Čtení exekučních plánů

Prague PostgreSQL Developer Day 2020 / 5.2.2020

Tomáš Vondra

tomas.vondra@2ndquadrant.com / tomas@pgaddict.com

Agenda

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

- Bez pochopení základních principů jak plánování dotazů funguje by pro vás interpretace exekučních plánů (a zejména pochopení kde je problém, protože to je důvod proč se exekuční plány čtou) daleko obtížnější. Je třeba alespoň trochu rozumět jak jsou vlastně dotazy vyhodnocovány, na základě jakých informací a jakým způsobem se databáze rozhoduje jak vypočítat výsledek, jak z plánů vybírá ten správný, co vše databáze zvažuje apod.
- 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 operací (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.

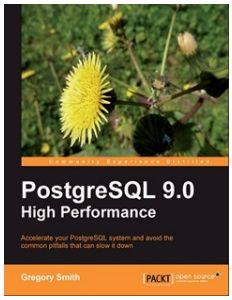
Zdroje

PostgreSQL dokumentace

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

PostgreSQL 9.0 High Performance

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



- Dynamické plánování dotazů jednou z hlavních vymožeností moderních databází, nicméně přesto není pořádně vysvětlené v téměř žádné běžné knize – knihy pro začátečníky se mu nevěnují vůbec, knihy pro pokročilé ho zmiňují jen letmo.
- Částečně je to pochopitelné, protože plánování je víceméně ekvivalentní s oborem nelineárního programování (hledání extrémů nelineárních funkcí), a jedná se o jednu z nejkomplikovanějších komponent zdrojového kódu databáze.
- Navíc komerční databáze si bedlivě střeží všelijaké triky a fígle jak si optimalizaci a plánování zjednodušit, nicméně PostgreSQL tento problém nemá.
- V dokumentaci PostgreSQL sice nenajdete detailní popis fungování optimizéru, ale najdete tam základní příklady odhadování velikosti výsledků apod. A pochopitelně také dokumentaci k příkazům které se k inspekci exekučních plánů používají.
- Současně existuje kniha “PostgreSQL 9.0 High Performance” která tématu plánování věnuje cca 60 stran. Byla sice napsána pro verzi 9.0 (tj. 5 verzí zpět), nicméně ačkoliv je plánování průběžně vylepšováno, neprochází nijak radikálními změnami.

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, ne jen relační, potažmo na všechny IT systémy obecně).
- Plánovač se sice snaží dotazy analyzovat a případně vhodně přeformulovat, ale rozhodně nerozumí všem závislostem mezi sloupci/tabulkami, a některé z nich pochopitelně ani nejsou na úrovni schématu zachyceny. Nehledě na to že každá “chytrost” něco stojí – čím více heuristik, tím dražší plánování.
- 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
  - Mají se použít indexy? Které?
  - V jakém pořadí a jakým algoritmem se mají provést joiny?
  - Které podmínky se mají vyhodnotit první?
- koncovým kritériem je čas běhu dotazu
  - strašně špatně se odhaduje a modeluje
- namísto toho se pracuje s “cenou”
  - vyjadřuje nároky daného plánu na prostředky (CPU, I/O)
  - čím méně operací musím udělat, tím rychlejší dotaz
  - založeno na statistikách tabulek / indexů a odhadech

- 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 dotazu. Všechny zvažované plány samozřejmě musí mít vlastnost že vracejí správný výsledek na položený dotaz.
- Plánovač postupně konstruuje možné plány dotazu, odhaduje ceny a nechává si jenom ty nejzajímavější. Přitom plánů je ale obecně vzato exponencielně mnoho – jenom možností jak zjoinovat K tabulek je (K!) byť tedy reálný počet validních pořadí je většinou nižší díky vztahům mezi tabulkami. Další množství potenciální plánů vzniká z možností použít různé indexy na různých podmínkách apod.
- Takže plánovače vesměs nekonstruují všechny plány ale používají různé triky z dynamického programování jak co nejdříve eliminovat velké skupiny plánů (pomocí backtrackingu apod.).
- 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).

Cost proměnné

- udávají cenu některých “základních” operací
- celková cena se z nich vypočítává
- I/O operace jsou výrazně nákladnější

`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)
`parallel_setup_cost = 1000.0`	spuštění paralelních workerů
`parallel_tuple_cost = 0.1`	kopírování řádky z worker procesu

- Cost proměnné zachycují ceny základních operací, cena dalších operací (např. vytváření dočasných souborů apod.) se z nich odvozují. Všimněte si že seq_page_cost=1.0 což lze interpretovat tak že “sekvenční čtení stránky” je základní jednotka ceny a ceny ostatních operací jsou vztaženy k této operaci (Jak drahé je to ve srovnání se sekvenčním čtením?)
- Uvedené hodnoty jsou výchozí a vesměs se jedná o časem osvědčené hodnoty, nicméně lze očekávat že pro specifický hardware se mohou měnit. To platí např. o random_page_cost na SSD discích (nepamatuji se že bych musel nějak měnit některé z CPU proměnných).
- Z hodnot je také vidět dávná zkušenost že I/O operace v databázích dominují, i když toto se v poslední době díky nárůstu objemu RAM a rychlým diskům mění, a není problém narazit na databázi s CPU bottleneckem.
- Je dobrým zvykem seq_page_cost neměnit a ponechat ji jako referenční hodnotu.
- 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).
- Z řady vybočuje effective_cache_size, protože to neurčuje cenu ale jedná se pouze o nápovědu kolik RAM bude dostupné pro cache v některých plánech (např. Index Scan). Občas se uvádí že se jedná o (shared buffers + filesystem cache) a doporučuje se nastavit např. 75% dostupné RAM, ale to je většinou příliš vysoká hodnota protože nepočítá s paralelně běžícími dotazy. Správnější je tedy ((shared buffers + filesystem cache) / K) kde K je odhad počtu paralelních dotazů používajících Bitmap Index Scan.
- Na ceně plánu se projevuje i work_mem (in-memory vs. on-disk třídění apod.)

Ukázka výpočtu ceny

- hledáme “optimální” z ohromného množství plánů

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

- rozložíme na základní operace
  - sekvenční čtení tabulky stránka po stránce (8kB)
  - parsování řádek z tabulky
  - vyhodnocení podmínky na každé řádce
- např. tabulka má 1.000 stránek a 10.000 řádek

```
cena = 1000 * seq_page_cost +
       10000 * cpu_tuple_cost +
       10000 * cpu_operator_cost
```

- Uvedený příklad je samozřejmě velmi triviální, ale dostatečně ilustruje princip výpočtu ceny.
- 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).
- 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.
- Pokud vás odhad ceny zajímá, podrobnosti najdete ve velice přehledné podobě v souboru src/backend/optimizer/path/costsize.c

Ukázka výpočtu ceny

- hledáme “optimální” z ohromného množství plánů

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

- tabulka má 1.000 stránek a 10.000 řádek

```
cena = 1000 * 1.0 +
       10000 * 0.01 +
       10000 * 0.0025 = 1125
```

cvičení: 01-vypocet-ceny.sql

```
CREATE TABLE t1 (a INT, b TEXT);
INSERT INTO t1 SELECT i, repeat(md5(i::text), 23)
                 FROM generate_series(1, 10000) s(i);
ANALYZE t1;
SELECT relpages, reltuples FROM pg_class
                          WHERE relname = 't1';

 relpages | reltuples
----------+-----------
     1000 |     10000
(1 row)

EXPLAIN SELECT * FROM t1 WHERE b = 'xxx';
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on t1  (cost=0.00..1125.00 rows=1 width=744)
   Filter: (b = 'xxx'::text)
(2 rows)
```

Cena vs. čas

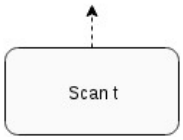
- víceméně virtuální hodnota
  - vyjadřuje nároky daného plánu na prostředky (CPU, I/O)
  - korelace s časem, ale nelineární vztah
- stabilita vzhledem ke vstupním parametrům
  - malá změna v selektivitě podmínek / odhadech => malá změna ceny
- stabilita vzhledem k času
  - malá změna ceny => malá změna času
  - vzhledem k času

Toto činí cost-based plánování odolné vůči nepřesným odhadům.

- Cena je konstruována tak aby byla stabilní, a to jak vzhledem ke vstupním parametrům a zadruhé vzhledem k času.
- Ideální cenu si lze představit jako spojitou funkci, tj. bez náhlých “skoků” - nemělo by se tedy stávat že při malé změně vstupních parametrů (např. podmínce odpovídá o 1% řádek více) se cena zvýší výrazně nepřiměřeně (např. 1000x).
- Co se týká stability vzhledem k času (trvání dotazu), měl by platit princip že malá změna ceny odpovídá malé změně času. Jak už bylo uvedeno, vztah mezi cenou a časem je velmi komplikovaný a obecně nelineární, nicméně nemělo by se stávat že cena se změní o 1% a čas narosto 1000x.
- Díky tomu není nutno základní parametry ceny (o kterých je řeč na následujícím slidu) 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).
- Je třeba brát v potaz že cena je jenom určitou aproximací skutečné ceny a vůbec nezahrnuje některé důležité informace – např. není k dispozici informace která data jsou aktuálně v paměti apod. protože to by výrazně komplikovalo a prodražovalo plánování.
- Výše uvedené platí pro tzv. "cost-based" plánovače, které vychází ze statistik a hodí se pro dynamicky se měnící data (objem, statistické rozložení). Druhým (starším) typem plánovačů 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.

Plán jako strom

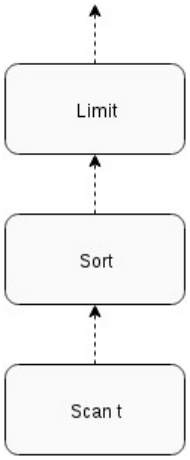
```
SELECT * FROM t;
```



- Plán dotazu má formu stromu, jehož vrcholy tvoří operace vykonávané databází.
- Pozici listů stromu většinou zaujímají “skeny” tabulek, které čtou data ze souborů různými způsoby (přímo, pomocí indexu) apod.
- Mezi základní typy skenů patří např.
  - sequential scan
  - (bitmap) index scan
- Existují ale i další typy uzlů které mohou zaujmout pozici listu a generovat data “z ničeho”. Příkladem je například generování dat z funkce – viděli jsme generate_series().

Plán jako strom

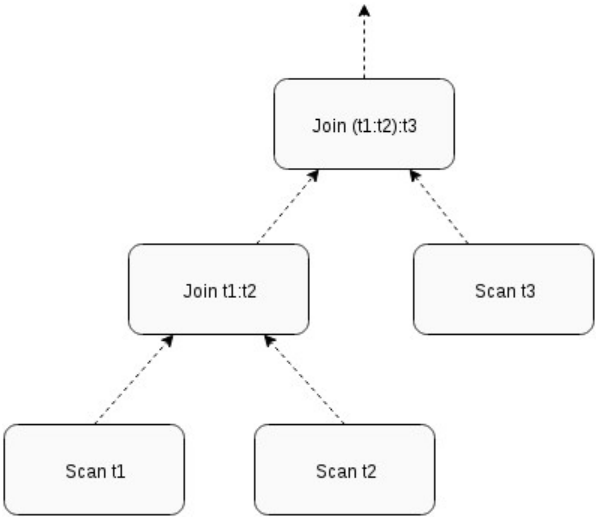
```
SELECT * FROM t ORDER BY a LIMIT 10;
```



- Pozici vnitřních uzlů stromu zaujímají další operace, které nějakým způsobem zpracovávají data získávaná z podřízených uzlů – ať už se jedná o skeny nebo nějaké další vnitřní operace.
  - V uvedeném příkladě se jedná o třídění (Sort) a omezení počtu řádek (Limit).
- Jakým způsobem je exekuce celého plánu řízena a jakým způsobem si uzly řádky předávají?
- Exekutor v PostgreSQL je “pull-based” tj. uzly na vyšší úrovni si “tahají” data z podřízených uzlů. Úplně nahoře je “klient” (tj. nějaký program načítající výsledky dotazu) který volá metodu “dej mi další řádek” na prvním uzlu, a ten si stejným způsobem načítá řádky z operací pod ním.
- Zpracování dotazu končí buď v situaci kdy jsou přečtena všechna data z listů a tato informace “probublá” až ke klientovi, nebo v okamžiku kdy se některý z uzlů rozhodne že “stačilo” (v příkladě výše stačí aby LIMIT vyprodukoval požadovaný počet řádek).
- Některé uzly umí řádky "streamovat" tj. průběžně načítat a přeposílat dál (např. sekvenční nebo index scan, třídění pomocí indexu apod.), ale některé uzly musí nejdříve načíst všechny řádky z podřazených uzlů a až pak mohou vrátit první výsledek (např. tradiční třídění nebo agregace pomocí hash tabulky).
- I toto (jak moc závisí na rychlosti produkování první řádky) je věc kterou plánovač musí při výběru plánu zvažovat.

Plán jako strom

```
SELECT * FROM t1 JOIN t2 ON (…) JOIN t3 ON (…)
```



- Další operací bez které se v relačních databázích neobejdeme je samozřejmě spojení tabulek, nebo-li join. Díky joinům získávají exekuční plány reálnou podobu stromu s větvením.
- Joiny zabudované v PostgreSQL mají vždy podobu se dvěma podřízenými tabulkami, nicméně existují i implementace které joinují i více tabulek naráz (např. s využitím GPU apod.).

Statistiky

- plánovač potřebuje odhadovat
  - velikosti tabulek (skeny)
  - velikosti mezivýsledků (vstupy vnitřních uzlů)
  - selektivitu podmínek (kvůli mezivýsledkům)
- databáze si udržuje statistiky o datech
  - `pg_class` (centrální katalog, obsahuje velikost relací)
  - `pg_statistic` (systémový katalog se statistikami sloupců)
  - `pg_stats` (pohled na `pg_statistic`, určeno pro lidi)
  - `pg_statistic_ext, pg_statistic_ext_data` (vícesloupcové statistiky)
  - `pg_stats_ext` (pohled na `pg_statistic_ext`, určeno pro lidi)
- neplést s “runtime” statistikami o provozu (pg_stat_*)

- V uvedených katalozích pg_class a pg_statistic jsou k dispozici informace o velikostech relací (tj. nejenom tabulek ale i indexů apod.) a distribuci hodnot v jednotlivých sloupcích.
- Z těchto velmi omezených statistik se odvozují odhady počtu řádek (např. selektivita podmínek), počty skupin v agregaci, apod.
- pg_class je centrální katalog ve kterém jsou registrovány všechny základní objekty (tabulky, indexy, ...) a mimo jiné obsahuje i základní informace o velikost objektu.
- pg_statistic je katalog se detailními statistikami sloupců, nicméně pokud se na statistiky chcete podívat použijte raději pg_stats což je "pohled" nad pg_statistic, určený pro lidi. To jaké statistiky o sloupcích jsou k dispozici uvidíme podrobněji na jednom z následujících slidů.
- PostgreSQL také poskytuje katalogy s dalším typem statistik o provozu databáze, přístupu k objektům apod.
  - pg_stat_all_tables / pg_stat_user_tables / pg_stat_sys_tables
  - pg_stat_all_indexes / pg_stat_user_indexes / pg_stat_sys_indexes
  - ... functions,
  - další “contrib” moduly pg_stat_statements / pg_stat_plans / ...
- Tyto statistiky jsou užitečné pro monitoring, plánovač / optimizér je nijak nepoužívá.

Statistiky

- pg_class – statistiky pro relaci jako celek
  - relpages – počet stránek relace (8kB bloky)
  - reltuples – počet řádek (nedpovídá COUNT(*))

```
CREATE TABLE t2 (id INT);
INSERT INTO t2 SELECT i FROM generate_series(1, 1000000) s(i);
SELECT relpages, reltuples FROM pg_class WHERE relname = 't2';

 relpages | reltuples
----------+-----------
        0 |         0

ANALYZE t2;
SELECT relpages, reltuples FROM pg_class WHERE relname = 't2';
```

- pg_class je centrální katalog objektů v databázi, kromě jiného obsahuje i informaci o počtu řádek a velikosti souboru (jako počet 8kB stránek).
- Často se setkáváme s nepochopením jak funguje MVCC a jaký dopad to má na chápání počtu řádek v tabulce, a proč např.

  SELECT COUNT(*) FROM t;

  nemůže jednoduše vrátit hodnotu z pg_class.reltuples.
- MVCC – Multi-Version Concurrency Control (aka “multigenerační architektura”) je způsob jak umožnit běh více paralelních dotazů díky zpřístupnění více “verzí” řádky pro jednotlivé sessions (tak aby každá session viděla právě jenom verzi která byla commitnuta před jejím spuštěním). Tím se výrazně snižuje počet zámků oproti předchozím architekturám, které vesměs zamykaly všechny řádky (nebo stránky) ke kterým session potřebovala přistupovat.
- V PostgreSQL to funguje tak že modifikace řádky (UPDATE / DELETE) vytvářejí kopii řádky, a každá session si vybere poslední verzi řádky kterou podle pravidel smí vidět. Tj. například pokud z tabulky s milionem řádek polovinu smažete, existující dotazy stále budou muset zpracovávat celý milion řádek. Naopak pokud provedete UPDATE poloviny tabulky, tabulka dočasně naroste na 1.5M řádek.
- Staré verze řádek se odstraňují až v okamžiku kdy neexistuje session která by je mohla vidět, a to buď explicitně během VACUUM nebo (automaticky) autovacuum.

Statistiky

- pg_stats (pg_statistic) - statistiky na úrovni sloupců
  - **`avg_width`** – průměrná šířka hodnot (v bytech)
  - **`n_distinct`** – počet různých hodnot ve sloupci (GROUP BY)
  - **`null_frac`** – podíl NULL hodnot ve sloupci
  - **`correlation`** – korelace hodnot s pořadím v tabulce
  - **`most_common_values`**, **`most_common_freqs`**
    - nejčastější hodnoty a jejich frekvence (MCV)
  - **`histogram_bounds`**
    - equi-depth histogram (přihrádky reprezentují stejné % hodnot)
    - jenom hodnoty které se nevešly na MCV list

- Hodnoty statistik jsou z podstaty nepřesné – jedná se o odhady získáváné z náhodného vzorku tabulky (několik tisíc řádek) a stává se že jejich hodnoty nejsou dostatečně věrnou aaproximací - např. pokud je distribuce nějak podivně vychýlena apod.
- Velikost vzorku je určována proměnnou "statistics target" která také určuje kolik hodnot se bude uchovávat ve statistikách (v MVC, v histogramu apod.). a to buď globálně nebo jenom pro konkrétní sloupec (to raději, je s tím spojen overhead).

  ALTER TABLE t ALTER COLUMNT 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)

Statistiky

```
CREATE TABLE t3 (a INT, b INT, c INT, d INT);

INSERT INTO t3
     SELECT
            mod(i,50),   -- 50 hodnot (uniform)
            mod(i,1000), -- 1000 hodnot (uniform)
            1000 * pow(random(),2), -- 1000 hodnot (skewed)
            (CASE WHEN mod(i,3) = 0 THEN NULL ELSE i END)
FROM generate_series(1,1000000) s(i);

ANALYZE t3;

SELECT a, COUNT(*) FROM t3 GROUP BY 1 ORDER BY 1;
SELECT * FROM pg_stats WHERE tablename = 't3' AND attname = $1;
```

cvičení: 02-statistiky.sql

- Příklad vytváří tabulku s 1M řádek a se 4 sloupci s různými distribucemi hodnot.
- a – 50 hodnot, každá ve 20.000 řádek (rovnoměrné rozdělení)
- b – 1000 hodnot, každá v 1.000 řádek (rovnoměrné rozdělení)
- c – 1000 hodnot, nerovnoměrná distribuce (nižší hodnoty mají vyšší frekvenci)
- d – 30% hodnot jsou NULL hodnoty

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

cvičení: 03-explain-analyze.sql

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 vyprodukování první řádky, druhé udává celkovou cenu na vyhodnocení dané části plánu až do vyprodukování 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;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 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

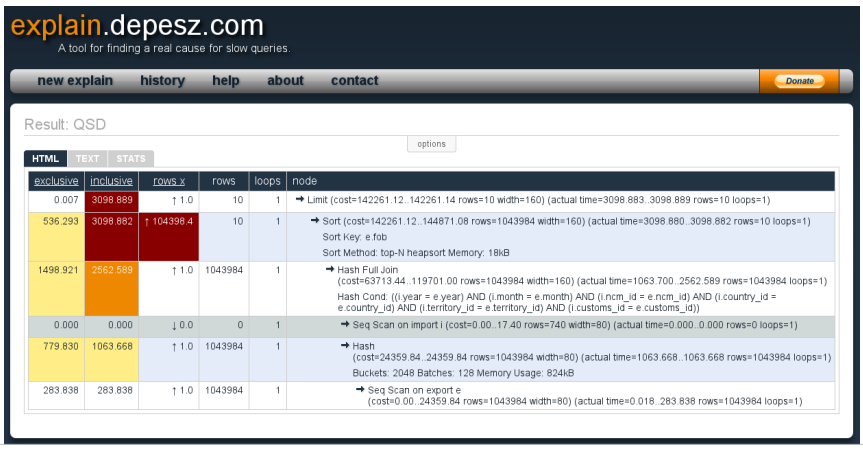
```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

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

explain.depesz.com

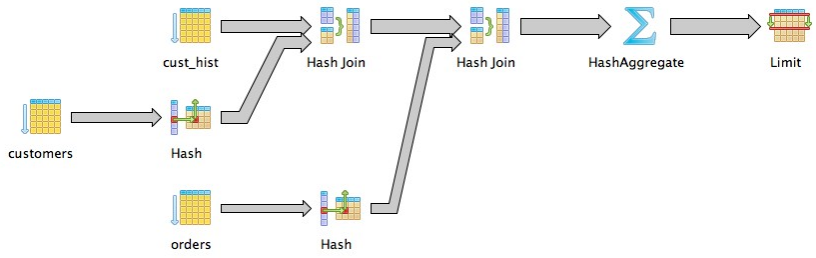
- zachovává strukturu z EXPLAIN (ANALYZE)
- vytáhne podstatné informace, zvýrazní problémy (statistiky, doba běhu)
- dobrý sdílení exekučních plánů např. po mailu / chatu
- http://explain.depesz.com/



- Asi nejčastěji používaný nástroj na vizualizaci plánů.
- Do velké míry zachovává podobu výstupu z EXPLAIN (pravá strana).
- Na levou stranu vytahuje podstatné informace (kardinality, trvání) a zvýrazňuje potenciálně problematické operace (špatné odhady, vysoká cena, dlouhý běh).
- Ne vždy úplně jednoduché na orientaci – např. dvě větve JOIN mohou být dost daleko od sebe a není jednoduché je identifikovat.
- Pokud posíláte exekuční plán do konference, je dobré poslat odkaz na explain.depesz.com ale také připojit původní plán jako textový soubor (tj. nevkládat inline, kvůli formátování).

PgAdmin

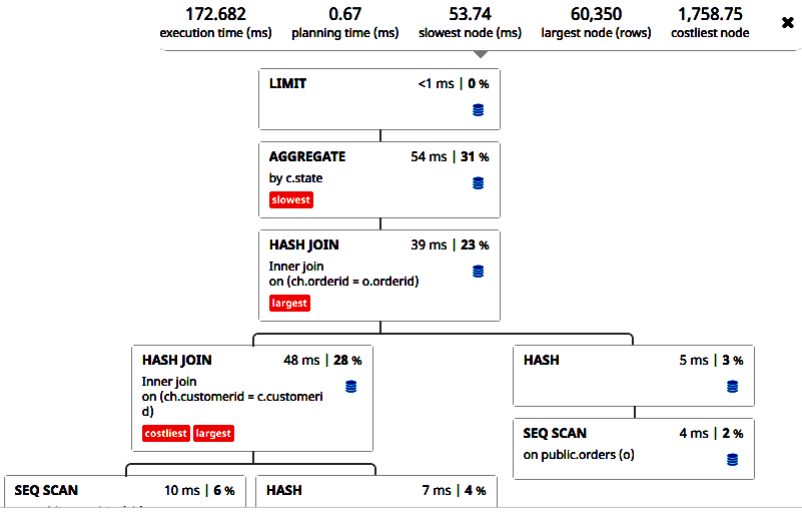
- pohled založený na “toku” dat mezi operacemi
- stromová struktura, nicméně tok “zleva doprava”
- znázornění počtu řádek pomocí šířky spojnice



- Pokud používáte PgAdmin, možná znáte tento způsob vizualizace.
- Lepší ilustrace stromové struktury, byť tedy strom “leží”.

pev (Postgres EXPLAIN Visualizer)

- jasně ukazuje stromovou strukturu
- http://tatiyants.com/pev/



- Nový projekt, také webové rozhraní.
- Hezčí vizualizace stromové struktury, anotace operací (slowest, largest, ...)

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 microsec

- 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

Soustřeďte se na operace 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.

- 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 odhady 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.
- Připravte se na zjištění že v některých situacích prostě plánovač není schopen dojít k ideálnímu plánu, např. v důsledku nepřesnosti statistik nebo nemožnosti spolehlivě odhadnout některé typy podmínek.

Neaktuální statistiky

- pokud plánovač nemá statistiky (pg_stats), používá “default” odhady (např. 33%)
- stává se také že statistiky jsou neaktuální – např. po aktualizaci velké části dat
- opraví se buď ručním ANALYZE nebo vyřeší autovacuum

```
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
```

cvičení: 04-neaktualni-statistiky.sql

- Dokud není tabulka alespoň jednou zanalyzována, nemá plánovač vůbec žádné statistiky a nedokáže tedy např. odhadovat selektivitu podmínek apod. Namísto toho používá různé “výchozí” konstatní hodnoty – např. v tomto případě 33% pro selektivitu WHERE podmínky.
- Druhou možností je že sice na tabulce nějaké statistiky jsou, ale jsou zastaralé a nezachycují aktuální stav. K tomu může dojít například po velké dávkové aktualizaci která přepíše velkou část tabulky.
- Existují dva způsoby jak toto vyřešit – autovacuum (resp. autoanalyze) a explicitní spouštění ANALYZE po aktualizacích modifikujících velké části dat.
- V OLTP aplikacích si typicky vystačíme s autovacuum - k velkým aktualizacím příliš nedochází a na průběžné aktualizace menších částí dat (jednotky procent) dokáže autovacuum dobře reagovat.
- Občas se setkáváme se situacemi kdy administrátoři autovacuum vypnou protože ho považují za zbytečné, což je až na výjimky zásadní chyba a vesměs to vede k daleko větším problémům později (index bloat, anti-wraparound vacuum ...).

Pokud to bolí, musíte to dělat častěji.

- V DWH aplikacích je situace složitější, protože dávkové aktualizace často aktualizují podstatnou část dat a následně na ní hned spouštějí další dotazy. V tomto případě autovacuum nestačí reagovat a nezbývá než do aktualizačního skriptu přidat explicitní ANALYZE.

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
```

- plánovač není schopen odhadovat komplexní výrazy (použije default)
- 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 lze manuálně provést “inverzi”
  - např: **“i*i <= 100”** => **“i BETWEEN -10 AND 10”**

cvičení: 05-komplexni-podminky.sql

- 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.
- V některých situacích lze opravit vytvořením “expression” indexu.

```
CREATE INDEX ON a((i*i));

ANALYZE a;
```

- Nicméně údržba indexů není zdarma.

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 (i < 1000) AND (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 kobinace podmínek založeny na předpokladu nezávislosti
  - selektivita kombinace je součin jednotlivých selektivit
- každá podmínka má selektivitu ~1/1000 = 0.001
  - 0.001 x 0.001 = 0.000001 – jeden řádek, ale i=j (závislost)

cvičení: 06-korelovane-sloupce.sql

- 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 ofixova t 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;
```

- Existuje patch který přidává možnost statistik na více sloupcích, ale bude nějakou dobu trvat než se ho podaří začlenit (určitě ne 9.6, možná 9.7).

Špatný odhad n_distinct

- odhad počtu různých hodnot překvapivě patří k nejtěžším problémům
- většinou sedí, ale pro nějak “divné” databáze může dojít k chybám
- n_distinct není přímo vidět, projevuje se přes “rows” (např. v 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 …`**”

- Všechny operace v PostgreSQL respektují work_mem nastavení, s jedinou výjimkou a tou je Hash Aggregate. V případě výrazného podhodnocení počtu skupin může jednoduše dojít až k vyčerpání paměti a OOM.
- Na úrovni sloupce lze odhady n_distinct opravit pomocí

  ALTER TABLE ... SET n_distinct = ...

  nicméně to nemusí fungovat pro GROUP BY s kombinací sloupců (zejména pokud jsou sloupce nějak korelované).

Prepared statements

- chceme ušetřit na plánování (včetně odhadu kardinalit apod.)
- vede na generický plán
  - nemůže používat konkrétní hodnoty parametrů, používá “nejčastější”
  - pro netypické hodnoty může dávat neoptimální plány
- od 9.2 se chová trochu jinak (kontroluje hodnoty a případně přeplánuje)

```
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)
```

cvičení: 07-prepared-statements.sql

- 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.
- 9.2 toto řeší kontrolou plánů
  - prvních 5 dotazů skutečně naplánuje s konkrétními hodnotami parametrů
  - při plánování dalšího se podívá na variabilitu plánů (cen)
  - pokud se ceny příliš neliší, vytvoří generický plán
  - pochopitelně to není neprůstřelné, ale výrazně snižuje počet problémů

Obtížné joiny

- joiny jsou jedny z nejdražších a nejhůře odhadnutelných operací
- tabulka obsahující jen sudé hodnoty

```
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)
```

cvičení: 08-komplikovane-jony.sql

- Kardinality joinů, to je pro plánovač dosti tvrdý oříšek.
- Odhady jsou založené na porovnávání MCV na obou stranách podmínky. Pokud MCV nejsou na obou stranách joinu, porovnává se průměrná selektivita (1/ndistinct).
- Optimalizátor nevidí spoustu "zřejmých" vztahů - například nám je jasné že join sudých a lichých čísel je prázdná množina, ale joiny přes výrazy prostě přesnější nebudou.
- Obzvláště komplikované je odhadování joinů na více sloupcích – jedná se o stejný problém jako u korelovaných podmínek.

Obtížné joiny

- joiny jsou jedny z nejdražších a nejhůře odhadnutelných operací
- změňme trochu podmínku (sudý = lichý)

```
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 - 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)
```

- Kardinality joinů, to je pro plánovač dosti tvrdý oříšek.
- Odhady jsou založené na porovnávání MCV na obou stranách podmínky. Pokud MCV nejsou na obou stranách joinu, porovnává se průměrná selektivita (1/ndistinct).
- Optimalizátor nevidí spoustu "zřejmých" vztahů - například nám je jasné že join sudých a lichých čísel je prázdná množina, ale joiny přes výrazy prostě přesnější nebudou.
- Obzvláště komplikované je odhadování joinů na více sloupcích – jedná se o stejný problém jako u korelovaných podmínek.

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_timing  = false
auto_explain.log_verbose = false
auto_explain.log_buffers = true
auto_explain.log_format = yaml
auto_explain.log_nested_statements = false
```

- Poměrně účinný nástroj jak identifikovat a analyzovat dotazy které “zlobí” jenom občas.
- Častá příčina – velká dávková aktualizace dat v noci, nenásledovaná bezprostředně explicitním ANALYZE. V intervalu než se spustí autovacuum plány používají staré statistiky.
- Pozor na overhead - instrumentaci není zadarmo a je nutno ji zapnout ještě před spuštěním dotazu (tj. nejde ji zapnout dodatečně, např. pokud dotaz běží déle než zvolený limit).
- Lze ale snížit vhodnými volbami – např. pokud nastavíte “log_analyze=false” potom je overhead v podstatě nulový, ale nebudete mít informace o reálné době běhu a počtu řádek.
- V případě kombinace “log_analyze=true” a “log_timing=false” dostanete alespoň informace o přesnosti odhadů – to je většinou dostačující pro kontrolu přesnosti odhadů a analýzu dotazu, a eliminuje se tím nejnákladnější část instrumentace (získávání času).

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`

- Použitelné hlavně pro analýzu problému (lze nastavit v rámci vybrané session).
- Rozhodně se nedoporučuje nastavovat jako výchozí hodnotu pro celý server, při nejhorším lze nastavit pro vybrané konkrétní dotazy.
- Poměrně těžkopádný nástroj – funguje na úrovni celého dotazu, tj. pokud se v něm daná operace vyskytuje vícekrát, nelze si vybrat kterou instanci “vypnete”.

Způsoby přístupu k tabulkám

V této části jsou probrány základní způsoby jak lze přistupovat k relacím, tzv. skeny. 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

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

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

Sequential Scan

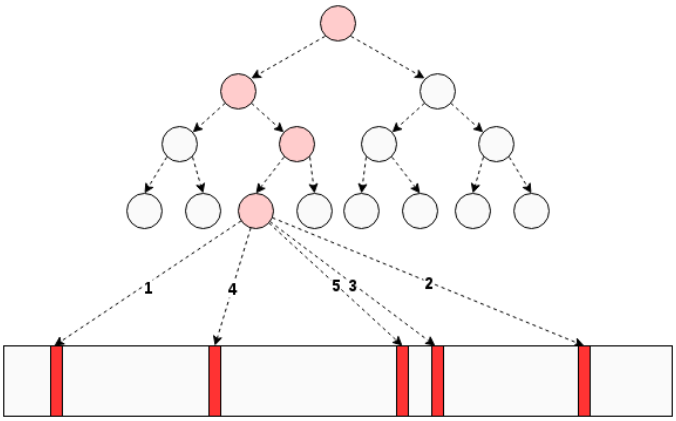
- nejjednodušší možný sken – sekvenčně čte tabulku
- řádky může zpracovat filtrem (WHERE podmínka)

```
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 dat
- “nešpiní” shared buffers (ring buffer), synchronizované čtení

- Malou tabulkou je míněna 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 tabulku.
- 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 tabulku.
- Tím že nešpiní shared buffers je míněno 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 stejné 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



Index Scan

- datová struktura optimalizovaná pro hledání (typicky strom, ale ne nutně)
  - efektivní pro čtení malé části z velké tabulky
  - ne každá podmínka je použitelná pro index

```
CREATE TABLE t4 AS SELECT i AS x, i AS y
                    FROM generate_series(1,100000) s(i);
CREATE INDEX t4_idx ON t4(x);
ANALYZE t4;
EXPLAIN ANALYZE SELECT * FROM t4 WHERE x = 1000 AND y = 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: (x = 1000)
   Filter: (y = 1000)
 Total runtime: 0.039 ms
(3 rows)
```

- Indexy jsou založeny na umožnění rychlého přístupu k řádkům tabulky podle hodnot sloupce / kombinace sloupců / podmínky. Tradiční jsou b-tree indexy (stromové), PostgreSQL ale umí např. také hash indexy nebo GIN/GiST indexy (ale to budeme opomíjet).
- Důsledkem index scanů je 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 korelován).
- 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").
- Pokud je index s tabulkou korelován, stránky budou čteny opakovaně a výrazně se sníží objem náhodných I/O operací. Zkuste pomocí indexu načíst tabulku (ORDER BY) setříděnou podle sloupce indexu. Zkuste to s indexem korelovaným a nekorelovaným s tabulkou.

  CLUSTER t4 ON t4_idx;
- Indexů může být na tabulce více – mohou být na různých podmínkách, kombinacích podmínek a různě se překrývat. Plánovač se snaží zvolit ten nejefektivnější 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 Conditions” pracují pouze se sloupci obsaženými v indexu, Filter může pracovat i se sloupci načtenými z tabulky.

Index Only Scan

- novinka v PostgreSQL 9.2, vylepšení Index Scanu
- pokud index obsahuje všechny potřebné sloupce, lze číst index
- efektivní pokud je dostupná informace o viditelnosti řádek na stránce

```
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)
```

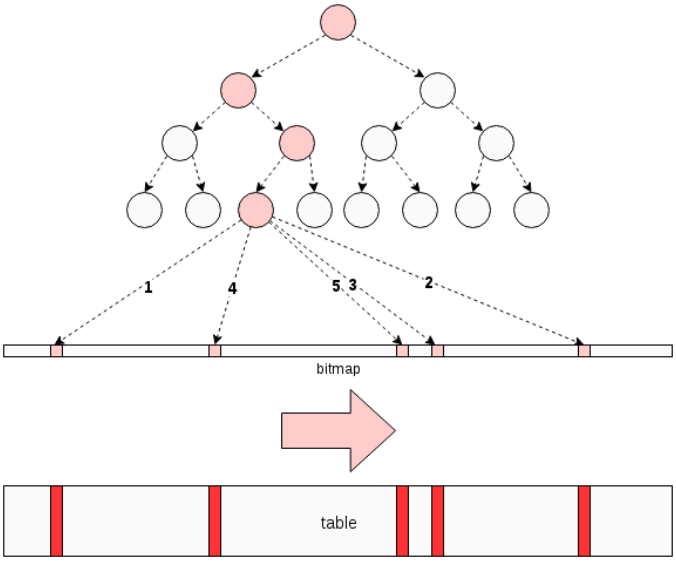
- Index Only Scan přichází v úvahu tam kde byl předtím možný index scan, a kde je v indexu vše potřebné (sloupce potřebné v dotazu).
- Index only neznamená že se do tabulky občas sáhnout nemůže - visibility map nemusí obsahovat dostatečné údaje pro všechny stránky. Samozřejmě čím častěji se musí skákat do tabulky kvůli kontrole viditelnosti, tím více se eliminuje přínos Index Only Skenů.
- Jediný typ skenu který produkuje setříděný výstup (podle klíčů indexu).
- Funguje tak že se čtou “listy” stromu – to nemusí být nutně čistě sekvenční čtení (není zaručeno že listy nejsou uloženy “na přeskáčku”), ale regulérní index scan se chová stejně.

Bitmap Index Scan

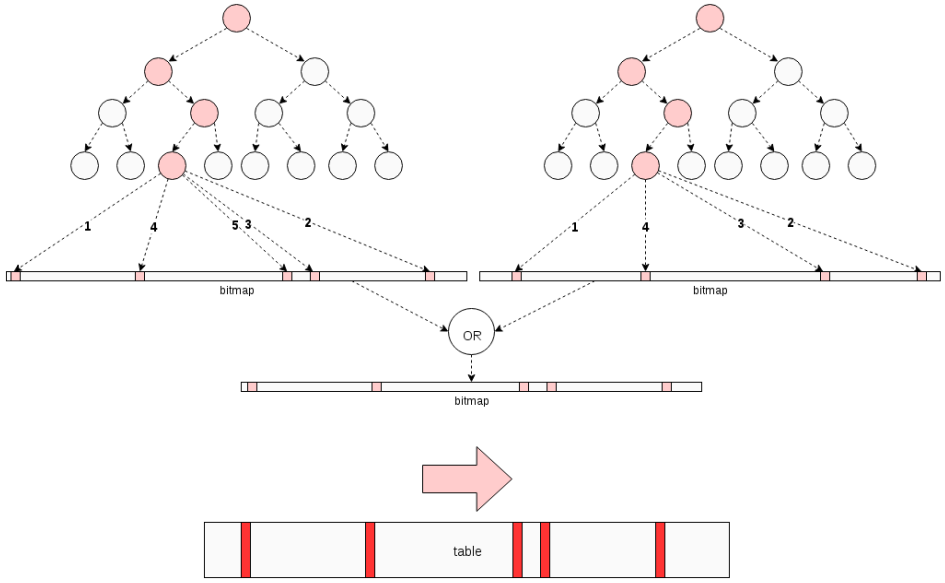
- Index Scan je efektivní pouze pro selektivní podmínky (např. <5%)
  - nepoužitelné pro podmínky s velkou selektivitou (např. 20%)
  - náhodné I/O nad tabulkou (Index Scan)
  - overhead při práci s indexem (Index Only Scan)
- Bitmap Index Scan čte tabulku sekvenčně pomocí indexu
  - nejdříve na základě indexu vytvoří bitmapu řádek nebo 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



Bitmap Index Scan



Bitmap Index Scan

```
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 podmínkách a sledujte jak se budou měnit rozhodnutí které bitmapy sestavit a které nikoliv.
- Často se používá pro dotazy zahrnující IN(pole) nebo ANY(pole).

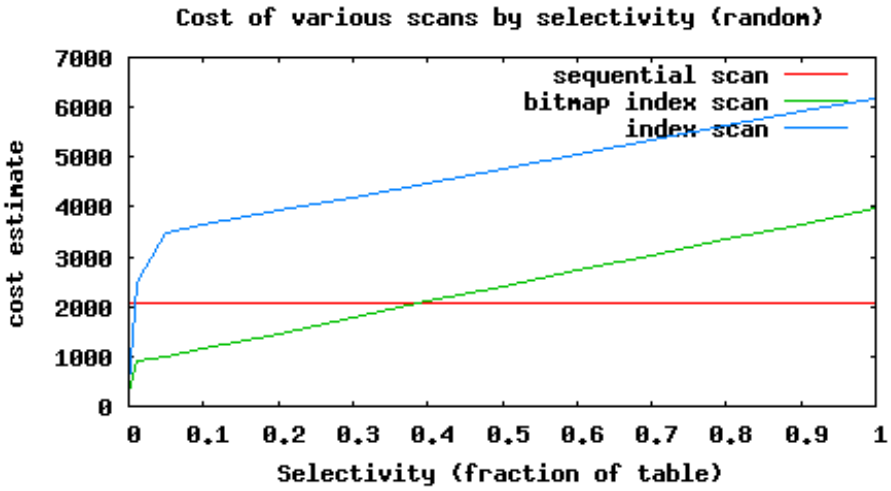
Srovnání skenů

- vezměme tabulku (1M integerů v náhodném pořadí)
- sledujme cenu 3 základních plánů pro podmínku s různou selektivitou
- nutno vypínat/zapínat jednotlivé varianty
  - enable_seqscan = (on|off)
  - enable_indexscan = (on|off)
  - enable_bitmapscan = (on|off)

```
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 < (1000000 * 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, korelaci slouců/indexů, konkrétních podmínkách atd.

Srovnání skenů



- Není to sice příliš vidět, ale pokud je selektivita dostatečně malá (~1%) vychází nejlépe obyčejný index scan.
- Jakmile selektivita vzroste nad několik málo procent, cena index scanu velmi rychle vzroste a to až přes sekvenční sken a nejefektivnějším plánem se stává bitmap index scan..
- Okolo 40% selektivity přestává bitmap index scan být efektivní (v podstatě se očekává že bitmapa bude obsahovat 1 pro všechny stránky tabulky, díky náhodnému pořadí) a nejefektivnějším plánem je prostě přečíst celou tabulku.
- Otázka: Co se stane pokud bude tabulka korelovaná, tj. pokud při jejím vytváření vynecháte ORDER BY random()? Jak se změní graf?

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;
```

- 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 buď čtou 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;
```

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

```
                                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

- CTE znamená "Common Table Expression" tj. "Společný Tabulkový Výraz" a označuje relaci (výsledek dotazu) odkazovanou na více místech dotazu.

CTE nejsou aliasy, mají daleko hlubší (positivní i negativní) důsledky.

- 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). Tam je CTE dokonce nutností a jinak to v podstatě udělat nejde (pomineme-li možnost implementovat jako funkci vracející tabulku).
- CTE ukládají data to 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 předeženou, v tom případě se stávají leaderem a přebírají vyhodnocování dotazu.
- Data mohou zabrat až "work_mem" v paměti, poté se začnou odkládat na disk (což může mít dopad na výkon, ale soubory jsou dočasné a nevolá se na ně fsync).
- 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.).

Další operace

Agregace, třídění, LIMIT, ...

Agregace

- PostgreSQL má tři implementace agregace
  - vybírá se během plánování (nelze měnit za běhu)
- **Aggregate**
  - v případech bez GROUP BY (takže vlastně jediná skupina)
- **Group Aggregate**
  - k detekci skupin využívá třídění vstupní relace
  - nemusí čekat na dokončení agregace, ale potřebuje setříděný vstup
- **Hash Aggregate**
  - využívá hash tabulku, neumožňuje “batching”
  - může alokovat hodně paměti (podhodnocený n_distinct)

- Varianta "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 agregace 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 setřídíte 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 agregačních 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

```
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)
```

cvičení: 09-agregace.sql

Agregace / OOM

- HashAggregate není adaptivní
  - plán nelze za běhu změnit (např. na Group Aggregate)
  - hash tabulka nepodporuje batching (na rozdíl od Hash Joinu)
- nestává se často, ale pokud OOM tak většinou z tohoto důvodu
- 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 / Index Only Scan)
  - v paměti (quick-sort)
  - 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.)
  - v případě Index Only Scan roste cena pomaleji

- Nemusí být nutně důsledkem ORDER BY - používá se pro DISTINCT, GROUP BY nebo UNION. Může být i důsledkem joinování pomocí MERGE JOIN (viz. dále).
- 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.
- Zdaleka to ale nemusí být tak že více paměti vede k rychlejším dotazům – paměť se ubírá například z page cache, interaguje s cache na CPU apod.
- Třídění pomocí indexu lze využít pouze na nejspodnější úrovni plánu – při čtení přímo z tabulky. Jakmile jste uprostřed stromu, tam již indexy dostupné nejsou a zbývají jen dvě tradiční metody.
- Zkuste tabulku vytvořit bez "ORDER BY random()," tak aby index byl dobře korelován, a nechte si znovu vypsat dotazu využívajícího index.
- Vyzkoušejte si třídění v případě že "ORDER BY" obsahuje další (neindexovaný) sloupec, a v případě že máte index nad více sloupci 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);
ANALYZE a;

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)
```

cvičení: 10-trideni.sql

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 “vnořené” operace
  - počty řádek (požadovaný a celkový)

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

- Asi jediný uzel který může mít nižší cenu než jeho vstupy (nemusí je vyhodnotit celé).
- 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 rovnoměrné rozložení

- řádky vyhovující podmínce rovnoměrně rozloženy v tabulce

```
CREATE TABLE a (id INT);

INSERT INTO a SELECT mod(i,10000)
                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=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
```

cvičení: 11-limit.sql

- 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 zahozeno 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.

LIMIT a nerovnoměrné rozložení

- identifikace tohoto problému je poměrně těžká
  - všimněte si “rows removed by filter”

```
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
```

cvičení: 11-limit.sql

- 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 zahozeno 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.
- Jakmile použijete LIMIT, budou se preferovat plány s nízkou počáteční cenou (jak rychle může být vygenerována první řádka), mimo jiné např.
  - index scan (namísto sekvenčního skenu)
  - index scan (namísto standardního třídění)
- Další složitost do odhadování LIMIT přináší klauzule OFFSET, která říká jaký počet úvodních řádek se má přeskočit.

Triggery

- dlouho “temná hmota” exekuce – 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)
```

cvičení: 12-triggery.sql

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 = 8**
  - ovlivňuje jak moc může plánovač měnit pořadí tabulek během joinu
  - lze zneužít ke “vnucení” pořadí použitím explicitního joinu

    ```
    SET join_collapse_limit = 1
    ```

- **geqo_threshold = 12**
  - určuje kdy se má opustit vyčerpávající hledání pořadí tabulek a přejít na genetický algoritmus
  - 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 možný 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)
```

cvičení: 13-nested-loop.sql

- 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

```
-- kartezsky soucin (bez podminky)
FOR x IN “outer tabulka” LOOP
    FOR y IN “inner tabulka” LOOP
        zkombinuj radky “x” a “y”
    END LOOP
END LOOP

-- podminka s indexem
FOR x IN “outer tabulka” LOOP
    FOR y IN “inner tabulka odpovidajici podmince” LOOP
        zkombinuj radky “x” a “y”
    END LOOP
END LOOP
```

cvičení: 13-nested-loop.sql

- SS

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

- 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 nevejde 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 tabulku 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 náležejícím 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 náležející 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 znovu 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

```
h := hash(“inner table”)

FOR x IN “outer table” LOOP

    FOR y IN lookup(h, key) LOOP

        zkontroluj join podminku

        zkombinuj “x” a “y”

    END LOOP

END LOOP
```

- 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 tabulku 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 náležejícím 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 náležející 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 znovu 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) :-(

cvičení: 14-hash-join.sql

- 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 “přihrádek” (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 zkonstruovat 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 zkonstruovat, 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) ORDER BY 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)
```

- 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 na jiný typ joinu

cvičení: 15-merge-join.sql

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 from_collapse_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 přávě hodnota from_collapse_limit.
- from_collapse_limit omezuje "flattening poddotazů" a je to ochrana aby příliš nenarostla složitost plánování dané části dotazu (kvůli počtu joinovaných tabulek apod.)

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.

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;
```

- 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 primárním klíčem který v klauzuli uveden je.

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.