Relační databáze versus CPU

Pavel Stěhule
COMMON, ČR a Slovensko
Dolní Morava 2018

Historie - 70 léta

- System R - překlad do strojového kódu
  - parsování, kontroly, optimalizace v preprocesoru
- Ingres - interpret
  - problémy s pamětí
    - optimalizátor v Lispu (následně přepis do C), zbytek v C
  - následně embedded QUEL

Historie 80léta - současnost

- běžné OLTP - dostatek RAM a dostatečně rychlá CPU eliminují potřebu kompilace do strojového kódu:
  - problémy s přenositelností
  - vyšší složitost systému, latence
  - hrdlem jsou IO operace
  - běžně se používá interpretované SQL
  - některé vzory přetrvávají: `PREPARE, EXECUTE`

Po roce 2000

- In memory (paměťové) databáze
  - extra rychlé OLTP, interaktivní OLAP
  - data jsou uložená primárně v RAM
  - hrdlem se stává CPU, případně čtení dat z RAM
  - SAP HANA, IBM SolidDB, HyPer, MonetDB

Možná řešení

- JITizace interpretů
  - PostgreSQL - llvm
  - SQLite - RPython
- Sloupcový, vektorový executor
  - MonetDB, VectorWise

Řádkový iterační executor

- Snahou je úspora paměti, jednoduchá, robustní a rychlé zastavení výpočtu
- Počítá se po řádcích (řádek obsahuje ntice)
- Prováděcí program je orientovaný graf
  - ke každému uzlu existuje ovladač (handler)
  - každý uzel má své potomky, odkud bere data
  - každý uzel má svého rodiče, kam posílá data
  - implementováno jako pomocí příkazu `switch`
    nebo nepřímým volání funkce

Pull executor (iterátor)

```
void *
limit(plannode *node)
{
    LimitNode *ln = (LimitNode *) node;

    if (ln->iterations++ == ln->max_iteration)
        return NULL;

    /* nepřímé (virtuální) rekurzivní volání */
    return eval_node(ln->left_node);
}
```

Sloupcový executor

- Optimalizováno pro co nejrychlejší výpočet
- Prováděcí plán je posloupnost sloupcových funkcí
- Operace jsou definovány na sloupcích (vektorech)
- Výpočty po sloupcích jsou náročné na paměť, efektivnější je výpočet po vektorech, tak aby se funkce spočítala v rámci L1 cache (vektor je část sloupce)

Funkce sum

```
/* řádkový výpočet */
void sum_accu(int *r, int *b)
{
        *r = *r + *b;
}
```

```
/* sloupcový výpočet */
void sum(int *r, int *a, int n)
{
    int i;
    for (i = 0; i < n; i++)
        *r = *r + *a++;
}
```

Sloupcové databáze

- Optimalizováno pro OLAP
- Optimalizováno pro SELECT (agregační funkce)
- Data jsou uložena po sloupcích - dobře spolupracuje se sloupcovým executorem. Data je možné efektivně a jednoduše komprimovat (delta, RLS, ..).

PostgreSQL

- Klasická Unix client/server architektura
- Do verze 9.5 - 1 proces, 1 session, 1 dotaz, 1 CPU
- OLTP, ACID, MGA (MVCC) - předpokládané úzké hrdlo: IO operace, zámky
  - více uživatelské náročnější dotazy
  - podpora široké škály datových typů včetně vlastních
  - podpora několika typů indexů včetně vlastních
  - možnost zpracovávat velké spektrum úloh na straně serveru (extenze)
  - iterační executor (interpret) + interpret výrazů

PostgreSQL - nové trendy

- Neatomické datové typy
  - hstore, XML, Json, Jsonb
- Paralelizace zpracování dotazu - více CPU pro dotaz
  - paralel seq scan, index scan, hashjoin ... (pg9.6)
- JIT
  - refaktoring expr. executoru - nepřímé volání fce, čtení/zápis cache datových stránek (pg10)
  - Podpora *llvm IR* JIT (pg11) pro výrazy
  - Podpora JIT pro relační operace (pg12)

Hw akcelerace

- PG-Strom - `JOIN`, `GROUP BY`
  - `GpuScan`, `GpuHashJoin`, `GpuNestLoop`, `GpuPreAgg`, `GpuSort`
- PL/CUDA - možnost využít CUDA na straně serveru
  - výpočetně náročné funkce
  - machine learning
  - hledání podobností řetězců

PostgreSQL - budoucnost

- Některé pomalé dotazy jsou způsobené špatnými odhady (korelace mezi sloupci, korelace mezi tabulkami), výsledkem je neoptimální plán.
- Adaptivní planner - po pomalém dotazu se opraví statistiky
- Adaptivní executor - možnost dynamicky měnit způsob provedení dotazu - nyní *quick sort*$\rightarrow$*external sort*, v budoucnu *nested loop*$\rightarrow$*hash join*.
- Push executor - prováděcí plán dotazu je klasickým programem, nikoliv grafem - snáze se JITuje, méně missprediction, možnost využití registrů

Push executor

```
/*
 * inmemory - lepší predikce
 * možnost držet hodnoty
 * v registrech + SIMD
 */
r[0:9] = 0
for each tuple t1 in foo
  if t1.rating > 0.5
    if t1.cost > r[9].cost
      for i in 0..9
        if t1.cost >= r[i].cost
          r[i+1:9] = r[i:8]
          r[i] = t1
```

```
/*
 * iteration model
 */
limit 10
  sort by cost
    seqscan foo
      filter(rating > 0.5)
```

Postgres a JIT

- `configure --with-llvm`
- benefity by měly být zřejmé u složitějších výrazů nebo při práci se širšími tabulkami

Výchozí konfigurace

```
postgres=# select name, setting from pg_settings where name like 'jit%';
+-------------------------+---------+
|          name           | setting |
+-------------------------+---------+
| jit                     | on      |
| jit_above_cost          | 100000  |
| jit_debugging_support   | off     |
| jit_dump_bitcode        | off     |
| jit_expressions         | on      |
| jit_inline_above_cost   | 500000  |
| jit_optimize_above_cost | 500000  |
| jit_profiling_support   | off     |
| jit_provider            | llvmjit |
| jit_tuple_deforming     | on      |
+-------------------------+---------+
(10 rows)
```

Ukázka - příprava test. dat

```
postgres=# create table foo(id int, v integer);
CREATE TABLE
postgres=# insert into foo select random()*10 from generate_series(1,10000000);
INSERT 0 10000000
postgres=# analyze foo;
ANALYZE
postgres=# \dt+ foo
                    List of relations
+--------+------+-------+-------+--------+-------------+
| Schema | Name | Type  | Owner |  Size  | Description |
+--------+------+-------+-------+--------+-------------+
| public | foo  | table | pavel | 346 MB |             |
+--------+------+-------+-------+--------+-------------+
(1 row)
```

Ukázka - bez JIT

```
postgres=# explain analyze
              select case when v = 1 then v + 1 when v = 2 then v + 2 when v = 3 then v + 3
                          when v = 4 then v + 4 when v = 5 then v + 5 when v = 6 then v + 6
                          when v = 7 then v + 7 when v = 8 then v + 8 when v = 9 then v + 9
                          when v = 10 then v + 10 end from foo;
+-------------------------------------------------------------------------+
|                            QUERY PLAN                                   |
+-------------------------------------------------------------------------+
| Seq Scan on foo  (cost=0.00..644250.88 rows=10000048 width=4)           |
|                  (actual time=1.571..2236.289 rows=10000000 loops=1)    |
| Planning Time: 0.140 ms                                                 |
| Execution Time: 2609.073 ms                                             |
+-------------------------------------------------------------------------+
(3 rows)
```

Ukázka - s JIT

```
+-------------------------------------------------------------------------------+
|                                QUERY PLAN                                     |
+-------------------------------------------------------------------------------+
| Seq Scan on foo  (cost=0.00..644250.88 rows=10000048 width=4)                 |
|                   (actual time=72.586..1348.023 rows=10000000 loops=1)        |
| Planning Time: 0.140 ms                                                       |
| JIT:                                                                          |
|   Functions: 2                                                                |
|   Generation Time: 2.387 ms                                                   |
|   Inlining: true                                                              |
|   Inlining Time: 6.185 ms                                                     |
|   Optimization: true                                                          |
|   Optimization Time: 38.875 ms                                                |
|   Emission Time: 25.858 ms                                                    |
| Execution Time: 1715.517 ms                                                   |
+-------------------------------------------------------------------------------+
(11 rows)
```

JIT

- cca o 1/3 rychlejší na 1 CPU
- pokud se použije více CPU pak je to 1/(3 * CPU)
- výraznější efekt v případě komplexnějších výrazů
  - náročnější optimalizace, inlining

Příprava dat

```
postgres=# select 'create table wt(' ||
              string_agg(format('%I integer', 'c' || i),',') || ')'
              from generate_series(1, 50) g(i) \gexec
CREATE TABLE

postgres=# \dt+ wt
                    List of relations
+--------+------+-------+-------+--------+-------------+
| Schema | Name | Type  | Owner |  Size  | Description |
+--------+------+-------+-------+--------+-------------+
| public | wt   | table | pavel | 223 MB |             |
+--------+------+-------+-------+--------+-------------+
(1 row)
```