Identifying Slow Queries, and Fixing Them

Presented to: PostgresOpen 2011, Chicago
Date: September 15th, 2011

noblis™
For the best of reasons

Introduction

- Stephen Frost
  - System Architect/Designer
  - DBA, Unix Administrator
  - PostgreSQL/PostGIS Hacker
  - Added Roles in 8.1, Column-level Privs in 8.4
- Noblis, Inc.
  - Nonprofit science, technology and strategy organization
  - http://www.noblis.org

Finding the slow ones

(Queries...)

Monitor your systems!

- PostgreSQL Logs
  - Configure what gets logged!
  - Log checkpoints, connections, DDL statements! Perhaps more..
- Your favorite monitoring solution
  - Availability, Alarm-based (eg: Nagios, w/ check_posgres)
  - Performance measuring (eg: munin, stats w/ pg_bouncer)
  - PgFouine for log file analysis
- check_postgres script
  - Includes lots of valuable checks
  - Bloat checking
  - Idle connection warnings
  - Number of WAL files (in case archiving fails)
  - Can integrate w/ munin/cacti/MRTG too!

Finding Slow Queries

- postgresql.conf
  - log_min_duration_statement – just needs reload
  - Lots of other logging options:
    - log_line_prefix
    - log_connections / log_disconnections
    - log_duration
    - log_lock_waits
    - log_statement
    - track_functions
- Reviewing PG logs
  - LOG: duration: 448.495 ms statement: select generate_series(1,1000000);
  - What's in that duration?
  - Difference with psql's \timing option

PG duration logging

- More PG logs
  - Just log_min_duration_statement:
    - LOG: duration: 448.495 ms statement: select generate_series(1,1000000);
  - vs. log_statement = all && log_min_duration_statement:
    - LOG: statement: select generate_series(1,1000000);
    - LOG: duration: 513.041 ms
  - vs. log_statement = none && log_min_duration_statement && log_duration:
    - LOG: duration: 0.659 ms
    - LOG: duration: 457.366 ms statement: select generate_series(1,1000000);
  - If you can afford log_statements=all and log_duration you can gather lots of info, but it's not free to log at that level (typically not done in high-transaction production systems)
  - log_min_duration_statement gives 'best of both worlds'- just log the slow ones, but be careful what other options you have enabled or it may get confusing
  - Lots of fast queries, done sequentially, can also make things (page loads) slow!

Now we've found them …

Why are they slow?

Understanding why queries are slow

- The “easy” stuff-
  - Poor PG configuration
  - Dead tuples / bloat
- The next level- Database Magic

Poor PG Configuration (you used the defaults...)

- Important PG GUCs (configuration options):
  - work_mem
  - maintenance_work_mem
  - effective_cache_size
  - shared_buffers
  - checkpoint_segments
- Watch for differences between Prod & Dev
  - Need to understand them, if any
  - May get different plans if different
  - "Unseen" differences
    - Statistics data may be different
    - Different hardware
    - Warm-up Time

Dead Tuples / Bloat

- VACUUM marks records reusable, if possible
  - Reusable tuples will be used for new inserts, etc
  - However, PG has to handle those tuples on queries
- Records marked as deleted but not reusable yet
  - Ongoing transactions
- Bloat can exist in both tables and indexes
- check_postgres.pl
  - Can identify bloat in tables/indexes
  - **Some** bloat is **GOOD**, but too much will make queries slower (lots of extra/unnecessary data to process)
- CLUSTER will re-write a table and eliminate dead tuples.

Database Magic, or how it works

- There is no magic here, sadly.
- Getting data:
  - Sequentially step through **EVERY** record
    - SeqScan Node
    - Bulk, very fast at going through a table
  - Pick out **SPECIFIC** records, using an index
    - Index Scan Node
    - Very slow for bulk data
    - Can return data in-order
    - Index needs to be there..

More Magic

- Putting things together (Joins)
  - Loop through and scan table for match
    - Nested Loop Node
    - Works for **small** data sets
  - Order two tables, then walk through each Merging them
    - Merge Join Node
    - Requires sorted inputs
    - Good for bulk operations, esp. work loads that won't fit in memory
  - Build a hash table (of the smaller table) then step through
    - Hash Join Node
    - Requires lots of memory
    - Very fast, but slow to start
- Adding it all up (Aggregates)
  - Look at all rows that qualify
  - Can be very expensive

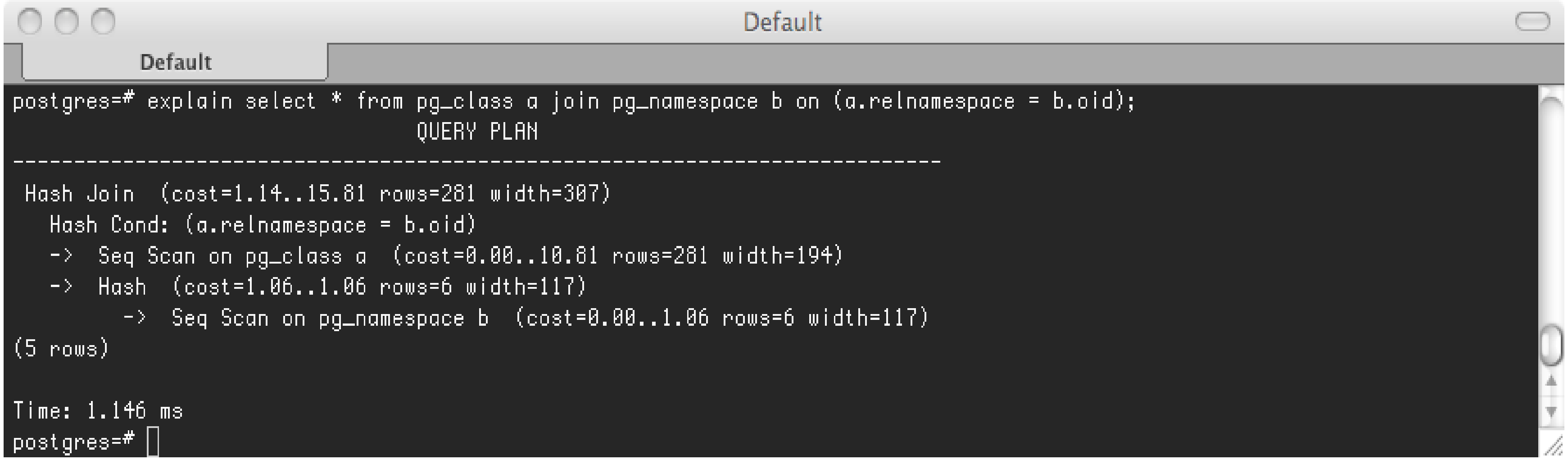
What's the best plan?

- It depends!
- How's the database know?
  - Gathers statistics using ANALYZE
  - Automatically done by auto-vacuum
- What if the database (aka- the stats) are wrong?
  - You get bad plans!
  - Look for differences in row estimates from explain analyze:
  - Index Scan using my_idx on my_table (cost=0.00..5719.56 rows=9055 width=10) (actual time=0.015..87.689 rows=163491 loops=1)
  - May need to adjust statistics target

What plan did PG decide to use?!

Understanding “explain”

- explain output:

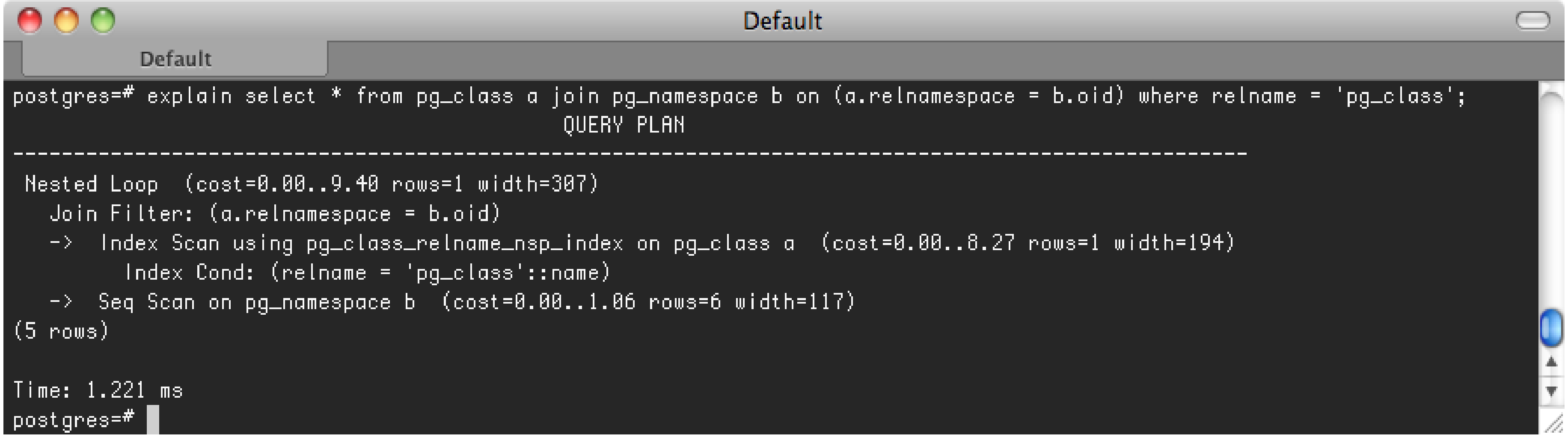


```
postgres=# explain select * from pg_class a join pg_namespace b on (a.relnamespace = b.oid);
                                QUERY PLAN
---------------------------------------------------------------------------
 Hash Join  (cost=1.14..15.81 rows=281 width=307)
   Hash Cond: (a.relnamespace = b.oid)
   ->  Seq Scan on pg_class a  (cost=0.00..10.81 rows=281 width=194)
   ->  Hash  (cost=1.06..1.06 rows=6 width=117)
         ->  Seq Scan on pg_namespace b  (cost=0.00..1.06 rows=6 width=117)
(5 rows)

Time: 1.146 ms
postgres=# 
```

- Node types: Hash, Hash Join, Seq Scan
- Lots of other node types
- What is the cost?

More Explain

- With “where”

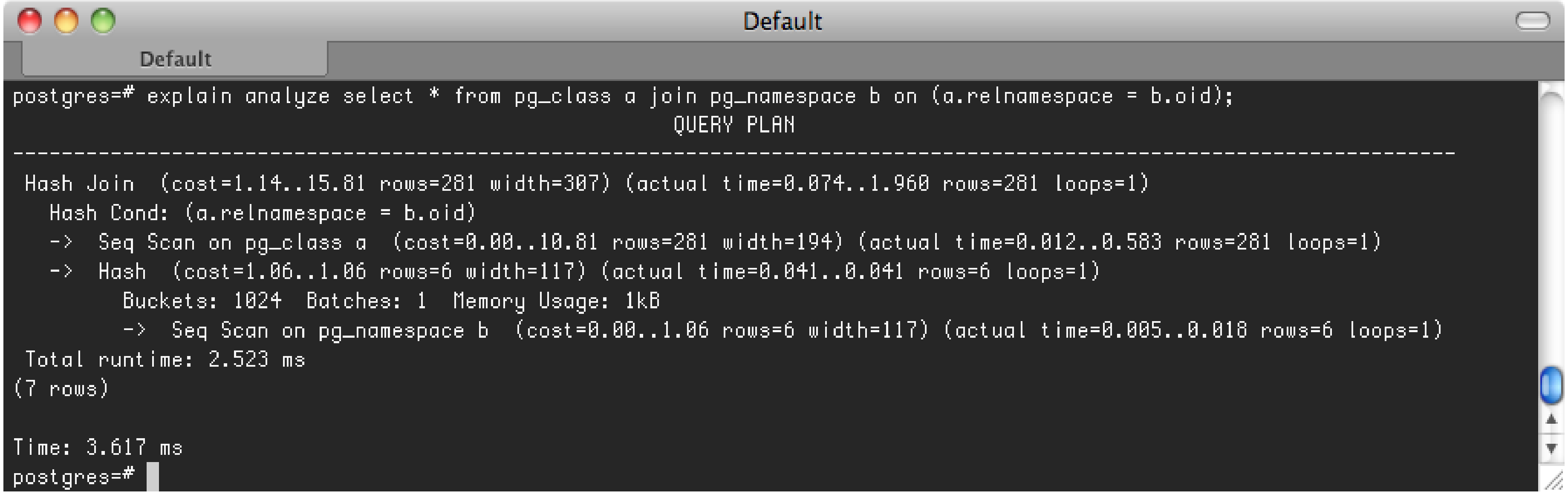


```
postgres=# explain select * from pg_class a join pg_namespace b on (a.relnamespace = b.oid) where relname = 'pg_class';
                                             QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..9.40 rows=1 width=307)
   Join Filter: (a.relnamespace = b.oid)
   ->  Index Scan using pg_class_relname_nsp_index on pg_class a  (cost=0.00..8.27 rows=1 width=194)
         Index Cond: (relname = 'pg_class'::name)
   ->  Seq Scan on pg_namespace b  (cost=0.00..1.06 rows=6 width=117)
(5 rows)

Time: 1.221 ms
postgres=# 
```

- Very different plan!
- More nodes: Nested Loop, Index Scan
- Lower cost, much fewer rows

Understanding “explain analyze”

- explain analyze output

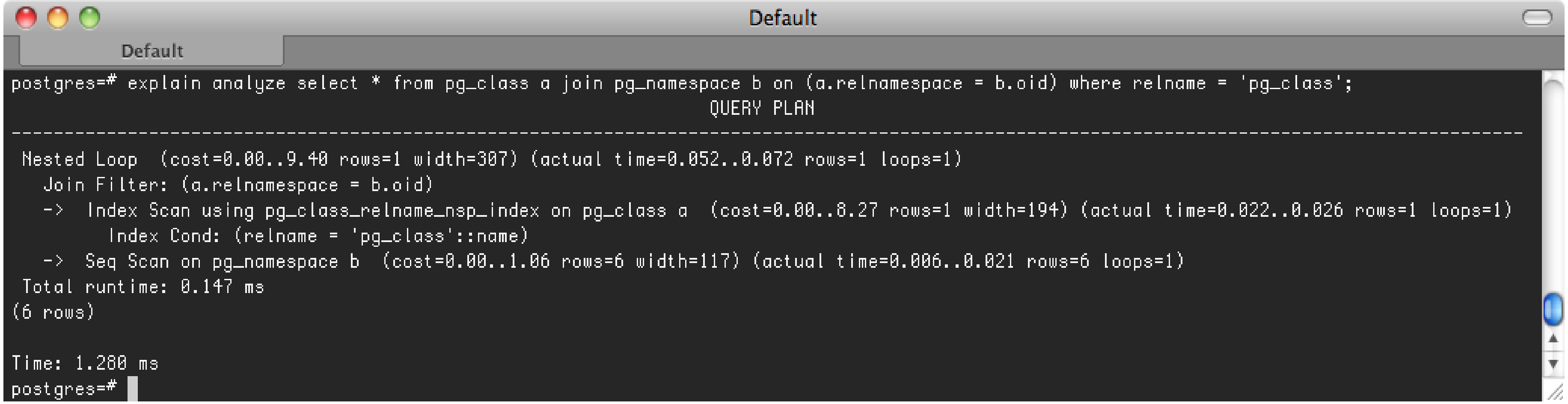


```
postgres=# explain analyze select * from pg_class a join pg_namespace b on (a.relnamespace = b.oid);
                                                   QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=1.14..15.81 rows=281 width=307) (actual time=0.074..1.960 rows=281 loops=1)
   Hash Cond: (a.relnamespace = b.oid)
   ->  Seq Scan on pg_class a  (cost=0.00..10.81 rows=281 width=194) (actual time=0.012..0.583 rows=281 loops=1)
   ->  Hash  (cost=1.06..1.06 rows=6 width=117) (actual time=0.041..0.041 rows=6 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Seq Scan on pg_namespace b  (cost=0.00..1.06 rows=6 width=117) (actual time=0.005..0.018 rows=6 loops=1)
 Total runtime: 2.523 ms
(7 rows)

Time: 3.617 ms
postgres=#
```

- Lots more info- **actual** times, per-node info, memory usage!
- Two times? backend runtime, psql timing

More explain analyze

- Explain analyze with where output



```
postgres=# explain analyze select * from pg_class a join pg_namespace b on (a.relnamespace = b.oid) where relname = 'pg_class';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..9.40 rows=1 width=307) (actual time=0.052..0.072 rows=1 loops=1)
   Join Filter: (a.relnamespace = b.oid)
   ->  Index Scan using pg_class_relname_nsp_index on pg_class a  (cost=0.00..8.27 rows=1 width=194) (actual time=0.022..0.026 rows=1 loops=1)
         Index Cond: (relname = 'pg_class'::name)
   ->  Seq Scan on pg_namespace b  (cost=0.00..1.06 rows=6 width=117) (actual time=0.006..0.021 rows=6 loops=1)
 Total runtime: 0.147 ms
(6 rows)

Time: 1.280 ms
postgres=# 
```

- Back to the other plan, with actuals, total runtime
- Still a seqscan on pg_namespace..?

Other explain output options

- Other output options
  - XML, JSON, YAML
- Tools to analyze explain output
  - PgAdmin3
  - explain.depesz.com
- PG Log file analyzer, includes tracking timing info
  - pgFouine

Automating collection of “explain”s

- auto_explain
  - Logs explain info for long queries
- Enabling:
  - shared_preload_libraries = 'auto_explain'
  - set explain.log_min_duration = 50;
  - Can also output 'explain analyze' (expensive!)
  - Set explain.log_nested_statements = true;
    - Considers logging plans which are inside functions

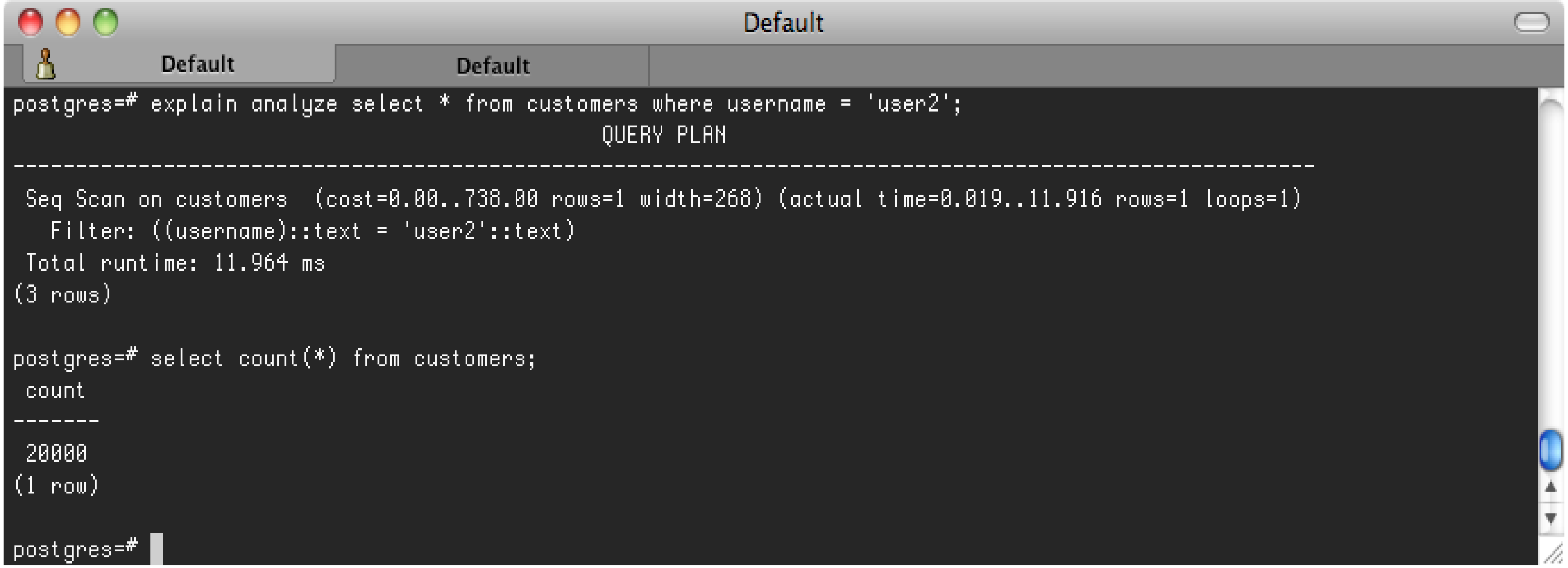
But how to change the plan?
How to fix the queries..?!

Fixing Them

- Low-hanging fruit, but catch a lot..
  - Query returning 1 row using SeqScan? - Check for an index
  - MergeJoin used for “small” data sets? - Check work_mem
  - Nested Loop used with large set? - Bad row estimates?
    - Make sure analyze is being done
    - Increase statistics target for the relations if possible
  - DELETE's slow? Make sure you have indexes on Foreign Keys
- Harder items:
  - Check over your long-running queries
  - Use stored procedures / triggers
  - Partitioning large tables
  - Consider Partial Indexes / Functional Indexes

SeqScan returns 1 row

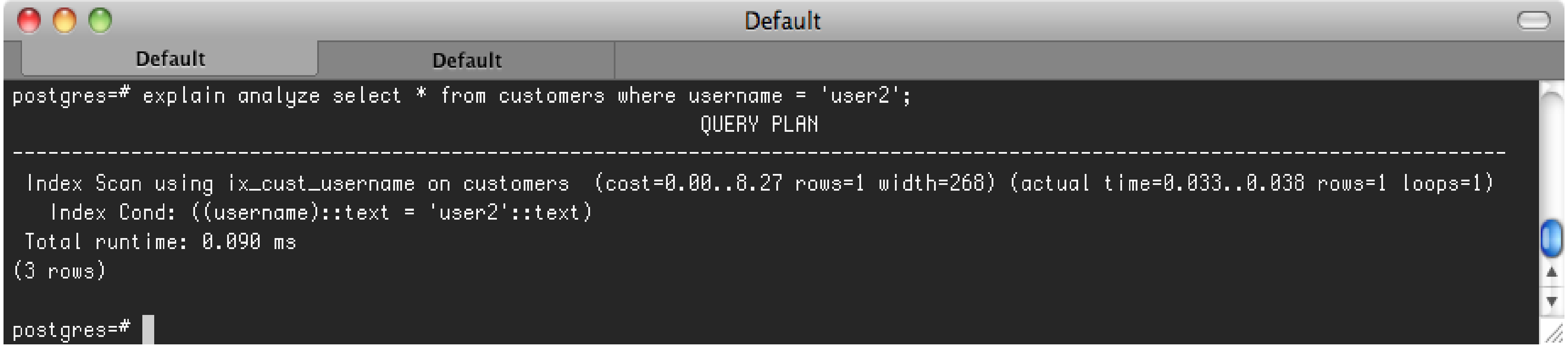
- Lack of index on username



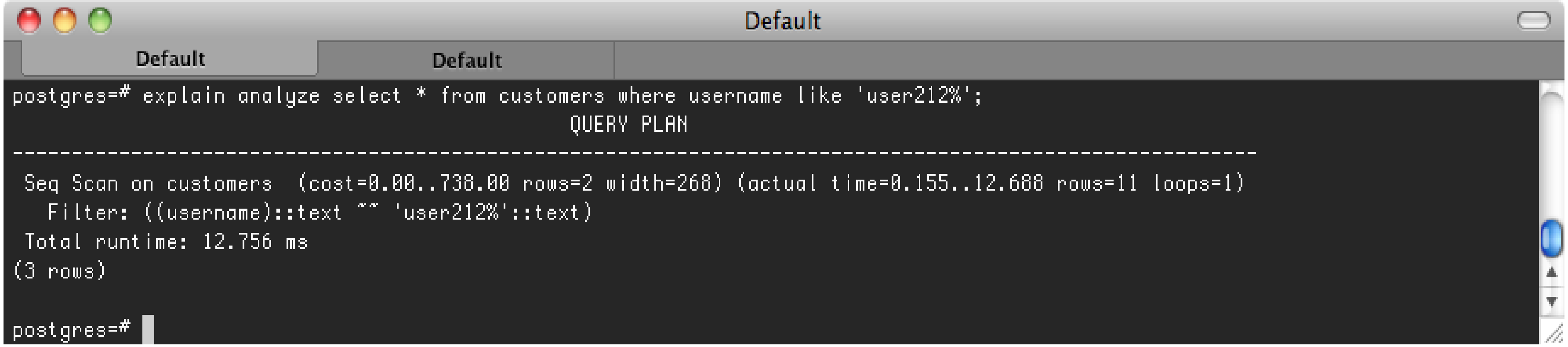
```
postgres=# explain analyze select * from customers where username = 'user2';
                                               QUERY PLAN
---------------------------------------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..738.00 rows=1 width=268) (actual time=0.019..11.916 rows=1 loops=1)
   Filter: ((username)::text = 'user2'::text)
 Total runtime: 11.964 ms
(3 rows)

postgres=# select count(*) from customers;
 count
-------
 20000
(1 row)

postgres=# 
```

Using an index scan

- Much better performing, what about like?



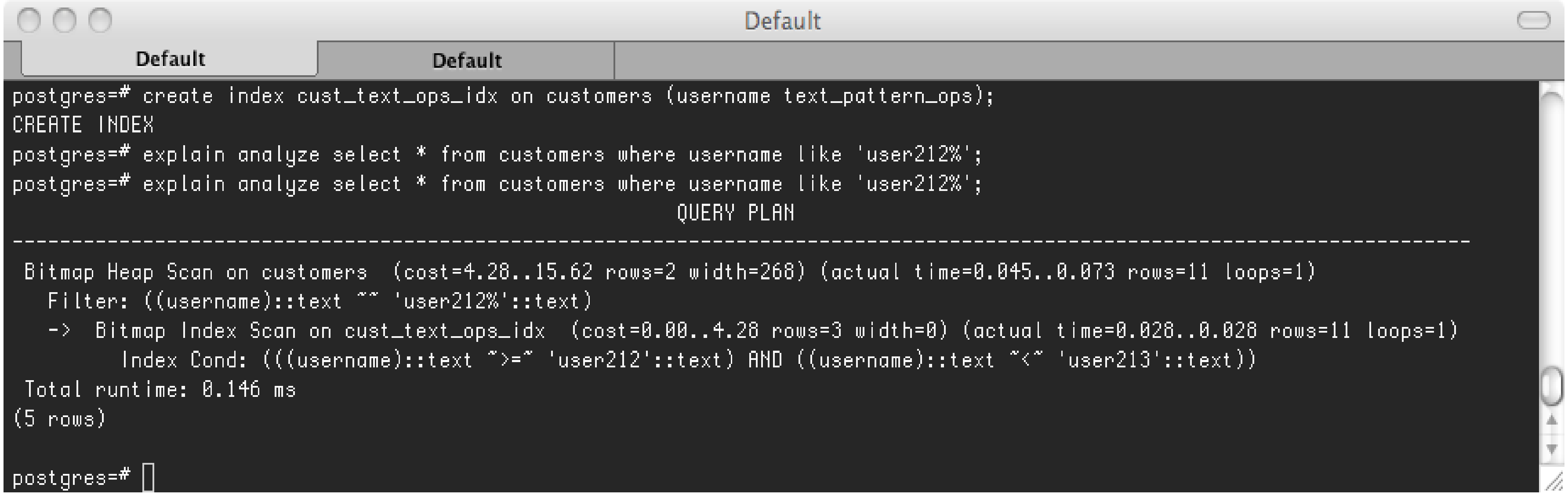
```
postgres=# explain analyze select * from customers where username = 'user2';
                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Index Scan using ix_cust_username on customers  (cost=0.00..8.27 rows=1 width=268) (actual time=0.033..0.038 rows=1 loops=1)
   Index Cond: ((username)::text = 'user2'::text)
 Total runtime: 0.090 ms
(3 rows)

postgres=# 
```



```
postgres=# explain analyze select * from customers where username like 'user212%';
                                                 QUERY PLAN
------------------------------------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..738.00 rows=2 width=268) (actual time=0.155..12.688 rows=11 loops=1)
   Filter: ((username)::text ~~ 'user212%'::text)
 Total runtime: 12.756 ms
(3 rows)

postgres=# 
```

Text Pattern Searches

- Need an appropriate index

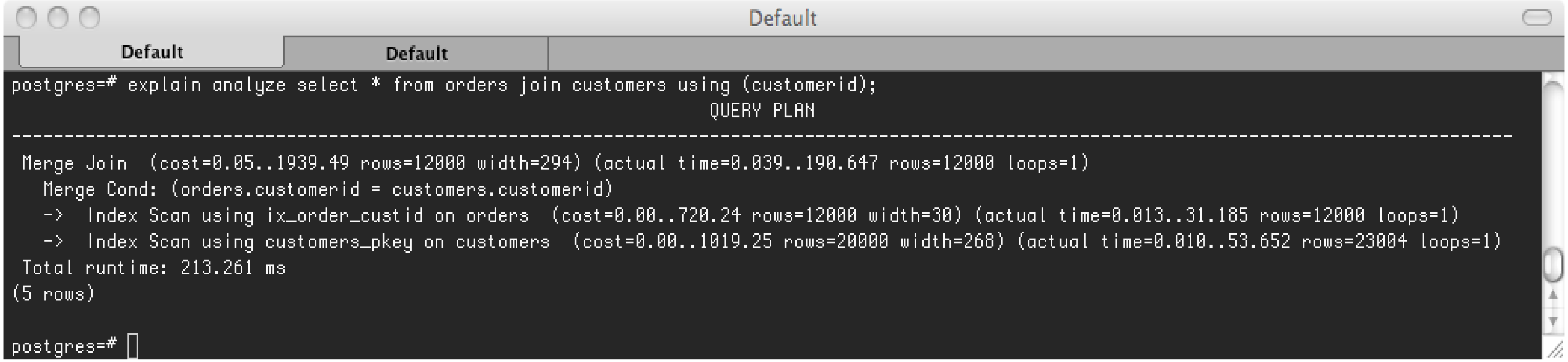


```
postgres=# create index cust_text_ops_idx on customers (username text_pattern_ops);
CREATE INDEX
postgres=# explain analyze select * from customers where username like 'user212%';
postgres=# explain analyze select * from customers where username like 'user212%';
                                                         QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on customers  (cost=4.28..15.62 rows=2 width=268) (actual time=0.045..0.073 rows=11 loops=1)
   Filter: ((username)::text ~~ 'user212%'::text)
   ->  Bitmap Index Scan on cust_text_ops_idx  (cost=0.00..4.28 rows=3 width=0) (actual time=0.028..0.028 rows=11 loops=1)
         Index Cond: (((username)::text ~>=~ 'user212'::text) AND ((username)::text ~<~ 'user213'::text))
 Total runtime: 0.146 ms
(5 rows)

postgres=# 
```

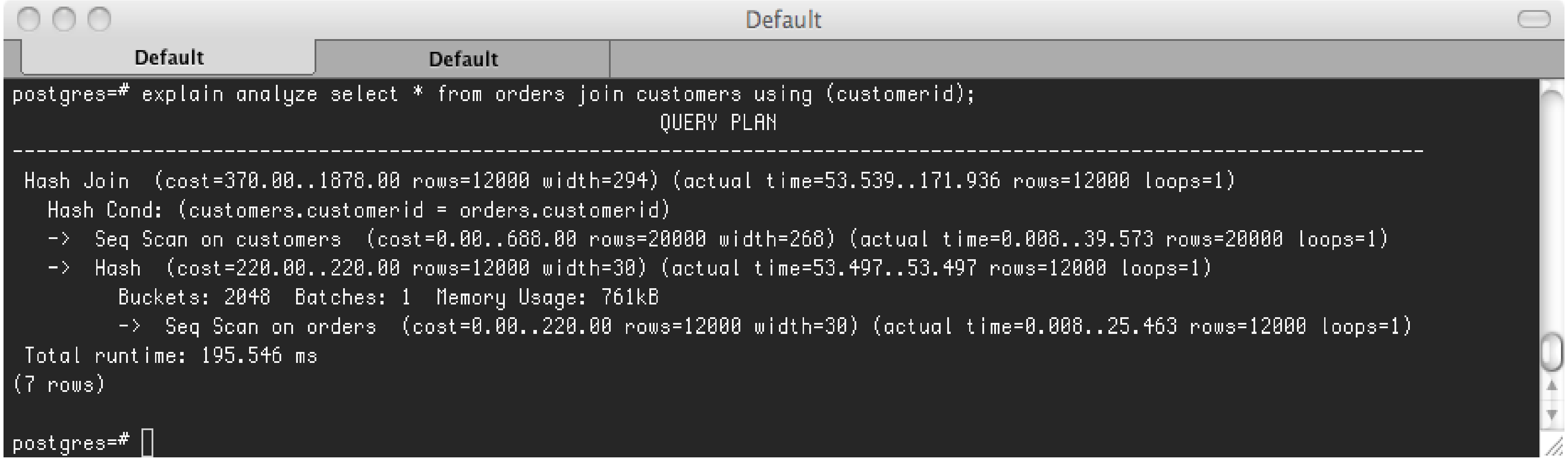
- Pattern needs to be anchored and simple
- PG has excellent Full Text Search & Indexing

MergeJoin for 'small' data

- Merge vs Hash and work_mem



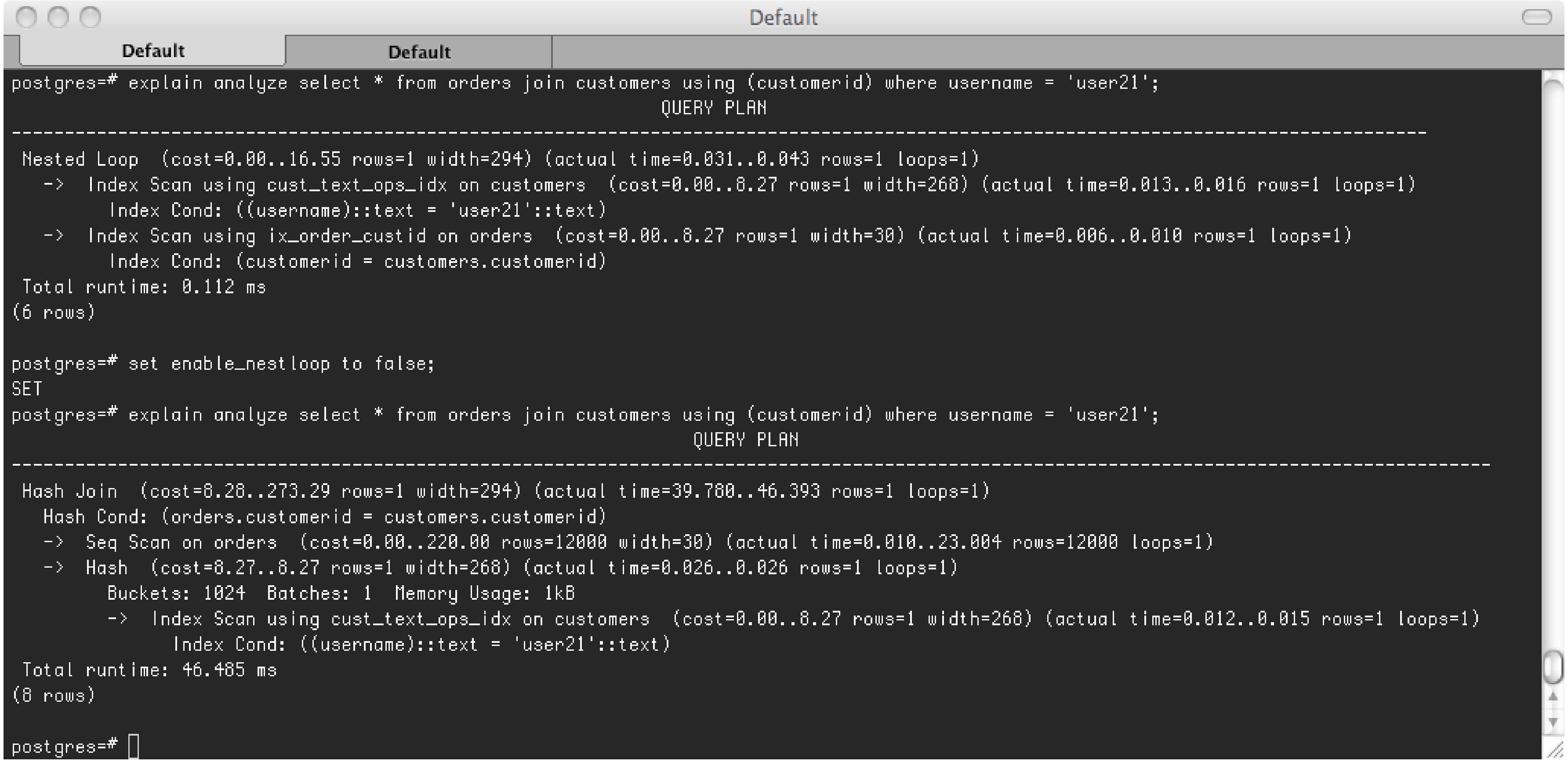
```
postgres=# explain analyze select * from orders join customers using (customerid);
                                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=0.05..1939.49 rows=12000 width=294) (actual time=0.039..190.647 rows=12000 loops=1)
   Merge Cond: (orders.customerid = customers.customerid)
   ->  Index Scan using ix_order_custid on orders  (cost=0.00..720.24 rows=12000 width=30) (actual time=0.013..31.185 rows=12000 loops=1)
   ->  Index Scan using customers_pkey on customers  (cost=0.00..1019.25 rows=20000 width=268) (actual time=0.010..53.652 rows=23004 loops=1)
 Total runtime: 213.261 ms
(5 rows)

postgres=# 
```



```
postgres=# explain analyze select * from orders join customers using (customerid);
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=370.00..1878.00 rows=12000 width=294) (actual time=53.539..171.936 rows=12000 loops=1)
   Hash Cond: (customers.customerid = orders.customerid)
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=268) (actual time=0.008..39.573 rows=20000 loops=1)
   ->  Hash  (cost=220.00..220.00 rows=12000 width=30) (actual time=53.497..53.497 rows=12000 loops=1)
         Buckets: 2048  Batches: 1  Memory Usage: 761kB
         ->  Seq Scan on orders  (cost=0.00..220.00 rows=12000 width=30) (actual time=0.008..25.463 rows=12000 loops=1)
 Total runtime: 195.546 ms
(7 rows)

postgres=# 
```

Nest Loops can be good

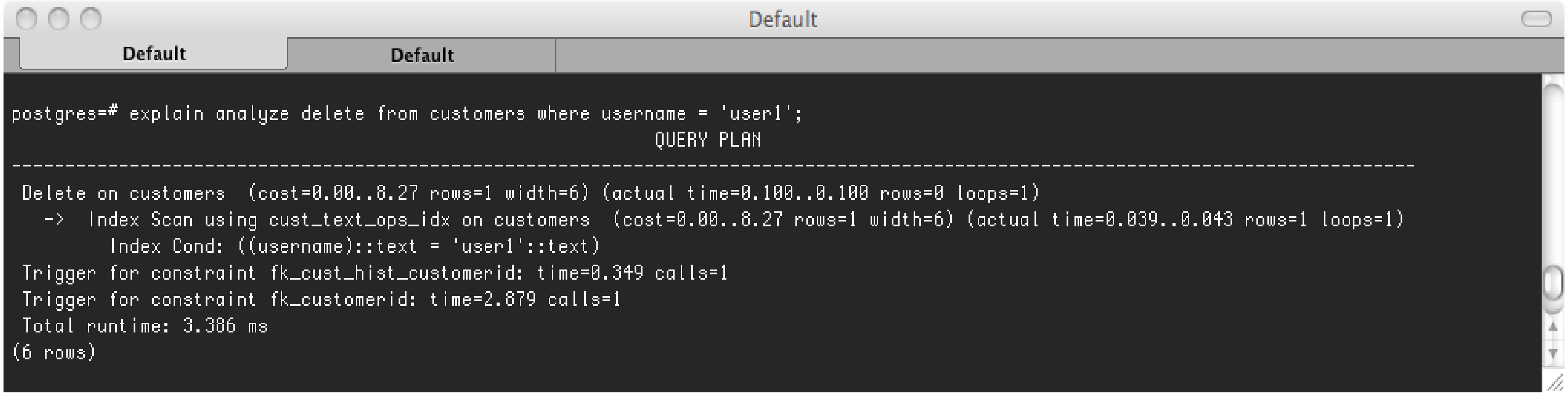
- For small sets



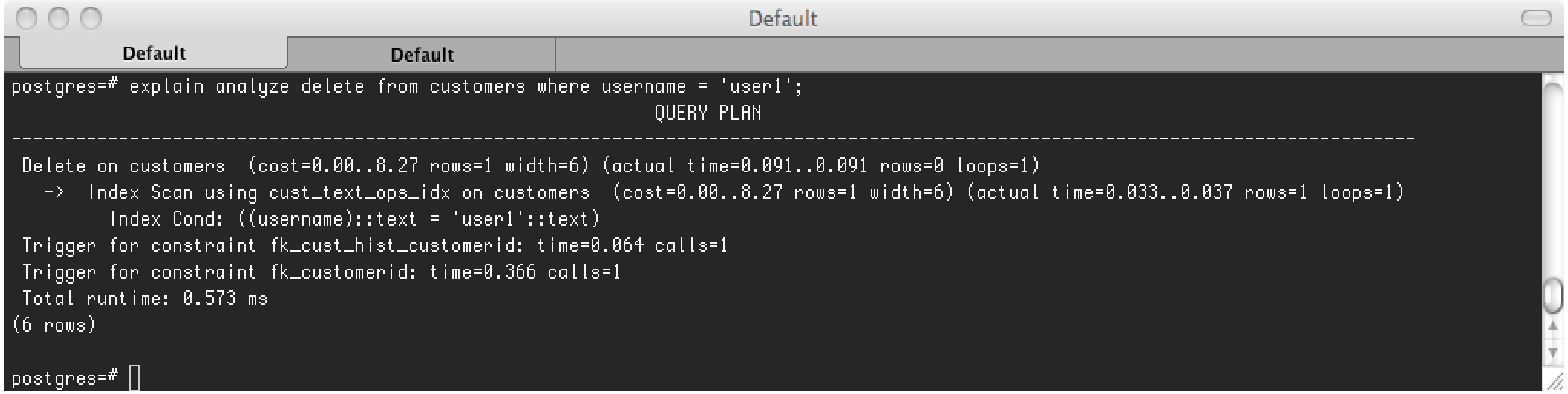
```
postgres=# explain analyze select * from orders join customers using (customerid) where username = 'user21';
                                                             QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..16.55 rows=1 width=294) (actual time=0.031..0.043 rows=1 loops=1)
   ->  Index Scan using cust_text_ops_idx on customers  (cost=0.00..8.27 rows=1 width=268) (actual time=0.013..0.016 rows=1 loops=1)
         Index Cond: ((username)::text = 'user21'::text)
   ->  Index Scan using ix_order_custid on orders  (cost=0.00..8.27 rows=1 width=30) (actual time=0.006..0.010 rows=1 loops=1)
         Index Cond: (customerid = customers.customerid)
 Total runtime: 0.112 ms
(6 rows)

postgres=# set enable_nestloop to false;
SET
postgres=# explain analyze select * from orders join customers using (customerid) where username = 'user21';
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=8.28..273.29 rows=1 width=294) (actual time=39.780..46.393 rows=1 loops=1)
   Hash Cond: (orders.customerid = customers.customerid)
   ->  Seq Scan on orders  (cost=0.00..220.00 rows=12000 width=30) (actual time=0.010..23.004 rows=12000 loops=1)
   ->  Hash  (cost=8.27..8.27 rows=1 width=268) (actual time=0.026..0.026 rows=1 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         ->  Index Scan using cust_text_ops_idx on customers  (cost=0.00..8.27 rows=1 width=268) (actual time=0.012..0.015 rows=1 loops=1)
               Index Cond: ((username)::text = 'user21'::text)
 Total runtime: 46.485 ms
(8 rows)

postgres=# 
```

Slow DELETE

- Explain analyze on delete, what's the difference?



```
postgres=# explain analyze delete from customers where username = 'user1';
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Delete on customers  (cost=0.00..8.27 rows=1 width=6) (actual time=0.100..0.100 rows=0 loops=1)
   ->  Index Scan using cust_text_ops_idx on customers  (cost=0.00..8.27 rows=1 width=6) (actual time=0.039..0.043 rows=1 loops=1)
         Index Cond: ((username)::text = 'user1'::text)
 Trigger for constraint fk_cust_hist_customerid: time=0.349 calls=1
 Trigger for constraint fk_customerid: time=2.879 calls=1
 Total runtime: 3.386 ms
(6 rows)
```



```
postgres=# explain analyze delete from customers where username = 'user1';
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Delete on customers  (cost=0.00..8.27 rows=1 width=6) (actual time=0.091..0.091 rows=0 loops=1)
   ->  Index Scan using cust_text_ops_idx on customers  (cost=0.00..8.27 rows=1 width=6) (actual time=0.033..0.037 rows=1 loops=1)
         Index Cond: ((username)::text = 'user1'::text)
 Trigger for constraint fk_cust_hist_customerid: time=0.064 calls=1
 Trigger for constraint fk_customerid: time=0.366 calls=1
 Total runtime: 0.573 ms
(6 rows)

postgres=# 
```

Prepared queries

- They're good, honest
- Plan once, run many
  - Not as much info to plan with, plans may be more stable
  - Variables aren't substituted in until execution
  - No constraint exclusion though
- How to explain/explain analyze:
  - prepare q as select * from table where x = $1;
  - explain execute q('myid');
  - explain analyze execute q('myid');
- Placeholders in explain output ($1 instead of 'myid')

Query Review

- select count(*) from table;
  - Expensive, must check every record in the table
- select * from table;
  - Returns every row, do you really need them all?
  - Order By / Limit can help PG optimize queries!
- select * from table where id = 1;
  - Do you need every column? Wide columns cost / TOAST
- select * from a, b, c where a.x = b.x;
  - Missing join condition for c!
  - Cartesian product with a/b to c
  - Use join syntax:
    - select * from a join b using (x) join c using (x);

More Queries

- select * from x where myid in
  (select myid from big_table);
  - Turn it into a join:
    - select x.* from x join big_table using (myid);
- select * from x where myid not in
  (select myid from big_table);
  - Left-join instead:
    - select x.* from x left join big_table using (myid)
      where big_table.myid is NULL;
  - Not exists also:
    - select * from x where not exists
      (select * from big_table where big_table.myid = x.myid)

More queries...

- Expensive to generate table? Use CTE (Common Table Expressions, aka WITH)

-
  ```
  select *,
      (select sum(my_expensive_view.x) from my_expensive_view)
      from my_expensive_view;
  ```

-
  ```
  WITH my_view AS (select * from my_expensive_view),
      my_sums AS (select sum(my_view.x))
      select my_view.*, my_sums.sum from my_view, my_sums;
  ```
- CTEs can also be used to implement recursion!

Really need fast count(*)?

- Does it have to be accurate or just an estimation?
  - Look at pg_class.reltuples for an estimate
- Use a trigger if it needs to be accurate
  - Handle bulk-loading independently though
- It's a trade-off
  - Faster to get count(*) information
  - Slower to insert/update the table

-
  ```
  create function my_count_func() returns trigger as $_$
          BEGIN
                  UPDATE my_count = my_count + 1;
                  RETURN NEW;
          END $_$ LANGUAGE 'plpgsql';
  ```

-
  ```
  create trigger my_count_trig after insert on my_table for each row execute procedure my_count_func();
  ```

What else can be done?

- Tuning PG GUCs
  - work_mem – default 1MB is **wayyy** small
  - maintenance_work_mem – default 16MB small
  - effective_cache_size – default 128MB
    - You have a server with 256MB of memory..?
  - shared_buffers – default 24MB
    - This is a real killer.. bump it to 1-2G, at least, on a server w/ >4G RAM, up to 8GB (don't go above that w/o good testing..).
- Partial Indexes / Functional Indexes
- Improving statistics / analyze / auto-vacuum
- Tuning the background writer
  - Consider making it more aggressive for heavy write loads
- Invest in hardware
  - Lots of memory (adjust shared_buffers..)
  - SSDs / Battery-Backed Write Cache RAID

Questions?