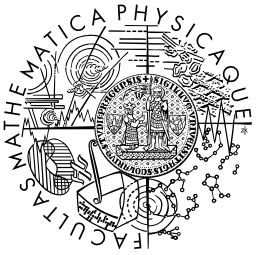


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Martin Gora

Vylepšení agregace dotazovacího enginu pro grafové databáze

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Tomáš Faltín
Studijní program: Informatika
Studijní obor: Softwarové a datové inženýrství

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Chtěl bych poděkovat mému vedoucímu Mgr. Tomáši Faltínovi za jeho pomoc, ochotu a nadšení při zpracovávání daného tématu. Déle bych chtěl poděkovat rodině, která mi poskytla zázemí pro práci a plnou podporu.

Název práce: Vylepšení agregace dotazovacího enginu pro grafové databáze

Autor: Martin Gora

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Tomáš Faltín, Katedra softwarového inženýrství

Abstrakt: Abstrakt.

Klíčová slova: grafové databáze agregace dat proudové systémy

Title: Improvement of data aggregation in query engine for graph databases

Author: Martin Gora

Department: Department of Software Engineering

Supervisor: Mgr. Tomáš Faltín, Department of Software Engineering

Abstract: Abstract.

Keywords: graph databases data aggregation streaming systems

Obsah

Úvod

Tady ma byt text.

1. prvni

aaa Anděl (2007, Věta 4.22) aa

2. druha

3. treti

4. Experiment

Aby bylo možné porovnat stávající řešení s nově navrženým řešením na poli rychlosti zpracovávání dotazů a ověřit naše předpoklady, podrobili jsme zmíněná řešení experimentu. Vykonaný experiment proběhne na reálných grafech různé velikosti s uměle vygenerovanými vlastnostmi naležící vrcholům. Nad danými grafy provedeme vybrané množství dotazů, které nám umožní sledovat a porovnat chování řešení v různých situacích. Experiment bude zakončen diskuzí nad výsledky.

4.1 Příprava dat

Pro náš experiment jsme použili tři orientované grafy z databáze SNAP[1].

	#Vrcholů	**#Hran**
Amazon0601	403394	3387388
WebBerkStan	685230	7600595
As-Skitter	1696415	11095298

Tabulka 4.1: Vybrané grafy pro experiment

- **Amazon0601:** Jedná se o graf vytvořený procházením webových stránek Amazonu na základě featury „Customers Who Bought This Item Also Bought“ ze dne 1.6.2003. V grafu existuje hrana z i do j, pokud je produkt i často zakoupen s produktem j.
- **WebBerkStan:** Graf popisuje odkazy webových stránek domén `https://www.stanford.edu/` a `https://www.berkeley.edu/`. Vrcholem je webová stránka a hrana představuje hypertextový odkaz mezi stránkami.
- **As-Skitter:** Topologický graf internetu z roku 2005 vytvořený programem `traceroutes`. Ačkoliv je uvedeno, že daný graf je neorientovaný, vnitřní hlavička souboru uvádí opak, proto jsme se daný graf rozhodli přesto využít.

Samotné grafy obsahují pouze seznam hran. Abychom mohli dané grafy využít, bylo nutné je transformovat a vygenerovat k nim Properties na vrcholech. Při příkladu transformace budeme vycházet z následující ukázky hlavičky (graf Amazon0601):

```
# Directed graph (each unordered pair of nodes is saved once):
    Amazon0601.txt:
# Amazon product co-purchaisng network from June 01 2003
# Nodes: 403394 Edges: 3387388
# FromNodeId        ToNodeId
0        1
0        2
0        3
0        4
```

[1] Leskovec a Krevl (2014)

4.1.1 Transformace grafových dat

Výstupem transformace budou soubory popisující schéma vrcholů/hran NodeTypes.txt/EdgeTypes.txt a datové soubory vrcholů/hran Nodes.txt/Edges.txt. V našem případě graf bude obsahovat pouze jeden typ hrany a jeden typ vrcholu. Dané omezení ovlivňuje pouze vyhledávání vzoru, které není určující pro náš experiment.

Ukázka zvoleného schématu pro Nodes.txt/Edges.txt:

```
Soubor EdgeTypes.txt:
[
{ "Kind": "BasicEdge" }
]

Soubor NodeTypes.txt:
[
{ "Kind": "BasicNode" }
]
```

Generování souborů Edges.txt/Nodes.txt provádí program, který je obsahem přílohy zdrojových kódů A.1 v souboru GrapDataBuilder.cs. Výstupní soubor Edges.txt bude obsahovat hrany v rostoucím pořadí dle položky `FromNodeId` z originálního souboru s přidělenými `IDs` od hodnoty `ID` posledního vrcholu v souboru Nodes.txt. Samotný soubor Nodes.txt obsahuje setříděné vrcholy podle `ID` v rostoucím pořadí. Je nutné zmínit, že setřídění dat podle `ID` není nežádoucí, jelikož nezaručuje nic o seskupení vrcholů v daném grafu. Pro připomenutí zmíníme, že prvni sloupeček v datových souborech Edges.txt a Nodes.txt odpovídá unikátnímu `ID` v rámci celého grafu.

Následuje ukázka výstupních souborů transformace pro graf Amazon0601:

```
Soubor Edges.txt:
403395 BasicEdge 0 1
403396 BasicEdge 0 2
...

Soubor Nodes.txt:
0 BasicNode
1 BasicNode
...
```

4.1.2 Generování Properties vrcholů

Posledním krokem přípravy dat pro experiment je vygenerování Properties vrcholů. Jsme si vědomi, že nejideálnější způsob testování je graf s reálnými daty. Nicméně, dané omezení jsme se rozhodli aplikovat kvůli problematickému hledání vhodných dat, které nevyžadují netriviální transformaci do vhodného vstupního formátu. Proto pro každý vrchol náhodně vygenerujeme hodnoty tří Properties.

Property	Type	Popis
PropOne	integer	`Int32` s rozsahem $[0, 100000]$
PropTwo	integer	`Int32` s rozsahem [`Int32.MinValue`, `Int32.MaxValue`]
PropThree	string	délka $[2, 8]$ ASCII znaků s rozsahem $[33, 126]$

Tabulka 4.2: Generované Properties vrcholů

- **PropTwo** hodnoty jsou rovněž generováný střídavě kladně a záporně, aby nastal rovnoměrný počet záporných a kladných hodnot.
- **PropThree** hodnoty jsou pouze ASCII znaky z rozsahu $[33, 126]$. Dané omezení výplývá z vlastností dotazovacího enginu, aby bylo možné bez obtíží načíst datový soubor.

Na základě tabulky generovaných Properties 4.2 následuje ukázka upraveného souboru schématu pro vrcholy:

```
Soubor NodeTypes.txt:
[
{
"Kind": "BasicNode",
"PropOne": "integer",
"PropTwo": "integer",
"PropThree": "string"
}
]
```

Výsledné hodnoty Properties do souborů Edges.txt/Nodes.txt jsou vygenerovány pomocí programu, který používá generátor náhodných čísel. Program je obsažen v příloze zdrojových kódů A.1 v souboru PropertyGenerator.cs. Pro každý graf bylo použité jiné `Seed` pro inicializaci náhodného generátoru. Samotná `Seeds` byla vygenerována rovněž náhodně.

	Seed
Amazon0601	429185
WebBerkStan	20022
As-Skitter	82

Tabulka 4.3: Inicializační hodnoty náhodného generátoru pro PropertyGenerator.cs

Program generuje hodnoty definované ve statické položce `propGenerators` a zachovává jejich pořadí ve výsledném datovém souboru. Aby nedocházelo k omylům při opakování experimentů, uvádíme útržek kódu použité inicializace položky dle tabulky generovaných vlastností 4.2 pro všechny tři grafy:

```
static PropGenerator[] propGenerators = new PropGenerator[]
{
    new Int32Generator(0, 100_000, false),
    new Int32Generator(true),
    new StringASCIIGenerator(2, 8, 33, 126)
};
```

Timto jsme dokončili poslední nutný krok k vygenerování platných vstupních dat pro dotazovací engine. Použité grafy k transformaci a výsledné datové soubory jsou obsahem přílohy grafů pro experiment A.3

4.2 Výběr dotazů

Dotazy použité při experimentu dělíme do tří kategorií a to Match, Order by a Group by. Pro připomenutí zmíníme, že proměnné použité v jiných částech než Match způsobují ukládání daných proměnných do tabulky. Přidělené zkratky dotazům budou uváděný ve výsledcích experimentu namísto celých dotazů.

4.2.1 Dotazy Match

Každý dotaz provádí vyhledáváním vzoru v grafu. Níže zmíněné dotazy nám při experimentu pomohou oddělit čas agregací od času stráveném vyhledáváním vzoru.

Zkratka	Dotaz
M_Q1	select count(*) match (x) -> (y) -> (z);
M_Q2	select x match (x) -> (y) -> (z);
M_Q3	select x, y match (x) -> (y) -> (z);
M_Q4	select x, y, z match (x) -> (y) -> (z);

Tabulka 4.4: Dotazy Match

- M_Q1 testuje pouze dobu strávenou vyhledáváním vzoru.
- M_Q2 testuje vyhledávání společně s ukládáním proměnné x do tabulky výsledků.
- M_Q3 testuje vyhledávání společně s ukládáním proměnné x a y do tabulky výsledků.
- M_Q4 testuje vyhledávání společně s ukládáním proměnné x, y a z do tabulky výsledků.

4.2.2 Dotazy Order by

Zkratka	Dotaz
O_Q1	select y match (x) -> (y) -> (z) order by y;
O_Q2	select y, x match (x) -> (y) -> (z) order by y, x;
O_Q3	select x.PropTwo match (x) -> (y) -> (z) order by x.PropTwo;
O_Q4	select x.PropThree match (x) -> (y) -> (z) order by x.PropThree

Tabulka 4.5: Dotazy Order by

- O_Q1 testuje třídění podle `ID` vrcholů y.
- O_Q2 přidává do kontextu O_Q1 overhead za porovnávání a ukládání další proměnné.
- O_Q3 testuje třídění náhodně vygenerovaných hodnot Int32 (viz 4.2).
- O_Q4 testuje třídění náhodně vygenerovaných řetězců (viz 4.2).

4.2.3 Dotazy Group by

Zkratka	Dotaz
G_Q1	select min(y.PropOne), avg(y.PropOne) _M;
G_Q2	select min(y.PropOne), avg(y.PropOne) _M group by y;
G_Q3	select min(y.PropOne), avg(y.PropOne) _M group by y, x;
G_Q4	select min(x.PropOne), avg(x.PropOne) _M group by x.PropTwo;
G_Q5	select min(x.PropOne), avg(x.PropOne) _M group by x;
G_Q6	select min(x.PropOne), avg(x.PropOne) _M group by x, y;
G_Q7	select min(x.PropOne), avg(x.PropOne) _M group by x.PropOne;

Pozn: _M = match (x) -> (y) -> (z).

Tabulka 4.6: Dotazy Group by

Pro výpočet agregačních funkcí jsme zvolili funkce `min` a `avg`, protože představují netriviální práci narozdíl od funkcí `sum`/`count`. V případě `min` v paralelních řešeních dochází k mechanismu `CompareExchange` a u `avg` dojde ke dvou atomickým přičtením.

- G_Q1 testuje single group Group by. Vše je agregováno pouze do jedné skupiny.
- G_Q2 testuje vytváření skupin podle `ID` vrcholů y.
- G_Q3 přidává k G_Q2 overhead za ukládání a zpracovávání (hash + compare) další proměnné.
- G_Q4 testuje vytváření skupin náhodně vygenerovaných hodnot Int32 (viz 4.2).

- G_Q5 testuje situaci, kdy při paralelním zpracování žádné jiné vlákno během vyhledávaní nenajde stejnou hodnotu.
- G_Q7 testuje situaci, kdy dojde k rozprostření několika stejných hodnot v grafu. Dodává větší šanci, že nějaké vlákno během paralelního zpracování dostane stejnou skupinu jako vlákno jiné.

4.3 Metodika

Pro provedení experimentu jsme připravili jednoduchý benchmark, který je součástí příloh zdrojových kódů A.1. Paralelizování řešení jsme otestovali při zatížení všech dostupných jader procesoru (argument `ThreadCount = 8`). Při spuštění programu dojde k navýšení priority procesu, aby docházelo k méně častému vykonávání ostatních procesů na pozadí během testování. Pro `ThreadCount = 1` navíc dochází k navýšení priority hlavního vlákna. To není možné u paralelního testování, protože vlákna běží v nativním `ThreadPool`, který neumožňuje navyšování priority vláken.

Následuje ukázka hlavní smyčky benchmarku:

```
...
 WarmUp(...);
...
double[] times = new double[repetitions];
for (int i = 0; i < repetitions; i++)
{
    CleanGC();
    var q = Query.Create(..., false);
    timer.Restart();

    q.Compute();

    timer.Stop();
    times[i] = timer.ElapsedMilliseconds;
    ...
}
```

Hlavní smyčka benchmarku se skládá z 5-ti opakování warm up fáze následovanou 15-ti opakováními měřené části. Měřená část obaluje pouze vykonání dotazu bez konstrukce dotazu. V konstruktoru `Query.Create(..., false)` argument `false` způsobuje, že vykonávaný dotaz neprovede `select` část dotazu, která není cílem testování. Výsledná doba je tedy čas strávený částí Match (výhledávání vzoru) společně s částí Group/Order by.

Před měřením dochází vždy k úklidu haldy.

```
static void CleanGC()
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
    GC.Collect();
}
```

K měření uplynulé doby jsme použili nativní třídu C# `Stopwatch`, protože náš hardware a operační systém podporuje high-resolution performance counter. Pro interpretaci výsledků jsme zvolili medián naměřených hodnot, který je doprovázen minimem a maximem.

4.3.1 Volitelné argumenty konstruktoru dotazu

Pro měření argumenty `FixedArraySize` a `VerticesPerThread` jsme volili následovně:

	FixedArraySize	**VerticesPerThread**
Amazon0601	4194304	512
WebBerkStan	4194304	512
As-Skitter	8388608	1024

Tabulka 4.7: Výber argumentů konstruktoru dotazu pro grafy

Dané argumenty se nám nejvíce osvědčili v průběhu vývoje dotazovacího enginu. Vyhledávání vzoru na nich docilovalo nejrychlejších výsledků.

4.3.2 Hardwarová specifikace

Všechny testy proběhly na notebooku Lenovo ThinkPad E14 Gen. 2 verze 20T6000MCK s operačním systémem Windows 10 x64.

- 8 jádrový procesor AMD Ryzen 7 4700U (2GHz, TB 4.1GHz)
- 24GB RAM DDR4 s 3200 MHz

4.3.3 Příprava hardwaru

Každému testování předcházel studený reboot systému a odpojení od internetu. V průběhu testování neběžel žádný klientský proces kromě benchmarku a nativních systémových procesů. Rovněž, použitý notebook byl napájen po celou dobu testování.

4.3.4 Překlad

Benchmark společně s dotazovacím enginem a potřebnými knihovnami byl přeložen v `Release` módu Visual Studia 2019 pro platformu x64 cílící na .NET Framework 4.8.

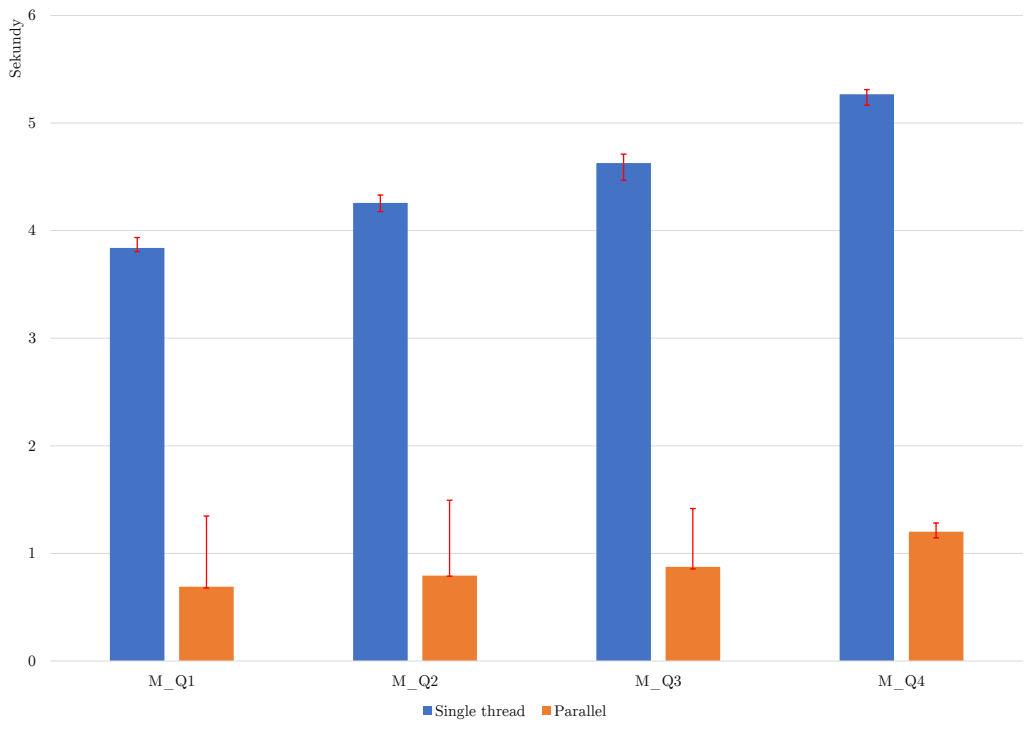
4.4 Výsledky a diskuze

V této sekci prezentujeme naměřená data pro všechny tři grafy (4.1), které jsme podrobili dotazům z sekce 4.2.

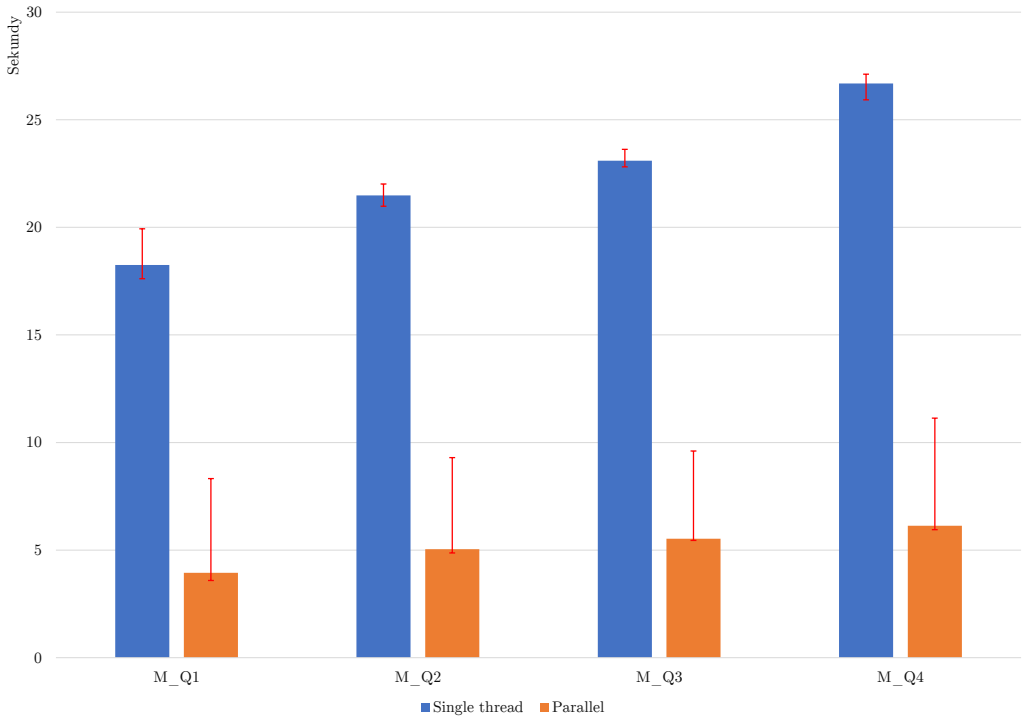
U grafů Group/Order by se držíme značení odpovídající z kapitoly implementace (tj. ve tvaru (mód enginu): (Název řešení) (způsob ukládání výsledků u Group by)?) Pro připomenutí zmíníme, že mód Normal vykonává Group/Order by až po dokončení vyhledávání, vylepšené módy Streamed a Half-Streamed je vykonávají v průběhu vyhledávání. U paralelního řešení Streamed jsou výsledky zpracovány globálně, zatímco u Half-Streamed řešení dochází k lokálnímu zpracování zakončeném mergováním.

4.4.1 Výsledky Match

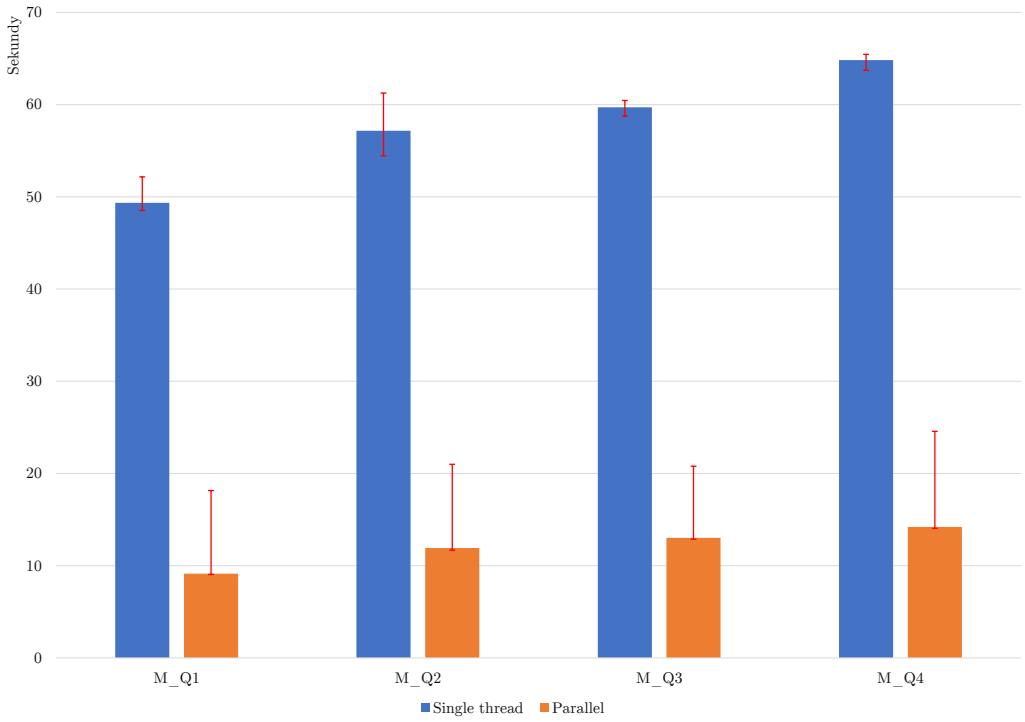
Stávající a vylepšené verze Group/Order by jsou značně ovlivněny vyhledáváním vzoru. Proto uvádíme výsledky a analýzu dotazů Match zvlášť, aby bylo možné sledovat čas výhradně strávený vyhledáváním a uložením všech nalezených výsledků do tabulky.



Obrázek 4.1: Doba vykonání dotazů Match pro graf Amazon0601 (sekce 4.1). Jedno vlákno vůči osmi vláknům. Počet nalezených výsledků je 32373599.



Obrázek 4.2: Doba vykonání dotazů Match pro graf WebBerkStan (sekce 4.1). Jedno vlákno vůči osmi vláknům. Počet nalezených výsledků je 222498869.



Obrázek 4.3: Doba vykonání dotazů Match pro graf As-Skitter (sekce 4.1). Jedno vlákno vůči osmi vláknům. Počet nalezených výsledků je 453674558.

Zbytek sekce věnujeme popisu obrázků 4.1, 4.2 a 4.3. Paralelizace startovního prohledávacího vrcholu (tj. každé vlákno dostává opakovaně množství vrcholů k prohledání určené argumentem `VerticesPerThread`, dokud se nevyčerpají všechny vrcholy grafu) dociluje zrychlení v rozmezí 4.17-krát až 5.56-krát pro všechny grafy. Výsledky pro jednotlivé dotazy dopadly podle našeho očekávání. Dotaz M_Q1 provádí pouze vyhledávání výsledků bez ukládání do tabulky a je nejrychlejší. Všechny ostatní dotazy dosahují zpomalení závislé na počtu ukládaných proměnných (počet ukládaných proměnných definuje část Select), tedy čím více proměnných k uložení tím je vykonání pomalejší a to platí i pro paralelní verzi. Pro představu, každá proměnná (element grafu) je uložena do vlastního sloupečku, který je lokální pro vlákno (`List<Element[FixedArraySize]>`). Lokalita sloupečků vede na potřebu mergování výsledků vláken.

Nicméně, díky ukládání do polí fixní délky nastává nutnost pouze zarovnat poslední nezaplněná pole, zbytek práce mergování je jen přesunutí několika pointrů na pole. Tento proces je paralelizovaný pouze přes sloupečky, tedy v dotazu M_Q2 mergování beží pouze v jednom vlákně a proto obsahuje nejvyšší skok rychlosti mezi dotazem před a dotazem po. Obecně vidíme, že ukládání výsledků nepřináší až tak velkou přítež na dobu vykonávání jako paměťovou, kdy všechny výsledky jsou uloženy v paměti. Například, všechny dotazy na grafu As-Skitter (obrázek 4.3), vygenerují 453674558 výsledků, což představuje na x64 platformě 3.629 GB pro jeden sloupeček. Zmíněné poznatky použijeme při analýze experimentů pro Group/Order by.

4.4.2 Výsledky Order by

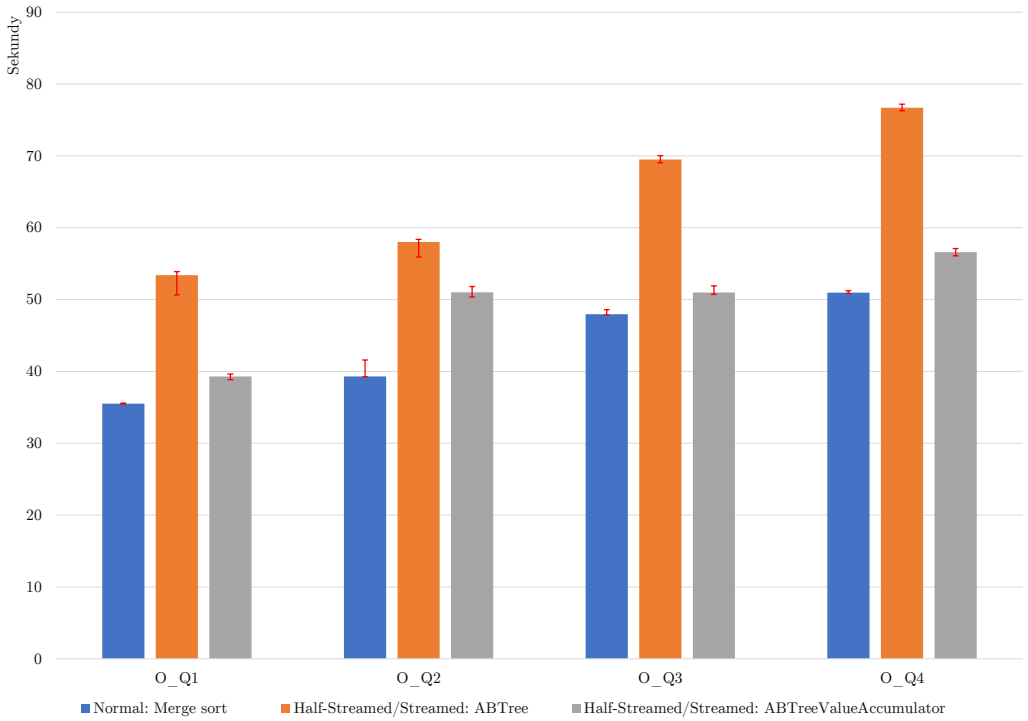
Z důvodu časové a prostorové složitosti třídění na grafu As-Skitter jsme se rozhodli jej vynechat pro Order by dotazy.

Vylepšená řešení při přichozím výsledku jej uloží do tabulky (stejné tabulky jako v řešení Normal) a následně vloží index výsledku v tabulce do indexovací struktury, tj. v našem případě (a, b)-strom[2], kde $b = 2a$. Používáme $b = 256$. V řešení ABTree se jedná o obecný (a, b)-strom, zatímco řešení ABTreeValueAccumulator výsledky (indexy) mající stejnou hodnotu klíčů třídění uloží do `List<int>`. Zástupce Normal řešení je Merge sort[3].

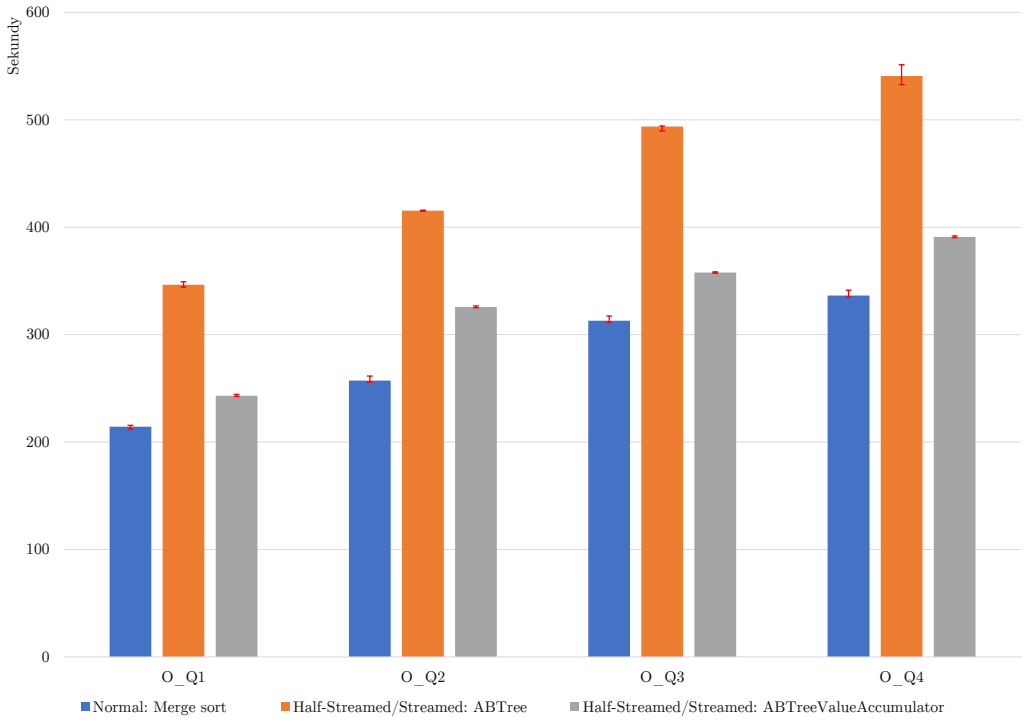
Začneme řešením běžícím v jednom vlákně, tj. obrázky 4.4 a 4.5. Můžeme si všimnout, že výsledky vypadají v rámci daných grafů konzistentně pro každý dotaz. Ani jedno z vylepšených řešení nedokázalo porazit mód Normal, což odpovídá našim předpokladům. Je to protože daný strom podléhá režii za insert $\Theta(\log n \cdot (a/\log a))$ (Mareš (2020, 03. (a,b)-trees str. 6)), kdy dochází k častému alokování nových vrcholů a překopírovávání prvků při splitu. Nejproblematičtější část je množství tříděných výsledků, kdy počet samotných hodnot klíčů třídění je omezen počtem vrcholů v grafu (tabulka 4.1). Daná situace vede k vytváření posloupnosti stejných hodnot ve vrcholech a následné nemožnosti využít jejich podstromy. Celý problém jsme vyřešelili v řešení ABTreeValueAccumulator, kdy se duplicitní hodnoty ukládají do zmíněného pole a tím omezujeme velikost výsledného stromu. Jak vidíme na obrázcích, řešení se přibližuje rychlosti řešení Normal.

[2]Mareš (2020, 03. (a,b)-trees)

[3]Duvanenko (2018)



Obrázek 4.4: Doba vykonání dotazů Order by pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu. Počet tříděných výsledků je 32373599.



Obrázek 4.5: Doba vykonání dotazů Order by pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu. Počet tříděných výsledků je 222498869.

Dle našich předpokladů se ukázalo, že třídění podle `ID` (O_Q1 a O_Q2) vůči Properties (O_Q3 a O_Q4) vede ke znatelnému overheadu. Je to způsobeno nutným přístupem k databázi, při kterém se ověřuje, jestli daná vlastnost existuje na daném elementu a následném čtení hodnoty ze struktury obsahující ji.

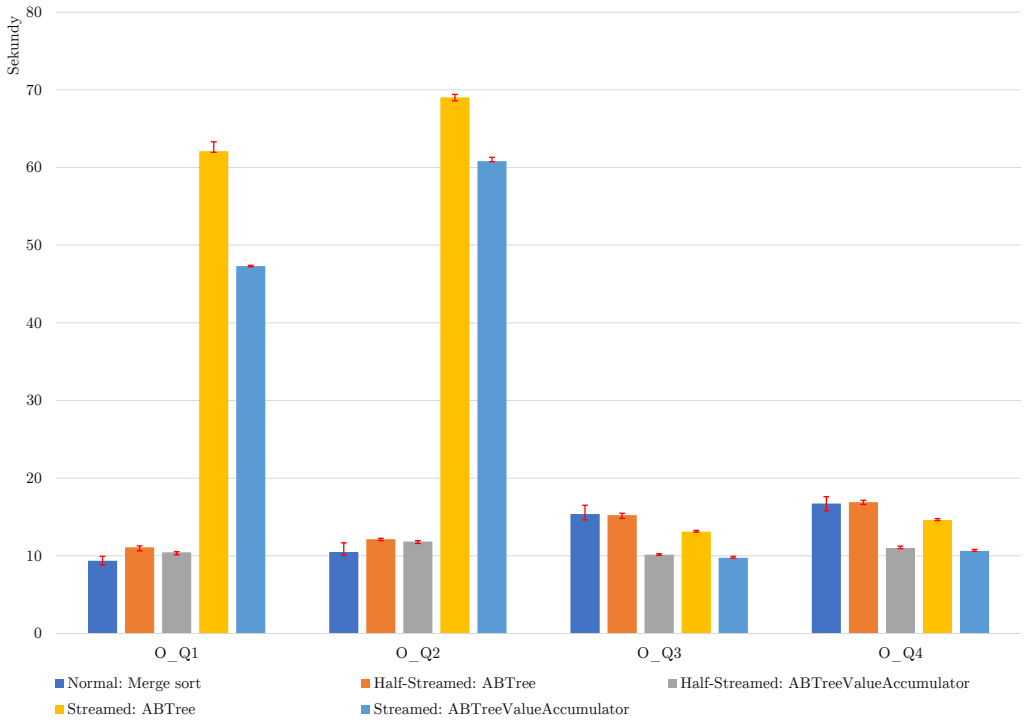
Paralelní zpracování aplikuje použité verze (a, b)-stromů ze zpracování pro jedno vlákno. Half-Streamed řešení obsahuje lokální tabulku a indexační strom pro každé běžící vlákno. Po dokončení vyhledávání se obsahy stromů překopírují do pole a dojde k paralelnímu 2-way merge používající stejnou funkci jako paralelní Merge sort. Streamed řešení rozdělí rozsah prvního třídícího klíče do přihrádek rovnoměrné velikosti. Při příchozím výsledku se získá hodnota prvního klíče třídění a určí se jeho přihrádka. Počet přihrádek je heuristicky zvolen jako $m = t^2$, kde $t = \#vláken$. Samotná přihrádka obsahuje opět tabulku a indexační strom přistupné pomocí zámku. Při porovnávání je nutné mít na paměti, že lokálně bežící části používájí cachování popsané v sekci (TODO).

Nyní budeme prezetovat výsledky paralelizace (obrázky 4.6 a 4.7). Pro dotazy O_Q1 a O_Q2 vidíme u Streamed řešení mnohonásobný rozdíl vůči ostatním řešením, protože přihrádky jsou rozděleny na základě rozsahu typu klíče (např. pro O_Q3 `[Int32.MinValue, Int32.MaxValue]`). Avšak, hodnoty třídění spadaní do rozsahu `ID` vrcholů grafu, což představuje rozsah $\approx [0, \#vrchol\text{ů}]$. Rozsah hodnot třídění spadá vždy do jedné přihrádky a výsledná doba je rovna době single thread řešení s overheadem za přistupovaný zámek. U dotazů O_Q3 a O_Q4 je tříděno pomocí hodnot vygenerovaných náhodně spadající do celého rozsahu typu klíče a zde Streamed řešení předčilo všechna ostatní. Pro budoucí rozšíření by bylo nutné zvážit vytvoření statistik rozsahů jednotlivých Properties, aby bylo možné lépe vytvořit rozdělení přihrádek.

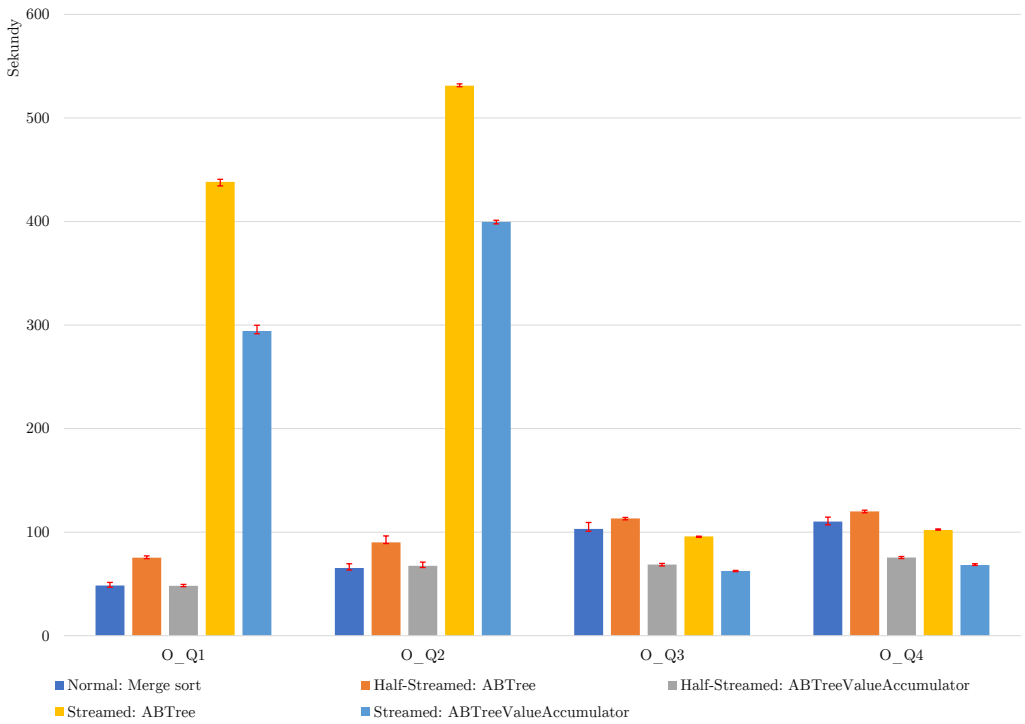
Half-Streamed řešení se přibližuje Normal řešení v prvních dvou dotazech a překonává jej ve třetím i čtvrtém dotazu pro řešení používající ABTreeValueAccumulator. U třetího a čtvrtého dotazu se porovnává pomocí Properties. V single thread zpracovávání jsme viděli overhead za dané porovnání. V druhém kroku u daného Half-Streamed řešení dochází k mergování pouze akumulovaných skupin, což rapidně sníží počet porovnávání při mergi a odtud výhoda oproti Normal Merge sort řešení. To samé platí u Streamed řešení, protože počáteční rozhashování způsobí vkládání do mnohonásobně menší skupiny výsledků. Celá situace je navíc umocněna zmíněným cachovaním porovnávaných hodnot.

Zajímavý výsledek testování je i poměr velikosti zrychlení vylepšených módů, který jsme neočekávali. Zrychlení Normal řešení zaostává vůči ostatním řešením. Implementace paralelního Merge sortu funguje na principu postupného rozdělování a při každém rozdělení se vytváří nové `Tasks` pro `ThreadPool`. U vylepšených řešení běži jedna metoda pro každé vlákno po dobu celého zpracování. Jestli se jedná o hlavní důvod poznatku by vyžadovalo dalšího testování.

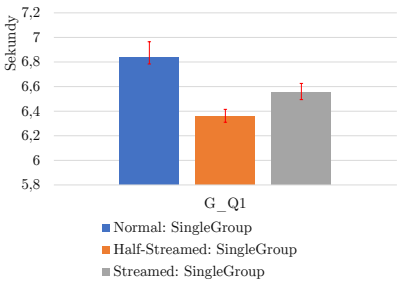
Jako důsledek testování můžeme konstatovat, že agregování v průběhu vyhledávání nepřináší předpokládané výhody. Zrychlení nastává pouze u paralelizace řešení při dostatečně náhodných datech třádění.



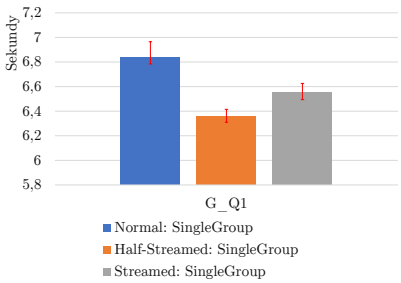
Obrázek 4.6: Doba vykonání dotazů Order by pro graf Amazon0601 (sekce 4.1). Běh osmi vláken. Počet tříděných výsledků je 222498869.



Obrázek 4.7: Doba vykonání dotazů Order by pro graf WebBerkStan (sekce 4.1). Běh osmi vláken. Počet tříděných výsledků je 222498869.



Obrázek 4.8: first figure



Obrázek 4.9: second figure

4.4.3 Výsledky Group by

asdksflrk jr arjl r t

Závěr

Tady ma byt text

Seznam použité literatury

Anděl, J. (2007). *Základy matematické statistiky*. Druhé opravené vydání. Matfyzpress, Praha. ISBN 80-7378-001-1.

Duvanenko, V. J. (2018). Hpcsharp. `https://github.com/DragonSpit/HPCsharp`.

Leskovec, J. a Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Mareš, M. (2020). Lecture notes on data structures. `http://mj.ucw.cz/vyuka/dsnotes/`. [Online; accessed 1-January-2021].

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

A. Přílohy

A.1 Zdrojové kódy

Přílohou této bakalářské práce jsou zdrojové kódy dotazovacího enginu, benchmarku a použité knihovny HPCsharp. Vše zmíněné je přiloženo v rámci jednoho projektu Visual Studia, kromě souborů Gitu. Dále, mimo projekt jsou přiloženy zdrojové kódy programů na generování vstupních grafů pro experiment. Jedná se o soubory GraphDataBuilder.cs a PropertyGenerator.cs.

A.2 Online Git repozitář

V době vydání tohoto textu probíhal vývoj dotazovacího enginu na GitHubu.

`https://github.com/goramartin/QueryEngine`

A.3 Použité grafy při experimentu

Grafy použité při experimentu jsou vloženy do odpovídajících složek dle názvu grafu. Složky obsahují originálni grafy před transformací a datové soubory po transformaci.

A.4 druha priloha

Priloha po prvni strance priloh