1_user t2_1  (cost=0.00..2500.02
                        ^^^^^^^^^^   ^^^^^^^^^^^^
                         rows=119702 width=26) (actual
                         time=0.010..0.010 rows=1 loops=1)
                     ->  Hash  (cost=18.12..18.12 rows=3 width=48) (actual
                         time=0.003..0.003 rows=0 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 8kB
                           ->  Seq Scan on t1_customer_users t4
                               (cost=0.00..18.12 rows=3 width=48) (actual
                               time=0.002..0.002 rows=0 loops=1)
                                 Filter: (customer_id =
                                 '3f9f4a09-2d8d-414e-5afc-038ef6cec3e0'::uuid)
 Planning time: 1.158 ms
 Execution time: 38.521 ms
(23 rows)
```

Explain works on any DML

… so not just selects. It works on UPDARE, DELERTE etc.

EXPLAIN v.s. EXPALIN ANALYZE

ANALYZE takes a bunch more time - but uses staticics from the last time you did an ANALYZE on the databse. Also EXPLAIN ANALYZE may run the query to get actual times. So if the query takes 43 minutes you should go for coffee.

`(cost=6052.40..6052.41` and Nodes. Each block is a "processing node" and the time it reports is the time that that node took plus the sum of all child nodes. At the top is the total cost for running the query. This lets you isolate the chunks of the query that take the most time.

6052.40 - first number is startup cost. This is cost to get to 1st row.

6052.41 - time to process entire set.

`(cost 1.339..38.258 rows=1 loops=1)` better example 1.339 for first, 38.258 for all rows. `(cost` is an *estimate* based on analyze data. DBs use a "cost" minimize approach - so with different possibilities it will pick the sub-nodes in the tree that give it the least cost. This also means that if you load 10,000,000 new rows to ta table that use to have 10 rows and you don't do a new ANALYZE on the database then the cost values are way way out of wack and the plans generated will be horrid.

`(actual time=0.024..16.775 rows=119702` this is initial/total time with data for actually running the query. The units are in miliseconds.

`... loops=1)` this shows where it is actually looping over something in the plan.

`Rows Removed by Filter: 5937` how many rows were eliminated.

`Seq Scan on t1_user t2_1` This is how it is accessing the data, and what table was used and any indexes used.

Let's start simple and work up:

```
create table exp_1 (
    i int
);

EXPLAIN
    SELECT *
    FROM exp_1
    WHERE i = 0;
```

```
create table exp_1 (
    i int
);
CREATE TABLE

EXPLAIN
    SELECT *
    FROM exp_1
    WHERE i = 0;
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on exp_1  (cost=0.00..41.88 rows=13 width=4)
   Filter: (i = 0)
(2 rows)
```

or

```
EXPLAIN ANALYZE
    SELECT *
    FROM exp_1
    WHERE i = 0;
                                           QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on exp_1  (cost=0.00..41.88 rows=13 width=4) (actual time=0.003..0.003
   rows=0 loops=1)
   Filter: (i = 0)
 Planning time: 0.059 ms
 Execution time: 0.034 ms
(4 rows)
```

`Seq Scan on <Table>` means full table scan.

Now a table with a primary key:

```
CREATE SEQUENCE exp_3_id_seq
  INCREMENT 1
  MINVALUE 1
```

```
    MAXVALUE 9223372036854775807
    START 1
    CACHE 1;

  create table exp_3 (
      id            bigint DEFAULT nextval('exp_3_id_seq'::regclass) NOT NULL primary key
      user_data     bigint
  );

  explain analyze
      select *
      from exp_3
      where id = 55
  ;

  explain analyze
      select *
      from exp_3
      where user_data = 55
  ;
```

output:

```
  ...

  explain analyze
      select *
      from exp_3
      where id = 55
  ;
                                                      QUERY PLAN
  -------------------------------------------------------------------------
   Index Scan using exp_3_pkey on exp_3  (cost=0.15..8.17 rows=1 width=16)
   (actual time=0.021..0.021 rows=0 loops=1)
     Index Cond: (id = 55)
   Planning time: 0.217 ms
   Execution time: 0.047 ms
  (4 rows)

  explain analyze
      select *
      from exp_3
      where user_data = 55
  ;
                                             QUERY PLAN
  --------------------------------------------------------------------------------
   Seq Scan on exp_3  (cost=0.00..33.12 rows=9 width=16) (actual time=0.002..0.002
   rows=0 loops=1)
     Filter: (user_data = 55)
   Planning time: 0.043 ms
   Execution time: 0.017 ms
  (4 rows)
```

You can tune the “costs” that it uses - this is set in potgresql.conf. In Oracle it is in `<database>_init.ora` , in DB2 in `<instanceName>_setup` . In MySQL… and in SQL Server… hm… Nobody knowns what nobody knowns.

postgresql.conf, these params:

```
seq_page_cost = 1.0                     # measured ON an arbitrary scale
random_page_cost = 4.0                  # same scale AS above
cpu_tuple_cost = 0.01                   # same scale AS above
cpu_index_tuple_cost = 0.005            # same scale AS above
cpu_operator_cost = 0.0025              # same scale AS above
```

Postgres lacks the ability to set this on a per-volume baseis - so you may want to set `random_page_cost = 1.0` on SSD and 4.0 is low for a real disk - it is more in the range of 16 to 50.

Let’s add an index to exp_3 and see the result:

```
create index exp_3_p1 on exp_3 ( user_data );

explain analyze
    select *
    from exp_3
    where user_data = 55
;
```

output:

```
create index exp_3_p1 on exp_3 ( user_data );
CREATE INDEX
explain analyze
    select *
    from exp_3
    where user_data = 55
;
                                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on exp_3  (cost=4.22..14.76 rows=9 width=16) (actual
 time=0.031..0.031 rows=0 loops=1)
   Recheck Cond: (user_data = 55)
```

```
     ->  Bitmap Index Scan on exp_3_p1  (cost=0.00..4.22 rows=9 width=0) (actual
         time=0.003..0.003 rows=0 loops=1)
           Index Cond: (user_data = 55)
 Planning time: 0.259 ms
 Execution time: 0.666 ms
(6 rows)
```

Copyright