Různé způsoby optimalizace zpracování SQL dotazů

Pavel Stěhule

P2D2 2024

Historie

- 60-½70 db první generace (pásky, hierarchický model, COBOL)
  - Databáze je soubor, program někým napsaný čte ze souboru a zapisuje do souboru,
  - Nenáročné na zdroje, jakékoliv změny jsou problém (schéma, defragmentace, lokalita)
- ½70 – 2000 db druhé generace (rotační disky, relační model, Oracle, DB2, Postgres)
  - Databáze je samostatný program (knihovna) a data
  - Snaha o maximální oddělení databáze a aplikace (jazyková (SQL), datová (formáty, způsob přístupu), síťová (ODBC), ... nezávislost (relace $\rightarrow$ relace)
  - Mnohem náročnější na zdroje (snaha o optimalizace)

Definice struktury

```
#include <stdio.h>
#include <stdlib.h>

typedef struct MyCompiste
{
        unsigned long id;
        double a1;
        double a2;
        double a3;
} MyComposite;
```

Otevření souboru

```
FILE       *file;
MyComposite row;
unsigned long i;

file = fopen("data.dat", "wb");
if (!file)
{
    perror("fopen: ");
    return -1;
}
```

Zápis do souboru

```
for (i = 0; i < 10000000; i++)
{
    row.id = i;
    row.a1 = drand48() * 100000;
    row.a2 = drand48() * 1000000;
    row.a3 = drand48() * 10000000;

    if (fwrite(&row, sizeof(MyComposite), 1, file) == -1)
    {
        perror("fwrite: ");
        return -1;
    }
}
```

Zavření souboru

```
if (fclose(file) != 0)
{
    perror("fclose: ");
    return -1;
}
```

SQL

```
INSERT INTO data
  SELECT i, random()*100000, random()*1000000, random()*10000000
    FROM generate_series(1, 10000000) g(i);

SELECT avg(a1), avg(a2), avg(a3)
  FROM data;
```

Porovnání

	velikost	Zápis	Čtení
C	306MB	1.2s	0.32s
SQL	575MB	16s	0.57s

Porovnání

```
Finalize Aggregate
  ->  Gather
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate
              ->  Parallel Seq Scan on data
```

Porovnání

	velikost	Zápis	Čtení	Čtení (1pJIT)	Čtení (1p)
C	306MB	1.2s	0.32s		
SQL	575MB	16s	0.57s	1.6s	1.8s

Porovnání

	velikost	Zápis	Čtení	Čtení (1pJIT)	Čtení (1p)	MySQL (Zápis)	MySQL (Čtení)
C	306MB	1.2s	0.32s				
SQL	575MB	16s	0.57s	1.6s	1.8s	5min*	4.8s

* MySQL - pomalé díky implicitnímu indexu (InnoDB) a pomalým SP,
V pg obdobný procedurální insert s jedním indexem 2.23min

Porovnání

	velikost	Zápis	Čtení	Čtení (1pJIT)	Čtení (1p)	MySQL (Zápis)	MySQL (Čtení)
C	306MB	1.2s	0.32s				
SQL	575MB	16s	0.57s	1.6s	1.8s	5min*	4.8s

	velikost	DuckDB (Zápis)	DuckDB (Čtení 8thr)	DuckDB (čtení 1thr)
SQL	206MB	3sec	0.05s	0.254s

Použití databáze 2. generace

- Znamená 10-100 pomalejší provedení úloh než s db 1. generace,
- Cena za paralelní přístup k datům, ochraně proti race condition, dynamické práci se schématem, nezávislosti na formátu, zabezpečení, ...

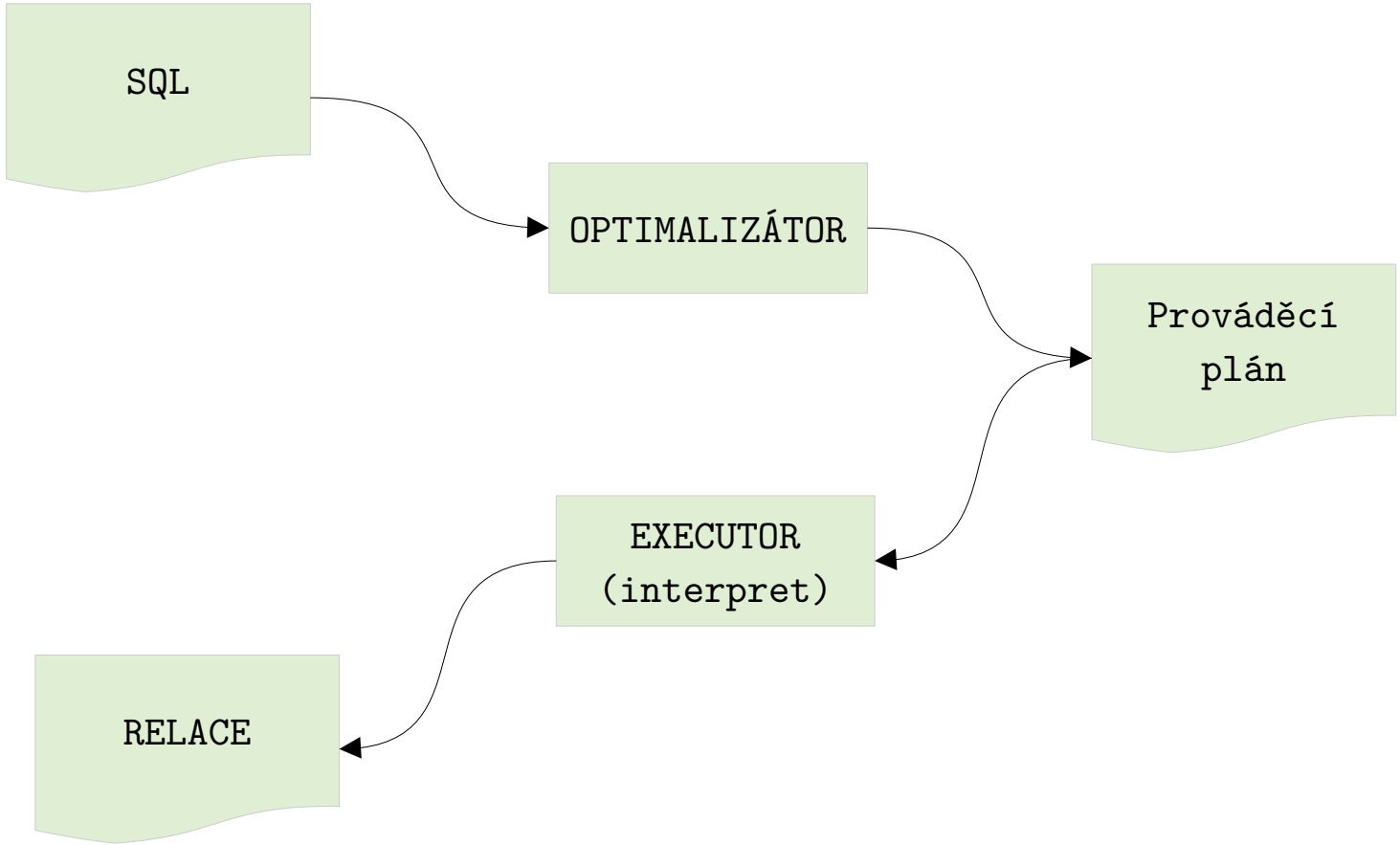
Očekávaný výkon

- Začátkem 90 let – 100tps, 1000 uživatelů, 6násobný JOIN
- Dnes – 5000-10000tps, 10000 uživatelů, 16 násobný JOIN

Použití databáze 2. generace

- Dynamické vytvoření kódu pro výpočet dotazu (generování prováděcího plánu)
- Dynamické vykonávání (interpretace) kódu dotazu (databáze neobsahují překladače do strojového kódu)

Použití databáze 2. generace



Prováděcí plán (Dotaz)

```
SELECT *
  FROM obce
 ORDER BY pocet_zen + pocet_muzu DESC
 LIMIT 10;
```

Prováděcí plán

(Plán)

```
                                QUERY PLAN
═══════════════════════════════════════════════════════════════════════════
Limit  (cost=272.19..272.21 rows=10 width=45)
  ->  Sort  (cost=272.19..287.81 rows=6250 width=45)
        Sort Key: ((pocet_zen + pocet_muzu)) DESC
        ->  Seq Scan on obce  (cost=0.00..137.12 rows=6250 width=45)
```

Problémy s generovaným prováděcím plánem

- Rule based optimizer (RBO, Oracle 8-10)
  - Magie se zápisem dotazu
- Cost based optimizer (CBO, Postgres)
  - Špatné odhady, špatné plány
    - Odhad procentem (chybějící statistiky – výrazy, JOIN)
    - Závislosti mezi sloupci, nehomogenní data v tab
    - Závislost na magických konstantách `cpu_page_cost`, `random_page_cost`, `cpu_operator_cost`, ...
    - Různě rychlé plány mohou mít podobnou cenu (Postgres nebere ohled na cache)

Problémy s generovaným prováděcím plánem

- Cost based optimalizer
  - Každá operace má vzorec pro kalkulaci ceny
  - Celková cena je součet cen všech operací
  - Vybírá se plán s nejnižší cenou

Problémy s generovaným prováděcím plánem

- Předpřipravené dotazy (Prepared statements) – opakované použití vygenerovaného prováděcího plánu
  - Snižují režii generování plánu
  - Přinášejí riziko použití vygenerovaného plánu pro jiné než aktuálně použité parametry (problém s invalidací plan cache)
  - Magie – client side PP X server side PP,
    custom pl. (5x), custom pl. (forever) X generic pl.

Problémy s generovaným prováděcím plánem

- Chytřejší optimalizátor – více metod exekutoru – rychlejší zpracování dotazu (v lepším případě) a naopak pomalejší zpracování dotazu (v horším případě)
- Hloupý optimalizátor – méně metod executoru (menší citlivost na chyby odhadu) – (Vertica) – chytrost je nutné dohnat silou (rychlostí exekutoru)
  `set enable_nestloop to off`

Problémy s generovaným prováděcím plánem

```
CREATE TABLE kalendar(id int PRIMARY KEY, val date);
CREATE TABLE data(val int, kalendar_id int references kalendar(id));

INSERT INTO kalendar VALUES(1, '2020-04-15'),(2, '2021-04-15');
INSERT INTO data SELECT random()*100, 1 FROM generate_series(1,10000);
INSERT 0 10000
ANALYZE kalendar, data;
```

Problémy s generovaným prováděcím plánem

```
 EXPLAIN SELECT *
    FROM data JOIN kalendar k ON kalendar_id = k.id
   WHERE k.val = '2021-04-15';
                              QUERY PLAN
```

```
Hash Join  (cost=1.04..227.91 rows=5000 width=16)
  Hash Cond: (data.kalendar_id = k.id)
  ->  Seq Scan on data  (cost=0.00..145.00 rows=10000 width=8)
  ->  Hash  (cost=1.02..1.02 rows=1 width=8)
        ->  Seq Scan on kalendar k  (cost=0.00..1.02 rows=1 width=8)
              Filter: (val = '2021-04-15'::date)
(6 rows)
```

Problémy s generovaným prováděcím plánem

```
CREATE TABLE data(a int, b int);

INSERT INTO data SELECT v, v
  FROM (SELECT random()*10000 AS v FROM
generate_series(1,100000));

ANALYZE data;
```

Problémy s generovaným prováděcím plánem

```
EXPLAIN ANALYZE SELECT * FROM data WHERE a BETWEEN 10 AND 100;
                                                  QUERY PLAN
```

```
Seq Scan on data  (cost=0.00..1943.00 rows=941 width=8) (actual rows=889 loops=1)
  Filter: ((a >= 10) AND (a <= 100))
  Rows Removed by Filter: 99111
(5 rows)

EXPLAIN ANALYZE SELECT * FROM data WHERE a BETWEEN 10 AND 100 AND b BETWEEN 10 AND 100;
                                                  QUERY PLAN
```

```
Seq Scan on data  (cost=0.00..2443.00 rows=9 width=8) (actual rows=889 loops=1)
  Filter: ((a >= 10) AND (a <= 100) AND (b >= 10) AND (b <= 100))
  Rows Removed by Filter: 99111
```

Problémy s generovaným prováděcím plánem

- Zlepšení odhadů
  - Vícesloupcové statistiky (PostgreSQL 10)
  - Podpora vícesloupcových MCV (PostgreSQL 12)
  - Funkcionální statistiky (PostgreSQL 14)
  - Experimentálně – PostgresPro (AQO) – adaptive query optimization – použití metod ML pro zlepšení odhadů (learning mode, use mode) – primitivní suplování mezi tabulkových statistik (nebere v potaz parametry)
  - Na prasáka – použití falešných immutable funkcí (nesmí dojít k použití generic plan cache), které startuje optimalizátor

Problémy s generovaným prováděcím plánem

- Adaptivní (dymamická) exekuce / Adaptive Query Processing
  - V postgresu pouze fallback stavy (hashjoin / rehash, memoize / off)
  - EXISTS – podporuje alternativní plány
  - Merge join $\rightarrow$ hashjoin (Spark)
  - Hashjoin$\rightarrow$ nestedloop (MSSQL, exp PG)

Problémy s generovaným prováděcím plánem

- Worst Case Optimal Joins (WCOJ)
  - Binární Join X multijoin
  - WCOJ – vždy lepší než nejhorší varianta binárního JOINu – horší pokud JOIN má méně řádek než vstupy, lepší pokud JOIN má více řádek než vstupy

Problémy s generovaným prováděcím plánem

- Robustní plán – hledá se takový plán, který pro všechny možné kombinace parametrů má nejnižší nejvyšší cenu

Zrychlení operací

- Memoize – lokální cache pro korelované poddotazy (JOIN, LATERAL)
- Materialize – lokální cache (pro rescan)
- Experimentálně – bloom filter + hashjoin, bloom filter + mergejoin

Změna formátu

- Sloupcové databáze (komprimace) - DuckDB
- LSM tree – rychlejší zápis, RockDB (exp PG)
- Index only scan (PG9.2)
- Coverigin index (index obsahuje sloupce tab) PG 11

Dynamické provádění (interpretace) plánů

- Query executor
  - Init – z prováděcích plánů vytvoří graf struktur PlanState
  - Run – iterace po PlanState a nepřímé volání určené položkou ExecProcNode
- Expr executor
  - Init (vytváří se pseudocód – opcodes)
  - Překlad opcodes pro llvm, exekuce
  - Interpretace (switch nebo nepřímé goto)

Dynamické provádění (interpretace) plánů

```
limitstate = makeNode(LimitState);
limitstate->ps.plan = (Plan *) node;
limitstate->ps.state = estate;
limitstate->ps.ExecProcNode = ExecLimit;
```

```
static inline TupleTableSlot *
ExecProcNode(PlanState *node)
{
    if (node->chgParam != NULL) /* something changed? */
      ExecReScan(node);         /* let ReScan handle this */

    return node->ExecProcNode(node);
}
```

Dynamické provádění (interpretace) plánů

```
void *
ExecLimit(PlanState *pstate)
{
    LimitState *node = castNode(LimitState, pstate);
    PlanState *outerPlan;

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

    outerPlan = outerPlanState(node)

    /* nepřímé (virtuální) rekurzivní volání */
    return ExecProcNode(outerPlan);
}
```

Kompilace plánů

- Umbra (navazuje na db HyPer)
  - Query plan $\rightarrow$ Umbra IR
  - Optimalizující překladač LLVM
  - Flying start (neoptimalizující překladač do x86) $\rightarrow$ asmJIT

Push vectorized executor

```
static void Sum(DataChunk &args,
                ExpressionState &state, Vector &result)
{
    result_data = GetData(result);
    result_mask = Validity(result);

    for (col_idx = 0; col_idx < args.ColumnCount(); i++)
    {
       args.data[col_idx].ToUnifiedFormat(args.size, vdata);
       input_data = UnifiedVectorFormat(vdata);

       for (i = 0; i < args.size(); i++)
         if (vdata.validity.RowIsValid(i))
           result_data[i] += input_data[i];
         else
           result_mask.SetInvalid(i);
    }
}
```

Rekurzivní executor (expr)

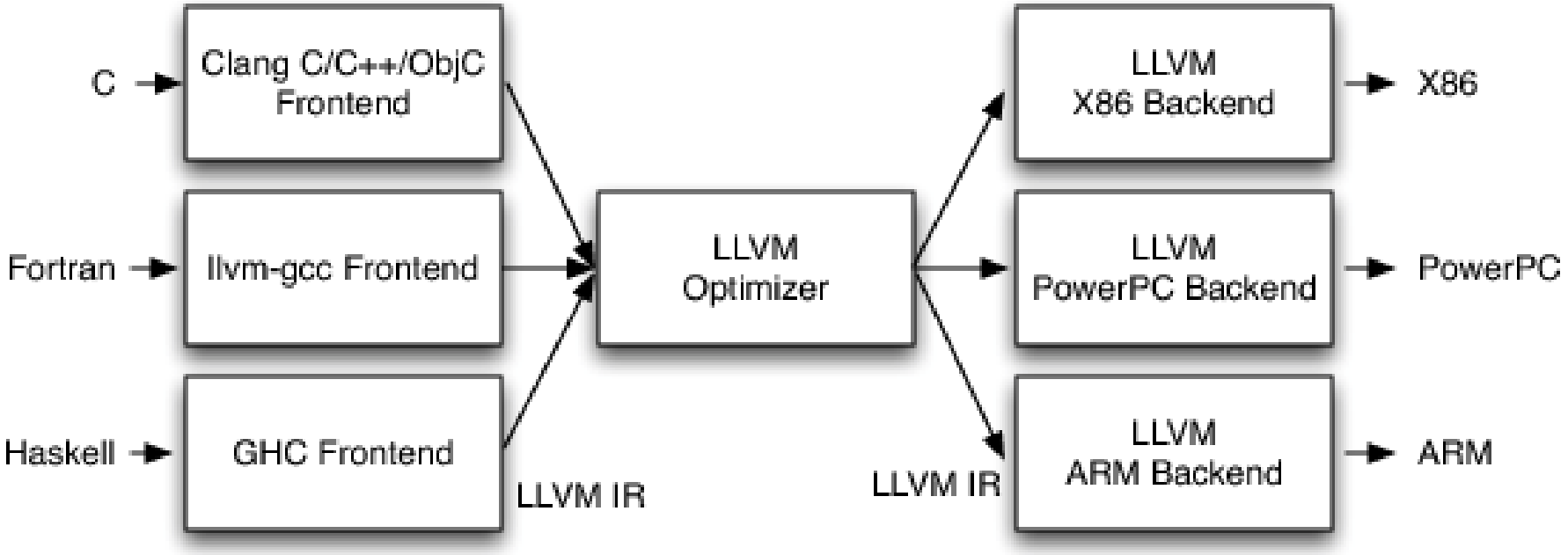
```
Datum
sum_int(FunctionCallInfoData *fcinfo)
{
        int a1, a2;
        /* pokud je některý z parametrů NULL, pak
           výsledek je NULL */
        if (fcinfo->argnull[0] || fcinfo->argnull[1])
        {
                fcinfo->isnull = true;
                return (Datum) 0;
        }

        a1 = DatumGetInt32(fcinfo->arg[0]);
        a2 = DatumGetInt32(fcinfo->arg[1]);
        fcinfo->isnull = false;
        return Int32GetDatum(a1 + a2);
}
```

Low Level Virtual Machine

- Univerzální prostředí pro tvorbu překladačů z libovolného vyššího jazyka do strojového kódu
- Clang – překladač C/C++ v tomto prostředí
- Překládá se do IR – pseudo RISC assambler
- IR se posléze překládá pro cílovou platformu

LLVM

intermediate representation



LLVM

intermediate representation

```
@.str = internal constant [14 x i8] c"Hello, world\0A\00"
declare i32 @printf(ptr, ...)
define i32 @main(i32 %argc, ptr %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], ptr @.str, i32 0, i32 0
    %tmp2 = call i32 (ptr, ...) @printf( ptr %tmp1 ) nounwind
    ret i32 0
}
```

JIT

- LLVM optimalizace je náročná a pomalá (několika průchodová)
- LLVM se volá vždy při každém spuštění (neexistuje cache)
- LLVM se volá pro každý proces (pokud se v dotazu použije n workerů, tak n-krát)
- Existují rychlejší překladače (MIR) – ale zatím nepodporované (nutný je překlad Postgresu)

JIT

- Zrychlení dotazu až o 30%
- Zpomalení rychlých krátkých dotazů o stovky ms
- Možnost použití více CPU přináší větší benefit (poslední roky se výrazně více času investovalo do podpory více CPU pro dotaz)
- Problémy s kompatibilitou různých verzí llvm
- V Postgresu je náročné předávání parametrů funkce (V1 volající konvence), pro efektivní volání vyžaduje inline a optimalizaci

Tuple deforming

- Transformace z formátu v jakém jsou data uložená v datových stránkách do formátu, v kterém se zpracovávají executorem
- Náhrada počtu sloupců konstantou
- Náhrada šířky a typu sloupce konstantou
- Redukce kontroly existence bitmapy s NULL

Inlining

- Každý operátor je ve výsledku funkce
- Zdrojové kódy funkcí jsou přeložené a připravené při překladu Postgresu ve formátu bitcode (binární reprezentace llvm IR)
- Způsob předávání parametrů v PG redukuje výhody inliningu, bez optimalizace malý efekt – první iterace pomalá (načtení velkého množství bitcode), je nezbytná optimalizace (která je pomalá)

JIT konfigurace

- `jit` (on)
- `jit_above_cost` (100000) cca 800MB tab
- `jit_inline_above_cost` (500000) cca 4GB
- `Jit_optimize_above_cost` (500000)

JIT problémy

- Opraveno několik memory leaku
- Aktivace na ceně dotazu, která nedostatečně dobře kalkuluje náročnost optimalizace
- Použitím více CPU klesá benefit JIT, a roste režie JIT – aktivace JIT toto nebere v potaz
- Špatný odhad může způsobit chybnou aktivaci JIT (kalkulace rekurzivních CTE - recuesive_worktable_factor)
- JIT optimalizace se týká všech výrazů – i těch, které nejsou používané často nebo nejsou použité vůbec (subplány)

Push vectorized executor (DuckDB)

- Push based executor – od zdrojů k operacím, jednodušší implementace paralelního zpracování (není rekurzivní)
- Vectorized – místo řádků se zpracovávají vektory (ntice řádků)

Trendy

- Čistě inmemory databáze jako např. MonetDB jsou pro OLAP ekonomicky neefektivní
- Pro maximální výkon se kombinuje hybridní uložiště sloupcových db (data jsou uložena jako bloky sloupců – pro lepší komprimaci) a executor z inmemory db (čtení dat z SSD a NVMe disků už je příliš rychlé na executor typu Volcano)
- Vektorizace se přidává i do klasických db – batch mode (SQL server 2012, 2019) – batch ~900 řádků (64kB)– cca 3x rychlejší v některých dotazech (masivní agregace)

Dotazy

- ??