Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Integrace a využití JIT překladače a prostředí v PostgreSQL pro OLAP využití

Bc. Ondřej Psota

Vedoucí práce: Ing. Pavel Stěhule

12. května 2015

Poděkování

Rád bych poděkoval svému konzultantovi a vedoucímu práce panu Ing. Pavlu Stěhulemu za jeho cenné rady a vstřícný přístup při řešení problémů. Dále bych chtěl rovněž poděkovat všem, kteří mi byli nápomocni a vytvořily podmínky pro to, abych se své práci mohl plně věnovat.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Psota, Ondřej. *Integrace a využití JIT překladače a prostředí v PostgreSQL pro OLAP využití.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato práce se zabývá integrací existujícího Just-In-Time kompilátoru do databáze PostgreSQL, který v přesně definovaných případech nahrazuje standardní interpret AST výrazů. Je stanovena úvodní motivace včetně výhod pro OLAP využití. V textu jsou popsány vybrané interní mechanizmy specifické pro danou databázi. Postup je takový, že se analyzuje vnitřní struktura interpretů. Získané poznatky ovlivňují výběr konkrétního řešení, které musí splňovat požadavky na přiměřenou obtížnost integrace a co nejvyšší rychlost.
Vybrané řešení ve formě LibJit knihovny splnilo očekávání jen částečně. Integrace probíhala poměrně přímočaře až na problémy s debugováním a hledáním chyb. Výsledky testování byly spíše zklamáním. Převažovaly případy, kdy naopak došlo k prodloužení doby zpracování dotazu.

Klíčová slova PostgreSQL, OLAP, prováděcí plán, Just-in-time kompilace, expression exekutor, LibJit, návrh integrace, TPC-H benchmark

Abstract

This diploma thesis focuses on the integration of existing Just-In-Time compiler into the PostgreSQL Database which in precisely defined cases replaces the standard interpreter of AST expressions. An original motivation including

advantages for OLAP systems usage is determined. Selected internal mechanisms specific for particular database are described. The procedure is based on the interpreters internal structure analysis. Obtained results influence particular solution selection which must meet the requirements for acceptable integration difficulty and the maximum speed.
Selected solution in the form of LibJit library fulfilled the expectations only partially. The integration process was quite direct except for problems with debugging and searching for errors. Results of testing were rather disappointing. There were more cases in which the time of task proceeding was prolonged.

Keywords PostgreSQL, OLAP, database plan, Just-in-time compilation, expression executor, LibJit, design of integration, TPC-H benchmark

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Rychlost vykonávání dotazů v PostgreSQL je zejména limitována IO operacemi. CPU naopak do výsledného času přispívá jen malou měrou a často lze takové hodnoty zanedbat. Tak jak jde dopředu vývoj počítačových komponent, zvyšuje se také kapacita operační paměti. Proto je v ní možné uchovávat stále více dat a eliminovat tak počet neefektivních diskových operací. V nejlepším případě by data mohla být v paměti uložena všechna. Vliv CPU by se rapidně zvýšil. Velice by záleželo na jeho rychlosti a také na sekvenci instrukcí, na kterou by byl zaslaný dotaz přeložen. Tato práce se zabývá zmíněným druhým bodem týkajícího se generování instrukcí. Problém se snaží přiblížit navazující odstavec.

Databáze PostgreSQL v současnosti využívá minimálně dvou interpretů. Interpret prováděcích plánů, jak sám název již napovídá, prochází sestavený strom prováděcího plánu. Postupně volá obslužné metody, jenž zpracovávají konkrétní typy uzlů. Druhý interpret AST výrazů slouží k vyhodnocování netriviálních hodnot sloupců, které jsou výsledkem operací provedenými nad jedním či více atributy. Stejně jako v předchozím případě dochází k průchodu určitého stromu. Obslužné funkce zpracovávají uzly, které dohromady reprezentují složitější (matematický) výraz.

Přístup založený na interpretaci zdaleka nemůže konkurovat překladu do nativního kódu cílové platformy. Proto byly do známých virtuálních strojů zaintegrovány Just-in-time kompilátory, které mají za úkol provést zmíněný překlad. Rozdíly ve výkonu mohou být velice výrazné. Například zkompilovaný kód skriptovacího jazyka Lua může být až padesátkrát rychlejší než jeho interpretovaná varianta. Zmíněný fakt potvrzuje graf 4.5 v kapitole 4.1. Naskytuje se tu přirozeně otázka, zda by PostgreSQL nemohl využívat JIT kompilátor pro překlad databázových dotazů? Samozřejmě, že mohl, a proč rovnou nevyužít již hotového řešení. Pomalu se dostáváme k hlavní náplni této práce.

Kapitola 1 se zabývá vybranými interními mechanismy, které jsou typické pro databázi PostgreSQL. Mimo jiné řeší otázku, kdy je lepší data ukládat po sloupcích a kdy po řádcích. Kapitola 2 se snaží přiblížit průběh zpracování do-

tazu a podrobněji se zaměřuje na oba dva zmíněné interprety. Speciální péči věnuje analýze struktury AST interpretu, protože vybraný JIT kompilátor bude v přesně definovaných bodech nahrazovat jeho funkcionalitu. Kapitola 3 nejprve ozřejmuje možnosti 64 bitové architektury procesorů, na které bude postavena a otestována realizovaná integrace. Protože úspěch výsledného řešení stojí a padá na vybraném Just-in-time kompilátoru, značná část kapitoly 3 se věnuje jejich analýze. Ze začátku se zaobírá vestavěnými řešeními v rámci virtuálních strojů. Postupně se však dostane k popisu zcela nezávislých JIT kompilátorů. U každého z nich jedna podkapitola přibližuje a nastiňuje možné provedení integrace. První část kapitoly 4 se zabývá implementací testovacích algoritmů určených pro porovnání jednotlivých Just-in-time kompilátorů, případně interpretů. Přehledné grafy v kapitole zobrazují rozdíly v rychlostech. Na základě předchozí analýzy a porovnání výpočetního výkonu byla vybrána nezávislá LibJit knihovna. Hned v návaznosti dochází k návrhu integrace. Závěrečná část kapitoly 4 se zaměřuje na vybrané implementační detaily a případné problémy. Vyvrcholení vynaložené píle se odehraje v kapitole 5 věnující se testování. Uvnitř je popisován návrh a provedení vhodného TPC-H benchmarku se zaměřením na složité výrazy. Porovnává se rychlost zpracování dotazu tří různých implementací (standardní databáze, databáze s integrovaným JIT kompilátorem, databáze se sloupcovým uložením dat na disku) Dosažené výsledky jsou zobrazeny v přehledných tabulkách a grafech. Kapitolu uzavírá diskuse naměřených výsledků.

OLTP vs OLAP

Práce se převážně zaměřuje na využití v OLAP [1] systémech. Z tohoto důvodu je při testování v kapitole 5 využito TPC-H benchmarku zaměřeného na analytické výpočty pomocí agregačních funkcí.

OLAP (Online Analytical Processing) je technologie uložení dat v databázi, která umožňuje uspořádat velké objemy dat tak, aby byla data přístupná a srozumitelná uživatelům zabývajícím se analýzou obchodních trendů a výsledků. Způsob uložení dat se svým zaměřením liší od běžněji užívaného OLTP (Online Transaction Processing), kde je důraz kladen především na snadné a bezpečné ukládání změn v datech v konkurenčním (víceuživatelském) prostředí. OLAP technologie pracuje s tzv. multidimenzionálními daty . Na rozdíl od dvourozměrného uložení dat v relačních databázích (sloupec, řádek). Jestliže OLAP využívá k analýze přímo data uložená v relační databázi, hovoříme o relačním OLAPu - ROLAP [1] . OLAP v tomto případě v relační databázi vytvoří další tabulky pro uložení agregací, protože návrh OLTP databází nevyhovuje požadavkům ROLAPu. Samotná analýza je náročná z důvodu nutnosti používání standardních SQL příkazů k načtení dat z tabulek propojenými cizími klíči. Zobrazení analyzovaných dat je již ovšem multidimenzionální.

Kapitola 1

Vybrané interní mechanismy PostgreSQL

V úvodní podkapitole 1.1 se čtenář seznámí s databází PostgreSQL včetně důležitých položek v systémovém katalogu. Navazující podkapitoly se snaží přiblížit důležité interní mechanismy databáze, na které se budeme v textu odkazovat. Kapitolu uzavírá podkapitola 1.7 zabývající se způsobem ukládání dat na disku (po řádcích, po sloupcích).

1.1 PostgreSQL

PostgreSQL[2] je plnohodnotný objektově - relační databázový systém jehož vývoj trvá již přes sedmnáct let. Postupně si vybudoval vynikající pověst za svou bezpečnost a spolehlivost. Vývojáři se snaží co nejvíce respektovat standard ANSI SQL. Databáze běží nativně na většině známých operačních systémech (Linux, různé Unixy a v neposlední řadě na Windows) a nabízí mnoho moderních rysů:

- komplexní dotazy
- cizí klíče
- triggery
- pohledy
- transakce
- MVCC (multiversion concurrency control).

Uživatelé mohou databázi rozšířit dle potřeby například o další datové typy, funkce, operace, agregační funkce, procedurální jazyky, atd. Uložené procedury lze psát v několika programovacích jazycích. (Perl, Pyhon, v jazyku C, PL/-pgSQL, který vycházejí z PL/SQL od Oraclu).

1.1.1 Systémový katalog

V dalších několika kapitolách budou postupně rozebírány vybrané interní mechanizmy, ve kterých se několikrát vyskytne pojem systémový katalog [2]. Jelikož se jedná o podstatnou část databáze PostgreSQL a jednotlivé hodnoty jsou využity ve vlastní integraci JIT kompilátoru, zaslouží si vlastní podkapitolu.

Systémový katalog je místo, kde si RDBMS[1] ukládá metadata o schématu ve formě tabulek, nad kterými lze vykonávat dotazy. V žádném případě se nedoporučuje upravovat jejich obsah, jelikož je modifikován automaticky například při tvorbě databází, tabulek, sloupců, indexů, atd. Pro účely této práce postačuje zmínit následující části katalogu:

- pg_database - katalog obsahuje informace o dostupných databázích, vytvořených příkazem *create database*.
- pg_class - pro každou relaci či jinou strukturu podobající se tabulce (indexy, sekvence, pohledy, atd) ukládá informace o názvu, vlastníkovi, typu či o názvu souboru, ve kterém je relace fyzicky na disku uložena.
- pg_type - udržuje informace o jednotlivých datových typech. Základní typy se vytvářejí pomocí příkazu *create type*. Komplexní typy jsou vytvářeny buď příkazem, nebo automaticky při založení nové tabulky. V takovém případě typ reprezentuje řádkovou strukturu tabulky.
- pg_attribute - pro každý sloupec v každé tabulce (attrelid) uchovává informace o jeho jménu (attname), typu (atttypid), číslu sloupce (attnum), atd.
- pg_operator - operátory uložené v katalogu obsahují informace o názvu (oprname), typu levého operandu (oprleft), typu pravého operandu (oprright), typu výsledné hodnoty (oprresult), funkci implementující daný operátor (oprcode), atd.
- pg_proc - tento katalog ukládá informace o funkcích (procedurách) včetně těch agregačních. Pokud atribut *proisagg* nabývá hodnoty true, tak by měl odpovídající záznam existovat také v katalogu pg_aggregate, který uchovává dodatečné informace pro agregační funkce.

Tabulky katalogu z toho seznamu drží pouze základní informace. Ostatní hodnoty jsou uloženy ve zbývajících tabulkách katalogu. Vazbou mezi daty je hodnota Oid - object identificator

[1] RDBMS - Relational database management system

1.2 Programovací jazyk

Z historických důvodů je většina kódu PostgreSQL naspána v jazyku C. Od verze 9.0 podporuje externí moduly implementované v C++. Jazyk C neumožňuje objektové programování a v mnoha ohledech není tolik komfortní. Vývojáři se snažili některé chybějící prvky jazyka nasimulovat a zároveň vytvořili rozsáhlou sadu maker, funkcí, které usnadňují vývoj. Výhodou je naopak vysoká rychlost a široká podpora napříč platformami. [3]

1.3 Typový systém

PostgreSQL používá vlastní typový systém [3]. Interní datový typ Datum slouží pro uložení všech datových typů dostupných z SQL. Jedná se o 4(8) bytový generický typ obsahující ukazatel či hodnotu, která se však musí vejít do 4(8) bytů v případě 32(64) bitové platformy. Ukazatel ve většině případů ukazuje na strukturu nazývanou varlena, nebo na string jazyka C.

Na následujících několika řádcích je ukázáno, jak je v PostgreSQL řešen převod hodnoty typu float na typ Datum:

1. Ke zmíněnému převodu slouží funkce Float4GetDatum(float4 X)

2. Ve vlastním tělu funkce se nachází typ union, pomocí kterého může být sekvence 4 bytů interpretována buď jako float4 hodnota, či jako int32 hodnota.

3. Nyní přichází na řadu volání makra SET_4_BYTES(int32 value), které provede jednoduchou operaci bitového součinu. ((Datum) (value)) & (0xffffffff).

4. Tímto postupem došlo k požadovanému převodu float4 hodnoty na typ Datum. Pro opačnou konverzi slouží funkce DatumGetFloat4(Datum x).

Konverze hodnot typu Int32, Int16, UInt16, Char, atd mají na starosti jednotlivá makra. Nejedná se tedy o funkce. Z pohledu JIT kompilátorů nelze takové řešení považovat za šťastné. Většina z nich umožňuje volat nativní funkce, které jsou již zkompilované. Kde ale získat potřebný ukazatel na funkci? V podstatě není na co ukazovat, jelikož makro pouze nahrazuje dlouhou sekvenci znaků, které na dané místo umístí preprocesor. Řešením je taková makra obalit funkcí, abychom získali potřebný ukazatel. V případě krátkých funkcí se vyplatí provést kompilaci kódu pomocí zvolené JIT knihovny, aby zbytečně nedocházelo k režii spojené s voláním funkce.

Varlena - Předek všech datových typů SQL s variabilní délkou. Dále reprezentuje typ, který se z důvodu větší velikosti nevejde do typu Datum. Struktura není nijak složitá. Prvních n bytů uchovává informaci o délce,

za kterými následuje sekvence bytů s vlastním obsahem. Několik bitů hlavičky uchovává dodatečné informace.

Programátor by měl mít vždy na paměti s jakým konkrétním typem pracuje, protože v typu Datum není žádný prostor pro jeho uložení. Nejedná se zcela o pravdivé tvrzení, protože uvnitř funkce lze typ dohledat v jejím deskriptoru.

Typový systém je pro zákaznická data nezávislý na typovém systému C. Překladač nedokáže identifikovat chybný typ parametru, jelikož se popis funkcí nachází mimo jeho dosah. Z pohledu překladače jsou všechny SQL funkce téměř stejné, odlišují se pouze názvem.

1.4 Předávání parametrů

Parametry lze funkcím předávat dvěma základními způsoby[3, 2]:

Version 0 - Argumenty, návratové hodnoty jsou deklarovány ve stylu jazyka C. Nevýhodou je, že si vývojář musí dávat pozor při volbě typu, který musí odpovídat SQL protějšku. U parametrů předávaných hodnotou nelze odlišit hodnotu NULL od 0. Ze zmíněných důvodů se nedoporučuje tento styl předávání používat.

Version 1 - Parametry se volané funkci předávají pomocí komplexnější struktury nazvané FunctionCallInfoData, která je zobrazena v kódu [1.1].

Výpis 1.1: Struktura FunctionCallInfoData

```
typedef struct{
        FmgrInfo *flinfo;          // deskriptor funkce
        fmNodePtr context;         // kontext volání
        fmNodePtr resultinfo;      // deskriptor výsledku
        bool isnull;               // výsledek je NULL ?
        short nargs;               // počet parametrů
        Datum arg[FUNC_MAX_ARGS];        // parametry
        bool argnull[FUNC_MAX_ARGS];     // jsou NULL?
} FunctionCallInfoData
```

Při vývoji není často zapotřebí pracovat přímo s touto strukturou. Existuje velké množství maker, která programátorům usnadňují život. Následuje jednoduchý příklad1.2:

Výpis 1.2: Použití maker, která získají a vrátí hodnotu z funkce.

```
Datum add_one_float8(PG_FUNCTION_ARGS) {
        float8 arg = PG_GETARG_FLOAT8(0);
        PG_RETURN_FLOAT8(arg + 1.0);
}
```

Funkce typu V_1 nelze volat klasicky. Nejprve je zapotřebí naplnit výše zmíněnou strukturu1.1, což usnadňuje předpřipravené makro InitFunctionCallInfo. Poté již nic nebrání samotnému volání funkce pomocí makra s názvem FunctionCallInvoke(fcInfo). To nedělá nic jiného než, že vezme deskriptor funkce FmgrInfo *flinfo, zjistí z něho adresu funkce a zavolá ji.

1.5 Typ Node

V jazyku C neexistuje dědičnost. Typ Node [3] lze chápat jako pokus o implementaci abstraktní třídy. Jedná se vlastně o jakýsi interface, který umožňuje vytvářet kompozitní typy s podporou serializace, deserializace, porovnání, kopírování a zobrazení. Všechny moduly (parser, optimalizér, exekutor, interní knihovny) obsahují typy, které respektují interface Node. Není žádným překvapením, že pro něj existuje mnoho užitečných maker. Například makro s názvem IsA ověřuje, zda je struktura určitého typu.

1.6 Správa paměti

Programy napsané v jazyku C používají obvykle pro alokaci dynamické paměti funkci malloc. Záleží pouze na programátorovi, kdy paměť opět uvolní. Při vývoji pro PostgreSQL se předpokládá, že si uživatelé budou psát své zákaznické funkce sami. S vysokou pravděpodobností svůj kód vždy dobře nezabezpečí proti úniku paměti. Toto riziko je eliminováno používáním tzv. paměťových kontextů, které lze vnímat jako abstraktní evidenci jednotlivých alokací paměti. Nejedná se o nic jiného než, že V PostgreSQL [3] je dynamická paměť alokována z aktuálního nebo jinak určeného kontextu, k čemuž také slouží funkce palloc. Jednotlivé kontexty mohou tvořit hierarchickou strukturu, čehož se využívá například při exekuci prováděcích plánů či expression výrazů. Řada důležitých procesů v PostgreSQL má přidělen jeden nebo více paměťových kontextů. Před zavoláním každé funkce executorem je vytvořen dočasný kontext, který se prohlásí za aktuální. Uvnitř funkce se pak paměť alokuje z tohoto kontextu, pokud ovšem nedošlo k přepnutí do jiného kontextu. Využitá paměť je uvolněna po zrušení kontextu, nebo po zavolání funkce pfree. Podstatná nevýhoda takového řešení spočívá v režii, která se projeví zejména při alokaci velkého množství malých bloků. Jelikož je PostgreSQL velice komplexní systém, kdy jednotlivé tuples vznikají, zanikají, či různě probublávají ve stromové struktuře, tak je takové řešení nezbytné.

V samotném kódu se často setkáme s funkcí nazvanou MemoryContextSwitchTo, která provede přepnutí do zvoleného kontextu. Například, chceme-li z funkce vrátit *tuplestore,* nemůžeme využít kontextu vytvořeného při inicializaci funkce. K tomuto účelu slouží kontext, který je přednastavený ve struktuře *fcinfo.* [3]

1.7 Způsob uložení dat na disku

Jednotlivé exekutory PostgreSQL určitým způsobem pracují s řádky databáze (tzv. tuples). Je třeba mít na paměti, že se vždy nemusí jednat o skutečné řádky, tak jak jsou uloženy v tabulkách. Důležitou roli hraje cachovaní a takzavné virtuální *tuples*, které vznikají postupným procházením prováděcího plánu. Existují dva základní způsoby uložení dat na disku:

1. Po řádcích (Row-Store) - Obrázek 1.1[4] přibližuje způsob uložení [2], které používá PostgreSQL. Databázový cluster obsahuje jednu nebo více pojmenovaných databází, které seskupují jedno či více schémat. Všechna data clusteru jsou uložena v datovém adresáři, na který odkazuje případná systémová proměnná PGDATA. Obsah zmíněného adresáře je tvořen mnoha podadresáři a důležitými soubory. Pokud rozdělíme slovo *database* na dvě části, dostaneme název adresáře base, který obsahuje *data* (*složky*) vztahující se k jednotlivým databázím. Název složky odpovídá *Oid* záznamu v systémovém katalogu 1.1.1 (pg_database). Každá tabulka a index jsou uloženy separátně ve svém vlastním souboru. Pro běžné relace platí, že název souboru odpovídá *relfilenode* atributu uloženém v systémovém katalogu (pg_class). Obsah souboru tvoří jednotlivé stránky obvykle o velikosti 8 KB. Na začátku každé stránky se nachází hlavička a položky, které odkazují na konkrétní *tuple*. Zbytek stránky je vyplněn sekvencí řádků (tuples), které jsou seřazené od konce stránky směrem jejímu začátku. Zcela na konci se nachází blok označovaný jako *Special*. Samotný *tuple* tvoří hlavička a položky odpovídající příslušným atributům.

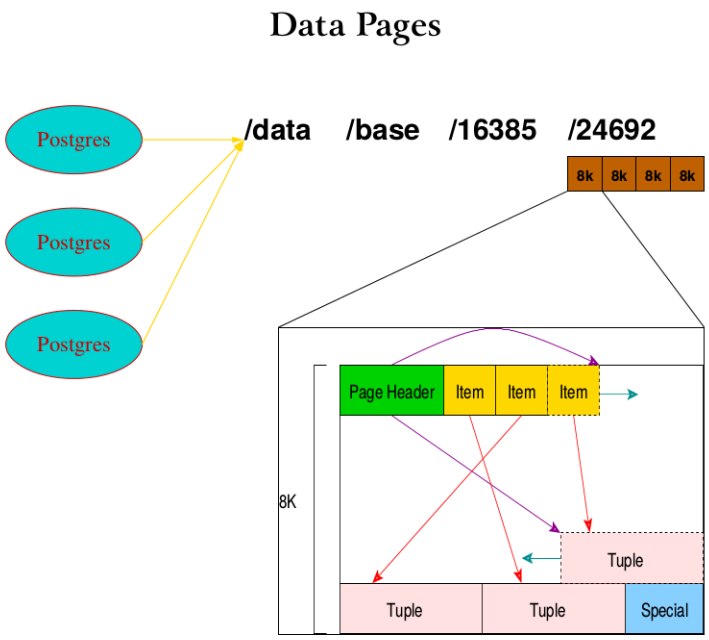
2. Po sloupcích (Column-Store) - Jedná se o způsob zápisu, kdy jsou za sebou nejprve uloženy první atributy všech tuples, následují druhé atributy všech tuples, atd. PostgreSQL interně ukládá data po řádcích, ale existuje rozšíření[5], které umožňuje zápis po sloupcích.

1.7.1 Výhody, nevýhody zápisu po sloupcích [6]

Pokud často dochází ke vkládání, upravování, mazání řádků, tak jednoznačně vítězí řádkový způsob uložení. Své místo si tedy nalezne v OLTP[2] systémech, kde často dochází k manipulaci s daty (bankovní převody, transakce všeho druhu, tvorba faktur, objednávek, atd).

Dochází-li k analytické činnosti pouze nad některými atributy, je zbytečné, aby se načítaly celé řádky. V každém případě je vždy zapotřebí načíst minimálně jednu stránku o velikosti 8 KB z disku, i když chceme přečíst pouze jeden řádek. Například, chtě-li bychom zjistit, kolik peněz jsme utržili za dnešní

[2]OLTP - On-Line Transaction Processing



Obrázek 1.1: Řádkový způsob uložení dat na disku, tak jak ho implementuje PostgreSQL

objednávky, použili bychom agregační funkci sum(cena objednávky). Požadujeme sice jen jednu hodnotu z odpovídajících řádků, ale systém i tak načte řádky celé, respektive stránky. V případě zápisu po sloupcích by se počet načítaných stránek rapidně snížil. Jelikož by se načetly pouze ty, které obsahují atribut *cena objednávky*. Sloupcový druh zápisu se hodí pro OLAP[3] systémy, které jsou optimalizovány pro uložení informací a následnou jejich analýzu. Své místo si nalezne v oblasti Business intelligence či v datových skladech. Protože se tímto směrem ubírá také tato práce, bude zmíněný plugin2, který umožňuje zápis po sloupcích v PostgreSQL, zahrnut do testování.

[3]OLAP - On-Line Analytical Processing

Kapitola 2

Analýza exekutorů

Podkapitola 2.1 se snaží přiblížit obecný průběh zpracování dotazu. Navazující podkapitola 2.2 detailněji popisuje funkci interpretu prováděcích plánů, který postupně volá obslužné funkce pro jednotlivé uzly. Druhý interpret AST výrazů popsaný v podkapitole 2.3 slouží k vyhodnocování netriviálních hodnot sloupců, které jsou výsledkem operací provedenými nad jedním či více atributy. Na základě informací z této kapitoly bychom měli být schopni rozpoznat případná úskalí integrace a zohlednit je při výběru JIT kompilátoru.

2.1 Posloupnost zpracování dotazu

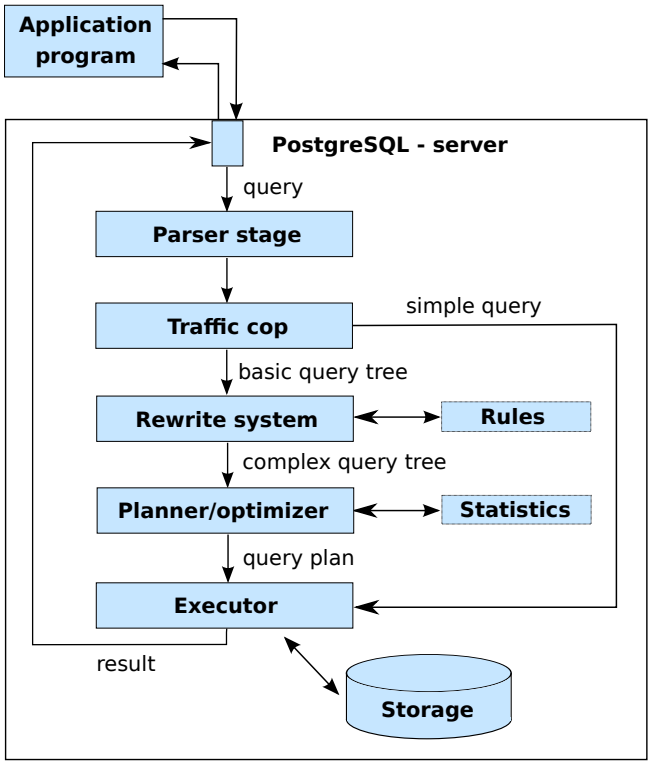
Mezi zasláním dotazu na server a přijmutím výsledku proběhne sekvence přesně definovaných operací[6, 2] Průběh zpracování přibližuje obrázek 2.1.

1. Na začátku klientská aplikace naváže spojení se serverem. Následně dojde k odeslání dotazu a čeká se na odpověď serveru.

2. *Parser stage*: Server dotaz přijme a provede syntaktickou analýzu. Parser textovou reprezentaci dotazu převede na sekvenci tokenů, aby rozpoznal jeho gramatickou strukturu. Na konci zkontroluje syntaxi a vytvoří výsledný strom.

3. *Traffic cop* nedělá nic jiného než, že identifikuje jednoduché dotazy typu ROLLBACK, atd. V takovém případě není totiž nutné provádět další kroky a dotaz může být přímo přesměrován na *Exekutor*.

4. *Rewrite system* na strom dotazu aplikuje tzv. *Rules,* která jsou uložena v systémovém katalogu. Tím dojde k transformaci původního stromu. Například pokud je proveden dotaz nad pohledem, tak je potřeba vzít v potaz interní SQL dotazy, ze kterých je pohled složen. V této části dojde také k identifikaci tabulek, funkcí, operátorů„., kterých se dotaz týká.



Obrázek 2.1: Průběh zpracování dotazu

5. Existuje mnoho způsobů, kterými lze dotaz zpracovat. *Planner/optimizer* provede srovnání všech cest. Na základě výpočtů a různých statistik vybere prováděcí plán s nejnižší cenou.

6. *Exekutor* rekurzivně prochází strom prováděcího plánu vykonává příslušné operace v daném uzlu. (sekvenční sken, řazení, spojování tabulek,..). Na konci kořen stromu obsahuje výsledek dotazu, který je odeslán klientské aplikaci.

2.2 SQL exekutor

Jak již bylo zmíněno v 6, SQL exekutor postupně interpretuje prováděcí plán, který se svou strukturou podobá AST [4][7]. Funkce exekutoru a příprava dotazu bude přiblížena na jednoduchém příkladu ve výpisu:2.1

Výpis 2.1: Prováděcí plán jednoduchého dotazu

[4]AST - Abstraktní syntaktický strom je reprezentace abstraktní syntaktické struktury zdrojového kódu napsaného v programovacím jazyce. Jednotlivé uzly obsahují operátory a listy tvoří operandy.

```
SELECT price FROM items WHERE price > 600;

QUERY PLAN

Seq Scan on items  (cost=0.00..12.12 rows=57 width=4)
Filter: (price > 600::double precision) (2 rows)
```

2.2.1 Průběh zpracování dotazu

Informace o průběhu zpracování dotazu lze získat pomocí debuggeru. Nejprve je potřeba identifikovat ID procesu, který má na starosti aktuální session. K tomuto účelu existuje v PostgreSQL funkce s názvem pg_backend_pid(). Následuje připojení debuggeru k danému procesu, nastavení požadovaných breakpointů a odeslání příslušného dotazu na server.

Umístěný breakpoint zastavil provádění dotazu těsně před samotným exekutorem. Ve výpisu2.2 jsou vidět metody, které se aktuálně nachází na stacku:

Výpis 2.2: Zavolané metody do doby než se program zastavil na beakpointu před spuštěním exekutoru.

```
Thread [1] 10295 [core: 1] (Suspended : Breakpoint)
standard_ExecutorRun() at execMain.c:320 0x5b6924
PortalRunSelect() at pquery.c:946 0x6b4bd8
PortalRun() at pquery.c:790 0x6b61b6
exec_simple_query() at postgres.c:1 103 0x6b39b0
PostgresMain() at postgres.c:4 023 0x6b39b0
BackendRun() at postmaster.c:4 141 0x464268
BackendStartup() at postmaster.c:3 826 0x464268
ServerLoop() at postmaster.c:1 594 0x464268
PostmasterMain() at postmaster.c:1 241 0x659a4e
main() at main.c:221 0x464ff4
```

1. Již z názvu prvních metod si lze celkem dobře představit, co se na daném místě odehrává. Klientská aplikace se nejdříve spojí s postmasterem, což je víceuživatelský databázový server. Postmaster provede nezbytné úkony a následně spustí separátní proces (postgres), který zpracuje vlastní dotaz.

2. Proces postgresu v průběhu zpracování zavolá metodu exec_simple_query(), která obdrží dotaz v textovém formátu. Uvnitř se mimo jiné volají metody určené pro parsování dotazu(pg_parse_query), pro analýzu a transformaci dotazu(pg_analyze_and_rewrite) a v neposlední řadě pro tvorbu prováděcích plánů (pg_plan_queries). Poslední zmíněná metoda zavolá pro každý dotaz funkci pg_plan_query a vrátí seznam struktur

nazvaných *PlannedStmt*. Struktura zaobaluje vlastní strom prováděcího plánu(*planTree*), identifikátor dotazu, typ (select | insert | update | delete), atd.

3. Proces postgresu následně vytvoří *portál* (struct PortalData), kterému jsou předány potřebné informace (text dotazu, seznam struktur *PlannedStmt*, atd). Dále například obsahuje atributy atStart, atEnd, portalPos, které indikují aktuální pozici kurzoru. Nakonec se z *portálu* vytvoří a inicializuje struktura QueryDesc, která obsahuje pouze jeden prováděcí plán a všechny potřebné informace pro exekutor. Jelikož je dotaz typu select (portal->strategy = portal_one_select), spustí se metoda PortalRunSelect. Nyní již nic nebrání spuštění metody ExecutorRun, které je předána zmíněná struktura QueryDesc. Následně jsou z QueryDesc extrahovány vybrané atributy, které jsou předány metodě ExecutePlan.

4. Uvnit metody ExecutePlan se nachází nekonečná smyčka, v jejímž těle se volá metoda ExecProcNode(planstate). Jednotlivé volání metody spustí tělo exekutoru. Návratová hodnota odpovídá nově získanému *tuple* příslušného dotazu. Smyčka se ukončí v případě, že získáme nulový *tuple*, nebo jsme již načetli požadovaný počet *tuples* (například při použití kurzorů nechceme načítat najednou všechny záznamy).

2.2.2 Základ SQL exekutoru

ExecProcNode obsahuje rozsáhlý switch (viz. výpis 2.3), který na základě tagu uzlu vždy spustí metodu určenou pro jeho zpracování. Návratová hodnota odpovídá struktuře TupleTableSlot.

Výpis 2.3: Switch metody ExecProcNode, který tvoří základ SQL exekutoru.

```
switch (nodeTag(node))
{
case T_ResultState:
        result = ExecResult((ResultState *)node);
break;
....
case T_SeqScanState:
        result = ExecSeqScan((SeqScanState *)node);
break;
case T_IndexScanState:
        result = ExecIndexScan((IndexScanState *)node);
break;
....
case T_NestLoopState:
        result = ExecNestLoop((NestLoopState *)node);
break;
```

```
case T_MergeJoinState:
        result = ExecMergeJoin((MergeJoinState *)node);
break;
case T_HashJoinState:
        result = ExecHashJoin((HashJoinState *)node);
break;
....
case T_AggState:
        result = ExecAgg((AggState *) node);
break;
....
}
```

Některé metody opět volají ExecProcNode a tím dochází k rekurzivnímu procházení prováděcího plánu. Například ExecHashJoin nejprve potřebuje načíst menší relaci a tudíž provede několik volání metody ExecSeqScan (počet volání odpovídá cca počtu řádků). Dále použije sekvenční scan k načtení jednoho *outer tuple* z druhé relace. Za pomocí hashovací funkce ověří, zda lze najít korespondující záznam v již načtené relaci. V případě úspěchu vrátí tuple odpovídající danému spojení. Tím se opět dostaneme zpět do smyčky metody ExecutePlan4. Jelikož nejsou načteny všechny řádky, tak znovu dojde k zavolání metody ExecProcNode. Jako první se spustí metoda ExecHashJoin, protože kořenový uzel je otagován jako T_HashJoinState. Menší relace je již načtená, tudíž postačuje provést jeden sekvenční sken nad druhou relací a použít hashovací funkci. Opakováním stejného postupu dojde postupně k načtení všech požadovaných řádků.

Ukázkový dotaz2.1 obsahuje jen sekvenční sken, takže se několikrát zavolá metoda ExecSeqScan (podle počtu řádků).

TupleTableSlot - *Exekutor* ukládá *tuples do* tzv. "*tuple table*". Jedná se v podstatě o List nezávislých *TupleTableSlot*(ů). Existují čtyři základními typy *tuples*:

1. Fyzický *tuple* uložený v diskovém bufferu stránek.
2. Fyzický *tuple* uložený v paměti za použití metody palloc
3. "Minimální" fyzický *tuple* uložený v paměti za použití metody palloc.
4. "Virtuální" *tuple* obsahující pole prvků Datum / isNull.

První dva případy jsou si podobné v tom, že pracují s "materializovanými" *tuples*, ale správa je odlišná. "Minimální" *tuples* neobsahují systémové sloupce (mohou mít Oid), jsou spravovány podobně jako ve druhém případě a nikdy nejsou ukládány do bufferu. "Virtuální" *tuples* slouží k optimalizaci přenesených dat mezi vnitřními uzly prováděcího plánu. Jeho jednotlivé prvky

odkazují na různé hodnoty jiných *tuples* nacházejích se v nižších vrstvách prováděcího plánu.

TupleTableSlot (viz. výpis2.4) je zde uveden také proto, že jsou z něho načítány jednotlivé hodnoty atributů, které využije expression exekutor.

Výpis 2.4: Struktura TupleTableSlot

```
typedef struct TupleTableSlot {
NodeTag type;
....
HeapTuple tts_tuple;
TupleDesc tts_tupleDescriptor;
MemoryContext tts_mcxt;
Buffer tts_buffer;
int tts_nvalid;
Datum *tts_values;
bool *tts_isnull;
....
}
TupleTableSlot;
```

- tts_tuple - Jedná se o ukazatel na strukturu HeapTupleData, která v sobě zahrnuje ukazatel na samotná data, Oid tabulky a ukazatel na strukturu hlavičky HeapTupleHeader. Pokud je tuple virtuální, nabývá hodnoty NULL.
- tts_tupleDescriptor - Reprezentuje desktiptor daného *tuple*. Pro *tuples* načtené z disku platí, že obsahuje uživatelské informace získané ze systémových katalogů pg_attribute, pg_attrdef a pg_constraint.
- tts_nvalid - Udržuje informaci o počtu validních sloupců tts_value/isnull.
- tts_values - Pole obsahuje autoritativní data, pokud se jedná o "virtuální" *tuple*. V případě fyzických tuples disponuje hodnotami extrahovanými ze samotných tuples. Data se načítají podle potřeby (lazily).
- tts_isnull - Pole indikuje hodnoty null ve sloupcích.

2.3 Expression exekutor

Expression exekutor zpracovává výraz, který operuje nad jednotlivými sloupci vybraných tabulek. V této práci nás budou nejvíce zajímat různé matematické výrazy, booleovské formule, funkce a v neposlední řadě podmíněný výraz CASE.

Stejně jako v případě SQL exekutoru, také zde bude funkce exekutoru a příprava dotazu přiblížena na příkladu ve výpisu2.5:

Výpis 2.5: Prováděcí plán dotazu nad uvedenou tabulkou expression

```
Table "public.expression"
Column |  Type   | Modifiers
-------+---------+-----------
id     | integer | not null
a_real | real    |
b_real | real    |
a_int  | integer |
b_int  | integer |


SELECT (a_real - abs(b_real)) * (pow(a_int + b_int,2))
FROM expression;

QUERY PLAN

Seq Scan on expression
(cost=0.00..52.50 rows=1700 width=16) (1 row)
```

Pro úspěšnou integraci JIT kompilátoru je naprosto nezbytné znát proces tvorby stromu výrazu. PostgreSQL postupně volá všechny metody, které jsou zmíněné v podkapitole 2.2.1 s tím, že navíc musí zpracovat strom výrazu.

2.3.1 Hrubé parsování výrazu

První zásadní okamžik se odehrává po zavolání metody pg_parse_query, kde probíhá parsování dotazu včetně výrazu. Využívá se unixových nástrojů s názvem Bison a Flex. Lexer je definován souborem scan.l. Jeho úkolem je rozpoznat identifikátory, SQL klíčová slova, atd. Výstup tvoří tokeny, které jsou následně zpracovávány parserem definovaným souborem gram.y. Uvnitř souboru se nachází jednotlivá pravidla gramatiky a akce, které se mají nad daným vstupem provést. Na následujícím výpisu2.6 si lze prohlédnout několik pravidel zodpovědných za základní matematické operace.

Výpis 2.6: Ukázka gramatických pravidel základních matematických výrazů ze souboru gram.y

```
a_expr:
....
| a_expr '+' a_expr
{$$ = (Node*)makeSimpleA_Expr(AEXPR_OP, "+", $1, $3, @2);}
| a_expr '-' a_expr
{$$ = (Node*)makeSimpleA_Expr(AEXPR_OP, "-", $1, $3, @2);}
| a_expr '*' a_expr
```

```
{$$ = (Node*)makeSimpleA_Expr(AEXPR_OP, "*", $1, $3, @2);}
| a_expr '/' a_expr
{$$ = (Node*)makeSimpleA_Expr(AEXPR_OP, "/", $1, $3, @2);}
....
```

V základě se pracuje se dvěma typy výrazů:

- a_expr - používá se v případech, kdy nejsou požadována žádná speciální omezení.
- b_expr - fakticky se jedná o určitou podmnožinu a_expr. Používá se v případech, kdy může nastat konflikt mezi operátory či klíčovými slovy. Například ve výrazu "BETWEEN a_expr AND a_expr" by vznikl konflikt, protože AND koliduje se stejnojmennným booleovským operátorem. Proto je BETWEEN součástí b_expr, ze kterého jsou vyjmuty booleovské operátory.

Funkce makeSimpleA_Expr nedělá nic jiného než, že vytvoří nový "Node" typu A_Expr, uloží si informaci o typu (AEXPR_OP), levém podstromu (Node - A_Expr), pravém podstromu (Node - A_Expr), názvu operátoru a o pozici operátoru.

2.3.2 Transformace výrazu

Předchozí fáze parsování vytvořila hrubý výstup, který je potřeba dále zpracovat. Pro analýzu a transformaci expression výrazů slouží metoda transformExpr, která vrátí strom výrazu s plně stanovenou sémantikou. V jejím těle dojde k zavolání metody, která obsahuje hlavní switch pro identifikaci stávajících uzlů stromu. Samotná transformace je provedena následovně:

1. Uzel je identifikován jako T_A_Expr.
2. Vnitřní switch určí o jaký přesný typ ((A_Expr*) a ->kind)) se jedná. V našem případě jde o AEXPR_OP, tím pádem je zavolána metoda transformAExprOp. Existují také AEXPR_OP_ANY, AEXPR_OP_IN, atd.
3. V případě jednoduchého skalárního operátoru je nejprve rekurzivně zpracován levý podstrom, pravý podstrom a nakonec je vytvořen metodou make_op uzel typu Expr (konkrétně OpExpr). Samotná struktura Expr v sobě obsahuje pouze indikátor typu uzlu.
4. Metoda make_op provede potřebné kroky k inicializaci a naplnění struktury OpExpr2.7. Protože se jedná o jednu z nejdůležitějších struktur z pohledu integrace JIT kompilátoru, je zde uveden její popis.

Výpis 2.7: Struktura OpExpr

```
typedef struct OpExpr {
Expr xpr;          /* obsahuje NodeTag              */
Oid opno;          /* PG_OPERATOR OID operátoru */
Oid opfuncid;      /* PG_PROC OID funkce */
Oid opresulttype; /* PG_TYPE OID návratového typu */
bool opretset;     /* jedná se o množinu? */
Oid opcollid;      /* OID collation výsledku */
Oid inputcollid;  /* OID collation operátoru */
List *args;        /* list argumentů */
int location;      /* pozice operátoru */
} OpExpr;
```

Struktura obsahuje generickou supertřídu Expr společnou pro spustitelné expression uzly. Většina atributů je typu Oid, které odkazují na záznam v systémovém katalogu. Funkce provádějící příslušnou matematickou operaci je zavolána na základě opfuncid.

2.3.3 Inicializace uzlů prováděcího plánu

Další důležitá část týkající se zpracování dotazu se odehraje v místě, kde dochází k inicializaci zvoleného prováděcího plánu. Vrátíme-li se zpět k výpisu 2.1 nacházejícího se v popisu SQL exekutoru, zjistíme, že zmínka o inicializaci zcela chybí. Pojďme ji ukotvit na správné místo. Před spuštěním metody PortalRun() dojde k volání metody PortalStart provádějící potřebnou přípravu portálu před spuštěním. Mimo jiné je zde vytvořena již uvedená struktura QueryDesc, která je postupným voláním několika funkcí předána metodě InitPlan. Jejím úkolem je inicializovat plán, otevřít soubory, alokovat "storage" a spustit "rule manager". Téměř na konci tohoto řetězce se spustí metoda ExecInitNode jež inicializuje jednotlivé uzly prováděcího plánu. Tělo metody se podobá ExecProcNode s tím rozdílem, že objemný switch na základě typu uzlu volá metody ExecInit..., místo Exec... .Vstupem je uzel odpovídající struktuře Plan a výstupem uzel odpovídající "abstraktní" struktuře PlanState s potomky (SeqScanState,...).

Struktura PlanState hraje při vykonávání expression výrazů zásadní roli. Vybrané atributy jsou zobrazeny na následujícím výpisu:

Výpis 2.8: Struktura PlanState

```
typedef struct PlanState {
NodeTag type;
Plan *plan;              /* vstupní Plan node  */
EState *state;
....
List *targetlist;
```

```
List* qual;
struct PlanState *lefttree;  /* levý podstrom */
struct PlanState *righttree; /* pravý podstrom */
TupleTableSlot *ps_ResultTupleSlot;
ExprContext *ps_ExprContext;
ProjectionInfo *ps_ProjInfo;
} PlanState;
```

Z výše uvedeného seznamu atributů nás nejvíce zajímají tyto:

- List * targetlist - jedná se o list výrazů, které představují výsledek dotazu. Korespondují s výrazy umístěnými mezi příkazem SELECT a FROM.
- List * qual - jedná se o list výrazů, které se vyhodnocují například v klauzuli WHERE. Každý výraz musí být vyhodnocen jako true, aby celá klauzule byla pravdivá.
- ProjectionInfo *ps_ProjInfo - struktura obsahuje informace o projekci. Svým významem si zaslouží svoji vlastní podkapitolu2.3.4.
- ExprContext *ps_ExprContext - struktura uchovává "aktuální kontextové" informace nutné pro vyhodnocení expression výrazů. Například pokud výraz odkazuje na sloupec v aktuálním "inner tuple", potřebujeme vědět, co je daný tuple zač.

Jelikož se v ukázkovém dotazu volá metoda ExecInitSeqScan, je nyní na místě se podívat, jak se v jejím těle pracuje s výrazem.

Ze všeho nejdříve se v metodě vytvoří uzel odpovídající struktuře SeqScanState jehož prvním atributem je právě PlanState. V další fázi se postupně začne plnit či inicializovat několik atributů z výpisu 2.8. (plan, state, ps_ExprContext,..).

Nás však především zajímá přiřazení hodnoty atributu targetlist, která je získána zavoláním metody ExecInitExpr((*Expr)plan.targetlist, (PlanState *) scanState). Metoda ExecInitExpr, jak je uvedeno výše, přijímá na vstupu (*Expr)plan.targetlist a na výstup vrátí instanci struktury ExprState. Vstup vlastně odpovídá transformovanému stromu z podkapitoly 2.3.2. Expression exekutor pracuje nad jednotlivými instancemi, které vycházejí právě z ExprState struktury. Struktura obsahuje následující atributy:

- NodeTag type - definuje, o jaký typ uzlu se jedná.
- Expr *expr - obsahuje asociaovaný expression uzel (OpExpr, Var, atd).
- ExprStateEvalFunc evalfunc - jedná se o ukazatel na funkci, která má na starosti provedení daného uzlu v expression výrazu. Zatímco SQL exekutor obsahuje zmíněný velký switch, naopak expression exekutor využívá metodu založenou na volání funkcí.

Tělo metody ExecInitExpr je tvořeno velkých switchem, který větví program podle typu vstupního uzlu. (T_Var, T_Const, T_OpExpr, T_List,...). Počítáme-li opět s jednoduchou matematickou operací, program nás zavede do větve, která zpracovává Expr typu T_OpExpr. Uvnitř se odehrají tři důležité operace:

- Ze všeho nejdříve dojde k vytvoření nového uzlu odpovídající struktuře FuncExprState, která jako první atribut obsahuje strukturu ExprState. Dále zapouzdřuje atribut FuncCallInfoData fcinfo_data, který slouží k předání vstupních dat do funkce vykonávající samotný výpočet. Několik vět bylo k této struktuře napsáno v kapitole 1.4

- V dalším kroku se nastaví výše zmíněný ukazatel na funkci evalfunc na hodnotu (ExprStateEvalFunc) ExecEvalOper. Jedná se totiž o funkci, která v expression exekutoru zpracovává tento typ uzlu.

- V neposlední řadě se inicializuje atribut args, který uchovává všechny operandy (podvýrazy). Samotná inicializace se provede rekurzivním voláním funkce ExecInitExpr((Expr *)opexpr->args, parent)) na jednotlivé argumenty. Zajímavý je fakt, že se funkci předá celý seznam argumentů. Ta totiž obsahuje větev, která zpracovává samotný seznam typu (T_List). Teprve uvnitř se nachází cyklus, ve kterém se rekurzivně volá ExecInitExpr na každý argument zvlášť.

Rekurzivním voláním metody ExecInitExpr se postupně inicializují všechny uzly stromu expression výrazu. Ať už se jedná o volání funkcí, matematické operátory, booleovské formule či podmíněný výraz CASE.

2.3.4 ProjectionInfo node

Ještě než se blíže podíváme na ProjectionInfo node, je dobré se zmínit o atributu struktury PlanState jménem ps_ResultTupleSlot, který nebyl blíže specifikován. Již jeho název napovídá, že se do inicializovaného slotu bude ukládat výsledek. Pokud se budeme držet předchozího příkladu, jeho inicializace se provede v metodě ExecInitSeqScan.

Jeden z posledních atributů struktury PlanState, který nebyl zmíněn, je právě *ps_ProjInfo. Jeho hodnota se získá zavoláním metody ExecBuildProjectInfo, které jsou předány následující hodnoty:

- planstate->targetlist - popsáno níže
- planstate->ps_ExprContext - uložen do projectInfo->pi_exprContext
- planstate->ps_ResultTupleSlot - uložen do projectInfo->slot
- TupleDesc inputDesc

V těle metody ExecBuildProjectInfo a dále v expression exekutoru hraje jednu z hlavních rolí struktura Var, která je uvedena na následujícím výpisu2.9:

Výpis 2.9: Struktura Var, která představuje konkrétní sloupec v tabulce

```
typedef struct Var {
Expr xpr;
Index varno;
AttrNumber varattno;
Oid vartype;
....
int location;
} Var;
```

Struktura Var je ”potomek” struktury Expr a reprezentuje konkrétní proměnnou (sloupec v tabulce). Význam vybraných atributů:

- Index varno - index relace v takzvané ”Range table”, která uchovává list relací použitých v dotazu, a do které zmíněný Var patří. Atribut však může pouze říkat, zda se jedná o INNER_VAR, OUTTER_VAR či INDEX_VAR.
- AttrNumber varattno - určuje číslo sloupce. V případě použití * v dotazu je nulový.
- Oid vartype - OID typu daného sloupce.
- int location - představuje lokaci původního tokenu v procesu parsování.

Uvnitř metody ExecBuildProjectInfo se nachází cyklus, který iteruje přes všechny hodnoty atributu targetlist. Kořen stromu každého prvku seznamu je přetypován na GenericExprState strukturu. Ta se používá v případech, kdy není potřeba ukládat žádný run-time stav, ale obsahuje jeden uzel typu Expr. Právě takový uzel je z ní extrahován a za použití metody IsA((Var*)variable, Var) je zjištěno, zda se jedná skutečně o Var uzel. Pokud se jedná o Var, tak dojde k inkrementaci proměnné numSimpleVars a provede se pro nás méně podstatná inicializace uvnitř struktury ProjectionInfo. Z pohledu expression exekutoru je zajímavější situace, kdy se nejedná o jednoduchý Var. V takovém případě je kořen stromu přidán na konec seznamu s názvem exprlist. Po zpracování poslední složky seznamu jsou do ProjectionInfo uloženy následující hodnoty:

- projInfo->pi_targetList = exprlist (proměnná zmíněna v odstavci výše)
- projInfo->pi_numSimpleVars = numSimpleVars (proměnná zmíněna v odstavci výše)

Z pohledu JIT kompilátoru se metoda ExecBuildProjectInfo jeví jako nejlepší pro zařazení kódu provádějící samotnou kompilaci složitějšího výrazu a to hned z několika důvodů:

1. Máme jednoznačně identifikován složitější výraz.

2. Bez větší námahy jsme schopni do uzlu stromu uložit zkompilovanou funkci zodpovědnou za vykonání celého stromu výrazu.

3. ExecBuildProjectInfo je volána také z metody ExecInitAgg, která je zodpovědná za inicializaci uzlů týkajícího se agregací.

2.3.5 Spuštění expression exekutoru

V předešlých podkapitolách jsme se detailněji seznámili s přípravou expression výrazu a s vybranými strukturami. Proto, abychom se podívali na místo, kde dochází ke spuštění expression exekutoru, je nutné opustit ExecInitSeqScan a zaměřit se na jejího sourozence s názvem ExecSeqScan v SQL exekutoru. Uvnitř se volá následující významná metoda:

- TupleTableSlot* ExecProject(ProjectionInfo *projInfo, ExprDoneCond *isDone) - Uvnitř metody jsou postupně plněna dvě pole:

- projectInfo->slot->tts_values

- projectInfo->slot->tts_isnull

Výše uvedená pole byla již zmíněna v kapitole 2.2.2v rámci popisu TupleTableSlotu. Odpovídají projekci nad relacemi v dotazu(hodnotám atributů a indikátoru, zda jsou null). Kód metody nejdříve načítá jednoduché hodnoty atributů, které lze určit přímo z relací bez dalšího zpracování. Nás však především zajímá uložení hodnot, které jsou tvořeny složitějšími výrazy. Z minulé podkapitoly 2.3.4 víme, že se nacházejí v listu projInfo->pi_targetList. Postupné volání expression exekutoru nad každým prvkem listu má na starosti stejnojmenná metoda ExecTargetList, které je předán samotný list, dále projectInfo->pi_exprContext pro přístup k jednotlivým tuples, ukazatelé na dvě zmíněná pole a jakýsi indikátor isDone.

Kořen stromu (struktura ExprState) výrazu s ostatními zmíněnými hodnotami týkajících se však jednoho konkrétního sloupce je předán makru s názvem ExecEvalExpr. Jeho úkolem je zavolat a předat parametry funkci uložené ve struktuře ExprState, která provede zpracování daného uzlu. Jelikož tyto funkce rekurzivně volají opět makro ExecEvalExpr, dojde ke zpracování celého stromu. Návratová hodnota odpovídá datovému typu Datum popsaného v kapitole 1.3. Seznam vybraných funkcí, jejichž funkcionalita bude zajisté nahrazena JIT kompilátorem:

- Datum ExecEvalFunc - úkolem metody je zavolat příslušnou funkci ze systémového katalogu.
- Datum ExecEvalOper - metoda má na starosti vykonání základních aritmetických operací.
- ExecEvalAnd - výstupem metody je výsledek booleovského operátoru AND.
- ExecEvalOr - výstupem metody je výsledek booleovského operátoru OR.
- ExecEvalNot - výstupem metody je negace hodnoty typu boolean.
- ExecEvalCase - metoda zpracovává podmíněný výraz CASE.
- ExecEvalScalarVar - hlavním úkolem metody je načíst konkrétní hodnotu atributu z řádku (tuple).
- ExecEvalConst - metoda vrátí konstantu použitou v dotazu.
- ExecEvalAggref - uvnitř se spouští kód, který vrátí předpočítanou hodnotu agregační funkce z daného expression kontextu.

2.3.6 Strom odpovídající ukázkovému dotazu

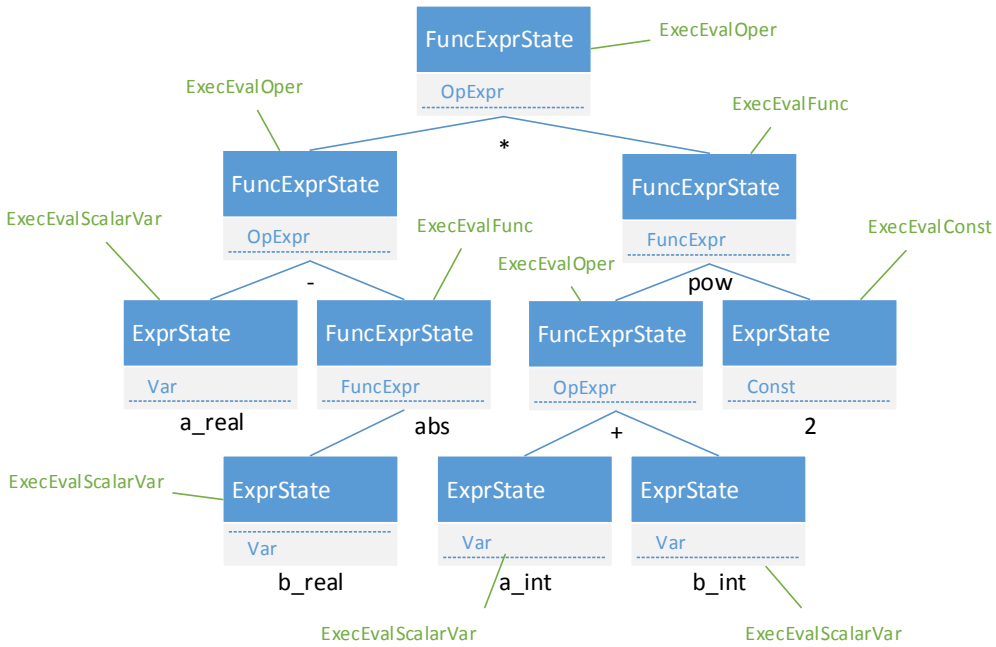
Strom expression výrazu na obrázku 2.2 výše nepotřebuje květnatý komentář, pokud zůstaneme na abstraktní úrovni. Postupně se načítají hodnoty z databáze (eventuálně konstanty), jsou na ně aplikovány požadované matematické funkce a aritmetické operátory. Jak již bylo zmíněno v předchozí podkapitole 2.3.5, návratový typ z interních funkcí zpracovávajících konkrétní uzel, odpovídá typu Datum, který je vrácen uzlu, který danou funkci zavolal.

Zajímavý se jeví fakt, že funkce s názvem ExecEvalOper je volána pro konkrétní uzel pouze jednou. V jejích útrobách se totiž změní hodnota atributu xprstate.evalfunc. Konkrétně se ukazatel na funkci může změnit na dvě odlišné hodnoty:

- ExecMakeFunctionResult - metoda je zvolena v případě, pokud volaná funkce či určitá část jejího expression podvýrazu může vrátit množinu hodnot.
- ExecMakeFunctionResultNoSets - ukazatel se použije v případě, že volaná funkce vrací spolu s jejími podvýrazy pouze jednu skalární hodnotu.

Pro zjednodušení integrace JIT kompilátoru nebude brána v potaz možnost navrácení množiny hodnot. Tímto rozhodnutím se zjednoduší plánovaná integrace. Kód bude však navržen tak, aby mohlo dojít ke snadnému rozšíření.

Mimo jiné se v ExecEvalOper inicializuje ze vstupních parametrů struktura FunctionCallInfoData 1.4 sloužící k zavolání funkce provádějící příslušnou



Obrázek 2.2: Strom expression výrazu odpovídající ukázkovému dotazu

matematickou operaci. S ExecEvalOper je velice úzce spjata ExecEvalFunc, protože se od ni téměř neliší. Jediný důvod, proč se používají dvě odlišné funkce je ten, že uzel obstarávající volání obecné (uživatelské) funkce uvnitř obsahuje strukturu FuncExpr a ne OpExpr.

Obě výše uvedené funkce zpracovávající uzel volají ExecMakeFunctionResultNoSets. V těle se rekurzivním voláním ExecEvalExpr získají výsledky z podvýrazů a vloží se do fcinfo->arg[i] jako argumenty. Následně se ověří jejich nulovost a za pomocí makra jménem FunctionCallInvoke(fcinfo) dojde k zavolání funkce obstarávající požadovanou funkcionalitu.

Zbytek funkcí není v tuto chvíli třeba detailněji rozebírat a určitě se na ně ještě podíváme ve fázi návrhu a samotné realizace integrace.

Kapitola 3

Analýza JIT kompilátorů

Po úvodním seznámení s termínem Just-in-time kompilace v podkapitole 3.1 navazující podkapitola 3.2 ozřejmuje možnosti 64 bitové architektury procesorů, na které bude postavena a otestována realizovaná integrace. Následně byla zmíněna alternativa ve formě vektorizace. Protože úspěch výsledného řešení stojí a padá na vybraném Just-in-time kompilátoru, značná část kapitoly se věnuje jejich analýze. Ze začátku se zaobírá vestavěnými řešeními v rámci virtuálních strojů. Postupně se však dostane k popisu zcela nezávislých JIT kompilátorů. U každého z nich jedna podkapitola přibližuje a nastiňuje možné provedení integrace.

3.1 Just-in-time kompilace

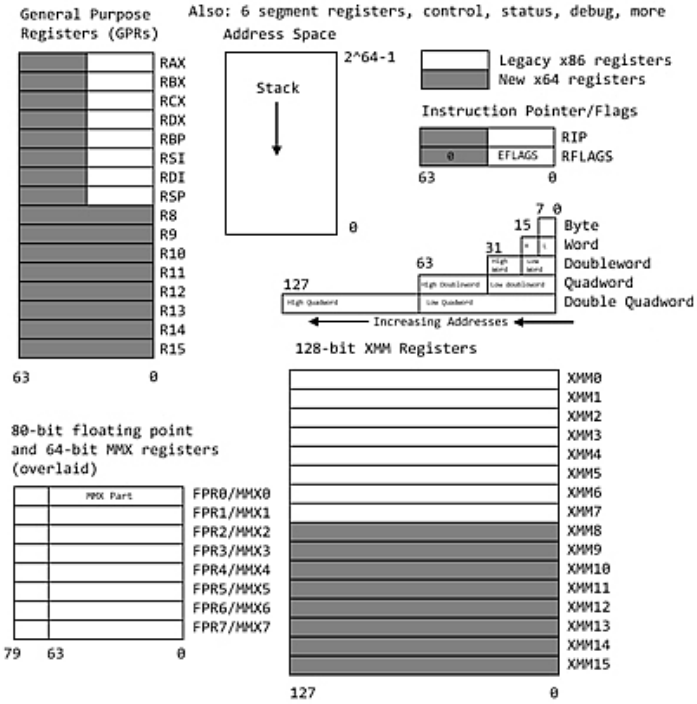
Just-in-time kompilátory [8] překládají takzvaný bytecode v průběhu běhu programu do instrukční sady cílového počítače. Některé JIT kompilátory pracují s pojmem bytecode, jiné operují s takzvaným IR "intermediate representation", ale prakticky se jedná o jiné pojmenování toho samého. Tento mezikód vzniká překladem zdrojového kódu vyššího programovacího jazyku, kterým může být například oblíbená Java. Obecně nedochází ke kompilaci celého bytecodu, ale přeloží se jen jeho vybraná část, která je určitým způsobem významná, či může být častěji volána (tzv. hot-path). Hlavním cílem je zkrátit čas běhu programu, což se například jak uvidíme u Javy, velmi daří. Ostatní části bytecodu jsou interpretovány. S názornými příklady interpretů jsme se již setkali v kapitole 2 v podobě SQL, expression exekutorů. Zpravidla se jedná o veliký switch, který postupně prochází jednotlivé instrukce a na základě toho dochází k volání obslužných funkcí. V případě PostgreSQL se interpretují uzly stromu prováděcího plánu. S tím je však úzce spojená určitá režie. Například pro sečtení dvou čísel za pomocí interpretu Javy, je nejprve potřeba daná čísla umístit na zásobník operandů, který je součástí "framu" funkce . Poté dojde k samotné operaci sečtení. Výsledek je umístěn na vrchol zásobníku. Jeho hodnota může být následně uložena zpět do lokální proměnné

ve "framu". S tím kontrastuje pouhé sečtení dvou čísel umístěných v registrech procesoru.

JIT kompilátory mají za sebou již několik desítek let vývoje a rozhodně se nejedná o snadnou záležitost. Při JIT kompilaci musí být řešen optimální poměr mezi časem překladu a provedenou optimalizací kódu. Obvykle je kód nejprve interpretován, v průběhu jsou uchovávány, zpracovávány a vyhodnocovány různé statistické údaje a na jejich základě jé kód optimálně přeložen.

3.2 64 bitová Architektura

Jelikož se počítá s tím, že JIT kompilátor bude integrován do PostgreSQL běžícím na 64 bitové architektuře [9][10], podíváme se, co užitečného nám tato architektura nabízí.



Obrázek 3.1: 64 bitová architektura

Obrázek 3.1 převzatý z [9] zobrazuje šestnáct 64 bitových obecných registrů. Prvních osm je označeno jiným způsobem z historických důvodů. V případě, že počáteční R u prvních osmi registrů přepíšeme na E, budeme přistupovat k prvním 32 bitům. Podobně pokud z RAX, RBX,..,RBP odstraníme počáteční R, budeme přistupovat k prvním 16 bitům (AX, BX,..). Stále nejsme u konce, protože existuje možnost načíst / uložit zvlášť 8 dolních bitů(AL, BL, ..) a 8 horních bitů (AH, BH,..). Podobným způsobem lze pracovat s registry

R8 až R15: R8(qword), R8D (lower dword), R8W (lowest word), R8L (lowest byte), neexistuje žádné R8H. Většina obecných registrů se používá jako úložiště pro operandy, výsledky a ukazatele. Mohou však sloužit ke speciálním účelům. Určitá obezřetnost se předpokládá u registru ESP, jelikož uchovává ukazatel na vrchol paměťového zásobníku. ECX například slouží jako čítač ve smyčce. Mnoho instrukcí předpokládá ve vybraných registrech své operandy, jiné do nich naopak ukládají výsledky operace.

Floating point unit (FPU) používá osm 80 bitových registrů, kde jsou hodnoty uloženy ve speciálním formátu [5]. Pokud jsou načítána data v jiném formátu, dochází k automatické konverzi. FPU instrukce zacházejí se zmíněnými registry jako se zásobníkem. Speciální položka (TOP) ve stavovém slovu určuje registr, který reprezentuje vrchol zásobníku. Například instrukce load sníží hodnotu TOP o jedničku a uloží zde požadovanou hodnotu.

3.2.1 SIMD[6] architektura[10]

je založena na jednotlivých instrukcích, které umožňují paralelní zpracování většího množství dat. Zatímco instrukce spjaté s FPU pomalu zastarávají, technologie týkající se SIMD jsou více než aktuální. Před mnoha lety společnost Intel uvedla na trh technologii MMX, která do architektury IA-32 zavedla právě zmiňované SIMD. Jelikož bylo potřeba zavést určitou hardwarovou podporu, vznikly staronové 64 bitové registry s příznačným jménem MMX0 - MMX7. Slovo staronové zde není náhodou. Ve skutečnosti jsou registry namapované na již stávající, a které využívá výše zmíněná jednotka FPU. Technologie MMX uvedla následující 64 bitové datové typy:

- 64 bitové "packed byte integers" - osm "packed bytes"
- 64 bitové "packed word integres" - čtyři "packed words"
- 64 bitové "packed doubleword integres" - dva "packed doublewords"

Pro demonstraci MMX technologie může být zmíněna instrukce paddsw, která paralelně sečte čtyři "signed word integers" uloženými v jednom MMX registru se čtyřmi "signed word integers" uloženými v jiném MMX registru.

Zatímco MMX pracuje nad celočíselnými typy, novější technologie s názvem SSE[7] přidává podporu typů s plovoucí řádovou čárkou (pouze single precision). Spolu s ní přichází na scénu osm 128 bitových XMM registrů. V případě 64 bitové architektury je těchto registrů šestnáct. Později Intel vydává rozšíření SS2, které navazuje na MMX a SSE. Přidává především podporu "packed double precision" typů s plovoucí řádovou čárkou a 128 bitové "packed integers". Protože můžeme využít 128 bitové registry, tak se vlastně

[5] double-extended precision floating-point format
[6] SIMD - Single Instruction Multiple Data
[7] Streaming SIMD Extensions

zdvojnásobil počet typů, které je možné umístit vedle sebe do registru. (šestnáct "packed bytes, osm packed words",..). Navíc lze najednou zpracovat dva "quadword integers". Další navazující SSE3, SSE4 již nepřidávají podporu dalších typů a nebudou se zde dále více rozebírat.

3.2.2 Volací konvence [11]

popisuje rozhraní mezi volajícím a volaným kódem (funkcí). Jelikož bude nutné pracovat s interními funkcemi PostgreSQL, podíváme se, jak se na nízké úrovni volají jednotlivé funkce. Zmíněné rozhraní popisuje:

- Pořadí předávaných parametrů (skalárních, komplexních).
- Jakým způsobem jsou parametry předány (za pomocí zásobníku, registrů, či kombinací obojího).
- Jaké hodnoty registrů musí být zachovány.
- Rozdělení zodpovědností mezi volající a volanou funkci (příprava zásobníku, obnovení hodnot registrů, atd).

Na celém světě existuje mnoho kompilátorů a každý může volací konvence definovat trochu jinak. Někdy je obtížné propojit dva kódy, pokud používají dva odlišné kompilátory. My se podíváme na jednu volací konvenci s názvem "cdecl", která vychází z jazyka C a používá ji mnoho kompilátorů pro architekturu x86. Patří pod takzvané "Caller clean-up" volací konvence, kdy volající funkce má na starosti vymazání předávaných parametrů ze zásobníku. V "cdecl" se parametry předávají pomocí zásobníku. Hodnoty typu integer a ukazatelé jsou vráceny v registru EAX. Hodnoty typu float se vracejí pomocí registru FPR0 (XMM0). V kontextu programovacího jazyka C (Linux) jsou parametry ukládány na zásobník v opačném pořadí. (Microsoft je ukládá zprava doleva).

Výpis 3.1 převzatý ze zdroje [11] se snaží věc více ozřejmit.

Výpis 3.1: Výpis přeložené funkce v jazyku symbolických adres

```
caller:
push ebp      // uložení hodnoty v reg. ebp na zásobník
mov ebp, esp  // vrchol zásobníku do reg. ebp
push 3        // vložení 3. argumentu na zásobník (int)
push 2        // vložení 2. argumentu na zásobník (int)
push 1        // vložení 1. argumentu na zásobník (int)
call callee   // zavolání funkce callee (úprava EIP reg.)
add esp, 12   // nepotřebujeme již argumenty =>
              // posuneme esp o 3*4 byty (velikost int)
add eax, 5    // přičtení pětky k vrácené hodnotě
pop ebp       // načtení aktuální stack hodnoty do ebp
```

```
ret                 // návrat z metody (úprava EIP) registru
```

3.2.3 Stack 3.1

je nepřerušené pole paměťových míst. Nachází se v segmentu definovaným registrem SS. Pokud je použit takzvaný "flat model", může se nacházet kdekoliv v paměti. Položky lze na zásobník umístit instrukcí push. Pro jejich odstranění slouží instrukce pop. Při použití instrukce push procesor sníží hodnotu v ESP registru (stack pointer) a zapíše požadovanou hodnotu na zásobník. Opačný postup je uplatněn v případě použití instrukce pop. Program nebo operační systém může vytvořit mnoho zásobníků. Jejich maximální počet je limitován maximálním počtem segmentů a velikostí operační paměti. Zásobník se typicky skládá z jednotlivý "frames". Každý může obsahovat lokální proměnné, parametry pro volanou funkci a informace spojující jednotlivé procedury. Takzvaný "stack-frame base pointer" uložený v registru EBP určuje paměťové místo na zásobníku pro volanou proceduru. Ta typicky zkopíruje obsah ESP registru do EBP registru před vložením lokálních proměnných na zásobník. Díky tomu můžeme snadněji přistupovat k jednotlivým hodnotám (relativně od začátku procedury).

Před skokem na první instrukci volané procedury instrukce CALL vloží aktuální obsah EIP registru (instruction pointer) na zásobník. Instrukce RET načte hodnotu ze zásobníku a uloží ji zpět do EIP registru.

Celou dobu se vlastně bavíme o 32 bitových registrech, v případě 64 architektury se nic zásadního nemění. Více registrů přináší více místa, a proto se využívají také pro předávání parametrů funkcím. Většinou jsou pro tento účel vyhrazeny alespoň tři obecné registry a tři XMM registry.

Nyní je již snadnější pochopit kód uvedený ve výpisu 3.1.

3.3 Vektorizace jako alternativa

Rychlost provádění SQL dotazů může být zvýšena technikou využívající takzvanou vektorizaci [12]. Řádným příkladem je VectorWise[12] DBMS, který zpracovává data po větších blocích najednou, což snižuje vliv režie samotné interpretace. Využívá se především schopností moderních CPU provádět operace nad vektory a to za pomocí technologie SIMD zmíněné v předešlé podkapitole 3.2. Oba přístupy, tedy vektorizace a JIT kompilace rapidním způsobem zrychlují provádění SQL dotazů.

VectorWise DBMS je databáze zaměřena na analytické výpočty , business intelligence, podporu rozhodování, atd. Předpokládá se, že dotazy budou provádět výpočty nad tabulkami s velkým počtem atributů. Ve skutečnosti budou operovat spíše nad určitou podmnožinou atributů, ale přes velké množství řádků.

Konkrétní operátory v obecném DBMS obvykle následují model poprvé uvedený ve Volcano. Operátory tvoří strom a implementují API s metodami open(), close() a next(). Metody open() a close() jsou použity pro inicializaci a finalizaci. Zatímco od metody next() očekáváme určitý výstup. Ve většině databázových systémem jedno volání next() způsobí načtení jednoho "tuple" (tzv. tuple-at-time). Rozdíl mezi VectorWise a ostatními databázemi implementující Volcano model, je ten, že metoda next() zmíněného API iterátoru vrátí vektor výsledků. Jednotlivým voláním next() získáme najednou dalších 100 až 10000 řádků, což snižuje režii interpretace.

VectorWise ukládá záznamy po sloupcích. Výhody takového uspořádání pro analytické výpočty byly již zmíněny v kapitole 1.7.

Z pohledu této práce nás zajímá princip výpočtu expression výrazů. VectorWise nazývá stěžejní funkce jako "primitives". V jejich těle se v úzké smyčce postupně zpracovává pole hodnot. Obvykle jsou zaměřeny na jednu konkrétní operaci, jako je vyhodnocení aritmetického výrazu, booleovské podmínky, načítání z paměti, atd. Výpočty nad jednotlivými poli se provádí pomocí instrukcí technologie SIMD.

Výsledky jsou úchvatné. Ve výkonnostních testech se dosahuje až padesátinásobného zrychlení.

3.4 Optimalizační techniky

Kompilátory obecně implementují nepřeberné množství optimalizačních technik [13]. Na následujících několika řádcích si uvedeme techniky, které jsou určitým způsobem svázány s expression výrazy.

Constant-Expression Evaluation optimalizace zajistí, že se vyhodnotí výrazy, jejichž operandy zůstávají za běhu programu konstantní. Pokud jsou všechny operandy konstantní, expression výraz se zredukuje pouze na skalární hodnotu. Optimalizace je aplikovatelná také na booleovské výrazy a provádí se v čase kompilace.

Algebraic SimplifIcations and Reassociation paří mezi základní optimalizace expression výrazů. "Algebraic SimplifIcations" využívá ke zjednodušení výrazů algebraické vlastnosti operátorů a jejich určité kombinace. "Reassociation" využívá asociativity, komutativity a distributivity k rozdělení výrazu na části, které jsou konstantní (loop-invariant). Jde o hodnoty, které zůstávají stejné v průběhu celého cyklu. Ukázka případných optimalizací:

- 1 + 0 = 0 + 1 = 1 - 0 = 1; 1 * 1 = 1 * 1 = 1 / 1 = 1
- -(-i) = i; i + (-j) = i - j
- b | true = true | b = true; b | false = false | b = b

Určitá zjednodušení pouze změní operátory na jiné, které jsou méně náročné na výpočet:

- i^2 = i * i
- 2 * i = i + i

Násobíme-li číslo malými konstantami, vyplatí se výraz převést na kombinaci bitového posunu a sčítání. Například výraz i + 5 lze přepsat ve dvou krocích:

1. t = i shl 2
2. t = t + i

Příklad uplatnění asociativity a distributivity může vypadat následovně:

- (i - j) + (i - j) + (i - j) + (i - j) = 4 * i - 4 * j

Value numbering je jedna z mnoha metod, jak rozpoznat, že dva rozdílné výpočty jsou ekvivalentní. Jeden z nich je následně eliminován. Technika přiřadí symbolickou hodnotu výpočtu, ale bez interpretace vlastního operátoru. Musí být však splněno, že výpočty se stejnou symbolickou hodnotou musí vést ke stejnému výsledku. Příklad:

1. j = i + 1
2. k = i
3. l = k + 1

Copy propagation je transformace, která po přiřazení hodnot x = y nahradí všechny výskyty x za y, dokud se nezmění hodnota jedné z nich.

Constant propagation je transformace, která po přiřazení hodnot x = c (konstanta) nahradí všechny výskyty x za c, dokud se nezmění hodnota x.

Common-Subexpression Elimination zajistí, že stejné podvýrazy nebudou vypočítávány dvakrát. Podmínkou je, že se jednotlivé operandy za dobu výpočtu nezmění. V databázovém světě se daná optimalizace uplatní vždy, protože hodnoty sloupců (operandů) se v průběhu dotazu nemění. Její přínos může být však někdy diskutabilní. Pokud by hodnota měla být v registru uložena na dlouhou dobu, tak by jeho blokace mohla zapříčinit zpomalení jiného výpočtu, jelikož by nemohl být použit. Například by muselo být využito daleko časově náročnějšího přístupu do paměti.

Dead-Code Eliminaion zajišťuje, že se ve zkompilovaném kódu nebude nacházet takzvaná mrtvá část, která nemá vliv na výsledek procedury. Je celkem jedno, zda se jedná o hodnoty či samotné instrukce.

Register Allocation řeší problém, jak co nejefektivněji provádět alokaci registrů. Jedná se o jednu z nejdůležitějších optimalizací. Zatímco registry jsou rychlé, k dispozici jich máme malý počet. Paměti se nám dostává podstatně více, ale přístup k ní je několikanásobně pomalejší. Určitou roli zde také hrají cache paměti procesorů. Kompilátory například používají techniku založenou na obarvování grafů, která významným způsobem neovlivňuje rychlost kompilace.

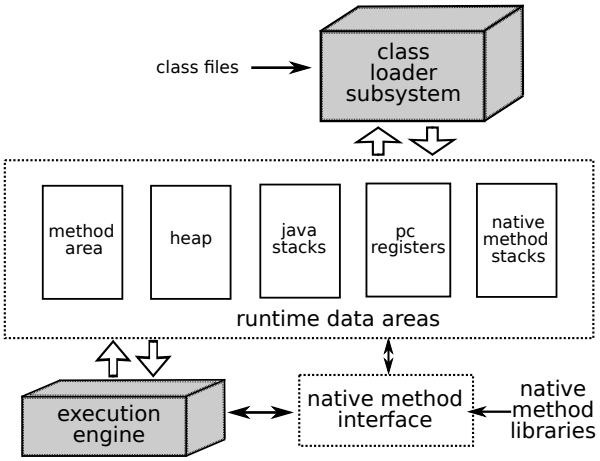
Instruction Scheduling se snaží co nejvíce vytěžit z takzvaného pipeliningu, kdy je zároveň zpracováváno více instrukcí a každá se nachází v jiné fázi zpracování.

3.5 Java virtual machine (JVM)

Java dnes patří mezi nejoblíbenější staticky typované jazyky. Na internetu existuje její kompletní specifikace založená na popisu subsystémů, paměťových míst, datových typů a instrukcí. Každý si tedy může vytvořit svůj vlastní virtuální stroj. Zdrojové soubory Javy jsou nejprve zkompilovány do bytecodu, který je tvořen sekvencí instrukcí. Ten může být buď interpretován, či přeložen do nativního kódu za pomocí JIT kompilátoru. Obě dvě metody tvoří takzvaný execution engine.

Každý virtuální stroj (viz. obrázek 3.2) Javy[14] disponuje subsystémem pro nahrávání class souborů, které příslušným způsobem rozpracuje. Runtime využívá data z několika různorodých oblastí. Některé jsou sdílené, jiné naopak jednotlivá vlákna střeží a nikoho jiného do nich nepustí. Halda (heap) a takzvaná "Method area" se v JVM nacházejí pouze v jedné kopii. Tyto oblasti jsou sdíleny mezi aktuálně běžícími vlákny. "Method area" uchovává informace o datech (typech), které byly načteny ze zmíněných class souborů. Všechny nové instance vytvořené za běhu programu jsou ukládány do oblasti označované jako "heap".

Každé nově vytvořené vlákno obdrží svůj vlastní zásobník (java stack) a "pc register", který ukazuje na konkrétní místo (instrukci) v bytekódu. Na "java stack" se ukládají data pro všechny zavolané metody, které však nejsou nativní. Pro ně je určen tzv. "native method stack". Bytecode obsahuje několik instrukcí majících na starosti volání funkcí (invokestatic, invokevirtual,..). Obě dvě se odkazují na záznam v "constant poolu", ze kterého se získají bližší informace o metodě (název, deskriptor,..). Po jejím zavolání se na zásobníku vytvoří nový "frame" obsahující lokální proměnné, parametry a v neposlední



Obrázek 3.2: Interní architektura Java Virtual Machine

řadě zásobník operandů. Po provedení instrukce return dojde k odebrání framu ze zásobníku.

3.5.1 Datové typy

Typ	Rozsah
byte	8-bitový integer se znaménkem, dvojkový doplněk, od -2^7 do $2^7 - 1$
short	16-bitový integer se znaménkem, dvojkový doplněk, od -2^{15} do $2^{15} - 1$
int	32-bitový integer se znaménkem, dvojkový doplněk, od -2^{31} do $2^{31} - 1$
long	64-bitový integer se znaménkem, dvojkový doplněk, od -2^{63} do $2^{63} - 1$
char	16-bitový Unicode znak
float	32-bit IEEE 754 single-precision float
double	64-bit IEEE 754 double-precision float
reference	Ukazatel na objekt uložený na heapě, nebo null

Tabulka 3.1: Datové typy jazyka Java

Na Javě je zajímavé, že ji zcela chybí číselné typy bez znaménka. Například typ Datum databáze PostgreSQL je typu ”unsigned long”, který bychom museli namapovat na znaménkový ”long” Javy.

3.5.2 Obsah class souboru

Constant Pool - Každý typ (třída), který JVM nahraje do své vnitřní reprezentace obsahuje takzvaný Constant pool. Uvnitř jsou záznamy uloženy

ve formě uspořádané množiny konstant, na které se typ za běhu programu odkazuje. Jedná se o hodnoty typu "string", "integer", "float" a o reference na metody, fieldy a jiné typy. Constant pool hraje zásadní roli při linkování typů (tříd) mezi sebou a to zejména díky zmíněným referencím. Složitější položky jsou často organizovány hierarchicky. Například položka týkající se metody obsahuje další dvě reference, kde jedna odkazuje na mateřskou třídu a druhá na název metody včetně jejího deskriptoru.

Informace o třídních proměnných (fields) - Kromě názvu, typu a modifikátoru (public, static,...) se uchovává také původní pořadí v class souboru, které je důležité například při implementaci dědění.

Informace o metodách - Mimo následně uvedených informací je třeba jako u třídních proměnných zachovat také pořadí. Mezi ukládané hodnoty patří název, návratový typ, typ argumentů včetně jejich počtu a modifikátory. Zvláštní pozornost si zaslouží pole speciálních atributů, jehož položky odpovídají formátu uvedeném ve výpisu 3.2.

Výpis 3.2: Struktura attributu

```
attribute_info {
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_length];
}
```

Struktura "attribute_info" se svoji stavbou zásadně neliší od jakékoliv jiné informace uložené v class souboru. Její první dva byty odkazují do Constant poolu na název atributu. Následující čtyři byty určují délku vnitřní struktury "info" v bytech. Jaký byl to byl program bez uloženého mezikódu. Existuje pro něj speciální atribut s názvem "Code", který je reprezentován strukturou "Code_attribute". Uvnitř se nachází mnoho informací. K nejvýznamnějším patří pole "code" obsahující samotný bytecode a pole exception_table[]. Každý záznam v "exception_table" definuje jeden "exception handler".

Výše uvedené položky odpovídají větším strukturám. Soubor class však obsahuje několik bytů, které ukazují na záznamy v Constant poolu. Postupným procházením bychom našli ukazatele na třídu, nadtřídu a na implementovaná rozhraní. Mezi nejprimitivnější položky o velikosti dva byty patří počty metod, třídních proměnných a záznamů v Constant poolu. Ještě nebyly zmíněny doplňující atributy class souboru a informace o verzi.

3.5.3 Java Native Interface

Bytecode Javy neobsahuje instrukce, které by uměly komunikovat s okolním světem. Tím je myšlen například zápis do souboru, výpis textu do konzole,

otevření síťového spojení, atd. K výše zmíněným účelům slouží nativní metody jejichž zdrojový kód je napsaný v jazyku C / C++. Uvnitř standardní knihovny Javy se nachází velké množství nativních metod. (pro práci se systémovým časem, pro kopírování pole, atd).

Java Native Interface [15] lze chápat jako spojovací most mezi jazykem Java a C / C++, po kterém lze chodit oběma směry. Toto rozhraní vzniklo také z důvodu potřeby napojení se na knihovny systému a další platformně specifické věci.

3.5.4 AMS

Úkolem ASM [16] knihovny je generovat, transformovat a analyzovat zkompilované Java class soubory reprezentované jako pole bytů. Z tohoto důvodu ASM poskytuje techniky pro čtení, zápis a transformaci takového pole na vyšší úrovni. Za pomocí vhodně zvoleného API můžeme pracovat s řetězci, konstantami, třídami, metodami, strukturovanými elementy, atd.

3.5.5 Možnost integrace do PostgreSQL

Případná integrace JIT kompilátoru Javy do PostgreSQL se nejeví jako vůbec jednoduchá. Strom expression výrazu by bylo třeba transformovat na class soubor, kterému by JVM rozuměla. Přesněji řečeno na bytecode určité metody. Ke zmíněné transformaci by mohl posloužit uvedený nástroj ASM. Potíž je v tom, že kód generující class soubory je třeba psát v Javě. My bychom potřebovali nástroj určený pro jazyk C. Další nepříjemnosti lze očekávat při vzájemné komunikaci Java -> C a naopak. V PostgreSQL se občas střídá kód, který souvisí přímo s vykonáváním expression výrazu, s kódem, který pouze například připravuje funkce před zavoláním či načítá data. Ke zmíněným účelům by posloužil popsáný Java Native Interface. Transformace jednoduchého matematické výrazu by zas tak složitá být nemusela. Určité nepříjemnosti by přišly na scénu v případě volání funkcí. Java Native Interface podporuje určitou konverzi datových typů mezi oběma jazyky, ale i v tomto směru by bylo nutné počítat s určitými problémy.

Java konkrétně disponuje dvěma typy JIT kompilátorů [17]:

- klient - na základě určité šablony překládá bytecode do nativního kódu. Neprovádí žádné složité optimalizace.
- server - provádí různé lokální a globální optimalizace a delší dobu stráví nad alokací registrů procesoru. Ovšem aby tento typ překladače pracoval skutečně efektivně, musí mít k dispozici větší množství informací o dynamických vlastnostech metody/smyčky. Tyto informace získá opět z interpretru. Zjednodušeně řečeno to znamená, že čím vícekrát je metoda/smyčka zpracována pomalým interpretrem, tím lepší informace má k dispozici optimalizující JIT překladač.

Z výše uvedeného taktéž vyplývá, že JIT překladače typu server nemusí být vždy výhodnější a pro aplikace běžící jen krátkou dobu může být JIT překladač typu client lepší volbou.

Na závěr se sluší dodat, že JIT kompilátor Javy patří mezi to nejlepší, co v dnešní době existuje. Tvrzení potvrzuje provedené měření, které se nachází v další kapitole. Bohužel je silně provázán s Javou.

3.6 Lua VM / LuaJIT

3.6.1 Zásobníkové vs. registrové virtuální stroje

Virtuální stroje mají za sebou dlouhá léta vývoje a není proto divu, že vzniklo mnoho jejich implementací [18] společně s bytekódy. Tak jako existují vysokoúrovňové a nízkoúrovňové programovací jazyky, podobné příklady bychom našli ve světě bytekódů. Vysokoúrovňové například mohou podporovat polymorfismus a oproti tomu nízkoúrovňové jsou blíže strojovému kódu. Naprostou většinu bytekódu lze rozdělit do dvou skupin [18]:

Založené na zásobníku operandů - S tímto typem bytekódu jsme se již setkali v kapitole 3.5 věnující se JVM. Bajtkódy a virtuální stroje využívající zásobník operandů a instrukce pro práci s argumenty uloženými na tomto zásobníku většinou obsahují mnoho bezparametrických instrukcí. Jejichž operační kódy tak mohou být velmi krátké a typicky bývají uloženy v jednom bajtu. Mezikód byl velice oblíbený v dobách osmibitových procesorů, protože jeho interpret ke své práci nepotřeboval mnoho pracovních registrů.

Využívající sadu registrů virtuálního stroje - Jedná se o bytekódy, jejichž instrukce dokážou pracovat s obsahem množiny pracovních registrů zvoleného virtuálního stroje. Počet pracovních registrů významně ovlivňuje délku instrukčního slova a možnosti bytekódů. V dnešních VM se setkáváme minimálně s šestnácti či dvaatřiceti registry, což znamená, že mnoho instrukcí má délku minimálně dva bajty. Můžeme se ale také setkat s tříbajtovými a čtyřbajtovými instrukcemi. Aby toho nebylo málo, odlišují se od sebe počtem operandů. Některé bytekódy využívají takzvaný dvouadresový kód. V takovém případě jeden registr reprezentuje zdroj a druhý registr se používá, jak zdrojový, tak cílový zároveň. Další bytekódy pracují s tříadresovým kódem. V jeho případě již není potřeba jeden registr využívat jako zdrojový a cílový. Přibyl totiž další registr, který převzal jednu z rolí.

Způsob interpretace bytekódů využívajících registry VM může být problematičtější v případě, že mikroprocesor, na němž interpret běží, obsahuje menší množství fyzických pracovních registrů. Bylo ale zjištěno, že při použití JIT kompilátoru se rozdíly mezi oběma způsoby práce s operandy do značné míry rozostřují.

3.6.2 Bytekód využívaný jazykem Lua

Bytekód [18] jazyka Lua se v mnoha ohledech odlišuje od Javy. Pravděpodobně nejnápadnějším rozdílem mezi bajtkódem JVM a bytekódem jazyka Lua je fakt, že se v Lua VM nepoužívá zásobník operandů, protože indexy operandů jsou přímo součástí instrukčního slova. JVM se dále odlišuje tím, že kód instrukce ukládá v celém bajtu (s několika málo výjimkami), zatímco Lua VM kód instrukce ukládá v pouhých šesti bitech, zatímco zbylých 26 bitů instrukčního slova je rezervováno pro uložení indexů operandů či konstant. Bytekód Lua VM taktéž obsahuje spíše vysokoúrovňové instrukce, které dobře reflektují vlastnosti tohoto programovacího jazyka. Jelikož se chceme zaměřit na JIT kompilátor jménem LuaJIT, který používá pozměněnou instrukční sadu, bytekód samotného původně interpretovaného jazyka Lua zde nebude více rozebírán.

3.6.3 Skriptovací jazyk Lua

Programovací jazyk Lua [19] patří do poměrně rozsáhlé a stále častěji používané skupiny vysokoúrovňových skriptovacích jazyků, do níž můžeme zařadit například populární Python, Perl, Scalu, Ruby a na síle nabývající JavaScript. Tyto programovací jazyky nabízí vývojářům jednoduchou práci se strukturovanými daty (většinou je použita nějaká forma asociativního pole), dynamicky typované proměnné, automatickou správu paměti (garbage collector) a mnohé další vysokoúrovňové techniky zjednodušující a zrychlující vývoj. Jazyk Lua má navíc velmi jednoduchou syntaxi inspirovanou Modulou a Pascalem, zatímco sémantika jazyka se v mnohém podobá spíše moderním verzím JavaScriptu. Navíc je přímo v jazyku zabudovaná podpora pro paralelní programování, což je stále oblast, kterou mnohé výše zmíněné jazyky prozatím nepodporují. Lua nepřináší žádné nové zbytečné syntaktické prvky. Naopak se snaží programátorům ulehčovat život tím, že je možné najednou přiřadit hodnotu více proměnným. Dále například obsahuje syntaktický cukr ve formě tečkové notace - structure["value"] = structure.value.

Lua se na trhu uplatnila v celé řadě odvětví (ve vestavných systémech, klávesnicích, VLC přehrávači, hrách, atd.) Několik celosvětově známých her používá skriptovací jazyk Lua pro jeho jednoduchost, malost (méně než 200KB), přenositelnost a vestavitelnost (C/C++, Java, Python,..). Často bylo potřeba herní engine vytvořit co nejefektivněji s využitím všech možností nízkoúrovňového jazyka. Lua naopak umožnila herní scénář a logiku hry naskriptovat s co největším urychlením cyklu oprava–překlad–spuštění. Na následujících několika řádcích si ještě uvedeme další významné prvky jazyka:

- Na rozdíl od programovacích jazyků odvozených syntakticky od Céčka se v Lua mnoho konstrukcí zapisuje pomocí klíčových slov, zatímco „céčkové jazyky“ z větší míry využívají speciální znaky typu {, } či &

- V programovacím jazyku Lua lze používat osm datových typů (nill, boolean, number, string, function, userdata, thread, table). Jazyk patří mezi dynamicky typované jazyky, tj. typy nejsou přiřazené proměnným, ale jejich hodnotám. všechny složitější datové typy jsou implementovány pomocí typu table (objekty, pole, seznamy). Příklad konstruktoru : {x = 5, y = 10}. Každá tabulka může mít přiřazenu další tabulku (metatable)

- Při spuštění interpretru je vytvořeno několik globálních objektů (proměnných, konstant a funkcí), které může skript při svém běhu využít. Bytekód se na tyto prvky odkazuje pomocí čísel.

- Jazyk implementuje takzvané first-class funkce včetně anonymních. Můžeme je přiřazovat proměnným a předávat je jako parametry jiným funkcím. Takovou třešničkou na dortu je, že lze vracet více hodnot.

- Objekty jsou tvořeny kombinací tabulek a first-class funkcí. Pro funkce a objekty existuje opět syntaktický cukr. Například (”function a:function_name (x)” odpovídá ”a.function_name = function (self,x)”). Takovým funkcím se předává skrytě jako první tabulka reprezentující objekt.

- Delegace - Pokud tabulka ”a” deleguje z tabulky ”b”, tak každá proměnná chybějící v ”a” je získána z ”b”. Využívá se takzvaných metametod (___index, ___add,...). Metametoda ___index zajišťuje získání proměnné (funkce) z metatabulky. Například se využívá při tvorbě nových instancí, které vycházejí z nějaké základní tabulky (třídy).

3.6.4 LuaJIT

LuaJIT [20][21] je z velké části dílem jediného programátora. Což si zaslouží obdiv, protože LuaJIT je v současné verzi velmi kvalitním produktem podporující větší množství procesorových architektur. Zatímco bytekód jazyka Lua je optimalizován dle možností interpretru, mezikód LuaJITu byl navržen s ohledem na snadnost a rychlost překladu do nativního kódu vybrané architektury mikroprocesoru. IR používá takzvaný tříadresový kód a instrukce o pevné šířce třiceti dvou bitů. Existují dva způsoby rozdělení dvaatřicetibitového slova na jednotlivá bitová pole:

1. První formát se používá u instrukcí s trojicí operandů. Postupně za sebou následuje: 8-bit. vstupní operand (zdrojová proměnná), 8-bit. druhý vstupní operand (zdrojová proměnná, numerická konstanta atd.), 8-bit. výsledek operace (proměnná pro uložení výsledku), 8-bit. operační kód instrukce.

2. Druhý formát dokáže adresovat pouze dva operandy, ovšem první z nich má šířku šestnáct bitů a lze ho v některých instrukcích použít například k přímému uložení šestnáctibitové celočíselné konstanty se znaménkem

V obou formátech má operační kód instrukce šířku osmi bitů, obsazení dalších čtyřiadvaceti bitů je však odlišné. Následující tabulky ukazují vybrané instrukce:

Instrukce	A	D	Popis
KNIL	base	base	Nastaví sloty číslo A až D na hodnotu nil
KSHORT	dst	lits	Nastaví A na 16 bit. integer se znaménkem v D
KNUM	dst	num	Nastaví A na číselnou konstantu D
MOV	dst	var	Zkopíruje data z D do A

Tabulka 3.2: Instrukce pro práci s konstantami a pro přesun hodnot mezi registry

Instrukce	A	B	C/D	Popis
RET0	rbase	lit		Návrat z funkce
CALL	base	lit	lit	Zavolá funkci jejíž ukazatel je v A. Parametry: A+1, A+2, A+C-1,Návratové hodnoty: A+1, A+2, A+B-2

Tabulka 3.3: Instrukce pro volání funkce a vrácení hodnot

Z výše uvedené tabulky 3.3 vyplývá, že návratové hodnoty jsou uloženy do slotů s indexy A až A+B-2. Konstanta C tedy ukládá počet předávaných parametrů a konstanta B počet návratových hodnot.

LuaJit obsahuje i další instrukce. Například instrukce GGET, GSET získávají, nastavují položky v globální tabulce, ze které například můžeme získat reference na funkce. Nad standardními tabulkami pracují instrukce TGETV, TGETS, TGETB, TSETV, TSETS, TSETB, TNEW, TDUP, TSETM. Přehled aritmetických instrukcí je uveden v příloze A.1. Na závěr přehledu instrukcí je nutno podotknout, že standardní bytekód jazyka Lua verze 5.1 pracuje pouze s osmatřiceti instrukcemi. LuaJIT jich zavádí přibližně dvakrát více. (například Lua disponuje instrukcí ADD pro sčítání, ale Lua JIT používá hned tři - ADDVN, ADDNV, ADDVV).

LuaJIT využívá takzvaný „trasovací JIT překladač“, který se v několika ohledech odlišuje od dnes známějších JIT překladačů typu hot spot, který například nalezneme v JVM.

Činnost trasovacích JITů [20] je založena na dvou předpokladech. Prvním předpokladem je, že typické aplikace tráví nejvíce strojového času v programových smyčkách (předpoklad je někdy rozšířen i o tail rekurzi). Druhým předpokladem je, že pokud se vykonává programová smyčka, bude cesta v kódu pravděpodobně vždy stejná popř. v horším případě že bude existovat jen několik cest v programovém kódu (cestou je myšlena sekvence instrukcí). Trasovací JITy založené na těchto předpokladech se soustředí na detekci tzv.

hot-loops, tedy často vykonávaných programových smyček. Tyto smyčky jsou následně optimalizovány a přeloženy do nativního kódu mikroprocesoru. Trasovací JITy se obecně zaměřují na sledování sekvence generovaných a následně prováděných instrukcí a nikoli na potenciálně složitou analýzu bytekódu.

Při optimalizacích se provádí mnoho operací, s nimiž se můžeme setkat také v běžných překladačích – eliminace mrtvého kódu , rozbalení smyček (snižuje se počet skoků a tím pádem se vylepšuje využití instrukční pipeline) atd. Detekce hot-loops byla v tradičních trasovacích JITech implementována analýzou zpětných podmíněných skoků (vedoucích na začátek smyčky), ovšem v LuaJITu to není nutné, a to díky speciálním instrukcím bytekódu: LOOP a FORI.

Zpracování programu v trasovacích JITech je rozdělena do několika fází:

1. Interpretation+Profiling - v této fázi se provádí interpretace bytekódu a sumarizace, kolikrát se daná programová smyčka provedla.

2. Tracing - na základě speciálního běhu interpretru se zaznamenává historie prováděných instrukcí.

3. Code generation and Optimization - překlad vybrané stopy/stop do nativního kódu s provedením optimalizací, které byly naznačeny v předchozím odstavci či v podkapitole 3.4 věnující se převážně expression výrazům.

4. Execution - Na konci se již spouští samotný nativní kód místo původního interpretru.

Po nalezení vhodné stopy je sekvence instrukcí bytekódu tvořící tuto stopu přeložena do sekvence pseudoinstrukcí. V tomto mezistupni se již pracuje s konkrétními datovými typy a samotný formát pseudoinstrukcí je zvolen takovým způsobem, aby se dobře prováděly optimalizace a poslední krok. V něm se transformují pseudoinstrukce do nativního strojového kódu. Využívá se připravených šablon pro konkrétní typy mikroprocesorů, kde hlavním úkolem JITu je alokace registrů.

Ve vztahu k expression výrazům LuaJIT mimo jiné provádí optimalizace [21] jako "Constant Folding, Algebraic Simplifications, Elimination of Unneeded Results, Reassociation, Strength Reduction, Function Inlining a mnoho jiných". Některé se přímo či nepřímo vztahují k optimalizacím zmíněným v podkapitole 3.4. LuaJIT také dokáže velmi efektivně využít SSE2 technologie, o které je několik řádek napsáno v podkapitole 3.2.

3.6.5 Možnost integrace do PostgreSQL

Integrace LuaJITu do PostgreSQL se na první pohled jeví snadnější než v případě JVM. Bytekód Javy čítá cirka 200 instrukcí, zatímco LuaJit pracuje přibližně s 60. Nemuseli bychom řešit žádný registr operandů, jelikož bychom

pracovali s registry virtuálního stroje. Obecně bychom se nemuseli zabývat podstatně složitější strukturou JVM. Komunikace Lua -> C a C ->Lua je příjemnější než v Javě a to díky minimalismu jazyka Lua, který se velice snadno integruje do jazyka C. Java Native Interface je oproti tomu veliký moloch, se kterým je spjata velká režie v podobě dalšího kódu. Expression výraz odeslaný ke zpracování by bylo třeba převést na soubor s bytekódem, kterému by LuaJIT rozuměl. Zajisté by bylo nutné vyřešit plnění jednotlivých slotů, vzájemnou konverzi hodnot s ohledem na to, že Lua je dynamicky typovaný jazyk; generování instrukcí na základě výrazu; volání nativních funkcí jazyka C - například pro načtení hodnot z databáze. Také není zcela zřejmé, jak dlouho by takové generování struktury souboru s bytekódem trvalo.

3.7 LLVM

Low Level Virtual Machine (LLVM) [22] dodává potřebnou infrastrukturu a sadu nástrojů pro vývoj kompilátorů pro různé programovací jazyky. Nejvýznamnějším prvkem infrastruktury je takzvaný LLVM IR (mezikód), od kterého se odvíjí následný vývoj. Návrh už od začátku počítal s tím, že se bude jednat o sadu knihoven a ne o žádný monolitický kompilátor spustitelný z příkazové řádky. Design nelze ani přirovnávat k jasně specifikovaných virtuálním strojům (Java, .Net). Jsme-li schopní zkompilovat jazyk do bytekódu JVM, tak můžeme využívat všech výhod jejího runetimu, optimalizátoru, JITu, atd. Musíme ale přijmout fakt, že ztrácíme určitou flexibilitu ve formě volby runetimu. Například v případě kompilace jazyka C do bytekódu JVM to může vést k ne zcela optimálnímu výkonu.

3.7.1 LLVM implementace třífázového designu

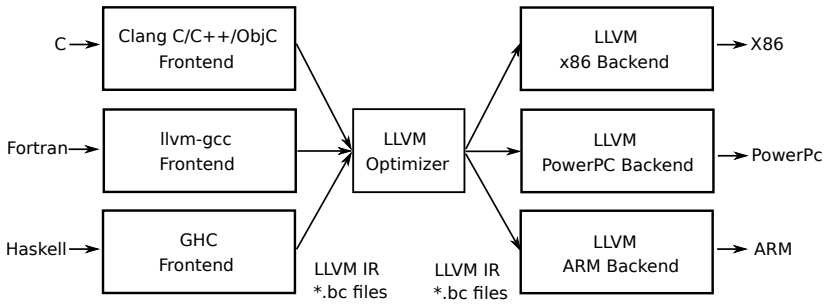
Celý kompilátor založený na LLVM [23]se skládá ze tří důležitých částí (viz. obrázek 3.3):

1. Frontend - tento pojem dobře známe například z vývoje webových aplikací, kde frontend slouží k interakci programu s uživatelem a také k získávaní vstupních dat. V případě kompilátorů je vstupem zdrojový kód vyššího programovacího jazyka. Frontend provede parsování, validaci a kontrolu chyb na vstupu. Následně se získaný výstup přeloží do LLVM IR. Je třeba dodat, že se velice často využívá převodu na abstraktní syntaktický strom a až poté dojde k transformaci do formátu odpovídajícímu IR.

2. Optimizer - v této fází se nad IR provede série vybraným optimalizací, které by měly vést k celkovému zrychlení běhu programu.

3. Backend - výsledek bloku vykonávající optimalizace je předán backandu (generátoru), který převede tří-adresovou strukturu LLVM IR nejprve

na DAG (orientovaný acyklický graf) a následně do nativního kódu zvolené cílové architektury. Provádí se zde operace jako: alokace registrů, transformace smyček, optimalizace specifické pro konkrétní cíl, atd. Zde je nutné podotknout, že LLVM nabízí neomezené množství virtuálních registrů.



Obrázek 3.3: LLVM implementace třífázového designu kompilátorů

Poměrně novou a výbornou vlastností LLVM je, že lze napsat zcela nezávislý frontend, který vyprodukuje textovou formu IR. LLVM kromě binární reprezentace totiž podporuje také ”first-class” textový formát mezikódu. Následně můžeme použít rouru v Linuxu a na vygenerovaný kód spustit optimalizátor. Výstup lze opět vložit do absolutně nezávislého generátoru nativního kódu, který na vstupu očekává IR. Je s podivem, že architektonicky výborné GCC kompilátory takovou nezávislost plně neumožňují.

3.7.2 Vybrané Nástroje LLVM

V předchozí podkapitole bylo uvedeno, že lze celý program zkompilovat postupně [22]. To však neznamená, že překlad nelze provést jedním příkazem. Můžeme použít ”kompilovacího driveru”, který uživatele odstíní od průchodu jednotlivými fázemi. V případě programovacího jazyka C ve spojení s LLVM a ”Clang”em jde o utilitu ”clang” Například příkaz ”clang hello.c –o hello” provede kompilaci. Následuje seznam vybraných příkazů, kterými LLVM disponuje:

- opt - tento tool je zaměřen na optimalizaci programu na úrovni IR reprezentace.
- llc - tool konvertuje LLVM bytekód do strojového jazyka cílové architektury ve dvou formátech (language file či object file) přes specifikovaný backend.
- lli - tento tool implementuje jak interpreter, tak JIT kompilátor pro LLVM IR.

3.7.3 Intermediate representation (IR)

LLVM IR [23] je navržen tak, aby byl co nejvíce nezávislý na cílové architektuře. Na druhou stranu například některé knihovny jazyka C a typy jsou platformově závislé, což představuje určitou výzvu při generování IR. V příloze B.1 je ukázán překlad jednoduché funkce. Důležité vlastnosti:

- LLVM IR používá takzvanou Static Single Assignment (SSA) formu. Což jednoduše řečeno znamená, že žádná hodnota není přiřazena dvakrát. Vždy je vytvořena další proměnná. U LuaJITu to nebylo zmíněno, ale také s touto formou pracuje. Její dodržování usnadňuje a zrychluje některé optimalizace (například zjednodušování aritmetických výrazů).
- Instrukce pracují se třemi operandy - zdroj, zdroj, cíl. (tak tomu bylo také u LuaJITu).
- Není omezen počet použitých registrů. Jelikož se následně mapují na omezený počet fyzických registrů, tak některé hodnoty mohou být uloženy v paměti.

Ve výpisu 3.3 jsou ukázány vybrané funkce, které slouží ke generování IR a mohli by se hodit při případné integraci. Pod výpisem 3.3 se nachází popis očíslovaných metod.

Výpis 3.3: Ukázka vybraných funkcí, které generují IR

```
1) BinaryOperator::Create(instr, lhs.codeGen(context),
rhs.codeGen(context),..);

2) Value* ConstantFP::get(Type::getDoubleTy(
globalContext),value);

3) Function *fn = context.module->getFunction(name);
CallInst *call = CallInst::Create(fn, args.begin(),
args.end(), "",context.currentBlock());

4) StoreInst(...);

5) FunctionType* fnType = FunctionType::get(
Type::Int32Ty, argList, false);
```

1. Metoda slouží ke generování IR, který provádí binární aritmetickou operaci. ”instr” může odpovídat podle typu operace například Instruction::Add, Instruction::Add, Instruction::Sub, atd. Jako argumenty je potřeba předat levý a pravý podstrom - lhs.codeGen(context), rhs.codeGen(context).
2. Vytvoří konstantu typu double.

3. Slouží k vytvoření kódu majícího na starosti volání funkce.

4. Metoda slouží pro přiřazení hodnoty.

5. Vytvoří se nový typ funkce

Metoda codeGen slouží ke generování příslušného IR pro jeden konkrétní uzel abstraktního syntaktického stromu. S každým novým typem uzlu přichází jiné tělo metody (volání funkce, deklarace funkce, binární operátor, přiřazení hodnoty, atd).

3.7.4 LLVM Execution Engine

LLVM disponuje exekučním enginem [23] pro spouštění modulů. Jeho návrh podporuje implementace založené, jak na interpretování, tak na Just-In-Time kompilaci. Třída ExecutionEngine.h je zodpovědná za spuštění požadovaného programu a za provedení všech potřebných příprav. Obecně, JIT zapíše do paměti takzvané "binary blobs". ExecutionEngine poskytuje ukazatel na funkci, pomocí kterého můžeme ukázat na konkrétní místo v paměti s vygenerovanými instrukcemi, které se mají provést. LLVM původně implementovalo llvm::JIT framework. Později byl přidán llvm:MCJIT, který od verze 3.5 zcela nahradil původní llvm::JIT. Přibily například různé stavy modulů, atd. Na rozdíl od LuaJITu standardní verze llvm::MCJITu nepodporuje trasování. Celý engine je jako celé LLVM co nejvíce platformově nezávislý. V případě přidání vlastního JITu pro konkretní cílovou architekturu je pro ni třeba implementovat takzvaný "binary instruction emission step".

V literatuře se lze dočíst [23], že se LLVM soustředí převážně na kvalitní optimalizace prováděné při samotné Just-In-Time kompilaci a je brán spíše jako statický kompilátor. Jeho JIT není schopný se učit za běhu programu a provést případnou rekompilaci. Proto se spíše hodí například pro déle běžící webové aplikace, na které byl již úspěšně použit. Na nich se tolik neprojeví počáteční "overhead", který by jiné aplikace (běžící kratší dobu) mohl negativně ovlivnit.

3.7.5 Možnost integrace do PostgreSQL

LLVM je jeden z hlavních kandidátů na integraci. Mezi jeho hlavní výhody patří třífázový design a generátor kódu IR. Nebylo by tedy nutné se zdlouhavě zabývat strukturou bytekódu. Splňuje téměř všechny požadavky (obsahuje pomocné metody pro volání nativních funkcí, generování binárních operátorů, přiřazování hodnot, atd). V samotném textu výše se píše, že se zdrojový kód často překládá na AST a až poté do IR. Strom expression výrazu byl detailněji popsán v podkapitole 2.3.6. Stačilo by každému uzlu přidat metodu codeGen(), ve které by se nacházel kód generující výsledný IR. LLVM umožňuje zvolit si vlastní sadu optimalizací (tvz. jednotlivé optimization passes

- createBasicAliasAnalysisPass(), createGVNPass() pro eliminaci podvýrazů, atd). Vlastní volbou vhodných optimalizací pro expression výrazy by bylo možné upozadit počáteční "overhead". Další plus si LLVM připisuje za alokaci registrů, kterých je neomezený počet. Mezi největší nevýhody patří, že se JIT nedokáže učit za běhu a samotná velikost celého LLVM.

3.8 LibJit

Zatím probírané JIT kompilátory mimo LLVM byly zcela závislé na svém mateřském jazyku a virtuálním stroji. Vývoj LibJit [24] knihovny se vydal zcela opačným směrem. Nejenže nepracuje nad bytekódem jiného programovacího jazyka, ale ani se žádným není svázán. Tato knihovna napsána v jazyku C vznikla původně jako součást většího projektu DotGNU (portable .NET runtime). Jelikož se jedná open-source , tak jeho největší autoritou je poslední známá verze zdrojového kódu. Najít hlubší informace popisující všechny možné vlastnosti knihovny je témě nemožné. Dokumentace se většinou zaměřují na praktickou stránku věci a na ukázkové příklady. Nejlepším zdrojem informací je tedy samotný zdrojový kód s komentáři, kde se však vyskytují především komentáře samotných metod a celkový obraz si člověk musí udělat sám.

LuaJIT interně pracuje stejně jako většina ostatních kompilátorů s mezikódem IR. Pokud však nebude mít pro cílovou architekturu žádný generátor instrukcí, začne IR interpretovat.

3.8.1 LibJit API

Kromě výhody absolutní nezávislosti LibJit nabízí velice pěkné a přehledné API [24] pro generování jeho vnitřní reprezentace IR. LibJit stejně jako LuaJIT či LLVM využívá formátu tříadresových instrukcí (zdroj, zdroj, cíl). Na druhou stranu ukládá IR pouze v paměti a neexistuje možnost uchování jeho binární či textové formy na disku. Mezi nejdůležitější datové struktury patří:

- jit_function_t - datová struktura zastřešující funkci. Výpis kódu 3.4 ukazuje vytvoření nové funkce v LibJitu. Nejdříve se zavede kontext, do kterého daná funkce bude patřit. Následně se připraví pole reprezentující typy parametrů. Co by to bylo za funkci bez definované signatury. Její první parametr říká, jaká volací konvence má být použita. Konkrétně typická "cdecl" pro jazyk C byla detailněji rozebrána v podkapitole 3.2.2. Druhý parametr specifikuje typ návratové hodnoty. Další přijímá připravené pole typů. Poslední dvě hodnoty upřesňují počet předávaných parametrů a počet vrácených hodnot z funkce. Funkce na rozdíl od LuaJITu může vracet maximálně jednu hodnotu. Po instrukcích tvořících tělo funkce se provede kýžená kompilace. Na závěr již nic nebraní samotnému volání funkce použitím odpovědné metody. Ta na svém vstupu očekává adresy parametrů a ukazatel na návratovou hodnotu.

- jit_value_t - datová struktura reprezentující hodnotu. Pro jejich vytváření a inicializaci se používá funkce jit_value_create(fn, jit_type_int). První parametr je typu jit_function_t a druhý jit_type_t. Pokud se však bude jednat o konstantní hodnotu, je třeba využít například schopností funkce s názvem jit_value_create_nint_constant(fn, jit_type_int, 1), která vytvoří celočíselnou konstantu nastavenou na jedničku.

- jit_label_t - datová struktura popisující místo v kódu, na které lze skočit (podmíněně, nepodmíněně). Přesná pozice se specifikuje funkcí jit_insn_label(fn, &label); Například funkce jit_insn_branch_if(fn, isTrue, &label) zapříčiní skok v případě pravdivé hodnoty isTrue na místo specifikované popiskem &label.

- jit_type_t - datová struktura reprezentující typ. Ve výše napsaném textu již byl uveden typ jit_type_int. V podkapitole 2.3.5 o expression výrazech bylo napsáno, že jednotlivé funkce PostgreSQL vracejí typ Datum. Jelikož se interně jedná o typ ulong, použil by se pro něj jit_type_ulong. Pro ukazatele slouží jit_type_void_ptr.

Výpis 3.4: Vytvoření nové funkce se dvěma parametry v knihovně LibJit

```
jit_context_build_start(context);
jit_type_t params[2] = {jit_type_int, jit_type_int};
jit_type_t sigtr = jit_type_create_signature(
jit_abi_cdecl, jit_type_int, params, 2, 1);
jit_function_t fn = jit_function_create(context,sigtr);
... // instrukce tvorici telo funkce
jit_context_build_end(context);
jit_function_compile(fn); // kompilace

//volání
void* args[2] = {&a, &b}; // adresy argumentu
jit_int result;           // misto pro vysledek
jit_function_apply(fn, args, &result); // volání
```

LibJit obsahuje sadu dalších funkcí generujících IR (aritmetické, porovnávací, pro načítání hodnot z/do paměti, pro tvorbu složitějších struktur a mnoho dalších). Pomocí ukazatele a funkce pro načítání hodnot z paměti není problém přečíst libovolnou hodnotu původně uloženou pomocí jazyka C. Adresu si můžeme například uložit do konstanty výše uvedeným způsobem.

Opravdu velká výhoda spočívá ve faktu, že se často nemusíme starat o typ vstupních parametrů. Například funkce jit_insn_sub(fn, leftOperand, rightOperand) pro odečítání přijímá parametry (jit_value_t) a na základě jejich typu vygeneruje kód cílové architektury. Pro float hodnoty může využít popsaných xmm registrů, atd.

3.8.2 Volání nativních funkcí

API LibJitu samozřejmě podporuje volání jiných funkcí a to za pomocí jit_insn _call(..). Metodě se předá funkce (jit_function_t) a potřebné parametry. V následné integraci do PostgreSQL se počítá s tím, že celý strom expression výrazu bude zkompilován jako jedna funkce. Proto pro nás zmíněná schopnost není zas tak zajímavá. Daleko užitečnější je funkce s názvem jit_insn_call_native(..), která volá nativní funkce jazyka C. Určitá zvláštnost spočívá v předání čistého ukazatele na funkci, aniž by se jednalo o hodnotu typu jit_value_t. LibJit kromě pointeru potřebuje znát její signaturu, která je předtím vytvořena stejně jako ve výpisu 3.4. Hodnoty vstupních parametrů se předávají v poli, jehož prvky již odpovídají typu jit_value_t. Stejného typu je také návratová hodnota.

3.8.3 Optimalizace

Knihovna LibJit nedisponuje velkým počtem prováděných optimalizací [25]. Není zrovna lehké zjistit, které jsou implementované. Ze zkušeností z vývoje lze říci, že LibJit úspěšně odstraňuje kód, který je nedosažitelný "Dead code elimination". Určitě optimalizuje za pomocí takzvaného "Copy propagation". Více o těchto optimalizačních technikách je napsáno v podkapitole 3.4. Není třeba si dělat starosti s alokováním fyzických registrů, kterých je navíc v LibJitu neomezený počet. Při generování instrukcí používá technologii SIMD (SEE/SEE2/SEE3) pro operace s plovoucí řádovou čárkou. Na nízké úrovni provádí platformově závislé optimalizace. Mimo jiné v sobě LibJit zahrnuje sadu matematických a trigonometrických funkcí. V neposlední řadě je třeba zmínit, že podporuje rekompilaci funkcí za běhu s lepší optimalizací. Kód, který funkci zavolal obdrží ukazatel na její vylepšenou verzi.

3.8.4 Možnost integrace do PostgreSQL

LibJit jasně vítězí mezi zatím uvedenými JIT kompilátory z pohledu jednoduchosti integrace. V prvé řadě se může chlubit výborným API pro generování IR, které je jak stvořené pro integraci do PostgreSQL. Při volání nativních funkcí jazyka C se programátor vůbec nezapotí. Musí si však dávat pozor na datové typy, aby odpovídaly cílové architektuře a skutečným předávaným parametrům. V průběhu analýzy jsme identifikovali vhodný typ "jit_type_ulong" pro interní typ Datum databáze PostgreSQL. Na druhou stranu to vypadá, že LibJit neumožňuje žádné velké debuggování kódu, takže ladění chyb nemusí být zrovna lehká disciplína. Kladně lze hodnotit možnost zobrazení nativního vygenerovaného kódu. U LibJitu bude více než u jiných zkoumaných kompilátorů zajímavé sledovat, jak si povede rychlostně, jelikož nelze zjistit všechny jeho implementované optimalizace.

3.9 GNU lightning

GNU lightning [26] nabízí jednoduchý, přenositelný a dynamický generátor kódu. Neemituje žádný mezikód jako tradičně používané optimalizační kompilátory, čímž se snaží ušetřit drahocenný výpočetní čas. GNU lightning překládá kód přímo z "machine independent interface" do nativního kódu cílové architektury. Díky této vlastnosti se může chlubit malými požadavky na paměťové místo. Rozhraní pro generování instrukcí je silně ovlivněno architekturou RISC úzce spojenou s SPARC a MIPS čipy. Generátor kódu disponuje (šesti) všeobecnými registry, mezi které se však nepočítají registry sloužící pro přenos dat mezi podprocesy. Aritmetické instrukce pracují se třemi operandy. Buď můžeme použít tří registry a nebo pouze dva s tím, že poslední hodnota bude zadána přímo. Na jednu stranu se jedná o opravdu univerzální nástroj, protože dokáže generovat efektivní instrukce také pro CISC architekturu (Intel x86, atd).

Velkou nevýhodou je malý počet registrů. které kompilátor nabízí. Jejich množství je specifikováno konstantami JIT_R_NUM (caller-saved, nemusí zůstat stejné mezi voláními. Hlavní zodpovědnost uložení hodnoty je na volajícím), JIT_V_NUM (callee-saved, volající procedura očekává, že volaná funkce po ukončení zachová původní obsah registrů). Zatím jsme se bavili jen o celočíselných registrech. Pro hodnoty s plovoucí řádovou čárkou slouží ty s označením JIT_F0, JIT_F1,.., JIT_F6.

GNU lightning dále nenabízí žádné "peephole" optimalizace. "Peephole" znamená, že pokud je pro daný problém specifikována daná sada instrukcí, tak ji kompilátor neumí nahradit sadou efektivnější. (Constant folding, Algebraic SimplifIcations and Reassociation, viz. optimalizace z podkapitoly 3.4). Od překladače nemůžeme očekávat ani žádné plánování instrukcí a takzvaný "symbolic debugger". Poslední zmíněný nedostatek má za následek, že pro debuggování kódu je třeba krok po kroku procházet instrukce vygenerované pro cílovou platformu.

Z výše zmíněných důvodů bylo GNU lightning rovnou vyřazeno ze seznamu kandidátu pro integraci do PostgreSQL. My bychom chtěli využívat co nejvíce vlastností cílových architektur (větší počet registrů, SIMD - Xmm, SSE2, ..) a zmíněný kompilátor je až příliš obecný. Jeho optimalizační schopnosti také neodpovídají výběrovým kritériím, jelikož neumí zjednodušovat vygenerovaný kód. Možnost debuggování je přímo katastrofální.

Abychom kompilátor jen nehanili, musíme zmínit, že nabízí poměrně dobré rozhraní a funkce pro generování kódu. Ty však zde nebudou více rozebírány a každý si je může prohlédnout na přiloženém CD, kde se nachází kód příkladů určených pro testování a srovnávání rychlostí kompilátorů.

3.10 ASMJit

ASMJit[27] patří mezi další generátory nativního kódu, jehož API lze použít v jazyku C++. Z tohoto důvodu nebude integrován do PostgreSQL, jelikož je databázový stroj napsaný v jazyku C. ASMJit by musel rychlostně převyšovat ostatní JIT kompilátory, aby se vyplatilo mixovat C a C++ kód.

Samotný kompilátor podporuje dva backendy typu X86 a X64 s kompletní instrukční sadou (od MMX, SSE až po nové AVX2). Nejedná se o žádný velký virtuální stroj se zabudovaným kompilátorem. Snaží se pouze zrychlit a zjednodušit generování kódu. Oproti GNU lightning je úzce zaměřen, a a tím pádem může těžit z celé řady výhod nabízených zmíněnými cílovými architekturami.

ASMJit nabízí dva zcela odlišné způsoby generování nativního kódu. První koncept nazvaný "Assembler" je založen na čistém generování a sám programátor se stará o alokaci registrů, volání funkcí, atd. Výhoda spočívá ve vytvořeném API, které zaobaluje všechny instrukce, usnadňuje alokaci paměti, registrů, atd.

Druhý způsob generování kódu s názvem "Compiler" se již více podobá dříve uvedeným JIT kompilátorům. Na rozdíl od "Assembler" konceptu pracuje s virtuálními registry, kterých je neomezený počet. Dále podstatně zjednodušuje celý proces spojený s generováním kódu. Kompilátor zná pojem funkce a rozumí volacím konvencím. Každý vygenerovaný kód patří do určité funkce s prototypem, tak jak tomu bývá u jiných programovacích jazyků. ASMJit zde nebude dále více rozebírán. Případní zájemci si mohou snadno pochopitelné algoritmy učené pro testování prohlédnout na přiloženém CD. Práce s kompilátorem je poměrně intuitivní. Složitější konstrukce však v dokumentaci nejsou dostatečně popsány.

3.11 Parrot

Parrot [28] je kompletní virtuální stroj, který byl původně vyvíjen jako runtime pro Perl6. Časem se však stal univerzálním runtimem optimalizovaným pro dynamicky typované jazyky jako je právě Perl, PHP či Python. Design Parrotu počítá s přenositelností mezi jazyky, jež generují kód pro tento virtuální stroj. Například je možné napsat třídu v Perlu, jejího potomka v Pythonu a instanci použít v jazyku Tcl. Na rozdíl o Javy jeho interpret nevyužívá služeb zásobníku operandů, ale virtuálních registrů.

Zdrojový kód určený přímo pro Parrot lze psát ve dvou typech jazyků symbolických adres. Jeden z nich se jmenuje PIR a druhý se pyšní názvem PASM. Hlavní rozdíl mezi nimi je v úrovni abstrakce. Pomocí PIR není potřeba napsat tolik kódu vedoucímu ke stejnému cíli. Oba dva se kompilují do stejného bytekódu, kterému Parrot rozumí.

Pro náš účel je tento virtuální stroj nepoužitelný a to z jednoho prostého důvodu, neintegruje vlastní JIT kompilátor. Uplynul již poměrně dlouhý čas

od doby, kdy z něho vývojáři vlastní JIT odstranili. Přešlo se na filozofii, že bude lepší využít služeb již hotových řešení (LLVM JIT, LibJit, GNU lightning, atd).

Pro zajímavost, jak si interpreter Parrotu poradí s příklady určenými pro testování, budou dva vybrané z nich pomocí PIR implementovány.

Kapitola 4

Návrh a realizace

4.1 Porovnání rychlostí JIT kompilátorů

Analýza již naznačila možnosti jednotlivých JIT kompilátorů. Jako přední kandidáti pro integraci se v tuto chvíli jeví především LibJit a LLVM. Teorie je sice hezká, ale určitě by nebylo špatné provést měření, které by dokázalo všechny JIT kompilátory porovnat. Přesně tento cíl si vytyčila tato kapitola. Nejdříve budou představeny testovací algoritmy. Následně se podíváme, jaké JIT kompilátory a interpretry byly testovány a jaké byly použity přepínače při samotné kompilaci či při spuštění. Navazující grafy 4.1 až 4.5 zobrazují naměřené výsledky. V závěru se odehraje diskuse naměřených výsledků.

4.1.1 Testovací sestava

Pro testování byla použita následující testovací sestava v tabulce 4.1:

Procesor	Intel® Core™2 Duo CPU T9300 @ 2.50GHz × 2
Grafika	GeForce 8600M GT/PCIe/SSE2
Paměť	4 GB
OS	Ubuntu 14.04 LTS 64bit

Tabulka 4.1: Testovací sestava

4.1.2 Spouštění testů

Pro testovací účely byl vytvořen mnohařádkový skript v shellu. Jeho hlavním úkolem je na základě vstupním parametrů provést kompilaci, spustit desetkrát stejný test, zapsat výsledky z testu do souboru včetně výsledného průměrného času. Například příkaz "./runTest.sh java Arithmetic.java false" se postará o spuštění zmíněné sekvence operací pro aritmetický test napsaný v Javě. Třetí parametr "false" skriptu říká, zda prováděný program sám vrátí informaci o

délce jeho běhu. Pokud by byl nastaven na hodnotu "true", použil by se ke zjištění délky běhu programu příkaz "time".

Skript navíc ihned na začátku spustí příkaz sudo cpupower frequency-set -g performace, aby byl procesor ihned naplno využit. Zcela nevhodné by bylo ponechat například režim "powersave".

4.1.3 Přehled implementovaných algoritmů

Algoritmy byly vybrány s ohledem na základní testování JIT kompilátorů s vyšším důrazem na aritmetické výpočty. Snahou bylo napsat všechny algoritmy stejným stylem v různých programovacích jazycích či v jazycích symbolických adres.

- Arithmetic.xxx - v cyklu, který se provede 20000000x, postupně dojde ke sčítají, odečítání, násobí a dělí hodnot.
- Gcd.xxx - v cyklu, se 2000000x provede výpočet největšího společného dělitele čísel 715225741 a 735632797.
- Pi.xxx - algoritmus v 10000000 iteracích přibližně vypočte číslo Pí na několik desetinných míst. Jeho kód v jazyku C je uveden v příloze C.1.
- PrimeNumbersCounter.xxx - algoritmus vypočte počet prvočísel do čísla 50000. Jeho kód v jazyku C je uveden v příloze C.2.
- GaussElimination.xxx - algoritmus provede Gaussovu eliminační metodu nad přesně definovanou maticí velikosti 800x800. Výsledkem je horní trojúhelníková matice s jedničkami na diagonále. Jako jediný pracuje s operační pamětí.

4.1.4 Testované JIT kompilátory, interpretry

- Java - interpreter: zdrojový kód algoritmů byl přeložen "javac" kompilátorem. Samotné spuštění se provedlo příkazem "java -Xint $fileName", který zajistil provádění programu v režimu interpretu.
- Java - client: zdrojový kód algoritmů byl přeložen "javac" kompilátorem. Samotné spuštění se provedlo příkazem "java -client $fileName", který zajistil JIT kompilaci typu "client".
- Java - server: zdrojový kód algoritmů byl přeložen "javac" kompilátorem. Samotné spuštění se provedlo příkazem "java -server $fileName", který zajistil JIT kompilaci typu "server".
- Java - tiered: zdrojový kód algoritmů byl přeložen "javac" kompilátorem. Samotné spuštění se provedlo příkazem "java -XX:+TieredCompilation $fileName", který zajistil JIT kompilaci typu "tiered". "Tiered" přepínač

jako jediný z výše uvedených nebyl zmíněn v analýze JVM 3.5. Jeho použití způsobí JIT kompilaci metod, které získávají data o běhu programu v režimu interpretu. Tím by mělo dojít ke zrychlení jejich činnosti. Pozitivní efekt se projeví v JIT kompilátoru typu "server", který bude moci využít více informací za stejný časový úsek či se urychlí jeho spuštění.

- LLVM - clang O0: zdrojový kód algoritmů v jazyku C++ byl přeložen příkazem "clang++ -fno-use-cxa-atexit -O0 -S -emit-llvm", který posloužil jako frontend generující LLVM IR (přepínač emit-llvm). V aktuálním nastavení překladač provede jen hloupý překlad bez větších optimalizací (přepínač O0). Příkaz "lli $fileIR" zajistí provedení programu v režimu JIT kompilátoru. Bez použití "-fno-use-cxa-atexit" ve frontendu nebylo možné volat funkce z různých knihoven (pro měření času, std::cout, atd), což nejspíš způsobuje bug v samotném LLVM.

- LLVM - clang O3: testovaný případ se od výše uvedeného liší prováděním mnoha optimalizací, které jsou vynuceny přepínačem "O3" ve frontendu.

- Lua - interpretr: zdrojový kód algoritmů ve skriptovacím jazyku Lua byl přeložen a spuštěn příkazem "lua $sourceFile". Překlad bohužel nešel rozdělit na dvě fáze jako v předešlých příkladech. Do výsledného času měření je tím pádem započítán samotný překlad do vnitřní reprezentace.

- LuaJIT: zdrojový kód algoritmů ve skriptovacím jazyku Lua byl přeložen příkazem "luajit -b $sourceFile $bytecodeFile", čímž vznikl soubor s vnitřní reprezentací. Samotné spuštění se provedlo příkazem " luajit -jon $bytecodeFile".

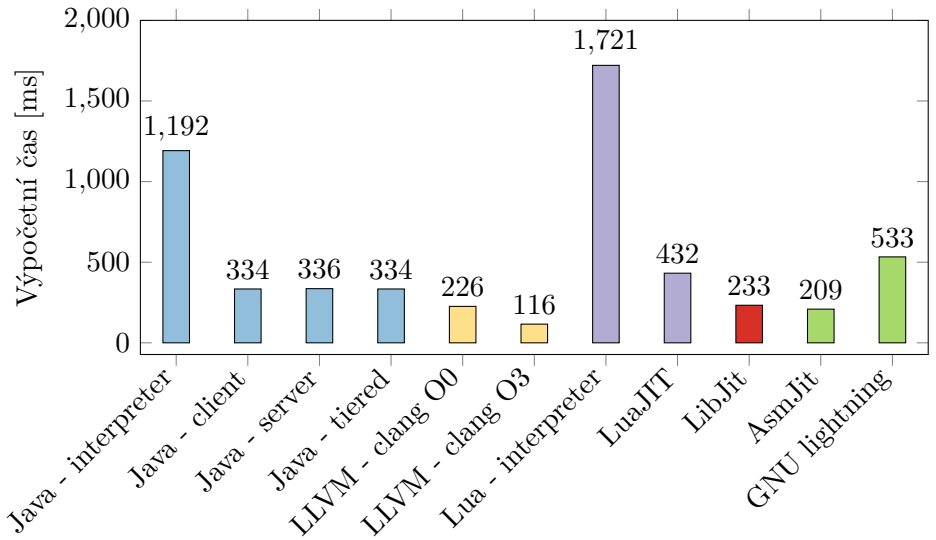
- LibJit: zdrojový kód algoritmů napsaných pomocí speciálního LibJit API v jazyku C++ byl přeložen pomocí kompilátoru "g++-4.9" do spustitelného binárního souboru.

- GNU lightning: způsob kompilace zdrojových souborů se téměř nelišil od LibJitu. Jejich těla však obsahují kód napsaný pomocí GNU lightning API.

- AsmJit: způsob kompilace se opět zásadně neliší. Zdrojové soubory tvoří kód napsaný za pomocí ASMJit API, které nabízí koncept s názvem "Compiler".

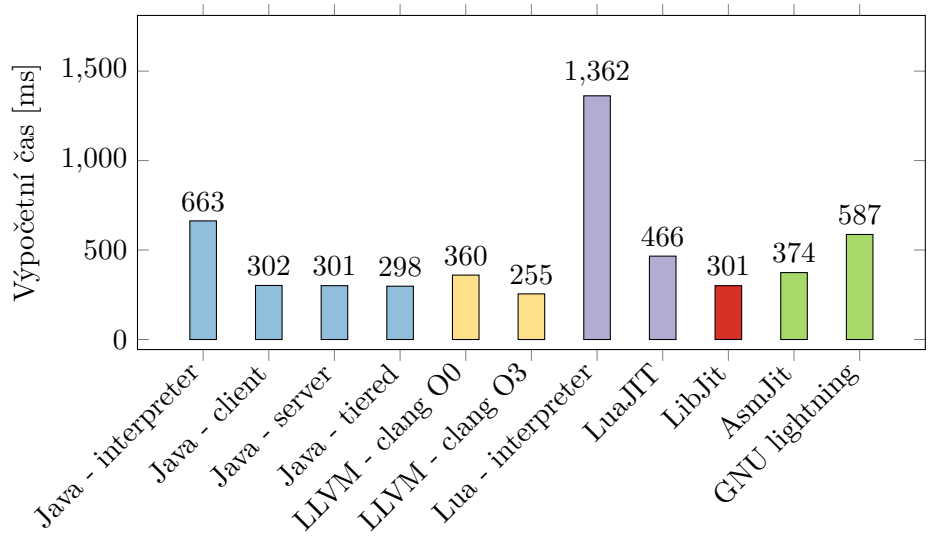
- Parrot: kód napsaný v jazyku symbolických adres s názvem PIR je nejprve přeložen do bytekódu s příponou pbc. Příkaz "parrot -G -R fast" provede spuštění bez zapnutého Garbage Collectoru (přepínač G) a s nejrychlejším jádrem (přepínač -R fast).

4.1.5 Algoritmus - Rychlost aritmetických operací



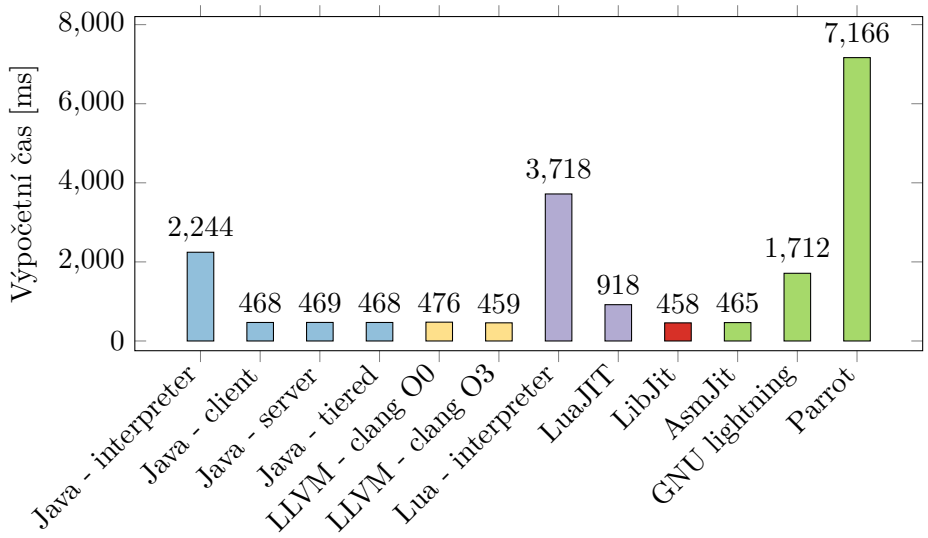
Obrázek 4.1: Porovnání JIT kompilátorů podle rychlosti výpočtu algoritmu Arithmetic.xxx z přiloženého CD

4.1.6 Algoritmus - Gcd



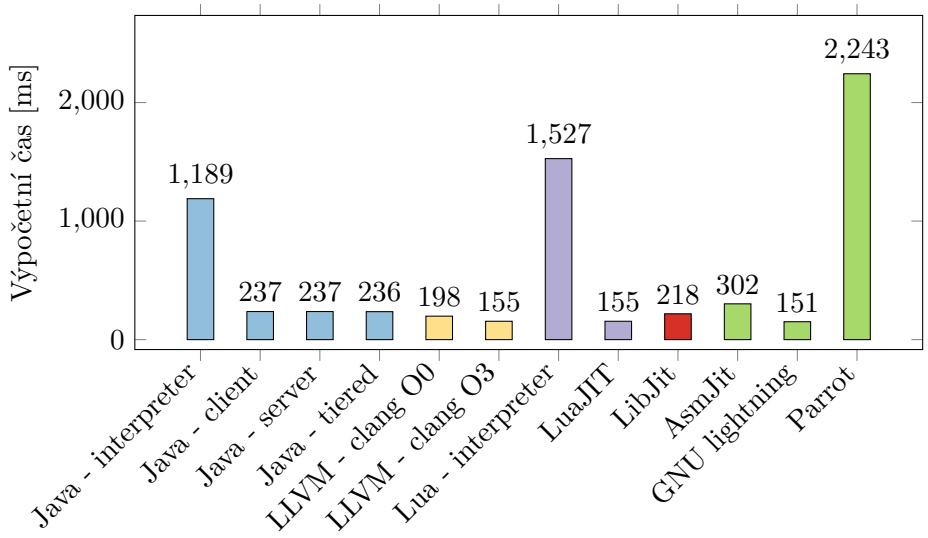
Obrázek 4.2: Porovnání JIT kompilátorů podle rychlosti výpočtu algoritmu Gcd.xxx z přiloženého CD

4.1.7 Algoritmus - Výpočet počtu prvočísel



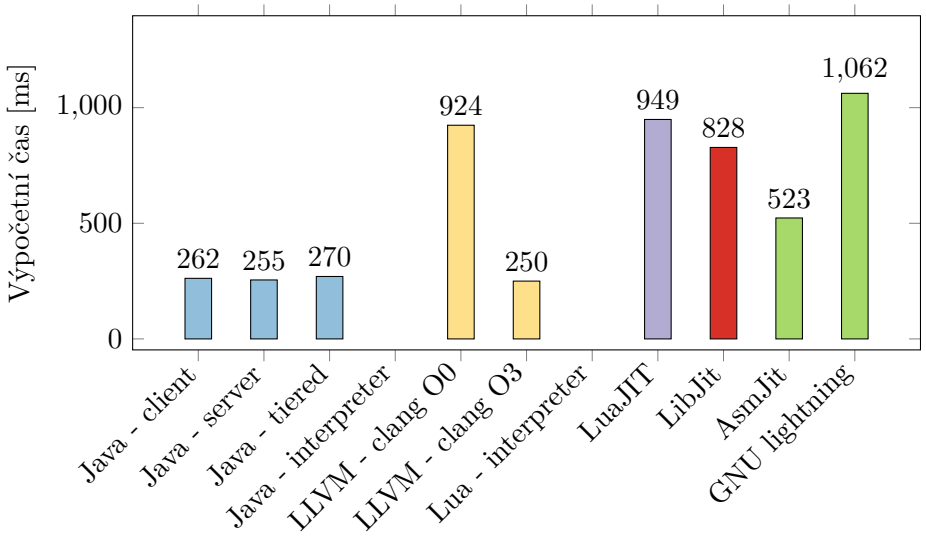
Obrázek 4.3: Porovnání JIT kompilátorů podle rychlosti výpočtu algoritmu PrimeNumbersCounter.xxx z přiloženého CD

4.1.8 Algoritmus - Výpočet čísla Pí



Obrázek 4.4: Porovnání JIT kompilátorů podle rychlosti výpočtu algoritmu Pi.xxx z přiloženého CD

4.1.9 Algoritmus - Gaussova eliminační metoda



Obrázek 4.5: Porovnání JIT kompilátorů podle rychlosti výpočtu algoritmu GaussElimination.xxx z přiloženého CD. Interpretery jazyků Lua a Java jsou v grafu vynechány z důvodu vysokých hodnot. Lua - interpreter = 50920 ms, Java - interpreter = 10401 ms

4.1.10 Diskuse naměřeným výsledků

Hned při prvním zběžném pohledu na naměřené výsledky jedna zásadní informace vyplyne na povrch. JIT kompilace několikanásobně zrychluje provádění algoritmů. Nejmarkantněji se zrychlení projevuje v případě Gaussovy eliminační metody, kde nebylo ani možné uvést výsledky pro interpretry jazyka Lua a Javy, jelikož by se ostatní hodnoty v grafu upozadily. V případě jazyka Lua došlo dokonce k padesátinásobnému zrychlení. Java sice takového výsledku nedosahuje, ale čtyřicetinásobné zrychlení je více než konkurenceschopné. Navíc je interpret jazyka Lua v případě zmíněného algoritmu pětkrát pomalejší než interpret Javy. Všeobecně si ze všech interpretrů nejhůře vedl Parrot, který je však zaměřen na dynamicky typované jazyky. Na druhou stranu byl kód psán přímo pomocí PIR (jazyku symbolických adres), takže by dělení "statický / dynamický" jazyk nemělo hrát roli.

Druhý rychlý pohled prozradí, že u takto koncipovaných algoritmů je jedno, jaký se použije JIT kompilátor Javy. Všechny vycházejí rychlostně téměř nastejno. Je velice pravděpodobné, že je to způsobeno častým opakováním stejného kódu, kdy se rozdíly mohou stírat. Větší rozptyl hodnot by se mohl objevit při kratším běhu algoritmů bez tolika opakování. JIT kompilátory Javy se nejvíce vyznamenaly v případě Gaussovy eliminační metody,

kde spolu s LLVM vyhrály. V mnoha případech byly často poraženi jinými kompilátory, což může být časem potřebným pro překlad do nativního kódu.

Další nevyvratitelná skutečnost se týká použití různých optimalizačních technik u frontendu generujícího LLVM IR. V případě použití přepínače O3 vždy dojde ke zrychlení běhu programu. Nejméně se zrychlení projevilo u algoritmu vypočítávajícího počet prvočísel. Naopak nejvíce u Gaussovy eliminační metody. Což může být způsobeno tím, že se v prvním uvedeném případě používá více skoků a podmínek než výpočtů.

GNU lightning se jeví jako nejhorší ze všech JIT kompilátorů, což je nejspíše způsobeno jeho všeobecným zaměřením a malým počtem registrů. Výjimka potvrzuje pravidlo, protože v případě výpočtu čísla Pí naopak všechny ostatní porazil. LuaJIT jednoznačně několikanásobně zrychluje provádění algoritmů napsaných ve skriptovacím jazyce Lua. Po GNU lightning je ale druhým nejhorším, což ale opět neplatí u výpočtu čísla Pí. Je ale pravda, že se v algoritmu objevují složitější aritmetické výpočty, takže může být naopak dobrý v jejich zjednodušování. Velkým překvapením je však ASMJit od českého vývojáře, který si nevede vůbec špatně v porovnání s ostatními. Nejspíše vděčí svému zaměření na architekturu X64.

4.2 Výběr JIT kompilátoru

V diskusi naměřených výsledků z konce minulé kapitoly zcela chybí zmínka o LibJitu. Nestalo se tak náhodou. Jeho správný čas přijde až v této kapitole. Na základě měření lze bez rozmyslu ihned vyřadit GNU lightning a LuaJit. GNU lightning rychlostně neoslnil a LuaJit nepřináší v porovnaní s ostatními takové zrychlení, aby se vyplatilo ručně generovat jeho bytecode. Závěrečné pořadí bylo zvoleno následovně:

1. LibJit - své první místo uhájil i přes to, že byl v měření několikrát přeskočen svými nejbližšími konkurenty. Nejlépe si vedl ve výpočtu počtu prvočísel. V případě výpočtu čísla Pí a Gcd se držel někde uprostřed. Nikdy vyloženě nezklamal. Nebýt výhod uvedených v analýze, ocitl by se na druhém místě.

2. LLVM - jeho optimalizovaná verze držela krok s JITem Javy. Ve výpočtu aritmetických operací byl dokonce třikrát rychlejší. Jeho úctyhodné hodnoty mohou být však způsobeny provedenými agresivními optimalizacemi v čase kompilace, která může zapříčinit velký počáteční "overhead". Proto se realitě bude více přibližovat průměrný čas neoptimalizované a optimalizované verze. I přes takovou korekci by se jednalo o předního kandidáta na integraci. Svou pověst si LLVM trochu pošramotil šedesátiminutovou kompilací svých zdrojových souborů a zmíněným bugem, který si vynutil použití přepínače -fno-use-cxa-atexit v clangu. Malost,

univerzálnost, hezké API a dostatečná rychlost LibJitu ho jen těsně posunula na první místo před LLVM.

3. JitAsm - je asi největším překvapením testování. Nikde vyloženě nepropadl a dvakrát dokonce přeskočil LibJit. Ve spojení s poznatky z analýzy mu jasně patří třetí místo.

4. Java - celou dobu si JIT kompilátor udržel svůj vysoký standard. Kvůli složitější integraci v podobě generování "class souborů" a z důvodu možné vyšší počáteční režie se nachází až na čtvrtém místě.

4.3 Návrh integrace

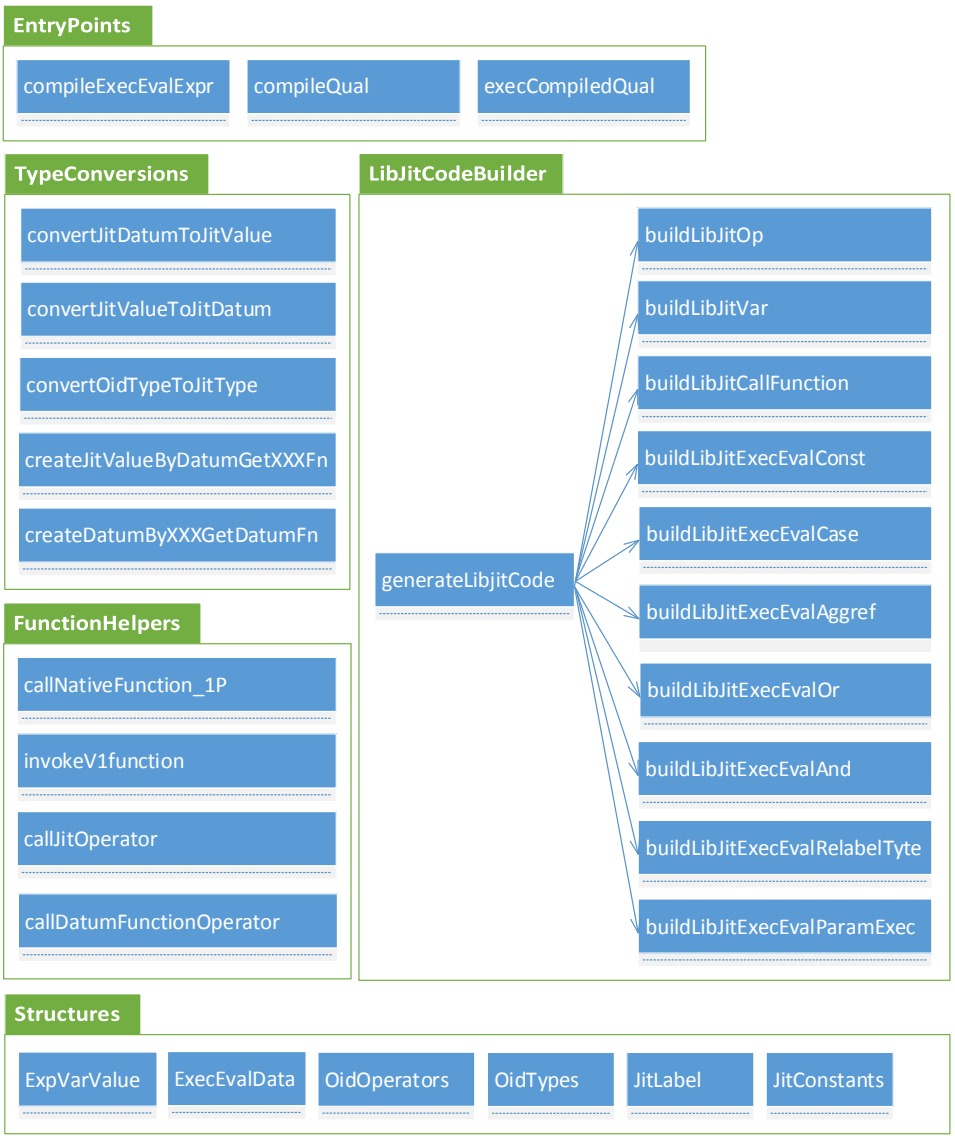
Návrh integrace bude volně navazovat na popis expression exekutoru v podkapitole 2.3 s ohledem na vybraný JIT kompilátor. Připomeňme si jednu důležitou větu z analýzy: *"ExprStateEvalFunc evalfunc - jedná se o ukazatel na funkci, která má na starosti provedení daného uzlu v expression výrazu. Zatímco SQL exekutor obsahuje velký switch, xpression exekutor naopak využívá metodu založenou na volání funkcí."*. V dřívějších verzích se v PostgreSQL také pro expression exekutor používal velkých switch, ale v novějších vydáních byl nahrazen zmíněným voláním funkcí. Pro jednodušší kompilaci se vrátíme ke kořenům. Základ překladače bude tvořit velký switch, který bude dle typu uzlu volat funkce generující instrukce LibJitu.

Pojďme si nejprve z analýzy 2.3.5 zopakovat , jaké parametry se předávají expression exekutoru:

- Kořen stromu celého expression výrazu.
- Kontext, který si nese informace o jednotlivých "tuples".
- Indikátor udávající, zda výsledek nabývá hodnoty null.
- Indikátor itemIsDone, který se uvnitř exekutoru nastaví na jednu z mnoha konstant podle výsledku.

Hned na začátku je nutné si rozmyslet, které výše zmíněné parametry jsou potřeba už v čase kompilace, a které až při spuštění. Bez velkého zaváhání lze dojít k závěru, že při kompilování jsou nepostradatelné uzly expression výrazu a kontext. Při spuštění se funkci předají poslední dva parametry a pro jistotu ještě samotný kontext.

Nyní nastává správná chvíle pro popis obrázku 4.6. Navrženou strukturu lze rozdělit do čtyř základních bloků, které popíší následující podkapitoly.



Obrázek 4.6: Blokové schéma základních stavebních prvků překladače

4.3.1 LibJitCodeBuilder

LibJitCodeBuilder tvoří jádro celého překladače. Ve velkém ”switchi” uvnitř metody generateLibJitCode se během kompilace postupně projde celý strom expression výrazu. Na základě typu uzlu je sám uzel spolu s kontextem a vstupními parametry (isNull, itemIsDone) předán konkrétní metodě mající na starosti generování kódu pro příslušný uzel. Naprostá většina z nich může opět zavolat generateLibJitCode, protože potřebují zpracovat nelistové uzly.

BuildLibJitOp - Metoda zcela nahrazuje původní ExecEvalOper určenou pro vyhodnocování operací (=, +, -, /, atd). Vše podstatné bylo o její původní verzi napsáno v kapitole 2.3.6. Nová verze je generována za pomocí LibJit API. Zásadní změna spočívá ve vynechání volání obslužné funkce operátoru. Místo toho bude matematická operace vykonána funkcí LibJitu. Například jit_insn_add slouží pro sečtení dvou hodnot. Díky tomu zcela zmizí jedno volání vestavěné funkce a příprava struktury FunctionCallInfoData. Původně se parametry připravovaly v cyklu, protože šlo o volání obecné funkce, které mohou mít různé počty parametrů. Nová funkce může vždy počítat se dvěma parametry a nepotřebuje žádný cyklus. Pokud operátor nebude pocházet z pevně dané podmnožiny uložené ve speciální struktuře OidOperators, tak se pro vyhodnocení použije funkce ze struktury OpExpr. Jako příklad může sloužit část dotazu dotazu "where jmeno = 'jarda"', kde oba operandy reprezentují typ s názvem text.

BuildLibJitVar - Nahrazuje původní ExecEvalScalarVar a ExecEvalScalarVarFast pro načítání hodnot z konkrétního "tuple". V metodě je třeba řešit volání nativních funkcí a práci s pamětí. Nejdříve je potřeba ze struktury Var 2.9 vydolovat číslo sloupce, z kontextu získat správný tuple (INNER_VAR, OUTER_VAR, či INDEX_VAR) a pomocí nativní funkce slot_getattr získat hodnotu atributu.

BuildLibJitCallFunction - Metoda nahrazuje původní ExecEvalFunc. Připomeňme si tvrzení z kapitoly 2.3.6, které říká, že nahrazovaná metoda je téměř identická s ExecEvalOper a volá se pouze jednou. Operátor (uzel stromu) totiž poté obsluhuje nativní funkce "ExecMakeFunctionNoSets" (nahradí se ukazatel ExprStateEvalFunc evalfunc v ExprState struktuře). Díky tomu, že neočekáváme na výstupu množinu, mohli jsme a můžeme také pro buildLibJitCallFunction tento mezikrok vynechat. Dostáváme se opět k samotnému volání obslužné (může být uživatelská) funkce. Zde už nelze počítat se stejným počet parametrů v každé funkci. Smyčka zajišťující vyhodnocování všech potomků uzlu musí být zachována. Můžeme ji však rozvinout, což by o něco mohlo zlepšit pipelining. Uzel stromu FuncExprState již obsahuje předpřipravenou strukturu FunctionCallInfoData, která musí být z podvýrazů naplněna daty. Poté by již teoreticky nemělo nic bránit zavolání nativní funkce pomocí makra FunctionCallInvoke(fcInfo) (viz. kapitola 1.4). Opak je pravdou. V LibJitu nemůžeme používat makra, protože se nejedná o funkce a nelze na ně tedy získat ukazatel. Proto budou vytvořeny speciální volací funkce, které blíže přibližuje podkapitola 4.3.4. Pro přípravu struktury FunctionCallInfoData existuje též makro. Ta je v uzlu naštěstí již připravena z předchozí fáze inicializace (viz. kapitola 2.3.3), takže není potřeba dále nic řešit. Na závěr je nutné ve funkci buildLibJitCallFunction zabezpečit nastavení indikátoru "null" na správnou hodnotu.

BuildLibJitExecEvalConst - Funkce má na starosti získání konstanty. Zde se nebude dít žádná velká magie. Pouze dojde k naplnění indikátorů a k navrácení vyextrahované konstanty z uzlu.

BuildLibJitExecEvalCase - Metoda zcela nahrazuje původní ExecEvalCase. Uvnitř opět dojde k rozvinutí smyček, které vyhodnocují podvýrazy v CASE a WHEN blocích. Vrátí se hned první hodnota, jejíž podmínka byla vyhodnocena jako true. Důležité je dodat, že NULL se nepovažuje za true. V případě neúspěchu všech podmínek dojde k navrácení hodnoty z bloku ELSE. Pokud ani ten není definován, indikátor null se nastaví na true a vrátí se (Datum)0.

BuildLibJitExecEvalAggref - Nahrazuje původní ExecEvalAggref. Z informací uložených v uzlu (AggrefExprState) se zjistí index do pole výsledků agregačních funkcí. Kontext, který se předává překladači neobsahuje jen informace o "tuples", ale také zmíněné pole (econtext->ecxt_aggvalues). Navíc uchovává informace, jež indikují, zda jsou výsledky null. Na závěr se nastaví indikátory a vrátí se jeden konkrétní prvek z pole.

BuildLibJitExecEvalOr - Nahrazuje původní ExecEvalOr. Vnitřní cyklus vyhodnocující podvýrazy bude opět rozvinut. Narazí-li se na první hodnotu true, bude neprodleně vráceno true. Pokud by byla celá klausule false a některá hodnota null, vrátí se null. V opačném případě bude výsledek false.

BuildLibJitExecEvalAnd - Nahrazuje původní ExecEvalAnd. Vnitřní cyklus vyhodnocující podvýrazy bude taktéž rozvinut. Narazí-li se na první hodnotu false, bude neprodleně vráceno false. Pokud by nevyšla žádná hodnota negativně a některá z nich byla null, vrátilo by se null. V opačném případě bude výsledek true.

BuildLibJitExecEvalNot - Metoda nahrazuje původní ExecEvalNot. Uvnitř se pouze vyhodnotí potomek uzlu, jehož návratová hodnota má být znegována. Menší nepříjemnost je zapříčiněna absencí negace v LibJitu. Ten přímo podporuje jen její bitovou verzi. Proto je potřeba ji doimplementovat.

BuildLibJitExecEvalRelabelType - Zcela nahrazuje původní ExecEvalRelabelType. Její původní tělo tvoří jen jeden řádek, který z GenericExprState vyextrahuje atribut arg. Tohoto potomka předá volanému makru ExecEvalExpr, které se postará o rekurzivní zpracování. Nová verze je vygenerována pomocí LibJit API. Uzel definuje tag s názvem T_RelabelType. Určitě se objevuje ve výrazech, které obsahují textové operandy.

BuildLibJitExecEvalParamExec - Standardní metoda ExecEvalParamExec se zcela jistě volá v případech, kdy se v dotazu používá vnořený SELECT vracející jednu skalární hodnotu. Jde o metodu poměrně složitou. Proto bylo přistoupeno k řešení, že nová generující funkce BuildLibJitExecEvalParamExec pouze pomocí LibJit API volá její starou verzi. Díky tomu může být například druhý podstrom zpracováván dále pomocí LibJitu a dále generován.

4.3.2 TypeConversions

Nativní a uživatelské metody vracejí typ Datum, který bude ukládán jako každá jiná hodnota v proměnné jit_value_t. Jit_type_t 3.8.1 bude přesně kopírovat skutečný typ Datum, který na platformě x64 odpovídá velikosti ulong (tedy jit_type_ulong). Funkce LibJitu ale potřebují pracovat se skutečnými číselnými typy. Z kapitoly 1.3 víme, že je nutné mít na paměti, jakou hodnotu zrovna Datum reprezentuje. V kódu jsme naštěstí většinou schopní zjistit Oid typu (viz kapitola 2.3.2), které odpovídá záznamu v katalogu 1.1.1. Získané informace nám již postačují k přetypování. Stejným způsobem bychom mohli chtít skutečný typ převést zpět na Datum, protože ho potřebujeme vložit do volané uživatelské funkce. Přesně k těmto účelům budou sloužit funkce uvedené v bloku "TypeConversions". Služeb maker pro přetypování využít nemůžeme, protože na ně nezískáme odkaz. Pro přetypování "float" hodnot již existují skutečné funkce, které lze z LuaJitu zavolat. V takovém případě nám budou nápomocné funkce createDatumByXXXGetDatum a createJitValueByDatumGetXXXFn, které obdrží potřebné ukazatele a postarají se o jejich zavolání. Trojce písmen XXX substituuje hodnoty jako FLOAT4, FLOAT8.

4.3.3 Structures

OidTypes - Jak již bylo naznačeno v kapitole 4.3.2 výše, v kódu je často potřeba pracovat s Oid jednotlivých typů. Neprofesionální by bylo takové hodnoty ukládat jako "integer". Proto vznikl výčtovým typ s názvem OidTypes, který v sobě zahrnuje hodnoty jako OID_BOOL(16), OID_INT8(20), atd.

OidOperators - Katalog 1.1.1 operátorů obsahuje množinu záznamů. Například jen pro sčítání hodnot existuje několik kombinací pro různé typy levých a pravých operandů. Vykonání každého operátoru má na starosti jiná funkce v databázi PostgreSQL. Pro reprezentaci konkrétního operátoru byl vytvořen výčtový typ s názvem OidOperators obsahující (OID_INT_4_ADD (551), OID_FLOAT_4_FLOAT8_DIV(1118), atd).

JitLabel - V kódu vznikají různá návěští, na která je možné skočit z jiných částí. Uvnitř struktury JitLabel se nachází jit_label_t 3.8.1 reprezentující návěští a index do pole všech "labels". Samotné pole je zabaleno ve struktuře s názvem jit_Labels, která si drží informace o ukazateli na první prvek, o délce

pole a o indexu posledního prvku. Pro tvorbu a získání potřebných návěští byly vytvořeny speciální metody, které pracují se zmíněným polem.

ExpVarValue - Metody uvnitř LibJitCodeBuilderu potřebují vracet výsledky z podvýrazů metodám, které je zavolaly. Přesně k tomuto účelu je určena struktura ExprVarValue, která uchovává potřebné informace o vrácené skalární hodnotě. Vnitřní uspořádání bylo částečně ovlivněno návrhem typu Node 1.5. Na prvním místě se nachází tag blíže určující typ struktury. Jiné metody by teoreticky nemusely vracet ExprVarValue, ale jinou strukturu s jiným tagem. Druhé místo je zarezervováno pro prvek typu OidTypes, který určuje Oid návratové hodnoty. Třetí booleovská hodnota říká, zda struktura aktuálně reprezentuje typ Datum. Poslední nejdůležitější prvek typu jit_value_t uchovává vlastní hodnotu (buď Datum nebo číselný typ, případně ukazatel).

JitConstants - V programu se vyskytuje určité množství opakujících se konstant, které by bylo třeba stále znovu vytvářet příkazem podobnému tomuto "jit_value_create_nint_constant". Proto je zavedena struktura JitConstants, která ve svých útrobách uchovává základní konstanty různých typů (0,1 - jit_type_int, 0,1 - jit_type_ubyte, 0xFF - jit_type_ulong, atd).

ExecEvalData - Struktura slouží jako jakási přepravka potřebných informací, které ke své činnosti potřebují jednotlivé metody LibJitCodeBuilderu. Ve svých útrobách ukrývá typ 3.8.1 jit_function_t, který reprezentuje generovanou funkci pro vyhodnocení celého expression stromu. Dále obsahuje výše zmíněné pole všech použitých návěští. V neposlední řadě obsahuje struktury JitConstants a ExprContext. Poslední zmíněnou uchovává rovnou ve dvou formách (jit_value_t pro runtime a ExprContext pro čas kompilace).

4.3.4 FunctionHelpers

CallNativeFunction_1P - Již sám název pomocné funkce napovídá, že bude volat nativní funkce s jedním parametrem. Převážně se bude používat pro převod hodnot typu FLOAT4, FLOAT8 na Datum a zpět. Jak již bylo napsáno výše 4.3.2, PostgreSQL takové funkce implementuje a nepoužívají se pro ně makra. Před převodem je třeba znát ukazatel na funkci (jit_value_t), předávanou hodnotu(jit_value_t), typ návratové hodnoty(jit_type_t). Původní nahrazované funkce přijímají převážně čtyři parametry. Speciálně pro ně bude existovat další metoda stejného názvu s koncovou _4P (čtyři parametry, jedna návratová hodnota Datum).

InvokeV1function - Tato funkce bude naopak pomáhat při volání funkcí, které jsou založené na V1 volací konvenci (viz. kapitola 1.4). Jako parametr ji tedy stačí předat strukturu FunctionCallInfo. Z deskriptoru volané funkce

se zjistí její adresa. Velká výhoda spočívá ve faktu, že návratovou hodnotou je vždy Datum (jit_value_t typu jit_type_ulong). Na závěr stačí zavolat již zavedenou funkci CallNativeFunction_1P a předat jí vyextrahovanou adresu, ukazatel na strukturu FunctionCallInfo a návratový typ jit_type_ulong.

CallJitOperator - Funkce zajistí provedení matematické operace (+, - , <, =, atd.) pomocí LibJit API Na vstupu očekává strukturu OpExpr pro identifikaci operátoru, levý operand (ExpValValue) a pravý operand (ExpVarValue). Konkrétní číselná hodnota operandu uložená v proměnné typu jit_value_t se ze struktury získá pomocí funkce getJitValueFromExprValue. Více o této metodě je napsáno v následující podkapitole 4.4.5.

CallDatumFunctionOperator - Pokud se na vstupu CallJitOperator objeví operátor či operand, který nedokáže zpracovat, tak se automaticky zavolá funkce s názvem CallDatumFunctionOperator. Ta se postará o zavolání funkce specifikované v OpExpr struktuře zodpovědné za vykonání daného operátoru. Uvnitř generující funkce se připraví oba operandy (včetně ověření, zda nejsou NULL) a předají se známé struktuře FunctionCallInfo. Výsledek operace se na konci získá zavoláním InvokeV1function.

4.4 Realizace

4.4.1 Klauzule WHERE

Klauzule WHERE určuje, které řádky jsou vráceny příkazem SELECT, a které jsou ovlivněny příkazy UPDATE a DELETE. V průběhu realizace bylo zjištěno, že je zapotřebí doimplementovat část, která zpracuje celou klauzuli. Doposud jsme se převážně věnovali zpracování složitějších výrazů vyskytujících se v příkazu SELECT před klíčovým slovem FROM.

Tělo klauzule je tvořeno seznamem s názvem Qual, kterému patří jeden atribut ve struktuře PlanState. Ostatně onen seznam byl již zmíněn v kapitole 2.3.3 v rámci popisu uvedené struktury. PostgreSQL implementuje metodu s názvem ExecQual, která postupně prochází prvky seznamu a předává je expression exekutoru. Cyklus je ukončen v okamžiku, kdy první vyhodnocený výraz nabyl hodnoty false. V takovém případě totiž není splněna celá klauzule WHERE. Funkce ExecQual navíc přijímá parametr, který říká, zda se má hodnota NULL vyhodnocovat jako false.

Implementovaná JIT kompilace kopíruje chování metody ExecQual. Pro každou klauzuli WHERE je pomocí LibJit API vygenerován kód, který postupně prochází jednotlivé prvky seznamu. Tak jak tomu bylo v těle některých metod překladače, tak i zde opět došlo k rozbalení smyčky. Kořen stromu výrazu jednoho prvku seznamu je vždy předán již uvedené ústřední funkci kompilátoru s názvem generateLibjitCode (viz. obrázek 4.6 a následující podkapitola 4.4.3). Není-li funkce schopna vygenerovat kód pro konkrétní výraz, dojde k zavolání nativní metody mající na starosti zpracování daného uzlu a případných potomků. Vyhodnocování výsledků probíhá stejně jako v metodě ExecQual.

Předchozí odstavec popisoval způsob generování nativního kódu pro klauzuli WHERE. Zatím ale nepadla ani jedna zmínka o tom, kde by měl být zkompilovaný kód uložen. Jako ideální se jeví struktura PlanState, do které byl přidán ke stávajícímu ”qual” atributu nový ”jit_function_t compiledQual” uchovávající zkompilovaný kód vygenerované funkce. Při vyhodnocování klauzule se již nevyužívá původní metody ExecQual, ale nově zkompilované funkce.

4.4.2 Vstupní body

Vstupním bodem je myšlena ta část kódu, kde databáze PostgreSQL přijde do kontaktu s naším překladačem a předá mu vstupní parametry. Ty se uvnitř nacházejí přesně čtyři:

1. Kompilace (SELECT) - bod se nachází ve funkci ExecBuildProjectInfo známé z kapitoly 2.3.4, kde bylo její tělo detailněji rozebráno. Již ve zmíněné kapitole bylo naznačeno, že by zkompilovaná funkce mohla být přidána do struktury GenericExprState, která obaluje kořen stromu expression výrazu. Její obsah po provedení úspěšné modifikace si lze spolu

se samotnou kompilací prohlédnout ve výpisu kódu 4.1. Kompilace se v cyklu funkce ExecBuildProjectInfo provede pro každý netriviální atribut vyjádřený pomocí složitějšího expression výrazu. Pokud se v jejím průběhu narazí na uzel, který překladač nezná, vrátí hodnotu null.

2. Spuštění (SELECT) - bod je umístěn v metodě ExecTargetList uvedené v kapitole 2.3.5. Vše podstatné si lze prohlédnout ve výpisu 4.2. V něm se postupně prochází seznam se všemi složitějšími expression výrazy. Z každé položky získáme dobře známou strukturu GenericExprState, která obsahuje ukazatel na zkompilovanou funkci. Pokud není nulový, tak se připraví potřebné parametry a pomocí funkce jit_function_apply dojde k jejímu zavolání. Výsledek ve formátu Datum se ukládá na jedno místo v poli (projectInfo->slot->tts_values), které dobře známe z kapitoly 2.3.5. Pokud by byl odkaz na zkompilovanou funkci null, spustilo by se standardní provádění stromu expression výrazu pomocí makra ExecEvalExpr.

3. Kompilace (klauzule WHERE) - vše podstatné o JIT kompilaci klauzule WHERE bylo napsáno v podkapitole 4.4.1. Všechny potřebné informace pro překlad jsou dostupné po každé inicializaci struktury PlanState, která se odehrává rekurzivně ve funkci ExecInitPlan (viz. kapitola 2.3.3). Ke kompilaci dojde vždy na konci těsně před vrácením výsledku. Ve skutečnosti tedy neexistují přesně čtyři vstupní body, protože potomci uzlů v prováděcím plánu mohou také iterovat nad svým qual atributem (vnořený SELECT, JOIN atd).

4. Spuštění (klauzule WHERE) - v kapitole 2.3.5 bylo uvedeno, že metoda ExecProject je zodpovědná za provedení projekce. Před tím, než dojde k jejímu zrealizování, je nutné vědět, zda má být řádek vůbec zobrazen mezi výsledky. Tím se dostáváme zpět k původní metodě ExecQual, jejíž výstup určoval, zda se má provést projekce nad daným řádkem a zda má být vrácen. Qual atribut se však netýká jen klauzule WHERE, ale hraje také například svou roli při vyhodnocování výrazu při spojování tabulek. Původní funkce ExecQual byla nahrazena funkcí execCompiledQual, která přijímá jediný parametr (ukazatel na strukturu PlanState). Uvnitř se pomocí zkompilované funkce provede vyhodnocení a vrátí se výsledek. Pokud by náhodou zkompilovaná funkce nebyla k dispozici, využije se služeb původní nahrazované metody. Na základě této úpravy musely být všechny výskyty funkce ExecQual v adresáři executor kódu PostgreSQL nahrazeny (v případě, že to bylo možné) novou verzí se jménem execCompiledQual. Výměna se převážně týkala těl funkcí, které zpracovávají konkrétní uzel prováděcího plánu.

Výpis 4.1: Struktura GenericExprState, která obsahuje kořenový element "ExprState *arg" stromu expression výrazu; Dále je uvedena kompilace funkce a její uložení do instance zmíněné struktury.

```
typedef struct GenericExprState {
 ExprState xprstate;
 ExprState *arg; // stav kořenu expression výrazu
 Jit_function_t jit_expr_function;
} GenericExprState;

// kompilace a uložení vygenerovaného kódu funkce
gstate->jit_expr_function = compileJitEvalExpr(
        gstate->arg, projInfo->pi_exprContext);
```

Výpis 4.2: Spuštění zkompilované funkce v metodě ExecTargetList

```
....
foreach(tl, targetlist){
 GenericExprState *gstate=(GenericExprState*)lfirst(tl);
 TargetEntry *tle = (TargetEntry*)gstate->xprstate.expr;
 AttrNumber resind = tle->resno - 1;
 jit_function_t jit_fn = gstate->jit_expr_function;
 if(jit_fn != NULL){
....
  void* args[3] = {&jit_econtext, &jit_isNull,
                   &jit_itemIsDone};
  Datum result;
  jit_function_apply(jit_fn, args, &result);
  values[resind] = result;
 }
....
}
```

4.4.3 Jádro překladače

Již v návrhu 4.3 bylo uvedeno, že stěžejní část překladače je tvořena velkým switchem, který volá metody generující kód pro daný uzel. Využívá se typů, které již zavedl PostgreSQL a je možné je přečíst z každého uzlu. Jedná se vlastně o jediný atribut, který se nachází v čisté struktuře Expr (viz. kapitola 2.3.2). Výpis 4.3 ukazuje určitý výřez, který ale znázorňuje vše podstatné. Konkrétně je možné si prohlédnout volání obslužných metod pro matematický operátor, načtení hodnoty z databáze (T_VAR) a booleovský operátor. Například právě u booleovského operátoru je potřeba provést další dělení pomocí switche, abychom zjistiti jeho konkrétní druh (and, or, not). Všechny ostatní

struktury, které se v metodě generateLibjitCode používají, již byly také zmíněny v návrhu 4.3.

Výpis 4.3: Velký switch, který tvoří jádro překladače

```
ExpVarValue* generateLibjitCode(ExprState *exprState,
jit_ExecEvalData* jitExecEvalData,
jit_value_t jit_resultIsNull,
jit_value_t jit_isDone ){
 switch(exprState->expr->type){
  case T_OpExpr:
   return buildLibjitOp(exprState, jitExecEvalData,
                jit_resultIsNull, jit_isDone );
   break;
  case T_Var: return buildLibjitVar(exprState,
        jitExecEvalData, jit_resultIsNull);
   break;
....
  case T_BoolExpr:{
   BoolExpr* boolExpr = (BoolExpr*)exprState->expr;
   switch(boolExpr->boolop){
    case OR_EXPR:
     return buildLibJitExecEvalOr(exprState,
        jitExecEvalData, jit_resultIsNull, jit_isDone);
     break;
....
   }
  }
  break;
  default:
   return NULL;
  break;
 }
```

4.4.4 Výčtový typ OidOperators

Pojďme si v této části ukázat, jak je možné pro příslušné operátory získat relevantní data ze systémového katalogu pg_operator (viz. kapitola 1.1.1). Nejprve je třeba vytvořit dotaz, který vypadá následovně:

- select (select typname from pg_type where pg_type.oid = oprleft) as left , (select typname from pg_type where pg_type.oid = oprright) as right, oprname, oprkind, oid from pg_operator;

V dotazu se využívá systémového katalogu pg_type 1.1.1, protože potřebujeme získat jména levých a pravých operandů. Tabulka 4.2 ukazuje prvních šest výsledků vyhledávání.

left	right	oprname	oprkind	oid
int4	int8	=	b	15
int4	int8	<>	b	36
int4	int8	<	b	37
int4	int8	>	b	76
int4	int8	<=	b	80
int4	int8	>=	b	82

Tabulka 4.2: Výsledek databázového dotazu do systémového katalogu pg_operator

Z výsledků v tabulce 4.2 lze snadno usoudit, že PostgreSQL používá pro zpracování různých typů operandů, různé funkce. Po provedení dalšího jednoduchého dotazu do katalogu pg_proc lze nalézt funkce s názvy int48eq, int48ne, int48lt, int48gt, int48le, int48ge, které mají na starosti provedení oněch operátorů v tabulce.

V našem překladači potřebujeme určitou podmnožinu operátorů identifikovat a jejich vyhodnocení nechat na funkci v LibJitu. Vše přibližuje výpis 4.4. Ve výčtovém typu OidOperators jsou uložena všechna Oid požadovaných operátorů. Toho využívá funkce callJitOperator, která na základě jeho hodnoty vybere správnou funkci pro vyhodnocení.

Výpis 4.4: Ukázka výčtového typu OidOperators a jeho využití při vyhodnocování matematického operátoru

```
typedef enum OidOperators{
 OID_INT4_ADD = 551,
 OID_INT4_SUB = 555,
....
 OID_INT4_NE = 518
....
}

// funkce callJitOperator
ExpVarValue* callJitOperator(
 jit_ExecEvalData* jitExecEvalData,
 jit_value_t jit_isDone, OpExpr* opExpr,
 ExpVarValue* leftOperand, ExpVarValue* rightOperand){
  jit_value_t temp;
  switch(oidOperator){
   case OID_FLOAT4_ADD:
```

```
....
case OID_FLOAT8_FLOAT4_ADD:
case OID_INT4_ADD:
case OID_INT8_ADD:
 temp = jit_insn_add(jitExecEvalData->jit_fn,
 getJitValueFromExprVarValue(jitExecEvalData,
 leftOperand),
 getJitValueFromExprVarValue(jitExecEvalData,
 rightOperand));
break;
....
```

4.4.5 Manipulace se strukturou ExprVarValue

Několik řádek popisovalo strukturu ExprVarValue již v návrhu 4.3. Pro práci s jejími prvky byly vytvořeny speciální funkce. Není tedy doporučeno se strukturou manipulovat přímo, ale pomocí následujících funkcí:

- createExprVarValue - slouží pro zkonstruování struktury ExprVarValue s novými daty. Funkci je třeba předat prvek z výčtového typu OidTypes, samotnou hodnotu a na závěr indikátor specifikující, zda jsme ji předali typ Datum. Na základě těchto vstupním informací dojde k její inicializaci. Funkce se často používá těsně před vrácením hodnoty z obslužných metod v překladači.

- getJitValueFromExprVarValue - funkce LibJitu neumějí pracovat s typem Datum, ale potřebují znát konkrétní číselný typ. Původní návrh počítal s tím, že metody generující kód budou vždy vracet hodnotu konkrétního typu. Tím pádem by bylo vždy nutné získaný typ Datum přetypovat. Co kdyby náhodou rodičovský uzel volal uživatelskou funkci a potřeboval by přetypovat hodnotu zpět na typ Datum? Naopak by to nevadilo, pokud by se za sebou nacházelo například deset sčítání pomocí LibJitu. Nejhorší případ by nastal, kdyby se volání funkcí kombinovalo s operátory. Zbytečně by se vždy přetypovávalo na Datum a o patro výše hned zase zpět. Pokud tedy například metoda generující kód pro uzel T_VAR načte z databáze Datum, tak neprovede přetypování, ale pouze vytvoří strukturu ExprVarValue pomocí metody createExprVarValue výše. Pokud by rodičovský uzel potřeboval hodnotu konkrétního číselného typu, zavolal by právě metodu getJitValueFromExprVarValue, které by předal nám známou strukturu. V té by došlo k přetypování hodnoty a k jejímu následnému vrácení. Struktura by si již tedy nedržela hodnotu typu Datum (jit_type_ulong), ale například typu jit_type_int. Oid by však zůstalo stejné, protože se skutečný typ

nezměnil. Musela se však upravit booleovská hodnota říkající, že struktura aktuálně uvnitř uchovává typ Datum.

- getJitDatumValueFromExprVarValue - vše podstatné bylo napsáno pro funkci výše. Tato její sestra se použije v případě, že aktuální funkce generující kód potřebuje načíst typ Datum, například pro uživatelskou funkci. Tím opět dojde ke změně reprezentace hodnoty ve struktuře ExprVarValue.

4.4.6 Problém s agregačními funkcemi

Při implementaci metody BuildLibJitExecEvalAggref se narazilo na problém, že v čase kompilace ještě nebylo vytvořené pole econtext->ecxt_aggvalues (více o tomto poli bylo napsáno v kapitole 4.3.1). Z tohoto důvodu bylo nutné trochu přeorganizovat kód metody ExecInitAgg, která provádí inicializaci pro agregační funkce. Změna spočívala v přesunutí metody ExecAssignProjectInfo až téměř na konec za velkou smyčku.

Další problém se týkal absence typu návratové hodnoty v původní funkci ExecEvalAggref, který je potřeba při vytváření struktury ExprVarValue. Naskytovaly se dva způsoby řešení. Buď zavolat další interní funkci PostgreSQL a na základě dostupných informací načíst Oid návratové hodnoty. Nebo přidat další atribut struktuře AggrefExprState, který by v průběhu inicializace byl naplněn na správnou hodnotu. Jelikož nebylo žádoucí přidávat další režii v podobě volání funkce, přistoupilo se k druhému řešení. Do funkce ExecInitAgg byl na správné místo v kódu umístěn řádek "aggrefstate->aggtype = aggref->aggtype", který atribut naplnil správnou hodnotou.

Kapitola 5

Testování

5.1 Návrh testování

Již v úvodu bylo uvedeno, že se budeme zaměřovat na analytické dotazy, které jsou specifické pro OLAP systémy. Proto bez rozmyslu můžeme vyloučit příkazy UPDATE, INSERT a DELETE.

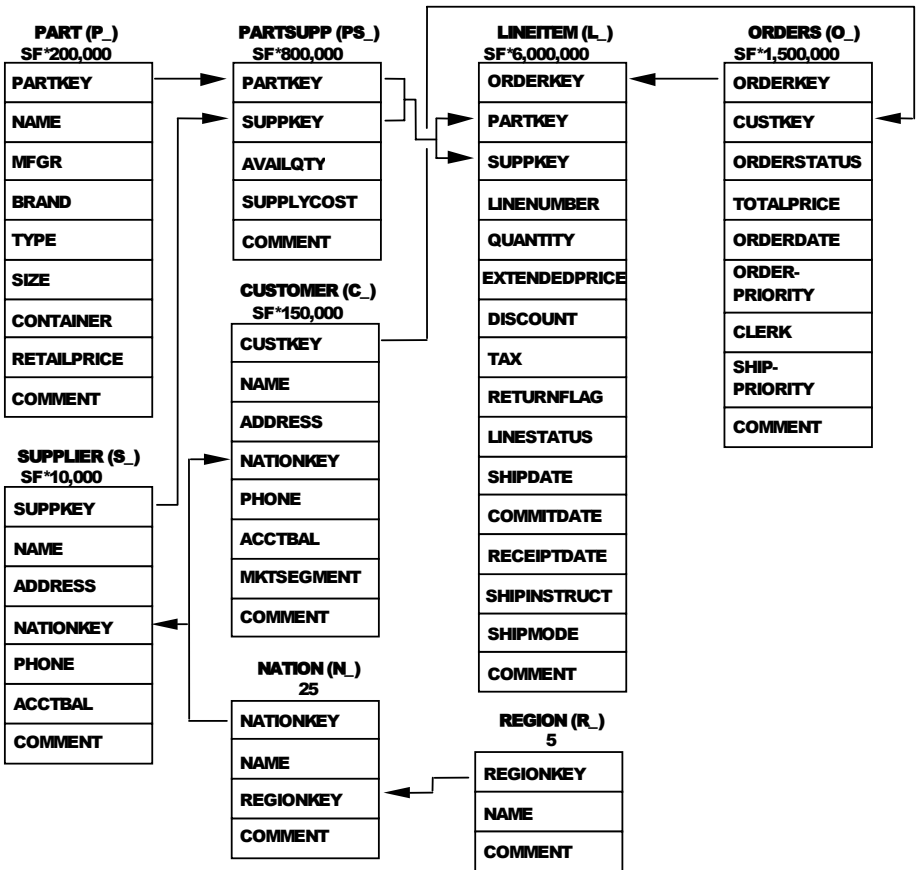
Na začátku je třeba si ujasnit, jaké oblasti by vzhledem k implementovanému řešení měly dotazy pokrývat:

- složité matematické výrazy
- podmíněné výrazy WHEN, CASE
- volání funkcí databáze PostgreSQL
- výrazy uvnitř agregačních funkcí
- složitější výrazy s booleovskými formulemi (OR, AND, NOT)
- dotaz zaměřený speciálně na klauzuli WHERE
- dotaz zaměřený pouze na SELECT s matematickými výrazy

5.1.1 Testovací případy

Prvním testovacím případem je samotná databáze PostgreSQL bez jakýchkoliv úprav. Druhý testovací případ vyšel z rozboru možných způsobů ukládání dat na disku (viz. kapitola 1.7). Proto byl do druhé testované databáze PostgreSQL přidán plugin zajišťující ukládání dat po sloupcích. V literatuře [6] se lze dočíst, že takový způsob zrychluje výpočet agregačních funkcí, které jsou prováděné na co nejmenším počtu sloupců. Z tohoto důvodu bude do testování zařazen dotaz operující pouze nad jedním sloupcem. Poslední nejdůležitější případ tvoří PostgreSQL spolu s integrovaným Just-in-time kompilátorem.

Předchozí odstavec přesně stanovil tři různé instance databáze, na kterých budou prováděny jednotlivé testy. Musíme však přidat další testovací případy, které souvisejí s cachováním dat do paměti. V PostgreSQL k takovému účelu slouží takzvané "shared_buffers". Je obecně doporučováno, aby tato hodnota za normálních okolností byla nastavena na 25% velikosti operační paměti. Jelikož testovací sestava disponuje 4GB paměti, hodnota "shared_buffers" bude v průběhu testování nastavena na 1GB. Nastavená velikost ovlivňuje volbu dalších testovacích případů. Jelikož by naimplementované řešení mělo být závislé na množství dat uložených v operační paměti, budou požity vygenerované datové balíčky o velikosti 1, 2, a 4GB. Předpokládá se, že čím je větší datová sada, tím se začne více projevovat sloupcová metoda uloženi.



Obrázek 5.1: E-R diagram testovací sady dat TPC-H benchmarku

5.1.2 Návrh postupu měření

Nejprve je nutné získat data, nad kterými budeme testovat. Použijeme standardní generátor TPC-H [29], který na základě vstupních parametrů vytvoří

soubory s daty pro konkrétní tabulky. Na soubory je zapotřebí spustit jednoduchý skriptík, aby bylo možné data příkazem COPY nakopírovat do databáze PostgreSQL. Její E-R model si můžete prohlédnout na obrázku 5.1 Abychom mohli na plno vyzkoušet možnosti naimplementovaného řešení, bylo nutné všude změnit datový typ Decimal na Float, jehož velikost je přesně stanovena. Typ Decimal by LibJit nemohl zpracovat vlastní matematickou operací a naimplementovaný program by využil standardní funkci databáze PostgreSQL.

Ještě v rámci příprav bylo rozhodnuto, že na notebooku (jeho specifikace byla uvedena již v podkapitole 4.1), kde bude probíhat testování nepojede nic jiného, než samotná databáze a samozřejmě operační systém. Klientská aplikace, aby zbytečně neovlivňovala výsledky testování poběží na stolním PC, který se bude k databázi připojovat přes lokální síť.

Samotné měření se bude vždy odehrávat podle následujícího scénáře:

1. Pomocí createdb bude vytvořena nová databáze pro všechny tři instance. Původní verze PostgreSQL a druhá doplněná o JIT mohou sdílet stejný adresář na disku.

2. Následně dojde k vytvoření jednotlivých tabulek. Pro testovací případ založený na sloupcovém ukládání dat je třeba vytvořit databázi zvlášť a mít připravený speciální skript pro tvorbu tabulek, ve kterém je nutné používat příkaz CREATE FOREIGN TABLE.

3. Jednoduchý skript pomocí série příkazů COPY nahraje datový set požadované velikosti do databáze.

4. Dojde ke spuštění jedné ze tří instancí databázového serveru.

5. Na stolním PC bude spuštěn skript, který přijímá 2 parametry. První určuje název adresáře, kam se mají ukládat výsledky. Druhý parametr reprezentuje název databáze. Samotný skript nejprve vytvoří kořenový adresář s požadovaným názvem. Uvnitř vytvoří tři podadresáře s názvy results, explain a times. Do adresáře results budou za sebe ukládány výsledky pro jeden dotaz a jeden testovací případ. Stejný dotaz bude vždy spuštěn také s příkazem EXPLAIN pro získání prováděcího plánu. Po každém provedení dotazu dojde k extrahování hodnoty s časem a k uložení do příslušného souboru v adresáři times. Uvnitř skriptu se nastavuje parametr, který říká, kolikrát za sebou se má stejný dotaz poslat do databáze. V rámci realizace byla tato hodnota nastavena na sedm. Proto bude každý soubor ve zmíněných adresářích obsahovat sedm záznamů. Stejné tak bude v adresáři times uloženo sedm časů. Na konci bude vždy vypočítána průměrná hodnota. Ta však není závazná pro naplnění tabulek s výsledky, jelikož se často stávalo, že první časová hodnota byla vždy vyšší než ostatní. Nejspíše z důvodu vlivu cache. Skript postupně projde

všech šest testovacích dotazů včetně dotazů obsahující příkaz EXPLAIN. Do adresářů uloží získané výsledky.

6. Stejný scénář se opakuje pro jiné testovací případy. Tedy při výměně testované instance databáze a při změně velikosti datové sady.

5.2 Vybraná sada testovacích dotazů

Naměřené výsledky byly spíše zklamáním, jelikož v mnoha případech došlo k navýšení doby zpracování dotazu oproti standardní verzi databáze.

5.2.1 Agregační funkce, group by, order by

Dotaz byl vybrán ze standardizované sady testovacích dotazů samotného TPC-H. Zaměřuje se na agregační funkce a použití WHERE, GROUP BY, ORDER BY.

```
SELECT l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)),
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
FROM lineitem
WHERE l_shipdate<=date '1998-12-01' - interval '79' day
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus
LIMIT 1;
```

5.2.1.1 Naměřené výsledky

Velikost dat	DB	DB + LibJit	DB + sloupcové uložení dat
1GB	7549	9885	7930
2GB	14897	19860	15786
4GB	51721	52523	31862

Tabulka 5.1: Naměřené průměrné časy (ms) pro všechny testovací případy

5.2.2 Dotaz s podmíněným výrazem

Jelikož naimplementované řešení podporuje také podmíněné výrazy, byl vytvořen dotaz, který takové výrazy obsahuje.

```
SELECT sum(case when l_quantity = 1 then (l_discount + 1)
  else null end),
 sum(case when l_quantity = 2 then (l_discount + 2)
  else null end),
 sum(case when l_quantity = 3 then (l_discount + 3)
  else null end),
 sum(case when l_quantity = 4 then (l_discount + 4)
  else null end),
 sum(case when l_quantity = 5 then (l_discount + 5)
  else null end),
 sum(case when l_quantity = 6 then (l_discount + 6)
  else null end),
 sum(case when l_quantity = 7 then (l_discount + 7)
  else null end),
 sum(case when l_quantity = 8 then (l_discount + 8)
  else null end),
 sum(case when l_quantity = 9 then (l_discount + 9)
  else null end),
 sum(case when l_quantity = 10 then (l_discount + 10)
  else null end),
 sum(case when l_quantity = 11 then (l_discount + 11)
  else null end)
FROM lineitem;
```

5.2.2.1 Naměřené výsledky

Velikost dat	DB	DB + LibJit	DB + sloupcové uložení dat
1GB	8646	9334	8402
2GB	17619	18926	16280
4GB	53144	52026	33885

Tabulka 5.2: Naměřené průměrné časy (ms) pro všechny testovací případy

5.2.3 Dotaz obsahující složitější výraz

Dotaz je zaměřen na provedení složitějších matematických dotazů v rámci agregačních funkcí. Schválně nebyly použity žádné jiné příkazy.

```
SELECT avg(((l_extendedprice - l_discount)*l_quantity)),
 avg(((l_extendedprice - l_discount) * l_quantity)+
  ((l_extendedprice-l_discount)*l_quantity)) as column2,
 avg(((l_extendedprice - l_discount) * l_quantity) +
  ((l_extendedprice-l_discount)*l_quantity)) as column3,
 avg((l_extendedprice - l_extendedprice * l_tax)+
  (l_extendedprice - l_extendedprice * l_tax) -
  (l_extendedprice - 1)) as column4,
 avg(((l_extendedprice - 1) - (l_extendedprice - 1) *
  (l_tax + 1)) + ((l_extendedprice - 1))) as column5
FROM lineitem;
```

5.2.3.1 Naměřené výsledky

Velikost dat	DB	DB + LibJit	DB + sloupcové uložení dat
1GB	10066	9714	10106
2GB	20070	18588	20313
4GB	55206	51811	39851

Tabulka 5.3: Naměřené průměrné časy (ms) pro všechny testovací případy

5.2.4 Dotaz zaměřený na klauzuli WHERE

Jedná se o druhý a poslední dotaz, který patří do rodiny standardního setu TCP-H. Zaměřuje se na klauzuli WHERE a na booleovské operátory OR a AND. Uvnitř jsou prováděný jednoduší operace nad konkrétními atributy.

```
SELECT
 sum(l_extendedprice* (1 - l_discount)) as revenue
FROM lineitem, part
WHERE(
 p_partkey = l_partkey
 AND p_brand = 'Brand#12'
 AND p_container in ('SM␣CASE','SM␣BOX','SM␣PACK','..')
 AND l_quantity >= 1 AND l_quantity <= 1 + 10
 AND p_size between 1 AND 5
 AND l_shipmode in ('AIR', 'AIR␣REG')
 AND l_shipinstruct = 'DELIVER␣IN␣PERSON'      )
OR(
```

```
 p_partkey = l_partkey
 AND p_brand = 'Brand#23'
 AND p_container in ('MED␣BAG','MED␣BOX','MED␣PKG','..')
 AND l_quantity >= 10 AND l_quantity <= 10 + 10
 AND p_size between 1 AND 10
 AND l_shipmode in ('AIR', 'AIR␣REG')
 AND l_shipinstruct = 'DELIVER␣IN␣PERSON')
OR (
 p_partkey = l_partkey
 AND p_brand = 'Brand#34'
 AND p_container in ('LG␣CASE','LG␣BOX','LG␣PACK','..')
 AND l_quantity >= 20 AND l_quantity <= 20 + 10
 AND p_size between 1 AND 15
 AND l_shipmode in ('AIR', 'AIR␣REG')
 AND l_shipinstruct = 'DELIVER␣IN␣PERSON');
```

5.2.4.1 Naměřené výsledky

Velikost dat	DB	DB + LibJit	DB + sloupcové uložení dat
1GB	2461	3374	2900
2GB	4949	6953	5857
4GB	51249	51493	11630

Tabulka 5.4: Naměřené průměrné časy (ms) pro všechny testovací případy

5.2.5 Dotaz obsahující zanořené funkce

Dotaz se zaměřuje na mnohonásobné vnořování matematických funkcí, kterými je vybavena databáze PostgreSQL. LibJit dokáže takové funkce volat a připravit ze stromu dotazu jednotlivé parametry.

```
SELECT  avg(log(o_totalprice + o_totalprice  +
   o_totalprice)),
 avg(log(pi() + pi() + pi())) as column2,
 avg(log(o_totalprice + o_totalprice + o_totalprice)),
 avg(abs(o_totalprice) * abs(o_totalprice))
FROM Orders
WHERE (log(o_totalprice) < abs(
 log(abs(log(abs(log(o_totalprice +
 o_totalprice + 1)))))) + log(o_totalprice))
AND (log(o_totalprice) < abs(
 log(abs(log(abs(log(o_totalprice +
 o_totalprice + 2)))))) + log(o_totalprice))
```

```
AND (log(o_totalprice) < abs(
 log(abs(log(abs(log(o_totalprice +
 o_totalprice + 3)))))) + log(o_totalprice))
AND (log(o_totalprice) < abs(
 log(abs(log(abs(log(o_totalprice +
 o_totalprice + 4)))))) + log(o_totalprice))
GROUP BY o_orderdate;
```

5.2.5.1 Naměřené výsledky

Velikost dat	DB	DB + LibJit	DB + sloupcové uložení dat
1GB	7231	5660	6870
2GB	14337	10591	13552
4GB	28488	21026	27040

Tabulka 5.5: Naměřené průměrné časy (ms) pro všechny testovací případy

5.2.6 Dotaz nad jedním sloupcem

Již jsme se v textu zmínili, že sloupcové uložení dat nejlépe pracuje při malém počtu použitých sloupců v dotazu. Abychom také spustily náš zkompilovaný kód, byl použit v rámci agregační funkce jednoduchý matematický výraz.

```
SELECT avg(l_quantity + l_quantity),
 min(l_quantity + l_quantity),
 max(l_quantity + l_quantity),
 count(*),
 avg(l_quantity + l_quantity),
 min(l_quantity + l_quantity),
 max(l_quantity + l_quantity),
 count(*)
FROM lineitem;
```

5.2.6.1 Naměřené výsledky

Velikost dat	DB	DB + LibJit	DB + sloupcové uložení dat
1GB	3108	6679	2765
2GB	6216	11976	5595
4GB	50915	50778	11225

Tabulka 5.6: Naměřené průměrné časy (ms) pro všechny testovací případy

5.3 Diskuse naměřených výsledků

Naměřené výsledky byly z pohledu implementovaného řešení spíše zklamáním, jelikož pouze ve dvou případech z šesti došlo k plnému zkrácení doby provádění dotazu. Jednalo se o dotazy, které obsahovaly komplikovanější matematické výrazy. Dotaz 5.2.3 je toho řádným příkladem. Druhý dotaz 5.2.5 zanořuje velké množství funkcí, s čímž je spjata určitá režie, kterou dokázal LibJit úspěšně eliminovat. U 1GB datového setu je v prvním případě rozdíl 352 ms a u 4GB datové setu 3395 ms. V případě dotazu zaměřeného na vnořování funkcí je rozdíl výraznější. 1GB datový set vykazuje rozdíl 1571 ms. Provádění dotazu se v případě 4GB datového setu dokonce zrychlilo o 7462 ms. Obecně jsou výsledky horší na sadách s menším objemem dat. Se stoupajícím množstvím se začínají rozdíly stírat. Podmíněný výraz dokonce jako jediný na sadě o velikosti 4GB vykázal mírné zlepšení. K největšímu zhoršení došlo hned u prvního dotazu 5.2.1, kde byly sice použity matematické výrazy, ale zároveň byly aplikovány příkazy WHERE, GROUP a ORDER BY. Vypadá to, že s větším objemem dat narůstá důležitost počtu IO operací, takže se ve výsledcích stírá rozdíl. Ale je na pováženou zařadit do této úvahy také oba dva úspěšné dotazy, kde s větším objemem dat narůstal rozdíl mezi změřenými časy. Ke zhoršení došlo také u dotazu 5.2.4 zamířeného na klauzuli WHERE a na operátory OR a AND. Je také možné, že integrace obsahuje bug, který zapříčiní, že se projdou všechny části OR, aniž by musely. Tato hypotéza však nebyla ověřena.

Poměrně jasnější výsledky vykazuje řešení založené na sloupcovém ukládání dat. Se zvyšujícím se objemem dat rapidně stoupá rozdíl mezi původním a novým časem. Podle předpokladu z podkapitoly 5.1.1 došlo k nejvýraznějšímu nárůstu rychlosti při použití agregační funkce nad jedním sloupcem. Abychom LibJitu zase tolik nekřivdili, tak v případě vnořování funkcí došlo ve všech testovaných případech k překonání výsledného času instance, která ukládá data po sloupcích. LibJit dále u složitějšího matematického výrazu v dotazu 5.2.3 musel kapitulovat pouze v případě 4GB datového setu, kde byl rozdíl již výrazný. Nejméně se sloupcovému přístupu dařilo v případě vnořování funkcí v dotazu 5.2.5, ačkoliv byl prováděn pouze nad jedním atributem. To dokazuje, že jeho režie je opravdu veliká a LibJit se v takové situaci uplatňuje ze všech možností nejvíce. Rozdíl mezi původním časem a časem metody založené na ukládání po sloupcích je i v případě velkého datového setu 4GB pouze 1448 ms. V jiných případech bývá rozdíl okolo 20000 ms. Je ale pravda, že se velká část výrazů vyskytuje v klauzuli WHERE.

Kapitola 6

Závěr

V první kapitole 1 jsme se seznámili se samotnou databází PostgreSQL. Ukázalo se, že databáze zavádí své volací konvence funkcí, aby se eliminovaly nedostatky standardní verze. V jádru dále obsahuje sadu pomocných maker, které usnadňují programátorovi život. Z pohledu integrace Just-in-time kompilátoru nejde o šťastné řešení, protože na makra nelze získat odkaz, zatímco na funkce ano. Dalším významným prvkem databáze PostgreSQL je, že funkce na vstupu a na výstupu pracují se speciálním datovým typem Datum. Její kód je napsán v jazyku C, který nepodporuje objektové programování. Proto se vývojáři snažili nahradit dědičnost speciálním typem Node a sadou maker. Podkapitola věnující se možným způsobům ukládání dat na disk uzavírá celou sekci.

Druhá kapitola 2 se nejprve zabývala obecným zpracováním dotazu. Dále jsme pochopili činnost a vnitřní mechanismy obou dvou interpretů. Několik stránek navíc bylo věnováno převážně interpretu složitějších výrazů. Důvod byl jednoduchý, právě ten měl být ve výsledku nahrazen Just-in-time kompilací. Seznámili jsme se s principem parseru a s tvorbou stromu odpovídajícímu zmíněnému výrazu. Identifikovali jsme jeho důležité uzly a vnitřní struktury. Důležité je poznamenat, že hlavní interpret prováděcího plánu je založen na velkém switchi, zatímco interpret výrazů využívá volání funkcí. Jejich ukazatelé jsou právě uloženy v uzlech stromu výrazu.

Třetí kapitola 3 se nejprve věnovala možnostem 64 bitové architektury. Poté se zabývala jednotlivými virtuálními stroji a Just-in-time kompilátory. Hned na začátku byly zmíněny komplexní virtuální stroje, které ve svých útrobách ukrývají funkční JIT kompilátor. Jeho služeb by bylo teoreticky možné využít. Bylo by ale nutné držet se specifikace virtuálního stroje (například Javy) a vygenerovat soubor s bytekódem, kterému by rozuměl. Druhou přijatelnější skupinu tvoří nezávislé Just-in-time kompilátory. Ty se právě staly díky této vlastnosti největšími kandidáty pro výslednou integraci. Kapitolu uzavírají spíše jen pomocné generátory, které usnadňují tvorbu nativního kódu.

Následná kapitola 4 se věnovala návrhu a realizaci. Nejprve byla naimplementována sada testovacích algoritmů, na kterých byly ověřeny rychlosti jednotlivých řešení z předchozí analýzy. Vybraná knihovna LibJit jasně nezvítězila, co se rychlosti týče, ale díky skutečnostem uvedených v analýze byla nakonec zvolena. Zejména potěšilo její hezky navržené API, automatická alokace všech registrů 64 bitové architektury a snadné volání nativních funkcí. Samotná integrace spočívala ve vytvoření jádra překladače ve formě velkého switche, který pro jednotlivé typy uzlů volal funkce generující příslušný nativní kód. Větší problémy nastaly při hledání chyb a debuggování. Chyby nebylo snadné odhalit a debugger poměrně často padal.

Testování v kapitole 5 nepotvrdilo jasnou dominanci LibJitu v případě složitějších výrazů. Nejvíce se rychlost zvýšila při provádění složitého matematického výrazu a při několikanásobném vnořovaní funkcí. V ostatních situacích byly naměřené časy často horší než původní. Z toho plyne, že sám PostgreSQL neoptimalizuje tyto dotazy špatně. Samotnou příčinu můžeme hledat v knihovně LibJit. Je teoreticky možné, že druhý kandidát LLVM by danou úlohu zvládl lépe. Metoda založená na ukládání dat po sloupcích zlepšuje rapidně výkon zejména na větších datových sadách. Největší výsledky jsou dosaženy při aplikaci agregačních funkcí na co nejmenší počet sloupců. Metoda je naopak poražena LibJitem, pokud se uvnitř dotazu vyskytuje opravdu velké množství složitých matematických výrazů.

Literatura

[1] Types of OLAP Database Systems. ©2015. Dostupné z: http://olap.com/types-of-olap-systems/

[2] PostgreSQL. © 1996-2015. Dostupné z: http://www.postgresql.org/

[3] Stěhule, P.: PostgreSQL. 2011. Dostupné z: http://postgres.cz/wiki/C_a_PostgreSQL_-_interní_mechanismy

[4] Momjian, B.: Inside PostgreSQL Shared Memory. 2009. Dostupné z: https://www.cs.drexel.edu/~twc24/cs500_su11/slides/inside_shmem.pdf

[5] Moshayedi, H.: Cstorefdw. 2015. Dostupné z: https://github.com/citusdata/cstore_fdw

[6] Column-Oriented Database Implementation in PostgreSql for Improving Performance of Read-Only Queries. 2012. Dostupné z: http://www.coep.org.in/page_assets/341/121022008.pdf

[7] Syntaktický strom. 2001-2014. Dostupné z: http://cs.wikipedia.org/wiki/Syntaktick%C3%BD_strom

[8] JIT through the ages. 2012. Dostupné z: http://www.cs.columbia.edu/~aho/cs6998/reports/12-12-17_Ramanan_JIT.pdf

[9] Introduction to x64 Assembly. 2012. Dostupné z: https://software.intel.com/en-us/articles/introduction-to-x64-assembly

[10] Intel® 64 and IA-32 Architectures Developer's Manual. 2015. Dostupné z: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html

[11] X86 calling conventions. 2001-2015. Dostupné z: http://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions

[12] Sompolski, J.: Just-in-time Compilation in Vectorized Query Execution. 2011. Dostupné z: http://homepages.cwi.nl/~boncz/msc/2011-JuliuszSompolski.pdf

[13] Muchnick, S. S.: *Advanced compiler design and implementation*. San Francisco: Morgan Kaufmann, 1997, ISBN 15-586-0320-4.

[14] Venners, B.: *Inside the Java virtual machine*. New York: McGraw-Hill, druhé vydání, c1999, ISBN 00-713-5093-4.

[15] Doležel, L.: Java Native Interface. 2010. Dostupné z: http://www.abclinuxu.cz/clanky/java-native-interface-propojujeme-javu-a-c-cplusplus-1#jni-co-a-proc-to-je

[16] Bruneton, E.: ASM 4.0. 2011. Dostupné z: http://download.forge.objectweb.org/asm/asm4-guide.pdf

[17] Tišnovský, P.: Pohled pod kapotu JVM. 2013. Dostupné z: http://www.root.cz/clanky/pohled-pod-kapotu-jvm-zaklady-optimalizace-aplikaci-naprogramovanych-v-jave/

[18] Tišňovský, P.: Seriál Programovaci jazyk Java a JVM. 2014. Dostupné z: http://www.root.cz/serialy/programovaci-jazyk-java-a-jvm/#ic=serial-box&icc=more

[19] Tišňovský, P.: Programovací jazyk Lua. 2009. Dostupné z: http://www.root.cz/clanky/programovaci-jazyk-lua/

[20] Tišňovský, P.: LuaJIT. 2014. Dostupné z: http://www.root.cz/clanky/luajit-just-in-time-prekladac-pro-programovaci-jazyk-lua/

[21] The LuaJIT Wiki. 2013. Dostupné z: http://wiki.luajit.org

[22] Lopes, B. C.; Auler, R.: *Getting Started with LLVM Core Libraries*. 2014, ISBN 978-1-78216-692-4. Dostupné z: http://www.pdf-library.ir/files/uploads/1416250066.1782166920.pdf

[23] Lattner, C.: LLVM. 2012. Dostupné z: http://www.aosabook.org/en/llvm.html

[24] Bendersky, E.: Getting started with libjit. 2013. Dostupné z: http://eli.thegreenplace.net/2013/10/17/getting-started-with-libjit-part-1

[25] Just-In-Time Compiler Library. 2004. Dostupné z: https://inst.eecs.berkeley.edu/~cs164/fa11/docs/libjit.pdf

[26] Using GNU lightning. © 1996 - 2015. Dostupné z: http://www.gnu.org/software/lightning/manual/lightning.html

[27] Kobalicek, P.: Asmjit. 2015. Dostupné z: https://github.com/kobalicek/asmjit

[28] Parrot 7.3.0 - Home. © 2002-2014. Dostupné z: http://docs.parrot.org/parrot/latest/html/

[29] TPC-H. © 2015. Dostupné z: http://www.tpc.org/tpch/

Příloha A

Ukázka instrukcí LuaJITu

A.1 Aritmetické instrukce

- var: číslo slotu proměnné
- dst: číslo slotu cílové proměnné
- num: číselná konstanta, popřípadě index do tabulky konstant

Instrukce	A	B	C	Popis
ADDVN	dst	var	num	A = B + C
SUBVN	dst	var	num	A = B - C
MULVN	dst	var	num	A = B * C
DIVVN	dst	var	num	A = B / C
MODVN	dst	var	num	A = B % C
ADDNV	dst	var	num	A = C + B
SUBNV	dst	var	num	A = C - B
MULNV	dst	var	num	A = C * B
DIVNV	dst	var	num	A = C / B
MODNV	dst	var	num	A = C % B
ADDVV	dst	var	var	A = B + C
SUBVV	dst	var	var	A = B - C
MULVV	dst	var	var	A = B * C
DIVVV	dst	var	var	A = B / C
MODVV	dst	var	var	A = B % C
POW	dst	var	var	A = B ^ C

Tabulka A.1: Aritmetické instrukce LuaJITu

Příloha B

Překlad funkce do LLVM IR

Výpis B.1: Ukázka překladu jednoduché funkce do IR

```
int sum(int a, int b) {
 return a+b;
}

define i32 @sum(i32 %a, i32 %b) #0 {
entry:
 %a.addr = alloca i32, align 4
 %b.addr = alloca i32, align 4
 store i32 %a, i32* %a.addr, align 4
 store i32 %b, i32* %b.addr, align 4
 %0 = load i32* %a.addr, align 4
 %1 = load i32* %b.addr, align 4
 %add = add nsw i32 %0, %1
 ret i32 %add
}
```

Z výpisu B.1 (zdroj [22]) výše je na první pohled vidět, že vygenerovaný kód není optimalizovaný. Vysvětlení ukázky LLVM IR:

- Nejprve se pomocí instrukce alloca zarezervuje místo na zásobníku funkce pro obě proměnné. (2x 4 byty(i32) s ohledem na 4 bytové zarovnání)
- %a, %b jsou uloženy na adresy %a.addr, respektive %b.addr na zásobníku. Hodnoty %a, %b jsou nahrány zpět ze stejných adres.
- Nakonec je použita instrukce sčítání a dojde k navrácení hodnoty.

Příloha C

Vybrané testovací algoritmy

C.1 Přibližný výpočet čísla Pí

Algoritmus 1 Přibližný výpočet čísla Pí v programovacím jazyku C

```
double piFunc(){
 double pi = 3;
 int i = 2;
 double piAdd, piSub;
 for( int l=0; l <= 10000000; l++){
  piAdd = 4.0 / (i * (i + 1.0) * (i + 2.0));
  piSub = 4.0 / ((i + 2.0) * (i + 3.0) * (i + 4.0));
  pi= pi + piAdd;
  pi= pi - piSub;
  i= i + 4;
 }
 return pi;
}
```

C.2 Počet prvočísel do konkrétního čísla

Algoritmus 2 Algoritmus, který nalezne počet prvočísel do zadaného čísla

```
int primeNumbersCount(int number){
 bool ok;
 int count = 0;
 for(int l = 2; l <= number; l++){
  ok = true;
  for(int i = 2; i < l; i++){
   if ((l % i) == 0) {
    ok = false;
    break;
   }
  }
  if(ok){
   count = count + 1;
  }
 }
 return count;
}
```