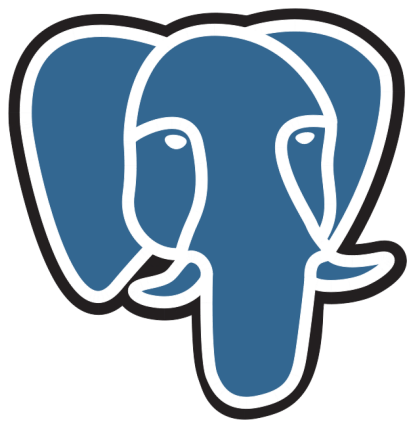
Čteme EXPLAIN

CSPUG, Praha

Tomáš Vondra (tv@fuzzy.cz)

Czech and Slovak PostgreSQL Users Group



21.6.2011

Agenda

- K čemu slouží EXPLAIN a EXPLAIN ANALYZE?
- Jak funguje plánování, jak se vybírá "optimální" plán?
- Základní fyzické operátory : scany, joiny, ...
- Jak poznat že je něco špatně?
- Další užitečné nástroje.

K čemu slouží EXPLAIN a EXPLAIN ANALYZE?

SQL je deklarativní jazyk

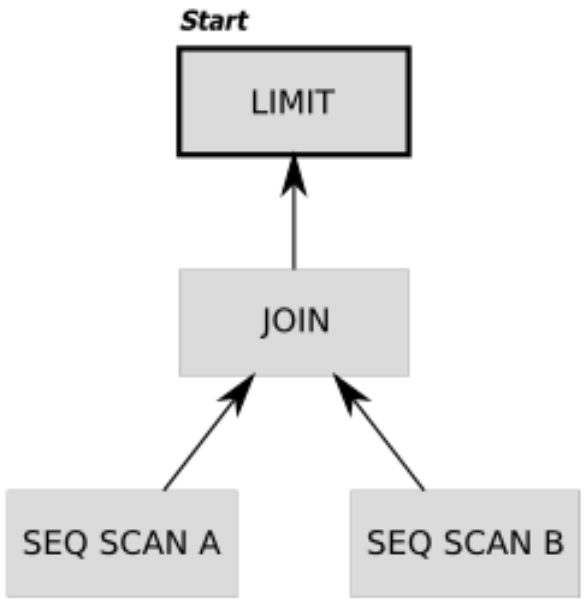
- SQL dotaz není program, popisuje výsledek (logická algebra).
- Existuje mnoho způsobů jak daný dotaz vyhodnotit (fyzická algebra).
- Nalezení "optimálního" způsobu je starostí databáze.
- Optimální = nejméně náročný na zdroje (CPU, I/O, paměť, ...)
- Závisí na podmínkách (počet uživatelů, velikost work_mem, ...).

stupně volnosti

- access strategy (sequential scan, index scan, ...)
- join order
- join strategy (merge join, hash join, nested loop)
- aggregation strategy (plain, hash, sorted)

Stromová struktura exekučního plánu

```
SELECT * FROM a JOIN b ON (a.id = b.id) LIMIT 100;
```



Výpočet ceny

- chci porovnat několik variant řešení a vybrat tu "nejlevnější"
- přístup obvyklý v (ne)lineárním programování
- ze statistik se odhadne počet řádek
- s využitím "cost" proměnných se spočte cena plánu
  - seq_page_cost = 1.0
  - random_page_cost = 4.0
  - cpu_tuple_cost = 0.01
  - cpu_index_tuple_cost = 0.005
  - cpu_operator_cost = 0.0025
  - ...
- porovnám ceny možností, vyberu tu s nejnižší cenou

Orientační principy

- I/O tradičně dominuje - minimalizace I/O operací
- náhodné I/O je náročnější než sekvenční I/O
- minimalizace CPU operací
- nepoužívat příliš mnoho paměti
- minimalizace toku dat
- preferovat nižší startup nebo celkovou cenu (?)

Cena je zhruba čas proporčně k sekvenčnímu načtení stránky z disku.

Základní fyzické operátory / přístup k tabulce

sequential scan

- přečti všechny řádky tabulky (a až pak filtruj)
- data (bloky) se čtou sekvenčně, každý právě 1x

index scan

- najdi v indexu odkazy na odpovídající řádky
- z tabulky načti jen ty potřebné bloky (i opakovaně)
- kombinace sekvenčního a **náhodného** I/O

bitmap index scan

- přečti listy indexu, vytvoř z nich bitmapu řádků
- načti jen ty bloky tabulky pro které je v bitmapě "1"
- **sekvenční** I/O ale "startup" cena (tvorba bitmapy)
- možnost kombinace více indexů (OR, AND)
- flexibilnější než multi-column indexy

Příklad - vytvoření tabulky

tabulka se 100.000 řádků

```
CREATE TABLE tab (id INT);

INSERT INTO tab SELECT * FROM generate_series(1,100000);

ANALYZE tab;

SELECT relpages, reltuples FROM pg_class
                         WHERE relname = 'tab';

 relpages | reltuples
----------+-----------
      393 |    100000
(1 row)
```

Příklad - sequential vs. index scan

sekvenční sken

```
EXPLAIN SELECT * FROM tab WHERE id BETWEEN 1000 AND 2000;

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tab  (cost=0.00..1893.00 rows=927 width=4)
   Filter: ((id >= 1000) AND (id <= 2000))
```

index scan

```
CREATE INDEX idx ON tab(id);
EXPLAIN ANALYZE SELECT * FROM tab WHERE id BETWEEN 1000 AND 2000;

                               QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using idx on tab  (cost=0.00..39.54 rows=1014 width=4)
                              (actual time=0.108..1.703 rows=1001 loops=1)
   Index Cond: ((id >= 1000) AND (id <= 2000))
 Total runtime: 2.840 ms
```

Příklad - bitmap index scan

bitmap index scan

```
EXPLAIN SELECT * FROM tab WHERE (id = 110 OR id = 130);

                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on tab (cost=8.53..16.14 rows=2 width=4)
   Recheck Cond: ((id = 110) OR (id = 130))
   ->  BitmapOr  (cost=8.53..8.53 rows=2 width=0)
         ->  Bitmap Index Scan on idx  (cost=0.00..4.27 rows=1 width=0)
               Index Cond: (id = 110)
         ->  Bitmap Index Scan on idx  (cost=0.00..4.27 rows=1 width=0)
               Index Cond: (id = 130)
```

Join strategies

- nested loop
- hash join
- merge join

Nested loop

- velice jednoduchý - v principu dvě vnořené smyčky
- pro větší relace pomalý, ale rychle produkuje první řádek
- jediný join použitelný pro CROSS JOIN a non-equijoin podmínky
- většinou je k vidění v OLTP systémech (práce s malými počty řádek)

```
FOR a IN vnejsi_relace
   FOR b IN vnitrni_relace
      RETURN (a,b) pokud splňuje JOIN podmínku
```

Nested Loop

```
CREATE TABLE vnejsi  (id INT, val INT UNIQUE);
CREATE TABLE vnitrni (id INT PRIMARY KEY);

INSERT INTO vnejsi
SELECT i, i+1 FROM generate_series(1,1000) s(i);

INSERT INTO vnitrni
SELECT i FROM generate_series(1,1000) s(i);
```

```
EXPLAIN SELECT 1 FROM vnejsi, vnitrni
                WHERE vnejsi.id = vnitrni.id
                  AND vnejsi.val = 10;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..16.55 rows=1 width=0)
   ->  Index Scan using vnejsi_val_key on vnejsi  (cost=0.00..8.27 rows=1 width=4)
         Index Cond: (val = 10)
   ->  Index Scan using vnitrni_pkey on vnitrni  (cost=0.00..8.27 rows=1 width=4)
         Index Cond: (vnitrni.id = vnejsi.id)
(5 rows)
```

Merge Join

- setřídí obě relace dle joinovací podmínky (jen equijoin)
- potom čte řádek po řádku a posouvá se kupředu
- někdy jsou potřeba rescany (duplicity ve vnější tabulce)
- velmi rychlý pro setříděné relace, jinak náročný startup
- většinou k vidění v DSS/DWH systémech

Merge Join

```
CREATE TABLE vnejsi (id INT);
CREATE TABLE vnitrni (id INT);

INSERT INTO vnejsi
SELECT i FROM generate_series(1,100000) s(i);

INSERT INTO vnitrni
SELECT i FROM generate_series(1,100000) s(i);
```

```
EXPLAIN SELECT 1 FROM vnejsi JOIN vnitrni USING (id);

                                QUERY PLAN
-------------------------------------------------------------------------
 Merge Join  (cost=19395.64..21395.64 rows=100000 width=0)
   Merge Cond: (vnejsi.id = vnitrni.id)
   ->  Sort  (cost=9697.82..9947.82 rows=100000 width=4)
         Sort Key: vnejsi.id
         ->  Seq Scan on vnejsi  (cost=0.00..1393.00 rows=100000 width=4)
   ->  Sort  (cost=9697.82..9947.82 rows=100000 width=4)
         Sort Key: vnitrni.id
         ->  Seq Scan on vnitrni  (cost=0.00..1393.00 rows=100000 width=4)
```

Hash Join

1. načti menší (vnitřní) relaci a vygeneruj z ní hash tabulku (přes join klíč)
2. čti vnější tabulku a vyhledávej v hash tabulce před hash klíče

```
CREATE TABLE vnejsi  (id INT);
CREATE TABLE vnitrni (id INT);

INSERT INTO vnejsi  SELECT i FROM generate_series(1,100000) s(i);
INSERT INTO vnitrni SELECT i FROM generate_series(1,100000) s(i);
```

```
EXPLAIN SELECT 1 FROM vnejsi_tabulka JOIN vnitrni_tabulka USING (id);

                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=2985.00..7029.00 rows=100000 width=0)
   Hash Cond: (vnejsi.id = vnitrni.id)
   ->  Seq Scan on vnejsi  (cost=0.00..1393.00 rows=100000 width=4)
   ->  Hash  (cost=1393.00..1393.00 rows=100000 width=4)
         ->  Seq Scan on vnitrni  (cost=0.00..1393.00 rows=100000 width=4)
(5 rows)
```

Hash Join / batches

Co když se hash tabulka nevejde to paměti (work_mem)?

1. rozděl menší tabulku na části, aby se tabulka do paměti vešla
2. pro každou část sestav tabulku a proveď join s "velkou" tabulkou
3. méně efektivní (opakované čtení vnější tabulky)
4. pozná se dle "batches" v plánu

```
EXPLAIN ANALYZE SELECT 1 FROM vnejsi_tabulka JOIN vnitrni_tabulka USING (id);

                                    QUERY PLAN
------------------------------------------------------------------------------- ...
 Hash Join  (cost=2985.00..7029.00 rows=100000 width=0) (actual time=277.886..792 ...
   Hash Cond: (vnejsi.id = vnitrni.id)
   ->  Seq Scan on vnejsi  (cost=0.00..1393.00 rows=100000 width=4) (actual time= ...
   ->  Hash  (cost=1393.00..1393.00 rows=100000 width=4) (actual time=277.836..27 ...
         Buckets: 8192  Batches: 4  Memory Usage: 589kB
         ->  Seq Scan on vnitrni  (cost=0.00..1393.00 rows=100000 width=4) (actua ...
 Total runtime: 900.664 ms
(7 rows)
```

- zvyšte work_mem (čím méně batchů, tím většinou lépe)

Srovnání join metod

Nested Loop

- špatně funguje pro dvě velké relace
- ideální pro malou vnější relaci + rychlý dotaz do vnitřní (index scan)
- jediná metoda pro non-equijoin :-(

Merge Join

- ideální pro již setříděné relace (např. CLUSTER + index scan)
- pokud vyžaduje extra třídění, problém (hlavně velké on-disk třídění)

Hash Join

- nevyžaduje třídění, musí ale vytvořit hash tabulku
- vyžaduje ale dostatek paměti (work_mem pro hash tabulku)
- pokud je hash tabulka moc velká, dělí se do batchů (pomalejší)

Sort & Limit

- ORDER BY ale i spousta dalších (DISTINCT, GROUP BY, UNION)
- tři možnosti
  - quicksort (v paměti, omezeno work_mem)
  - merge sort (na disku)
  - index scan (dostatečně korelovaný index, např. CLUSTERED)
- LIMIT říká "chci jenom pár řádek, preferuj rychle startující plány"
- většinou malý startovní čas znamená velký celkový čas

```
EXPLAIN ANALYZE SELECT * FROM tab ORDER BY id;

 Sort  (...) (actual time=446.089..591.714 rows=100000 loops=1)
   Sort Key: id
   Sort Method:  external sort  Disk: 1368kB
   ->  Seq Scan on tab  (...) (actual time=0.016..129.756 rows=100000 loops=1)
```

v paměti

```
SET work_mem = '8MB';

EXPLAIN ANALYZE SELECT * FROM tab ORDER BY id;

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (...) (actual time=312.709..432.410 rows=100000 loops=1)
   Sort Key: id
   Sort Method:  quicksort  Memory: 4392kB
   ->  Seq Scan on tab  (...) (actual time=0.020..146.975 rows=100000 loops=1)
```

s dobře korelovaným indexem

```
CREATE INDEX idx ON tab(id);

EXPLAIN ANALYZE SELECT * FROM tab ORDER BY id;

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using idx on tab  (cost=0.00..2780.26 rows=100000 width=4)
                              (actual time=0.088..162.377 rows=100000 loops=1)
 Total runtime: 272.881 ms
```

Typy uzlů - ostatní

- agregace (GROUP BY, DISTINCT)
- LIMIT
- modifikace tabulky (INSERT, UPDATE, DELETE)
- množinové operace (INTERSECT, EXCEPT)
- subplan (pro korelované subselecty), initplan (nekorelované)
- CTE, window functions
- materializace
- zamykání řádek
- append (inheritance)
- ...

Chybný odhad počtu řádků (relace resp. vyhovujících podmínce).

V čem spočívá problém?

- Plánovač si myslí že tabulka je malá ale ve skutečnosti je velká.
- Plánovač si myslí že podmínce vyhovuje pár řádek, ve skutečnosti mnoho.
- nebo naopak ...

Jak se projevuje?

- volí se nevhodný způsob přístupu k tabulce (index vs. sekvenční sken)
- volí se nevhodný způsob joinování (nested loop namísto hash/merge joinu)

Co je příčinou?

- zastaralé statistiky (např. hned po loadu)
- chybné statistiky - občas počet distinct hodnot, nevhodná formulace podmínek
- podmínky na korelovaných sloupcích (cross-column statistiky zatím nejsou)
- LIMIT situaci většinou výrazně zhoršuje (preferuje plány s levným startem)

Příklad - zdánlivě velká selektivita

založeno na “race condition” - spustím dotaz ještě než se stačí přepočítat statistiky

```
CREATE TABLE tab (id INT);
CREATE INDEX idx ON tab(id);
INSERT INTO tab SELECT * FROM generate_series(1,100000);
ANALYZE tab;

DELETE FROM tab;
INSERT INTO tab SELECT 1111 FROM generate_series(1,100000);
```

```
EXPLAIN ANALYZE SELECT * FROM tab WHERE id = 1111;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using idx on tab  (cost=0.00..8.29 rows=1 width=4)
                              (actual time=0.049..166.562 rows=100000 loops=1)
   Index Cond: (id = 1111)
(3 rows)
```

```
... wait ....
```

```
EXPLAIN ANALYZE SELECT * FROM tab WHERE id = 1111;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on tab  (cost=0.00..2035.00 rows=100000 width=4)
                  (actual time=0.949..158.568 rows=100000 loops=1)
   Filter: (id = 1111)
```

Příklad - korelované sloupce

```
CREATE TABLE tab (a INT, b INT);
INSERT INTO tab SELECT i, i FROM generate_series(1,100000) s(i);
ANALYZE tab;
```

```
EXPLAIN ANALYZE SELECT * FROM tab WHERE a >= 50000 AND b <= 50000;

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on tab  (cost=0.00..1943.00 rows=25000 width=8)
                  (actual time=26.196..58.715 rows=1 loops=1)
   Filter: ((a >= 50000) AND (b <= 50000))
 Total runtime: 58.762 ms
(3 rows)
```

Další problematická místa

Nevhodné nastavení “cost” proměnných

- výchozí hodnoty vychází z “typického” systému
- nemusí nutně odpovídat tomu vašemu
- např. pokud máte SSD, stírá se rozdíl mezi náhodným a sekvenčním I/O
- pokud máte rychlé disky (15k SAS) tak částečně také, byť ne tak markantně
- malá effective_cache_size znevýhodňuje indexy

Černé díry

- triggery
- referenční integrita (cizí klíče bez indexů)

EXPLAIN kuchařka

Zkontrolujte uzly kde nesedí odhad počtu řádek.

- Malé rozdíly nevadí, řádové rozdíly už jsou problém.
- Pokud je špatně odhad, nemůže být volba plánu spolehlivá.
- Zkuste aktualizovat statistiky, přeformulovat podmínky, ...

Podívejte se na na uzly s největším proporčním rozdílem mezi cenou a časem.

- Jste si jisti že máte rozumně nastaveny proměnné?
- Změňte nastavení (v session) a sledujte jak se změní plán a výkon dotazu.

Při optimalizaci se soustřeďte na uzly s nejvyšší cenou / skutečným časem.

- Tam kde se tráví nejvíc času můžete optimalizací nejvíce získat.
- Nelze např. přidat index nebo zvýšit work_mem?

explain.depesz.com

Result: St4

options

HTML TEXT

exclusive	inclusive	rows x	rows	loops	node
117.921	250273.108	↑ 89.1	342107	1	→ Unique (cost=30938464.86..31166982.10 rows=30468966 width=89) (actual time=249353.521..250273.108 rows=342107 loops=1)
7460.052	250155.187	↑ 89.1	342108	1	→ Sort (cost=30938464.86..31014637.27 rows=30468966 width=89) (actual time=249353.518..250155.187 rows=342108 loops=1) Sort Key: (lower(u.samaccountname[1])), (g.cn[1]) Sort Method: external merge Disk: 13176kB
63.215	242695.135	↑ 89.1	342108	1	→ Append (cost=0.00..19340392.34 rows=30468966 width=89) (actual time=44.687..242695.135 rows=342108 loops=1)
204697.356	240132.584	↑ 11986.5	2535	1	→ Nested Loop (cost=0.00..19031015.08 rows=30385836 width=89) (actual time=44.685..240132.584 rows=2535 loops=1) Join Filter: ((u.primarygroupid[1] = ANY (tmp_g.primarygrouptoken)) OR (u.gidnumber[1] = ANY (tmp_g.gidnumber)) OR (tmp_g.dn = ANY (u.memberof)) OR (tmp_g.cn[1] = ANY (u.memberof)) OR (tmp_g.dn = ANY (u.groupmembership)) OR (tmp_g.cn[1] = ANY (u.groupmembership)) OR (u.samaccountname[1] = ANY (tmp_g.memberuid)) OR (u.dn = ANY (tmp_g.member)) OR (u.cn[1] = ANY (tmp_g.member)))
4.781	116.528	↑ 1.0	1350	1	→ Nested Loop (cost=0.00..1421.74 rows=1350 width=986) (actual time=0.054..116.528 rows=1350 loops=1)
8.864	76.647	↑ 1.0	1350	1	→ Nested Loop (cost=0.00..734.12 rows=1350 width=1023) (actual time=0.038..76.647 rows=1350 loops=1)
1.633	1.633	↑ 1.0	1350	1	→ Seq Scan on ldap_group_inheritance i (cost=0.00..46.50 rows=1350 width=166) (actual time=0.015..1.633 rows=1350 loops=1)
66.150	66.150	↑ 1.0	1	1350	→ Index Scan using ldap_import_groups_dn_key on ldap_import_groups tmp_g (cost=0.00..0.50 rows=1 width=940) (actual time=0.048..0.049 rows=1 loops=1350) Index Cond: (tmp_g.dn = i.groupdn)
35.100	35.100	↑ 1.0	1	1350	→ Index Scan using ldap_import_groups_dn_key on ldap_import_groups g (cost=0.00..0.50 rows=1 width=129) (actual time=0.022..0.026 rows=1 loops=1350) Index Cond: (g.dn = i.parentdn)
35318.700	35318.700	↑ 1.0	83130	1350	→ Seq Scan on ldap_import_users u (cost=0.00..3856.30 rows=83130 width=372) (actual time=0.006..26.162 rows=83130 loops=1350)
2499.336	2499.336	↓ 4.1	339573	1	→ Seq Scan on ldap_import_users u (cost=0.00..4687.60 rows=83130 width=126) (actual time=0.098..2499.336 rows=339573 loops=1)

- http://explain.depesz.com
- výborný nástroj pro vizualizaci a analýzu explain planu
- skvělé pro posílání plánu např. do e-mailových konferencí (nezmrší se)

explain.depesz.com

exclusive	inclusive	rows x	rows	loops	node
117.921	250273.108	↑ 89.1	342107	1	→ Unique (cost=30938464.86..31166982.10 rows=30468966 width=89)
7460.052	250155.187	↑ 89.1	342108	1	→ Sort (cost=30938464.86..31014637.27 rows=30468966 width=89 Sort Key: (lower(u.samaccountname[1])), (g.cn[1]) Sort Method: external merge Disk: 13176kB
63.215	242695.135	↑ 89.1	342108	1	→ Append (cost=0.00..19340392.34 rows=30468966 width=89)
204697.356	240132.584	↑ 11986.5	2535	1	→ Nested Loop (cost=0.00..19031015.08 rows=30385836 wi Join Filter: ((u.primarygroupid[1] = ANY (tmp_g.primarygro (tmp_g.dn = ANY (u.memberof)) OR (tmp_g.cn[1] = ANY (u (tmp_g.cn[1] = ANY (u.groupmembership)) OR (u.samacc (tmp_g.member)) OR (u.cn[1] = ANY (tmp_g.member)))
4.781	116.528	↑ 1.0	1350	1	→ Nested Loop (cost=0.00..1421.74 rows=1350 width=98
8.864	76.647	↑ 1.0	1350	1	→ Nested Loop (cost=0.00..734.12 rows=1350 width=
1.633	1.633	↑ 1.0	1350	1	→ Seq Scan on ldap_group_inheritance i (cost=0. (actual time=0.015..1.633 rows=1350 loops=1)
66.150	66.150	↑ 1.0	1	1350	→ Index Scan using ldap_import_groups_dn_key (cost=0.00..0.50 rows=1 width=940) (actual time Index Cond: (tmp_g.dn = i.groupdn)
35.100	35.100	↑ 1.0	1	1350	→ Index Scan using ldap_import_groups_dn_key on l

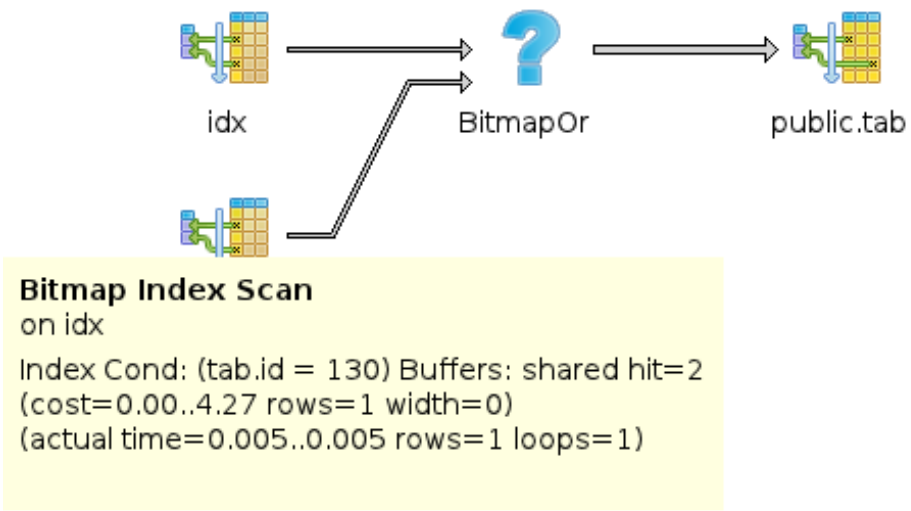
- Jak dlouho trval daný krok (samostatně / včetně podřízených)?
- Jak přesný byl odhad počtu řádek?
- Kolik řádek se vyprodukovalo?

explain.depesz.com

```
Unique  (cost=30938464.86..31166982.10 rows=30468966 width=89) (actual
time=249353.521..250273.108 rows=342107 loops=1)
->  Sort  (cost=30938464.86..31014637.27 rows=30468966 width=89) (actual
time=249353.518..250155.187 rows=342108 loops=1)
Sort Key: (lower(u.samaccountname[1])), (g.cn[1])
Sort Method:  external merge  Disk: 13176kB
->  Append  (cost=0.00..19340392.34 rows=30468966 width=89) (actual
time=44.687..242695.135 rows=342108 loops=1)
->  Nested Loop  (cost=0.00..19031015.08 rows=30385836 width=89) (actual
time=44.685..240132.584 rows=2535 loops=1)
Join Filter: ((u.primarygroupid[1] = ANY (tmp_g.primarygrouptoken)) OR
(u.gidnumber[1] = ANY (tmp_g.gidnumber)) OR (tmp_g.dn = ANY (u.memberof)) OR
(tmp_g.cn[1] = ANY (u.memberof)) OR (tmp_g.dn = ANY (u.groupmembership)) OR
(tmp_g.cn[1] = ANY (u.groupmembership)) OR (u.samaccountname[1] = ANY
(tmp_g.memberuid)) OR (u.dn = ANY (tmp_g.member)) OR (u.cn[1] = ANY
(tmp_g.member)))
->  Nested Loop  (cost=0.00..1421.74 rows=1350 width=986) (actual
time=0.054..116.528 rows=1350 loops=1)
->  Nested Loop  (cost=0.00..734.12 rows=1350 width=1023) (actual
time=0.038..76.647 rows=1350 loops=1)
->  Seq Scan on ldap_group_inheritance i  (cost=0.00..46.50 rows=1350 width=166)
(actual time=0.015..1.633 rows=1350 loops=1)
->  Index Scan using ldap_import_groups_dn_key on ldap_import_groups tmp_g
(cost=0.00..0.50 rows=1 width=940) (actual time=0.048..0.049 rows=1 loops=1350)
Index Cond: (tmp_g.dn = i.groupdn)
->  Index Scan using ldap_import_groups_dn_key on ldap_import_groups g
(cost=0.00..0.50 rows=1 width=129) (actual time=0.022..0.026 rows=1 loops=1350)
Index Cond: (g.dn = i.parentdn)
->  Seq Scan on ldap_import_users u  (cost=0.00..3856.30 rows=83130 width=372)
(actual time=0.006..26.162 rows=83130 loops=1350)
->  Seq Scan on ldap_import_users u  (cost=0.00..4687.60 rows=83130 width=126)
(actual time=0.098..2499.336 rows=339573 loops=1)
Total runtime: 250301.001 ms
(17 rows)
```

pgadmin3

- http://www.pgadmin.org/
- GUI umožňující mimo jiné i vizualizaci SQL dotazů



auto_explain & explanation

auto_explain

- jakýsi doplněk k `log_min_duration_statement`
- umožňuje logovat EXPLAIN (či EXPLAIN ANALYZE) pro dlouhé dotazy
- http://developer.postgresql.org/pgdocs/postgres/auto-explain.html

explanation

- flexibilnější práce s informacemi o plánu přímo v SQL
- http://www.pgxn.org/dist/explanation/doc/explanation.html

```
SELECT node_type, strategy, actual_startup_time, actual_total_time
  FROM explanation(
      query    := $$ SELECT * FROM pg_class WHERE relname = 'users' $$,
      analyzed := true
  );
```

```
 node_type  | strategy | actual_startup_time | actual_total_time
------------------------------------------------------------------
 Index Scan |          | 00:00:00.000017     | 00:00:00.000017
```

Odkazy

- Query Execution Techniques in PostgreSQL, Neil Conway, 2007
  `http://neilconway.org/talks/executor.pdf`
- Čtení prováděcích plánů v PostgreSQL, Pavel Stěhule, 2008
  `http://www.root.cz/clanky/cteni-provadecich-planu-v-postgresql/`
- Using EXPLAIN @ wiki
  `http://wiki.postgresql.org/wiki/Using_EXPLAIN`
- Introduction to VACUUM, ANALYZE, EXPLAIN, and COUNT @ wiki
  `http://wiki.postgresql.org/wiki/Introduction_to_VACUUM,_ANALYZE,_EXPLAIN,_and_COUNT`
- Explaining EXPLAIN, R. Treat, G. S. Mullane, AndrewSN, Magnifikus, B. Encina, N. Conway, 2008
  `http://wiki.postgresql.org/images/4/45/Explaining_EXPLAIN.pdf`