VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Více-vláknová main-memory hash tabulka

Multi-Core Main-Memory Hash Table

2018 Vojtěch Zdzieblo

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Vojtěch Zdzieblo**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Více-vláknová main-memory hash tabulka
Multi-Core Main-Memory Hash Table

Jazyk vypracování: čeština

Zásady pro vypracování:

Hash tabulka je jedna ze základních datových struktur, která se uplatní v mnoha odlišných problémech computer science. Cílem této práce je prozkoumat existující neperzistentní (main-memory) hash tabulky, které podporují více-vláknový přístup, implementovat vlastní hash tabulku a porovnat ji na vybraném vytížení.

Kroky při řešení tohoto úkolu tedy jsou:
1. Rešerše existujících main-memory hash tabulek podporujících více vláken.
2. Návrh, implementace a důkladné odladění vlastní hash tabulky.
3. Výkonnostní porovnání vlastní implementace s existujícími na vybraných vytíženích. Porovnání bude provedeno pro jedno a více vláken a pro více typů vytížení (s různým poměrem read a write operací).

Seznam doporučené odborné literatury:

[1] Blanas, Spyros, Yinan Li, and Jignesh M. Patel. "Design and evaluation of main memory hash join algorithms for multi-core CPUs." Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. června 2018 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 29. června 2018 .....

Rád bych na tomto místě poděkoval vedoucímu mé diplomové práce Ing. Radimovi Bačovi, Ph.D. za cenné rady, konzultace a trpělivost při vytváření této práce.

Abstrakt

V této práci popisuji datovou strukturu hashovaná tabulka, která podporuje přístup z více vláken naráz. Zabývám se tématy hashování, paralelismu a řešení kolizí. Poté následuje rešerše několika stávajících implementací více-vláknové hashované tabulky a nakonec podrobný popis vlastní implementace a srovnání s implementací z knihovny jménem Junction.

Klíčová slova: hash map, hash, hashovaná tabulka, hash table, slovník, vlákna, souběh, zámek, lock, lock free, hash join, spojení tabulek, compare and swap cas, mutex

Abstract

In this thesis I describe data structure named hash table, which supports accessing from multiple threads simultaneously. I am going through sections as hashing, paralelism and collision avoidance. Next is research of existing implementations of multi-core hash table. In the end there is detailed description of my own implementation and comparison with implementation from library named Junction.

Key Words: hash map, hash, hash table, dictionary, threads, multi core, concurrent, lock, lock free, hash join, compare and swap, cas, mutex

Obsah

Seznam použitých zkratek a symbolů **9**

Seznam obrázků **10**

Seznam tabulek **11**

Seznam výpisů zdrojového kódu **12**

1 Úvod **13**

2 Hashovaná tabulka **14**
2.1 Hashování . . . 15
2.2 Kolize . . . 16
2.3 Navyšování velikosti . . . 18
2.4 Faktor vytížení . . . 18
2.5 Výhody . . . 18
2.6 Nevýhody . . . 19
2.7 Použití . . . 19

3 Paralelismus **20**

4 Hashované tabulky podporující souběh **23**
4.1 Intel® TBB concurrent_hash_map . . . 23
4.2 Folly AtomicHashMap . . . 23
4.3 Junction . . . 24
4.4 Libcuckoo . . . 25
4.5 NBDS (Non-Blocking Data Structures) . . . 27
4.6 CDS (Concurrent Data Structures) . . . 27
4.7 Porovnání . . . 27

5 Vlastní implementace **28**
5.1 Problém . . . 28
5.2 Implementace lock-free hashované tabulky . . . 29
5.3 Vstupní data . . . 33
5.4 Řešení . . . 36
5.5 Řešení s Junction . . . 41
5.6 Struktura programu . . . 43
5.7 Umístění v souteži . . . 47

6 Porovnání výkonu vlastní implementace s Junction **48**
6.1 Kolekce ”small” 48
6.2 Kolekce ”public” 50
6.3 Experiment výkonu vláken u malých relací 52

7 Závěr **53**

Literatura **54**

8 Seznam příloh **56**

Seznam použitých zkratek a symbolů

CAS	–	Compare And Swap
FAA	–	Fetch And Add
ASCII	–	American Standard Code for Information Interchange
CMPXCHG	–	Compare and Exchange
RAM	–	Random Access Memory

Seznam obrázků

1 Ukázka hashované tabulky 14
2 Ukázka hopscotch hashování 17
3 Vložení klíče do cuckoo tabulky 26
4 Přehashování klíče v cuckoo tabulce 26
5 Škálovatelnost hashovaných tabulek 27
6 Ukázka použití hashované tabulky v projektu 29
7 Histogram počtu řádků kolekce small 33
8 Histogram počtu sloupců kolekce small 34
9 Histogram počtu řádků kolekce public 35
10 Histogram počtu sloupců kolekce public 35
11 Paralelizace indexování 36
12 Ukázka spojení tabulek 37
13 Plán vykonání dotazu 37
14 Odevzdané řešení do soutěže 47
15 Časy prvních pěti soutěžících 47
16 Škálovatelnost vláken - indexování ”small” 48
17 Škálovatelnost výkonu vláken - dotazy ”small” 49
18 Škálovatelnost výkonu s alokací paměti ve vláknech - ”small” 49
19 Škálovatelnost výkonu vláken - indexování ”public” 50
20 Škálovatelnost vláken - dotazy ”public” 51
21 Škálovatelnost výkonu s alokací paměti ve vláknech - ”public” 51
22 Škálovatelnost výkonu vláken při indexování menší relací 52
23 Škálovatelnost výkonu vláken při indexování větších relací 52

Seznam tabulek

1 Statistiky (min, max a průměr hodnot) o relacích kolekce small 33
2 Statistiky (min, max a průměr hodnot) o relacích kolekce public 34
3 Obsah CD 56

Seznam výpisů zdrojového kódu

1 Pseudo kód instrukce CMPXCHG 21
2 Ukázka testovacího dotazu v jazyce SQL 28
3 Tělo metody pro vložení prvku do zásobníku 30
4 Tělo metody pro vložení prvku do hashované tabulky 31
5 Tělo metody pro nalezení hodnoty podle klíče v hashované tabulce 32
6 Metoda pro spojení a průchod tabulkami 38
7 Metoda NextRow s vyhledáváním v indexu 39
8 Vytvoření instance s předalokovanou pamětí 40
9 Použitelné argumenty programu 41
10 Tělo metody pro vložení prvku do Junction hashované tabulky 42
11 Tělo metody pro nalezení hodnoty v Junction hashované tabulce 42
12 Větvení podle vybrané implementace hashované tabulky 43
13 Metoda pro získání paměti z bloku 44
14 Metoda pro zaindexování části relace v novém vlákně 45
15 Metoda NextRow u iteratoru 46

1 Úvod

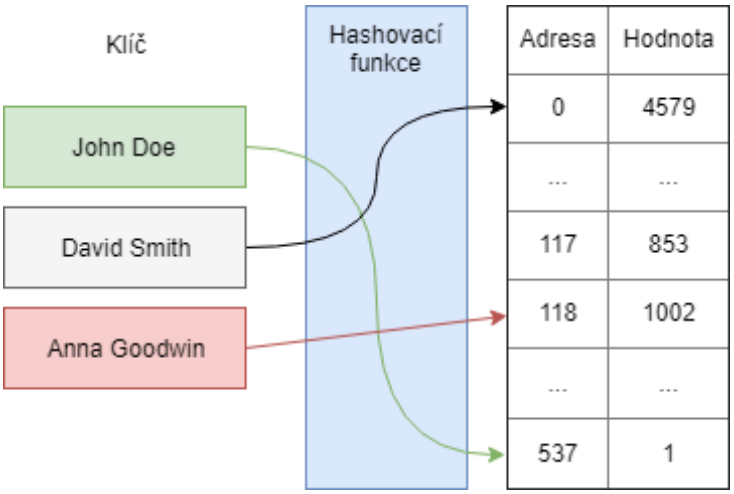
S nárustem jader a vláken procesorů se otevírají nové možnosti pro datové struktury podporující přístup z více vláken.
Tato práce popisuje hashovanou tabulku hlavně pro účely použití s více vlákny, zabývá se tématem hashování, řešením kolizí a paralelismem.
Hashovaná tabulka je ušita na míru pro řešení problému soutěže SIGMOD 2018, ve kterém se provádí dotazy s agregační funkcí a spojováním relací. Přístup více vláken je řešen lock-free operací CAS, což umožňuje větší propustnost, než řešení se zámky.

Diplomová práce má následující strukturu: Nejprve podrobněji popíšu a vysvětlím co je to hashovaná tabulka a k čemu se používá. Dále je řeč o paralelismu, tedy práci s více vlákny a výhody či nebezpečí, které s sebou přináší. Následuje rešerše existujících implementací hashované tabulky, které podporují přístup z více vláken a na základě této rešerše je další část o implementaci vlastní lock-free hashované tabulky, ktrou v poslední části porovnám s implementací hashované tabulky z jedné z existujících knihoven.

2 Hashovaná tabulka

Hashovaná tabulka (též mapa nebo slovník) je datová struktura, která umožňuje mapovat klíče na hodnoty. Uvnitř obsahuje pole tzv. slotů (bucketů), do kterých lze ukládat záznamy. Hashovací tabulka v podstatě dělá jen to, že na základě klíče vybere vhodný slot a operaci provede v něm. K tomu využívá tzv. hashovací funkci, pomocí které si z klíče vypočítá pozici slotů (bucketů) v poli, ve kterých se hodnota uchovává.

Je využívána hlavně kvůli své rychlosti jednotlivých operací vyhledávání a vkládání. V nejlepším případě je jednomu klíči pomocí hashovací funkce přiřazen právě jeden slot v poli a proto taky hashované tabulky mívají většinou konstatní asymptotickou složitost *O(1)* pro operace čtení, vkládání a mazání. Pokud je ale hashovací funkce špatná, může docházet ke kolizím, což znamená, že pro více odlišných klíčů bude nalezen jeden a tentýž slot. Proto je asymptotická složitost v nejhorším prípadě *O(n)*, kde n značí počet prvků hashované tabulky.



Obrázek 1: Ukázka hashované tabulky

2.1 Hashování

Hlavní myšlenkou hashování je použití hashovací funkce k přeložení unikátního identifikátoru (klíče) na relativně nízké číslo. Toto číslo odpovídá adrese v paměti (pozice v tabulce) ve které je objekt uložen. Hashovací funkce by měla namapovat unikátní klíč na unikátní pozici v tabulce. Toho můžeme docílit pomocí pokročilých technik perfektního hashování, kde se používají injektivní hashovací funkce. Když však dopředu neznáme množinu klíčů, určených k ukládání, není jednoduché definovat takovou injektivní funkci, což může zapříčit to, že jedna pozice v tabulce může být přiřazena k více klíčům a tím vznikají kolize.

Složitosti vyhledávání, vkládání a mazání v hashované tabulce se odvíjejí od typu zvolené hashovací funkce. Pokud máme kolekci prvků a hashovací funkci, která mapuje každý prvek do unikátního slotu tak máme perfektní hashovací funkci. Pokud víme, že kolekce prvků, které budeme vkládat, se nikdy nezmění, pak je možné vytvořit perfektní hashovací funkci. Bohužel, vzhledem k libovolnosti prvků v kolekci neexistuje systematický způsob, jak vyvořit dokonalou hashovací funkci. Naštěstí, abychom dosáhli efektivního výkonu, nepotřebujeme aby byla hashovací funkce perfektní.

Jedním způsobem, jak mít vždy perfektní hashovací funkci, je nastavit velikost hashovací tabulky tak, aby každá hodnota měla svoje vlastní místo v poli. Toto garantuje, že každá hodnota bude mít unikátní slot. Je to však praktické pro menší množství prvků, protože pokud je rozsah možných hodnot velký jako např. u rodného čísla, což je desetimístné číslo, včetně číslic za lomítkem, tak by se muselo vytvořit téměř deset miliard slotů a pokud zároveň potřebujeme uložit data pouze pro třídu o 20 studentech, bylo by to velké plýtvání pamětí.

Pro vytváření hashovacích funkcí se využívá například skládací metoda (folding method), která spočívá v tom, že se číslo rozdělí na několik stejně velkým kusů. Tyto kusy jsou poté sečteny a a tím je hodnota zahashována. Např. máme telefonní číslo 735 512 331, které bychom rozdělili do skupin po třech číslech a ty potom spolu sečetly. Takže 735 + 515 + 331 se rovná 1581 a pokud má hashovaná tabulka třeba 50 slotů, bude výsledná pozice v poli 1581 % 50, což je 31.

Další hashovací funkce s čísly je mid-square metoda. Ta nejprve umocní číslo na druhou a poté z něj vyextrahuje určitý počet číslic. Např. máme číslo 44 a jeho druhá mocnina je 1936. Když z prostředku vybereme dvě čísla tj. 93 a provedeme zbytek po dělení v případě hashované tabulky o 50 slotech, výsledná adresa bude 93 % 50 = 43.

Nejjednodušší hashovací funkce pro práci s textem je rozložit text na jednotlivé znaky, každý znak si představit jako číselnou hodnotu podle tabulky ASCII a tyto hodnoty sečíst. Slovo "pes" bychom rozložili na 112 + 101 + 115, takže výsledek hashovací funkce je 328.

2.2 Kolize

Když nastane kolize[14], je nutné uložit prvky s kolidujícími klíči do alternativní pozice.

2.2.1 Separate chaining

Jedním z možných řešení kolizí je zřetězené hashování (separate chaining) což je nejjednodušším způsobem, jakým se lze vypořádat s kolizemi. Hodnoty se ukládají do spojových seznamů. Nastane-li kolize, prvek se pouze přidá na konec adresovaného spojového seznamu. Nevýhoda tohoto přístupu je, že při velkém zaplnění tabulky dojde k postupné degradaci výkonu kvůli sekvenčnímu prohledávání příslušných seznamů. Další dvě základní strategie, pomocí níž se tabulka vypořádává s kolizemi – linear probing a double hashing.

2.2.2 Linear probing

V linear probing strategii nejprve vypočteme adresu, na kterou daný prvek uložíme. Je-li adresa obsazená, tak se posuneme o jedno místo dál a zkusíme prvek uložit znovu. Takto postupujeme tak dlouho, dokud se nám prvek nepodaří uložit.

2.2.3 Dvojité hashování

Využívá dvojici hashovacích funkcí. První funkce vypočte adresu stejným způsobem jako u linear probingu nebo zřetězeného rozptylování a pokud je dané místo již obsazené, nastoupí druhá funkce, která vypočte posun. Za předpokladu, že je i nové místo plné, dojde opět k posunu na základě druhé funkce. Dvojité hashování je v této práci dále rozvedeno včetně ukázky v sekci popisu knihovny *Libcuckoo*.

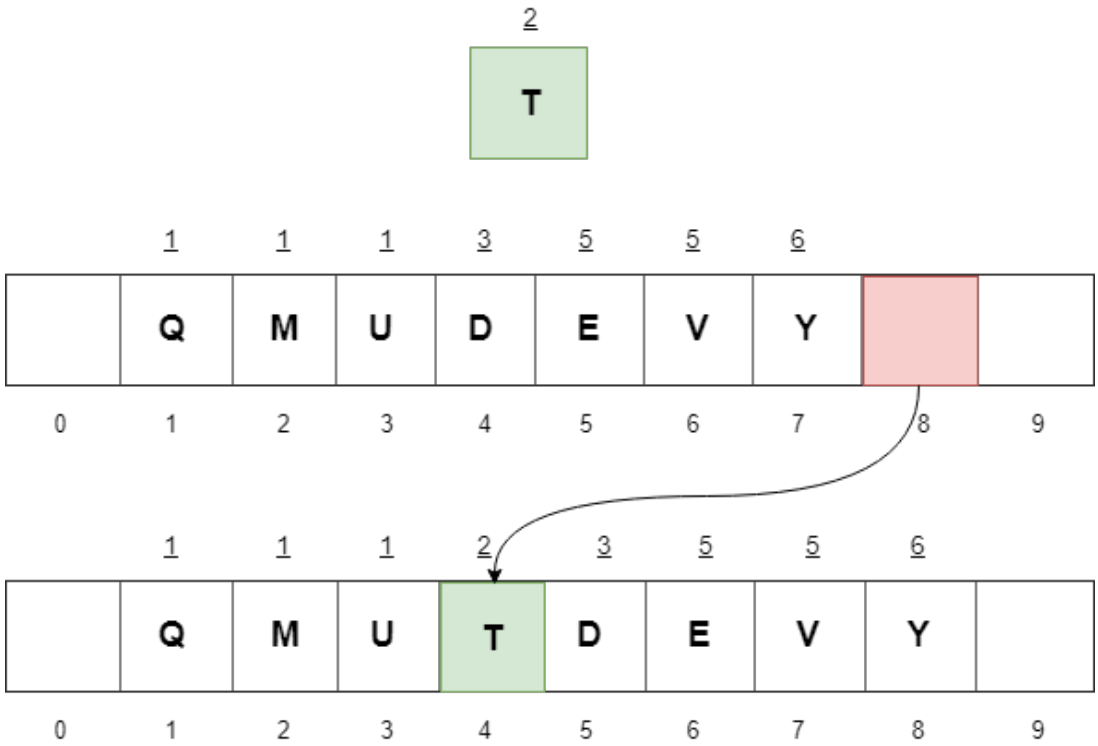
2.2.4 Quadratic probing

Chová se stejně jako linear probing s tím rozdílem, že neprochází pole sekvenčně, ale adresu následujícího slotu vypočte tak, že ke klíči přičte vždy druhou mocninu čísla, které určuje kolikátý pokud o nalezení kliče se jedná. Takže při prvním pokusu bude výpočet: (Key + 1^2) % Size, při druhém pokusu to bude: (Key + 2^2) % Size atd.

2.2.5 Hopscotch hashing

[18] Je obdoba linear probingu, s tím rozdílem, že místo toho abychom našli prázdné místo a do něj uložili novou hodnotu v případě kolize, tak přesuneme prázdné místo co nejblíže místu, kde měla být hodnota původně vložena. Při přesouvání prázdného místa se kontroluje tzv. sousedství (neighborhood) jednotlivých hodnot. Pokud má hashovaná tabulka nastaveno sousedství např. na hodnotu 3, znamená to, že pokud při ukládání nastane kolize, musí být prvek uložen nejvýše 3 místa od svého původního místa určení. Pokud při přesouvání hodnoty nelze toto pravidlo split, třeba protože, že už okolo místa určení jsou 3 sousedi, tak je potřeba tabulku rozšířit.

Jako příklad uvedu hashovanou tabulku, kde jsou uvedeny jednoznakové klíče ve slotech, horní čísla značí adresu, kterou pomyslná hashovací funkce vypočítala pro daný klíč a spodní čísla značí adresy přímo v paměťovém poli. Pokud se tedy pokusíme přidat nový klíč s hodnoutou "T", prvek by se měl zařadit na adresu 2. Ta je však obsazena a okolo ní tvoří sousedství klíče, které patří do adresy 1. V tomto případě algoritmus zařadí klíč "T" na první volné místo, což je slot s adresou 8 a postupně začne procházet klíče z prava do leva, jeden po druhém a zjišťuje, zda se může nově přidaný klíč vyměnit se svým kolegou vlevo. Při tom musí dodržovat výše zmíněné pravidla. Nakonec dojde k adrese 4 a tam klíč uloží. Nemůže ho uložit ještě více vlevo, protože by porušil sousedství jedničkám a zároveň musí být poblíž své skutečné adresy, tj. 2.



Obrázek 2: Ukázka hopscotch hashování

2.3 Navyšování velikosti

Při vytváření hashované tabulky je většinou potřeba specifikovat počáteční velikost. Ta ale v budoucnu, po vložení několika prvků, nemusí stačit a pokud se chceme vyhnout degradaci výkonu hashované tabulky, je potřeba tento problém vyřešit navýšením velikosti. Tento problém částečně řeší separate chaining, který řetězí prvky za sebe, ale tím výrazně klesá výkon a rychlost operací se blíží k *O(n)*. Pokud bychom se tomuto chtěli vyhnout a tabulku opravdu rozšířit, nejjednodušší způsob je vytvořit novou větší tabulku (většinou o dvojnásobné velikosti předešlé tabulky) a data z tabulky menší přehashovat a přesunout do nové. Tato situace je časově náročná, protože se musí projít všechny sloty.

V případech, kde si nemůžeme dovolit překopírovat celou tabulku naráz z důvodu časové náročnosti, je zde možnost postupného přehashovaní. Při potřeba rozšíření se alokuje nová tabulka a každá operace čtení nebo mazání nyní bude vyhledávat v obou tabulkách, nové i staré. Operace vkládání už bude pracovat pouze s novou tabulkou a prvky vkládat pouze do ní. Zároveň při každém vložení do hashované tabulky, se přesune x prvků ze staré do nové a jakmile se přenesou všechny, tak se stará tabulka dealokuje.

2.4 Faktor vytížení

Většina hashovaných tabulek vyžadují pro inicializaci parametry, které odhadují počet unikátních klíčů, které se budou vkládat a tabulka z nich určí počet slotů, tak aby se co nejvíce snížil počet kolizí. Tento parametr má velký vliv na výkon. Když chceme optimalizovat využití paměti, počet slotů v tabulce bude menší než počet klíčů. Pokud chceme optimalizovat výkon, počet slotů v tabulce nastavíme blíže k počtu klíčů, abychom co nejvíce předešli kolizím.

2.5 Výhody

If the set of key-value pairs is fixed and known ahead of time (so insertions and deletions are not allowed), you can reduce the average lookup cost by a careful choice of the hash function, bucket table size and internal data structures.

- Hashovaná tabulka se správně nastaveným počtem slotů má průměrnou cenu vyhledávání nezávislou na počtu prvků uložených v hashované tabulce
- Hashované tabulky jsou nejvíce efektivní, když dopředu známe počet záznamu, to proto, abychom alokovali místo pro tabulku pouze jednou s optimální velikostí, tak aby se nemusela nikdy velikost navyšovat
- Pokud je množina klíčů pevně daná a známe ji dopředu, lze snížit čas vyhledávání zvolením správné hashovací funkce, velikosti tabulky a interních datových struktur

2.6 Nevýhody

- Pro setřízená data není hashovaná tabulka vhodná.
- Hashované tabulky nejsou efektivní, když je počet prvků uvnitř tabulky velmi malý, i přesto, že mají operace v průměru konstantní složitost, cena při výpočtu dobré hashovací funkce může být mnohem větší, než mít data uložené v obyčejném poli a vyhledávat v něm sekvenčním průchodem.
- Pro aplikace, které zpracovávají řetězce, jako např. kontrola pravopisu, jsou hashované tabulky méně efektivní, než stromy
- I přesto, že jednotlivé operace jsou rychlé, může dojít k situaci, při které je potřeba tabulku rozšířit a operace se tak zdrží na základě počtu prvků v tabulce, které se musí při rozšíření zpracovat
- Mohou být neefektivní, pokud v tabulce dochází k velkému počtu kolizí

2.7 Použití

Hashované tabulky jsou užitečné v každé situaci, kde máme nepředvídatelnou sadu symbolů (klíčů) a chceme přidružit nějaké informace s každým z nich. Tyto informace můžeme zapisovat a číst v konstantním čase. Předpoklad je, že klíče nemusí být jakkoliv setřízené, jako např. když bychom chtěli klíče mít seřazené od největšího po nejmenší. Využívají se např. v databázích jako indexy nebo jako cache pro zrychlení přístupu k datům v aplikaci.

3 Paralelismus

[15][17] Existuje několik známých programovacích modelů, které se používají při implementaci souběžných aplikací. Aplikace je souběžná, pokud dva nebo více procesů pracují spolu na nějakém úkolu. Tyto procesy mohou běžet na více strojích, procesorech, jádrech, atd. Tato práce se však zabývá programovacím modelem zaměřeným na sdílenou pamět více procesory/jádry.

Procesory a paměťové modely jsou propojeny jako síť a proto je možné, že procesy spuštěné na jiných procesorech mohou zapisovat a číst ze stejné paměti. Každý procesor si uchovává svou lokální cache, kterou synchronizuje se sdílenou cache. Problém s konzistencí paměti je rozhodování kdy provést synchronizaci a jak obsloužit souběžné zápisy a čtení.

Mimo hardwarovou synchronizaci, existuje také softwarová synchronizace. Pro dosažení lepšího výkonu a lepší využití hardwaru, můžeme používat vlákna a s pomocí nich tak paralelizovat část nebo celou aplikaci. Vlákno je speciální entita, kterou označujeme jako logickou část běžící aplikace. Platí, že každá spuštěná aplikace má na úrovni operačního systému alespoň jeden proces a každý proces má minimálně jedno vlákno. Musíme zajistit, aby paralelizovaná verze aplikace splňovala stejné požadavky jako aplikace sekvenční. Tento úkol však může být velmi těžký, vzhledem k tomu jak vlákna přistupují a modifikují sdílené data.

Použitím těchto synchronizačních technik na úrovník vláken umožňuje vytvořit velkou škálu algoritmů a kontejnerů (kolekcí). V závislosti na tom, jak jsou tyto techniky používány, jsou implementace vystaveny několika nástrahám, kterým musí čelit pro správný souběh. Nástrahy jako závodění, deadlock, livelock a hladovění vláken (thread starvation).

Algoritmy, které neblokují vlákna jsou třídou algoritmů bez vzájemného vyloučení (mutex) kritických sekcí a namísto toho používají hadrwarové atomické operace. Na toto téma byl proveden rozsáhly výzkum, takže tyto algoritmy zaručují spolehlivost. Blokující algoritmy používají k vzájemné vyloučení k zajištění přístupu ke kritickým sekcím. K tomu se většinou používá zámek nebo semafor. Dalším způsobem jak dosáhnout paralelismu, je rozložení problém mezi více stroji. V distribuovaném výpočtu je problém rozložen do několika úkolů, z nichž každý je řešen jedním nebo více počítači.

Na rozdíl od sdílených paměťových systémů, které umožňují procesům komunikovat prostřednictvím sdílené paměti, musí zařízení v distribuovaném systému komunikovat přes síť. Tento způsob má většinou velký dopad na výkon. Rychlost komunikace záleží na umístění strojů v síti, jak jsou daleko od sebe. Stroje blízko sebe budou mít menší odezvu při komunikaci a tím pádem mohou za stejný čas odeslat více zpráv než stroje situované v jiné síti. V distribuovaných algoritmech je tedy komunikace minimalizována a optimalizována, kdežto u algoritmů se sdílenou

pamětí je potřeba zkrátit čas strávený v kritických sekcích a zajistit přístup.

Nejběžnější synchronizační operace:

- Compare-And-Swap (CAS) je operace, která atomicky porovná hodnotu adresy s adresou stávající hodnoty a poté nahradí stávající hodnotu novou hodnotou pokud se stávající hodnota shoduje s očekávanou
- Fetch-And-Add (FAA) je operace, které atomicky inkrementuje hodnotu na dané adrese o specifikované množství a vrátí hodnotu ještě v původním stavu
- Store atomicky zapíše hodnotu do specifikované adresy a zajistí, aby souběžné zápisy hodnotu nepoškodily
- Load atomicky načte hodnotu ze specifikované adresy a zajistí, aby částečně zapsané hodnoty nebyly přečteny

Níže 1 je ukázka instrukce CMPXCHG[19], která se využívá při operaci CAS. Cílová hodnota se porovná s hodnotou, která je momentálně v registru, pokud se shoduje tak cílovou hodnotu nahradí nová hodnota a instrukce vrátí *true*. Klíčové slovo *lock* udělá instrukci atomickou.

```
1 lock cmpxchg dest, newval
2
3 if (dest == accumulator) {
4    ZF = 1
5    dest = newval
6 } else {
7    ZF = 0
8    accumulator = dest
9 }
```

Výpis 1: Pseudo kód instrukce CMPXCHG

Metody neblokujících algoritmů se řadí do skupin podle úrovně postupu, který garantují:

- Obstruction-Free: metoda je bez obstrukcí, jestliže v kterémkoliv okamžiku jejího provádění je odizolována od ostatních metod, pak je schopna dokončit provádění v konečném počtu kroků
- Lock-Free: metoda je lock-free, pokud zaručuje, že nekonečně časté volaní metody se dokončí v konečném počtu kroků
- Wait-Free: metoda je wait-free, pokud zaručuje, že každé její volání se dokončí v konečném počtu kroků

Tak jak jsme zvyklí chápat sekvenční (ne-paralelní) programování, tak souběžné programování s sebou přináší řadu nebezpečí. Bez ohledu na to, jakým způsobem je implementována synchronizace, tak algoritmy mohou být náchylné (nebo je přímo obsahovat) na scénáře, které se těžko detekují a mohou vyústit v něočekávané chování. Tyto nebezpečí jsou vetšinou nedetekovatelné klasickou testovací technikou, protože se vyskytují pouze za velmi specifických podmínek.

Kromě těchto problémů, jsou zde další, které ohrožují životnost algoritmů. Např. deadlock je situace, ve které dvě nebo více soutěžících/závodících akcí čekají jeden na druhého až se dokončí a tím pádem se nedokončí nikdy. Podobný deadlocku je livelock, což je situace ve které se stavy procesů navzájem mění a tím pádem, se žádná nehne z místa. A poslední hladovění vláken nastane, když se nějakému vláknu nedostane prostředek, který potřebuje a tím je mu bráněno v dalším postupu.

Při tvoření algoritmů pro kritické systémy, je důležité aby neobsahovaly tyto nebezpečí. V opačném případě může nastat výrazné poškození v důsledku nereagujícího systému nebo jeho částí. Wait-Free algoritmy jsou bez všech tří výše zmíněných nebezpečí a jsou tedy pro takové systémy velmi vhodné.

Další běžnou technikou je použití CAS operace k podmíněné změně hodnoty na konkrétní adrese. Přečtením návratové hodnoty operace můžeme zjistit, zda jiné vlákno modifikovalo onu adresu. Pokud se návratová hodnota rovná očekávané hodnotě, tak hodnotu změnilo právě špuštěné vlákno. V opačném případě byla hodnota změněna jiným vláknem a záleží na nás jakým způsobem oba případy bude řešit.

Pokud bychom místo toho použili atomický Store, nevěděli bychom jaká hodnota byla přepsána. Např. pokud inkrementujeme číslo a vlákno z čísla přečte hodnotu 0, ale než to vlákno hodnotu uloží, jiné vlákno mezitím také uloží hodnotu 1. První vlákno tím pádem neví, že už byla hodnota inkrementována a hodnotu znovu přepíše.

4 Hashované tabulky podporující souběh

4.1 Intel® TBB concurrent_hash_map

Intel® Threading Building Blocks[6] je knihovna, která podporuje škálovatelné paralelní programování pomocí standardního kódu C++. Nevyžaduje speciální jazyky nebo překladače.

Hashovaná tabulka řídí souběh pomocí tzv. "accessor" a "const_accessor", které se chovají jako smart pointery (chytré ukazatele). Accessor je používán pro zápis do tabulky nebo mazání z tabulky a dokud odkazuje na nějaký prvek, všechny pokusy o vyhledání klíče tohoto prvku musí čekat než se dokoncí operace zápisu. Const accessor je podobný, až na to, že se používá pouze pro čtení, takže více const_accessorů může odkazovat na stejný prvek ve stejný čas. Tato vlastnost výrazně zlepšuje souběh v situacích, kde jsou prvky často čteny a méně často zapisovány.

Vzájemné vyloučení při přístupu více vláken je řešeno pomocí "spin_mutexu", který zablokuje vlákno jakmile si vyžádá zámek. Tento typ je vhodný pokud je zámek držen pouze pro pár instrukcí. Při zavolání metody acquire vlákno čeká, než získá zámek a metoda release zámek uvolní. Ruční uvolnění zámku, může ale způsobit, že při vyjímce v programu se zámek neuvolní, a proto je vhodné použít "scoped_lock", což je zámek platný pro celý blok kódu vymezený složenými závorkami. V případě blokového zámku se o uvolnění zámku (destrukci objektu) postará objektově orientovaný jazyk C a zavolá destruktor automaticky na konci bloku nebo po vyjímce.

4.2 Folly AtomicHashMap

Folly[16] (Facebook Open Source Library) je knihovna skládající se z několika C++11 komponent a AtomicHashMap je jednou z nich. Je navržena pro extrémní výkon v prostředí s více vlákny a také má dobré vlastnosti co se využití paměti týče. Operace vyhledávání a iterace je možné provádět bez jakýchkoliv zámků či čekání, ale vkládání uzamkne přístup na úrovni klíče.

Ačkoli může poskytovat extrémní výkon, AtomicHashMap nepodporuje uvolnění obsazené paměti při operaci mazání, takže prvky přidané do hashované tabulky zabírají místo i po jejich odstranění, není proto výhodné mapu používat, pokud se z hashované tabulky často maže. Dále mapa podporuje pouze 32 či 64 bitové číselné klíče, protože je použito atomické porovnání a prohození (compare and swap). Je také vhodné dopředu vědět přibližný počet prvků, který do kolekce budeme vkládat, protože způsob, kterým komponenta navyšuje svou kapacitu je neefektivní, takže povede ke zpomalení. Iterace přes mapu jsou vždy platné i přesto že jiné vlákno do mapy zapisuje nebo z ní maže.

AtomicHashMap se skládá z komponent zvaných AtomicHashArray (subtabulek) zřetězených za sebou. Tato komponenta umožňuje využívání klíčové funkcionality - operace bez zámků, ale

nemůže zvětšit svou kapacitu, než s jakou byla inicializována. AtomicHashMap tedy při zaplnění první subtabulky za sebe skládá další subtabulku. Pokud bude mapa obsahovat více subtabulek, bude je muset při vyhledávání klíče projít postupně, což snižuje výkon. Pokud zvolená kapacita při inicializaci bude optimální, bude existovat pouze jedna instance podmapy a výkon bude optimální. AtomicHashArray na základě klíče (hashe) pracuje se svými prvky typu klíč-hodnota.

Algoritmus pro vkládání je jednoduchý, klíč je přehashován podle velikosti komponenty a poté je klíč prvku atomicky porovnán a vyměněn s uzamčenou hodnotou klíče. Pokud se zápis podaří, klíč je odemknut tím, že se nastaví na novou hodnotu. Pokud neuspěje, pokusí se o porovnání u dalšího prvku nebo dokud není hashovaná tabulka plná. Při vyhledávání je klíč opět přehashován podle velikosti mapy a ten se použije pro vyhledávání. Tato operace je provedena bez čekání na zámek a bez atomické instrukce, protože prvky jsou vždy ve validním stavu. Mazání porovná a nastaví klíč prvku na speciální rezervovanou hodnotu.

4.3 Junction

[1] Je C++ Knihovna obsahující datové struktury s podporou paralelismu a k tomu využívá knihovnu Turf[2], která funguje jako vrstva nad POSIX, Win32, Mac, Linux, Boost a C++11, takže díku tomu Junction podporuje velkou škálu platforem. Junction obsahuje tři implementace hashované tabulky - Linear, Leapfrog a Grampa [3].

- Základní implementace je Linear a ostatní dvě implementace ji rozšiřují. Linear, stejně jako hashovaná tabulka ve Folly, využívá CAS operaci při vkládání prvků. Narozdíl od Folly však svým způsobem podporuje různé datové typy než pouze 32 a 64 bitové číslo, je ale potřeba doimplementovat metodu, která libovolný typ předělá na číselný typ. Interně využívá číselný typ jako klíč právě proto, že provádí CAS operaci, která toto vyžaduje.
- Leapfrog zakládá na implementaci Linear, ale vylepšuje vyhledávání klíče v mapě při velkém zaplnění použitím tzv. "hopscotch hashing". Tato implementace je hlavně výkonější, protože modifikuje sdílený stav mapy mnohem méně.
- Grampa je podobná implementaci Leapfrog, nýbrž u velkém zaplnění mapy se kolekce rozdělí na několik menších Leapfrog map s pěvně danou velikostí. Jakmile jedna z těchto map opět dosáhne velkého zaplnění, opět se rozdělí do dvou nových.

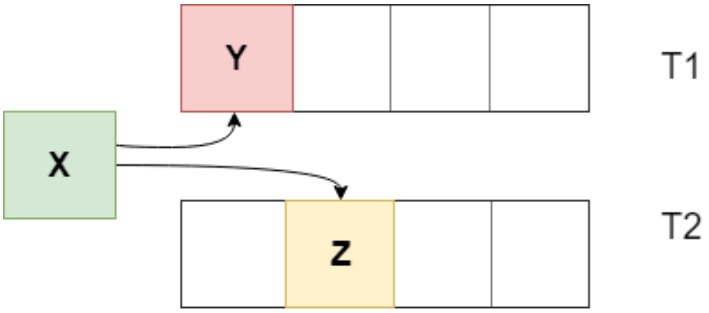
4.4 Libcuckoo

[10] [11] Implementace využívá tzv. kukaččí hashování, jméno metody je odvozeno od chování některých druhů kukaček, u kterých mládě po vylíhnutí vyhodí původní vajíčka nebo mláďata z hnízda. Kukaččí hashování je založeno na myšlence, že nepoužívám jednu hashovací funkci, ale dvě (a někdy i více). To poskytne dvě možné umístění v hashovací tabulce pro každý klíč. V tomto algoritmu je hashovací tabulka rozdělena na dvě menší stejně veliké tabulky a každá hashovací funkce indexuje do jedné z nich. Když je vkládán nový klíč, tak se vloží na jedno ze svých dvou možných umístění, přičemž "vykopne", tj. nahradí jiný klíč, který tam je případně umístěn. Tento nahrazovaný klíč je pak umístěn na svoje alternativní umístění, kde případně opět vykopne prvek tam sídlící. Přemísťování pokračuje, dokud se nenajde volná pozice anebo se metoda zacyklí. V tom případě je hashovací tabulka přehashována na místě za použití nových hashovacích funkcí.

Všechny prvky jsou uloženy ve velkém poli, bez jakýkoliv ukazatelů nebo spojovaných seznamů. K řešení kolizí se používají dvě techniky. První, prvky mohou být uloženy v jednom ze dvou bucketů v poli a mohou být přesunuty na své druhé místo, pokud je první plné. Druhá, při běžném použití má každý bucket B "slotů" pro prvky. V praxi užívaná hodnota je B = 4. Klasické kukaččí hashování umožňuje využít pouze 50% procent záznamů tabulky, které mají být obsazeny předtím, než dojde k neřešitelným kolizím. Toto je vylepšeno použitím čtyřcestné (nebo vyšší) asociativní množiny, díky kterým lze využít více než 90% tabulky. Vyhledání klíčů probíhá výpočtem dvou hashů jednoho klíče, aby se nalezly buckety b1 a b2, které by mohly být použity k uložení klíče, a prozkoumáním všech slotů uvnitř každého z těchto bucketů, aby se zjistilo, zda je klíč přítomný. Základní hashovací tabulka ("2,4-cuckoo") (dvě hashovací funkce, čtyři sloty v bucketu) je zobrazena na obrázku. Důsledkem toho je, že vyhledávací operace jsou rychlé a předvídatelné, vždy kontrolují 2B klíčů. Pro vložení nového klíče do tabulky, pokud má některý ze dvou prázdný slot, vloží se do tohoto bucketu, pokud žádný z bucketů nemá místo, náhodný klíč z jednoho z bucketů je novou položkou nahrazen. Nahrazený prvek se pak přemístí do své vlastní alternativní pozice, případně přemístí jiný prvek atd. Dokud nedosáhne maximálního počtu posunutí. Pokud není nalezen prázdný slot, hashovací tabulka je považována za příliš plnou pro vložení a naplánuje se rozšiřovací proces. Posloupnost nahrazených prvků při vkládání se nazývá "kukaččí cesta". Výkon zápisu kukaččího hashování degraduje zaplňováním hashovací tabulky, protože se zvetšuje délka kukaččí cesty a více náhodných čtení/zápisů je potřeba pro každou operaci vložení.

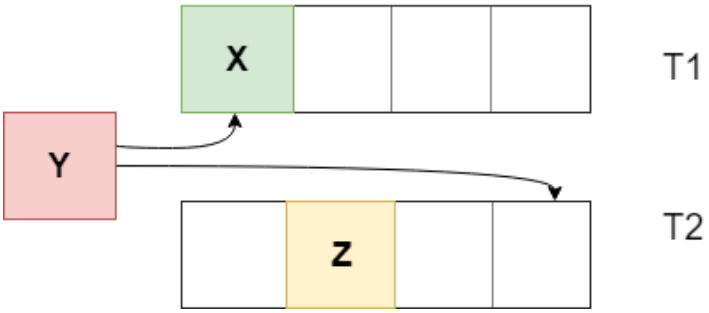
Zámek se použije jenom v případě, že je potřeba udělat zápis do hashované tabulky. Jelikož vyhledávací fáze není chráněna zámkem, existuje potenciální problém při tzv. ”závodění” vláken. Když první vlákno čte z bucketu a prochází kukaččí cestou, druhé vlákno může zapisovat do stejného bucketu a tím pádem první vlákno čte špatné data. Proto, při zápisu je potřeba pokaždé zkontrolovat zda příslušné položky byly změněny před posunem položek ve fázi provádění. Pokud je stávající cesta neplatná, algoritmus se restartuje a začne vyhledávat novou cestu. Každé přemístění přemístí pouze jednu položku do svého alternativního bucketu, takže není potřeba jej vracet zpět, jakmile se operace přeruší.

Níže je ukázka vložení klíče do cuckoo hashované tabulky. Máme dvě hashovací funkce a dvě tabulky, pro rozptýlení kolizí. Chceme vložit prvek s klíčem X, ale první hashovací funkce vypočítá adresu, ve které je klíč Y, druhá hashovací funkce vypočítá adresu, která je také zaplněná a je v ní klíč Z. Algoritmus tedy ”vykopne” klíč Y a uloží místo něj X.



Obrázek 3: Vložení klíče do cuckoo tabulky

Nyní je potřeba někam uložit prvek s klíčem Y, jeho první hashovací funkce ukazuje na původní adresu, ve které se nacházel a druhá ukazuje na volné místo, kde se ve výsledku uloží.



Obrázek 4: Přehashování klíče v cuckoo tabulce

4.5 NBDS (Non-Blocking Data Structures)

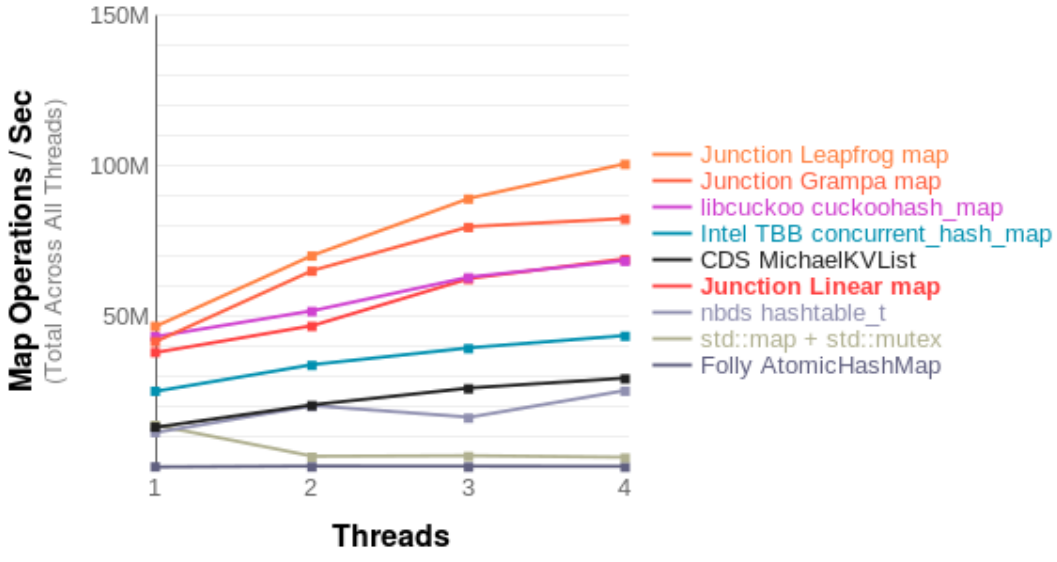
[12][13] Implementace hashované tabulky podporující lock-free přístup pomocí operace CAS. Při zaplnění tabulky disponuje možností zvětšení kapacity tak, že vytvoří novou větší tabulku a staré data do ni přesune, s tím že všechny klíče přehashuje kvůli nové velikosti tabulky.

4.6 CDS (Concurrent Data Structures)

[7][8] Knihovna skládající se z mnoha komponent podporující více vláken. Implementace hashované tabulky využívá CAS operaci pro lock-free přístup. Řešení kolizí a kapacitu tabulky řeší jednoduše metodou separate chaining, tj. řetězení hodnot. Knihovna používá garbage collector, který se stará o správné smazání nepoužívaných instancí či ukazatelů. K tomu využívá tzv. Hazard Pointer[9] a ten funguje tak, že při každém smazání nějakého prvku v hashované tabulky, se smazaný prvek přidá do spojovaného seznamu pro pozdější znovupoužití. Jakmile tento spojovaný seznam dosáhne určitého počtu, spustí se znovu získání využité paměti tak, že se prvky uvolní pomocí *free* a poté se musí použít stejný kus paměti k vytvoření nové instance.

4.7 Porovnání

Při prohlídce gitového repozitáře Junction, jsem narazil na to, že v rámci knihovny je i test, který testuje mnohem více hashovaných tabulek než jen z knihovny Junction a to všechny výše uvedené. Testů je tam hned několik jako například test škálovatelnosti vláken, test využití paměti a další. Na základě výsledku tohoto testu jsem se rozhodl svou implementaci hashované tabulky porovnat s implementací z knihovny Junction.



Obrázek 5: Škálovatelnost hashovaných tabulek

5 Vlastní implementace

5.1 Problém

Vlastní implementaci hashované tabulky jsem přizpůsobil a využil pro řešení problému v soužeži SIGMOD 2018 [20]. Problém spočíval ve spojování několika relací a provedení agregační funkce sum nad požadovanými atributy.

Testování řešení funguje tak, že nejprve programu pošle přes stardartní vstup informace o tom, jaké relace si má načíst - cestu k souborům s daty. Každý soubor s relacemi je binární soubor, který obsahuje hodnoty typu *uint64_t*, což je 64-bitové celé číslo bez záporné části.

Po zpracování dat relací začne testovací program posílat jednotlivé dotazy, které chce zpracovat a jejich výsledek se má vypsat na standartní výstup. Každý dotaz se skládá ze tří částí, které jsou od sebe odděleny znakem svislé čáry '|'.

Ukázka dotazu: 3 0 1 | 0.2=1.0 & 0.1=2.0 & 0.2>3499 | 1.2 0.1

1. část nám říká jaké tabulky budeme v dotazu používat. Reference na tyto tabulky v dotazu je značena čísly nuly až do n podle toho jak jsou tabulky vypsány z leva do prava
2. část obsahuje informace o tom, které sloupce z tabulek se mají spolu spojit a dále taky selekci (podmínku). Jednotlivé podmínky jsou od sebe odděleny znakem &.
3. část je projekce sloupců, nad kterými se provede agregační funkce sum

Výše zmíněný dotaz by se v jazyce SQL dal zapsat jako:

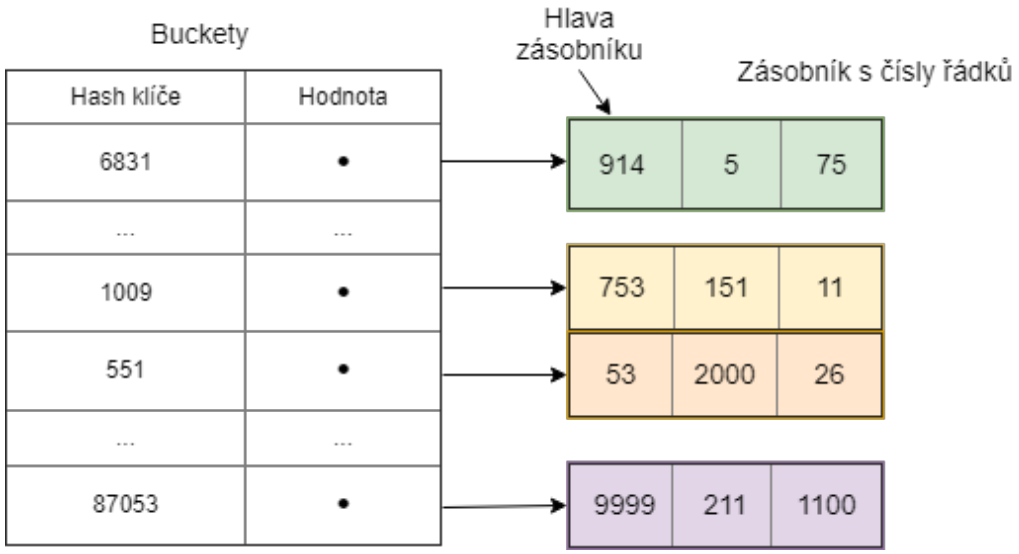
```
1 SELECT SUM(1.2), SUM(0.1)
2 FROM r3 '0' r0 '1' r1 '2'
3 WHERE 0.2 = 1.0 AND 0.1 = 2.0 AND 0.2 > 3499
```

Výpis 2: Ukázka testovacího dotazu v jazyce SQL

Tohle vše musí soutěžící program stihnout provést v co nejkratším čase.

5.2 Implementace lock-free hashované tabulky

Pro potřeby rychlého vykonávání dotazů, jsem využil lock-free hashovanou tabulku. Hashovaná tabulka obsahuje dvojice (klíč, hodnota), přičemž klíčem jsou hodnoty jednoho atributu a hodnotou je odkaz na řádek, kde se hodnota atributu nachází. Implementaci hashované tabulky jsem přizpůsobil pro řešení problému soutěže. Hashovaná tabulka neumožňuje mazání, ani nárust v případě velkého zaplnění.



Obrázek 6: Ukázka použití hashované tabulky v projektu

5.2.1 Zápis do hashované tabulky

Nejprve se z klíče spočítá hash hodnota a získá se bucket, ve kterém je umístěna hodnota (zásobník, možno chápat také jako spojovaný seznam). Tato hodnota je typu *Atomic* z knihovny Turf a to proto, že jelikož se do hashované tabulky přistupuje z více vláken, musíme zajistit, aby si vlákna navzájem nepřepsali hodnoty. Pokud hodnota (zásobník) ještě neexistuje, je potřeba jej vytvořit a tady přichází na řadu operace CAS, kterou zajistíme, že nepřepíšeme hodnotu, pokud už ji jiné vlákno zapsalo. Pokud operace po provedení vrátí *false*, znamená to, že jiné vlákno mezitím také uložilo hodnotu, takže si můžeme již existující zásobník načíst a pracovat s ním.

Pro uložení hodnoty (číslo řádku v tabulce) do zásobníku[4][5] je také využita operace CAS. Ta je volaná tak dlouho, dokud se nepodaří nový záznam vložit do hlavičky zásobníku. Protože pokaždé když operace CAS vrátí *false*, tak to znamená, že jiné vlákno mezitím změnilo hlavičku zásobníku (první uzel) a proto se musíme pokusit znovu o vložení.

Níže je zobrazena implementace vkládání záznamu do hashované tabulky 4 a také do zásobníku 3, který je hodnotou v bucketu hashované tabulky.

```
1 void cStack::Push(uint data, cMemory *memory) {
2
3    cNode * node = new (memory->New(sizeof(cNode))) cNode(data);
4
5    do {
6       node->Next = mHead.loadNonatomic();
7    } while (!mHead.compareExchangeWeak(node->Next, node, turf::Relaxed, turf::
         Relaxed));
8 }
```

Výpis 3: Tělo metody pro vložení prvku do zásobníku

```
1  cStack * cHashTableMy::Put_internal(ColumnType originalKey, ColumnType newKey,
       uint i, cMemory * memory) {
2
3     uint index = newKey % mSize;
4
5     turf::Atomic<cStack *> *atomic = mHashTable[index];
6     cStack * stack = atomic->loadNonatomic();
7
8     if (stack == nullptr) {
9
10       cStack *tmp = new (memory->New(sizeof(cStack))) cStack(originalKey);
11
12       if (!atomic->compareExchangeWeak(stack, tmp, turf::Relaxed, turf::Relaxed
             )) {
13          stack = atomic->loadNonatomic();
14       } else {
15          stack = tmp;
16       }
17    }
18
19    if (stack->Key() != originalKey) {
20       return Put_internal(originalKey, originalKey + (i * i), i + 1, memory);
21    }
22
23    return stack;
24 }
```

Výpis 4: Tělo metody pro vložení prvku do hashované tabulky

5.2.2 Vyhledávání v hashované tabulce

Kód umožňující vyhledávání v hashované tabulce pak vypadá následovně. Je dost podobný vkládání, až na to, že se nezapisuje do zásobníku, ale pouze se z hashované tabulky čte. Hlavně musí být konzistentní řešení kolizí, v tomto případě - linear probing.

```
1  cStack * cHashTableMy::Find_internal(ColumnType originalKey, ColumnType newKey,
       uint i) {
2
3     ColumnType index = newKey % mSize;
4
5     turf::Atomic<cStack *> *atomic = mHashTable[index];
6     cStack * stack = atomic->loadNonatomic();
7
8     if (stack == nullptr) {
9        return nullptr;
10    }
11
12    if (stack->Key() != originalKey) {
13       return Find_internal(originalKey, originalKey + (i * i), i + 1);
14    }
15
16    return stack;
17 }
```

Výpis 5: Tělo metody pro nalezení hodnoty podle klíče v hashované tabulce

5.2.3 Kolize

Kolize klíčů jsem řešil pomocí linear probingu. To znamená, že pokud chci vyhledat klíč, pro který už existuje hodnota, ale tato hodnota byla vytvořena pro jiný klíč, přidám ke klíči postupně 1^2, 2^2, 3^2, 4^2, .. a z něj znovu pomocí zbytku po dělení získám pozici v poli a toto se opakuje tak dlouho, dokud se nepodaří najít buď volný slot nebo existující hodnota s požadovaným klíčem.

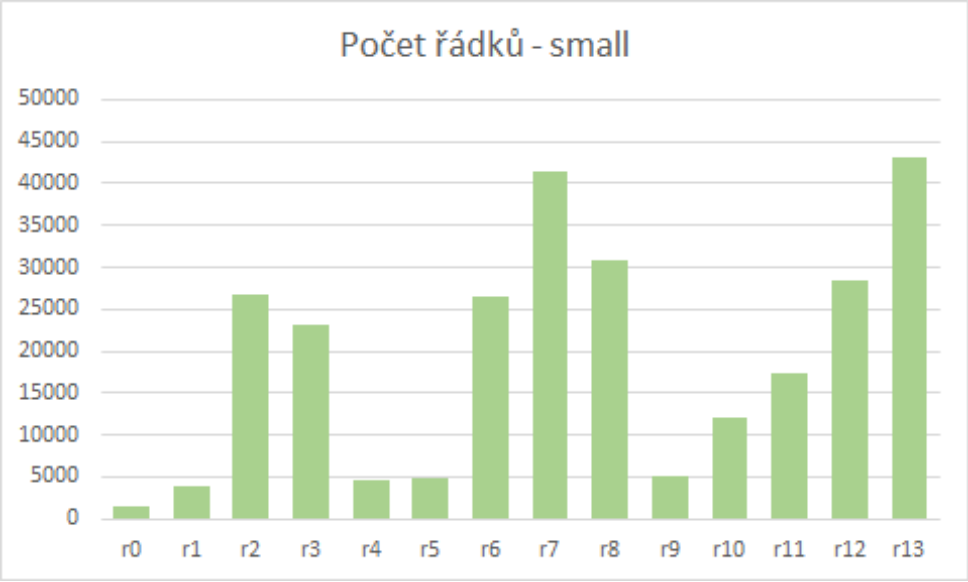
5.3 Vstupní data

Vstupem jsou relace, kde jednotlivé atributy jsou 64 bitová čísla (žádný jiný datový typ se v relacích nevyskytuje).

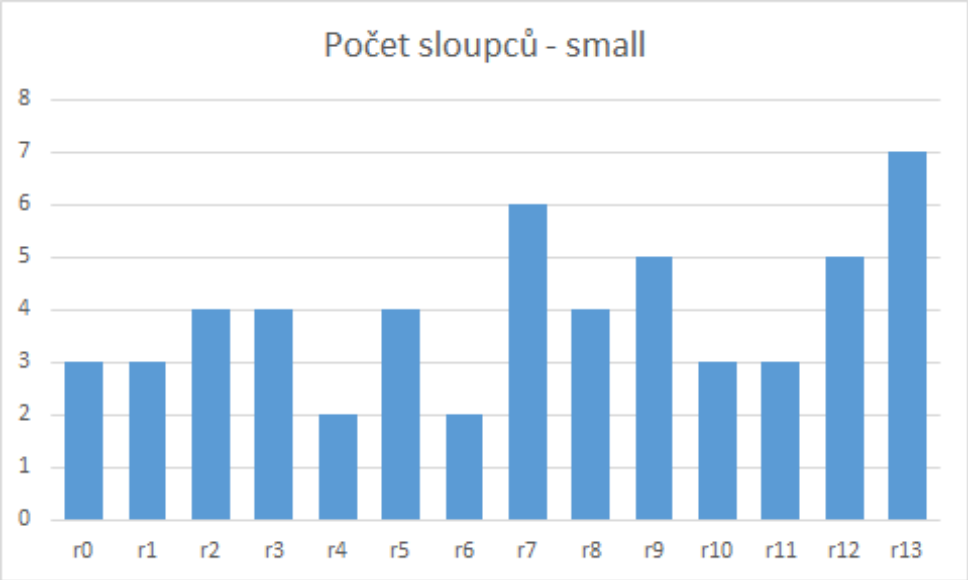
5.3.1 Statistiky kolekce small

Relace	Minimum	Maximum	Průměr
r0	1	10262	4710
r1	4	11410	5731
r2	1	80073	12652
r3	1	69465	12053
r4	4	14076	6352
r5	1	14730	5921
r6	2	79159	22679
r7	1	124644	15355
r8	1	92597	16142
r9	1	14874	5978
r10	1	35961	8696
r11	1	51953	11343
r12	2	85296	14087
r13	1	129328	16088

Tabulka 1: Statistiky (min, max a průměr hodnot) o relacích kolekce small



Obrázek 7: Histogram počtu řádků kolekce small

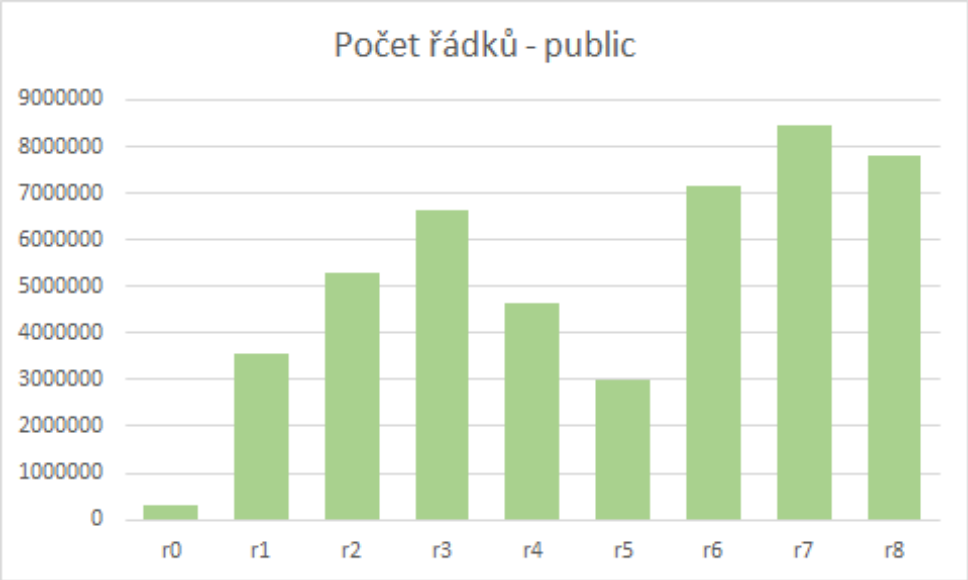


Obrázek 8: Histogram počtu sloupců kolekce small

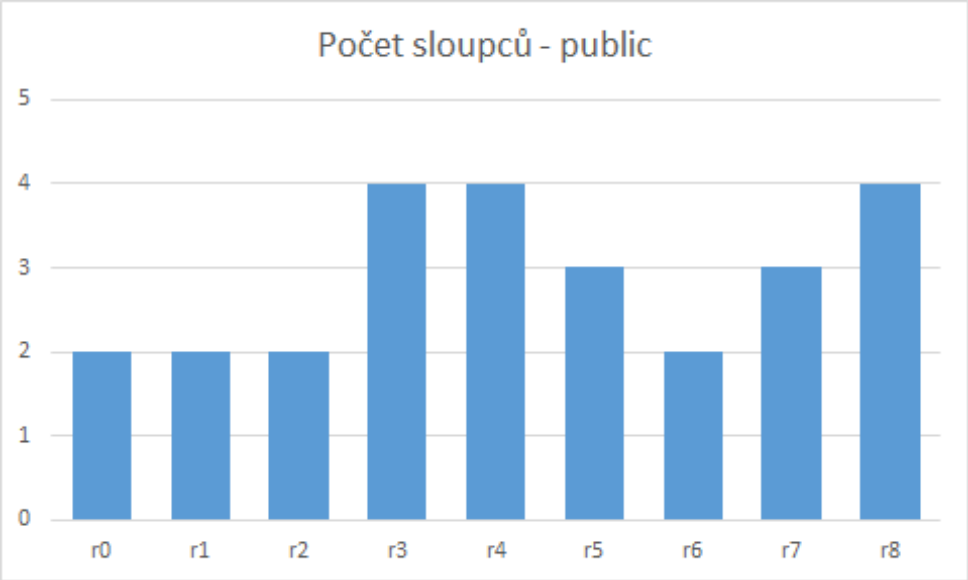
5.3.2 Statistiky kolekce public

Relace	Minimum	Maximum	Průměr
r0	1	938495	238483
r1	2	10675202	2671319
r2	1	15884573	6639624
r3	1	19909499	3942604
r4	1	13893270	3189949
r5	1	10675202	3438216
r6	1	21517061	5614341
r7	1	25411652	4548040
r8	1	23369569	3157867

Tabulka 2: Statistiky (min, max a průměr hodnot) o relacích kolekce public



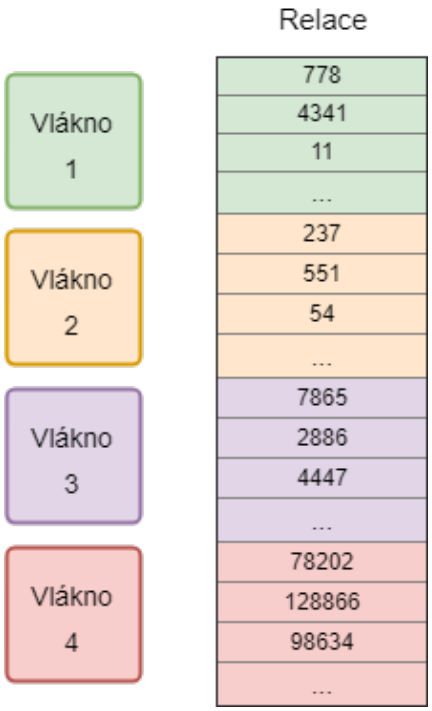
Obrázek 9: Histogram počtu řádků kolekce public



Obrázek 10: Histogram počtu sloupců kolekce public

5.4 Řešení

Hashovaná tabulka se vytváří při načítání relací a to pro každý sloupec vstupní relace zvlášť. Množina hodnot ve sloupci je rozdělena na N disjunktních množin, které jsou zpracovávány odděleně 11 14. Každá část je následně zpracována svým vláknem a jednotlivé řádky jsou zaindexovány do hashované tabulky.



Obrázek 11: Paralelizace indexování

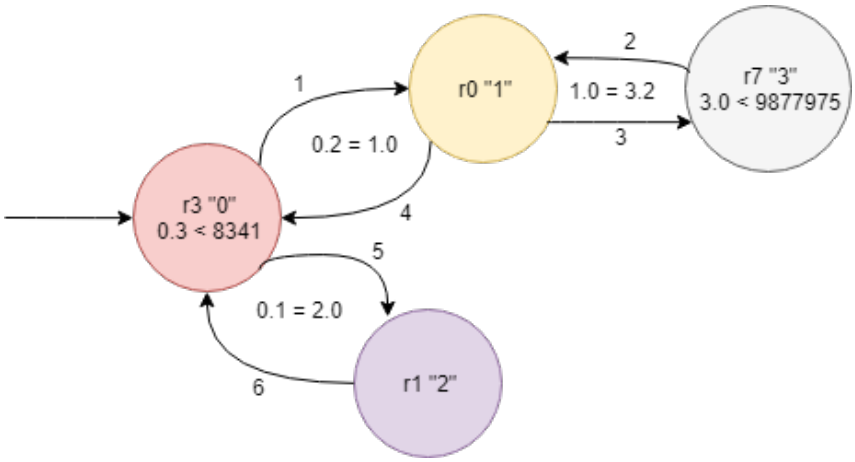
Po zaindexování všech relací, se začnou provádět jednotlivé dotazy. Ty je však potřeba nejprve rozložit na jednotlivé části, tak aby program věděl z jakých tabulek má číst a jak je mezi sebou spojit a nad jakými sloupci má provádět agregaci. K tomu slouží třída a metoda *cQueryModel#Parse*. Určí se tabulka, která bude počáteční pro průchod a to podle podmínek nad jednotlivými tabulkami.

- Největší váhu v rozhodování jsou tabulky, které obsahují v podmínce bodový dotaz, podmínka se znakem rovná se (=), znamená to, že můžeme využít index pro vyhledávání v tabulce, protože hashované tabulky podporují pouze bodové vyhledávání.
- Na druhém místě je rozsahový dotaz v podmínce, znaky menší/větší (< >). U nich sice nemůžeme využít index, ale i přesto se nám selekcí zmenší počet dat, se kterými budeme pracovat.

5.4.1 Spojování tabulek

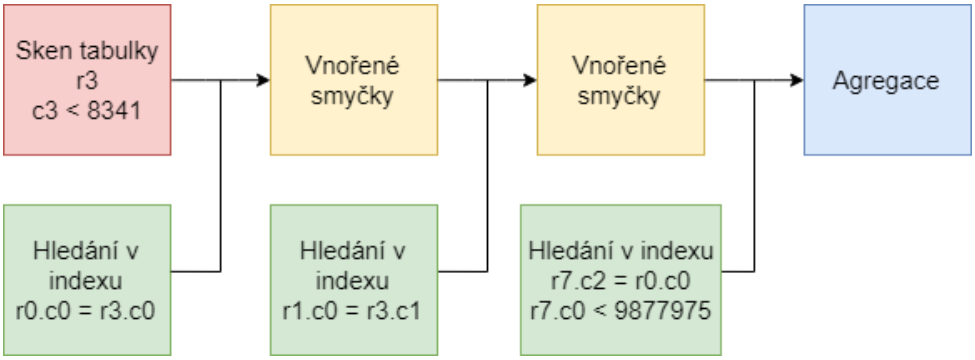
Ukázku procházení tabulek, popíši na dotazu níže.

3 0 1 7 | 0.2=1.0 & 0.1=2.0 & 1.0=3.2 & 0.3<8341 & 3.0<9877975 | 2.1 0.0 1.0



Obrázek 12: Ukázka spojení tabulek

Jako první tabulku pro průchod se program rozhodne pro *r3*, protože se nad ní provede rozsahová podmínka (nad *r7* se taky provádí rozsahová podmínka, ale *r3* je dříve v pořadí při čtení). Pro průchod tabulkou je použitý tzv. iterator 15, který reaguje na podmínky v dotazu a podle toho prochází buď index (bodový dotaz) 7 , nebo řádky filtruje na základě rozsahové podmínky a v případě žádné podmínky musí projít celou tabulku řádek po řádku. Jakmile se najde nějaký vyhovující řádek, začnou se postupně procházet podmínky spojení. Každá podmínka spojení má referenci na levou i pravou tabulku. Začne se tedy procházet pravá tabulky ze spojení a na ni se opět aplikuje zmíněný postup. Pokud algoritmus narazí na poslední tabulku, tak se vrátí na předchozí a pokračuje v průchodu spojení. Jakmile se v každé tabulce na základě všech podmínek podaří najít stejná shoda, zapíše se hodnota řádku pro sloupce v projekci (2.1, 0.0 a 1.0).



Obrázek 13: Plán vykonání dotazu

```
1  void cQueryModel::Run_internal(cTableIterator * firstTable, cArray<
       cTableIterator*>* sortedTables, cArray<cProjection*>* projections) {
2     cTableIterator *ti = firstTable;
3     int level = 0;
4     while (true) {
5
6        while (ti->NextRow()) {
7           if (level == sortedTables->GetSize()) {
8              cProjection::Evaluate(projections);
9           }
10          else {
11             cTableIterator *back = ti;
12             ti = sortedTables->GetItem(level);
13             ti->ClearIterator();
14             ti->SetBackReference(back);
15             level++;
16          }
17       }
18
19       if (ti == firstTable) {
20          break;
21       }
22
23       level--;
24       ti = ti->GetBackReference();
25    }
26 }
```

Výpis 6: Metoda pro spojení a průchod tabulkami

```
bool ret = false;

cCondition *condition = mConditions.GetItem(0);
uint ci = condition->GetIndexColumn();
ColumnType value = condition->GetIndexValue();

if (mIndexNode == nullptr) {
   mIndexNode = mTable->GetIndex(ci)->Find(value);
}
else {
   mIndexNode = mIndexNode->Next;
}

ret = mIndexNode != nullptr;

if (ret) {

   uint rowId = mIndexNode->Data;

   while (!AreAllConditionsMet(rowId, true)) {
      mIndexNode = mIndexNode->Next;
      if (mIndexNode == nullptr) {
         ret = false;
         break;
      } else {
         rowId = mIndexNode->Data;
      }
   }

   if (ret) {
      mRowIndex = rowId;
      mLastItemInIndexFound = mIndexNode == nullptr;
   }

}

return ret;
```

Výpis 7: Metoda NextRow s vyhledáváním v indexu

5.4.2 Optimalizace

Pro urychlení procházení tabulek, jsem využil paralelismu.
Při načítání tabulek z disku, se data zároveň zaindexují, takže je potřeba projít všechny řádky. Toto nám urychlí vlákna, které zpracují části tabulky paralelně 14. V implementaci tohoto způsobu paralelizace jsem se potýkal s problémem paměťové alokace ve vykonávaných vláknech. Operace "new", kterou jsem využíval uvnitř paralelizovaného kódu k vytváření instancí zásobníků a jeho prvků, je velmi náročná obzvlášť v této kritické části kódu. Problém jsem vyřešil tak, že si pro každé vlákno naalokuji blok paměti za pomocí třídy *cMemory* a s ním poté vlákna pracují, takže každé vlákno má svůj blok paměti.

Na začátku před průchodem první tabulkou se vytvoří kopie potřebných informací pro každé vlákno zvlášť a první tabulka se rozdělí na tolik částí, kolik je definováno vláken. Takže když máme tabulku o 1000 záznamech a 4 vlákna, bude každé vlákno zpracovávat 250 rádků. Tohle ovšem neplatí pokud můžeme u první tabulky využít index, provádí se tedy bodový dotaz, který už je sám o sobě rychlý a proto není potřeba spouštět další vlákna.

Další optimalizace se týka alokace paměti. Místo toho, aby se při vytváření nových instancí pomocí ukazatelů alokovala pamět pro každou instanci zvlášť, předalokuje se velký blok paměti, momentálně 10 MB, ze které se podle potřeby "uřízne" potřebný kus paměti a ten se použije pro vytvoření nové instance. Pokud se blok zaplní, alokuje se nový.

```
1 char *mem = mMemory->New(sizeof(cProjection));
2 cProjection *proj = new (mem) cProjection();
```

Výpis 8: Vytvoření instance s předalokovanou pamětí

5.4.3 Argumenty

Pro jednodušší práci s programem, hlavně z hlediska kompilace řešení, jsem přidal pár argumentů, se kterými lze běh aplikace modifikovat bez nutnosti překompilování.

- Nejdůležitější z argumentů je počet vláken, ten určuje kolik vláken je schopna aplikace využít při paralelním indexování dat a také při paralelním vyhledávání.
- Další parametr určuje z jaké složky má číst data a pracovní soubory s testovacími dotazy.
- Parametr "-p" umožňuje vypnout výstup z dotazů a lze tak zobrazit pouze časy vykonání.

```
1 Usage:
2   -t 4          # number of threads to use
3   -d small      # name of folder with workload data (must be placed in the
      build folder)
4   -p false      # turn on/off printing of results
```

Výpis 9: Použitelné argumenty programu

5.4.4 Kompilace a spuštení

Jelikož je program závislý na knihovnách Turf a Junction, využil jsem sestavování testů v knihovně Junction a program vložil mezi jednotlivé testy. Nyní stačí ve složce s programem vytvořit novou složku např. s názvem "build", přepnout se do ní a zavolat nad projektem *cmake* a ten už se o vše postará. Při kompilaci máme možnost sestavit *release* nebo *debug* verzi. Toho docílíme přidáním parametru -DCMAKE_BUILD_TYPE=RelWithDebInfo respektive -DCMAKE_BUILD_TYPE=Debug k příkazu *cmake*.

Program můžeme následně spustit přímo ze složky, ve které jsme ho sestavili. Složka však musí ještě obsahovat testovací kolekci, se kterou program pracuje.

5.5 Řešení s Junction

Využití hashovaných tabulek z knihovny Junction, jsem zakompletoval do třídy *cHashTable*, kde jsou podle potřeby přizpůsobeny i metody pro vkládání a čtení z hashované tabulky.
Většina metod hashované tabulky z Junction pracuje s objektem zvaný *Mutator*, který se chová jako ukazatel na konkrétní slot v hashované tabulce. Když se tedy podíváme na kód pro vkládání, tak atomická metoda *insertOrFind* vytvoří nový slot pro daný klíč nebo vrátí ukazatel na existující slot. Nadbytečné kontroly jsou v kódu kvůli předhánění se vláken, protože než uložíme novou hodnotu (zásobník) do hashované tabulky, jiné vlákno tam již mohlo uložit jinou instanci. Až poté co jsme si jisti, že máme aktuální instanci, můžeme vložit nový záznam do zásobníku.

```
1  void cHashTableJunction::Put(ColumnType key, uint value, cMemory * memory) {
2
3     cStack *stack;
4     auto nodeMutator = mHashMap.insertOrFind(key);
5
6     if ((stack = nodeMutator.getValue()) == nullptr) {
7        cStack *tmp = new (memory->New(sizeof(cStack))) cStack();
8        nodeMutator.exchangeValue(tmp);
9        stack = nodeMutator.getValue();
10    }
11
12    stack->Push(value, memory);
13 }
```

Výpis 10: Tělo metody pro vložení prvku do Junction hashované tabulky

U vyhledávání v hashované tabulce je to mnohem jednodušší a význam je tedy evidentní přímo z kódu.

```
1  cNode * cHashTableJunction::Find(ColumnType key) {
2
3     cStack *stack = mHashMap.get(key);
4
5     if (stack == nullptr) {
6        return nullptr;
7     }
8
9     return stack->Head();
10 }
```

Výpis 11: Tělo metody pro nalezení hodnoty v Junction hashované tabulce

5.6 Struktura programu

5.6.1 cArray

Zaobaluje klasické paměťové pole. Třída má nastaven maximální počet prvků, které se mohou v poli nacházet. Umožňuje přidávat nové prvky, mazat a číst. Dále pak obsahuje metodu Contains, která umí zjistit, zda nějaký prvek je obsažen v poli.

5.6.2 cCondition

Podmínky jsou velmi důležitou součástí aplikace. Tato podmínka byla navržena s využitím dědičnosti. Obsahuje virtuální metody Evaluate, Clone, GetIndexColumn, GetIndexValue a CanUseIndex. Má celkem 4 potomky *cCGreaterCondition*, *cSmallerCondition*, *cEqualCondition* a poslední *cJoinCondition*. První tři jsou hodnotové podmínky a zkontrolují pouze, zda nějaký sloupec a příslušný řádek vyhovují porovnání s hodnotou. Spojovací podmínka je složitější. Obsahuje informace jak o levé, tak o pravé tabulce při spojení a v případě vyhodnocení porovná hodnoty sloupců a konkrétních řádků, které spojuje. Metoda Evaluate vyhodnotí podmínku pro řádek levé a pravé tabulky, na které má podmínka referenci. Clone pouze vytváří kopii samotné podmínky, pro účely paralelizace, kde každé vlákno musí mít své informace o tabulkách a jejich spojení, aby si vlákna navzájem nepřepisovali data při průchodu tabulkami. GetIndexColumn vrátí číslo sloupce z pravé tabulky pro účely vyhledávání v indexu. GetIndexValue vrátí hodnotu na řádku z levé tabulky pro sloupec na základě kterého spojujeme. A poslední CanUseIndex pouze říká, zda můžeme využít indexu při přídání této podmínky do tabulky (pro podmínky spojení a podmínky na rovnost hodnoty vrací *true*).

5.6.3 cHashTable

Implementace hashované tabulky je popsána výše, avšak tato třída zaobaluje implementaci mé vlastní hashované tabulky a hashované tabulky z knihovny Junction. Na základě konstanty USE_JUNCTION_HASH_TABLE, která se nachází v *core.h* se kompilátor rozhodne, kterou implementaci použít a pouze převolává příslušné metody.

```
1 void cHashTable::Put(ColumnType key, uint value, cMemory * memory) {
2    if (USE_JUNCTION_HASH_TABLE) {
3       implJunction->Put(key, value, memory);
4    } else {
5       implMy->Put(key, value, memory);
6    }
7 }
```

Výpis 12: Větvení podle vybrané implementace hashované tabulky

5.6.4 cProjection

Obsahuje informace o třetí částí dotazu (jednotlivé části jsou oddělené znakem "|"). Slouží k ukládání sumy pro konkrétní sloupec. Přičtení hodnoty do sumy probíhá vždy když se při spojování tabulek najde shoda ve všech tabulkách naráz.

5.6.5 cQueryParser

Převede dotaz v textové podobě na objekty - *cTableIterator*, *cJoinCondition* nebo *cValueCondition* a *cProjection*. Při parsování se chová jako stavový automat a postupně prochází dotaz znak po znaku a na základě předem definovaných pravidel vytváří objekty potřebné pro model dotazu.

5.6.6 cMemory

Tato třída je využívána napříč celým projektem. Stará se totiž o alokaci paměti pro nové instance tříd. Tuto paměť poskystuje z velkých bloků paměti, které si dopředu naalokuje. Momentálně má jeden blok velikost 10 MB, pokud si však někdo vyžádá ještě více paměti než 10 MB, alokuje se přesně tolik kolik si volající metody vyžaduje. Neumožňuje jakýmkoliv způsobem znovu získat využitou paměť, takže jakmile si někdo vyžádá kus paměti, už ji nelze získat zpět. Všechny naalokované bloky se automaticky uvolní na konci programu.

```
1  char * cMemory::New(uint size) {
2     if (mPosInBlock + size > BLOCK_SIZE) {
3        mPosInBlock = 0;
4        if (size > BLOCK_SIZE) {
5           mBlocks[++mBlockId] = new char[size];
6        }
7        else {
8           mBlocks[++mBlockId] = new char[BLOCK_SIZE];
9        }
10    }
11
12    char *p = mBlocks[mBlockId] + mPosInBlock;
13    mPosInBlock += size;
14    return p;
15 }
```

Výpis 13: Metoda pro získání paměti z bloku

5.6.7 cQueryModel

Je úzce spjatý s parserem (*cQueryParser*), který využívá k přeložení dotazu. Má na starost spuštění dotazu ve více vláknech, iteraci přes řádky tabulek a jejich následné spojení pro výpočet sumy. Předtím však uspořádá tabulky tak, aby se daly procházet jako graf. Zastává i rozhodování jakou tabulku použít jako počáteční. Rozhodování probíhá na základě podmínek definovaných na tabulce. V praxi chceme začít tou tabulkou, která obsahuje nejméně řádků, takže tabulka, která má podmínku, bude zvolena jako počáteční, akorát podmínka na rovnost má přednost před rozsahovou podmínkou, protože má větší pravděpodobnost, že více zredukuje prvotní tabulku. Spouští průchod relací 6

5.6.8 cStack

Zásobník je popsán výše 3 a využívá se výhradně pro udržování všech výskytů (řádků) hodnot v relacích.

5.6.9 cTable

Tato třída se stará o načtení relací ze souboru a jejich následné zaindexování 11. Udržuje indexy všech sloupců a veškeré data uchovává matici (počet sloupců x počet řádků).

```
1 void cTable::InsertIntoIndex(TableType *columnData, TIndex *index, uint loRowId
      , uint hiRowId, cMemory *memory) {
2
3    for (uint i = loRowId; i < hiRowId; ++i) {
4       index->Put(columnData[i], i, memory);
5    }
6 }
```

Výpis 14: Metoda pro zaindexování části relace v novém vlákně

5.6.10 cTableIterator

Jak už název napovídá, tato třída využívá návrhového vzoru iterator, který zajišťuje možnost procházení prvků bez znalosti jejich implementace.
Hlavní metoda využita pro iteraci se nazývá *NextRow*, která jednoduše dělá to, že pokud při zavolání najde v tabulce řádek, tak zastaví průchod a vrátí hodnotu *true*, kterou mohou využít volající třídy a na jejím základě třeba přestoupit na další tabulku ve spojení nebo pričíst hodnotu řádku do sumy. Uvnitř metody se toho ale děje více, pokud je v tabulce možnost využití indexu, volá se metoda, která prochází index na základě podmínek na rovnost. Pokud však index využít nelze, ale i přesto jsou na tabulce nějaké podmínky, volá se metoda pro sekvenční sken, který zároveň vyhodnocuje podmínky nad každým řádkem. A pokud tedy nelze využít index a zároveň nemá tabulka žádné podmínky, iteruje se přes všechny řádky v tabulce.

```
1  bool cTableIterator::NextRow()
2  {
3     if (mCanUseIndex)
4     {
5        return NextRow_index();
6     }
7     else if (mConditionUsed)
8     {
9        return NextRow_condscan();
10    }
11    else
12    {
13       return NextRow_scan();
14    }
15 }
```

Výpis 15: Metoda NextRow u iteratoru

5.7 Umístění v souteži

Při implementaci jsem se pokusil program odevzdat do soutěže SIGMOD 2018 [20], ale bohužel, program kvůli drobné cybě v kódu neprošel přes kolekce Large a X-Large. Problém spočíval v tom, že v některých případech se nevyužíval index, ale sekvenční průchod tabulkami, i přesto, že bylo možné index použít. Tento problém jsem později opravil, ale to už bylo pozdě, protože bylo po soutěži, která končila 8.4.2018 v 23:59 CEST. Takže výsledky opravené verze se už nedozvím. Níže je alespoň ukázka časů odevzdaných řešení, kde rychlejší běh je řešení s vlákny a pomalejší je bez vláken. Pro porovnání, první tým v žebříčku soutěže dosáhl pro kolekci *Small* času 0.027 sekund a pro kolekci *Medium* 0.133 sekund.

Your Submissions:

Submitted	Small (seconds)	Medium (seconds)	Large (seconds)	X-Large (seconds)	Status	Log	Notes
Apr 08, 2018 22:51:07 Europe/Berlin	0.106	2.835	failed	failed	✖ fail	stdout stderr	
Apr 08, 2018 18:01:26 Europe/Berlin	0.605	15.961	failed	failed	✖ fail	stdout stderr	

Obrázek 14: Odevzdané řešení do soutěže

Rank	Team	Time in the Lead	Small (seconds)	Medium (seconds)	Large (seconds)	X-Large (seconds)	Submitted
#1	Quickstep (formerly Robin) (University of Wisconsin Madison)	5 days 11:18:17	0.027	0.133	0.547	1.475	Apr 08, 2018 23:43:41 Europe/Berlin
#2	vsb_ber0134 (VSB_TUO)	23 days 07:50:47	0.035	0.083	1.261	2.396	Apr 08, 2018 14:08:16 Europe/Berlin
#3	Dataurus (University of Athens)	-	0.055	1.235	1.246	2.754	Apr 08, 2018 20:58:50 Europe/Berlin
#4	PaperCup (Hanyang University)	4 days 21:02:10	0.043	0.783	1.429	3.553	Apr 08, 2018 21:58:03 Europe/Berlin
#5	FloMiGe (SAP HANA Students)	01:43:30	0.056	0.935	1.910	6.350	Apr 08, 2018 20:22:11 Europe/Berlin

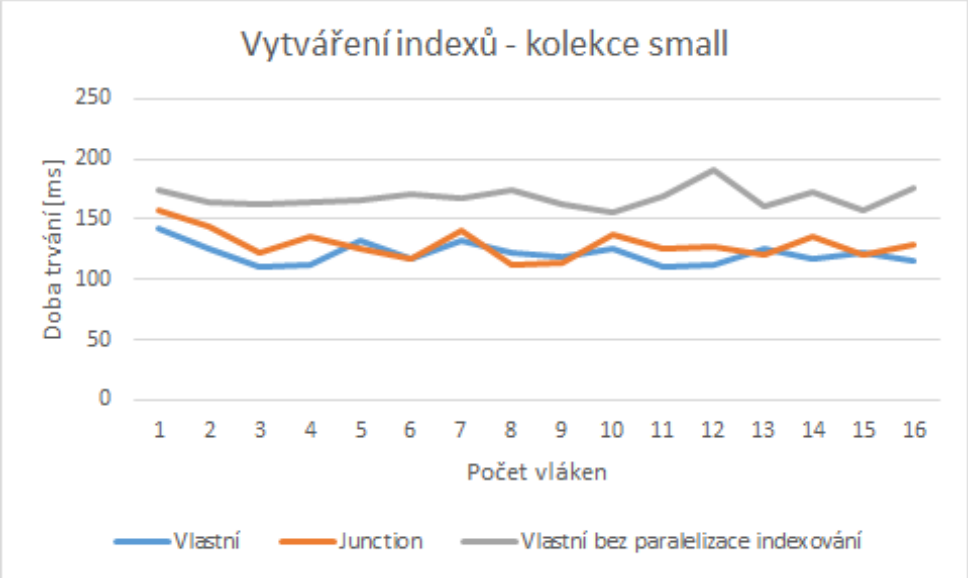
Obrázek 15: Časy prvních pěti soutěžících

6 Porovnání výkonu vlastní implementace s Junction

Testování proběhlo na školním virtuálním stroji s operačním systémem Ubuntu a konfigurací o 8 jádrovém procesoru a 32 GB paměti. Měření pro každou kolekci jsem provedl třikrát a udělal průměr z těch tří měření. Výsledky jsou rozdělené na dvě části - indexování, tedy zápis do hashovaných tabulek a provádění dotazů, takže vyhledávání v hashových tabulkách. Do testování jsem zahrnul i původní verzi bez hashované tabulky podporující paralelní operace, u které jsou tedy paralelizované pouze dotazy, protože ty jenom z hashované tabulky čtou, zápis (vytváření indexů) však nikoliv.

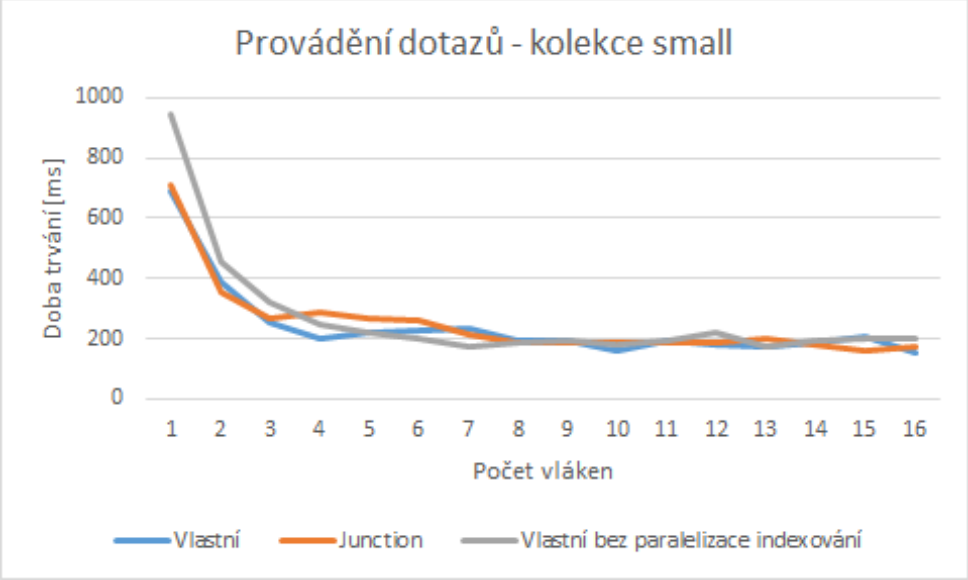
6.1 Kolekce ”small”

Kolekce ”small” byli první testovací soubory, které pořadatelé soutěže vystavili, aby si soutěžící měli na čem testovat své řešení. Dohromady má databáze pouhých 9.35 MB.
Při indexování se škálovatelnost neprojevila, protože jsou kolekce velmi malé a více vlákny už to zrychlit nešlo. Celý běh proběhl řádově v milisekundách a obě implementace se tady vykonostně shodovali. Nejlepší čas indexace u mé implementace je 110 ms a u Junction je to 112 ms, což je srovnatelné.



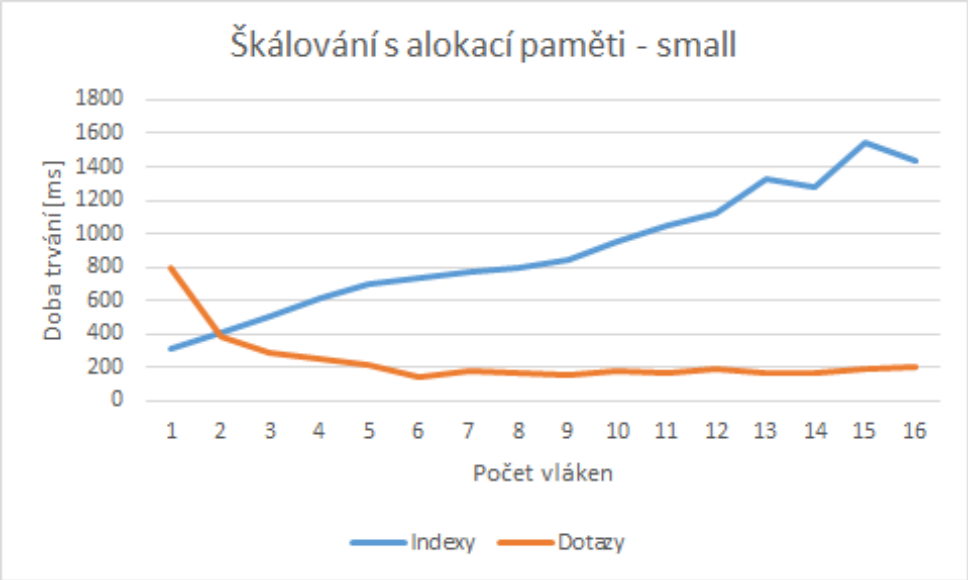
Obrázek 16: Škálovatelnost vláken - indexování ”small”

U provádění dotazů se výkon snížil vyrazně při paralelním zpracování. Nejrychlejší běh s mou implementací trval 155 ms a s Junction 158 ms, což je opět srovnatelné.



Obrázek 17: Škálovatelnost výkonu vláken - dotazy ”small”

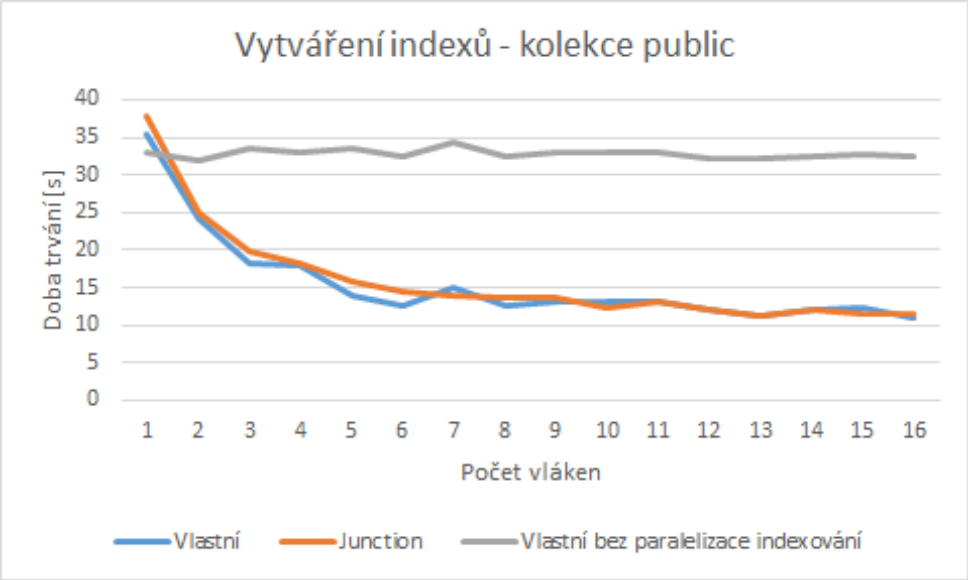
Změřil jsem také problém s alokací paměti v paralelizovaných částech kódu a opravdu jde vidět, že čím více vláken paralelně vytváří nové instance klíčovým slovem *new* tím je výkon horší, protože při každé alokaci paměti se vlákna synchronizují pomocí zámku.



Obrázek 18: Škálovatelnost výkonu s alokací paměti ve vláknech - ”small”

6.2 Kolekce ”public”

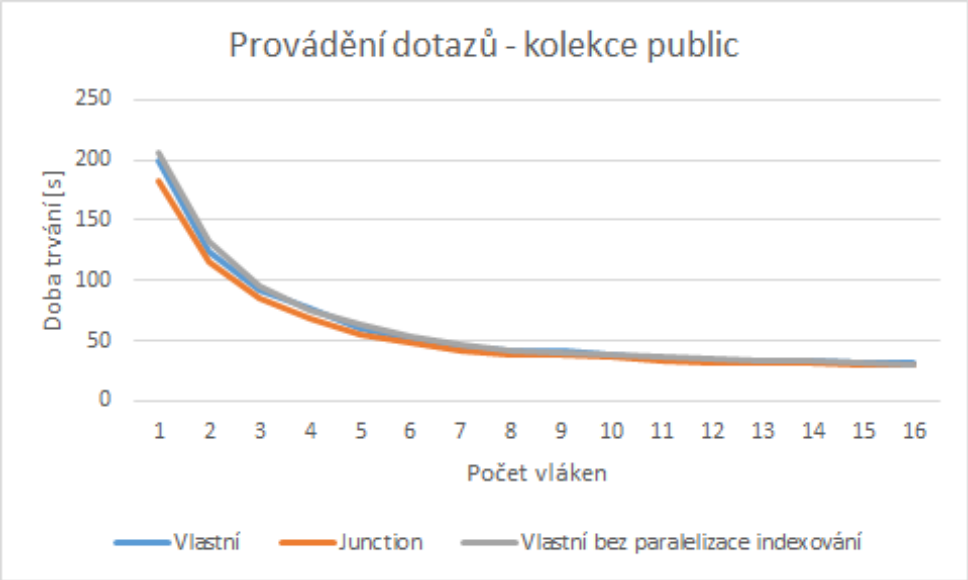
Kolekce ”public” byla později vydána pro testování s větším počtem dat. Celá databáze má dohromady 1.06 GB. Běh programu trval řádově desítky sekund. Nejrychlejší běh s mou implementací trval 11.1 sekund a Junction 11.3 sekund. Junction má o trochu plynulejší křivku škálování vláken. Tato kolekce je i mnohem náročnější na paměť, protože si načtením všech tabulek do paměti a vytvořením indexů program vezme kolem 12 GB v RAM.



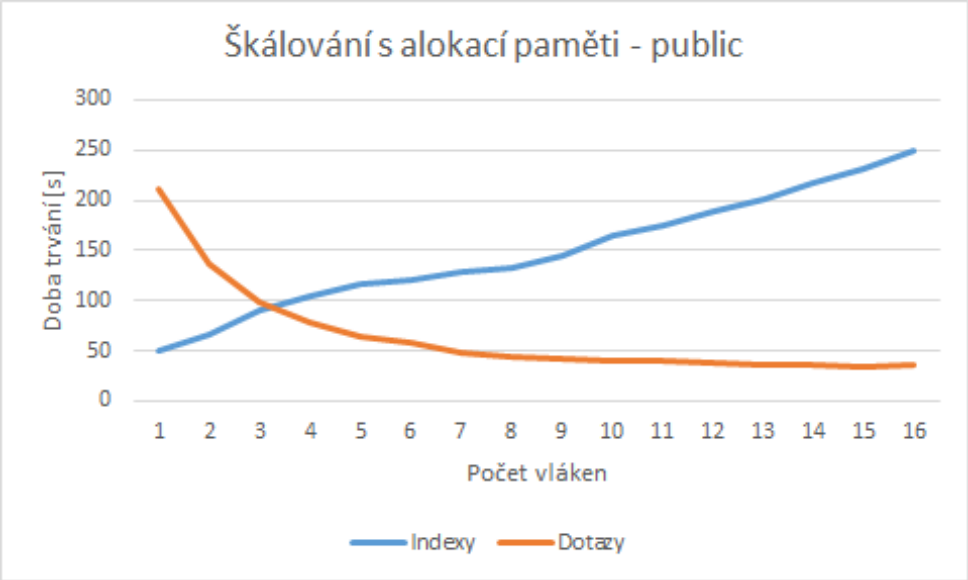
Obrázek 19: Škálovatelnost výkonu vláken - indexování ”public”

Při provádění dotazů byl nejlepší běh u mé implementace 31.2 sekund a u Junction 29.8 sekund.

Tak jako u kolekce ”small”, výkon indexování se výrazně zhoršovali při více vláknech.



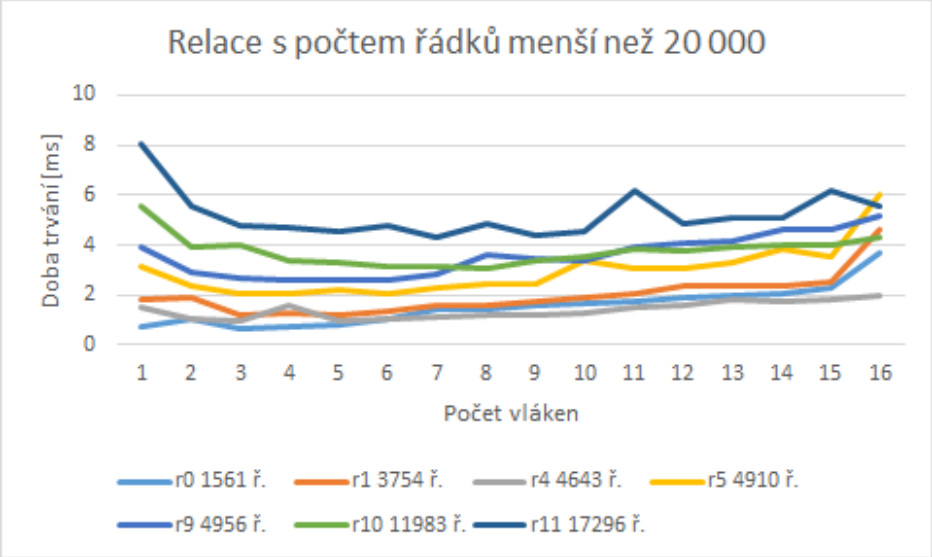
Obrázek 20: Škálovatelnost vláken - dotazy ”public”



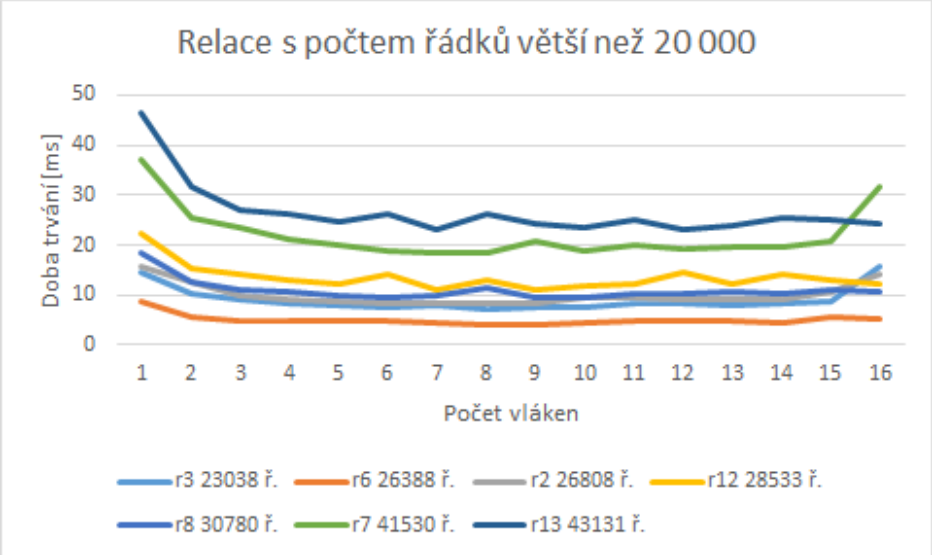
Obrázek 21: Škálovatelnost výkonu s alokací paměti ve vláknech - ”public”

6.3 Experiment výkonu vláken u malých relací

U implementace indexování relací jsem přemýšlel, zda vlákna používat až od určitého počtu řádků relací. Proto jsem udělal malý pokus na to, jestli se výkon sníží pokud bude malou tabulku indexovat příliš mnoho vláken. Meření je rozděleno do dvou grafů, první ukazuje výkon u relací s počtem řádků menší jak 20 000 a druhý ukazuje výkon u relací s počtem řádků větší jak 20 000. Změřena byla pouze kolekce ”small”, protože v kolekci ”public” jsou pouze větší relace.



Obrázek 22: Škálovatelnost výkonu vláken při indexování menší relací



Obrázek 23: Škálovatelnost výkonu vláken při indexování větších relací

Z obou grafů můžeme vidět, že ikdyž jsou relace malé, rozdělení práce do více vláken má i tak smysl.

7 Závěr

Cílem této práce bylo vytvoření a odlazení vlastní hashované tabulky, která podporuje přístup z více vláken. S tím souvisí i rešerše stávajících řešení hashované tabulky a výkonnostní porovnání vlastní implementace.
Popsal jsem jak taková hashovaná tabulka funguje, na jaké problémy můžeme narazit při práci s ní a jakým způsobem je řešit. Složitější rozhodně bylo to, aby hashovaná tabulka podporovala souběh, tedy přístup z více vláken. Od začátku jsme byli s vedoucím práce domluvení, že hashovaná tabulka bude lock-free, aby byl výkon co nejlepší.

Rešerší dalších knihoven, které podporují přístup z více vláken, jsem se rozhodl 4.7 pro porovnání mé vlastní implementace oproti Junction, se kterou má moje tabulka srovnatelný výkon, ale to je hlavně proto, že jsem hashovanou tabulku přizpůsobil pro potřeby řešení problému soutěže.

Odladit cokoliv co pracuje s více vlákny je vždy problém, protože můžeme narazit na problémy popisované v 3. I při vypracování této práce jsem na problémy s vlákny narazil a jelikož nejsem tak zkušený v C++ (většinou programuji v Javě), tak jsem se s některými věcmi vypořádával pomaleji.

Při implementaci jsem se značnou část času zdržel u špatných výkonnostních výsledků při zapisování do hashované tabulky z více vláken 5.4.2. Tento problém nakonec spočíval v tom, že při vytváření nové instance třídy respektive alokace paměti musí být tato operace synchronizována a v tomto případě to byla synchronizace zámky. Takže vlákna jak postupně alokovala pamět, tak musela čekat jedno na druhé 21, což mělo za následek velkou degradaci výkonu. Problém jsem nakonec vyřešil předalokováním paměti předtím, než se spustí vlákna.

Díky této práci jsem si rozhodně zlepšil znalosti C++ jako jsou dědičnost, práce s pamětí, se kterou jsem měl dost často problémy (chyby typu *Segmentation fault*) a práci s vlákny. Po celou dobu implementace jsem používal profiler zvaný *perf*, kterým jsem vždy analyzoval výkon programu. Zvykl jsem si na nové prostředí - Ubuntu, které jsem používal hlavně proto, že při kompilaci a spuštění programu s knihovnou Junction na Windows, program vždy spadl na nějakou chybu. I přesto, že jsem se to snažil řešit s autorem Junction knihovny na Github, tak se mi do teď nedostala odpověď.

Největší přínos mi však tato práce dala ve znalosti lock-free řešení s využitím atomických instrukcí CAS (CMPXCHG) přímo na procesoru. Předtím jsem neměl tušení, že něco takového je vůbec možné, takže jsem si určitě rozšířil obzor. Dále pak byli zajímavé různé řešení hashování.

Je škoda, že se mi nepodařilo umístit v soutěži SIGMOD 2018, protože mě zajímá, kde bych se v žebříčku umístil s touto implementací hashované tabulky.

Literatura

[1] PRESHING, Jeff. Junction - Concurrent data structures in C++. *Github* [online]. 2016 [cit. 2018-03-24]. Dostupné z: `https://github.com/preshing/junction`

[2] PRESHING, Jeff. Turf - Configurable C++ platform adapter. *Github* [online]. 2016 [cit. 2018-03-24]. Dostupné z: `https://github.com/preshing/turf`

[3] PRESHING, Jeff. New Concurrent Hash Maps for C++. *Preshing on Programming* [online]. 2016 [cit. 2018-03-24]. Dostupné z: `http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/`

[4] LANGDALE, Geoff. Lock-Free Programming. *Carnegie Mellon University* [online]. 2014 [cit. 2018-06-15]. Dostupné z: `https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf`

[5] WELLONS, Chris. C11 Lock-free Stack. *null program* [online]. 2014 [cit. 2018-05-28]. Dostupné z: `https://nullprogram.com/blog/2014/09/02/`

[6] INTEL. Intel® Threading Building Blocks Documentation. *Intel Software Developer Zone* [online]. 2017 [cit. 2018-04-20]. Dostupné z: `https://software.intel.com/en-us/node/506085`

[7] KHIZHINSKY, Max. libcds - CDS C++ library. *Github* [online]. 2014 [cit. 2018-04-10]. Dostupné z: `https://github.com/khizmax/libcds`

[8] KHIZHINSKY, Max. CDS: Concurrent Data Structures library. *cds* [online]. 2014 [cit. 2018-04-10]. Dostupné z: `http://libcds.sourceforge.net/doc/cds-api/index.html`

[9] MICHAEL, Maged. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *Computer Action Team* [online]. 2004 [cit. 2018-06-15]. Dostupné z: `http://web.cecs.pdx.edu/~walpole/class/cs510/papers/11.pdf`

[10] LI, Xiaozhou, ANDERSEN, KAMINSKY a FREEDMAN. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. *Princeton University - Department of Computer Science* [online]. 2014 [cit. 2018-05-15]. Dostupné z: `http://www.cs.princeton.edu/~mfreed/docs/cuckoo-eurosys14.pdf`

[11] GOYAL, Manu. libcuckoo *Github* [online]. 2014 [cit. 2018-04-10]. Dostupné z: `https://github.com/efficient/libcuckoo`

[12] DYBNIS, Josh. nbds *Github* [online]. 2008 [cit. 2018-04-10]. Dostupné z: `https://github.com/argv0/nbds`

[13] DYBNIS, Josh. nbds *Google Arvhive* [online]. 2008 [cit. 2018-04-10]. Dostupné z: https://code.google.com/archive/p/nbds/

[14] LÁNSKY, Lukáš. Kolize hashů pro mírně pokročilé *funkcionálně.cz* [online]. 2016 [cit. 2018-06-05]. Dostupné z: http://funkcionalne.cz/2016/02/kolize-hashu-pro-mirne-pokrocile/

[15] DAVID, Tudor, GUERRAOUI a TRIGONAKIS. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask *SIGOPS* [online]. 2013 [cit. 2018-05-18]. Dostupné z: http://sigops.org/sosp/sosp13/papers/p33-david.pdf

[16] BOSMAN, Tudor, RAO a DELONG. folly/AtomicHashmap.h *Github* [online]. 2016 [cit. 2018-05-18]. Dostupné z: https://github.com/facebook/folly/blob/master/folly/docs/AtomicHashMap.md

[17] MAIER, Tobias, SANDERS a DEMENTIEV. Concurrent Hash Tables: Fast and General(?)! *Cornell University Library* [online]. 2016 [cit. 2018-04-08]. Dostupné z: https://arxiv.org/pdf/1601.04017.pdf

[18] THIBAUT. Hopscotch hashing *Programming blog* [online]. 2016 [cit. 2018-06-21]. Dostupné z: https://tessil.github.io/2016/08/29/hopscotch-hashing.html

[19] CLOUTIER, Felix. CMPXCHG — Compare and Exchange *x86 and amd64 instruction reference* [online]. 2018 [cit. 2018-06-19]. Dostupné z: https://www.felixcloutier.com/x86/CMPXCHG.html

[20] KIPF, Andreas. ACM SIGMOD 2018 Programming Contest *ACM SIGMOD 2018 Programming Contest* [online]. 2018 [cit. 2018-03-18]. Dostupné z: http://sigmod18contest.db.in.tum.de/task.shtml

8 Seznam příloh

Obsah CD

Tabulka popisuje obsah jednotlivých složek a souborů na disku.

Adresář/Soubor	**Popis**
src	Oddělený zdrojový kód celého řešení
test	Řešení připravené ke kompilaci společně s knihovnou Junction a Turf
thesis	Tato práce
build.sh	Zkompiluje celé řešení ve složce *test* pomocí cmake
public.zip	Archiv velké kolekce ”public”
run.sh	Spustí zkompilované řešení s 4 vlákny a kolekcí ”small”

Tabulka 3: Obsah CD

Kompilovatelné řešení je vloženo ve složce /test/junction/samples/JunctionTest. Knihovna Junction už má přípravu pro testy, takže jsem své řešení přidal jako další ”test”, pro snažší kompilaci a vyřešení závislostí.