Čtení exekučních plánů

Tomáš Vondra <tv@fuzzy.cz>

pgconf.eu 2012

Praha, 23.10.2012

© 2012 Tomas Vondra, under Creative Commons Attribution-ShareAlike 3.0

http://creativecommons.org/licenses/by-sa/3.0/

Agenda

- úvod a trocha teorie
  - princip plánování, výpočet ceny
- praktické základy
  - EXPLAIN, EXPLAIN ANALYZE, ...
- základní operace, varianty
  - skeny, joiny, agregace, ...
- obvyklé problémy
- ukázky dotazů

Proč se o plánování starat?

- SQL je deklarativní jazyk
  - popisuje pouze požadovaný výsledek
  - volba postupu jeho získání je úkolem pro databázi

- Porozumění plánování je předpoklad pro
  - pochopení limitů databáze (implementačních, obecných)
  - definici efektivní DB struktury
  - analýzu problémů se stávajícími dotazy (pomalé, OOM)
  - lepší formulaci SQL dotazů

Plánování jako optimalizace

- hledáme “optimální” z ohromného množství plánů
- koncovým kritériem je čas běhu dotazu
  - strašně špatně se odhaduje a modeluje
- namísto toho se pracuje s “cenou”
  - založeno na statistikách tabulek/indexů a odhadech
  - zahrnuje nároky daného plánu na HW (CPU, RAM, I/O)
- cena
  - není čas ani s ním není lineárně závislá
  - měla by s časem korelovat (vyšší čas <=> vyšší cena)
  - měla by být stabilní (změna času ~ změna ceny)

Ukázka výpočtu ceny

- hledáme “optimální” z ohromného množství plánů

```
SELECT * FROM tabulka WHERE sloupec = 100
```

- tabulka má 1000 stránek a 1.000.000 řádek

```
cena = 1000 * cena_nacteni_stranky +
    1000000 * cena_zpracovani_radky +
    1000000 * cena_where_podminky
```

Cost proměnné

- udávají cenu některých základních operací
- celková cena se z nich vypočítává
  - **seq_page_cost = 1.0** – sekvenční čtení stránky (seq scan)
  - **random_page_cost = 4.0** – náhodné čtení stránky (index scan)
  - **cpu_tuple_cost = 0.01** – zpracování řádky z tabulky
  - **cpu_index_tuple_cost = 0.005** – zpracování řádky indexu
  - **cpu_operator_cost = 0.0025** – vyhodnocení podmínky (WHERE)

- je zřejmé že I/O operace jsou výrazně nákladnější

Ukázka výpočtu ceny - II.

- hledáme “optimální” z ohromného množství plánů

```
SELECT * FROM tabulka WHERE sloupec = 100
```

- tabulka má 1000 stránek a 1.000.000 řádek

```
cena = 1000 * 1.0 +
     1000000 * 0.01 +
     1000000 * 0.0025 = 22.500
```

Statistiky

- databáze si udržuje statistiky
- na úrovni tabulek – pg_class
  - relpages – počet stránek (8kB blok)
  - reltuples – počet řádek (nedpovídá COUNT(*))
- na úrovni řádek – pg_stats
  - avg_width – průměrná šířka hodnoty (v bytech)
  - n_distinct – počet různých hodnot
  - most_common_* - nejčastější hodnoty a jejich frekvence
  - histogram_bounds – histogram hodnot
  - null_frac – podíl NULL hodnot
  - correlation – korelace hodnot s pořadím v tabulce

Výběr plánu

- databáze musí
  - generovat možné exekuční plány
  - vybrat z nich ten nejlepší (s nejnižší cenou)
- co všechno je třeba brát v potaz
  - pořadí a způsob joinování tabulek (hash, merge, ...)
  - způsob čtení tabulek (sekvenčně, přes index, …)
  - pro které podmínky použít index
  - další operace (agregace, třídění, ...)
- počet plánů narůstá exponencielně
  - řešení jen hrubou silou není dostačující

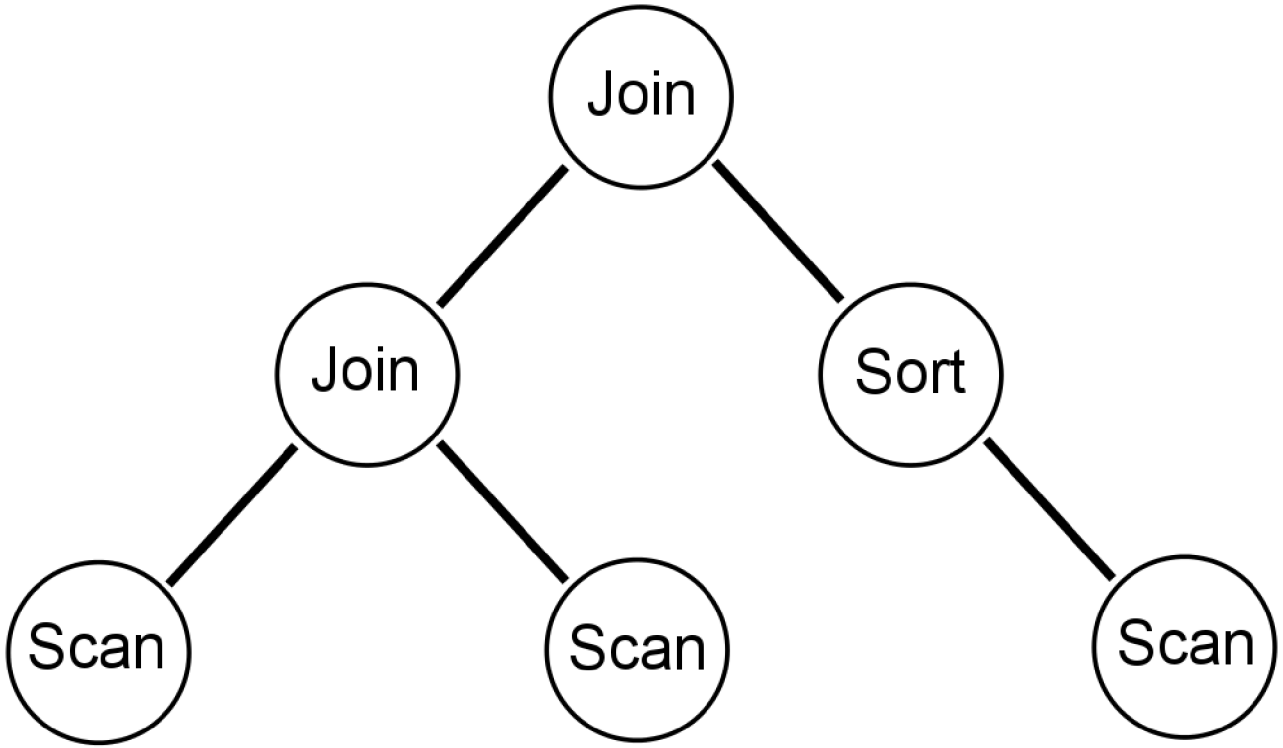
EXPLAIN

- zobrazí exekuční plán dotazu (nespustí ho)
- v plánu jsou uvedeny ceny a odhady počtu řádek

```
EXPLAIN SELECT SUM(a.id) FROM a,b WHERE a.id = b.id;

                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=58.75..58.76 rows=1 width=4)
   ->  Hash Join  (cost=27.50..56.25 rows=1000 width=4)
         Hash Cond: (a.id = b.id)
         ->  Seq Scan on a  (cost=0.00..15.00 rows=1000 width=4)
         ->  Hash  (cost=15.00..15.00 rows=1000 width=4)
               ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
```

- plán má stromovou strukturu
- listy jsou tradičně skeny tabulek, výše jsou operace

Plán jako strom



EXPLAIN

- každý uzel má dvě ceny
  - počáteční (startup) - do vygenerování první řádky
  - celkovou (total) - do vygenerování poslední řádky

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=58.75..58.76 rows=1 width=4)
   ->  Hash Join  (cost=27.50..56.25 rows=1000 width=4)
         Hash Cond: (a.id = b.id)
         ->  Seq Scan on a  (cost=0.00..15.00 rows=1000 width=4)
         ->  Hash  (cost=15.00..15.00 rows=1000 width=4)
               ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
```

- např. Hash Join má “startup=27.50” a total=”56.25”
  - očekávaný počet řádek je 1000, průměrná šířka 4B

EXPLAIN ANALYZE

- jako EXPLAIN, ale navíc dotaz provede a vrátí také
  - reálný čas (opět startup/total, jako v případě ceny)
  - skutečný počet řádek, počet opakování

```
EXPLAIN ANALYZE SELECT SUM(a.id) FROM a,b WHERE a.id = b.id;
-------------------------------------------------------------------
 Aggregate  (cost=58.75..58.76 rows=1 width=4)
            (actual time=4.149..4.149 rows=1 loops=1)
   ->  Hash Join  (cost=27.50..56.25 rows=1000 width=4)
                          (actual time=1.515..3.654 rows=1000 loops=1)
         Hash Cond: (a.id = b.id)
         ->  Seq Scan on a  (cost=0.00..15.00 rows=1000 width=4)
                          (actual time=0.036..0.533 rows=1000 loops=1)
         ->  Hash  (cost=15.00..15.00 rows=1000 width=4)
                    (actual time=1.440..1.440 rows=1000 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 36kB
               ->  Seq Scan on b  (cost=0.00..15.00 rows=1000 width=4)
                          (actual time=0.027..0.560 rows=1000 loops=1)
 Total runtime: 4.263 ms
```

pg_test_timing

- instrumentace v EXPLAIN ANALYZE není zadarmo
- často se stává že měření času má značný overhead
  - dotaz pak běží např. 10x déle a mění se poměr kroků
  - závisí na HW/OS
- možnost otestovat nástrojem v PostgreSQL

```
$ pg_test_timing
    Testing timing overhead for 3 seconds.
    Per loop time including overhead: 23.49 nsec
    Histogram of timing durations:
    < usec:       count   percent
            8:           2  0.00000%
            4:          39  0.00003%
            2:     2999907  2.34869%
            1:   124726830 97.65128%
```

- histogram, cílem je mít >90% pod 1 usec

Obvyklé problémy

Jak identifikovat problém?

Soustřeďte se na uzly s ...

- velkou odchylkou odhadu počtu řádek a reality
  - chyby menší než o řád jsou vesměs považovány za malé
  - skutečným problémem jsou odchylky alespoň o řád (10x více/méně)
- největším proporcionálním rozdílem mezi odhadem a reálným časem
  - např. uzly s cenami 100 a 120, ale časy 1s a 1000s
  - může ukazovat na nevhodné hodnoty cost proměnných, nebo selhání plánovače, např. v důsledku neodhadnutí efektu cache
- největším reálným časem
  - plán může být naprosto v pořádku - za daných podmínek optimální
  - např. vám tam můžech chybět index nebo ho nejde použít kvůli formulaci podmínky, apod.

Neaktuální statistiky

```
CREATE TABLE stale_t (id int);
INSERT INTO stale_t SELECT i FROM generate_series(1,100000) s(i);
-- ANALYZE;
EXPLAIN ANALYZE SELECT id FROM stale_t WHERE id < 100;

                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on stale_t  (cost=0.00..1772.00 rows=35440 width=4)
                      (actual time=0.014..9.566 rows=99 loops=1)
   Filter: (id < 100)
   Rows Removed by Filter: 99901
```

- Databáze ví o počtu řádek, ale nemá statistiky (histogramy), takže používá “default” odhad selektivity 33%.
- Po velké změně dat nedošlo k aktualizaci statistik. Plánovač něco ví (celkový počet řádek), něco (např. histogramy) ne. Odhad selektivity je díky tomu špatný.
- Buď se spolehnout na autovacuum (OLTP) nebo volat ANALYZE ručně (dávkové procesy, loady dat apod.).

Neodhadnutelné podmínky

```
CREATE TABLE a AS SELECT i FROM generate_series(1,10000) s(i);
ANALYZE a;
EXPLAIN SELECT * FROM a WHERE i*i < -1;

                            QUERY PLAN
-------------------------------------------------------------------
Seq Scan on a  (cost=0.00..207.00 rows=3600 width=4)
               (actual time=1.180..1.180 rows=0 loops=1)
   Filter: ((i * i) < (-1))
   Rows Removed by Filter: 10000
 Total runtime: 1.193 ms
```

- použití funkcí a operací často znemožníte odhadování
- někdy jde přepsat na odhadnutelnou podmínku
  - odstrašující příklad: “`datum::text LIKE '2012-08-%'`”
  - přepis např. “`datum BETWEEN '2012-08-01' AND '2012-09-01'`”
- někdy jde manuálně provést “inverzi”
  - např: “`i*i <= 100`” => “`i BETWEEN -10 AND 10`”
- nelze opravit vytvořením “expression” indexu (odhady nezlepší)

Korelované sloupce

```
CREATE TABLE a (i int, j int);
INSERT INTO a SELECT i,i FROM generate_series(1,1000000) s(i);
ANALYZE a;
EXPLAIN ANALYZE SELECT * FROM a WHERE (a.i < 1000 AND a.j < 1000);

                          QUERY PLAN
-----------------------------------------------------
 Seq Scan on a  (cost=0.00..19425.00 rows=1 width=8)
          (actual time=0.008..71.538 rows=999 loops=1)
   Filter: ((i < 1000) AND (j < 1000))
   Rows Removed by Filter: 999001
 Total runtime: 71.579 ms
(4 rows)
```

- Odhazy jsou založeny na předpokladu nezávislosti sloupců, tj. DB předpokládá že selektivita podmínky na více sloupcích je součin jednotlivých podmínek.
- V příkladu má každá podmínka selektivitu 1/1000, takže vynásobením 1/1000000 – to znamená jeden řádek. Ale jsou závislé (i=j).
- Nemá dobré řešení (zatím).

Špatný odhad n_distinct

- odhad počtu různých hodnot obecně patří k nejtěžším
- většinou sedí, ale pro hodně “divné” databáze k němu může dojít
- n_distinct není nikde přímo vidět, projevuje se přes “rows”
- například při agregaci

```
EXPLAIN ANALYZE SELECT i, sum(val) FROM a GROUP BY i;

                                QUERY PLAN
-----------------------------------------------------------------------
 HashAggregate  (cost=1788.00..1789.00 rows=100 width=8)
                (actual time=36.341..55.409 rows=100001 loops=1)
   ->  Seq Scan on a  (cost=0.00..1341.00 rows=89400 width=8)
                      (actual time=0.008..6.469 rows=101000 loops=1)
 Total runtime: 58.254 ms
(3 rows)
```

- v extrémních případech může vést až k “out of memory” chybám
- statistiku lze ručně opravit pomocí “`ALTER TABLE … SET n_distinct …`”

Prepared statements

- pojmenované prepared statements se plánuji při PREPARE
- nepojmenované prepared statements (v uložených procedurách) se plánuji při prvním volání (s prvními použitými hodnotami)

```
CREATE TABLE a (val INT);
INSERT INTO a SELECT 1 FROM gs(1,100000) s(i);
INSERT INTO a SELECT 2;
CREATE INDEX a_idx ON a(val);

PREPARE select_a(int) AS SELECT * FROM a WHERE val = $1;
EXPLAIN EXECUTE select_a(2);

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on a  (cost=0.00..1693.01 rows=100001 width=4)
   Filter: (val = $1)
(2 rows)
```

- plánuje se podle nejčastějších hodnot – pro vzácné dává neoptimální plány
- od 9.2 se chová trochu jinak (kontroluje hodnoty a případně přeplánuje)

Obtížné joiny

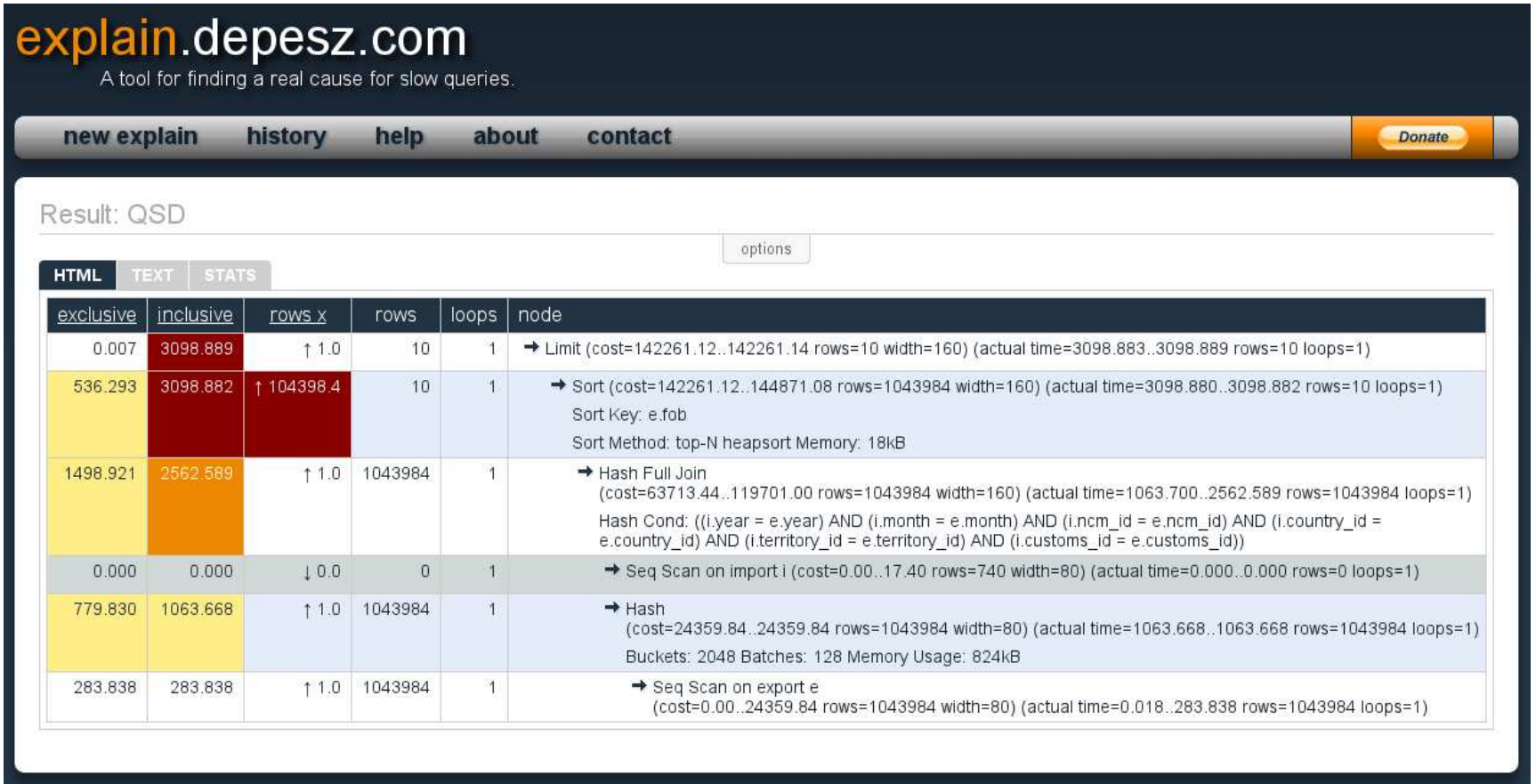
- joiny jsou jedny z nejdražších a nejhůře odhadnutelných operací

```
CREATE TABLE a AS SELECT 2*i AS i FROM gs(1,100000) s(i);

EXPLAIN SELECT * FROM a a1 JOIN a a2 ON (a1.i = a2.i);
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=2693.00..6136.00 rows=100000 width=8)
   Hash Cond: (a1.i = a2.i)
   ->  Seq Scan on a a1  (cost=0.00..1443.00 rows=100000 width=4)
   ->  Hash  (cost=1443.00..1443.00 rows=100000 width=4)
         ->  Seq Scan on a a2  (cost=0.00..1443.00 rows=100000 width=4)

EXPLAIN SELECT * FROM a a1 JOIN a a2 ON (a1.i = a2.i-1);

                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=2693.00..6886.00 rows=100000 width=8)
   Hash Cond: ((a2.i - 1) = a1.i)
   ->  Seq Scan on a a2  (cost=0.00..1443.00 rows=100000 width=4)
   ->  Hash  (cost=1443.00..1443.00 rows=100000 width=4)
         ->  Seq Scan on a a1  (cost=0.00..1443.00 rows=100000 width=4)
```

- první odhad je OK, ale druhý nemůže vrátit nic (sudý = lichý)

explain.depesz.com

- elegantní vizuální pohled na exekuční plán
- ideální způsob předávání exekučních plánů např. do konference
- často přímo zvýrazní problematické části (ústřel statistik, dlouhý běh)
- http://explain.depesz.com/



auto_explain

- často se stává že dotaz / exekuční plán blbne nepredikovatelně (například je pomalý jen v noci)
- při následném ručním průzkumu se všechno zdá naprosto OK - duchařina
- tento modul vám umožní exekuční plán odchytit právě když blbne
- máte stejné možnosti jako s EXPLAIN / EXPLAIN ANALYZE
- zalogovat můžete vše, jen dotazy přes nějaký limit apod.
- http://www.postgresql.org/docs/9.2/static/auto-explain.html

```
auto_explain.log_min_duration = 250
auto_explain.log_analyze = false
auto_explain.log_verbose = false
auto_explain.log_buffers = true
auto_explain.log_format = yaml
auto_explain.log_nested_statements = false
```

enable_*

- způsob jak ovlivnit exekuční plán (např. během ladění)
- nelze "hintovat" jako v jiných databázích (to je feature, ne bug)
- varianty operací ale lze zapnout/vypnout pro celý dotaz
  - ve skutečnosi nevypíná ale pouze výrazně znevýhodňuje

- enable_bitmapscan
- enable_indexscan
- enable_seqscan
- enable_tidscan
- enable_indexonlyscan
- enable_hashjoin
- enable_mergejoin
- enable_nestloop
- enable_hashagg
- enable_material
- enable_sort

Způsoby přístupu k tabulkám

Způsoby přístupu k tabulkám

- Sequetial Scan
- Index Scan
- Index Only Scan
- Bitmap Index Scan
- Function Scan

- *CTE Scan*
- *TID Scan*
- *Foreign Scan*
- ...

Sequential Scan

- nejjednodušší možný sken – sekvenčně čte tabulku
- řádky může zpracovat filtrem (WHERE podmínka)

```
CREATE TABLE a AS SELECT i FROM gs(1,100000) s(i);
ANALYZE a;
EXPLAIN ANALYZE SELECT i FROM a WHERE i = 1000;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on a  (cost=0.00..1693.00 rows=1 width=4)
                (actual time=0.080..6.866 rows=1 loops=1)
   Filter: (i = 1000)
 Total runtime: 6.880 ms
(3 rows)
```

- efektivní pro malé tabulky nebo při čtení velké části
- “nešpiní” shared buffers, synchronizované čtení

Index Scan

- využívá datovou strukturu optimalizovanou pro hledání (většinou nějaká forma stromu)

```
CREATE TABLE a AS SELECT i FROM gs(1,100000) s(i);
CREATE INDEX a_idx ON a(i);
ANALYZE a;
EXPLAIN ANALYZE SELECT i FROM a WHERE i = 1000;

                          QUERY PLAN
-----------------------------------------------------------------
 Index Scan using a_idx on a  (cost=0.00..8.28 rows=1 width=4)
                      (actual time=0.023..0.023 rows=1 loops=1)
   Index Cond: (i = 1000)
 Total runtime: 0.039 ms
(3 rows)
```

- efektivní pro čtení malé části z velké tabulky
- ne každá podmínka je použitelná pro index

Index Only Scan

- novinka v PostgreSQL 9.2
- vylepšení Index Scanu – odstranění nutnosti skákat do tabulky jen kvůli kontrole viditelnosti řádky
- nejedná se o tzv. “covering” indexy (tj. možnost číst indexy sekvenčně namísto tabulky)

```
CREATE TABLE a (id INT, val INT8);
INSER INTO a SELECT i,i FROM gs(1,1000000) s(i);
CREATE INDEX a_idx on a(id, val);

EXPLAIN SELECT val FROM a WHERE id = 230923;

                            QUERY PLAN
-------------------------------------------------------------------
 Index Only Scan using a_idx on a  (cost=0.00..9.81 rows=1 width=8)
   Index Cond: (id = 230923)
(2 rows)
```

Bitmap Index Scan

- Index Scan je efektivní pro malé počty řádek (např. 5%)
  - nepoužitelné pro podmínky s malou selektivitou
  - pro více řádek je smrtící náhodné I/O nad tabulkou

- Bitmap Index Scan čte tabulku sekvenčně pomocí indexu
  - nejdříve na základě indexu vytvoří bitmapu stránek
  - pokud alespoň jedna řádka odpovídá tak “1” jinak “0”
  - bitmap může být více a může je kombinovat (AND, …)
  - následně tabulku sekvenčně přečte pomocí bitmapy
  - musí dělat “recheck” protože neví které řádky vyhovují

Bitmap Index Scan

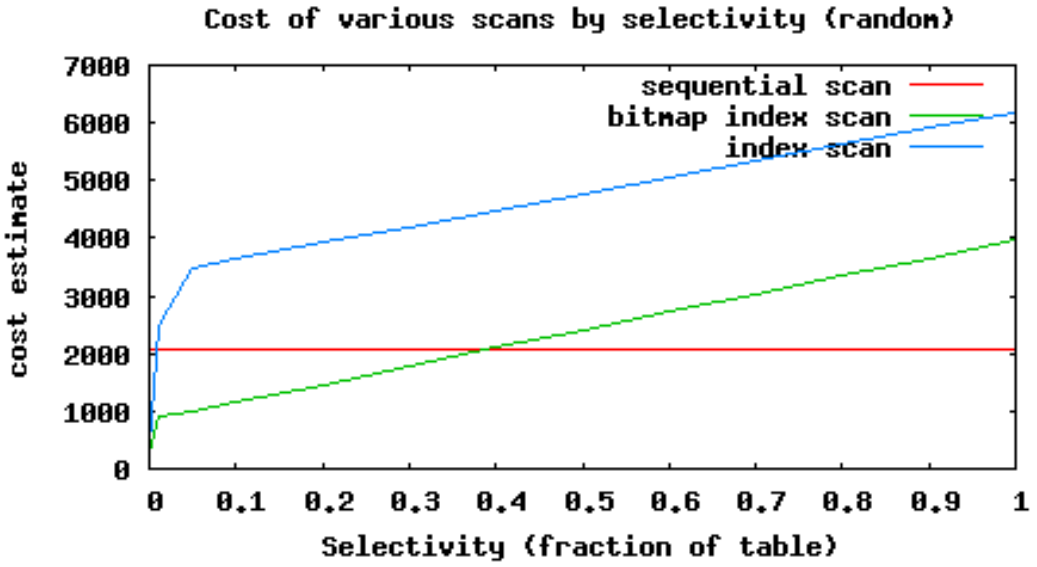
- Index Scan je efektivní pro malé počty řádek (např. 5%)
- nepoužitelné pro podmínky s malou selektivitou

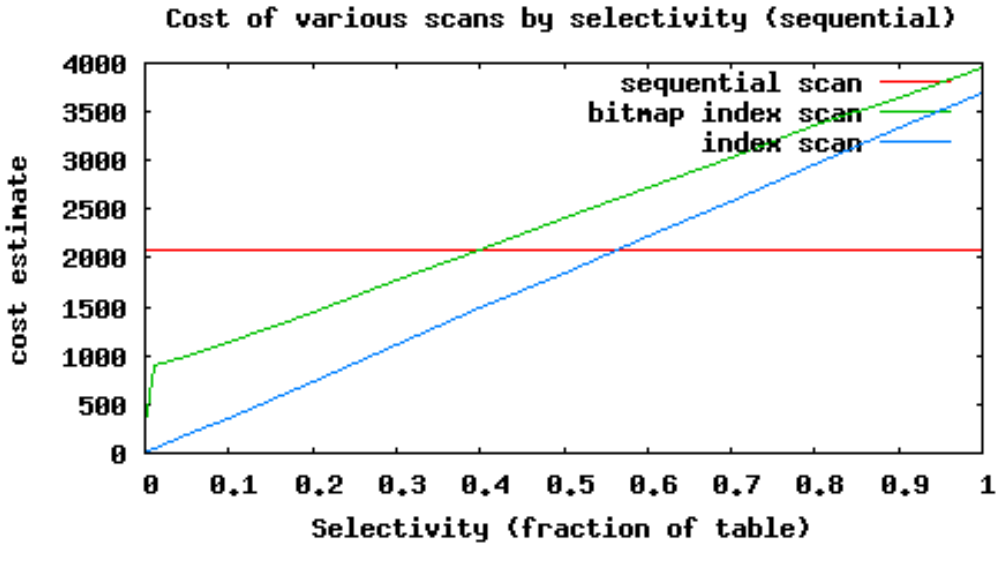
```
CREATE TABLE a AS SELECT mod(i,100) AS x,
                         mod(i,101) AS y FROM gs(1,1000000) s(i);
CREATE INDEX ax_idx ON a(x);
CREATE INDEX ay_idx ON a(y);

EXPLAIN SELECT * FROM a WHERE x < 5 AND y < 5;
                                    QUERY PLAN
-------------------------------------------------------------------------
Bitmap Heap Scan on a  (cost=1867.73..5844.45 rows=2537 width=8)
  Recheck Cond: ((x < 5) AND (y < 5))
  ->  BitmapAnd  (cost=1867.73..1867.73 rows=2537 width=0)
      ->  Bitmap Index Scan on ax_idx (cost=0.00..930.10 rows=50233 ...
            Index Cond: (x < 5)
      ->  Bitmap Index Scan on ay_idx (cost=0.00..936.12 rows=50503 ...
            Index Cond: (y < 5)
(7 rows)
```

Srovnání skenů

- vezměme tabulku (1000000 integerů v náhodném pořadí)
- sledujme cenu 3 základních plánů pro podmínku s různou selektivitou

```
CREATE TABLE a AS SELECT i, md5(i::text) m
                    FROM generate_series(1,100000) s(i)
                    [ORDER BY random()];

CREATE INDEX a_idx ON a(i);

SELECT * FROM a WHERE i < (100000 * selektivita);
```





Function Scan

- set-returning-functions (SRF) – funkce vracející tabulku
- ceny a počty řádek jsou konstanty, dané při kompilaci
- nepřesné odhady působí problémy při plánování
- zkuste “generate_series” s různými počty a podmínkami

```
CREATE FUNCTION moje_tabulka(n INT) RETURNS SETOF INT AS $$
DECLARE
    i INT := 0;
BEGIN

    FOR i IN 1..n LOOP
        RETURN NEXT i;
    END LOOP;

    RETURN;
END;
$$ LANGUAGE plpgsql COST 10 ROWS 100;
```

CTE Scan

```
WITH b AS (SELECT * FROM a WHERE i >= 100)
SELECT * FROM b WHERE i <= 110
  UNION ALL
SELECT * FROM b WHERE i <= 120;
```

- opakované výrazy je možno uvést jako “WITH”
- vyhodnotí se jen jednou, ne pro každou větev znovu

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Result  (cost=17906.00..69567.50 rows=666600 width=12)
   CTE b
     ->  Seq Scan on a  (cost=0.00..17906.00 rows=999900 width=12)
           Filter: (i >= 100)
   ->  Append  (cost=0.00..51661.50 rows=666600 width=12)
         ->  CTE Scan on b  (cost=0.00..22497.75 rows=333300 width=12)
               Filter: (i <= 110)
         ->  CTE Scan on b  (cost=0.00..22497.75 rows=333300 width=12)
               Filter: (i <= 120)
```

- nevyhodnocují se “na začátku” ale průběžně

Foreign Scan

- Foreign Data Wrappers – cizí datové zdroje
- značné výhody oproti prostým SRF ale složitější
- integrace s plánovačem
  - možnost použití některých podmínek z AST
  - možnost vlastních odhadů apod.
- data která dokážete reprezentovat jako tabulku
  - CSV soubory, další RDBMS, Twitter, ...

Foreign Scan

```
for i in `seq 1 1000`; do
    echo $i,"message $i" >> /tmp/my.csv;
done;

CREATE EXTENSION file_fdw;
CREATE SERVER csv FOREIGN DATA WRAPPER file_fdw;

CREATE FOREIGN TABLE csv_import (
  process_id integer,
  message text
)SERVER csv OPTIONS ( filename '/tmp/my.csv', format 'csv' );

EXPLAIN SELECT * FROM csv_import;

                            QUERY PLAN
-------------------------------------------------------------------
 Foreign Scan on csv_import  (cost=0.00..26.70 rows=247 width=36)
   Foreign File: /tmp/my.csv
   Foreign File Size: 15786
(3 rows)
```

Další operace

Agregace, třídění, LIMIT, ...

Agregace

```
CREATE TABLE a (i INT, j INT, k INT);

INSERT INTO a SELECT mod(i, 1000), mod(i, 1333), mod(i,3498)
                FROM gs(1,100000) s(i);

EXPLAIN SELECT i, count(*) FROM a GROUP BY i;

EXPLAIN SELECT DISTINCT i  FROM a GROUP BY i;

                           QUERY PLAN
-----------------------------------------------------------------
 HashAggregate  (cost=2041.00..2141.00 rows=10000 width=8)
   ->  Seq Scan on a  (cost=0.00..1541.00 rows=100000 width=8)
(2 rows)
```

- Aggregate - v případech bez GROUP BY (vlastně jeden řádek)
- Group Aggregate - k detekci skupin využívá třídění vstupní relace
  - nemusí čekat na dokončení agregace, ale potřebuje setříděný vstup
- Hash Aggregate - využívá hash tabulku, za určitých podmínek může alokovat hodně paměti (výběr metody nelze za běhu měnit)

Agregace / OOM

- HashAggregate není adaptivní – plán nelze za běhu změnit a tabulku nelze za běhu “dělit”
- spíše výjimečně, autoanalyze většinou včas odchytí
- typicky je důsledkem nepřesných statistik na tabulce

```
EXPLAIN ANALYZE
SELECT i, count(*) FROM generate_series(1,100000000) s(i)
GROUP BY i;

SELECT i, count(i) FROM a GROUP BY i;
ERROR:  out of memory
DETAIL:  Failed on request of size 20.
```

Třídění

- tři základní varianty třídění
  - pomocí indexu (Index Scan)
  - v paměti (quicksort)
  - na disku (merge sort)
- mezi quick-sort a merge-sortem se volí za běhu
  - dokud stačí RAM (work_mem), používá se quick-sort
  - poté se začne zapisovat na disk – nikdy OOM
- třídění pomocí indexu má malé počáteční náklady
  - nemusí čekat na všechny řádky, vrací je hned
  - cena ale rychle roste (podle korelace s tabulkou apod.)

Třídění

```
EXPLAIN ANALYZE SELECT * FROM a ORDER BY i;

                                QUERY PLAN
-----------------------------------------------------------------------------
 Sort  (cost=114082.84..116582.84 rows=1000000 width=4)
                  (actual time=1018.108..1230.263 rows=1000000 loops=1)
   Sort Key: i
   Sort Method: external merge  Disk: 13688kB
   ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=4)
                       (actual time=0.005..68.491 rows=1000000 loops=1)
 Total runtime: 1263.166 ms
(5 rows)


CREATE INDEX a_idx ON a(i);
EXPLAIN SELECT * FROM a ORDER BY i;

                                QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using a_idx on a (cost=0.00..43680.14 rows=1000000 width=4)
(1 row)
```

LIMIT/OFFSET

- zatím jsme pracovali s celkovou cenou (total cost)
- často ale není třeba vyhodnotit všechny řádky
  - například stačí jen ověřit existenci (LIMIT 1)
  - časté jsou "top N" dotazy (ORDER BY x LIMIT n)
- cena LIMIT je lineární interpolací – databáze zná
  - startup a total cost
  - počty řádek (požadovaný a celkový)

```
startup_cost + (total_cost - startup_cost) * (rows / limit)
```

LIMIT a nerovnoměrné rozložení

- identifikace tohoto problému je poměrně těžká
- problematický případ

```
CREATE TABLE a (id INT);
INSERT INTO a SELECT i/100 FROM generate_series(1,1000000) s(i);
EXPLAIN ANALYZE SELECT * FROM a WHERE id = 9999 LIMIT 1;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit (cost=0.00..172.70 rows=1 width=4) (actual time=71.00..71.00 rows=1 loops=1)
   -> Seq Scan on a (cost=0.00..16925.00 rows=98 width=4)
                    (actual time=71.00..71.00 rows=1 loops=1)
       Filter: (id = 9999)
       Rows Removed by Filter: 999899
```

- příznivý případ (dá se poznat pouze dle “rows removed by filter”)

```
INSERT INTO a SELECT mod(i,10000) FROM generate_series(1,1000000) s(i);

                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit (cost=0.00..172.70 rows=1 width=4) (actual time=0.72..0.72 rows=1 loops=1)
   -> Seq Scan on a (cost=0.00..16925.00 rows=98 width=4)
                    (actual time=0.72..0.72 rows=1 loops=1)
         Filter: (id = 9999)
         Rows Removed by Filter: 9998
```

Triggery

- dlouho “černá hmota” plánování – nikde nebylo vidět
  - kromě doby trvání dotazu ;-)
- zahrnuje i triggery které realizují referenční integritu
- častý problém – cizí klíč bez indexu na child tabulce
  - změny nadřízené tabulky trvají dlouho (např. DELETE)
  - vyžadují totiž kontrolu podřízené tabulky

Triggery

```
CREATE TABLE parent (id INT PRIMARY KEY);

CREATE TABLE child (id  INT PRIMARY KEY,
                    pid INT REFERENCES parent(id));

INSERT INTO parent SELECT i FROM generate_series(1,100) s(i);
INSERT INTO child  SELECT i, 1 from generate_series(1,10000) s(i);

EXPLAIN ANALYZE DELETE FROM parent WHERE id > 1;

                                QUERY PLAN
-----------------------------------------------------------------------
 Delete on parent  (cost=0.00..2.25 rows=100 width=6)
                                (actual time=0.081..0.081 rows=0 loops=1)
   ->  Seq Scan on parent  (cost=0.00..2.25 rows=100 width=6)
                           (actual time=0.007..0.019 rows=99 loops=1)
         Filter: (id > 1)
         Rows Removed by Filter: 1
 Trigger for constraint child_pid_fkey: time=75.671 calls=99
 Total runtime: 75.774 ms
(6 rows)
```

Joinování tabulek

Nested Loop, Hash Join, Merge Join

Joiny obecně

- všechny joiny pracují se dvěma vstupními relacemi
- první je označována jako vnější (outer), druhá jako vnitřní (inner)
  - nemá nic společného s inner/outer joinem
  - vychází z rozdílného postavení tabulek v algoritmech
- **join_collapse_limit**
  - proměnná ovlivňující jak moc může plánovač měnit pořadí tabulek během joinu
  - dá se zneužít ke “vnucení” pořadí použitím explicitního joinu a SET join_collapse_limit = 1
- **geqo_threshold**
  - určuje kdy se má opustit vyčerpávající hledání pořadí tabulek a přejít na genetický algoritmus
  - ten je rychlejší ale nemusí najít některé kombinace

Nested Loop

- asi nejjednodušší algoritmus (smyčka přes "outer" tabulku, dohledání záznamu v "inner" tabulce)
- vhodný pro málo iterací a/nebo levný vnitřní plán (např. maličká nebo dobře oindexovaná tabulka)
- jediná varianta joinu pro kartézský součin a nerovnosti

```
CREATE TABLE a AS SELECT i FROM generate_series(1,10000) s(i);
CREATE TABLE b AS SELECT i FROM generate_series(1,10000) s(i);

EXPLAIN SELECT * FROM a, b;
                              QUERY PLAN
-----------------------------------------------------------------
--
 Nested Loop  (cost=0.00..1250315.00 rows=100000000 width=8)
   ->  Seq Scan on a  (cost=0.00..145.00 rows=10000 width=4)
   ->  Materialize  (cost=0.00..195.00 rows=10000 width=4)
         ->  Seq Scan on b  (cost=0.00..145.00 rows=10000
width=4)
```

Nested Loop

- Kartézský součin není příliš obvyklý, přidejme index a podmínku na jednu tabulku.

```
CREATE INDEX b_idx ON b(i);
EXPLAIN SELECT * FROM a JOIN b USING (i) WHERE a.i < 10;

                                QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.00..240.63 rows=9 width=4)
   ->  Seq Scan on a  (cost=0.00..170.00 rows=9 width=4)
         Filter: (i < 10)
   ->  Index Scan using b_idx on b (cost=0.00..7.84 rows=1 width=4)
         Index Cond: (i = a.i)
(5 rows)
```

- vypadá rozumněji, podobné plány jsou celkem běžné
- uvnitř většinou index (only) scan, maličká tabulka, ...

Nested Loop

```
EXPLAIN ANALYZE SELECT * FROM a JOIN b USING (i) WHERE a.i < 10;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..240.63 rows=9 width=12)
              (actual time=0.013..0.735 rows=9 loops=1)
   ->  Seq Scan on a  (cost=0.00..170.00 rows=9 width=8)
                      (actual time=0.009..0.719 rows=9 loops=1)
         Filter: (i < 10)
         Rows Removed by Filter: 9991
   ->  Index Scan using b_idx on ba  (cost=0.00..7.84 rows=1 width=8)
                            (actual time=0.001..0.001 rows=1 loops=9)
         Index Cond: (i = a.i)
 Total runtime: 0.755 ms
```

- ceny uvedené u vnitřního plánu jsou průměry na jedno volání
- loops - počet volání vnitřního plánu (nemusí se nutně pustit vůbec)
- **obvyklý problém č. 1**: podstřelení odhadu počtu řádek první tabulky
- **obvyklý problém č. 2**: podstřelení ceny vnořeného plánu

Hash Join

```
EXPLAIN SELECT * FROM a JOIN b USING (i) WHERE a.i < 1000;

                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=182.50..375.00 rows=1000 width=12)
   Hash Cond: (b.i = a.i)
   ->  Seq Scan on b  (cost=0.00..145.00 rows=10000 width=8)
   ->  Hash  (cost=170.00..170.00 rows=1000 width=8)
         ->  Seq Scan on a  (cost=0.00..170.00 rows=1000 width=8)
               Filter: (i < 1000)
(6 rows)
```

- menší relaci načte do hash tabulky (pro rychlé vyhledání podle join klíče)
- pokud se nevejde do work_mem, rozdělí ji na tzv. "batche"
- následně čte větší tabulku a v hash tabulce vyhledává záznamy
- pokud byly použity batche, velká tabulka se čte několikrát (jednou pro každý batch hash tabulky) - je to vidět v EXPLAIN ANALYZE
- jde použít jenom pro equi-joiny (joinovací podmínka je rovnost)

Hash Join

```
EXPLAIN ANALYZE SELECT * FROM a JOIN b USING (i);

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=30832.00..74478.00 rows=1000000 width=12)
                    (actual time=247.928..759.196 rows=1000000 loops=1)
   Hash Cond: (a.i = b.i)
   ->  Seq Scan on a  (cost=0.00..14425.00 rows=1000000 width=8)
                      (actual time=0.007..66.813 rows=1000000 loops=1)
   ->  Hash  (cost=14425.00..14425.00 rows=1000000 width=8)
                    (actual time=247.384..247.384 rows=1000000 loops=1)
         Buckets: 4096  Batches: 64  Memory Usage: 625kB
         ->  Seq Scan on b  (cost=0.00..14425.00 rows=1000000 width=8)
                      (actual time=0.004..98.268 rows=1000000 loops=1)
 Total runtime: 788.159 ms
```

- čím víc segmentů, tím hůře - opakovaně se čte vnější tabulka :-(
- jediné řešení asi je zvětšit work_mem nebo vymyslet jinou query
- jedna hash tabulka nepřekročí work_mem - batchování se děje až za běhu podle situace (špatný odhad nezpůsobí OOM)

Merge Join

```
CREATE TABLE a AS SELECT i, md5(i::text) val FROM gs(1,100000) s(i);
CREATE TABLE b AS SELECT i, md5(i::text) val FROM gs(1,100000) s(i);
CREATE INDEX a_idx ON a(i);
CREATE INDEX b_idx ON b(i);
ANALYZE;

EXPLAIN SELECT * FROM a JOIN b USING (i);

                                  QUERY PLAN
-------------------------------------------------------------------------------
Merge Join  (cost=1.55..83633.87 rows=1000000 width=70)
  Merge Cond: (a.i = b.i)
  ->  Index Scan using a_idx on a  (cost=0.00..34317.36 rows=1000000 ..
  ->  Index Scan using b_idx on b  (cost=0.00..34317.36 rows=1000000 ..
(4 rows)
```

- může být lepší než hash join pokud je setříděné nebo potřebuji setříděné
- v případě třídění pomocí indexu závisí na korelaci index-tabulka
- na rozdíl od hash joinu může mít velmi malou startovací cenu (vnořený index), což je výhodné pokud je třeba jenom pár prvních řádek (LIMIT)

Merge Join

```
DROP INDEX b_idx;
EXPLAIN SELECT * FROM a JOIN b USING (i) ORDER BY i;

                                  QUERY PLAN
-------------------------------------------------------------------------
 Merge Join  (cost=10397.93..15627.93 rows=102582 width=69)
   Merge Cond: (a.i = b.i)
   ->  Index Scan using a_idx on a (cost=0.00..3441.26 rows=100000 ...
   ->  Sort  (cost=10397.93..10654.39 rows=102582 width=36)
         Sort Key: b.i
         ->  Seq Scan on b  (cost=0.00..1859.82 rows=102582 width=36)
(6 rows)
```

- názorná ukázka že při plánování dotazu může hrát roli i "nadřazený" uzel (v tomto případě "ORDER BY")
- zkuste odstranit ORDER BY část – exekuční plán by se měl změnit

Merge Join

- můžeme setkat s tzv. re-scany, pokud joinujeme přes neunikátní sloupce
- typicky 1:M nebo M:N joiny přes cizí klíč(e)
- pokud je toto potřeba, objeví se "Materialize" uzel (tuplestore)

```
CREATE TABLE a AS SELECT i, i/10 j FROM gs(1,1000000) s(i);
CREATE TABLE b AS SELECT i/10 i FROM gs(1,1000000) s(i);

CREATE INDEX a_idx ON a(j);
CREATE INDEX b_idx ON b(i);

EXPLAIN SELECT * FROM a JOIN b ON (a.j = b.i);
                                QUERY PLAN
-------------------------------------------------------------------------
 Merge Join  (cost=0.92..213436.27 rows=10008798 width=12)
   Merge Cond: (a.j = b.i)
   ->  Index Scan using a_j on a  (cost=0.00..30408.36 rows=1000000 ...
   ->  Materialize  (cost=0.00..32908.36 rows=1000000 width=4)
         ->  Index Scan using b_idx on b  (cost=0.00..30408.36 rows=...
(5 rows)
```

- efektivní způsob jak uchovat řádky (tuples), omezeno work_mem

Poddotazy

Korelované a nekorelované, semi/anti-joiny

Korelovaný subselect

```
CREATE TABLE a (id INT PRIMARY KEY);
CREATE TABLE b (id INT PRIMARY KEY, a_id INT REFERENCES a (id),
                val INT, UNIQUE (a_id));

INSERT INTO a SELECT i                FROM gs(1,10000) s(i);
INSERT INTO b SELECT i, i, mod(i,23) FROM gs(1,10000) s(i);

EXPLAIN ANALYZE
  SELECT a.id, (SELECT val FROM b WHERE a_id = a.id) AS val FROM a;

                          QUERY PLAN
-------------------------------------------------------------------------
Seq Scan on a (cost=0.00..82941.20 rows=10000 width=4)
              (actual time=0.023..14.477 rows=10000 loops=1)
 SubPlan 1
   -> Index Scan using b_a_id_key on b (cost=0.00..8.28 rows=1 width=4)
                           (actual time=0.001..0.001 rows=1 loops=10000)
         Index Cond: (a_id = a.id)
 Total runtime: 14.920 ms
(5 rows)
```

- SubPlan kroky jsou prováděny opakovaně (pro každý řádek skenu)

Korelovaný subselect

- často lze efektivně přepsat na join

```
EXPLAIN SELECT a.id, b.val FROM a LEFT JOIN b ON (a.id = b.a_id);
                            QUERY PLAN
------------------------------------------------------------------
 Hash Right Join  (cost=270.00..675.00 rows=10000 width=8)
   Hash Cond: (b.a_id = a.id)
   ->  Seq Scan on b  (cost=0.00..155.00 rows=10000 width=8)
   ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
         ->  Seq Scan on a  (cost=0.00..145.00 rows=10000 width=4)
(5 rows)
```

- výrazně nižší cena oproti ceně vnořeného index scanu (82941.20)
- není úplně ekvivalentní, takže to DB nemůže dělat automaticky
  - jinak se chová k duplicitám v "b" (join nespadne)
- přepis jde použít i na agregační subselecty, např.

```
SELECT a.id, (SELECT SUM(val) FROM b WHERE a_id = a.id) FROM a;

SELECT a.id, SUM(b.val) FROM a LEFT JOIN b ON (a.id = b.a_i)
                                          GROUP BY a.id;
```

Nekorelovaný subselect

```
EXPLAIN SELECT a.id, (SELECT val FROM b LIMIT 1) AS val FROM a;

                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on a  (cost=0.02..145.02 rows=10000 width=4)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.02 rows=1 width=4)
           ->  Seq Scan on b  (cost=0.00..155.00 rows=10000 width=4)
(4 rows)
```

- vyhodnoceno jen jednou na začátku
- přepis na join většinou méně efektivní (náklady na join převažují)

```
EXPLAIN SELECT a.id, x.val FROM a, (SELECT val FROM b LIMIT 1) x;
                            QUERY PLAN
-------------------------------------------------------------------
 Nested Loop  (cost=0.00..245.03 rows=10000 width=8)
   ->  Limit  (cost=0.00..0.02 rows=1 width=4)
         ->  Seq Scan on b  (cost=0.00..155.00 rows=10000 width=4)
   ->  Seq Scan on a  (cost=0.00..145.00 rows=10000 width=4)
(4 rows)
```

EXISTS

```
CREATE TABLE a (id INT PRIMARY KEY);
CREATE TABLE b (id INT PRIMARY KEY);

INSERT INTO a SELECT i FROM gs(1,10000) s(i);
INSERT INTO b SELECT i FROM gs(1,10000) s(i);

SELECT * FROM a WHERE EXISTS (SELECT 1 FROM b WHERE id = a.id);
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Semi Join  (cost=270.00..665.00 rows=10000 width=4)
   Hash Cond: (a.id = b.id)
   ->  Seq Scan on a  (cost=0.00..145.00 rows=10000 width=4)
   ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
         ->  Seq Scan on b  (cost=0.00..145.00 rows=10000 width=4)

SELECT * FROM a WHERE id IN (SELECT id FROM b);
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Semi Join  (cost=270.00..665.00 rows=10000 width=4)
   Hash Cond: (a.id = b.id)
   ->  Seq Scan on a  (cost=0.00..145.00 rows=10000 width=4)
   ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
         ->  Seq Scan on b  (cost=0.00..145.00 rows=10000 width=4)
```

NOT EXISTS

```
SELECT * FROM a WHERE NOT EXISTS (SELECT id FROM b WHERE id = a.id);
                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Anti Join  (cost=270.00..565.00 rows=1 width=4)
   Hash Cond: (a.id = b.id)
   ->  Seq Scan on a  (cost=0.00..145.00 rows=10000 width=4)
   ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
         ->  Seq Scan on b  (cost=0.00..145.00 rows=10000 width=4)

SELECT * FROM a WHERE id NOT IN (SELECT id FROM b);
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on a  (cost=170.00..340.00 rows=5000 width=4)
   Filter: (NOT (hashed SubPlan 1))
   SubPlan 1
     ->  Seq Scan on b  (cost=0.00..145.00 rows=10000 width=4)
```

- Hádanka: Proč se tyto plány liší když pro EXISTS a IN jsou stejné?
- Nápověda: NOT IN (NULL) => NULL
- Úkol: Zkuste místo poddotazu použít pole.

Ukázky dotazů

```
CREATE TABLE foo AS SELECT generate_series(1,1000000) i;
CREATE INDEX ON foo(i);
ANALYZE foo;

EXPLAIN ANALYZE
  SELECT i FROM foo
UNION ALL
  SELECT i FROM foo
ORDER BY 1 LIMIT 100;

 Limit  (cost=0.01..3.31 rows=100 width=4)
        (actual time=0.028..0.078 rows=100 loops=1)
   ->  Result  (cost=0.01..65981.61 rows=2000000 width=4)
               (actual time=0.026..0.064 rows=100 loops=1)
         ->  Merge Append  (cost=0.01..65981.61 rows=2000000 width=4)
                           (actual time=0.026..0.053 rows=100 loops=1)
               Sort Key: public.foo.i
               ->  Index Only Scan using foo_i_idx on foo
                                 (cost=0.00..20490.80 rows=1000000 width=4)
                                 (actual time=0.017..0.021 rows=51 loops=1)
                     Heap Fetches: 0
               ->  Index Only Scan using foo_i_idx on foo
                                 (cost=0.00..20490.80 rows=1000000 width=4)
                                 (actual time=0.007..0.012 rows=50 loops=1)
                     Heap Fetches: 0
 Total runtime: 0.106 ms
```

```
CREATE TABLE foo AS SELECT generate_series(1,1000000) i;
CREATE INDEX ON foo(i);
ANALYZE foo;

EXPLAIN ANALYZE
  SELECT i FROM foo WHERE i IS NOT NULL
UNION ALL
  SELECT i FROM foo WHERE i IS NOT NULL
ORDER BY 1 LIMIT 100;

Limit  (cost=127250.56..127250.81 rows=100 width=4)
       (actual time=1070.799..1070.812 rows=100 loops=1)
   ->  Sort  (cost=127250.56..132250.56 rows=2000000 width=4)
             (actual time=1070.798..1070.804 rows=100 loops=1)
         Sort Key: public.foo.i
         Sort Method: top-N heapsort  Memory: 29kB
         ->  Result  (cost=0.00..50812.00 rows=2000000 width=4)
                          (actual time=0.009..786.806 rows=2000000 loops=1)
               ->  Append  (cost=0.00..50812.00 rows=2000000 width=4)
                          (actual time=0.007..512.201 rows=2000000 loops=1)
                     ->  Seq Scan on foo
                                (cost=0.00..15406.00 rows=1000000 width=4)
                          (actual time=0.007..144.872 rows=1000000 loops=1)
                       Filter: (i IS NOT NULL)
                     ->  Seq Scan on foo
                                (cost=0.00..15406.00 rows=1000000 width=4)
                          (actual time=0.003..139.196 rows=1000000 loops=1)
                       Filter: (i IS NOT NULL)
 Total runtime: 1070.847 ms
```

```
EXPLAIN ANALYZE
SELECT initcap (fullname), initcap(issuer),
       upper(rsymbol), initcap(industry), activity
FROM changes WHERE activity IN (4,5)
                AND mfiled >= (SELECT MAX(mfiled) FROM changes)
ORDER BY shareschange ASC LIMIT 15

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.66..76.91 rows=15 width=98)
        (actual time=5346.850..5366.482 rows=15 loops=1)
   InitPlan 2 (returns $1)
     ->  Result  (cost=0.65..0.66 rows=1 width=0)
                 (actual time=0.076..0.077 rows=1 loops=1)
           InitPlan 1 (returns $0)
             ->  Limit  (cost=0.00..0.65 rows=1 width=4)
                        (actual time=0.063..0.065 rows=1 loops=1)
                   ->  Index Scan Backward using changes_mfiled on changes
                                    (cost=0.00..917481.00 rows=1414912 width=4)
                                    (actual time=0.058..0.058 rows=1 loops=1)
                         Index Cond: (mfiled IS NOT NULL)
   ->  Index Scan using changes_shareschange on changes
                              (cost=0.00..925150.26 rows=181997 width=98)
                              (actual time=5346.846..5366.430 rows=15 loops=1)
         Filter: ((activity = ANY ('{4,5}'::integer[])) AND (mfiled >= $1))
 Total runtime: 5366.578 ms
```

```
EXPLAIN ANALYZE
SELECT initcap (fullname), initcap(issuer),
       upper(rsymbol), initcap(industry), activity
FROM changes WHERE activity IN (4,5)
                AND mfiled >= (SELECT MAX(mfiled) FROM changes)
ORDER BY shareschange DESC LIMIT 15

                                        QUERY PLAN
-----------------------------------------------------------------------------------------
 Limit  (cost=0.66..76.91 rows=15 width=98)
        (actual time=3.167..15.895 rows=15 loops=1)
   InitPlan 2 (returns $1)
     ->  Result  (cost=0.65..0.66 rows=1 width=0)
                 (actual time=0.042..0.044 rows=1 loops=1)
           InitPlan 1 (returns $0)
             ->  Limit  (cost=0.00..0.65 rows=1 width=4)
                        (actual time=0.033..0.035 rows=1 loops=1)
                   ->  Index Scan Backward using changes_mfiled on changes
                                        (cost=0.00..917481.00 rows=1414912 width=4)
                                        (actual time=0.029..0.029 rows=1 loops=1)
                         Index Cond: (mfiled IS NOT NULL)
   ->  Index Scan Backward using changes_shareschange on changes
                                        (cost=0.00..925150.26 rows=181997 width=98)
                                        (actual time=3.161..15.843 rows=15 loops=1)
         Filter: ((activity = ANY ('{4,5}'::integer[])) AND (mfiled >= $1))
 Total runtime: 15.998 ms
```

```
SELECT email.stuff FROM email NATURAL JOIN link_url NATURAL JOIN email_link
                    WHERE machine = 'foo.bar.com';
-------------------------------------------------------------------------------------
Merge Join  (cost=3949462.38..8811048.82 rows=4122698 width=7)
             (actual time=771578.076..777749.755 rows=3 loops=1)
   Merge Cond: (email.message_id = link_url.message_id)
   ->  Index Scan using email_pkey on email  (cost=0.00..4561330.19 rows=79154951 width=11)
                                    (actual time=0.041..540883.445 rows=79078427 loops=1)
   ->  Materialize  (cost=3948986.49..4000520.21 rows=4122698 width=4)
            (actual time=227023.820..227023.823 rows=3 loops=1)
         ->  Sort  (cost=3948986.49..3959293.23 rows=4122698 width=4)
                  (actual time=227023.816..227023.819 rows=3 loops=1)
               Sort Key: link_url.message_id
               Sort Method:  quicksort  Memory: 25kB
               ->  Hash Join  (cost=9681.33..3326899.30 rows=4122698 width=4)
                           (actual time=216443.617..227023.798 rows=3 loops=1)
                     Hash Cond: (link_url.urlid = email_link.urlid)
                     ->  Seq Scan on link_url  (cost=0.00..2574335.33 rows=140331133 width=37)
                                        (actual time=0.013..207980.261 rows=140330592 loops=1)
                     ->  Hash  (cost=9650.62..9650.62 rows=2457 width=33)
                              (actual time=0.074..0.074 rows=1 loops=1)
                           ->  Bitmap Heap Scan on email_link
                                                 (cost=97.10..9650.62 rows=2457 width=33)
                                          (actual time=0.072..0.072 rows=1 loops=1)
                                 Recheck Cond: (hostname = 'foo.bar.com'::text)
                                 ->  Bitmap Index Scan on hostdex
                                                       (cost=0.00..96.49 rows=2457 width=0)
                                                (actual time=0.060..0.060 rows=1 loops=1)
                                       Index Cond: (hostname = 'foo.bar.com'::text)
 Total runtime: 777749.820 ms
(16 rows)
```

```
SELECT email.stuff FROM email NATURAL JOIN link_url NATURAL JOIN email_link
                    WHERE machine = 'foo.bar.com';
-------------------------------------------------------------------------------------
Merge Join  (cost=3949462.38..8811048.82 rows=4122698 width=7)
             (actual time=771578.076..777749.755 rows=3 loops=1)
   Merge Cond: (email.message_id = link_url.message_id)
   ->  Index Scan using email_pkey on email  (cost=0.00..4561330.19 rows=79154951 width=11)
                                    (actual time=0.041..540883.445 rows=79078427 loops=1)
   ->  Materialize  (cost=3948986.49..4000520.21 rows=4122698 width=4)
            (actual time=227023.820..227023.823 rows=3 loops=1)
         ->  Sort  (cost=3948986.49..3959293.23 rows=4122698 width=4)
                  (actual time=227023.816..227023.819 rows=3 loops=1)
               Sort Key: link_url.message_id
               Sort Method:  quicksort  Memory: 25kB
               ->  Hash Join  (cost=9681.33..3326899.30 rows=4122698 width=4)
                           (actual time=216443.617..227023.798 rows=3 loops=1)
                     Hash Cond: (link_url.urlid = email_link.urlid)
                     ->  Seq Scan on link_url  (cost=0.00..2574335.33 rows=140331133 width=37)
                                    (actual time=0.013..207980.261 rows=140330592 loops=1)
                     ->  Hash  (cost=9650.62..9650.62 rows=2457 width=33)
                            (actual time=0.074..0.074 rows=1 loops=1)
                          ->  Bitmap Heap Scan on email_link
                                                (cost=97.10..9650.62 rows=2457 width=33)
                                         (actual time=0.072..0.072 rows=1 loops=1)
                                Recheck Cond: (hostname = 'foo.bar.com'::text)
                                ->  Bitmap Index Scan on hostdex
                                                      (cost=0.00..96.49 rows=2457 width=0)
                                               (actual time=0.060..0.060 rows=1 loops=1)
                                      Index Cond: (hostname = 'foo.bar.com'::text)
 Total runtime: 777749.820 ms
(16 rows)
```

```
EXPLAIN ANALYZE SELECT * FROM parent ORDER BY id DESC LIMIT 100;

                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Limit  (cost=105288.65..105288.90 rows=100 width=4)
    (actual time=868.998..869.010 rows=100 loops=1)
   ->  Sort  (cost=105288.65..110288.65 rows=2000002 width=4)
          (actual time=868.996..869.002 rows=100 loops=1)
         Sort Key: public.parent.id
         Sort Method: top-N heapsort  Memory: 29kB
         ->  Result  (cost=0.00..28850.01 rows=2000002 width=4)
               (actual time=0.007..442.538 rows=2000001 loops=1)
                ->  Append  (cost=0.00..28850.01 rows=2000002 width=4)
                     (actual time=0.006..259.906 rows=2000001 loops=1)
                      ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=4)
                                   (actual time=0.001..0.001 rows=0 loops=1)
                      ->  Seq Scan on child_1 parent
                                  (cost=0.00..14425.00 rows=1000000 width=4)
                               (actual time=0.005..70.153 rows=1000000 loops=1)
                      ->  Seq Scan on child_2 parent
                                  (cost=0.00..14425.01 rows=1000001 width=4)
                               (actual time=0.005..70.757 rows=1000001 loops=1)
 Total runtime: 869.032 ms
(10 rows)
```

```
EXPLAIN ANALYZE SELECT * FROM parent ORDER BY id DESC LIMIT 100;

                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Limit  (cost=105288.65..105288.90 rows=100 width=4)
    (actual time=868.998..869.010 rows=100 loops=1)
   ->  Sort  (cost=105288.65..110288.65 rows=2000002 width=4)
          (actual time=868.996..869.002 rows=100 loops=1)
         Sort Key: public.parent.id
         Sort Method: top-N heapsort  Memory: 29kB
         ->  Result  (cost=0.00..28850.01 rows=2000002 width=4)
               (actual time=0.007..442.538 rows=2000001 loops=1)
               ->  Append  (cost=0.00..28850.01 rows=2000002 width=4)
                     (actual time=0.006..259.906 rows=2000001 loops=1)
                     ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=4)
                                 (actual time=0.001..0.001 rows=0 loops=1)
                     ->  Seq Scan on child_1 parent
                                 (cost=0.00..14425.00 rows=1000000 width=4)
                               (actual time=0.005..70.153 rows=1000000 loops=1)
                     ->  Seq Scan on child_2 parent
                                 (cost=0.00..14425.01 rows=1000001 width=4)
                               (actual time=0.005..70.757 rows=1000001 loops=1)
 Total runtime: 869.032 ms
(10 rows)
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
GroupAggregate  (cost=5533840.89..11602495531.58 rows=1 width=16)
   CTE subQuery_1
     ->  Hash Join  (cost=40901.65..1382282.90 rows=10153012 width=9)
           Hash Cond: (public.f_order.orderid_id = public.f_ordersummary.id)
           ->  Seq Scan on f_order  (cost=0.00..1108961.30 rows=10550730 width=13)
           ->  Hash  (cost=31005.97..31005.97 rows=791654 width=4)
                 ->  Seq Scan on f_ordersummary  (cost=0.00..31005.97 rows=791654 width=4)
                       Filter: (orderstatus_id <> ALL ('{15,86406,86407,86412}'::integer[]))
   CTE subQuery_0
     ->  Hash Join  (cost=40901.65..1382282.90 rows=10153012 width=8)
           Hash Cond: (public.f_order.orderid_id = public.f_ordersummary.id)
           ->  Seq Scan on f_order  (cost=0.00..1108961.30 rows=10550730 width=12)
           ->  Hash  (cost=31005.97..31005.97 rows=791654 width=4)
                 ->  Seq Scan on f_ordersummary  (cost=0.00..31005.97 rows=791654 width=4)
                       Filter: (orderstatus_id <> ALL ('{15,86406,86407,86412}'::integer[]))
   ->  Merge Full Join  (cost=2769275.09..7734093990.56 rows=515418263361 width=16)
         Merge Cond: ("subQuery_1".pk = "subQuery_0".pk)
         ->  Sort  (cost=1384637.54..1410020.07 rows=10153012 width=12)
               Sort Key: "subQuery_1".pk
               ->  CTE Scan on "subQuery_1"  (cost=0.00..203060.24 rows=10153012 width=12)
         ->  Sort  (cost=1384637.54..1410020.07 rows=10153012 width=12)
               Sort Key: "subQuery_0".pk
               ->  CTE Scan on "subQuery_0"  (cost=0.00..203060.24 rows=10153012 width=12)
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
GroupAggregate  (cost=5533840.89..11602495531.58 rows=1 width=16)
   CTE subQuery_1
     ->  Hash Join  (cost=40901.65..1382282.90 rows=10153012 width=9)
           Hash Cond: (public.f_order.orderid_id = public.f_ordersummary.id)
           ->  Seq Scan on f_order  (cost=0.00..1108961.30 rows=10550730 width=13)
           ->  Hash  (cost=31005.97..31005.97 rows=791654 width=4)
                 ->  Seq Scan on f_ordersummary  (cost=0.00..31005.97 rows=791654 width=4)
                       Filter: (orderstatus_id <> ALL ('{15,86406,86407,86412}'::integer[]))
   CTE subQuery_0
     ->  Hash Join  (cost=40901.65..1382282.90 rows=10153012 width=8)
           Hash Cond: (public.f_order.orderid_id = public.f_ordersummary.id)
           ->  Seq Scan on f_order  (cost=0.00..1108961.30 rows=10550730 width=12)
           ->  Hash  (cost=31005.97..31005.97 rows=791654 width=4)
                 ->  Seq Scan on f_ordersummary  (cost=0.00..31005.97 rows=791654 width=4)
                       Filter: (orderstatus_id <> ALL ('{15,86406,86407,86412}'::integer[]))
   ->  Merge Full Join  (cost=2769275.09..7734093990.56 rows=515418263361 width=16)
         Merge Cond: ("subQuery_1".pk = "subQuery_0".pk)
         ->  Sort  (cost=1384637.54..1410020.07 rows=10153012 width=12)
               Sort Key: "subQuery_1".pk
               ->  CTE Scan on "subQuery_1"  (cost=0.00..203060.24 rows=10153012 width=12)
         ->  Sort  (cost=1384637.54..1410020.07 rows=10153012 width=12)
               Sort Key: "subQuery_0".pk
               ->  CTE Scan on "subQuery_0"  (cost=0.00..203060.24 rows=10153012 width=12)
```

```
CREATE TABLE toasted (id SERIAL, val TEXT);
INSERT INTO toasted SELECT i, REPEAT(MD5(i::text),80)
                      FROM generate_series(1,1000000) s(i);

EXPLAIN ANALYZE SELECT id, LENGTH(val) FROM toasted;

                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on toasted  (cost=0.00..25834.00 rows=1000000 width=36)
                (actual time=0.018..8060.214 rows=1000000 loops=1)
 Total runtime: 8088.929 ms
(2 rows)

EXPLAIN ANALYZE SELECT id, LENGTH(id::text) FROM toasted;

                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on toasted  (cost=0.00..30834.00 rows=1000000 width=4)
                 (actual time=0.011..218.352 rows=1000000 loops=1)
 Total runtime: 246.334 ms
(2 rows)
```