Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Využití vektorových instrukcí v agregačních funkcích na platformě x86

Plzeň 2018 Václav Löffelmann

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 17. května 2018 Václav Löffelmann

Abstract

This work aims to maximize use of vector instructions on data aggregation tasks. A new vector instruction set AVX-512 is used in this work. Implementation proves that a proper usage of the vector instructions can significantly speed up the processing.

Abstrakt

Cílem této práce bylo maximalizovat využití vektorových instrukcí při výpočtech agregačních funkcí. Práce je postavena nad novou instrukční sadou vektorových instrukcí AVX-512. Implementací se podařilo ukázat, že vhodné použití vektorizace může výpočty agregačních funkcí výrazně urychlit.

Na tomto místě bych chtěl poděkovat panu Lukášovi Benešovi za podporu po celou dobu studia a panu Martinu Zímovi za ochotné vedení diplomové práce.

Obsah

1 Úvod **1**

2 Vektorové instrukce na rodině procesorů x86-64 **2**
2.1 Exekuční model x86-64 2
2.1.1 Instrukční část procesoru 2
2.1.2 Exekuční část procesoru 4
2.1.3 Latence a propustnost instrukcí 7
2.1.4 Paměťový subsystém procesoru 7
2.2 Vektorové instrukce na CPU - SIMD model 9
2.2.1 Použití vektorových instrukcí 10
2.3 Vektorová instrukční sada AVX-512 11
2.3.1 Rozšíření registrů 13

3 Koncepty agregovaní dat **15**
3.1 Užívané datové struktury 15
3.2 Kompilace vs. interpretace agregačních funkcí 16
3.3 Modely zpracování a organizace dat v paměti 18
3.3.1 Zpracování po n-ticích 18
3.3.2 Zpracování po sloupcích 19
3.3.3 Zpracování po vektorech 20
3.3.4 Zpracování po vektorových registrech 20
3.4 Paralelní zpracování, konzistence a maskované instrukce 22
3.5 Deklarativní zápis požadavků 23

4 Dostupné nástroje pro zpracování dat v paměti **24**
4.1 Kritéria výběru 24
4.2 MonetDB 25
4.3 MonetDB/X100 - VectorWise 25
4.4 Relační SQL databáze (PostgreSQL, MariaDB) 26

4.5 NoSQL databáze . . . 26
4.6 ClickHouse . . . 26
4.7 SQLite . . . 27

5 Implementace agregačních funkcí 28
5.1 Koncept implementace zpracování po vektorových registrech . . . 28
5.2 Příprava SQL požadavku . . . 29
5.3 Filtrace záznamů . . . 30
5.3.1 Vstupní maska . . . 31
5.3.2 Vyhodnocování aritmetických výrazů . . . 31
5.3.3 Vyhodnocování logických výrazů . . . 31
5.3.4 Optimalizace maskovaného zpracování . . . 32
5.4 Jednoduché agregace . . . 32
5.5 Agregace podle hodnoty - GROUP BY . . . 33
5.6 Ručně vektorizovaná hashovací tabulka . . . 34
5.6.1 Způsob řešení kolizí hashů . . . 35
5.7 Generátory algoritmů využívající vkladače pro různé datové typy 37
5.8 Finalizace požadavku . . . 38
5.8.1 Řazení výsledků . . . 39
5.9 Příklad zkompilovaného dotazu . . . 40
5.10 B+ strom s vektorizovaným vyhledáváním . . . 44

6 Dosažené výsledky - testování 47
6.1 Softwarové závislosti . . . 47
6.2 Hardware použitý pro testování . . . 47
6.3 Překlad a spuštění . . . 48
6.4 Ověření výsledků agregačních dotazů . . . 49
6.5 Vyhodnocení výkonnosti agregačních dotazů . . . 49
6.5.1 Testovací dataset . . . 49
6.5.2 Sledované dotazy . . . 50
6.5.3 Naměřené časy vyhodnocení agregačních dotazů . . . 53
6.5.4 Srovnání s databází SQLite . . . 53
6.5.5 Srovnání se skalární implementací . . . 55
6.6 Vyhodnocení výkonnosti agregačních hashovacích tabulek . . . 58
6.7 Vyhodnocení výkonnosti vektorizovaného vyhledávání v B+ stromu 62

7 Závěr 66

Literatura 66

Kapitola 1

Úvod

Dnešní procesory zvyšují výkon na jádro hlavně díky vektorovým instrukcím. Tato práce se zabývá tím, jak efektivně lze tyto vektorové instrukce využít při agregování dat. V dnešní době již existuje nepřeberné množství nástrojů a databázových systémů umožňující počítat agregované funkce, ale tyto nástroje často naráží na problém nemožnosti vektorizace kódu a tím i nedostatečného využití dostupných výpočetních prostředků.

Problém agregace dat obecně patří mezi vektorizovatelné úlohy, avšak návrhy současných systémů jsou moc komplikované na to, aby byly překladače schopny tyto části vektorizovat. Cílem této práce je právě příprava takového kódu, který umožní využívat vektorové instrukce, jak jen to bude možné. Zaměříme se na vektorizaci pomocí současné, nejpokročilejší, vektorové instrukční sady AVX-512.

V této práci je představen přístup překladu agregačního dotazu, zapsaném v syntaxi jazyka SQL, do vektorových instrukcí a volání funkcí, které zapouzdřují složitější vektorové operace. Při generování se preferuje, aby byla data mezi jednotlivými fázemi výpočtu předávána přímo ve vektorových registrech, což zcela eliminuje latenci přístupu do paměti. Jedná se tedy o model zpracování vektorového registru zároveň.

Pro zpracování agregací dle závislé hodnoty byla implementována vektorizovaná hashovací tabulka a využití vektorových instrukcí při vyhledávání je demonstrováno na případu B+ stromu.

Kapitola 2

Vektorové instrukce na rodině procesorů x86-64

V této kapitole se seznámíme s vybranými principy fungování procesorů na architektuře x86-64, konkrétně se zaměříme na aktuálně nejmodernější architekturu procesorů *Intel Skylake server* (kódové označení *SKX*), která zatím jako jediná podporuje vektorové instrukce ze sady AVX-512. Rozebereme zjednodušený exekuční model procesoru a poté se zaměříme na vektorové instrukční sady, konkrétně na relativně novou instrukční sadu AVX-512 a její podsady.

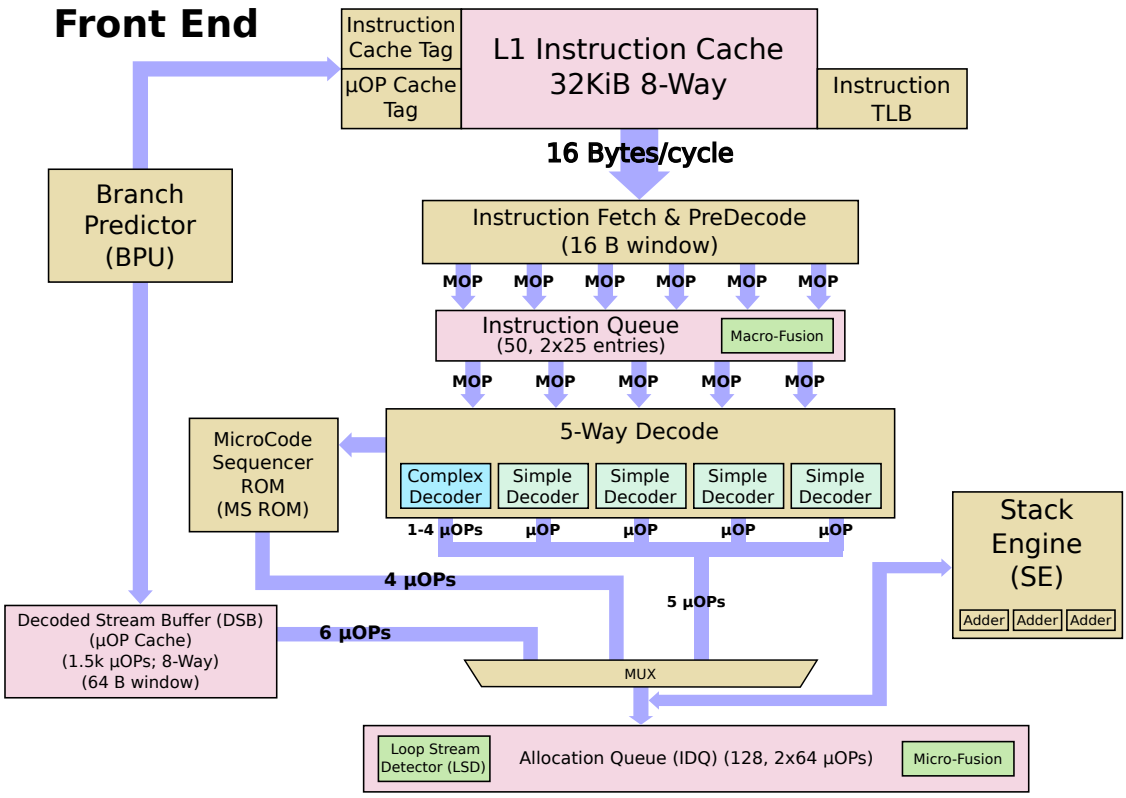
2.1 Exekuční model x86-64

Z vysokoúrovňového pohledu můžeme procesor rozdělit na tři hlavní části. První částí je instrukční část procesoru, kde se načítají a dekódují instrukce ke zpracování. Druhou částí je pak exekuční část procesoru, kde jsou dekódované instrukce vykonávány na příslušných hardwarových obvodech. Poslední částí, na kterou se zaměříme je paměťový subsystém, který obsahuje několik vyrovnávacích pamětí a několik úrovní mezipaměti.

2.1.1 Instrukční část procesoru

Dnešní CISC (Complex Instruction Set Computing) procesory mají více stupňovou pipeline, to jim dovoluje zpracovávat více instrukcí, které se nacházejí v různých fázích, zároveň. Tato pipeline má u architektury Intel Skylake celkem 14 až 19 kroků v závislosti na instrukci a ostatních optimalizacích. Vyšší

hloubka pipeline má ovšem jednu zásadní nevýhodu, a tou je vyčištění zbývajících částí pipeline a následné pozastavení zpracovávání v případě neúspěšné predikce skoku.



Obrázek 2.1: Schéma instrukční části jádra procesoru Intel architektury Skylake. Zdroj [5].

Na obrázku 2.1 je schématicky znázorněna struktura instrukční části (*front-end*) procesorového jádra architektury Skylake. V této části procesoru se instrukce přečtou z L1 paměti (*cache*). Tyto instrukce se dekódují na makro operace (*Macro-Operation* - MOP). Dekódované operace se zařadí do instrukční fronty. Některé kombinace dvou po sobě jdoucích instrukcí se dokonce za určitých okolností mohou spojit v jednu (například často používaná kombinace instrukcí `CMP JNE`). Jedná se o tzv. *Macro-Operation Fusion* optimalizaci. Makro operace jsou dále poslány na jeden z dekodérů. V dekodéru se makro operace rozloží na

jednu nebo více mikroinstrukcí (μOP[1]). Typicky instrukce kde jeden operand je registr a druhý paměť se interpretuje jako dvě mikroinstrukce. Složitější vektorové instrukce se pak rozkládají i na více než dvě mikroinstrukce. Pokud se jedná o opravdu komplexní makro operaci, tak je potřeba spustit sérii mikroinstrukcí, které jsou uloženy v paměti tzv. *Microcode Sequencer*. Nicméně volání Microcode Sequenceru přináší jistou režii a proto je vhodné se těmto operacím v některých případech vyhnout (podrobnosti viz [11] sekce 16.2.1.1).

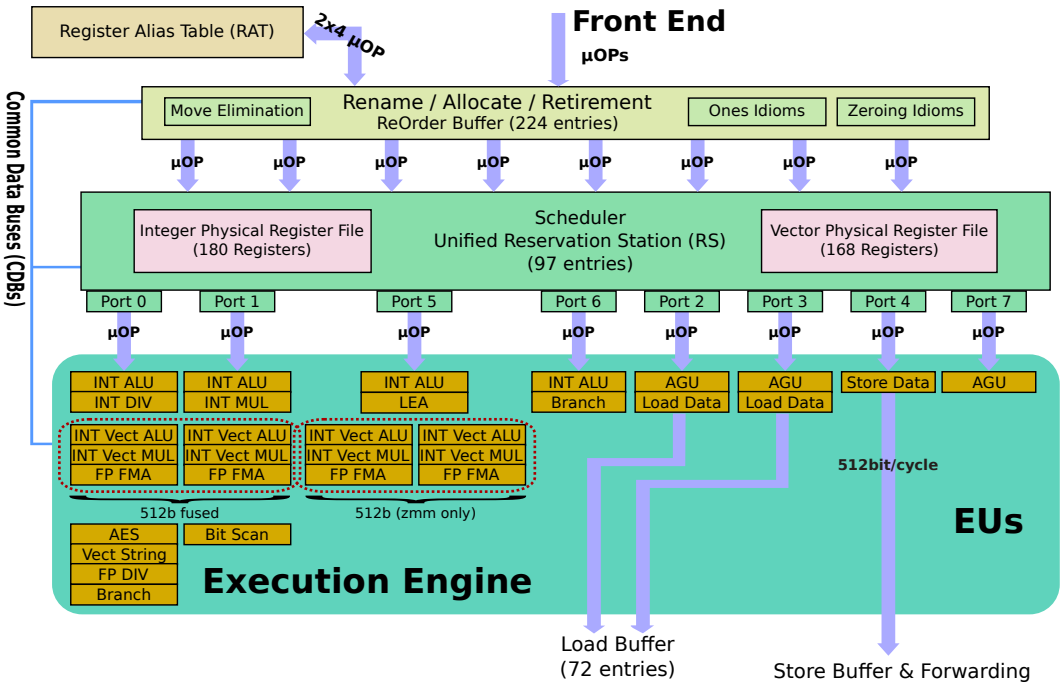
Aby se ušetřila režie s dekódováním instrukcí, které již byly dekódovány, front-end procesoru obsahuje také cache pro tyto mikroinstrukce - tzv. *Decoded Stream Buffer (DSB)*. Mikroinstrukce se ukládají v bufferu jelikož jejich uchovávání je energeticky efektivnější než je znovu dekódovat.

Všechny tyto mikroinstrukce putují do tzv. alokační fronty (Instruction Allocation Queue - IDQ), která slouží jako rozhraní mezi front-endem a back-endem procesoru. Toto rozhraní je potřeba, jelikož front-end zpracovává instrukce v pořadí, ve kterém se vyskytují v programu, ale back-end zpracovává instrukce mimo jejich pořadí v programu (Out-of-order - OOO). V rámci této logické jednotky se provádí ještě dva typy optimalizací. Prvním z nich je detekce krátkých smyček (Loop Stream Detection - LSD), jež zvládá detekovat smyčky mikroinstrukcí, které jsou dostatečně krátké na to, aby se vešly do alokační fronty. Tento mechanizmus se využívá například při krátkém prohledávání stringů, jelikož zvládá uchovávat až 28 mikroinstrukcí. Využívání LSD efektivně vypne celou předchozí část pipeline a díky tomu může šetřit elektrickou energii - [11] sekce 2.4.2.4. Druhou optimalizací, která se v rámci alokační bufferu děje, je spojení mikroinstrukcí - tzv. Micro-Fusion. Jedná se o sloučení takových jednoduchých mikroinstrukcí, které lze spojit do komplexnější mikroinstrukce. Může se jednat například o načtení paměti a následnou mikroinstrukci nad registry, která se sloučí na mikroinstrukci, jež má jeden z operandů přímo požadovanou paměť [11] - sekce 2.4.2.1.

2.1.2 Exekuční část procesoru

U moderních procesorů, pipeline není jediným způsobem, jak provádět operace paralelně. Paralelně totiž lze vykonávat mikroinstrukce na více exekučních portech procesoru. Pak se jedná o instrukční paralelismus (Instruction Level Parallelism - ILP). Obrázek 2.2 ukazuje schéma back-endu (exekuční části) procesoru.

[1]Nebo také zjednodušeně uOP



Obrázek 2.2: Schéma back-endu jádra procesoru Intel architektury Skylake. Zdroj [5].

Dekódované mikroinstrukce, které dodá front-end procesoru putují do vyrovnávací paměti tzv. Reorder Buffer, ve které se provádí hned několik operací. Jedna z operací je alokace fyzických registrů procesoru. Například procesor architektury Skylake má ve skutečnosti 180 fyzických registrů, avšak pouze zlomek takových, které jsou skalární, obecné a uživatelsky adresovatelné. Hlavním důvodem pro takový nepoměr mezi adresovatelnými a fyzickými registry je právě vykonávání instrukcí mimo pořadí. Další z těchto operací je eliminace nadbytečných přesunů dat mezi registry (Move Elimination). Pokud procesor detekuje přesun, který může být eliminován pouhým přejmenováním registrů, tak pouze upraví záznam v tabulce aliasů registrů (Register Alias Table - RAT), tedy pouze provede přejmenování. Vlastní přesun se tak vůbec nemusí provádět a tím pádem tyto instrukce nemají žádnou latenci a také se tím efektivně zvyšuje počet možných obsloužených instrukcí za procesorový takt (viz [11] sekce 16.2.2.7).

Majoritní výhodou přejmenovávání registrů je možnost odstranění datových závislostí mezi jednotlivými operacemi, které by jinak bránily spouštění instrukcí mimo jejich pořadí. Odstranění datových závislostí se také provádí nulováním

registrů pomocí specifikovaných instrukcí (Dependency Breaking Idioms), které jsou v této vyrovnávací paměti detekovány a vyhodnoceny jako nová alokace registru. Pokud by se registr nevynuloval, tak procesor nemá způsob, jak zjistit, jestli se program na jeho obsah nespoléhá někde dále v kódu, a tudíž by mohlo být zabráněno provádění instrukcí mimo jejich pořadí. Používaným idiomem nulování je například instrukce `XOR` se stejným cílovým i zdrojovým registrem. Obdobným způsobem lze také např. nastavit logické jedničky ve všech bitech registru `XMM1` za pomocí instrukce `CMPEQ XMM1, XMM1`. Poslední důležitou funkcionalitou této vyrovnávací paměti je uchovávání informací o již proběhlých instrukcích (Retirement) a příslušné modifikace stavu procesoru v závislosti na výsledku operací a případných výjimkách. V případě zpracování výjimek se tato část procesoru stará i o jejich korektní pořadí, které bylo narušeno vykonáváním mimo pořadí.

Mikroinstrukce jsou dále poslány do plánovače (Scheduler nebo také *Unified Reservation station*), kde se řeší plánování a logika spouštění mikroinstrukcí na exekučních jednotkách (Execution Units). Cílem plánovače je co nejvyšší saturace exekučních jednotek a předcházení *hazardům* [14]. Hazardy se vyskytují v případě, kdy by pipelining instrukcí mohl ohrozit správnost výsledku a obecně se rozdělují na více typů - datový, strukturální a řídící. Aby se při výskytu hazardů zabránilo nekorektním výsledkům výpočtů, tak se na několik cyklů pozastaví zpracování na dotčených stupních pipeline.

Plánovač má na architektuře Skylake k dispozici celkem osm portů, vedoucích k exekučním jednotkám, na které může posílat mikroinstrukce. Každá z exekučních jednotek obsahuje určité hardwarové obvody umožňující spouštět příslušné mikroinstrukce. Některé instrukce, například jednoduché aritmetické operace s celými čísly, lze spouštět na více exekučních portech (exekuční jednotky 0, 1 a 5). Jiné mikroinstrukce, typicky, specializované mikroinstrukce pro šifrování (AES) a nebo mikroinstrukce s vysokou latencí, jako například dělení v plovoucí desetinné čárce, lze naopak spouštět pouze na určitém portu - v případě procesorů architektury Skylake pouze na portu 0. Zajímavostí na procesorové architektuře Skylake Server (která jako jediná v době psaní obsahuje instrukční sadu AVX-512) je, že exekuční jednotky 0 a 1, které mohou samostatně spouštět operace nad až 256 bitovými vektory, sdílejí hardwarové obvody pro vektorové instrukce používající operandy o šířce 512 bitů. Pokud plánovač odešle instrukci s 512 bitovými operandy na port 0, tak tím zároveň vytíží obvody zajišťující vektorové zpracování na exekuční jednotce 1.

2.1.3 Latence a propustnost instrukcí

Realizace oddělených exekučních jednotek nám dovoluje v jednom cyklu naplánovat hned několik mikroinstrukcí. Celkem plánovač za jeden takt zvládne naplánovat (teoreticky) až 6 mikroinstrukcí v závislosti na druhu mikroinstrukcí, vytížení exekučních portů, bezprostředním okolí kódu, přítomností mikroinstrukcí ve vyrovnávacích pamětech, hazardech a ještě několika dalších okolnostech [11]. Častěji se však u jednoduchého a optimalizovaného skalárního kódu setkáme se čtyřmi vykonanými instrukcemi za jeden takt [8].

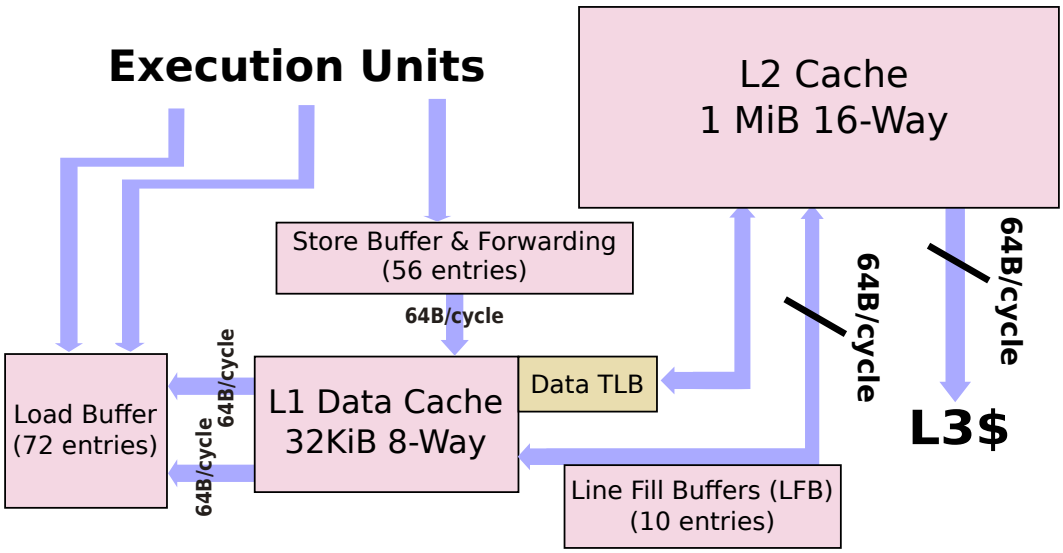
Vzhledem k povaze CISC instrukcí na rodině procesorů x86, jednotlivé instrukce se liší v latenci a propustnosti. Latence je definována jako počet cyklů potřebný k dokončení všech operací spuštěných danou instrukcí. Propustnost naproti tomu značí za kolik cyklů může být stejná instrukce spuštěna na stejném exekučním portu, resp. portech [11] příloha C.2.

Počet obsloužených instrukcí za takt značí metrika IPC (Instruction Per Cycle - počet instrukcí za cyklus) a její převrácená hodnota CPI (Clocks Per Instruction - průměrný počet cyklů na instrukci). Tyto metriky slouží k hrubému odhadu výkonnosti určitého kódu.

2.1.4 Paměťový subsystém procesoru

Na obrázku 2.3 je znázorněné schéma paměťového subsystému procesoru. Z exekučních jednotek se přistupuje buď do mezipaměti pro čtení (Load Buffer) a nebo do mezipaměti pro zápis (Store Buffer) v závislosti na exekuční jednotce. Architektura Skylake má dvě exekuční jednotky pro čtení a jednu pro zápis. Každý cyklus tedy procesor může provést až dvě čtení a jeden zápis 64 bytů. Mezipaměť pro zápis slouží k dočasnému uložení zápisů, které jsou spouštěny spekulativně. Mezipaměť pro čtení také slouží při spekulativním spouštění - hlídá pořadí. Dále pak uchovává vypočtenou adresu paměti pro čtení do té doby, dokud nejsou data k dispozici (například při výpadku cache nebo předchozím zápisu na stejné místo v paměti).

Zápisová i čtecí mezipaměť komunikují s datovou vyrovnávací pamětí první úrovně (L1), jež má velikost 32 KiB a je 8-cestně asociativní. Asociativita paměti značí, na kolik různých míst ve vyrovnávací paměti může být namapované libovolné místo v paměti. V našem případě tedy máme osm možností, kam uložit data příslušící jednomu místu v paměti. Vyšší asociativita paměti nám umožňuje vyšší flexibilitu při uchovávání dat, nicméně při hledání musíme prohledat více míst, avšak prohledávání nad možnými indexy, kde se data mohou nacházet, se provádí paralelně. Počet asociativních cest je kompromisem mezi složitostí zá-



Obrázek 2.3: Schéma paměťového subsystému procesoru. Zdroj [5].

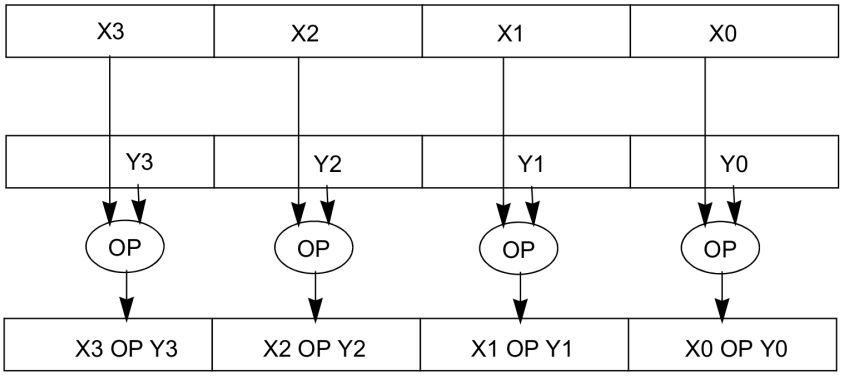
pisů, rychlosti čtení a energetické náročnosti. Přístup do L1 paměti trvá 4 takty v případě jednoduché adresy a 5 taktů v případě složeného výpočtu adresy. O tuto paměť soutěží všechny vlákna v rámci procesorového jádra.

V případě, že procesor udělá požadavek na data, která nejsou v L1 paměti, musí se požadavek propagovat do vyšších úrovní cache a nebo do hlavní paměti. Požadavky, které jsou právě uspokojovány se uchovávají v k tomu určené mezipaměti (Line Fill Buffers - LFB). Tato mezipaměť pojme pouze 10 položek, což je poměrně málo, vezmeme-li v úvahu, že nám jediná vektorová instrukce může vygenerovat i 16 požadavků na data, která se nenacházejí v žádné vyrovnávací paměti, a poté všechny další požadavky čekají i několik desítek cyklů, než se začnou obsluhovat. S rostoucí paralelizací na exekuční části procesoru přibývá úloh, které právě na toto omezení naráží.

Procesory architektury Skylake server obsahují 1 MiB privátní L2 paměti pro každé fyzické jádro a v průměru 1,375 MiB L3 sdílené paměti na každé jádro. Doba přístupu je 14 cyklů do L2 paměti a mezi 50 a 70 cykly do L3 paměti.

2.2 Vektorové instrukce na CPU - SIMD model

SIMD (Single Instruction Multiple Data) je exekuční model, který nad vektorem dat provádí jednu instrukci. Jak je znázorněno na obrázku 2.4, operandy instrukce jsou celé vektory, kde se pro příslušné prvky ze zdrojových vektorů `X` a `Y` provede operace `OP` a výsledek je uložen do cílového vektoru.



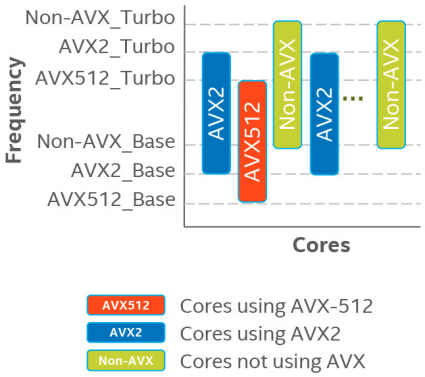
Obrázek 2.4: Znázornění SIMD instrukce o dvou operandech. Zdroj [12].

První instrukční sadou obsahující vektorové operace na rodině procesorů x86 bylo rozšíření 3DNow! od firmy Advacend Micro Devices [15]. Zanedlouho poté konkurenční firma Intel přidala do svých procesorů podobnou sadu s kódovým označením SSE (Streaming SIMD Extensions). Postupně k této sadě přibývala nová rozšíření v podobě SSE2, SSE3, SSSE3 a SSE4. Všechna tato rozšíření pracovala s maximálně 128 bitovými registry. Registry o šířce 256 bitů přinesla s nástupem procesorů Intel Core až instrukční sada AVX [6]. Nedlouho po ní Intel vydal sadu AVX2, jež přidává tzv. *Gathery* (shromažďovače), což jsou instrukce, které dovolují provádět čtení prvků vektoru z různých míst v paměti. Od roku 2017 jsou pak k dispozici vybrané procesory s instrukční sadou AVX-512, která umožňuje používat až 512 bitové vektorové operandy.

Co se týká očekávaného urychlení, tak to se v žádném případě nerovná velikosti vektoru. Jednak proto, že se liší základní i přetaktované frekvence obvodů zpracovávající vektorové a nevektorové operace, jak je znázorněno na obrázku 2.5. Nejvyšší frekvence procesor používá při zpracovávání skalárních a ostatních instrukcí, pro instrukční sadu AVX2 používá o něco nižší frekvence a pro instrukce ze sady AVX-512 pak používá nejnižší taktování. Kromě vyšší spotřeby elektrické energie, vyšší takt procesoru produkuje výrazně více vyzářeného tepla a vzhledem ke složitosti a objemu obvodů zpracovávajících vektorové instrukce,

není čip schopný udržet dlouhodobě vysoký takt, aniž by se přehřál (a nebo chladil extrémními způsoby).

Kromě frekvencí se liší i počet exekučních portů, kde se mohou operace spouštět, viz obrázek 2.2 - rozdělení exekučních jednotek, kde je vidět, že jednoduché celočíselné skalární operace mohou spouštět jednotky 0, 1, 5 a 6. Vektorové instrukce sady AVX2 se mohou spouštět pouze na portech 0, 1, 5 a instrukce ze sady AVX-512 dokonce jenom na portu 0 a 5, což samozřejmě ovlivňuje možnou propustnost jednotlivých instrukcí.



Obrázek 2.5: Vizualizace rozsahů frekvencí jader v závislosti na vykonávaných instrukcích. Zdroj [13].

2.2.1 Použití vektorových instrukcí

Obecně existují dva způsoby, jak využívat v programech vektorové instrukce. Prvním, přímočarým způsobem je automatická vektorizace překladačem. Z pohledu překladače se jedná o netriviální úlohu, jelikož využití vektorových instrukcí nemusí být za všech okolností výhodnější, tudíž překladač musí spočítat (odhadnout), jestli se vektorizace vůbec vyplatí. To jednak může udělat se znalostí latencí a propustnosti jednotlivých instrukcí, ale tyto čísla samotná mohou dávat nepřesné odhady. Jednak by překladač musel vědět, jak přesně se budou spouštět instrukce mimo pořadí. Za druhé si musí být vědom poměrně velkého okolí programu, jelikož vektorové jednotky je potřeba *zahřát,* aby podávaly plný výkon (viz [11] příklad 15.20). Tato fáze přípravy trvá asi 56000 cyklů (architektura Skylake), do té doby jsou vektorové instrukce asi 4,5 krát pomalejší [8].

Další problémy, se kterými se překladač při automatické vektorizaci může potýkat jsou zarovnání dat v paměti (týká se hlavně AVX2), falešné závislosti, závislosti na hodnotách z předchozích iterací, malá délka vnitřní smyčky, vnořování smyček, podmíněné opuštění smyčky, nepřímý přístup do paměti a některé další [11].

Druhým způsobem, jak použít vektorové instrukce, je možnost ruční (explicitní) vektorizace za použití vkladačů (*intrinsic*). Jedná se o funkce, které jsou

v překladači mapovány (většinou) na jednu konkrétní strojovou instrukci - za předpokladu, že je dostupná na cílové architektuře. V jazycích, určených pro systémové programování (C, C++, Rust) se vkladače běžně vyskytují a zdaleka se nemusí jednat jen o vektorové instrukce, ale například také instrukce dopředného načtení dat do cache (prefetch), počítání počátečních nul v binární reprezentaci čísla a další. Při použití ruční vektorizace je dnes stále možné psát výrazně efektivnější kód, než který je generovaný překladačem.

Vektorové instrukce najdou využití hlavně při výpočetně náročných úlohách (CPU bound jobs). Co se týká úloh, kde je výkon omezený propustností pamětí, tak tam nám (zatím) vektorizace tolik nepomůže, jelikož velice brzy narazíme na limit maximálních výpadků L1 cache, resp. velikosti Line Fill Bufferu. V současné době ještě nejsou opravdu paralelním způsobem implementovány instrukce pro shromažďování (Gathers) a rozptyl (Scatters) a jejich výhoda zatím spočívá převážně v tom, že data nemusíme přesouvat z a do vektorových registrů, abychom je přečetli a nebo uložili na různá místa v paměti. Pokud lze výpočet vhodně vektorizovat, nabízí vektorové instrukce jednodušší dekódovací logiku (rozpočteno na prvek vektoru) a paralelní zpracování operací, což navyšuje celkovou propustnost systému, která by se bez vektorizace musela zvyšovat frekvencí procesoru, což má své fyzikální limity a nebo počtem jader v procesoru, což je cesta, kterou výrobci procesorů následují, nicméně ani tato cesta není bez omezení.

2.3 Vektorová instrukční sada AVX-512

Tato instrukční sada nepřišla rovnou do procesorů, ale nejdříve ji obsahovaly pouze výpočetní koprocesory Intel Xeon Phi. Do procesorů se dostala až se serverovou architekturou Skylake (tedy Skylake X, EP/EX, SP a W). AVX-512 je souhrnné označení pro celou skupinu instrukčních sad. Zde jsou vyjmenované dosud oznámené podsady [19]:

AVX-512F Foudation (základ), jedná se o primární sadu pro práci s vektory o šířce 512 bitů. Dovoluje vektory kombinovat na úrovni 32 nebo 64 bitových čísel, poskytuje základní matematické a logické operace, komparace, práci s pamětí, komprese a dekomprese. Kompresí se rozumí souvislé uložení pouze některých prvků z původního vektoru. Dekomprese je pak opačná operace. Oproti AVX2 ještě přidává opak ke shromažďovačům - rozptýlení (*Scatter*), které dovoluje zapsat prvky vektoru na různá místa v paměti v jediné instrukci.

AVX-512CD Conflict Detection (detekce konfliktů), přidává testování prvků ve vektoru na shodu a počítání počátečních nul v bitové reprezentaci prvků vektoru.

AVX-512ER Exponential and Reciprocal, výpočet exponentů pro základ 2 a výpočet převrácené hodnoty a odmocniny převrácené hodnoty.

AVX-512PF Prefetch (dopředné načtení), dovoluje načítat data z hlavní paměti do L1 nebo L2 cache.

AVX-512VL Vector Length Extensions (rozšíření délky vektorů), přidává funkce dostupné na registrech o 512 bitech na kratší 256 nebo 128 bitové registry.

AVX-512DQ Doubleword Quadword (rozšíření instrukcí pro dvojslova a čtyřslova), přidává další operace nad 32 a 64 bitovými prvky. Jedná se například o násobení 64 bitových čísel s 128 bitovým mezivýsledkem, kombinace vektorů se 128 bitovými prvky, číselné konverze a výběr extrémů.

AVX-512BW Byte and Word, přidává manipulaci na úrovni bytů a nebo slov. Jedná se o jednoduché matematické a logické operace, komparace, konverze, výpočty extrémů a permutace nad 8 nebo 16 bitovými prvky.

AVX-512IFMA52 Integer Fused Multiply Add (sloučené násobení a součet nad celými čísly), přidává tyto operace nad 52 bitovými čísly. Mezivýsledky jsou 104 bitové a v závislosti na použité instrukci se uloží horní nebo spodní část čísla.

AVX-512VBMI Vector Byte Manipulation Instructions (manipulace na úrovni bytů), rozšiřuje možnosti o permutace a bitové posuny na bytové úrovni.

AVX-512_4VNNIW Vector Neural Network Instructions Word variable precision (instrukce pro neuronové sítě s variabilní přesností), dovoluje počítat skalární součin čtyř prvků.

AVX-512_4FMAPS Fused Multiply Accumulation Packed Single precision (spojené násobení, sčítání a následná akumulace čtyř prvků v jednoduché přesnosti jedinou instrukcí).

AVX-512VPOPCNTDQ Spočítá bity v prvcích vektoru, které jsou nastaveny na 1.

AVX-512VPCLMULQDQ Umožňuje násobení 64 bitových čísel bez přenosu řádů.

AVX-512VNNI Násobení a součet 8 a 16 bitových čísel v plovoucí desetinné čárce v jedné instrukci.

AVX-512GFNI Afinní transformace Galoisových těles a jejich inverze.

AVX-512VAES Nabízí vektorovou implementaci pro šifrování a dešifrování šifrou AES (Advanced Encryption Standard).

AVX-512VBMI2 Přidává bitové posuny, komprese a dekomprese 8 nebo 16 bitových prvků ve vektoru.

AVX-512BITALG Dovoluje počítat jedničky v binární reprezentaci nad 8 nebo 16 bitovými prvky vektoru. Dále pak umožňuje vybírat bity z vektoru, které se adresují pomocí jiného bytového vektoru.

Tyto podsady nejsou vzájemně výlučné a některé se částečně překrývají. Například původní koprocesory Xeon Phi architektury *Kings Landing* obsahovaly pouze sady AVX-512F, AVX-512CD, AVX-512ER a AVX-512PF. Pozdější architektura koprocesorů *Kinghts Mill* přidala ještě AVX-512_4FMAPS, AVX-512_4VNNIW a AVX-512VPOPCNTDQ.

Současné procesory založené na serverové architektuře Skylake obsahují podsady AVX-512F, AVX-512CD, AVX-512BW, AVX-512DQ a AVX-512VL. Oznámené procesory architektury *Cannonlake*, které mají vyjít v průběhu roku 2018, budou mít oproti architektuře Skylake navíc ještě podsady AVX-512IFMA a AVX-512VBMI. Zbylé podsady se očekávají v procesorech *Ice Lake*.

2.3.1 Rozšíření registrů

Úplně nové možnosti algoritmické optimalizace nám dává 32 adresovatelných vektorových registrů o šířce 512 bitů. Konkrétně se jedná o registry `ZMM0-ZMM31`. Jenom adresovatelné registry ze sady AVX-512 nám tedy dohromady poskytují 2048 bytů prostoru ve vektorových registrech. Otázkou pouze zůstává, jestli máme algoritmy připravené využít tuto možnost. Rozšíření vektorových registrů na 32 adresovatelných je zpětně kompatibilní se staršími instrukčními sadami, které mohou tento prostor částečně adresovat pomocí rozšířených registrů `XMM16-XMM31` a `YMM16-YMM31`.

Oproti AVX2 ještě na čip přibyly bitové maskové registry `k0-k7` s tím, že registr `k0` je konstantní a značí masku samých jedniček. Tyto masky mají s podsadou AVX-512BW 64 bitů, mohou se tedy odkazovat na úroveň konkrétních bytů. Nad těmito specializovanými registry lze spouštět základní logické operace a jejich obsah lze přesouvat mezi obecnými registry. Maskové registry se samozřejmě využívají i jako cílové registry vektorových komparací a nebo také na výběr znaménkových bitů ze všech prvků vektoru. Maskované instrukce na instrukční sadě AVX2 využívaly celý vektorový registr, odsunutí do samostatných registrů je tedy vítaným krokem.

Většina instrukcí ze sady AVX-512 využívá masku proto, aby spustila operaci pouze nad některými prvky vektoru. Do jisté míry se tím lze vyhnout nepredikovatelným skokům v programu, které v kritických částech programu přináší značnou režii. Pokud je například provedení *obou* větví programu rychlejší, než vyhodnocení dle nesprávně predikované podmínky, lze spustit obě větve výpočtu, ale každou část s opačnou maskou.

Kapitola 3

Koncepty agregovaní dat

V této kapitole budou představeny aspekty agregování dat, důležité z pohledu následné ruční vektorizace. Uvedeme základní datové struktury, které se využívají při agregování dat. Dále si zhodnotíme současný stav exekuce dotazů z pohledu kompilace nebo interpretace, se zaměřením hlavně na výhody kompilace. Asi nejdůležitějším aspektem zpracování dat je organizace struktur v paměti a následné paradigma přístupu k vlastnímu zpracování. Právě ve způsobu zpracování je z pohledu vektorizace nová příležitost pro optimalizaci, kterou si v této kapitole nastíníme. Dotkneme se i vztahu maskovaných vektorových instrukcí a jejich návaznosti na konzistenci výsledků.

3.1 Užívané datové struktury

Při zpracovávání dat, hlavně při jejich agregování se hojně využívají určité datové struktury. V této práci se budeme zaměřovat na B+ stromy a hashovací tabulky, konkrétně se budeme zabývat jejich vektorizací.

B+ stromy se v databázových systémech používají primárně jako datová struktura pro indexy. Tyto stromy ve vnitřních uzlech ukládají pouze klíče, hodnoty, nebo odkazy na hodnoty, jsou ukládány až v samotných listech. Návrh těchto stromů reflektuje vnitřní architekturu blokových zařízení, na kterých se, zpravidla při použití v databázích, indexy nacházejí. Ze zařízení se načítá celý uzel rovnou s několika odkazy na následující uzly a tím pádem je jednoduché používanou část indexu držet v paměti, část na disku a šetřit celkový počet přístupů k pomalejšímu médiu. Dotazy, které požadují pouze data, jež jsou zároveň klíči indexu, lze obsloužit pouze čtením z tohoto indexu.

Hashovací tabulky se při zpracování dat používají pro vyhodnocení agregač-

ních dotazů a pro některé strategie spojování tabulek, např. strategie HASH JOIN. Při této strategii se menší množina dat z jedné tabulky uspořádá do hashovací tabulky a následně se projde množina záznamů z druhé tabulky a data z této množiny jsou zapsána do právě vytvořené mapy. V této práci se spojováním záznamů zabývat nebudeme, vektorizované hashovací tabulky budeme používat pouze pro agregace záznamů.

3.2 Kompilace vs. interpretace agregačních funkcí

V dnešních databázových systémech je běžné, že jsou dotazy zpracovávány určitou formou interpretace. Ať už se jedná o specializované virtuální stroje a nebo za běhu (runtime) zřetězené volání funkcí. To s sebou přináší velkou režii. Jak zkoumá [3] v případě databáze PostgreSQL, při volání agregačních dotazů se stále dokola provolává několik funkcí. Ve zmíněné publikaci je vše ukázáno na příkladu výkonnostních testů TPC-H [1], konkrétně na dotazu Q1:

```
SELECT l_returnflag, l_linestatus,
       SUM(l_quantity) AS sum_qty,
       SUM(l_extendedprice) AS sum_base_price,
       SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
       SUM(l_extendedprice*(1-l_discount)*(1+l_tax)) AS charge,
       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'
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus;
```

Tento dotaz agreguje hned několik sloupců a s některými z nich dělá před samotnou agregací jednoduché matematické operace. Data jsou nejdříve vybrána pomocí sekvenčního průchodu (sequential scan), spočítány agregace a následně provedeno řazení. Celkem se vybírá necelých třicet milionů záznamů a pro vyhodnocení tohoto dotazu je potřeba provolat hned několik funkcí. Některé funkce

[1] TPC (Transaction Processing Performance Council) je organizace vydávají sadu výkonnostních testů pro databázové systémy. TPC-H pak odkazuje na dataset, který testuje výkonnost dotazů potřebných pro různá obchodní rozhodování. Více informací na http://www.tpc.org/tpch/default.asp.

se volají pro každý vybraný řádek a některé funkce se volají pro každou požadovanou agregaci - pro tento dotaz osmkrát pro každý řádek. Volání funkcí přináší jistou režii, například vkládání a rušení rámců funkcí na zásobníku, ukládání a vyčítání registrů, nepredikovatelné skoky v kódu, které vedou na přerušení zpracovávání na pipeline procesoru a dalším nárokům na instrukční mezipaměť. Jak je vidět na obrázku 3.1, který ukazuje počet volání jednotlivých funkcí, dotaz můžeme rozdělit do dvou skupin. V první skupině, označené ①, se jedná o funkce volané pro výběr dat z tabulky pomocí podmínky a druhá skupina (②) funkcí zařizuje agregace.

Autoři publikace se rozhodli, že zefektivní spouštění dotazů za pomoci kompilace dotazu při běhu (Just In Time - JIT). K tomu používají části z překladače LLVM[2]. Logika optimalizace vyhodnocování dotazů zůstává stejná, pouze se před spuštěním naplánovaného dotazu provede jeho kompilace. Je tedy možné funkce do sebe v binárním kódu vnořovat (function inlining), jelikož je známé jejich okolí, a tím ušetřit režii spojenou s voláními, popřípadě využít dalších pokročilých optimalizací překladače. Ovšem kompilace dotazů samozřejmě zabere nějaký čas. Co se týká dotazů ze sady TPC-H, tak kompilace zabrala maximálně 40 ms, což pro analytické dotazy trvající několik sekund nebo i minut, neznamená výraznou zátěž. Pokud bychom brali v úvahu i dočasné ukládání již zkompilovaných dotazů, tak za předpokladu, že budeme provádět, z exekučního pohledu stejné dotazy, tato režie bude spojená pouze s prvním takovým dotazem. Přesně takovým způsobem se uchovávají i exekuční plány, které ovšem trvá sestavit řádově kratší dobu. JIT kompilace přinesla u již zmíněného dotazu urychlení zhruba 36 % a u ostatních publikovaných dotazů urychlení o 5 až 58 %. Tato práce se však

# Calls	Function	
29 447 787	ExecProcNode	
5	├─ExecAgg	
29 447 776	│ ├─advance_aggregates	
235 582 212	│ │ └─**ExecProject**	
58 895 550	│ │ └─ExecMakeFunctionResultsNoSets	
	│ │ ├─ExecEvalConst	
	│ │ ├─ExecEvalScalarVarFast	
	②	│ │ ├─float8pl	
	②	│ │ ├─float8mul	
	②	│ │ ├─slot_getattr	
	②	│ │ └─ ···	
235 582 208	│ ├─advance_transition_function	
88 343 328	│ │ └─float8_accum	
235 582 212	│ ├─slot_getsomeattrs	
29 447 776	│ └─LookupHashTableEntry	
176 686 640	│ └─ ··· slot_getattr	
29 447 777	└─ExecScan	
29 999 799	└─**ExecQual**	
29 999 794	└─ExecMakeFunctionResultNoSets	
	├─ExecEvalConst	
	①	├─ExecEvalScalarVarFast	
	①	├─date_le_timestamp	
	①	└─slot_getattr	

Obrázek 3.1: Počet volání interních funkcí databáze PostgreSQL při vyhodnocování dotazu TPC-H Q1, převzato z [3].

[2]Low Level Virtual Machine - sada nástrojů používaných pro překlad zdrojových kódů https://llvm.org/.

JIT kompilací dotazů nezabývá, to by bylo nad její rozsah, nicméně používá variantu předkompilace dotazů před spuštěním programu. Ovšem nic nebrání tomu, aby se práce v budoucnu rozšířila a dotazy se překládaly za běhu.

3.3 Modely zpracování a organizace dat v paměti

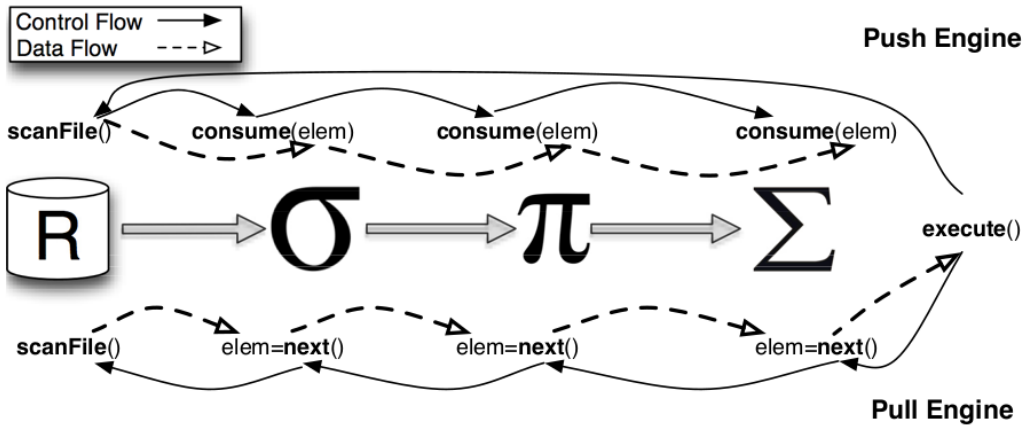
Existuje hned několik možností, jak přistupovat ke zpracovávání dat v paměti. S výběrem modelu zpracování se váže i způsob organizace dat v paměti. Data můžeme ukládat buď po řádcích a nebo po sloupcích, to nám určuje, jakou formu přístupu k datům máme k dispozici. V databázích se přístupy mohu kombinovat. Většinu dat můžeme uložit po řádcích a například indexy, které stojí nad jedním, případně více sloupci, uložíme ve speciálních strukturách do vedlejších souborů. Některé systémy zpracování dat jsou schopné, pokud počítají dotazy jenom nad indexem, takové dotazy uspokojit pouze za pomoci čtení tohoto indexu.

3.3.1 Zpracování po n-ticích

Často používaným modelem zpracování dat je tzv. zpracování *tuple-at-a-time*, které zpracovává data po n-ticích. Data jsou většinou uložena po řádcích a každá operace nad nimi produkuje výstupní proud (stream) n-tic, který vstupuje do další zřetězené operace. Existuje model, který využívá abstraktních volání *open, next* a *close*, které umožňují otevření nového proudu, iteraci přes prvky tohoto otevřeného proudu a následné ukončení a uvolnění prostředků - finalizace. Volání funkce *next* si vždy od předchozí transformace vyžádá data, jedná se tedy o tzv. *pull* model. Architektura takového systému je jednoduchá, robustní a dovoluje rozšiřování. Nevýhodou tohoto přístupu je, že zřetězené funkce, které se vzájemně provolávají musí být implementovány jako virtuální, což s sebou přináší jistou režii, které se lze vyhnout kompilací dotazů - viz sekce 3.2.

Opakem pull modelu je tzv. *push* model, kde jsou data aktivně předávány vždy volající stranou, implementovatelný pomocí návrhového vzoru Visitor. Příkladem takového zpracovávání, používaného v databázích, je například tzv. Vulcano model [9]. Výhodou tohoto modelu je snadná paralelizace operací, realizovatelná pomocí předávání dat mezi vlákny. Rozdíl mezi push a pull modelem je znázorněn na obrázku 3.2, kde je vidět vstupní soubor dat `R` a tři transformace σ, π a Σ. V případě push modelu (Push Engine) je spuštěním zpracování n-tic volána rovnou funkce `scanFile`, která provolá první funkcí σ a předá ji příslušná data. Tato funkce provede požadovanou operaci a provolá další funkci. Oproti tomu v případě pull modelu (Pull Engine) je první volání inicializováno

logicky poslední spuštěnou funkcí, která si od předchozí funkce vyžádá data. Pokud volaná předchozí funkce data nemá (nejedná se o funkci obsluhující vstupní soubor), tak zase provolá svoji logicky předchozí funkci. To se opakuje, dokud se narazí na funkci, která je schopna data přečíst nebo vytvořit a poté poslat funkci, která si o ně zažádala. Tyto modely mají samozřejmě rozdílnou charakteristiku grafu zpracování dat s tím, že pull model má obecně složitější exekuční graf a s tím spojenou režii nepredikovatelných skoků [17].



Obrázek 3.2: Znázornění rozdílu mezi push a pull modelem zpracování dat. Zdroj [17].

3.3.2 Zpracování po sloupcích

Column-at-a-time je označení pro dávkové zpracovávání dat po sloupcích. To vyžaduje, aby i data byla uložena po sloupcích. Oproti řádkovému uložení má tento přístup výhodu, že ušetří operace potřebné k přístupu na datové médium, což se pozná hlavně na rotačních discích, kde jsou přesuny hlaviček nejvíce limitujícím faktorem. Potřeba čtení je nejvýrazněji redukována při operacích, které nepotřebují všechny sloupce z tabulky, na rozdíl od řádkového uložení se totiž přečtou pouze ty sloupce, se kterými opravdu dotaz pracuje. Výhodou je ušetření režie spojené s provoláváním funkcí pro každý jednotlivý prvek v databázi, protože funkce jsou automaticky volány nad celými sloupci, avšak pokud je dat více, než se vejde do mezipaměti, výpočet je blokován přístupem do hlavní paměti. Ukládání po sloupcích ještě umožňuje efektivní kompresi dat, jelikož data z jednoho sloupce obecně vykazují nižší entropii než data napříč celým řádkem.

Celý tento koncept je optimalizovaný převážně pro čtení a analytické dotazy. Nevýhodou ukládání dat po sloupcích je náročnější vkládání nových záznamů, jelikož jeden řádek tabulky musí být zapsán na více fyzických lokací. Příkladem databází, které využívají zpracovávání po sloupcích, jsou C-Store a MonetDB.

3.3.3 Zpracování po vektorech

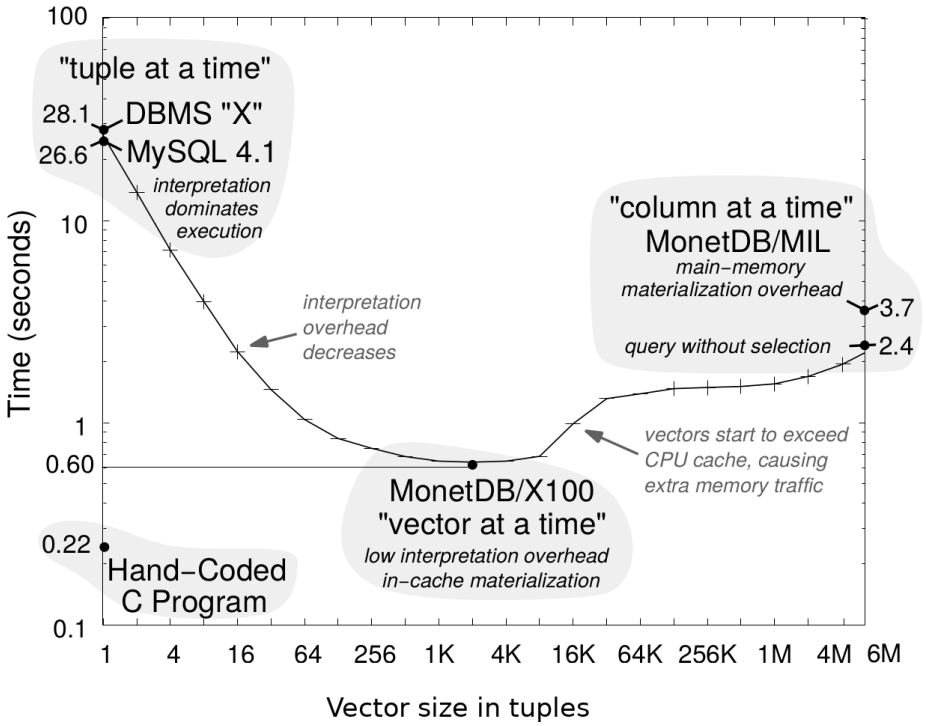
Historie zpracovávání dat po sloupcích sahá až do 70. let minulého století, nicméně až v roce 2005 byl představen nový koncept, který již nezpracovává celé sloupce, ale snaží se optimalizovat čtení z pamětí umístěných přímo v procesoru tím, že data zpracovává po vektorech, které se vejdou do cache procesoru (*vector-at-a-time)*. Tento koncept rozšířil databázi MonetDB a byl vydaný pod názvem MonetDB/X100 [20]. Autory k takové implementaci vedl nepoměr časů exekuce dotazů v databázích a ručně napsaným kódem vykonávajícím stejné operace. Tento nepoměr byl způsoben jednak režií při interpretaci dotazů, čekání na hlavní paměť, znemožnění optimalizací překladačem a následnému nízkému počtu vykonaných instrukcí za takt.

Ve studii byl publikovaný i nákres (obrázek 3.3), značící velikosti vektoru, který byl použit na mezivýpočty a závislost na době zpracování. Čím menší byla velikost vektoru, tím více se projevovala režie s interpretací kódu a výsledný čas se přibližoval databázím pracujícím s modelem tuple-at-a-time, konkrétně k času zpracování v databázi MySQL a jedné, blíže nespecifikované komerční databázi (autoři neuvádí o jakou databázi se jedná, pravděpodobně jim to neumožňovalo licenční ujednání). Dokud se data vešla do cache, čas zpracování se zmenšoval a poté již stoupal, až k výchozí hodnotě zpracování celého sloupce.

Tento přístup umožňuje lépe využívat optimalizace na procesoru, mezi které patří spekulativní exekuce, paralelizace na úrovni instrukcí a zároveň překladači umožnit automatickou vektorizaci. Zároveň utilizuje přístup do rychlých mezipamětí přímo na procesoru, jejichž efektivní využití podmiňuje rychlost dnešních algoritmů.

3.3.4 Zpracování po vektorových registrech

V této práci je ovšem představeno nové pojetí vektorového zpracování dat, které se od známého zpracovávání vector-at-a-time liší tím, že pro předávání dat mezi operacemi nevyužívá cache procesoru, ale data předává rovnou ve vektorových registrech v procesoru. Takové zpracovávání budeme označovat *zpracování po vektorových registrech* (*vector-register-at-a-time*). Samozřejmě je možné předávat



Obrázek 3.3: Závislost velikosti vektoru při zpracování v databázi MonetDB/X100 a době zpracování dotazu. Srovnání s dalšími databázemi a ručně napsaným programem v C, který provádí stejné operace. Zdroj [20].

data v registrech pouze mezi takovými operacemi, které samy o sobě nevyžadují přístup do paměti. Například vyhodnocení matematické operace a následná jednoduchá agregace všech hodnot se spokojí pouze s akumulátory, které lze uchovávat v registrech. Naproti tomu, pokud agregujeme data podle nějaké závislé hodnoty, musíme využít hashovací tabulku, kterou již musíme držet v paměti.

Pro exekuci dotazů, které si předávají data v registrech je již nezbytně nutné, aby se před vykonáváním provedla kompilace, jelikož režie potřebná pro interpretaci by degradovala výkon tak, že by tato optimalizace pozbyla významu. Implementace tohoto konceptu je podrobně rozebraná v kapitole 5.

3.4 Paralelní zpracování, konzistence a maskované instrukce

Při paralelním zpracování dat, nejen v databázích, dochází ke konkurentnímu přístupu k datům. S tím je spojená potřeba deklarace *modelu konzistence,* na který se uživatelé mohou spolehnout. Různé databáze garantují odlišné konzistenční modely, avšak v relačních SQL databázích je běžné garantovat alespoň tzv. *snapshot isolation.* Toho je docíleno většinou pomocí tzv. *multiversioningu* (Multiversion Concurrency Control - MVCC), který dovoluje spouštět několik transakcí zároveň, aniž by se transakce ovlivnily a to celé tak, aby nebylo potřeba provádět zbytečně permisivní zamykání. MVCC je nejčastěji implementováno tak, že ke každé hodnotě v databázi existuje časová značka transakce, která tuto hodnotu zapsala. V případě zápisu nové informace dojde k zápisu další hodnoty a příslušného identifikátoru zápisové transakce. Pokud chce nějaká jiná transakce přečíst hodnotu v dané buňce, přečte takovou hodnotu, která odpovídá poslední hodnotě z již úspěšně dokončené transakce, jež byla dokončena v čase před začátkem čtecí transakce. Každá transakce má tedy vlastní, oddělený, pohled na data. Jelikož se implementace této funkcionality neobejde bez výkonnostní penalizace, některé NoSQL databáze používají uvolněnější modely konzistence a tím jsou schopny nabízet lepší škálovatelnost, distribuovatelnost a vyšší výkon.

Není mi známá žádná databáze, která by používala masku u vektorových instrukcí, aby řídila výběr dat se správnou časovou značkou. Proč to tak současné databáze neimplementují bude jednak tím, že vnitřní logika je moc náročná, aby ji dnešní překladače vektorizovaly a také tím, že až AVX-512 je první instrukční sada, kde jsou maskované instrukce stejně výkonné jako jejich nemaskované protějšky (kde se stejně používá implicitní maska) a zároveň první sada, kde jsou specializované maskové registry, takže není potřeba plýtvat vektorovými. Zajisté to také bude tím, že by se musely provést zásadní změny v současném návrhu napříč celou databází. Avšak řešení za pomoci maskovaných instrukcí, má výhodu, že se vyhne nepredikovatelným skokům a když už data máme v registrech, výběr dat pro zpracování můžeme řešit pouze nastavením masky a při předávání dat mezi funkcemi přímo v registrech nepřináší navíc žádnou režii. Ve chvíli, kdy jsou data mezi kroky výpočtů předávány skrze paměť, můžeme použít kompresní instrukce, které do paměti uloží pouze validní data. Kompresní instrukce má samozřejmě menší propustnost než jiné jednodušší instrukce, avšak to je zanedbatelné vzhledem k faktu, že se jedná o práci s pamětí.

Masku dále můžeme využít i k vypínání výpočtů pro hodnoty `NULL`, pokud například budeme držet binární vektor nenastavených hodnot.

3.5 Deklarativní zápis požadavků

V SQL databázích je samozřejmé, že se využívá deklarativní zápis dotazů. Tento přístup má výhodu v expresivnosti ku délce zápisu, důležitější však je, že je možné provádět dotazy optimalizované. Optimalizace může spočívat v již zmíněné kompilaci, ale hlavně v sestavení exekučního plánu na základě informací známých za běhu. Například očekávaná selektivnost podmínek, optimalizace joinů dle mocnosti tabulek a nebo výběr a následné použití vhodného indexu.

S příchodem NoSQL databází se od tohoto trendu začalo upouštět ve prospěch nestandardizovaných různorodých rozhraní. Nicméně, po letech vývoje se ukazuje, že i NoSQL databáze dospívají do stavu, kdy se i do nich, díky své univerzálnosti, dostává jazyk SQL. Pro demonstraci uveďme například projekty jako Apache Phoenix, Apache Impala a Google Spanner [1]. Tím nám vznikají tzv. NewSQL databáze, které podporují jak přístup za pomoci SQL, tak pomocí vlastních rozhraní, a dále nabízejí dobrou škálovatelnost a požadovanou úroveň konzistence.

Kapitola 4

Dostupné nástroje pro zpracování dat v paměti

Tato kapitola popisuje kritéria výběru vhodného nástroje, pro který by se mohly implementovat agregační funkce za použití vektorových instrukcí. Dále pak rozebírá jednotlivé nástroje, které připadají v úvahu.

Dostupných nástrojů je celá řada, některé jsou známější a používanější, některé méně. I vzhledem k rychlosti, s jakou tyto nástroje vznikají, se jedná o oblast, kde je složité pojmout všechna dostupná řešení. V této kapitole se proto budeme zabývat jenom některými z nich.

4.1 Kritéria výběru

Přístup k implementaci vektorových instrukcí prezentovaný v této práci se liší od většiny konceptů používaných v současných programech. Nástroj by měl proto mít dostatečně jednoduchou a modulární implementaci, aby do něj bylo možné zapracovat agregační funkce realizované pomocí vektorů a/nebo by měl nabízet vhodné rozhraní k datům, které dovolí nad daty uloženými v patřičném nástroji spouštět agregační funkce.

Kvůli snadnému napojení a porovnatelnosti výsledků ručně vektorizované a skalární (automaticky vektorizované) implementace je dalším požadavkem, aby agregační software byl napsaný v jazyku C nebo C++. Jistě by šlo od tohoto požadavku ustoupit ve prospěch jiných kompilovaných jazyků bez automatické správy paměti (například Rustu), ale to by cílový jazyk musel korektně podporovat všechny potřebné vkladače a nebo by se musela řešit kompatibilita na binární úrovni.

Nástroj by měl mít otevřený zdrojový kód a licenci, která umožňuje rozšiřování programu.

4.2 MonetDB

MonetDB je renomovaná databáze ve skupině databází, kde jsou data organizována po sloupcích. Vznikla na akademické půdě a její původní počátky sahají až do 90. let minulého století. Šířená je pod svobodnou licencí Mozilla Public License 2.0. Při zpracování dotazů MonetDB nejprve převede dotaz do tvz. *MonetDB Assembly Language*, který reflektuje strukturu virtuální stroje uvnitř databáze a možnosti vnitřních procedur. V této reprezentaci dotazu se také provede celá škála optimalizací, ať už se jedná třeba o optimalizaci spojování tabulek, výběr exekučního plánu na základě cenových metrik a nebo optimalizaci celkového toku programu. Při zpracování dat tato databáze využívá tzv. *Binary Association Table*, což je prakticky minimální jednotka uložených dat, nad kterou se provádí vlastní výpočty. Pro některé typy analytických dotazů nabízí MonetDB, ve srovnání s ostatními databázemi, velice efektivní způsob zpracování.

Tato databáze by byla vhodným kandidátem pro zasazení vektorového zpracování dat v registrech, ale jak ukázal projekt MonetDB/X100, jedná se o objem práce odpovídající rovnou několika dizertačním pracem. Dalším limitujícím faktorem, je použití virtuálního stroje při vyhodnocování dotazů. Přístup by se musel upravit tak, aby dovoloval dopředu kompilaci dotazů.

4.3 MonetDB/X100 - VectorWise

Z databáze MonetDB změnou exekučního modelu vznikla odvozená verze MonetDB/X100. Tato změna spočívala ve vektorovém přístupu ke zpracování dat - viz sekce 3.3.3 a publikace [20]. Tento přístup byl natolik úspěšný, že se z původního čistě výzkumného projektu stal komerční produkt vydaný pod názvem VectorWise. Tato databáze pak v průběhu let zvítězila[1] v rychlosti vyhodnocení zátěžových testů TPC-H a to pro velikosti testovacích dat 100 GB, 300 GB, 1 TB a 3 TB. Společnost vlastnící databázi VectorWise byla později koupena společností Actian a přejmenována na *Actian Vector*. Pro využití v distribuovaných systémech pak ještě vznikl odvozený produkt *Actian Vector in Hadoop*.

Vzhledem ke komerční licenci této databáze, není možné navázat na tuto práci

[1] http://www.tpc.org/tpch/results/tpch_advanced_sort.asp

a změnit model zpracování po vektorech na model zpracování po vektorových registrech.

4.4 Relační SQL databáze (PostgreSQL, MariaDB)

Pro úplnost uveďme i běžně používané relační databáze. Mezi nejčastěji používané svobodné databáze patří neodmyslitelně PostgreSQL a MySQL/MariaDB. Obě tyto databáze mají shodný exekuční model - ukládají a zpracovávají data po řádcích. Oba zmíněné projekty jsou však vyspělé, pokročilé a v produkčním nasazení otestované databáze. Velikosti jejich repozitářů odpovídají jejich robustnosti a dostupné funkcionalitě, např. samotný PostgreSQL bez dalších rozšíření obsahuje přes jeden milion řádků zdrojového kódu. A právě díky rozsáhlosti a řádkovému zpracování dat se ani tato skupina databází nehodí pro implementaci zpracování agregačních dotazů ve vektorových registrech v rámci diplomové práce.

4.5 NoSQL databáze

V posledním desetiletí se v hojné míře začaly objevovat NoSQL databáze. Oblíbenost získaly hlavně díky možnosti škálování a práci s nestrukturovanými daty. Jsou však často nasazovány i na projekty, kde se jejich výhody nemohou úplně projevit a často by bylo lepší použít ověřenou relační databázi. Většina z nových NoSQL databází je napsána v jiném jazyku než C/C++ a proto jsou z výběru pro další rozšiřování v rámci této práce vyřazeny. Pro příklad uveďme několik známých NoSQL databází napsaných v programovacím jazyku Java: Apache Hbase, Apache Cassandra a Elasticsearch. Existuje samozřejmě i celá škála NoSQL databází napsaných v jiných programovacích jazycích, např. Redis, Riak a Hypertable. Většina těchto databází jsou tzv. klíč-hodnota úložiště (Key-Value storage) a na agregace se nezaměřují.

4.6 ClickHouse

Jedná se o databázi vytvořenou společností Yandex, což je společnost vyvíjející internetový vyhledávač, takže škálovatelnost a propustnost jejich řešení je na prvním místě. ClickHouse je svobodná databáze napsaná v C++, do které by šlo zasadit minimálně vektorizované zpracování části agregačních dotazů, jelikož kód

svojí strukturou na automatickou vektorizaci míří, avšak nahrazení celé logiky spouštění dotazů ve vektorizovaném prostředí, které by bylo potřeba pro úplné vytížení vektorových instrukcí, by bylo značně komplikované.

4.7 SQLite

SQLite [10] je relační databáze, která se většinou připojuje k programu jako knihovna. Nejedná se tedy o standardní vazbu klient - databázový server, jež je častěji využívána pro komunikaci s databází. Zdrojový kód je celý napsaný v jazyce C a jeho první verze byla vydána již v roce 2000. Jedná se o stabilní, pečlivě otestovaný[2] nástroj, který má široké uplatnění a to od vestavěných systémů, po mobilní zařízení až k serverovým nasazením. Zdrojový kód je licencován jako volné dílo (*public domain*), je tedy právně možné na dílo navázat nebo ho jinak modifikovat.

Vnitřně SQLite zadané příkazy přeloží do byte-kódu, který pak interpretuje specializovaný virtuální stroj (Virtual DataBase Engine), který který byl vyvinut přímo jako součást SQLite. Posílání příkazů do virtuálního stroje je plně otázkou vnitřních subrutin programu a propojené aplikace by s virtuálním strojem nikdy neměly interagovat napřímo.

Tato databáze je zajímavá i tím, že veškerá data (kromě dočasný dat řídících transakce) jsou v jediném souboru. Tento soubor je pak přenositelný mezi 32 bitovými a 64 bitovými systémy i systémy s různou endianitou.

Pro svoje cílové použití a vzhledem ke skutečnosti, že přístup prezentovaný v této práci se výrazně liší od všech současně používaných přístupů, byl pro manipulaci s daty a pro porovnání výsledků vybrán právě tento nástroj.

[2] SQLite má 100% pokrytí větvení testy, miliony testových případů a mnoho dalších automatických testů, viz https://www.sqlite.org/testing.html.

Kapitola 5

Implementace agregačních funkcí

Následující kapitola obsahuje rozbor implementace samotných agregačních funkcí pomocí vektorových instrukcí. Zaměříme se na problematiku zpracování dat po vektorech, využití masky při filtraci záznamů a vyhodnocování aritmetickologických výrazů. Podrobně si rozebereme implementaci jednoduchých agregací i agregací dle závislé hodnoty, se kterými souvisí i implementace speciálních ručně vektorizovaných hashovacích tabulek. Kapitola projde celý tok programu při zpracování dotazu, včetně závěrečné fáze převodu výsledků na řádky a k nim příslušné skalární hodnoty.

5.1 Koncept implementace zpracování po vektorových registrech

Abychom mohli zpracovávat data po vektorových registrech, resp. předávat mezivýsledky operací přímo v registrech, musíme převrátit pohled na návrh architektury systému. Základní jednotkou, se kterou můžeme manipulovat, se stává 512 bitový registr. V závislosti na šířce používaného datového typu tak máme k dispozici mezi 8 a 64 prvky. A zde narážíme na první problém. Pokud bychom manipulovali s datovými typy o různých šířkách, budeme muset vyřešit rozdílný tok programu. Například pokud budeme zpracovávat zároveň čísla s jednoduchou a zároveň s dvojitou přesností, pro čísla s dvojitou přesností budeme muset operace rozdělit na dvě a vykonat pro tato data dvojnásobek instrukcí. To by ovšem celý problém nevyřešilo, jelikož by bylo ještě potřeba kombinovat různé velikosti masek vektorových instrukcí. Jistě by šla vytvořit nějaká abstrakce, která by nás od této nízkoúrovňové záležitosti odstínila, avšak vytvoření takovéto abs-

trakce by nebylo jednoduché, vzhledem k různorodosti instrukcí a počtu jejich operandů. Proto byla v rámci této práce implementována pouze funkcionalita nad jedním datovým typem - nad 64 bitovým znaménkovým celým číslem, které používá pro všechny celočíselné operace i vybraná SQLite. Nicméně návrh, jak by generátor pro různé datové typy mohl vypadat, je představen v sekci 5.7.

Jak již bylo zmíněno v sekci 3.2, pro efektivní použití jmenované optimalizace je nutné, aby byl agregační kód zkompilován. Myšlenka, realizovaná v této práci, je taková, že se vygeneruje vektorizovaný kód pro agregační funkce, který se následně přeloží a optimalizuje překladačem a výsledný program bude připravený provádět agregační dotazy. Požadavek na vektorizaci je pro vývoj zásadní. V dnešní době ještě nemáme překladače, které by zvládly efektivně vektorizovat netriviální kód, kterým jsou například zřetězené agregační funkce. Aby tedy byl zajištěn výsledný efektivní strojový kód, musí vývoj probíhat převážně pomocí vkladačů, které kód tvoří neintuitivní a zvyšují celkovou dobu vývoje. Ve chvíli, kdy bude k dispozici překladač, který by zvládl zkompilovat zde publikované algoritmy zapsané skalárně (alespoň ve srovnatelné efektivnosti výsledného kódu), velká část této práce zastará a bude moci být nahrazena pokročilým překladačem.

5.2 Příprava SQL požadavku

Jelikož již existuje ověřený způsob zápisu agregačních požadavků, kterým je deklarativní jazyk SQL, pro zápis dotazů byl vybrán právě tento dotazovací jazyk. Nejprve je SQL dotaz rozparsován, na což je využívána knihovna pro zpracování SQL z databáze Hyrise [7]. Zmíněnou knihovnou je vytvořen strom, reprezentující tento dotaz. Při průchodu tohoto stromu je pak sestavován kód umožňující spouštět vlastní agregace.

Struktura generovaného kódu se dělí na čtyři základní bloky. Prvním blokem je inicializační část, kde se nachází kód pro načtení požadovaných sloupců z tabulky, inicializace agregačních funkcí a výstupního objektu. Druhým blokem je vnitřní smyčka iterující přes hodnoty sloupců. První důležitou operací ve vlastní smyčce je naplnění vektorových registrů záznamy z paměti. Načítání probíhá pomocí vektorových instrukcí, přičemž se využívá skutečnosti, že data jsou v programu uložena zarovnaná na 64 bytů, takže jsou rovnou v programu použity instrukce, které čtou data z paměti zarovnaně. Vzhledem ke čtení 64 bytů zároveň totiž nezarovnané čtení musí nutně číst data ze dvou linek cache, jelikož jedna linka vyrovnávací paměti první úrovně má právě 64 bytů. Následně se nad daty provede filtrace a další operace s maskou, která určuje, jaká data jsou

pro výpočet vůbec relevantní. Na konec tohoto bloku se přidávají i volání akumulátorů agregačních funkcí, jenž provádí vlastní agregace. Posledním blokem je finalizační část zajišťující transformace agregovaných výsledků na řádky. Typickým příkladem je konečná suma hodnot z akumulačního vektorového registru, kde částečně agregovaná data v průběhu výpočtu zůstávala pouze v příslušných prvcích vektoru.

Všechny tyto vygenerované části jsou uloženy do souboru, ze kterého se pak pomocí maker vkládají do kódu, kde se provádí vlastní agregační smyčka. To dovoluje generovat pouze proměnou část kódu a například poslední iteraci smyčky, která již nemusí mít nastavenou celou masku můžeme vložit do kódu vícekrát, pouze s jinou maskou. Vkládání pomocí maker pak má tu výhodu, že se v kódu můžeme jednoduše odkazovat na proměnné, které by se, v případě volání funkcí, musely předávat v rozhraní těchto funkcí, což by komplikovalo dynamickou práci s proměnnými napříč již zmíněnými bloky kódů. Kompilací souboru s vnořenými agregačními bloky vznikne část programu, která je připravena vykonávat agregace, a pokud to úloha dovolí, data budou mezi jednotlivými kroky výpočtu předávána přímo ve vektorových registrech.

Při generování výpočtů je generován kód v C++, který obsahuje buď volání na požadovanou zapouzdřenou funkcionalitu, a nebo přímo obsahuje vkladače, pomocí kterých je zajištěné volání požadovaných strojových instrukcí. Zapouzdření do vkladačů má oproti generování strojového kódu tu výhodu, že můžeme pracovat s proměnnými, kterých máme, na rozdíl od adresovatelných registrů, k dispozici vždy potřebné množství. Překladač se pak postará o vhodnou alokaci příslušných registrů, přičemž se využijí různé heuristiky optimalizace. V případě, že by bylo potřeba zároveň pracovat s více registry, než kolik jich máme na procesoru dostupných, překladač vygeneruje kód, který obsah některých registrů dočasně uloží do paměti, avšak vždy se bude snažit, aby těchto dočasných ukládání a následného vyčítání bylo co nejméně.

Po vygenerování souboru reprezentující části agregačního dotazu a celkového přeložení překladačem je možné tento kód přidat do programu za běhu jako dynamickou knihovnu, pokud se samozřejmě překlad spustí s příslušnými parametry.

5.3 Filtrace záznamů

Jak již bylo řečeno, nespornou výhodou vektorového zpracování dat je možnost výběru dat pro zpracování na základě masky a tím se vyhnout větvení v programu. Pro vyhodnocování logických výrazů, které jsou obsaženy v agregačních

dotazech, se v generátoru skládají příslušné operace nad maskami řídícími vektorové operace. Operace s maskou jsou tedy klíčové pro filtraci relevantních záznamů.

5.3.1 Vstupní maska

Do každé iterace smyčky přichází vstupní maska, která je ve všech iteracích, kromě té poslední, celá zapnutá. Protože data vždy zpracováváme po celých vektorech může se stát, že počet prvků nebude dělitelný velikostí vektoru, a proto jsou v poslední iteraci zapnuté pouze ty prvky, které existují ve vstupních datech. Vzhledem k použití maskovaných instrukcí i při přístupu do paměti, s korektně nastavenou maskou, nenastane neoprávněný přístup do paměti. Tato vstupní maska lze samozřejmě využít i k vynechání smazaných záznamů, pokud bychom před kompakcí dat drželi u smazaných prvků bitový příznak, jako to dělají některé databáze (např. Apache HBase). Jak bylo nastíněno v sekci 3.4, zde by bylo pomocí vstupní masky možné řídit i konzistenci pohledů na data z různých vláken.

5.3.2 Vyhodnocování aritmetických výrazů

Prvním krokem při filtraci dat je vyhodnocení aritmetických výrazů. Konstanty z těchto logických výrazů jsou nastaveny do celého vektoru, pak je možné provádět vyhodnocování aritmetických výrazů vektorově. Pro vyhodnocení podmínky výběru dat (WHERE) jsou implementovány základní aritmetické operace, konkrétně se jedná o sčítání, odčítání, násobení a dělení. S tím, že pro dělení celočíselných 64 bitových operandů neexistují vektorové instrukce a proto je potřeba operandy nejprve převést na hodnoty v plovoucí desetinné čárce, provést operaci dělení a poté převést zpět do celočíselné reprezentace.

Následuje vyhodnocení komparačních operátorů, což je převedeno na vyhodnocení vektorové komparační instrukce s příslušným parametrem, který řídí typ komparační operace. Implementovány jsou základní operace `>`, `<`, `>=`, `<=`, `=` a `!=`.

5.3.3 Vyhodnocování logických výrazů

Po vyhodnocení aritmetických a komparačních výrazů následuje vyhodnocení logických operátorů. Zde se již vychází ze skutečnosti, že stačí provádět pouze operace nad bitovou maskou. Operace logického součtu a součinu lze implementovat přímočaře, pouze pomocí příslušné operace. Složitější situace je s logickou

negací, která by mohla zapnout členy, které jsou vypnuté vstupní maskou, proto je nutné po negaci ještě provést logický součin se vstupní maskou. Implementovány byly základní logické funkce - *AND, OR, NOT.*

5.3.4 Optimalizace maskovaného zpracování

Vypínání výpočtu pro určité prvky na základě masky nemusí být výhodné, pokud má dotaz nízkou selektivitu a následuje výpočet, který pro exekuci potřebuje čas delší, než čas penalizace za špatně predikovaný skok. Může se totiž stávat, že pozdější fáze výpočtu nebudou zpracovávat žádná data, protože bude maska výpočet zcela vypínat. Můžeme ale sestavit heuristickou optimalizaci, která nám odhadne, zda-li by se více vyplatilo zkontrolovat masku, jestli vůbec obsahuje zapnuté prvky a případně předčasně ukončit iteraci, za cenu riskování špatně predikovaných skoků v programu. V zásadě by stačilo spočítat, kolik instrukcí ještě výpočet po filtraci obsahuje, stanovit průměrnou latenci používaných instrukcí a stanovit práh, kdy by se do výpočtu přidalo podmíněné opuštění iterace. Tato optimalizace v programu implementována nebyla, implementována byla o poznání jednodušší heuristika, která podmínku pro předčasné opuštění iterace přidává pouze v případě, jedná-li se o agregační dotaz seskupující výsledky (GROUP BY).

Další nepříjemnou věcí na malé selektivitě dat při vektorovém zpracovávání jsou situace, kdy je vektor zaplněný pouze z malé části. Pak by bylo rychlejší data zpracovávat skalárně, jelikož, jak je popsáno v části 2.2, vektorové instrukce nedosahují při zpracování jediného elementu takového výkonu, jako jejich skalární protějšky. Čeho by ale využít šlo, jsou kompresí instrukce, které by spojily data napříč iteracemi, pokud by se vybraná data z obou (více) iterací vešly do jediného vektoru. Vyžadovalo by to implementaci dalších heuristik a možná i přepnutí mezi modely za běhu v závislosti na datech a selektivnosti dotazů.

5.4 Jednoduché agregace

Poté, co jsou masky nastaveny pouze pro hodnoty, které náš dotaz vybírá, můžeme přistoupit k dalšímu kroku vyhodnocování. Tímto krokem je vyhodnocení jednoduchých agregačních funkcí. Jedná se o provolání metod z příslušných agregačních tříd, jejichž instance byly inicializovány před hlavní smyčkou iterující přes data ze sloupců. Inicializace těchto tříd spočívá v nastavení výchozích (neutrálních) hodnot vektorových akumulátorů, tedy například pro funkci sumy se jedná o samé nuly.

Do agregační funkce, při volání vlastní agregace, vstupuje kromě dat i maska. Uvnitř funkce se pak vyhodnotí akumulační funkce, která je zpravidla realizována jedinou vektorovou instrukcí, do které vstupuje maska, příchozí data a akumulátor a výstup je zase uložen do akumulátoru. Agregace se provede pouze po jednotlivých prvcích, takže pokud např. vybíráme maximum, na každé pozici v akumulačním vektoru je uchovávána pouze maximální hodnota z prvků, které se nacházely na dané vstupní pozici. Následující výpis obsahuje popsaný kód v případě funkce pro výběr maxima, kde je zřetelná i expresivnost instrukční sady AVX-512.

```
inline void aggregate(__m512i values, __mmask8 mask) {
        accumulator = _mm512_mask_max_epi64(
                accumulator, mask, values, accumulator);
}
```

Vzhledem k doporučení pro překladač klíčovým slovem *inline* a velikostí funkce, překladač s největší pravděpodobností kód této funkce přímo vloží do místa jejího volání, takže vstupní parametry zůstanou přímo v registrech. Co se týká akumulátoru, tak záleží, kolik vektorových registrů bylo potřeba pro předchozí výpočty. Pokud je celý agregační dotaz dostatečně jednoduchý, a nebo lze vhodně počet použitých registrů optimalizovat, akumulátor agregační funkce také v průběhu smyčky zůstává v registru, což umožňuje velice efektivní zpracování těchto jednoduchých agregačních funkcí.

Jelikož se v průběhu iterace přes vstupní data uchovávají v registrech pouze částečné agregace, je ještě potřeba po ukončení hlavní smyčky provést výpočet výsledné skalární hodnoty. To je realizováno v rámci finalizační funkce, která pomocí redukčního makra vypočte finální skalární hodnotu. Již se nejedná o jedinou strojovou instrukci, nýbrž o posloupnost pečlivě vybraných instrukcí, které redukční operaci provedou za co nejmenší počet procesorových cyklů.

Implementovány byly základní agregační funkce, jmenovitě `SUM`, `COUNT`, `AVG`, `MIN` a `MAX`. Funkce `COUNT` je efektivně implementována jako suma, do které vstupuje vektor, jenž má jako hodnoty samé jedničky. Funkce `AVG` uchovává ještě akumulátor na počet jednotlivých prvků a při konečném vyhodnocení pak provede výpočet průměru.

5.5 Agregace podle hodnoty - GROUP BY

Implementace agregací podle závislé hodnoty je z principu o poznání složitější než jednoduché agregace. Bohužel při těchto agregacích je potřeba držet v pa-

měti mapu hodnot, podle kterých agregujeme (a k nim příslušné mezivýsledky). Zde se nám nabízí v obecném případě dvě možnosti realizace. Buď pro klíče budeme používat stromovou strukturu a nebo hashovací tabulku. V případě, že potřebujeme mít data seřazená, může se nám vyplatit data ukládat do stromové struktury, která nám umožní prvky vyčíst v požadovaném pořadí. Naproti tomu ukládání do hashovací tabulky nabízí (amortizovanou) konstantní dobu přístupu k prvkům, avšak (obecně) negarantuje žádné pořadí prvků. Je otázkou použitých implementací, jaké datové struktury budou efektivnější, jestli stromová struktura a nebo hashovací tabulka a následné řazení. Speciálním případem by bylo, pokud bychom dopředu znali množinu hodnot, podle kterých agregujeme, pak by šla agregace zajistit v jednoduchém poli. Nicméně pokud by bylo prvků málo, řekněme tolik, kolik by se nám bezpečně vešlo do vektorových registrů, mohli bychom data agregovat přímo v registrech. Rozhodnutí zda-li použít tuto agregační strategii by šlo realizovat za běhu v případě, že by byly uchovávány statistiky o počtu různých hodnot v určitých sloupcích tabulky, což se v některých databázích děje.

V této práci je implementován přístup s ukládáním mezihodnot v hashovací tabulce, která reflektuje vektorový přístup zpracování dat a jejíž implementace je rozebrána v sekci 5.6.

5.6 Ručně vektorizovaná hashovací tabulka

K celému konceptu vektorového zpracování dat jsou potřeba i odpovídající datové struktury, které utilizují vektorové instrukce již v samotném návrhu. Jednou z potřebných datových struktur, kterou pro svoji složitost nejsou dnešní překladače schopné automaticky vektorizovat, jsou hashovací tabulky. V rámci této práce byla implementována hashovací tabulka, která pro zápis přijímá hodnoty ve vektorových registrech. Jedná se o specializovanou datovou strukturu, která se hodí pro tyto agregace. Od standardní hashovací tabulky se liší tím, že uchovává částečně agregované hodnoty, a to vždy pro každou pozici ve vstupním vektorovém registru.

Vnitřní paměť pro klíče a data je rozdělena na bloky o stejné velikosti, kde každý blok odpovídá jedné pozici ve vstupním vektoru, bloků je tedy stejně jako prvků ve vektorovém registru. Paměť v rámci každého bloku je organizována tak, že klíče a hodnoty okupují stejný blok paměti, resp. klíče se nachází na sudých a hodnoty na lichých pozicích. Tím je zajištěna lokalita klíčů a hodnot a je zde větší pravděpodobnost, že data budou načtena z paměti již spolu s klíčem a program na ně nebude muset čekat. Vzhledem k primárnímu účelu, demon-

strace vektorové hashovací tabulky, se do mapy ukládají již rovnou agregované hodnoty a ne ukazatele na řádky a nebo jiné datové struktury, vhodnější pro vektorové zpracování.

Při zápisu vektoru hodnot do tabulky se využívají instrukce, které zpracovávají celý vektor, výpočet hashovací funkce i následný přístup do paměti je zpracováván za pomoci datového paralelismu. Odděleným zápisem do jednotlivých bloků se zbavíme i případné datové závislosti, kolizních prvků, při zápisu celého vektoru hodnot. Pokud bychom předpokládali opravdu malý výskyt kolizních hodnot v rámci jednoho vektoru, agregovali bychom podle velkého množství hodnot a pravděpodobnost kolize by byla malá, bylo by možné použít jednu sdílenou datovou strukturu. Ovšem před vlastním zápisem by se musely kolize detekovat a následně provést agregace nad kolizními prvky a až poté provést vlastní zápis. I na případy této kategorie návrháři nové instrukční sady AVX-512 mysleli, a proto je instrukční podsada AVX-512-CD vybavena instrukcemi `vpconflictd` a `vpconflictq`, které zkontrolují, zda-li vektor obsahuje kolizní prvky. Nicméně, tyto instrukce mají opravdu velkou latenci, pro 32 bitové prvky v 512 bitových registrech má první jmenovaná instrukce latenci 67 cyklů. Druhá jmenovaná instrukce, která pracuje nad 64 bitovými prvky má pro vektory délky 512 bitů odezvu 37 cyklů[1].

Nevýhodou rozdělení do oddělených bloků je, že po provedení všech zápisů a částečných agregací, je ještě potřeba vypočítat výsledné agregace tím, že se vypočte finální agregace pro příslušné prvky napříč všemi bloky.

V rámci předvedené implementace bylo implementováno pouze agregování vkládaných záznamů, resp. byly vygenerovány implementace pro různé agregační funkce a různé datové typy. Změna velikosti vnitřního paměťového bloku a následné přehashování nebylo implementováno, velikost vnitřní paměti se shora omezí v závislosti na očekávaném počtu agregovaných hodnot.

5.6.1 Způsob řešení kolizí hashů

Jak již bylo naznačeno v předchozích odstavcích, jedná se o tabulku s otevřeným adresováním, zbývá ještě zmínit, jakým způsobem je implementováno vkládání prvků s kolizním hashem. Možnosti, které jsem pro implementaci zvažoval jsou lineární zkoušení, druhotné hashování a tzv. Robin Hood hashování [4]. Poslední jmenované hashování si pro hashovací tabulku drží maximální počet přeskoků, který je potřeba vykonat při hledání konkrétního klíče mezi kolizními hashy.

[1] https://software.intel.com/sites/default/files/managed/ad/dc/Intel-Xeon-Scalable-Processor-throughput-latency.pdf

Při zápisu pak hodnotu uloží na pozici po maximálně tomto počtu kolizí. Prvek, který se nacházel na této pozici naopak bude přesunut dále. Tímto mechanizmem je zaručené maximální množství kroků při hledání prvku.

Z pohledu ruční vektorizace náš výběr ovlivní možné způsoby implementací jednotlivých přístupů. Pro vektorové zpracování je značně nevýhodné, aby se pracovalo jenom s několika málo prvky ve vektoru a zpracování ostatních bylo vypnuto maskou. Proto chceme minimalizovat počet instrukcí, které budou spuštěny pouze nad zlomkem prvků ve vektoru. Zde se nabízí alespoň dva způsoby řešení zápisu kolizních dat a minimalizaci nevyužitých vektorových instrukcí. Prvním způsobem je duplikovat vektorový kód skalární implementací a v případě malého počtu zpracovávaných prvků přepnout na exekuci skalárního kódu. Druhým způsobem je, při nenalezeném bezkolizním místě, zbývajících pár prvků odložit vedle do paměti a poté je dávkově zpracovat, až jich bude více. Nicméně, to vyžaduje ukládat k prvkům ještě počet přeskočených pozic a je možné, že se při řešení těchto kolizních dat zase vyskytne malé množství dat, které musejí přeskočit výrazně více kolizních míst. Tento způsob byl v průběhu vývoje vyzkoušen, avšak průběžné výkonnostní testy ukázaly, že se nejedná o vhodný přístup. Jelikož přepínání do skalárního kódu přidává do kódu nepredikovatelné skoky a odložené řešení kolizí nenabízí dostatečně rychlé řešení, rozhodnuto bylo Robin Hood hashování neimplementovat. To by v případě vektorového zpracování kolizí spouštělo mnoho vektorových instrukcí nad pravděpodobně nevyužitými registry.

Zbývá tedy rozhodnout, jestli implementovat lineární zkoušení a nebo druhotné hashování. V [16] je srovnání obou implementací na procesoru (vektorová instrukční sada AVX2) i výpočetním koprocesoru Xeon Phi, kde je obojí implementováno za pomocí podporovaných podsad AVX-512. V publikaci vychází obě řešení na procesoru i výpočetním koprocesoru přibližně nastejno, pokud srovnáváme pouze na jednom typu zařízení. Nicméně je důležité si uvědomit, že instrukční sada AVX2 má mnohem užší registry a při využití shromažďovacích instrukcí (gather) přistupuje na poloviční počet míst v paměti, než v případě použití sady AVX-512, ale počet paralelně obsluhovaných přístupů mimo L1 paměť se mezi architekturami procesorů nezvýšil. Jak bylo uvedeno v sekci 2.1.4, Line Fill Buffer zvládne obsloužit pouze 10 takových přístupů a ze zkušenosti vím, že právě na toto omezení některé implementace algoritmů instrukcemi ze sady AVX-512 narážejí. Koprocesory Xeon Phi mají od procesorů odlišný paměťový subsystém a jinou penalizaci za přístup do paměti, takže z těchto měření nelze vyvodit, jak se bude chovat implementace na procesorech Skylake X. Kvůli minimalizaci přístupů na různá místa do hlavní paměti byl tedy z výběru vyloučen

i přístup druhotného hashování, které by vykonávalo přístupy, jež nejsou možné uspokojit pomocí automatického přednačtení (prefetch) dat.

5.7 Generátory algoritmů využívající vkladače pro různé datové typy

Ruční vektorizace zpravidla poskytuje pouze implementaci pro jediný datový typ, protože se v každé instrukci musí reflektovat datový typ a šířka vektoru. Pokud bychom se podívali na používaný koncept přetěžování operátorů, který je v jazyce C++ oblíbený, zjistíme, že se využívá v případě binárních operátorů. Nicméně pokud bychom chtěli přetěžovat operátory pro vektorové datové typy, museli bychom reflektovat *minimálně* ještě masku k těmto instrukcím, což i pokud by nám to syntaxe jazyka dovolila, v žádném případě by se nejednalo o intuitivní zápis. Druhou možností, jak si ulehčit práci, je obalit jednotlivé vkladače vlastními funkcemi a po jednotlivých datových typech je seskupit dohromady, což by nám částečně umožnilo zapouzdřit problematiku datových typů.

Další možností, jak vyřešit problém pro různé datové typy je vytvoření generátoru zdrojového kódu, který pro příslušné operace a datové typy sestaví názvy vkladačů. Cílem je ulehčit si práci, a proto pro skript na generování zvolíme jazyk, který má triviální syntaxi a co nejjednodušší práci s řetězci. V této práci je představen generátor postavený nad skriptovacím jazykem CoffeeScript, který generuje vektorizované hashovací tabulky, jenž zároveň agregují mezivýsledky. Agregace jsou skriptem také generované, aby se zajistilo zapouzdření funkcionality a zjednodušilo generování exekučního kódu dotazu při zpracování SQL dotazu. Generovány jsou tedy rovnou kombinace dostupných datových typů 32 a 64 bitových znaménkových celých čísel a čísel v plovoucí desetinné čárce a základních agregací (suma, minimum a maximum). Generování pro kratší datové typy a nebo pro typy bez znaménka nebylo implementováno, ale se současnou strukturou by nebyl nejmenší problém generátor o tyto typy rozšířit. Generátor totiž obsahuje definici vlastností jednotlivých datový typů, jak znázorňuje následující část kódu (algoritmus 5.1). Každý používaný datový typ má v obdobném objektu zadefinované příslušné vlastnosti a pak již stačí iterovat příslušnými objekty a ve smyčce používat jednotlivé vlastnosti datových typů. Celý generátor ještě obsahuje vnější smyčku, která pak iteruje nad jednotlivými agregačními operacemi a výsledkem jsou vygenerované hlavičkové soubory obsahující hashovací tabulky, které zapouzdřují požadované agregace. Pro klíče a hodnoty v tabulce se generují pouze stejné datové typy. Rozdílné datové typy pro klíč a hodnotu, pokud by

byla stejná velikost datových typů, by také nebyl problém generovat, ale pokud by byl záměr generovat klíče a hodnoty s různými délkami datových typů, musel by se generátor upravit tak, aby duplikoval instrukce kratšího datového typu.

Algoritmus 5.1 Ukázka definice objektu vlastností pro datový typ desetinného čísla s dvojitou přesností v rámci generátoru v jazyce CoffeeScript.

```
types.push {                        # vložení do pole typů
        native: "double"            # název skalárního typu
        suffix: "pd"                # přípona vkladače
        name: "Double"              # uživatelský název
        avx: "__m512d"              # datový typ vektoru
        vectorLength: 8             # počet prvků ve vektoru
        mask: "__mmask8"            # datový typ masky
        fullMask: "0xFF"            # plně nastavená maska
}
```

5.8 Finalizace požadavku

Poslední fáze, která se vykonává po hlavní iterační smyčce přes všechna data, zajišťuje volání výsledných celkových agregací, případné řazení výsledků a transformaci na řádky. Tato funkcionalita je zapouzdřena ve třídě *Result*. Právě do instance této třídy putují výstupní data jednotlivých agregací nebo rovnou vyfiltrované výsledky. V případě, že je zalimitován počet výsledků, které má dotaz vrátit, tak právě tato třída obsahuje logiku, která vyhodnocuje, zda-li již bylo dodáno maximální množství výsledků. Pokud dotaz obsahuje požadavek na řazení, není samozřejmě možné dopředu omezit počet výsledků. Třída tedy musí kontrolovat tok programu v závislosti na tom jestli dotaz obsahuje požadavek na výsledné seřazení dat a/nebo omezení na počet vrácených výsledků. Aby se ušetřilo rozhodování o následujícím toku programu za běhu, využívá se známého triku, kdy jsou tyto informace předány jako parametry do šablony a následně se vygeneruje takový kód, který již ve strojovém kódu rozhodování o větvení neobsahuje. V tomto případě se zrovna jedná o podmínky, jejichž výsledek se za běhu měnit nebude, takže režie s těmito podmínkami by byla velice malá, ale pokud je možné kód připravit tak, aby program tyto větve kódu vůbec neobsahoval, nemusí se spouštět instrukce na vyhodnocování podmínky a kód celkově bude menší.

Data jsou z poslední transformace vždy předávány po řádcích, jako vektor vektorů. A právě inicializace této výstupní struktury je jediné místo kde se používá klíčové slovo jazyka pro explicitní alokaci na haldě - `new`. Ač není běžnou praxí v moderním C++ tento konstrukt používat, zde použití bylo nutné, jelikož se tato struktura předává ven z knihovny, do které je dotaz zkompilován, musí se tedy předat čistý ukazatel bez automatické správy paměti. Na jiných místech v programu se využívá převážně předávání hodnotou s optimalizací eliminace kopírování dat, tzv. *Return Value Optimization*.

5.8.1 Řazení výsledků

V případě, že dotaz obsahuje požadavek na určité pořadí vrácených dat, data jsou stále uchovávána po sloupcích, neděje se zatím žádná transformace, data se z vektorových registrů pouze přesunou do paměti. Do paměti je také ukládán zvlášť sloupec, podle kterého se má vykonat seřazení dat. Poté, až s finálním voláním požadujícím výsledek dotazu, se seřadí sloupec s daty pro stanovení pořadí. K řazení se využívá funkce z [2], která dovoluje provádět stejné transformace pořadí i nad druhým polem dat. Této funkcionality je využito tak, že druhým argumentem do řadící funkce je vygenerovaná seřazená posloupnost, která po skončení řadící funkce obsahuje informaci o tom, jak přeskládat ostatní sloupce, aby výsledek byl seřazený právě podle hodnot specifikovaného sloupce. V publikaci je bohužel prezentované pouze řazení nad 32 bitovými čísly se znaménkem a 64 bitovými čísly v plovoucí desetinné čárce a kód je velice obsáhlý (soubor se zdrojovým kódem pro řazení celých 32 bitových čísel má přes 6000 řádek) a autoři bohužel nedodali kód, kterým byl tento soubor vygenerován, takže neexistuje jednoduchá cesta, jak algoritmus řazení předělat na 64 bitová čísla. Proto řazení v této práci bylo implementováno pouze nad 32 bitovými čísly, ač se ostatní zpracování děje s čísly o dvojnásobném rozsahu. Po seřazení dat, které jsou stále uloženy po sloupcích, následuje finální transformace výsledků - převod na řádky. A jestliže dotaz limituje počet vrácených řádků, po vyčtení požadovaného počtu položek dle seřazeného indexu se vyčítání ukončí.

5.9 Příklad zkompilovaného dotazu

Pro názornost rozeberme příklad dotazu a jak vypadá jeho forma zkompilovaná do vektorového zpracování za použití vkladačů. Uvažme následující dotaz:

```
SELECT AVG(c/a), SUM(b) FROM table1 WHERE d < 20 AND a > 4
```

Tento dotaz obsahuje výběr dat z tabulky `table1` podle podmínky, která je složená ze dvou komparačních větví spojených logickou spojkou AND. Vybraný sloupec `b` bude dotaz agregovat funkcí `SUM` a sloupce `c` a `a` nejprve podělí a poté spočte jejich průměr funkcí `AVG`. Jak již bylo popsáno na začátku kapitoly, dotaz je rozdělen na tři části - inicializační, tělo hlavní smyčky a finalizační část.

Algoritmus 5.2 Vygenerovaná inicializační část dotazu.

```
// kontrola existence tabulky
if (!datastore.count("table1")) {
        throw std::string("Table␣table1␣not␣found");
}
// kontrola přítomnosti sloupců
if (!datastore["table1"].count("a")) {
        throw std::string(
                "Column␣data_a␣of␣table1␣not␣found");
}
... // kontrola ostatních použitých sloupců - vynecháno

// reference na vektory s hodnotami
auto col_data_a = datastore["table1"]["a"];
...

// načtení velikosti tabulky
size_t tableSize = datastore["table1"]["a"].size();

// inicializace výstupního objektu
// neočekává se řazení
// ani omezení počtu vrácených záznamů
Result<long, false, false> result(0, 0);

// inicializace agregací, průměru a sumy
Avg agg_0;
Sum agg_1;
```

V příkladu vygenerovaní inicializační části - algoritmus 5.2, si můžeme povšimnout kontroly existence požadované tabulky a příslušných sloupců, se kterými dotaz pracuje. Následuje uložení referencí na vlastní data tabulky, inicializace objektu, který vytváří abstrakci nad výstupem a inicializace požadovaných agregačních funkcí.

Algoritmus 5.3 Vygenerované tělo hlavní smyčky pro zpracování dotazu.

```
// tělo hlavní smyčky
// proměnná _i obsahuje aktuální index
// proměnná inputMask pak validitu vstupních dat

// načtení dat do vektorových registrů
// načtena jsou pouze data vybraná vstupní maskou
__m512i curr_a = vectorLoader(inputMask, &col_data_a[_i]);
__m512i curr_b = vectorLoader(inputMask, &col_data_b[_i]);
__m512i curr_c = vectorLoader(inputMask, &col_data_c[_i]);
__m512i curr_d = vectorLoader(inputMask, &col_data_d[_i]);

// vyhodnocení výběrové podmínky
// nastavení prvního konstatního argumentu z podmínky
__m512i constarg_0 = _mm512_set1_epi64(20);

// nastavení masky dle porovnání s konstantním argumentem
__mmask8 mask_1 = inputMask
   & MaskFilter::filter(curr_d, constarg_0, _MM_CMPINT_LT);

// obdobně pro druhý výraz
__m512i constarg_1 = _mm512_set1_epi64(4);
__mmask8 mask_2 = inputMask
   & MaskFilter::filter(curr_a, constarg_1, _MM_CMPINT_GT);

// výsledné spojení masek
__mmask8 mask_3 = maskAnd(mask_2, mask_1);

// vyhodnocení aritmetického výrazu před agregací
// argumenty je před operací dělení nutno konvertovat
__m512i div_0 = _mm512_maskz_cvtpd_epi64(mask_3,
        _mm512_maskz_div_pd(mask_3,
                _mm512_maskz_cvtepi64_pd(mask_3, curr_c),
                _mm512_maskz_cvtepi64_pd(mask_3, curr_a)
        ));

// zápis vektorů a jejich masek do agregačních funcí
agg_0.aggregate(div_0, mask_3 & inputMask);
agg_1.aggregate(curr_b, mask_3 & inputMask);
```

Algoritmus 5.3 ukazuje tělo smyčky, která prochází data tabulky. Na začátku jsou vektorově načteny data z příslušných datových struktur. Při načítání se využívá skutečnosti, že jsou data ukládána zarovnaně na 64 bytů, a proto lze použít instrukce pro zarovnané čtení z paměti. Funkce `vectorLoader` pouze za-

obaluje instrukci `vmovdqa64`, která vyžaduje, aby se čtení z paměti provádělo zarovnané. Po načtení dat do vektorových registrů následuje vyhodnocení výběrové podmínky. V případě tohoto dotazu jsou data ze sloupců `a` a `d` porovnávána s konstantami 20 a 4. Tyto konstanty jsou nastaveny do všech prvků vektorových registrů a následně spolu s načtenými daty vstupují do filtrovacích funkcí, kde je provedena komparace a dle výsledku nastavena maska. Posledním krokem vyhodnocení výběrových podmínek je výpočet logického součinu nad oběma maskami. Výsledkem pak je korektně nastavená maska (zde proměnná `mask_3`), vybírající pouze požadovaná data. Na tomto případu je názorně vidět, jak můžeme mapovat základní operace, používané i při vysokoúrovňovém dotazování, přímo na vektorové instrukce. Po vyhodnocení podmínek následuje vyhodnocení aritmetického výrazu, který vstupuje do prvního z agregátorů. Jelikož se jedná o dělení a dostupná instrukční sada neobsahuje instrukce pro celočíselné dělení, je potřeba nejprve čísla převést na desetinná, provést dělení a výsledek konvertovat zpět na celé číslo. Posledním krokem v těle smyčky je provolání agregátorů, do kterých vstupuje jednak výsledná vypočtená maska a jednak data ze sloupce, resp. vyhodnocený aritmetický výraz.

Algoritmus 5.4 Příklad finalizační části vygenerovaného kódu dotazu.

```
std::vector<long> row;
row.push_back(agg_0.scalarize());
row.push_back(agg_1.scalarize());
result.put(row);
```

Poslední část vygenerovaného dotazu je ukázána v algoritmu 5.4. Jedná se o část, která následuje po hlavní smyčce a která se stará o zápis nového řádku s výsledky agregací do výsledného objektu. Před samotným přidáním hodnoty do řádku je potřeba převést data z jejich vektorové reprezentace v agregátoru do jediné, výsledné, skalární hodnoty. O to se zase postarají agregační objekty, které provolají příslušná redukční makra.

Všechny tyto tři části se pomocí maker vloží do funkce, která se postará o vstup datasetu, vlastní iteraci a správné nastavení masky pro zbylá data po poslední iteraci.

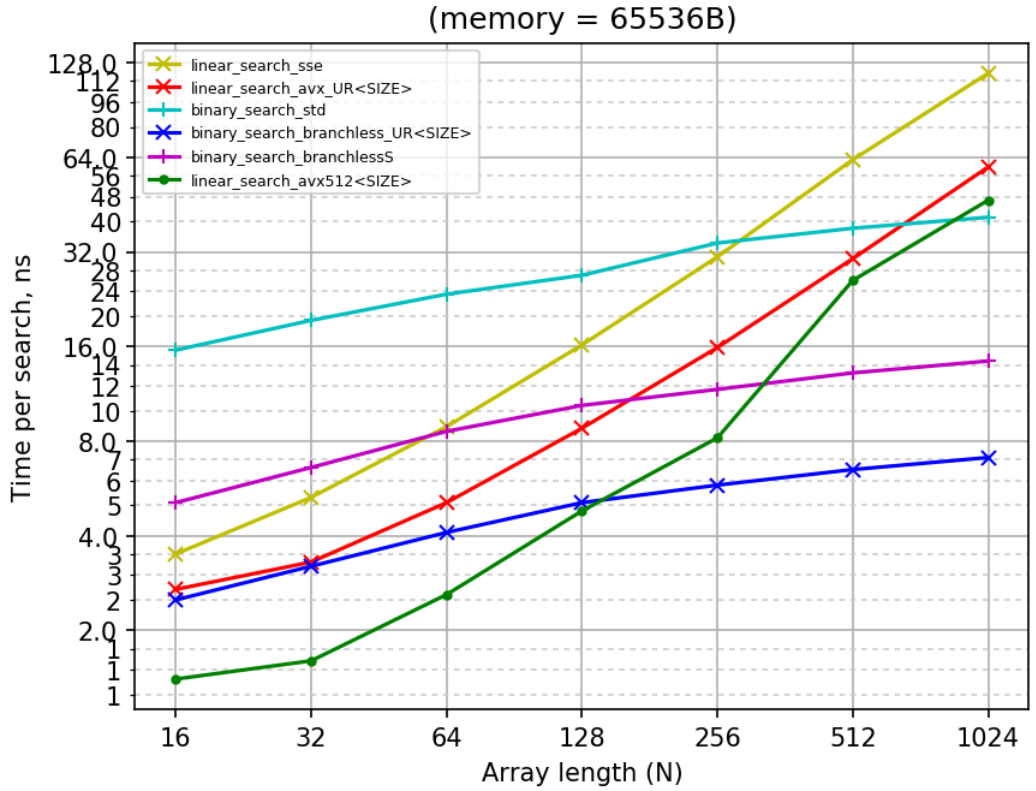
5.10 B+ strom s vektorizovaným vyhledáváním

Při práci s daty je často lepší, zvláště v případě malé selektivity dotazu, využívat pro výběr záznamů index. Správné použití indexů je kritickým místem v případě optimalizace dotazů. V této práci přímo indexy nad daty, které by šlo používat pro optimalizaci dotazů implementovány nejsou. Implementováno je ručně vektorizované vyhledávání nad B+ stromovou strukturou, která je hojně využívána v databázích, jelikož i zde lze vhodně využít vlastnosti nové vektorové instrukční sady.

Jak je známo, vnitřní uzly v B+ stromu obsahují vždy několik prvků, které odkazují buď na další vnitřní uzly a nebo přímo na listy, kde jsou uložená data (nebo ukazatele na data). A právě vyhledávání v rámci jednoho uzlu je místo, které je možné vektorizovat, jelikož se v zásadě jedná o lineární vyhledávání.

Pojďme se podívat na srovnání vybraných druhů vyhledávání v malých polích, jelikož právě vyhledávání v malých polích je speciální případ, kde díky nízkoúrovňové implementaci procesoru není asymptotická složitost algoritmů úplně vypovídající. Ta totiž nezahrnuje penalizaci za nesprávně predikované skoky, levný přístup do L1 cache, dopředné načítání, spekulativní exekuci, paralelizmus na úrovni instrukcí, vektorové instrukce, instrukční mezipaměť pro malé smyčky a další často používané optimalizační techniky v moderních procesorech. Díky těmto optimalizacím se tak může stát, že algoritmus s horší asymptotickou složitostí bude na malém objemu dat pracovat rychleji než jiný, z teoretického pohledu lepší, algoritmus.

Pro potřeby otestování byla rozšířena sada testů, publikovaná v [18], o lineární vyhledávání implementované pomocí vektorových instrukcí ze sady AVX-512. Výsledky srovnání vybraných přístupů ukazuje obrázek 5.1. Na zmíněném obrázku vidíme porovnání šesti implementací vyhledávání. První z nich, označená žlutou barvou je implementace lineárního vyhledávání za pomoci instrukcí ze sady SSE, červenou je vyznačený výkon vyhledávání za pomoci instrukční sady AVX2, kde je ručně odrolovaná vyhledávací smyčka. Tyrkysová barva v grafu značí volání vyhledávací funkce `std::lower_bound` ze standardní knihovny *libstdc++*, modrou barvou jsou pak vynesené časy vyhledávání pomocí binárního dělení, které namísto skoků v kódu používá instrukce podmíněného přesunu (`CMOVxx`) a zároveň vyhledávací smyčka je ručně odrolovaná. Fialová barva značí zase implementaci binárního vyhledávání s `CMOVxx` instrukcemi, avšak bez odrolování smyčky. Posledním, nově přidaným testovaným případem, je lineární vyhledávání za pomoci instrukcí ze sady AVX-512, které je značeno zelenou barvou a které také využívá ručního odrolování smyčky. Tyto srovnávací testy byly



Obrázek 5.1: Srovnání různých vyhledávacích algoritmů na malých datech. Testy pocházejí z [18], v rámci této práce byla testovací sada rozšířena o lineární vyhledávání implementované pomocí vektorových instrukcí ze sady AVX-512. Graf ukazuje závislost velikosti pole a doby vyhledávání.

přeloženy překladačem GCC ve verzi 7.2.0 s parametry `-O3 -funroll-loops -march=native` a spuštěny na stroji s konfigurací vyjmenovanou v sekci 6.2.

Z provedených měření je vidět, že lineární vyhledávání implementované pomocí vektorových instrukcí ze sad SSE a AVX2 je pro malý počet vyhledávaných prvků rychlejší než standardní binární vyhledávání s neodrolovanými cykly. Dále si můžeme povšimnout, že lineární vyhledávání realizované instrukcemi z vektorové sady AVX-512 je až do 128 prvků rychlejší než ostatní vybraná řešení. Nabízí se tedy využít konceptu hybridního vyhledávání a vždy po menších skupinách prvků vyhledávat lineárně a dle nalezeného prvku skočit na nějaký zlomek zbylých prvků. To nám vede na implementaci B stromu, kde v uzlech budeme vyhle-

dávat lineárně, a to za pomoci instrukcí ze sady AVX-512. Cílem je ve výsledku dosáhnout logaritmické časové složitosti s tím, že velikost lineárně prohledávané části nám tvoří základ logaritmické funkce ve výsledné asymptotické složitosti. Pro účely ověření tohoto konceptu byla vytvořena popsaná implementace ručně vektorizovaného vyhledávání ve struktuře B+ stromu, kde vnitřní uzly obsahují 16 prvků, nad kterými se provádí lineární vyhledávání.

Kapitola 6

Dosažené výsledky - testování

V této kapitole si podrobně rozebereme výsledky výkonnostního testování nových vektorizovaných implementací. Výsledky měření jsou porovnávány se srovnatelnými skalárními a nebo automaticky vektorizovanými protějšky. Zaměříme se jednak na celé agregační dotazy, hashovací tabulky, které jsou vyhodnocení dotazů klíčové a také na vyhledávací stromové struktury.

6.1 Softwarové závislosti

Při vývoji bylo využito celé škály nástrojů, které jsou nezbytné pro překlad zdrojových kódů. Které to jsou ukazuje tabulka 6.1 a to včetně použitých verzí. Program je psaný pro prostředí Unixu a testován na operačním systému Ubuntu 18.04, jádru 4.15.0-20. Přenositelnost na ostatní platformy nebyla cílem této práce. Většina softwarových závislostí, kromě knihovny Google Benchmark, lze nainstalovat z oficiálních zdrojů v programu `apt`.

6.2 Hardware použitý pro testování

Pro ověření správnost vracených výsledků i výkonnostní testy byla použita následující hardwarová konfigurace. Sestava obsahuje procesor Intel Core™ i7-7800X, architektury Skylake X, který je složen ze šesti jader taktovaných na 4,2 GHz. Každé jádro má k dispozici 32 KiB 8-cestně asociativní datové mezipaměti první úrovně (L1D) a stejnou konfiguraci pro instrukční mezipaměť (L1I), dále pak 1 MiB L2 paměti. Všechna jádra se pak dělí o sdílenou L3 paměť o velikosti 8,25 MB. Tento procesor samozřejmě obsahuje podporu technologie Hyper-

Název programu/knihovny	Použitá verze
GNU Compiler Collection	8.0.1 20180414 (experimental)
Clang/LLVM	6.0.0-1
Node.js	v8.10.0
CoffeeScript	1.12.7
Google Benchmark	z gitu, revize: 0526755
Google Test	1.8.0-6
STX B+ Tree	0.9-2build2
SQLite	3.22.0-1

Tabulka 6.1: Seznam softwarových závislostí pro překlad zdrojových kódů.

Threading, která byla v průběhu testů zapnutá, ale hlavně tento procesor, v současné době jako jeden z mála procesorů určených pro osobní použití, podporuje novou instrukční sadu AVX-512, konkrétně její podsady F, CD, BW, DQ, VL (viz kapitola 2).

Sestava dále obsahuje čtyři paměťové DDR4 moduly, taktované na frekvenci 2333 MHz a vzhledem k tomu, že zmíněný procesor obsahuje čtyři kanály pro přístup do hlavní paměti, je možné číst ze všech paměťových modulů současně.

6.3 Překlad a spuštění

Překlad a spuštění testovacích programů se provádí pomocí programu `Make`. V adresáři se zdrojovým kódem je připraven soubor Makefile s několika cíli. Jednoduchým příkazem tedy můžeme spustit vlastní překlad a zajistit následné spuštění přeloženého programu. Pro překlad a spuštění testů generátoru nové vektorizované implementace agregačních dotazů zadáme příkaz `make sql-test`. Pro překlad a spuštění výkonnostních testů zadáme příkaz `make sql-perf`. Pokud bychom chtěli vygenerovat hashovací agregační tabulky, zadáme příkaz `make hashtables`. Pro spuštění testů vektorizovaných agregačních hashovacích tabulek zadáme `make hashtables-perf`. Pro překlad a spuštění testů nad vektorizované stromové struktury zadáme `make vbtree-perf`. V případě, že chceme vygenerovat testovací dataset, použijeme příkaz `make dataset`. Pokud chceme překlad provést jiným překladačem než GCC, přidáme jako parametr proměnnou `CXX`, například takto, `make sql-perf CXX=clang++-6.0`.

Veškeré výkonnostní testy jsou překládány s parametry `-Ofast -march=native`.

6.4 Ověření výsledků agregačních dotazů

Správnost výsledků vrácených novou implementací je kontrolována sadou automatických testů. Tyto testy spouštějí dotazy v rámci nové implementace a stejné dotazy také nad databází SQLite. Vrácené výsledky z obou systémů jsou pak vzájemně porovnány a v případě nesrovnalostí je chyba nahlášena. Testovací dotazy jsou rozděleny do čtyř kategorií. První kategorie testů se zaměřuje hlavně na dotazy s aritmetickými výrazy ve výběrové klauzuli. Druhá skupina se zabývá především složením výběrové podmínky pomocí logických operátorů. Předposlední skupina testuje všechny implementované agregační funkce a poslední skupina pak testuje agregace podle závislé hodnoty. Pro snadné testování se využívá knihovna Google Test, která řeší logiku spouštění a vyhodnocování jednotlivých testovacích případů.

6.5 Vyhodnocení výkonnosti agregačních dotazů

6.5.1 Testovací dataset

Pro účely otestování výkonnostních charakteristik systému byl vygenerován testovací dataset, který reprezentuje vybrané případy entropie dat v databázích. Vzhledem k zacílení systému na agregační dotazy byla vygenerována tabulka o 20 milionech záznamů, kde každý záznam obsahuje šest hodnot. Na tyto hodnoty se budeme odkazovat jako na sloupce `a` až `f`. Nejprve si uveďme, jaká data se nacházejí v jednotlivých sloupcích.

a Obsahuje číslo řádku, 0-20 mil

b Dvojnásobek čísla řádku 0-40 mil.

c Pseudonáhodné číslo 0 nebo 1.

d Pseudonáhodné číslo 0 až 99.

e Pravidelné střídání 0 a 1.

f Opakování posloupnosti 0-9.

6.5.2 Sledované dotazy

Následující výkonnostní měření bylo prováděno na dotazech, které jsou ukázány v rámečku algoritmus 6.1. Prvních 6 dotazů obsahuje pouze jednoduché agregace a následujících 6 dotazů agreguje podle závislé hodnoty. Rozeberme si ale každý dotaz zvlášť a uveďme proč byl zařazen do sady výkonnostních testů.

1. Jednoduchá suma součtu dvou sloupců, která musí projít všechny hodnoty ze sloupců `a` a `b`.

2. Počítání rovnou 5 agregačních funkcí nad 5 různými sloupci tabulky.

3. Agregace stejných sloupců jako v případě předchozího dotazu, ale data jsou navíc filtrována s 50% selektivností dle sloupce `c`, který obsahuje pseudonáhodnou hodnotu, takže lze předpokládat, že pokud překladač nepoužije instrukce podmíněného přesunu, bude program velice často penalizován za nesprávně predikovaný skok.

4. Obdobně jako předchozí dotaz, rozdíl je akorát v selektivitě, jenž je v tomto případě pouze 1%. Predikování případných skoků tedy bude v naprosté většině případů správné.

5. Tento dotaz před výpočet vlastních agregací přidává ještě vyhodnocení aritmetických operací. Jedná se o základní operace, které pracují s hodnotami ze sloupců a také demonstrují dělení konstantou.

6. Dotaz zaměřený na vyhodnocení složené podmínky. I zde se předpokládá správné odhadnutí skoku ve většině případů.

7. Tímto dotazem začíná skupina dotazů zaměřených na agregace dle závislé hodnoty. V tomto případě budeme počítat sumu sloupce `a` podle závislého sloupce `b`, můžeme si tedy povšimnout, že pro každou závislou hodnotu z `b` náleží pouze jedna hodnota. Těchto hodnot budeme vybírat přibližně půl milionu.

8. Agregace všech záznamů podle 100 možných hodnot ze sloupce `d`.

9. Agregace všech záznamů podle 2 možných hodnot ze sloupce `e`.

10. Agregace všech záznamů podle 10 možných hodnot ze sloupce `f`.

11. Agregace milionu hodnot do milionu skupin.

12. Výpočet rovnou tří agregačních funkcí, po 10 skupinách.

Všechny tyto dotazy byly postupně spouštěny v databázi SQLite a také v nové implementaci vektorového zpracování, která si data uchovává v paměti organizována po sloupcích. Dále byly vytvořeny implementace provádějící shodné agregace, které ovšem byly napsány v jazyce C++. Záměrem bylo vytvoření takových implementací, které by programátor vytvořil v případě, že by implementoval shodnou funkcionalitu s využitím nástrojů ze standardní knihovny jazyka. Místo agregačních objektů tyto implementace využívají akumulační proměnné, výběr extrémů je zjištěn funkcemi `std::min/max` a pro agregace podle závislé hodnoty se využívá implementace hashovací tabulky `std::unordered_map`. Tyto implementace agregačních dotazů provádějí výpočty nad stejně organizovanými daty jako v případě vektorového zpracování.

Algoritmus 6.1 Dotazy použité pro výkonností měření implementace.

```
1.  SELECT sum(a + b) FROM t1;

2.  SELECT max(a), min(b), sum(c), sum(d), count(e)
      FROM t1;

3.  SELECT max(a), min(b), sum(c), sum(d), count(e)
      FROM t1 WHERE c = 1;

4.  SELECT max(a), min(b), sum(c), sum(d), count(e)
      FROM t1 WHERE d = 1;

5.  SELECT max(a + b), min(b * c), sum(e * f / 51)
      FROM t1;

6.  SELECT max(a + b), min(b * c)
      FROM t1 WHERE a < b AND c > b OR d = 1;

7.  SELECT sum(a) FROM t1
      WHERE c = 0 AND a < 1000000 GROUP BY b;

8.  SELECT sum(a) FROM t1 GROUP BY d;

9.  SELECT sum(a) FROM t1 GROUP BY e;

10. SELECT sum(a) FROM t1 GROUP BY f;

11. SELECT sum(b) FROM t1
      WHERE a < 1000000 GROUP BY a;

12. SELECT sum(a), min(b), max(c) FROM t1
      GROUP BY f;
```

6.5.3 Naměřené časy vyhodnocení agregačních dotazů

Pro potřeby porovnání efektivity jednotlivých řešení byly za pomoci testovacího nástroje Google Benchmark změřeny časy vyhodnocení zmíněných dotazů napříč různými implementacemi. Všechny dotazy byly spuštěny nad vygenerovaným datasetem popsaným v sekci 6.5.1. Tabulka 6.2 obsahuje průměrné časy z 10 spuštění testovaných kódů a sledované směrodatné odchylky. Měřené bylo spuštění dotazů v SQLite a nové implementaci - zpracováním po vektorových registrech, představené v této práci, obojí překládáno překladačem GCC. Dále pak zmíněná tabulka obsahuje průměrné časy podprogramů v C++ vypočítávající ty samé agregace, u kterých je sledovaný čas v závislosti na použitém překladači. Konkrétní verze použitých překladačů jsou uvedeny na začátku této kapitoly - tabulka 6.1. Všechny implementace zpracovávají data pouze v jenom vlákně. Vícevláknové zpracování dat v této práci řešeno nebylo.

	Průměrný čas [ms]				**Směrodatná odchylka [µs]**			
Dotaz	SQLite	GCC	Clang	Vektor.	SQLite	GCC	Clang	Vektor.
1	976	18	17	19	1819	30	37	31
2	2408	313	305	49	7154	57	31	9
3	1665	347	306	49	4398	74	38	7
4	692	300	306	49	375	65	29	19
5	2457	324	320	50	1504	109	50	5
6	1297	312	306	42	320	45	49	282
7	880	411	402	330	532	171	187	188
8	4022	427	325	276	1274	35	215	474
9	3440	428	328	238	1308	69	124	748
10	3225	351	325	407	1097	70	76	3821
12	3794	1005	452	736	1338	104	335	458

Tabulka 6.2: Průměrné časy vyhodnocení agregačních dotazů v jednotlivých implementacích.

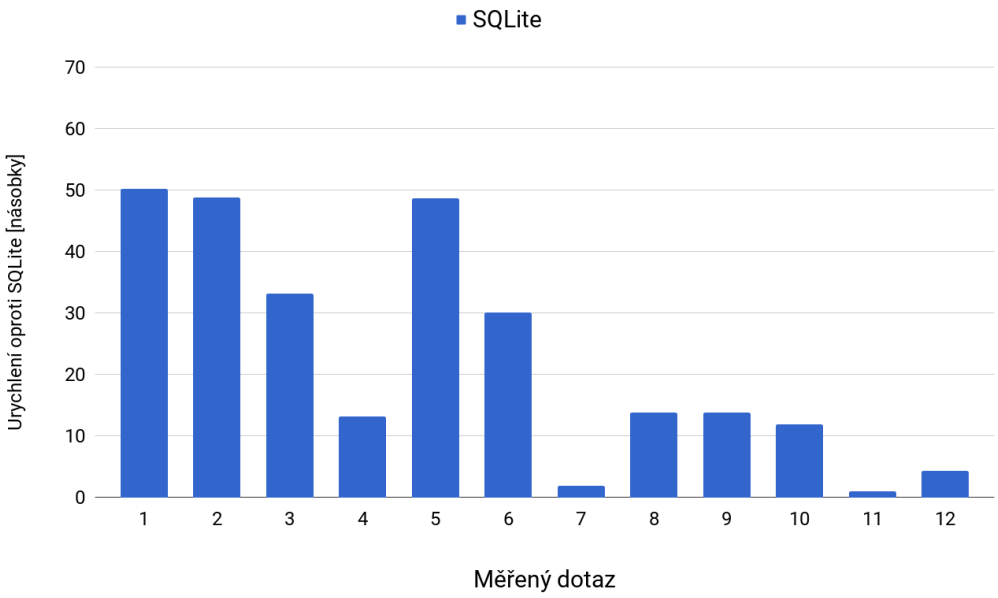
6.5.4 Srovnání s databází SQLite

Začněme s porovnáním dosažených časů oproti databázi SQLite. Zde je nezbytné uvést, že se nejedná o rovnocenné srovnání, nová vektorová implementace totiž řeší jen a pouze vlastní agregaci dat. Neřeší například zamykání dat kvůli vícenásobnému přístupu, kontrolu nullových hodnot ve sloupcích, automatický převod

datových typů, kontrolu přetečení datových typů, ošetřování výjimek a chybových stavů. V časech vyhodnocení SQLite je samozřejmě také čas parsování dotazu a přípravy exekučního plánu. V případě zpracování po registrech je dotaz rozparsován a zkompilován dopředu. Srovnání je z uvedených důvodů pouze orientační.

Data z tabulky 6.2 jsou vynesena do grafu, obrázek 6.1, kde je vizualizované urychlení zpracování agregací po vektorových registrech oproti vyhodnocení dotazů v databázi SQLite. Největší urychlení je dosaženo právě u dotazů, které počítají jednoduché agregace. U těchto dotazů sledujeme přibližně 13 až 50 násobné urychlení. Tyto dotazy jsou v nové implementaci o tolik rychlejší, jelikož kromě samotného vektorizovaného výpočtu nepřidávají při vyhodnocování téměř žádnou režii.

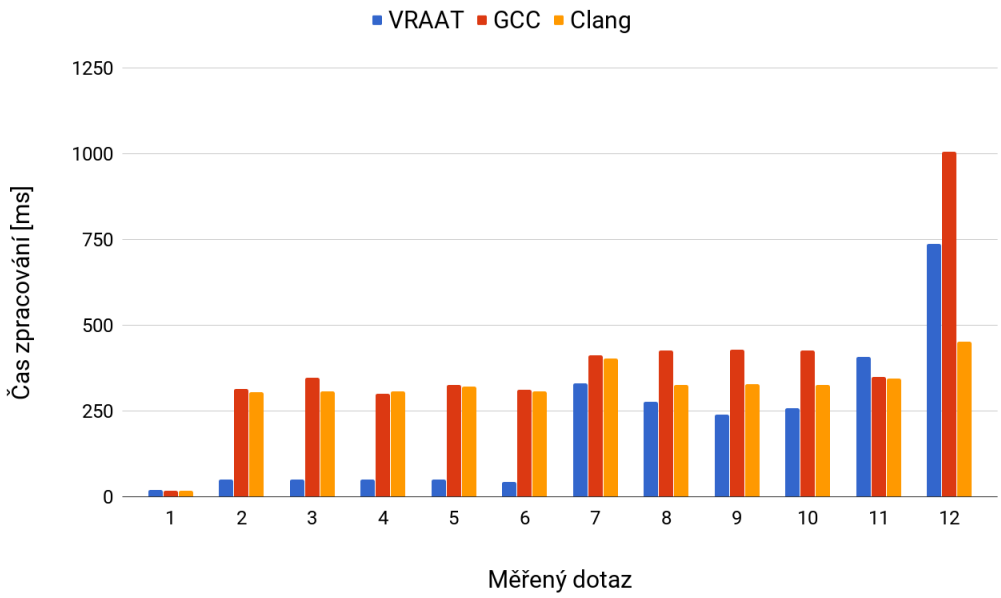
Druhou kategorií dotazů jsou dotazy agregující dle závislé hodnoty (dotazy 7 až 12). Tam již tak výrazné urychlení není. Je to tím, že i vektorová implementace je nucená přistupovat do paměti a v tom už vektorové zpracování, díky hardwarovým omezením, není o tolik efektivnější. Urychlení zde vidíme pouze v rozmezí 2,6 až 14,5 násobku.



Obrázek 6.1: Urychlení jednotlivých agregačních dotazů zpracováním po vektorových registrech oproti vyhodnocení v SQLite.

6.5.5 Srovnání se skalární implementací

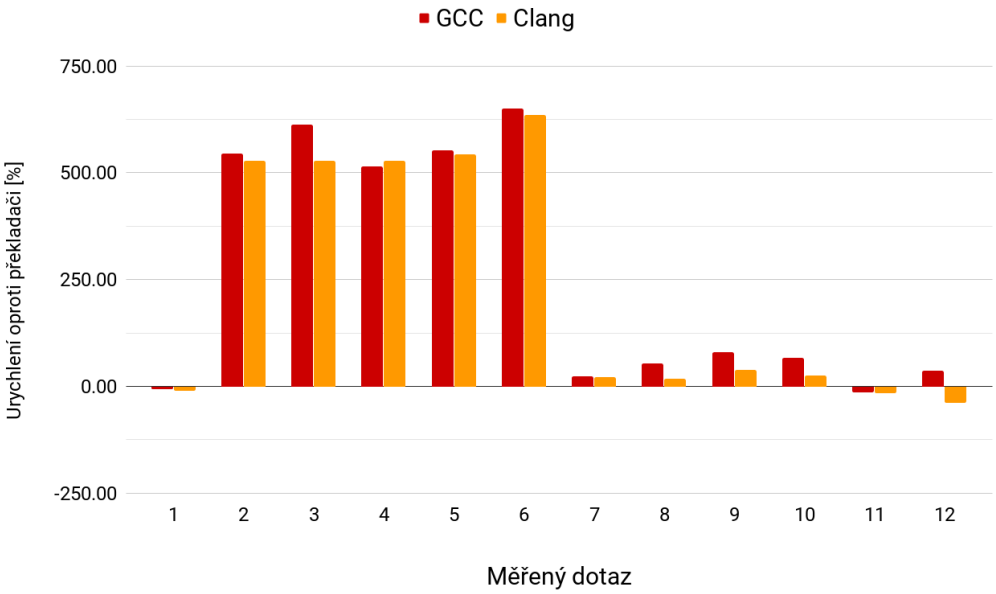
Časy zpracování dotazů v nové vektorizované implementaci a skalární nebo automaticky vektorizovanou implementací, uvedené v tabulce 6.2, jsou vyneseny do grafu - obrázek 6.2. Srovnání vygenerovaného vektorizovaného kódu a skalárního kódu, který je případně automaticky vektorizovaný překladačem, je, oproti srovnání s SQLite, vypovídající. Obě implementace totiž vykonávají totožnou funkcionalitu, měření tedy není zatížené operacemi, které by jedna implementace prováděla a druhá ne. Zároveň pomocí tohoto porovnání můžeme zjistit, zda-li se vynaložená práce pro složité vektorizování dotazu vyplatí anebo ne. V odkazovaném grafu je jasně vidět, že se ve většině případů vyplatilo, aby systém generoval zpracování po vektorech.



Obrázek 6.2: Porovnání časů zpracování jednotlivých testovacích dotazů v nové implementaci - VRAAT (Vector-register-at-a-time) a kódem zapsaným skalárně a přeloženým překladačem GCC nebo Clang.

Z grafu jsou zřetelně vidět některé zajímavé případy dotazů. První zajímavostí je hned první dotaz u kterého je jasně vidět, že pouze pro opravdu jednoduché zpracování dat jsou současné překladače schopné úspěšně vektorizovat. Vektorizace byla ověřena inspekcí překladačem vygenerovaných instrukcí, nejedná

se tedy pouze o rychlejší exekuci skalárních instrukcí. Při úspěšné vektorizaci vidíme dopad minimální režie nové implementace, která v absolutních číslech přidává 1 resp. 2 milisekundy oproti Clangu resp. GCC. Což při zpracování 20 milionů záznamů znamená režii odpovídající pouze pár strojovým instrukcí. Dotazy 2 až 6 se již překladačům vektorizovat nepodařilo a proto je zde největší urychlení - urychlení je samostatně vizualizováno na obrázku 6.3. Původně jsem předpokládal, že GCC verze 8 bude schopno některé tyto případy (alespoň částečně) vektorizovat, jelikož se nejedná o složité výpočty, ani komplikovaný tok programu, a nová verze překladače slibovala výrazné zlepšení automatické vektorizace, ale měření ukazují, že na automatickou vektorizaci netriviálních případů si ještě budeme muset počkat.



Obrázek 6.3: Vizualizace urychlení nové implementace oproti kódu vygenerovaném překladači GCC a Clang.

Co se týká urychlení dotazů 7 až 12, které agregují podle závislé proměnné, tak tam již nová implementace urychlení nepřináší ve všech případech. Dotazy 11 a 12 jsou rychleji vyhodnoceny kódem vygenerovaným překladačem Clang. V případě dotazu 11 je pak rychlejší než nová implementace i kód vygenerovaný překladačem GCC. V případě dotazu 12 a překladače GCC vidíme, že generuje

výrazně pomalejší kód než Clang. Vzhledem k tomu, že se jedná o vývojové vydání překladače (stabilní verze byla vydána až letos v květnu), může tento fakt být způsoben nějakou regresí ve vývojové větvi překladače. Všechny dotazy 7 až 12 agregují podle závislé hodnoty, přistupují tedy do hlavní paměti, kde již vektorizace nemusí být vždy výhodná. Výkonností vektorizovaných hashovacích tabulek se věnuje celá následující sekce 6.6.

Dotaz 11 je v nové implementaci pomalejší, jelikož se agreguje podle velkého množství závislých hodnot, konkrétně se jedná o jeden milion hodnot. V případě dotazu 12 se ukazuje, že nová implementace by mohla obsahovat jinou strategii agregací při agregování většího množství sloupců. Současná implementace totiž pro každý sloupec vytváří vlastní specializovanou hashovací tabulku a pro každý řádek se musí vložit záznam do každé této tabulky. Implementace by šla upravit tak, aby se pro všechny agregace používala pouze jedna tabulka s odkazem na množinu agregovaných hodnot. Poté by i tento kód byl pravděpodobně rychlejší, než překladačem generovaný skalární kód a použité struktury `std::unordered_map`.

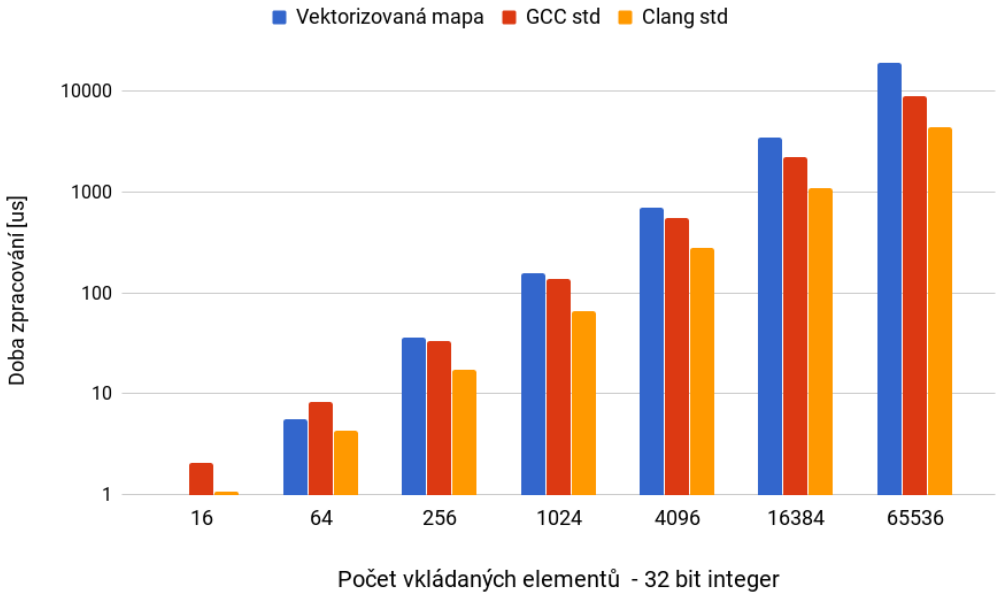
6.6 Vyhodnocení výkonnosti agregačních hashovacích tabulek

V této sekci se zaměříme na samotné hashovací tabulky, které byly použity pro agregaci podle závislé hodnoty. Pro srovnání implementací byly použity vektorizované tabulky uchovávající sumu a příslušné tabulky `std::unordered_map` ze standardní knihovny používané překladačem. Struktura testovacího scénáře je taková, že je vždy do hashovací tabulky vložen určitý počet elementů s tím, že každý element je do tabulky vložen vícekrát. Při prvním vložení se pouze do tabulky vloží nová hodnota a pro všechny další stejné klíče se provede jejich součet s již uloženou hodnotou. Tabulka 6.3 ukazuje naměřené časy v závislosti na počtu prvků v testovacím scénáři, kdy jsou klíče i hodnoty 32 bitová celá čísla.

	Průměrný čas [µs]			**Směrodatná odchylka [ns]**		
Počet elementů	Vektor.	GCC	Clang	Vektor.	GCC	Clang
16	0.98	2.08	1.07	1	1	1
64	5.55	8.36	4.31	6	138	1
256	36	34	17	5	125	4
1024	159	136	67	62	236	23
4096	704	550	283	65	482	29
16384	3498	2207	1092	15235	966	682
65536	19315	8892	4431	63925	3729	10791

Tabulka 6.3: Naměřené časy vkládání a nebo modifikace 32 bitových čísel v hashovací tabulce. Sloupec *Vektor.* odkazuje na vektorizovanou hashovací tabulku.

Obrázek 6.4 ukazuje graf vizualizovaných hodnot z tabulky 6.3. Vidíme, že do 64 různých vkládaných prvků je vektorizovaná implementace rychlejší, než kód vygenerovaný překladačem GCC. Ale pouze do 16 prvků je vektorizovaná implementace rychlejší než kód vygenerovaný překladačem Clang. Pro více prvků se již vektorizovanou tabulku nevyplatí používat. Toto zpomalení je způsobené hlavně tím, že se data načítají pomocí střadačů po celých vektorech a to z různých míst v paměti. Celý vektorový výpočet je tedy zastavený, dokud z paměti nepřijde poslední prvek vektoru. Vzhledem k tomu, že se v případě 32 bitových čísel vybírá z 16 různých míst, je zde velká pravděpodobnost, že alespoň jeden požadavek bude uspokojen až z vyšší úrovně mezipaměti a nebo se dokonce musí čekat i na hlavní paměť.



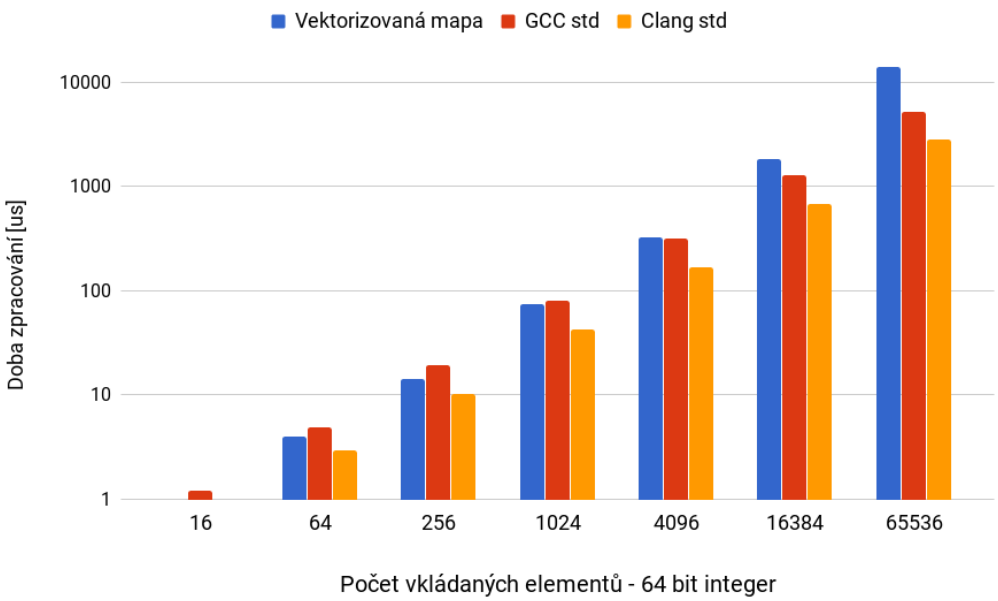
Obrázek 6.4: Vizualizace naměřených časů pro vkládání nebo změnu záznamů hashovací tabulky. Klíče i hodnoty jsou 32 bitová čísla.

Tabulka 6.4 obsahuje data naměřená ze stejného scénáře, avšak klíče i hodnoty jsou 64 bitová čísla. Opakované vložení stejného elementu se však provádí již pouze 8 krát, což odpovídá velikosti jednoho vektoru pro vektorové vkládání do vektorizované tabulky. Vizualizace těchto dat je znázorněna na obrázku 6.5. Pokud pomineme překladač Clang, tak zde je vektorizovaná tabulka rychlejší než kód vygenerovaný překladačem GCC až do více než 1000 vkládaných rozdílných prvků. Toto relativní zrychlení je způsobené tím, že v případě 64 bitových čísel se pracuje pouze s vektory o 8 prvcích, takže se zároveň přistupuje na menší počet různých míst v paměti a je i menší pravděpodobnost, že celý vektor bude čekat na jeden element z vyšší úrovně mezipaměti.

Při očekávaném malém množství kolizí mezi prvky ve vkládaném vektoru by pravděpodobně bylo výhodnější detekovat kolize ve vkládaných prvcích a ušetřit čas za poslední iteraci všech dat a finální agregaci viz sekce 5.6. Poté by se možná vektorizovaný přístup do tabulky oproti `std::unordered_map` vyplatil. Profilace kódu ukázala, že vektorizovaná tabulka s větším množstvím prvků naráží hlavně na omezení maximálního počtu současných přístupů mimo L1 mezipaměť.

	Průměrný čas [µs]			**Směrodatná odchylka [ns]**		
Počet elementů	Vektor.	GCC	Clang	Vektor.	GCC	Clang
16	0.73	1.22	0.85	1	1	90
64	4.04	4.88	2.93	4	9	130
256	14	19	10	5	6	12
1024	75	80	42	35	68	155
4096	321	321	170	14	378	339
16384	1823	1283	679	629	571	573
65536	14019	5198	2791	65345	1454	3493

Tabulka 6.4: Naměřené časy vkládání a nebo modifikace 64 bitových čísel v hashovací tabulce. Sloupec *Vektor.* odkazuje na vektorizovanou hashovací tabulku.



Obrázek 6.5: Vizualizace naměřených časů pro vkládání nebo změnu záznamů hashovací tabulky. Klíče i hodnoty jsou 64 bitová čísla.

6.7 Vyhodnocení výkonnosti vektorizovaného vyhledávání v B+ stromu

Poslední srovnání, které si v této práci ukážeme, je srovnání výkonu vektorizovaného vyhledávání v B+ stromu a dalších dvou používaných stromových struktur. První implementací, se kterou budeme srovnávat je implementace `stx:btree` což, jak název napovídá je implementace B+ stromu. A druhou srovnávanou implementací je `std:map` u které standard jazyka C++ definuje vlastnosti, které odpovídají stromovým strukturám a proto tato implementace vnitřně využívá Červeno-černý strom (Red-Black tree). Zajisté se nejedná o nejvýkonnější implementace[1], nicméně se jedná o implementace které se běžně využívají a proto se s nimi lze udělat vhodné srovnání.

Vytvoření indexu nad vloženými daty, výběr vhodného indexu a následné vyhledávání nad vybraným indexem, nebylo v rámci této práce implementováno. Implementováno a otestováno bylo vektorizované vyhledávání nad vytvořeným indexem.

Pro potřeby srovnání jednotlivých implementací bylo nejprve do stromové struktury vloženo n záznamů (od 0 do n), poté bylo přeházené pořadí prvků v tomto vkládaném vektoru a následně bylo provedeno vyhledávání podle zpřeházeného pořadí. Bylo tedy testováno pouze vyhledávání prvků, které se ve struktuře nacházejí. Výsledky měření výkonnostních testů v závislosti na použitém překladači se nacházejí v tabulce 6.5. Všechny struktury ukládají jako klíče i hodnoty 64 bitová celá čísla.

Pro lepší názornost, byla data z tabulky 6.5 vizualizována do dvou grafů. Obrázek 6.6 znázorňuje urychlení vektorizovaného vyhledávání v B+ stromu oproti zmíněným implementacím. Z grafu je patrné, že pro stromy obsahující méně než 256 prvků je výhodnější používat nevektorizované stromové struktury. To je dáno tím, že do 256 prvků má vektorizovaná implementace zbytečně vysoký faktor větvení. Výpočetní čas tedy není efektivně využit. Pro 256 prvků je faktor větvení 16 ideální a proto rychlostí předčí obě srovnávané varianty. Pro 1024 a 2048 prvků pak je zase implementace `stx::btree_map` rychlejší. Stejná situace se pak opakuje ještě pro cca 2 miliony prvků. Ve srovnání se strukturou ze standardní knihovny na tom vektorizovaná implementace vychází o poznání lépe. Ač jsou i pro větší počty elementů ve stromu patrné jisté výkonnostní výkyvy, vždy se však jedná o alespoň čtyřnásobné urychlení, pokud budeme uvažovat pouze struktury s více než tisíci prvky.

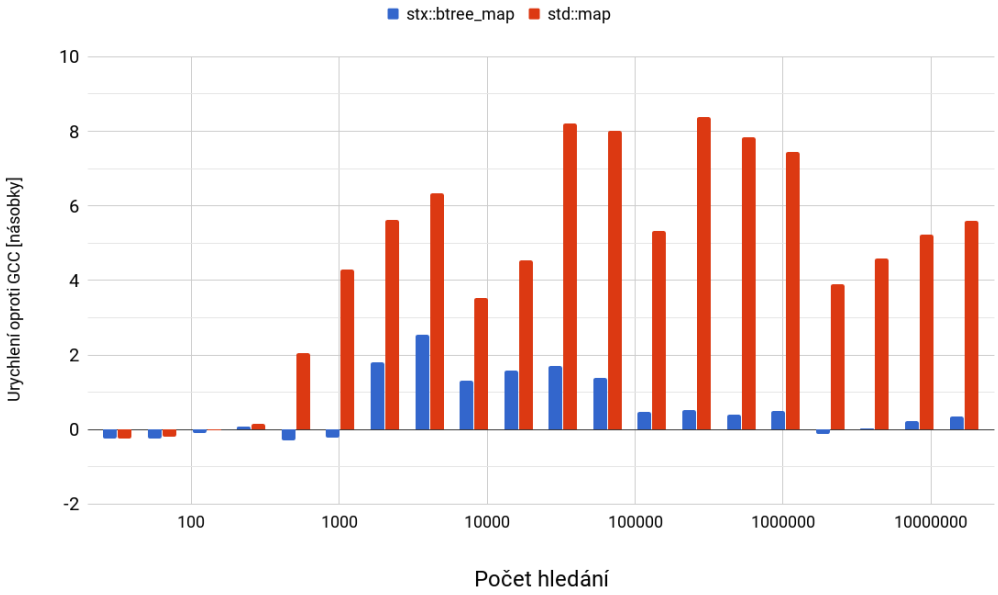
[1] https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html

	GCC - čas v [ns]			Clang - čas v [ns]		
Počet	Vekt.	stx::btree	std::map	Vekt.	stx::btree	std::map
32	164	125	125	152	148	200
64	346	265	277	306	336	527
128	655	594	650	609	837	1292
256	1315	1419	1528	1213	2136	3292
512	4749	3313	14523	4256	9236	10231
1024	9683	7582	51364	8928	28948	31808
2048	19943	55754	132069	19374	68347	79884
4096	42033	149066	308151	39177	154391	195827
8192	150921	350211	684597	137244	360695	523924
16384	307981	794138	1.71E+06	295533	814210	1.24E+06
32768	632688	1.71E+06	5.83E+06	627240	1.77E+06	4.13E+06
65536	1.61E+06	3.86E+06	1.45E+07	1.53E+06	4.22E+06	1.03E+07
131072	6.45E+06	9.57E+06	4.07E+07	6.30E+06	9.74E+06	2.95E+07
262144	1.43E+07	2.17E+07	1.34E+08	1.44E+07	2.19E+07	9.96E+07
524288	4.17E+07	5.80E+07	3.68E+08	4.05E+07	5.71E+07	2.47E+08
1048576	1.04E+08	1.54E+08	8.75E+08	1.00E+08	1.54E+08	6.45E+08
2097152	4.17E+08	3.67E+08	2.04E+09	4.17E+08	3.59E+08	1.44E+09
4194304	8.47E+08	8.71E+08	4.74E+09	8.54E+08	8.40E+08	3.32E+09

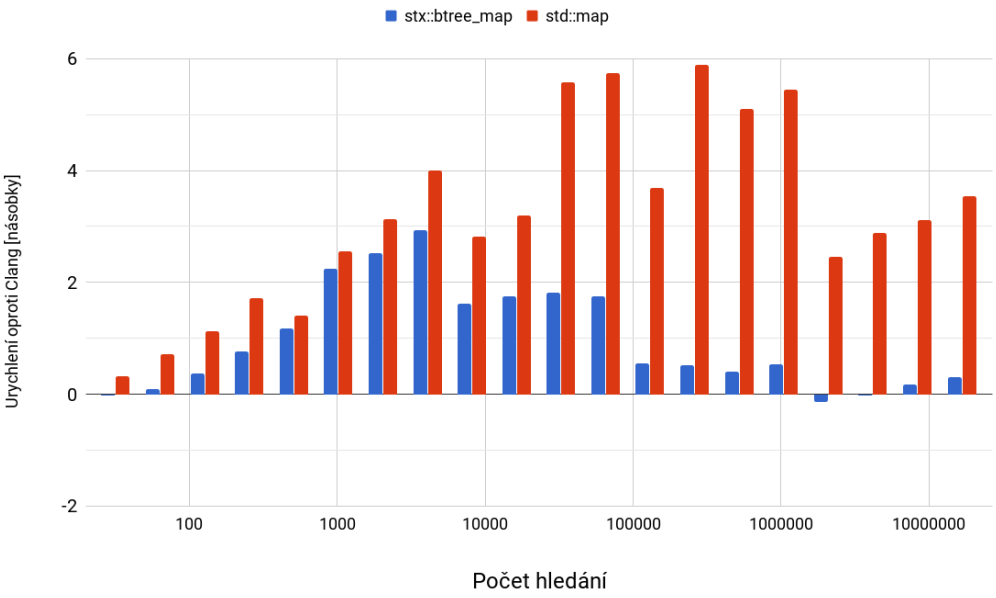
Tabulka 6.5: Srovnání časů provedení n vyhledání ve stromu s n záznamy. Sloupce *Vekt.* odkazují na vektorizovanou implementaci vyhledávání v B+ stromu s faktorem větvení 16.

Urychlení při použití překladače Clang ukazuje graf na obrázku 6.7. Je zajímavé, že tento překladač nebyl schopen optimalizovat vyhledávání ve stromu z knihovny `stx` tak dobře, jako překladač GCC. Co se týká implementace `std::map`, tak ta pro víc než 1024 prvků dosahuje naopak lepších výsledků, než v případě použití implementace překladače GCC. Dále Clang lépe optimalizoval vektorizované vyhledávání, což ještě umocňuje rozdíl mezi zrychleními u knihovny `stx` napříč překladači. Nicméně i zde je pro některé počty záznamů rychlejší knihovna `stx`.

Demonstrované výsledky ukazují na problém velkého větvení stromu a následnou závislost výkonu na počtu vložených prvků. Vysokým výkyvům ve výkonu by se jistě dalo předejít použitím dynamického stupně větvení a nebo změny velikosti jednotlivých listů.



Obrázek 6.6: Grafické znázornění urychlení vyhledávání ve vektorizovaném B+ stromu oproti jiným implementacím stromových struktur. Přeloženo překladačem GCC.



Obrázek 6.7: Grafické znázornění urychlení vyhledávání ve vektorizovaném B+ stromu oproti jiným implementacím stromových struktur. Přeloženo překladačem Clang.

Kapitola 7

Závěr

V rámci této diplomové práce byly shrnuty vlastnosti nové instrukční sady AVX-512, návaznost na hardware a zhodnoceny její hlavní výhody. Ve třetí kapitole byly projednány přístupy ke zpracování dat a navazující organizací struktur v paměti. Pro tuto práci klíčový model zpracování dat po vektorech byl posunut ještě blíže k hardwarovému návrhu a to ke zpracování po vektorech v registrech. Tento nový koncept zpracování byl implementován v systému, který umožňuje překládat agregační dotazy přímo do vektorizované formy, a to bez zbytečné režie za běhu agregace. Nová implementace byla otestována oproti kódu v C++, který by pravděpodobně použil programátor, pokud by měl vyřešit úlohu zadanou SQL dotazem za použití standardní knihovny jazyka.

Porovnání výkonu vektorizované generované implementace a skalárního kódu ukázaly, že při použití základních agregací může být vektorová implementace v průměru více než 4x rychlejší. Co se týká agregací podle závislé hodnoty, tak tam je vektorizovaná implementace rychlejší, na testovaných dotazech, v průměru pouze o 24% než přímočará implementace v C++.

Tato práce poukázala na možnosti, které přináší nová instrukční sada AVX-512 a zároveň předvedla způsob, jakým je možné vektorové instrukce prakticky použít při vyhodnocování agregačních funkcí.

Literatura

[1] BACON, D. F. et al. Spanner: Becoming a SQL System. In *Proc. SIGMOD 2017*, s. 331–343, 2017.

[2] BRAMAS, B. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. *International Journal of Advanced Computer Science and Applications (IJACSA)*. 2017.

[3] BUTTERSTEIN, D. – GRUST, T. Precision performance surgery for CostgreSQL: LLVM—based Expression Compilation, Just in Time. *Proceedings of the VLDB Endowment*. 2016, 9, 13, s. 1517–1520.

[4] CELIS, P. – LARSON, P.-A. – MUNRO, J. I. Robin hood hashing. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, s. 281–288. IEEE, 1985.

[5] CONTRIBUTORS, W. Skylake (server) - Microarchitectures - Intel, 2018. Dostupné z: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`. Cit. 6.3.2018.

[6] FIRASTA, N. et al. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper*. 2008, 19, s. 20.

[7] FLEMMING, P. – SCHWALB, D. HyriseSQL: A SQL Interface for Hyrise A Technical Documentation. 2015.

[8] FOG, A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*. 2018, s. 149–159.

[9] GRAEFE, G. *Encapsulation of parallelism in the Volcano query processing system*. 19. ACM, 1990.

[10] Hipp, R, et. al. SQLite. SQLite Development Team, 2018. Dostupné z: `https://www.sqlite.org/download.html`.

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Č. 248966-039. 12 2017.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Č. 325462-066US. March 2018.

[13] Intel Press. Intel Xeon Scalable Processors, Prezentace, 2017.

[14] MANO, M. M. – KIME, C. R. *Logic and computer design fundamentals*. 3. Prentice Hall, 2008.

[15] OBERMAN, S. – FAVOR, G. – WEBER, F. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro*. 1999, 19, 2, s. 37–48.

[16] POLYCHRONIOU, O. – RAGHAVAN, A. – ROSS, K. A. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, s. 1493–1508. ACM, 2015.

[17] SHAIKHHA, A. – DASHTI, M. – KOCH, C. Push vs. Pull-Based Loop Fusion in Query Engines. *arXiv preprint arXiv:1610.09166*. 2016.

[18] STGATILOV. *Performance comparison: linear search vs binary search* [online]. 2017. Cit: 18.3.2018. Dostupné z: `https://dirtyhandscoding.wordpress.com/2017/08/25/performance-comparison-linear-search-vs-binary-search/`.

[19] YUKHIN, K. Intel Advanced Vector Extensions, 2015/2016 support in GNU compiler collection. *GNU Tools Cauldron*. 2014.

[20] ZUKOWSKI, M. et al. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.* 2005, 28, 2, s. 17–22.