



Čtení exekučních plánů

Prague PostgreSQL Developer Day 2020 / 5.2.2020

Tomáš Vondra

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

© 2020 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

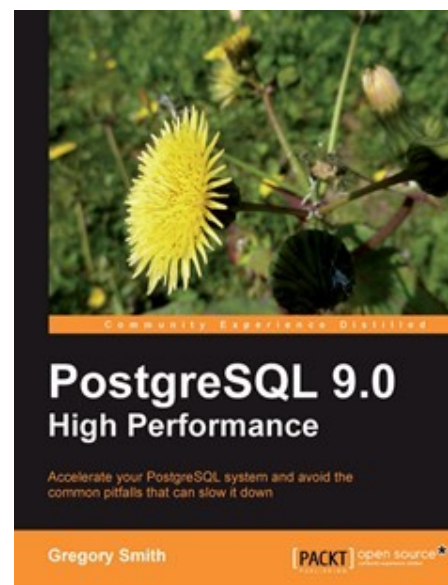
Zdroje

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



Proč se o plánování starat?

- SQL je deklarativní jazyk
 - popisuje pouze požadovaný výsledek
 - volba postupu jeho získání je úkolem pro databázi
- Porozumění plánování je předpoklad pro
 - pochopení limitů databáze (implementačních, obecných)
 - definici efektivní DB struktury
 - analýzu problémů se stávajícími dotazy (pomalé, OOM)
 - lepší formulaci SQL dotazů

Plánování jako optimalizace

- hledáme “optimální” z ohromného množství plánů
 - 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

Cost proměnné

- udávají cenu některých “základních” operací
 - celková cena se z nich vypočítává
 - I/O operace jsou výrazně nákladnější
-
- | | |
|---|------------------------------------|
| • <code>seq_page_cost = 1.0</code> | sekvenční čtení stránky (seq scan) |
| • <code>random_page_cost = 4.0</code> | náhodné čtení stránky (index scan) |
| • <code>cpu_tuple_cost = 0.01</code> | zpracování řádky z tabulky |
| • <code>cpu_index_tuple_cost = 0.005</code> | zpracování řádky indexu |
| • <code>cpu_operator_cost = 0.0025</code> | vyhodnocení podmínky (WHERE) |
| • <code>parallel_setup_cost = 1000.0</code> | spuštění paralelních workerů |
| • <code>parallel_tuple_cost = 0.1</code> | kopírování řádky z worker procesu |

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

$$\begin{aligned} \text{cena} = & 1000 * \text{seq_page_cost} + \\ & 10000 * \text{cpu_tuple_cost} + \\ & 10000 * \text{cpu_operator_cost} \end{aligned}$$

Ukázka výpočtu ceny

- hledáme “optimální” z ohromného množství plánů

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

- tabulka má 1.000 stránek a 10.000 řádek

$$\begin{aligned} \text{cena} &= 1000 * 1.0 + \\ &\quad 10000 * 0.01 + \\ &\quad 10000 * 0.0025 = 1125 \end{aligned}$$

cvičení: 01-vypocet-ceny.sql

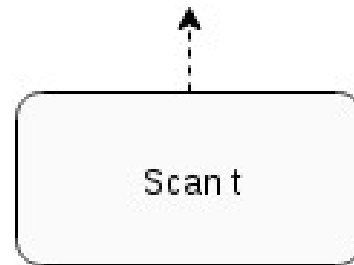
Cena vs. čas

- 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.

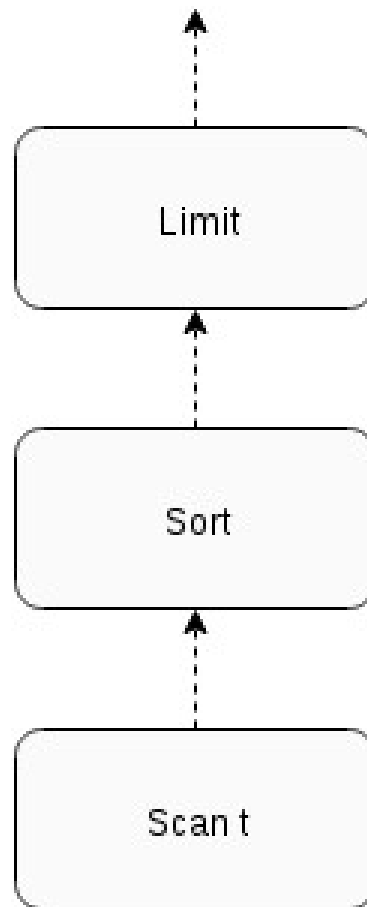
Plán jako strom

```
SELECT * FROM t;
```



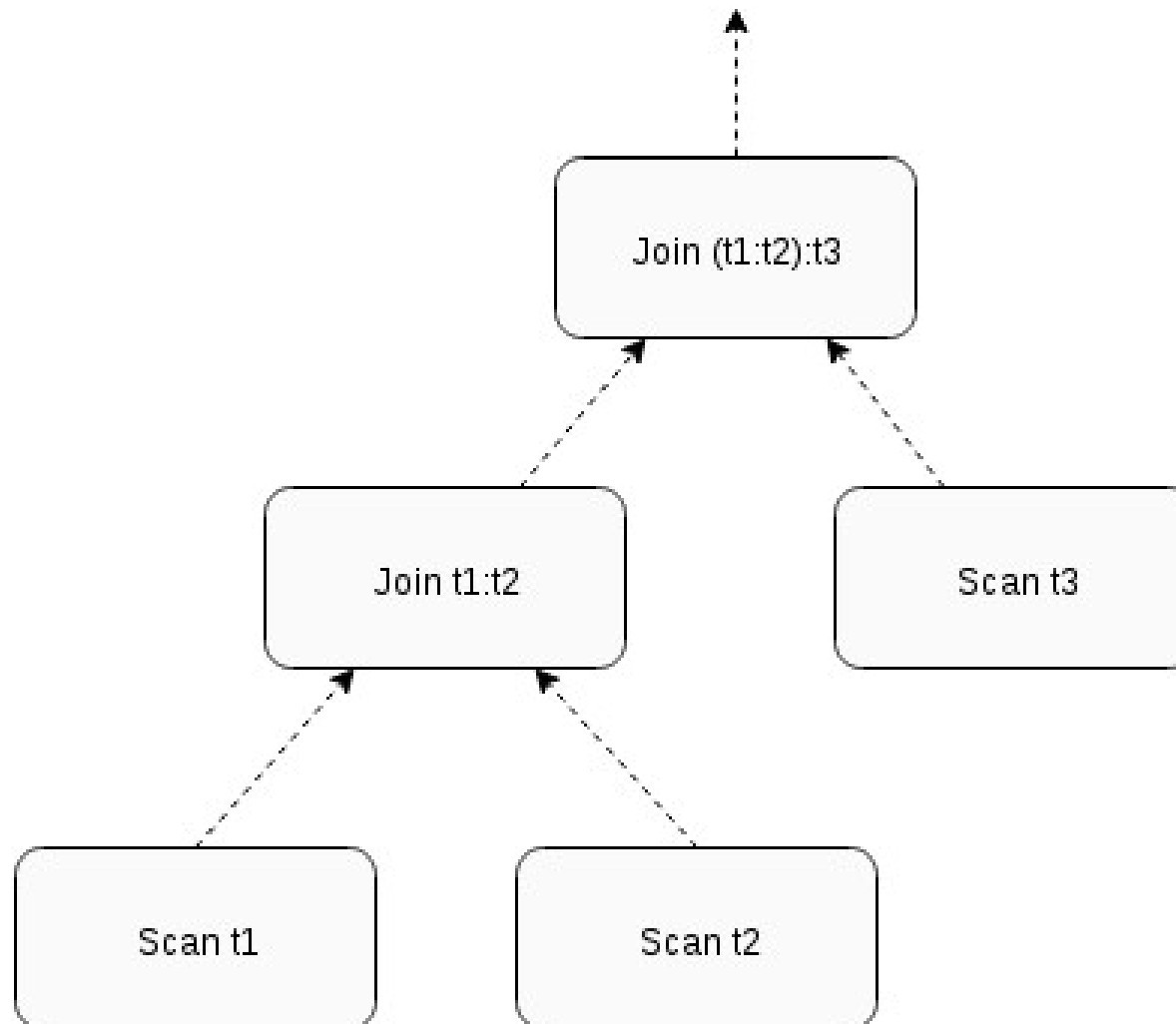
Plán jako strom

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



Plán jako strom

SELECT * FROM t1 **JOIN** t2 ON (...) **JOIN** t3 ON (...)



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_*`)

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';
```

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 (příhrádky reprezentují stejné % hodnot)
 - jenom hodnoty které se nevešly na MCV list

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;
```


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

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
```

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/>

explain.depesz.com
A tool for finding a real cause for slow queries.

[new explain](#) [history](#) [help](#) [about](#) [contact](#) [Donate](#)

Result: QSD

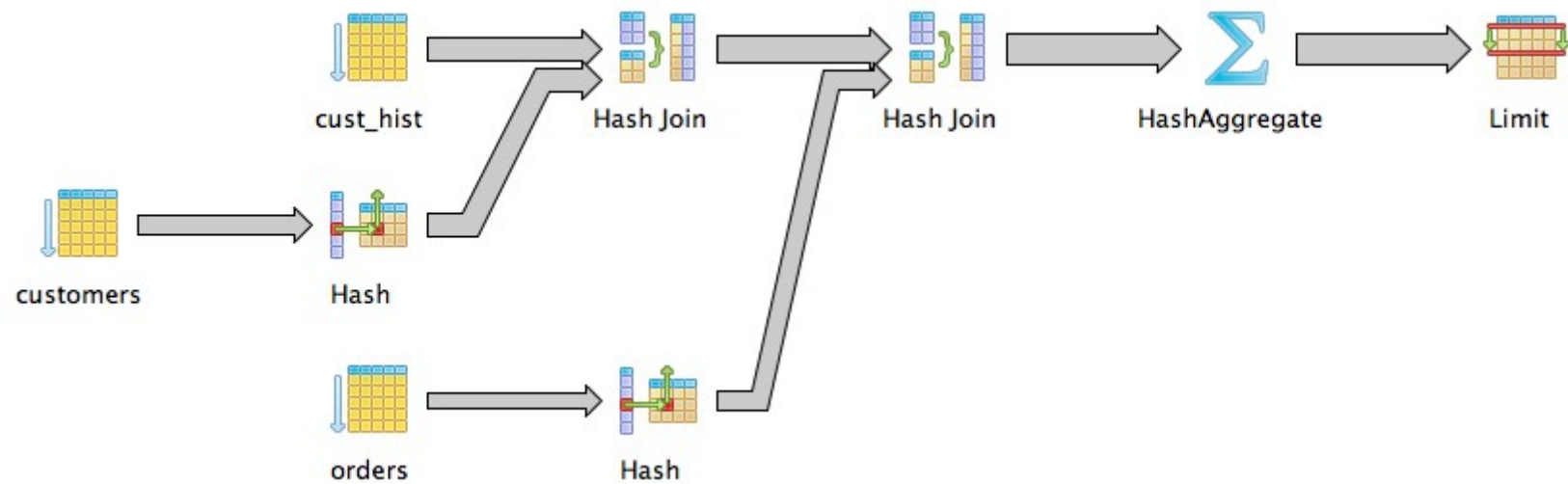
options

HTMLTEXTSTATS

exclusive	inclusive	rows x	rows	loops	node
0.007	3098.889	↑ 1.0	10	1	→ Limit (cost=142261.12..142261.14 rows=10 width=160) (actual time=3098.883..3098.889 rows=10 loops=1)
536.293	3098.882	↑ 104398.4	10	1	→ Sort (cost=142261.12..144871.08 rows=1043984 width=160) (actual time=3098.880..3098.882 rows=10 loops=1) Sort Key: e.fob Sort Method: top-N heapsort Memory: 18kB
1498.921	2562.589	↑ 1.0	1043984	1	→ Hash Full Join (cost=63713.44..119701.00 rows=1043984 width=160) (actual time=1063.700..2562.589 rows=1043984 loops=1) Hash Cond: ((i.year = e.year) AND (i.month = e.month) AND (i.ncm_id = e.ncm_id) AND (i.country_id = e.country_id) AND (i.territory_id = e.territory_id) AND (i.customs_id = e.customs_id))
0.000	0.000	↓ 0.0	0	1	→ Seq Scan on import i (cost=0.00..17.40 rows=740 width=80) (actual time=0.000..0.000 rows=0 loops=1)
779.830	1063.668	↑ 1.0	1043984	1	→ Hash (cost=24359.84..24359.84 rows=1043984 width=80) (actual time=1063.668..1063.668 rows=1043984 loops=1) Buckets: 2048 Batches: 128 Memory Usage: 824kB
283.838	283.838	↑ 1.0	1043984	1	→ Seq Scan on export e (cost=0.00..24359.84 rows=1043984 width=80) (actual time=0.018..283.838 rows=1043984 loops=1)

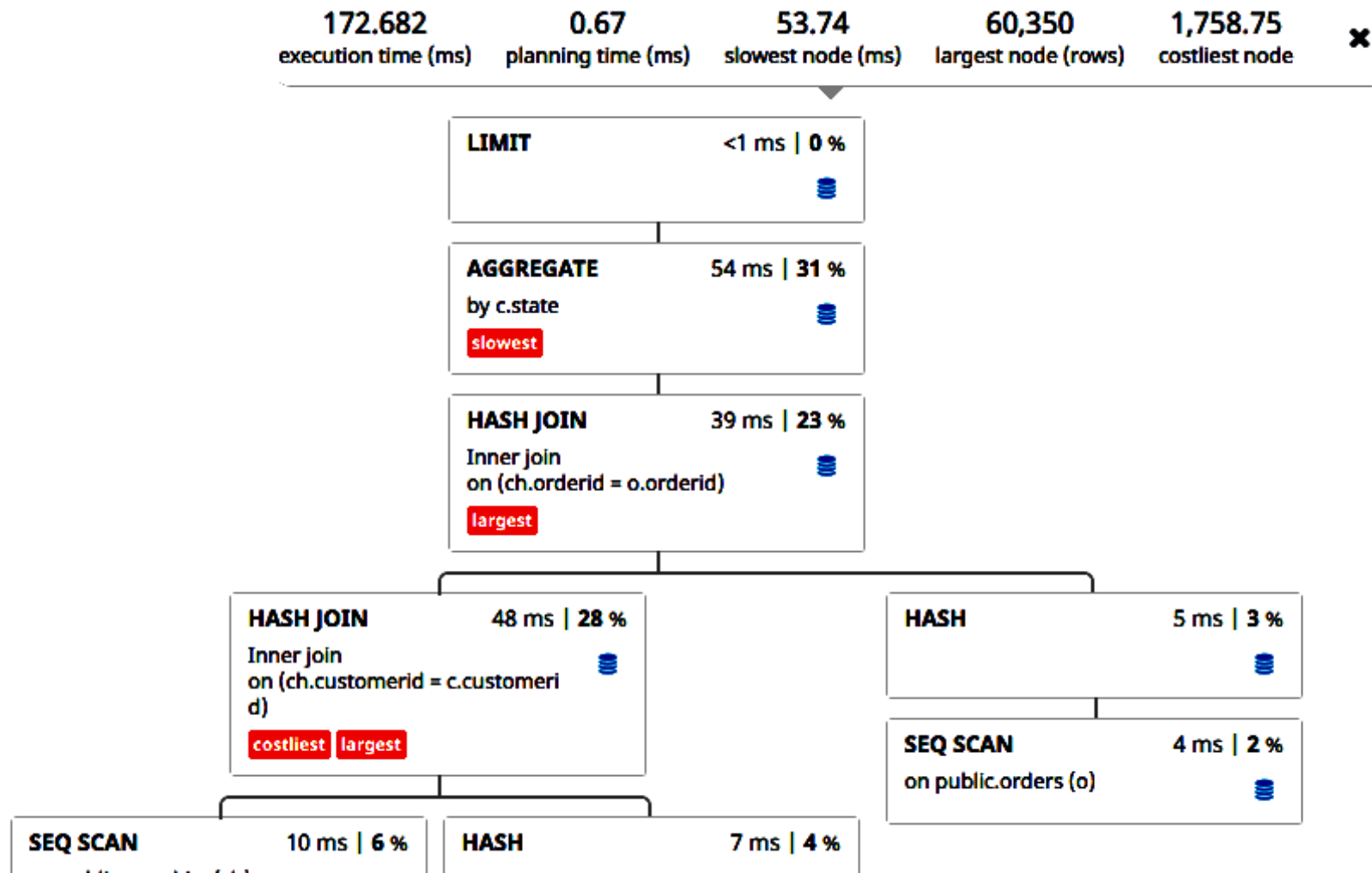
PgAdmin

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



pev (Postgres EXPLAIN Visualizer)

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



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

Obvyklé problémy

Soustřed'te 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.

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

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"**

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 kombinace podmínek založeny na předpokladu nezávislosti
 - selektivita kombinace je součin jednotlivých selektivit
- každá podmínka má selektivitu $\sim 1/1000 = 0.001$
 - $0.001 \times 0.001 = 0.000001$ – jeden řádek, ale $i=j$ (závislost)

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 ...**”

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

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

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)
```


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
```

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

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

Způsoby přístupu k tabulkám

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 tabulku
- řádky může zpracovat filtrem (WHERE podmínka)

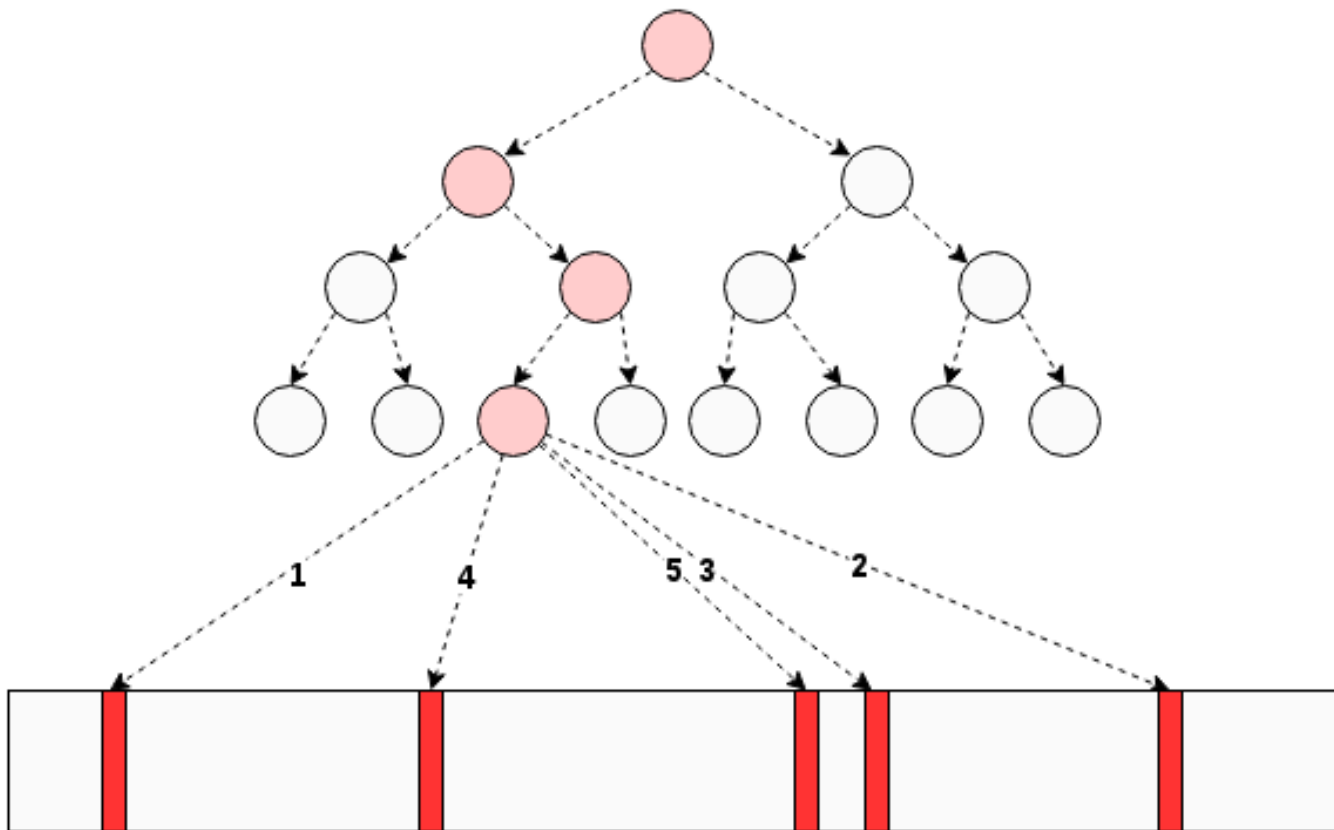
```
CREATE TABLE a AS SELECT i FROM gs(1,100000) s(i);  
ANALYZE a;  
EXPLAIN ANALYZE SELECT i FROM a WHERE i = 1000;
```

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

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)
```

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);  
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;
```

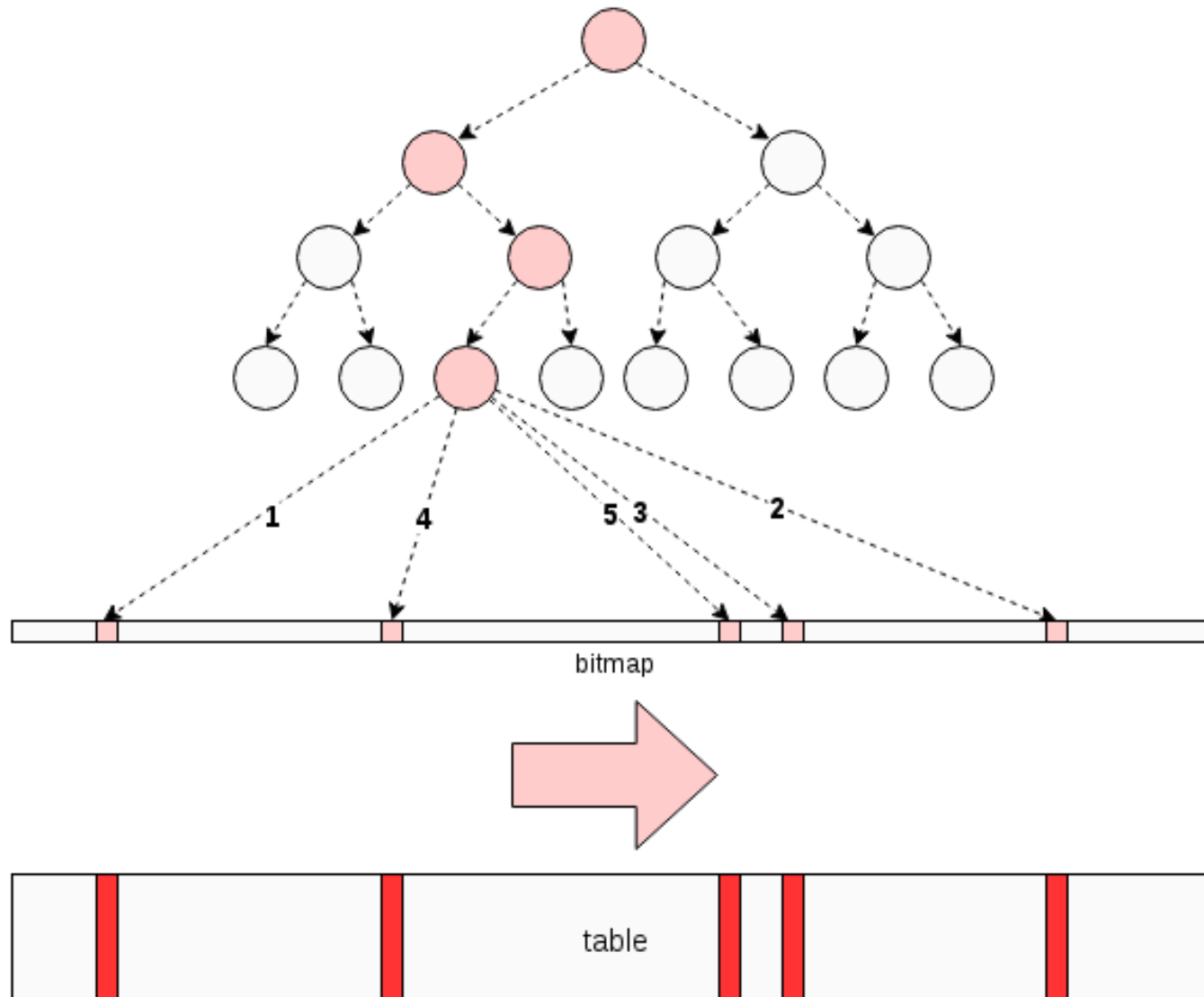
QUERY PLAN

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

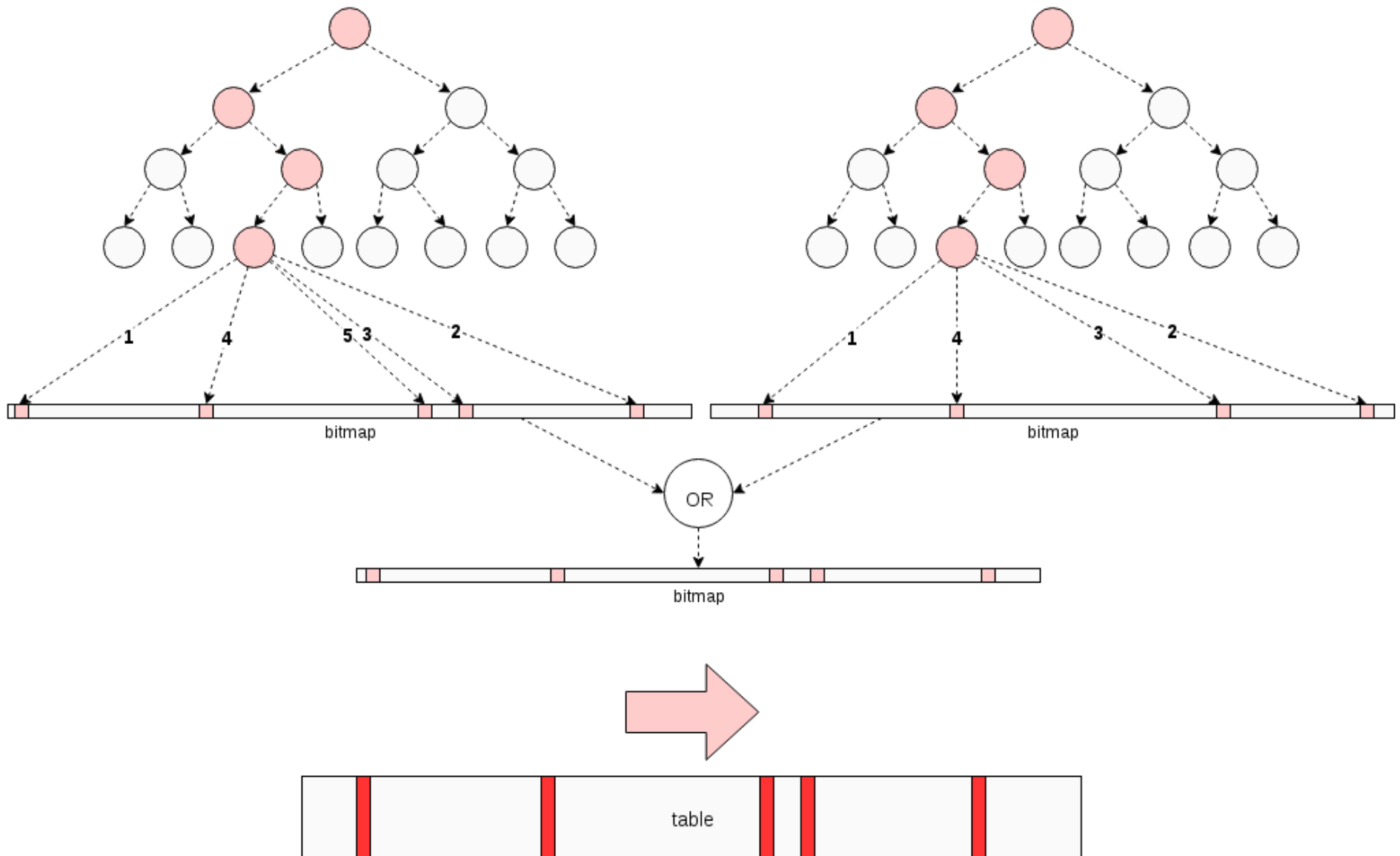

Bitmap Index Scan

- 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

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)
```

Srovnání skenů

- vezmeme tabulku (1M integerů v náhodném pořadí)
- sledujeme 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)

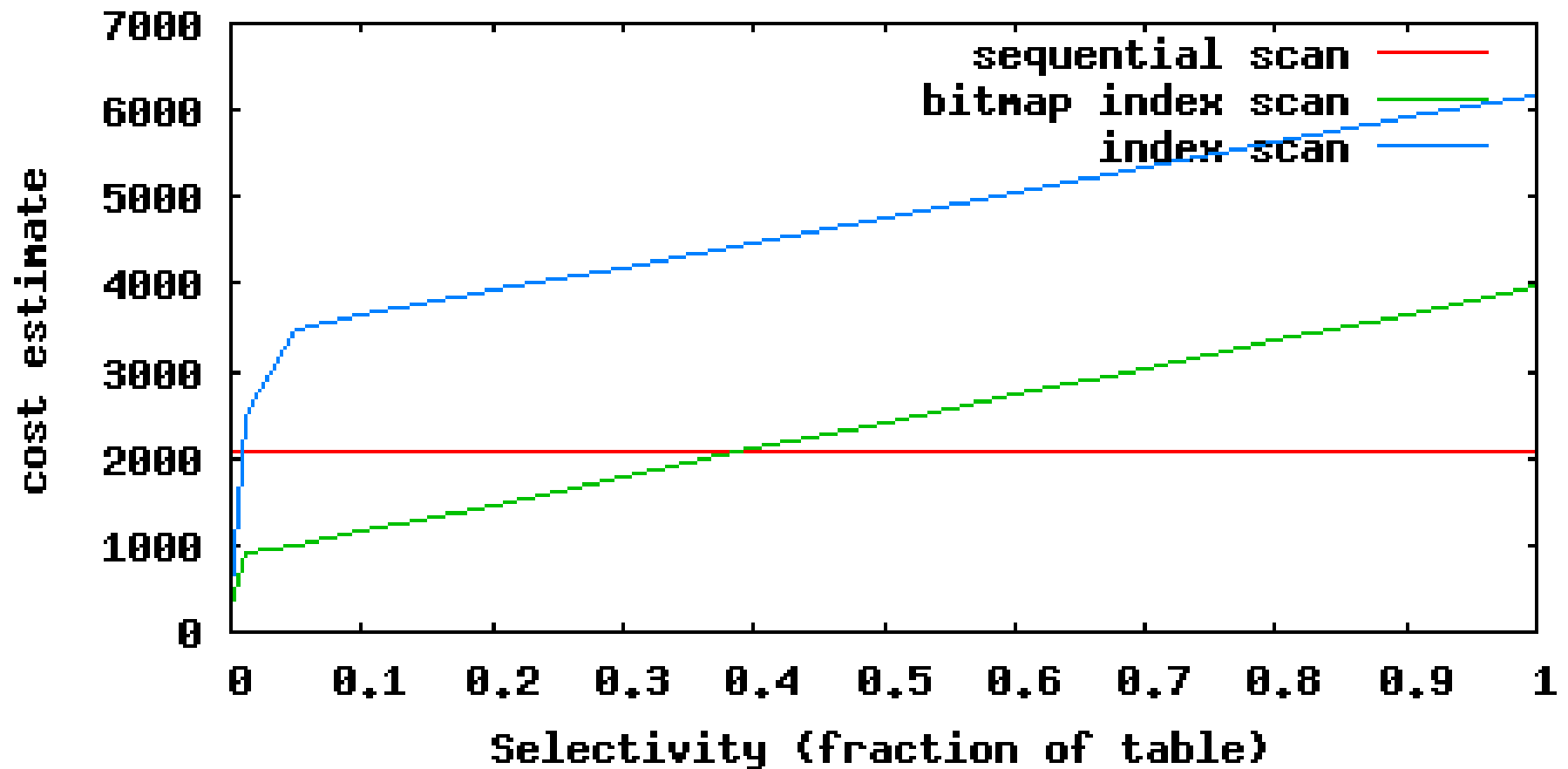
```
CREATE TABLE a AS SELECT i, md5(i::text) m
                        FROM generate_series(1,1000000) s(i)
                        ORDER BY random();
```

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

```
SELECT * FROM a WHERE i < (1000000 * selektivita);
```

Srovnání skenů

Cost of various scans by selectivity (random)



Function Scan

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

```
CREATE FUNCTION moje_tabulka(n INT) RETURNS SETOF INT AS $$  
DECLARE  
    i INT := 0;  
BEGIN  
  
    FOR i IN 1..n LOOP  
        RETURN NEXT i;  
    END LOOP;  
  
    RETURN;  
  
END;  
$$ LANGUAGE plpgsql COST 10 ROWS 100;
```

CTE Scan

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

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

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)

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

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)

$$\text{startup_cost} + (\text{total_cost} - \text{startup_cost}) * (\text{rows} / \text{limit})$$

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

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

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

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

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
```

Nested Loop

- kartézský součin není příliš obvyklý
- přidejme index a podmínku na jednu tabulku

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

QUERY PLAN

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

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

Nested Loop

```
EXPLAIN ANALYZE SELECT * FROM a JOIN b USING (i) WHERE a.i < 10;  
QUERY PLAN
```

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

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

Hash Join

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

QUERY PLAN

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

- menší relaci načte do hash tabulky (pro rychlé vyhledání podle join klíče)
 - pokud se nevejde do work_mem, rozdělí ji na tzv. "batche"
- následně čte větší tabulku a v hash tabulce vyhledává záznamy
 - velká tabulka se batchuje "odpovídajícím" způsobem
 - řádky prvního batche se zjoinují rovnou
 - ostatní se zapíší do batchů (temporary soubory, může znamenat I/O)

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
```

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

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

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)

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

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

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)
```

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;
```

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?