



# Čtení exekučních plánů

Prague PostgreSQL Developer Day 2015 / 11.2.2015

**Tomáš Vondra**

[tomas.vondra@2ndquadrant.com](mailto:tomas.vondra@2ndquadrant.com) / [tomas@pgaddict.com](mailto:tomas@pgaddict.com)

© 2015 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

- Bez pochopení základních principů jak plánování funguje - co vše musí databáze zvážit, na základě jakých informací a jakým způsobem z plánů vybírá ten správný, by pro vás interpretace exekučních plánů (a zejména pochopení kde je problém, protože to je důvod proč se exekuční plány čtou) daleko obtížnější.
- Stejně tak je potřebné porozumět technickým nástrojům které jsou k dispozici, zejména se jedná o příkazy EXPLAIN a EXPLAIN ANALYZE, ale i případné další. Opět je třeba vědět jaká slabá místa tyto nástroje mají, případně jakými chybami je jejich použití zatíženo.
- Následně si ukážeme varianty alespoň základních uzelů (skeny tabulek, joiny a některé další jako např. agregace), tak jak jsou implementovány v PostgreSQL, stručně si shrneme jak jsou implementovány a v jakých případech jsou efektivní, kdy ne, za jakých podmínek selhávají a jak je to vidět v exekučním plánu (pokud vůbec).
- Samozřejmě není lepšího zdroje poučení než příklady skutečných problematických dotazů z praxe, takže si jich několik projdeme a pokusíme se určit kde došlo k chybě a případně jak ho napravit.

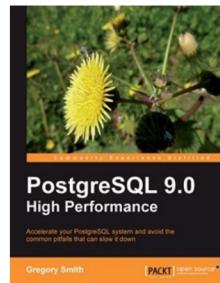
# Zdroje

## PostgreSQL dokumentace

- EXPLAIN  
<http://www.postgresql.org/docs/current/static/sql-explain.html>
- Using EXPLAIN  
<http://www.postgresql.org/docs/current/static/using-explain.html>
- Row Estimation examples  
<http://www.postgresql.org/docs/devel/static/row-estimation-examples.html>

## PostgreSQL 9.0 High Performance

- Query optimization (p. 233 - 296)
  - planning basics, EXPLAIN usage
  - processing nodes
  - statistics
  - planning parameter



- Bez pochopení základních principů jak plánování funguje - co vše musí databáze zvážit, na základě jakých informací a jakým způsobem z plánů vybírá ten správný, by pro vás interpretace exekučních plánů (a zejména pochopení kde je problém, protože to je důvod proč se exekuční plány čtou) daleko obtížnější.
- Stejně tak je potřebné porozumět technickým nástrojům které jsou k dispozici, zejména se jedná o příkazy EXPLAIN a EXPLAIN ANALYZE, ale i případné další. Opět je třeba vědět jaká slabá místa tyto nástroje mají, případně jakými chybami je jejich použití zatíženo.
- Následně si ukážeme varianty alespoň základních uzelů (skeny tabulek, joiny a některé další jako např. agregace), tak jak jsou implementovány v PostgreSQL, stručně si shrneme jak jsou implementovány a v jakých případech jsou efektivní, kdy ne, za jakých podmínek selhávají a jak je to vidět v exekučním plánu (pokud vůbec).
- Samozřejmě není lepšího zdroje poučení než příklady skutečných problematických dotazů z praxe, takže si jich několik projdeme a pokusíme se určit kde došlo k chybě a případně jak ho napravit.

# 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

- To že vyhodnocení dotazu je zodpovědností databáze neznamená že je soběstačná. Pokud nevhodně navrhnete databázové schéma, nevytvoříte indexy nebo je naopak vytvoříte tam kde nejsou efektivní, nebo SQL dotazy zformulujete nevhodným způsobem, databáze si s tím neporadí (to ostatně platí pro libovolný typ databáze, nejen relační, potažmo na všechny IT systémy obecně).
- Plánovače (a plánovač implementovaný v PostgreSQL není výjimkou) mají různá implementační omezení a jsou založeny na zjednodušených statistických modelech které mají omezené schopnosti jaksi z principu, neboť se rozhodují na základě statistik.
- Pokud budete schopni postup plánovače interpretovat, budete mít možnost analyzovat různé výkonnostní problémy (to je asi primární důvod zájmu o exekuční plány), budete mít možnost lépe navrhovat databázové schéma a psát efektivnější SQL dotazy. Případně budete moci aktuální plánovač vylepšit ;-)

# 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)

- Optimalizace spočívá v hledání takového exekučního plánu který minimalizuje "cenovou funkci" - pokud víte co je lineární či nelineární programování, jedná se o stejný princip, s tím že vstupem funkce je plán. Všechny zvažované plány samozřejmě musí mít vlastnost že vracejí správný výsledek na položený dotaz.
- Korelace ceny dotazu a doby jeho vyhodnocení je ideál, kterého se v praxi víceméně dosáhnout nedá, a to ze dvou hlavních důvodů. Zaprvé model výpočtu ceny je značně zjednodušený a nezachycuje všechny možné vlivy, zadruhé vyhodnocení dotazu je často ovlivňováno vnějšími okolnostmi jejichž vliv nelze předem s jistotou predikovat - např. efekty cachování, vliv dotazů běžících současně, rozdílné charakteristiky hardware apod. Samozřejmě by bylo možné všechno toto detailně analyzovat a profilovat, ale systém by se stal natolik složitým že by ho nebylo možno v reálném čase vyladit (pro daný stroj).
- Se stabilitou je to o něco jednodušší - cena je konstruována (víceméně) jako spojitá funkce, a nestává se že změnou některého vstupu o 1% se cena změní řádově více. Díky tomu není nutno základní parametry ceny (o kterých je řeč na následujícím sladu) ladit zcela přesně protože víme že malé odchylky mají malý vliv na cenu. Jak uvidíme dále, zdroje hlavních chyb jsou zcela jinde (většinou ustřelených odhadech).
- Výše uvedené platí pro tzv. "cost-based" plánovače, které vychází ze statistik a hodí se pro dynamicky se měnící data (objem, statistické rozložení). Druhým typem jsou "rule-based" plánovače, založené na sadě statických pravidel - ty jsou vhodné pro statická data kde jde pravidla určit předem. Hintování dotazů (v PostgreSQL nedostupné) je víceméně zanášení pravidel do cost-based prostředí.

## 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_radek +
       1000000 * cena_where_podminky
```

- Uvedený příklad je samozřejmě velmi triviální, ale dostatečně ilustruje princip výpočtu ceny.
- Cena je velmi dobrý technický parametr pro plánovač / optimalizátor, ale pro člověka se jedná o dost nesrozumitelnou hodnotu - těžko se z ní určuje zda je plán dobrý nebo špatný.
- V následujících částech uvidíme plány a odhady cen pro mnoho různých dotazů, nicméně odhady cen detailně pitvat nebudeme (rozhodně ne tak že bychom ceny počítali).
- Často lze narazit na nepochopení jak funguje MVCC a na to co znamená "počet řádek" - multigenerační architektura v PostgreSQL funguje tak že při DELETE řádek nedochází k jejich okamžitému odstranění z tabulky, a dotazy s nimi stále musí pracovat. Např. pokud z tabulky s milionem řádek polovinu smažete, dotazu stále budou muset zpracovávat celý milion řádek, a to až do chvíle kdy dojde k VACUUM. Obdobně pro datové stránky a VACUUM FULL.
- Pokud vás odhad ceny zajímá, podrobnosti najdete ve velice přehledné podobě v souboru src/backend/optimizer/path/costsize.c

## 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

- To jaké statistiky o tabulce jsou k dispozici uvidíme podrobněji za chvíliku, zatím stačí že jsou k dispozici informace o velikosti tabulek (na disku, počet řádků) apod.
- Cost proměnné zachycují základní parametry, cena dalších operací (např. vytváření dočasných souborů apod.) se z nich odvozují. Uvedené jsou výchozí a vesměs se jedná o časem osvědčené hodnoty, nicméně lze očekávat že pro specifický hardware se mohou měnit. To platí zejména o random\_page\_cost např. na SSD discích (neznám případy kdy by bylo třeba nějak měnit některé z CPU proměnných).
- Z řady vybočuje effective\_cache\_size, protože to neurčuje cenu ale jedná se pouze o nápočtu kolik cache (shared buffers + filesystem cache) v systému celkem je, a používá se pro odhad efektů cache v některých plánech (např. Bitmap Index Scan). Doporučení jak správně nastavit najdete např. na [wiki.postgresql.org](http://wiki.postgresql.org) (běžně se uvádí 60%-75% RAM).
- Je dobrým zvykem nijak neměnit seq\_page\_cost a ponechat si ji jako referenční hodnotu - na cenu pak jde nahlížet tak trochu jako násobky oproti času sekvenčnímu čtení jedné stránky (ale neočekávejte nějakou velkou přesnost).
- Z hodnot je také vidět dávná zkušenosť že I/O operace v databázích dominují, i když toto se v poslední době mění a není problém narazit na databázi s CPU bottleneckem.
- Mnoho lidí si také myslí že díky SSD diskům mohou nastavit random\_page\_cost = 1.0 - prosím, nedělejte to, není to pravda. Ani SSD disky nemají stejně rychlé náhodné a sekvenční I/O, s náhodným I/O je spojena další režie (např. 1.5 je rozumnější hodnota).

## 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 sířka 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

- pg\_stats je ve skutečnosti jenom "pohled" nad daleko složitějším katalogem pg\_statistic, ale tam stejně nenajdete nic člověku srozumitelného.
- Z těchto velmi omezených statistik se odvozují odhady počtu řádek (např. selektivita podmínek), počty skupin v agregaci, apod.
- Hodnoty statistik nejsou přesné, ale již samy o sobě jsou odhadem získáváným z náhodného vzorku tabulky (několik tisíc řádek) a stává se že jejich hodnoty nejsou dostatečně věrným zachycením skutečnosti - např. pokud je statistické rozdělení nějak velmi podivně vychýleno, pokud přesnost statistiky není dostačující apod.
- Přesnost statistik lze zvýšit pomocí proměnné "statistics target" která určuje kolik hodnot se bude uchovávat (v MVC poli, v histogramu apod.). a to buď globálně nebo jenom pro konkrétní sloupec (to raději, je s tím spojen overhead).
- ALTER TABLE t ALTER COLUMN c SET STATISTICS 10000;
- Notoricky problematický sloupec je n\_distinct - jeho "ustřelení" většinou nejde opravit zvýšením přesnosti statistik, ale je nutno to provést ručně pomocí ALTER TABLE ... ALTER COLUMN ... SET (n\_distinct = N)

# 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 exponenciálně
  - řešení jen hrubou silou není dostačující

# EXPLAIN

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

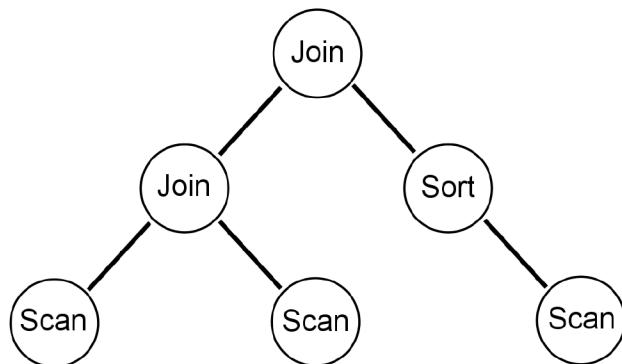
```
EXPLAIN SELECT SUM(a.id) FROM a,b WHERE a.id = b.id;
```

## 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



- Exekuční plán funguje tak trochu jako velký stromový iterátor, tj. objekt s metodou "next()", složený z malých iterátorů (jednotlivých uzlů).
- Exekutor (komponenta PostgreSQL provádějící vyhodnocení dotazu) si na vrchním uzlu řekně o další řádek, a tento uzel si následně říká o další řádky podřízeným uzlům, až tyto požadavky nakonec "propadnou" až na úroveň fyzických tabulek.
- Je to značně zjednodušený pohled, uzly implementují i různé další metody, ale zhruba tak to funguje.
- Některé uzly umí řádky "streamovat" tj. průběžně získávat a přeposílat dál (např. sekvenční nebo index scan, třídění pomocí indexu apod.), některé uzly musí nejdříve zpracovat všechny řádky z podřazených uzlů a až pak mohou vrátit první výsledek (např. tradiční třídění nebo agregace přes hash tabulku). I toto je věc kterou plánovač musí při výběru plánu zvažovat.

# 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

- Všimněte si první dvojice čísel v závorce - jedná se o tzv. "startup" a "total" cenu. První udává cenu kterou je nutno "zaplatit" do vyprodukovaní první řádky, druhé udává celkovou cenu na vyhodnocení dané části plánu až do vyprodukovaní poslední řádky.

Např. u operace “Hash Join” je uvedeno 27.50..56.25 – to znamená že získání prvního řádku bude stát 27.50, a kompletní vyhodnocení 56.25.

- V závorce jsou dále odhady počtu řádek produkovaných danou operací, a průměrná šířka řádku (tak aby bylo možno spočítat nároky na paměť apod.).

# 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
```

Kompletní syntaxe EXPLAIN ANALYZE je takováto

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

kde "option" může být jedno z:

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
BUFFERS [ boolean ]
TIMING [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

a význam těchto voleb je

- VERBOSE – podrobnější informace (kvalifikovaná jména objektů apod.)
- COSTS – ceny (startup/total)
- BUFFERS – informace o hit/miss při přístupu do shared\_buffers
- TIMING – umožňuje vypnout měření času (viz. následující slide)
- FORMAT – alternativní formáty (např.) pro strojové zpracování

## 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  0.0000%
     4:         39  0.00003%
     2:    2999907  2.34869%
     1: 124726830 97.65128%
```

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

- Pokud váš systém uvedeným testem neprochází (resp. má pomalé hodiny), můžete zkontrolovat zda není chybně nakonfigurovaný – jak to provést např. na Linuxu najdete na <http://www.postgresql.org/docs-devel/static/pgtesttiming.html>
- Pomalé hodiny neznamenají nutně že EXPLAIN ANALYZE dává nepoužitelné výsledky, ale je třeba na to při analýze a ověřovat např. s TIMING OFF.

## 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 proporcionalním 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 naprostě v pořádku - za daných podmínek optimální
  - např. vám tam můžete chybět index nebo ho nedostat kvůli formulaci podmínky, apod.

- Výše uvedené kroky jsou ve víceméně doporučovaném pořadí, ale není to dogma. V reálných případech se většinou na problematických uzlech projevuje více než jeden z výše uvedených bodů (ustřelené odhady většinou způsobí pomalost příslušného uzlu, apod.)
- Je zřejmé že středobodem všeho jsou dostatečně přesné odhadы počtu řádek - pokud se plánovač v tomto výrazně splete, je dost nepravděpodobné že by došel k dobrým odhadům cen a tedy i vybral dobrý plán. Naopak pokud jsou odhady počtu řádek dostatečně přesné, je dost pravděpodobné že plánovač vybere za daných podmínek dobrý exekuční plán (a pak přicházejí na řadu další dva kroky).
- Nesnažte se zbytečně dešifrovat složité exekuční plány - použijte <http://explain.depesz.com> nebo jiný nástroj který vám plány převede do vizuální podoby a případně i zvýrazní problematické uzly.

# 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.
- Bud' 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 podmíinku
    – 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)
```

- 333333 (33%) je výchozí odhad pro podobně neodhadnutelné podmínky (ostatně ta podmínka je nesmysl, pokud je "i" reálné číslo)
- PostgreSQL neumí provést inverzi funkcí, to musíte udělat vy (pokud to vůbec jde), například místo mocniny aplikovat odmocninu apod.

# 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).

- S korelovanými sloupci je potíž – estimátor PostgreSQL si s tímto zatím neumí poradit a hinty podporovány nejsou (a nebudou). Jiné databáze vesměs řeší přávě přes hinty a/nebo explicitní volbou sběru statistik o závislostech ve skupině sloupců.
- V podstatě jediný trik jak toto ofixovať v SQL je použití OFFSET, což zabrání “flatteningu” a donutí PostgreSQL odhadnout podmínky samostatně.

```
EXPLAIN ANALYZE SELECT * FROM (SELECT * FROM a WHERE a.i < 1000
OFFSET 0) foo WHERE j < 1000;
```

# 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 plánují při PREPARE
- nepojmenované prepared statements (v uložených procedurách) se plánují 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)

- Typické "nepochopitelné" chování kdy dotaz puštěný v psql proběhne rychle, ale v okamžiku vložení do PL/pgSQL funkce najednou běží daleko pomaleji.
- Pojmenované prepared statements se plánují v okamžiku kdy ještě nejsou známy žádné hodnoty - plánuje se na běžné hodnoty (podle statistik).
- Nepojmenované prepared statements (tj. nativní SQL uvedené v PL/pgSQL funkcích) se naplánují s prvními hodnotami - dost nepředvídatelné chování.
- Ale jak už bylo zmíněno, 9.2 toto řeší kontrolou plánů (ale není to samozřejmě zdarma).

# 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)

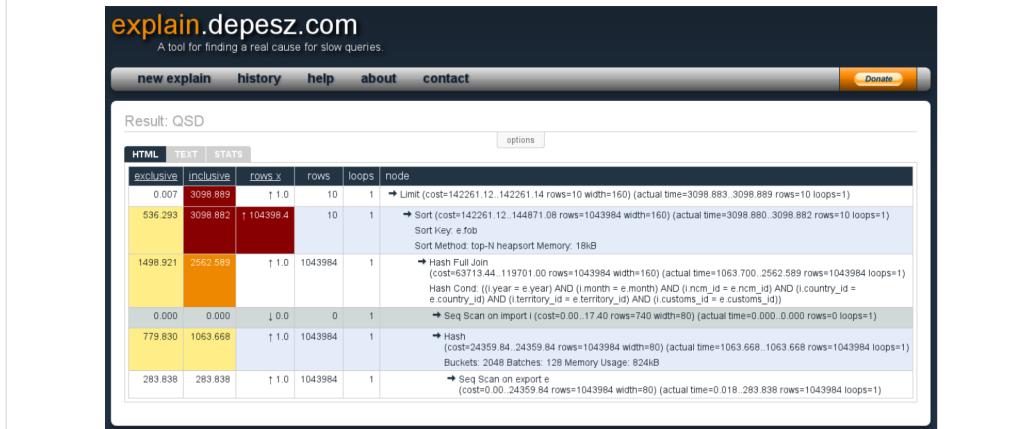
EXPLAIN SELECT * FROM a a1 JOIN a a2 ON (a1.i = a2.i-1);
          QUERY PLAN
-----
Hash Join  (cost=2693.00..6886.00 rows=100000 width=8)
Hash Cond: ((a2.i - 1) = a1.i)
-> Seq Scan on a a2  (cost=0.00..1443.00 rows=100000 width=4)
-> Hash   (cost=1443.00..1443.00 rows=100000 width=4)
      -> Seq Scan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

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

- Kardinality joinů, to je oříšek. Optimalizátor spoustu "zřejmých" věcí nevidí - například nám je jasné že join sudých a lichých čísel je prázdná množina, ale joiny přes výrazy prostě přesnější nebudou.

# 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 bude nepredikovatelně (například je pomalý jen v noci)
- při následném ručním průzkumu se všechno zdá naprostě OK - duchařina
- tento modul vám umožní exekuční plán odchytit právě když bude
- 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 skutečnosti nevypíná ale pouze výrazně znevýhodňuje

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>• enable_bitmapscan</li><li>• enable_indexscan</li><li>• enable_seqscan</li><li>• enable_tidscan</li><li>• enable_indexonlyscan</li><li>• enable_hashjoin</li></ul> | <ul style="list-style-type: none"><li>• enable_mergejoin</li><li>• enable_nestloop</li><li>• enable_hashagg</li><li>• enable_material</li><li>• enable_sort</li></ul> |
|---|---|

- Použitelné hlavně pro analýzu problému, maximálně pro konkrétní dotazy. Rozhodně se nedoporučuje nastavovat jako výchozí hodnotu pro celý server.
- Lze samozřejmě nastavit i v rámci jedné “session”.

## Způsoby přístupu k tabulkám

V této části jsou probrány základní způsoby jak lze přistupovat k relacím, tzv. skenům. Spadají sem jak tradiční způsoby používané pro přístup k fyzickým tabulkám, tj. zejména "sequential scan", "index scan" a "bitmap index scan," ale také způsoby používané pro přístup k tabulkám generovaným ("function scan" a "foreign scan") a další jako např. "cte scan."

# Způsoby přístupu k tabulkám

- Sequential Scan
- Index Scan
- Index Only Scan
- Bitmap Index Scan
- Function Scan
- *CTE Scan*
- *TID Scan*
- *Foreign Scan*
- ...

# Sequential Scan

- nejjednodušší možný sken - sekvenčně čte tabulkou
- 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

- Malou tabulkou je méněna tabulka o několika (desítkách) blocích, tj. potenciálně až tisících řádek. Pro takto malé tabulky je čtení dat z indexů a následné (náhodné) čtení dat z tabulky velmi neefektivní a je jednodušší přečíst celou tabulkou.
- Obdobný princip funguje u velkých tabulek ze kterých je nutno číst více než několik málo jednotek procent - overhead způsobený náhodným I/O je natolik velký že je jednodušší přečíst celou tabulkou.
- Tím že nešpiní shared buffers je méněno že při sekvenčním skenu velké tabulky nejsou ze shared buffers vytlačovány bloky jiných relací. Dříve se zhusta stávalo že cache se dostala do stavu kdy bylo nacachováno právě to co dává největší smysl (a databáze tak dosáhla v jistém smyslu optimálního výkonu) a následně došlo k sekvenčnímu skenu velké tabulky která z paměti vytlačila vše ostatní a "zahřívání" mohlo začít na novo. Od v. 8.3 je používán kruhový buffer takže toto už se nestává.
- Obdobně pokud před verzí 8.3 bylo spuštěno několik sekvenčních skenů stejně tabulky paralelně, každý četl tabulku samostatě což neúměrně zatěžovalo I/O. Od verze 8.3 je možná jejich synchronizace, tj. později spuštěné sekvenční skeny poznají že již stejný sekvenční sken běží a připojí se k němu (např. v půlce). To ale znamená že data tabulky můžete dostávat v jiném pořadí než jak je uložena na disku.

# 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

- Indexy jsou založeny na rychlém přístupu k řádkům tabulky podle hodnoty sloupce / kombinace sloupců / podmínky.
- Tradiční jsou b-tree indexy (stromové), PostgreSQL umí např. hash indexy nebo GIN/GiST indexy (ale to budeme opomíjet).
- Indexy způsobují náhodné I/O při přístupu k tabulce - díky tomu jsou efektivní jen pro přístup k malému procentu tabulky (závisí i na tom jak je index s tabulkou korelovaný).
- Zkuste formulovat dotazy s různou selektivitou a sledujte jak se mění cena plánu.
- Zkuste pomocí "SET enable\_seqscan = off" vypnout sekvenční sken a následně přečtěte celou tabulku pomocí indexu. Jak se změnil čas? (můžete ho změřit pomocí "\timing on").
- Stránky tabulky mohou být čteny opakovaně, podle toho jak korelovaný s tabulkou index je. Zkuste tabulku pomocí indexu načíst tabulku (ORDER BY) setříděnou podle sloupce indexu. Zkuste to s indexem korelovaným a nekorelovaným s tabulkou.
- Indexů může být více - může být více nezávisle indexovaných podmínek, nebo jedna podmínka s několika indexy (mohou být nad více sloupci). Plánovač se snaží zvolit ten nejfektivnější index s ohledem na selektivitu příslušné podmínky, velikost indexu apod.
- To který index byl vybrán je zřejmé (using jméno\_indexu), podmínka (sloupce) použitá pro vyhledávání v indexu je uvedena jako "Index Cond(ition)" a podmínky vyhodnocené až na výsledku (před předáním dalšímu skenu) jsou uvedeny jako "Filter."

# 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);
INSERT 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)
```

- Index only scan přichází v úvahu jen tam kde byl předtím možný index scan, a kde je v indexu vše potřebné (často např. některých joinech, v EXISTS klauzulích apod.).
- Index only neznamená neznamená že se do tabulky občas sáhnout nemůže - visibility map nemusí obsahovat aktuální údaje pro všechny stránky.
- To neznamená že index only scan není možné ke čtení tabulky v daném pořadí, ale kvůli možným splitům není možné číst index sekvenčně tak jak je uložen na disku.
- Jediný typ skenu který produkuje setříděný výstup.

## 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

- V podstatě to nejlepší z obou světů - selektivita indexu, sekvenční přístup k tabulce. Cenou je pomalý start.
- Bitmap index nejdříve z indexu sestaví bitmapu která říká pro které datové stránky je podmínka splněna (alespoň pro jeden řádek) a následně pomocí této bitmapy z tabulky vyčte pouze stránky které jsou potřeba - právě načtení řádek z tabulky (heap) je úkolem posledního kroku "Bitmap Heap Scan."
- Díky práci s bitmapami se oproti tabulce jedná vesměs o sekvenční I/O a tudíž je (většinou) poměrně rychlé a méně zatěžující než náhodné.
- Bitmapy jsou udržovány na úrovni datových stránek, protože jejich načtení je znatelně časově náročnější než následná iterace přes položky na stránce. Současně bitmapy mohou být daleko menší a odpadají i další implementační obtíže související s tím jak PostgreSQL interně funguje.
- Díky práci s bitmapami na úrovni celých datových stránek může být ale načtena i řádka která podmínce neodpovídá - právě odfiltrování těchto "nechtěně načtených" řádek je úkolem předposledního kroku "Recheck Cond" který "kontroluje" platnost podmínek.

# 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)
```

- Jak je vidět z plánu, PostgreSQL může vygenerovat bitmapy z více indexů (nebo i více bitmap z jednoho) a následně je spojovat pomocí logických spojek. To které bitmapy se budou generovat opět rozhoduje optimalizátor na základě odhadu ceny.
- Zkuste měnit hodnoty v podmírkách a sledujte jak se budou měnit rozhodnutí které bitmapy sestavit a které nikoliv.
- Často se používá pro dotazy zahrnující IN(pole) nebo ANY(pole).

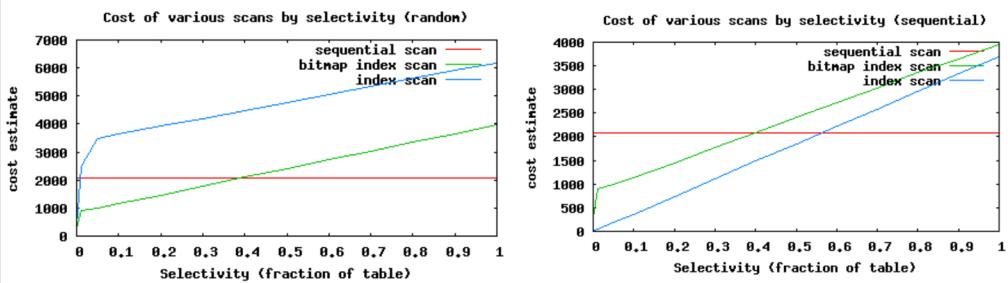
# Srovnání skenů

- vezměme tabulku (1000000 integerů v náhodném pořadí)
- sledujme cenu 3 základních plánů pro podmínu 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);
```



- Je to hodně umělý a zjednodušený příklad, ale celkem dobře ilustruje realitu. Pro podmínky s velkou selektivitou je nejlepší prostý index scan. Jak selektivita klesá (tj. více řádek odpovídá podmínce), začne v jednu chvíli být výhodnější bitmap scan, který je následně přebit sekvenčním skenem.
- To kde k těmto protnutím dojde záleží na velikosti tabulky, korelace slouců/indexů, konkrétních podmínek atd.

# Function Scan

- set-returning-functions (SRF) - funkce vracející tabulkou
- ceny a počty řádek jsou konstanty, dané při komplikaci
- nepřesné odhadování 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;
```

- SRF (Set-Returning Functions) jsou oblíbené ale bohužel neumožňují zasahovat do procesu plánování dotazu - nemohou nijak předávat informace o tom kolik řádek vrátí apod. takže plánovač musí vycházet z konstant.
- V závislosti na implementaci funkce se hodnoty budou čítat postupně a nikde v paměti se neštosují (typicky funkce implementované přímo v C), a nebo se nejdříve vyhodnotí kompletně funkce a výsledek se uloží do “tuple store” (např. PL/pgSQL).

## 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;
```

- opakování 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)
```

- nevyhodnocuje se "na začátku" ale průběžně

- CTE znamená "Common Table Expression" tj. "Společný Tabulkový Výraz" a označuje relaci (výsledek dotazu) odkazovanou na více místech dotazu.
- Ve výše uvedeném příkladě je použití CTE celkem zbytečné - CTE se hodí zejména pokud je daná tabulka používána opakovaně na více místech (tj. například v joinech které jsou probírány dále, v semijoinech/antijoinech apod.).
- Velmi pěkné je použití CTE pro rekurzivní dotazy (např. výpis stromové struktury položek v tabulce přes parent-child vztahy).
- CTE ukládají data do tzv. "tuplestore" tak aby je bylo možno efektivně a nezávisle číst z více míst exekučního plánu (tam kde je CTE referencováno).
- CTE dotaz vyhodnocuje (a do tuplestore ukládá) uzel který je nejvíce napřed, tzv. "leader" - ostatní uzly jsou někde za ním a čtou data z tuplestore. Může se stát že ho předezenou, v tom případě se stávají leaderem a přebírají vyhodnocování dotazu.
- Data mohou zabrat až "work\_mem" v paměti, poté se začnou sypat na disk (a to je samozřejmě značný performance hit).
- Alternativou k CTE jsou buď pojmenované poddotazy (ve FROM části), se kterými není spojen overhead zápisu na disk, nebo tradiční TEMPORARY tabulky (na kterých lze vytvářet indexy, sbírat statistiky apod.).
- CTE nejsou aliasy, mají daleko hlubší (positivní i negativní) důsledky.

## 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 tabulkou, za určitých podmínek může alokovat hodně paměti (výběr metody nelze za běhu měnit)

- Variant "aggregate" se používá pokud dotaz nevyžaduje údržbu informace o několika skupinách, tj. pokud je SQL dotaz zadán bez GROUP BY. Databázi stačí udržovat jeden řádek s výsledkem, a průběžně ho aktualizovat, takže se jedná o paměťově velice efektivní záležitost.
- Group Aggregate se používá v situacích kdy vstup aggregace je setříděný nebo pokud je požadován setříděný výstup, případně pokud je relace příliš velká než aby se vešla do hash tabulky (omezené velikostně na work\_mem). Fungování si lze představit tak že vstupní relaci setříďte podle GROUP BY klíčů, a při čtení reagujete na změny kteréhokoliv klíčového sloupce vypsáním skupiny a započetím další. Opět, paměťově velmi efektivní krok.
- Hash Aggregate je velmi efektivní pokud se agregovaná relace vejde do hash tabulky v paměti (hashují se hodnoty v aggregačních sloupcích). To je velmi efektivní (pokud není třeba setříděný výsledek), ale má to vadu v tom že pokud estimátor výrazně podhodnotí počet distinct hodnot (což je parametr určující velikost hash tabulky), poté tabulka může za běhu vytéci z paměti.

## 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.)

- Nemusí být nutně důsledkem ORDER BY - používá se pro DISTINCT, GROUP BY nebo UNION.
- Při třídění bez indexu zkuste zvyšovat hodnotu work\_mem a sledujte kdy dojde k přepnutí na in-memory quicksort, a kolik paměti potřebuje.
- Obvykle je to tak že quick-sort potřebuje cca 4x až 5x více paměti než merge sort, např. ve výše uvedeném případě by měl potřebovat cca 72MB.
- Zkuste tabulku vytvořit bez "ORDER BY random()," tak aby index byl dobře korelován, a nechte si znova vypsat dotazu využívajícího index.
- Velký problém třídění pomocí indexu je že ho lze využít pouze na nejspodnější úrovni stromu - přímo na tabulky. Jakmile jste uprostřed stromu, tam již indexy dostupné nejsou a zbývají jen dvě tradiční metody.
- Často se stává že index je použit z důvodu že někde výše je ORDER BY případně MERGE JOIN.
- Vyzkoušejte si třídění v případě že "ORDER BY" obsahuje další (neindexovaný) sloupec, a v případě že máte index nad více sloupců ale v ORDER BY jsou uvedeny v opačném pořadí.

# 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)
  - částé 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)
```

- Asi jediný uzel který má nižší cenu než jeho vstup.
- Zkuste si dotaz s LIMIT na tabulce bez indexu (případně s index scanem vypnutým přes "SET enable\_indexscan = off;").
- LIMIT při použití s ORDER BY dokáže využívat modifikovaný třídicí algoritmus (stačí udržovat definovaný počet řádek).
- Plán musí vygenerovat všechny počáteční řádky, včetně těch přeskočených - často se stává že pro malé hodnoty OFFSET se použije např. index scan, ale od určité hodnoty OFFSET dojde k přepnutí na sekvenční sken s tříděním nebo jiný plán s velkou počáteční cenou ale následně levnější.

# 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
```

- V problematickém případě je zřejmé že sken musel projít (a zahodit 999899 řádek než přišel na tu správnou první). To je samozřejmě velmi neefektivní, vezmeme-li v potaz že tabulka má 1 milion řádek. Znamená to že bylo nutno projít 99.9% tabulky a to sekvenčně. Přitom odhadovaná cena (172) byla jen zlomkem celkové ceny (16925).
- V příznivém případě je zahzeno pouze 9998 řádek, pak je běh ukončen protože už další řádky nejsou potřeba.
- Na úrovni fyzických tabulek by se toto snad ještě dalo korigovat pomocí korelace, ale jakmile se LIMIT objeví někde výše v plánu, je to v podstatě neřešitelný problém.

# 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

- Obecně nedoporučuji s tímto moc hýbat, většinou se tím nadělá víc škody než užitku.
- Pokud se vám zdá že plánovač některý plán chybně neuvažuje, zkontrolujte tyto dvě hodnoty a případně je pokusně zvyšte (ale počítejte s tím že plánování může trvat dlouho a nebo může vytéci z paměti).

# Nested Loop

- asi nejjednodušší algoritmus (smyčka přes "outer" tabulkou, dohledání záznamu v "inner" tabulce)
- vhodný pro málo iterací a/nebo levný vnitřní plán (např. malíčká 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)
```

- Rychle produkuje první řádek, ale pro větší tabulky většinou pomalý.
- Většinou minimální paměťová náročnost (zálaží na konkrétních plánech - hlavně vnitřním), ale ty jsou většinou Index Scan apod.
- Častý v OLTP aplikacích, ne v DWH.
- Jako vnitřní je většinou volena menší tabulka, nebo tabulka na které je efektivní index.

# Nested Loop

- Kartézský součin není příliš obvyklý, přidejme index a podmínu na jednu tabulkou.

```
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

- K prvnímu problému typicky dochází v případě složitějších podmínek na korelovaných sloupcích, kdy planner předpokládá že selektivity se násobí. To ve výsledku vede ke špatnému odhadu počtu řádek (i o několik řádů nižšímu než realita) a ke špatnému plánu.
- Ke druhému problému může dojít obdobně - např. podhodnocením kardinality (a díky tomu i ceny) index scanu apod.
- Pokud se vnitřní plán nikdy nepustí (protože vnější tabulka je prázdná, resp. nejsou v ní odpovídající řádky), bude místo skutečných hodnot ve druhé závorce "(never executed)"

# 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 nevezde do work\_mem, rozdělí ji na tzv. "batche"
- následně čte větší tabulku a v hash tabulce vyhledává záznamy
  - velká tabulka se batchuje "odpovídajícím" způsobem
  - řádky prvního batche se zjoinují rovnou
  - ostatní se zapíší do batchů (temporary soubory, může znamenat I/O)

- V prvním kroku se přečte inner tabulka a vytvoří se z ní buď jedna hash tabulka nebo několik batchů. Iniciální počet batchů se rozhoduje během plánování, ale během exekuce se může zvýšit – hash join je díky tomu odolný vůči nepřesným odhadům (zejména podhodnocení velikosti hash tabulky) a jejímu následnému vytečení z paměti.
- Po vygenerování hash tabulky je znám konečný počet batchů – je-li použit jediný batch (tj. tabulka se celá vejde do work\_mem), vnější tabulka se přečte pouze jednou a join je hotový.
- Pokud bylo nutno použít více batchů, je vnější tabulka je nutno rozdělit ekvivalentním způsobem, tak aby bylo možno joinovat "po batchích." To je provedeno tak že do paměti je načten první batch hash tabulky, vnější tabulka je přečtena celá – řádky s klíčem nalezejícím do prvního batche jsou rovnou zjoinovány (vyhledáním v batchi hash tabulky), a zbývající řádky jsou zapsány do dočasných souborů – pro každý batch jeden soubor. Následně je vždy načten batch hash tabulky, batch vnější tabulky a dochází k joinu "per batch" (tj. jen nad zlomkem dat).
- Například pokud se hash tabulka rozdělí na 16 batchů, potom při prvním průchodu se řádky nalezející do 1. batche rovnou zjoinují, a řádky batchů 2 – 16 se zapíší do souborů, a každý z těchto batchů se následně přečte právě 1x. To znamená že pro N=2 se zhruba 50% vnější tabulky zapíše do dočasného souboru a znova přečte, pro N=4 se jedná zhruba o 75%, a podíl dat která se musí zapsat/načíst limitně roste ke 100% (pro N jdoucí do nekonečna).
- Batche se zapisují do dočasných souborů, které se nemusí nutně zapsat na disk. V případě systémů s nedostatkem volné paměti to ale může být nutné (jádro musí zapsat), což má za následek mnoho I/O operací (ale vesměs sekvenčních).

# 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)
```

- čím víc segmentů, tím hůře
  - může znamenat zapsání / opakovaného čtení velké části tabulky
  - jediné řešení asi je zvětšit work\_mem (nebo vymyslet jinou query)
- jedna hash tabulka nepřekročí work\_mem (dynamické batchování)
  - ale v plánu může být více hash joinů (násobek work\_mem) :-(

- Většinou pomalý start, v závislosti na velikosti vnitřní tabulky (kterou je nutno celou vygenerovat, než se vůbec začne se čtením vnější tabulky).
- V jednu chvíli je pro Hash Join uzel v paměti vždy pouze jeden segment hash tabulky (omezený work\_mem), tj. jeden uzel hash joinu nezpůsobí OOM. Může se ale stát že v plánu je několik Hash Joinů - každý si může v paměti držet svou hash tabulku.

# 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)
```

- počet "bucketů" hash tabulky je také důležitý
  - podhodnocení => velký počet hodnot na jeden bucket
  - dlouhý seznam => pomalé vyhledávání v tabulce :-(
- vylepšeno v 9.5 - dynamický počet bucketů, load faktor 1.0

- Hash join používá hash tabulky se zřetězenými hodnotami, tj. pole "příhrádek" (bucketů) a hodnoty spadající do jednoho bucketu jsou uloženy ve spojovém seznamu (linked list).
- Počet bucketů se určuje na základě odhadu počtu řádek ve vnitřní tabulce, tj. například pokud je očekáváno 1M řádek, použije se ~100 tisíc bucketů, protože PostgreSQL používá load faktor (faktor naplnění) 10, tj. snaží se nemít v průměru více než 10 řádek v jedné buňce (musí se projít sekvenčně, což zhoršuje výkon).
- Až do PostgreSQL 9.4 se počet bucketů určoval staticky na základě odhadů, tj. pokud byl chybný odhad (nižší než realita), byly seznamy v buňkách delší a výkon hash joinu výrazně poklesl.
- Ve verzi 9.5 došlo k výraznému zlepšení – počet bucketů se určuje stejně dynamicky jako počet batchů, a současně byl snížen load faktor na 1 (ale díky optimalizacím alokací paměti se i poté využívá méně paměti než v předchozích verzích).
- Poznámka č. 1: Toto neznamená že v žádném bucketu nemůže být více než "load faktor" řádek – pokud je v tabulce mnoho "duplicitních" řádek (se stejnou hodnotou ve sloupci přes který se joinuje), všechny nutně spadnou do stejného bucketu a vytvoří dlouhý spojový seznam. Toto není neobvyklá situace, a mimo jiné je to jeden z hlavních důvodů proč se pro hasj join nehodí tabulky s tzv. otevřeným adresováním.
- Poznámka č. 2: Alternativně je možné zkonztruovat hodnoty které po zhashování dají "kolizní" hodnoty – to není až tak obtížné, protože ačkoliv PostgreSQL používá 32-bit hash, bucket je určen jen daleko menším počtem bitů (např. pro 16384 bucketů jen 14 bitů). Tj. takový dataset není příliš obtížné zkonztruovat, ale v praxi to problémy vesměs nečiní.

# 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)

- Vyžaduje setřídění vstupů, což může být značně pomalé - záleží na datovém typu (např. pro texty se složitým LOCALE velmi pomalé).

# 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

- často se překládá na joiny, proto zmiňováno až tady
- Při optimalizaci subselectů hraje podstatnou roli proměnná `fromCollapse_limit`, která omezuje přepis subselectů na joiny. Subselect může být na join přepsán jen pouze pokud by v příslušném FROM seznamu nebyl vyšší počet relací než právě hodnota `fromCollapse_limit`.
- `fromCollapse_limit` omezuje "flattening poddotazů" a je to ochrana aby příliš nenařostla složitost plánování dané části dotazu (kvůli počtu joinovaných tabulek apod.)

# Korelovány 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 opakováně (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_id)  
GROUP BY a.id;
```

- Efektivní zejména pokud je z tabulky potřeba několik sloupců – jeden join namísto několika SubPlan uzlů.
- GROUP BY umožňuje vypsat i sloupce které nejsou přímo v klauzuli, ale jsou jednoznačně dané primáním klíčem který v klauzuli uveden je.

# 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)
```

- Může být efektivnější za situace kdy v SELECT části je několik samostatných subselectů protože každý může obsahovat jen jeden sloupec - v joinu je možné je spojit do jednoho.

# 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)
```

- H danka: Pro  se tyto pl ny li  kdy  pro EXISTS a IN jsou stejne?
- N pov da: NOT IN (NULL) => NULL
-  kol: Zkuste m sto poddotazu pou it pole.

## Ukázky dotazů

- často se překládá na joiny, proto zmiňováno až tady
- Při optimalizaci subselectů hraje podstatnou roli proměnná `fromCollapse_limit`, která omezuje přepis subselectů na joiny. Subselect může být na join přepsán jen pouze pokud by v příslušném FROM seznamu nebyl vyšší počet relací než právě hodnota `fromCollapse_limit`.
- `fromCollapse_limit` omezuje "flattening poddotazů" a je to ochrana aby příliš nenařostla složitost plánování dané části dotazu (kvůli počtu joinovaných tabulek apod.)

```

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

```

- Tento plán není ukázkou problému – je to ukázka jak plán funguje.
- <http://archives.postgresql.org/pgsql-hackers/2012-08/msg00418.php>
- Append operace se objevují buď v kombinaci s UNION (ALL) nebo partitioningem implementovaným přes INHERITS. Je to asi jediný uzel který může mít více než dvě vstupní relace.
- Merge Append je zvláštní případ operace, kdy se vstupy “spojují” stejně jako při merge-sortu, tj. zachovává se řazení vstupů. V tomto případě jsou vstupní relace čteny přes indexy jako by byly seřazeny podle sloupce “i”.
- Odhad ceny LIMIT je skutečně 1/20000 ceny “Result” uzlu.
- Zkuste namísto UNION ALL použít UNION.
- Zkuste odstranit index.

```

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

```

- <http://archives.postgresql.org/pgsql-hackers/2012-08/msg00418.php>
- Toto je ukázka jak úprava dotazu – doplnění WHERE podmínky – může dost zásadně ovlivnit plán, a to v negativním smyslu.
- Plánovač nyní nevěří že může použít index scan pro čtení vstupních tabulek setříděně (vadí mu tam právě podmínky), a použije sekvenční sken.
- Díky tomu ale nemůže použít Merge Append, musí provést “prostý” Append a operace setřídit až následně tradiční Sort operací.
- Použit je zde paměťově nenáročný “top-N sort” který ale stejně musí přečíst všechny řádky (na rozdíl od třídění přes Index Scan, jak je na předchozím slidu).
- Zkuste přepsat takto:

```

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

```

- Pokud budete používat UNION, UNION ALL a/nebo partitioning, toto jsou poměrně časté problémy. Konkrétní chování záleží na joiny – ve starších verzích dělá problém i prosté MIN/MAX.

```

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

```

- <http://archives.postgresql.org/pgsql-performance/2012-02/msg00047.php>
- Srovnejte s dotazem na následující stránce (liší se jen řazením).
- Najděte a vysvětlete v čem je pravděpodobně problém.
- Zkuste navrhnut možná řešení.

```

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

```

- Problém je zřejmě v nerovnoměrném rozdělení řádek vyhovujících podmínce (uvedené ve „Filter“ řádce).
- Na konci indexu (s velkými hodnotami ve sloupci „shareschange“) je jich zřejmě hodně, a proto „Index Scan Backward“ skončí velice rychle.
- Naopak na začátku indexu je jich málo, a tak je z indexu nutno načíst hodně řádek než je nasbíráno 15 vyhovujících.
- S odhadem rozložení příliš udělat nejde – alespoň ne na úrovni SQL.
- Jediná možnost je asi zkusit vymyslet lepší index (nad více sloupcí apod).

```

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';
- <http://archives.postgresql.org/pgsql-performance/2011-11/msg00258.php>
- Popis problému: PostgreSQL náhle přestal používat index na tabulce "link\_url" a namísto toho tabulku začal skenovat sekvenčně. Vzhledem k tomu že tabulka má 140 milionů řádek, je to problém.
- Zkuste identifikovat kde je problém – proč se nepoužije index?
- Zkuste nakreslit stromovou strukturu plánu.
- Jak by to bylo možné napravit?

```

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)

```

- Nemáme data takže můžeme jenom spekulovat, ale je pravděpodobné že příčinou všeho je že si databáze myslí že dostane příliš mnoho řádek z tabulky “email\_link” a to o 3 řády. V důsledku toho nezvolí Nested Loop, který by s 1 řádkou krásně fungoval (ve velké tabulce by se index scanem vše dohledalo).
- Druhým důsledkem je že pro druhý join se použije Merge Join, přičemž první tabulka je čtena přes Index Scan, což situaci ještě dále zhoršuje.
- Proč se tak děje není úplně jasné, ale nejspíše je to malou přesností statistik – mělo byt tedy pomocí zvýšit přesnost histogramu a/nebo MCV, například takto:
- ALTER TABLE email\_link ALTER COLUMN hostname SET STATISTICS 10000;

```
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)
```

- Pokuste se vysvětlit co se v tomto plánu děje.
- Identifikujte problematické místo a navrhněte řešení.

```

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)

```

- Použití uzlu “Append” naznačuje že se jedná o partitionovanou tabulkou, kterou lze získat takto:

```

CREATE TABLE parent (id INT);
CREATE TABLE child_1 () INHERITS (parent);
CREATE TABLE child_2 () INHERITS (parent);

INSERT INTO child_1
SELECT i FROM generate_series(1,1000000) s(i);

INSERT INTO child_2
SELECT i FROM generate_series(1000000, 2000000) s(i);

```

- Díky použití LIMIT bylo možno použít optimalizovaný třídící algoritmus a udržet se v paměti (což by při standardním třídění pravděpodobně možné nebylo). To je zřejmé z řádky

Sort Method: top-N heapsort Memory: 29kB

- Nenechte se zmást rozdílem v “rows” u operace “Sort” - to je v případě LIMIT normální (třídění předem neví kolik řádek z něj bude nadřazeným uzlem “vytaženo” a předpokládá že všechny).
- Možná optimalizace spočívá ve vytvoření indexů nad child tabulkami. Zkuste toto:

```

CREATE INDEX child_1_idx ON child_1(id);
CREATE INDEX child_2_idx ON child_1(id);

```

A znovu provedte exekuční plán.

```

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)

```

- Celý dotaz je zde:

WITH

```

subQuery_1 AS (
  SELECT f_order.id AS pk, f_order.f_orderfullprice AS val
  FROM f_order, f_ordersummary
  WHERE (f_order.orderid_id = f_ordersummary.id)
  AND orderstatus_id NOT IN ( 15, 86406, 86407, 86412 )
),
subQuery_0 AS (
  SELECT f_order.id AS pk, f_order.f_grossorderquantity AS val
  FROM f_order, f_ordersummary
  WHERE (f_order.orderid_id = f_ordersummary.id))
  AND orderstatus_id NOT IN ( 15, 86406, 86407, 86412 )
(
  SELECT SUM(subQuery_0.val * subQuery_1.val) AS val
  FROM "subQuery_1" FULL OUTER JOIN "subQuery_0" USING (pk)
  GROUP BY NULL::int
)
```

- Dotaz v rozumné době nedoběhl, musel být ukončen. CPU bylo celou dobu vytížené na 100% - vysvětlete kde je pravděpodobně problém.

```

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)

```

- V tomto příkladu je spojeno mnoho problémů najednou, zkusme je vzít popořadě.
- Zaprvé, je třeba si všimnout že oba poddotazy ve WITH části jsou zcela stejné, s výjimkou jednoho sloupce v SELECT části. Pokud se na SELECT podíváte znovu, zjistíte že se vlastně tabulka řeže na sloupce které jsou následně přes PK spojovány dohromady. To je ale spíše logický problém, ne problém plánu.
- Problém plánu spočívá v tom že neví že obě joinované relace jsou 1:1. Kromě toho že počet řádek zcela nesmyslně na 515418263361, tj. o několik řádů mimo, je zvolen Merge Join, tj. obě části je nutno nezávisle setřídit a následně zjoinovat. To nejen že spotřebuje paměť na mezivýsledky, ale také CPU na třídění a na join.
- Špatný odhad joinu je primárně způsoben použitím common table expressions (CTE), což je sofistikované uložiště řádek (v paměti či na disku), které mimo jiné komplikuje odhady statistik.
- Možná řešení – namísto CTE použít prosté aliasy ve FROM části dotazu.
- Ještě lepší řešení – neřezat tabulku na nudle a nejoinovat ji zbytečně.

```

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)

```

- Pokuste se vysvětlit jak to že první dotaz trvá o tolik déle než druhý, když má o tolik nižší odhad ceny.
- Odpověď: Je to proto že hodnoty ve sloupci val jsou dlouhé textové řetězce, a jako takové jsou ukládány pomocí TOAST, tj. mimo tabulku. Jejich čtení (kvůli spočítání funkce LENGTH) je poměrně náročné protože zahrnuje tzv. de-toasting, ale není nijak zohledněno v ceně.

Naopak přetypování sloupce ID na text (aby bylo možno spočítat MD5), i samotné spočtení MD5, jsou do ceny započteny ale jsou výrazně levnější než de-toasting.

- Toto je příklad toho že je mnoho důležitých faktorů do ceny nezapočtených.