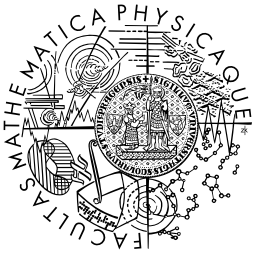


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

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. Požadavky

1.1 Property graf

1.2 PGQL

aaa Anděl (2007, Věta 4.22) aa

2. Analýza implementace

V této kapitole se pokusíme analyzovat možná řešení výstavby dotazovacího enginu a jeho úpravy, tak abychom splnili námi kladené požadavky společně se zadáním práce. Analýza bude probíhat v několika krocích. Začneme obecným návrhem dotazovacího enginu a projdeme hlavní koncepty pro implementaci. V druhém kroku zvážíme kroky vykonávání dotazů a postup výběru řešení částí Order by a Group by, které se budou vykonávat po dokončení vyhledávání dotazu. V třetím kroku provedeme analýzu úprav pro agregaci v průběhu vyhledávání. Součástí této části bude analýza algoritmů Order by a Group by pro dané úpravy.

2.1 Obecný pohled na engine

V naší představě je dotazovací engine určen pro práci nad grafem, který je celý obsažen v paměti, včetně vlastností elementů grafu. Graf bude načten v definovém formátu a následně na něm budou vykonávány dotazy. Při obecném pohledu na engine jsme lokalizovali hlavní bloky výstavby. Jsou to tři bloky: reprezentace grafu, parser a reprezentace dotazu.

Reprezentace grafu určuje pohled na načtený graf v paměti a určuje formát objektů, nad kterými bude vykonáván dotaz. Parser načítá uživatelký dotaz do interní reprezentace. Na základě reprezentace se vytváří struktury dotazu a definuje se exekuční plán. Z obecných úkonů částí vidíme, že se nejedná o stand-alone části. Vytváří se nám závislosti, které budeme muset uvážit.

2.2 Reprezentace grafu

Musíme uvážit, jak reprezentovat graf. Z části 1 je hlavní faktor podmnožina jazyku PGQL pro Property grafy.

2.2.1 Elementy grafu a jejich typ

Musíme zvažovat reprezentaci elementů grafu a jejich typu. V našem případě jsou elementy pouze vrcholy a orientované hrany. Typ definuje seznam Properties na elementu. Properties jsou také typované. Každá hodnota Property musí být přístupná skrze daný element:

- Pokud držíme element grafu, musíme být schopni jej rozlišit od ostatních elementů.
- Pokud držíme element grafu, musíme být schopni přistoupit k jeho vlastnostem.

V naší představě je řešení následovné. Každý element grafu bude potomkem jednoho abstraktního předka a potomci si budou definovat svá specifika. Potomek bude vrchol a hrana. Předek si bude pamatovat unikátní IDs v celém grafu, abychom elementy dokázali rozlišit. Předek navíc bude znát svůj typ. Ideálně ukazatel na tabulku. Tabulka by reprezentovala typ a byla by společná všem elementům majících stejný typ. V tabulce by byl obsažen pouze seznam IDs, jejich pořadí a Properties v podobě polí s hodnotami. Properties by byly přístupné pomocí unikátního identifikátoru pro celý graf. Hodnoty Properties každého elementu by ležely na pozicích dle pořadí IDs. Nyní, pokud držíme element grafu, můžeme přistoupit k hodnotě Property skrze tabulku pomocí jeho ID.

2.2.2 Struktury obsahující elementy

Nyní musíme analyzovat, jaké struktury by byly idální pro uchovávání elementů grafu. Musíme brát v potaz, že propojení mezi vrcholy pomocí hran přímo ovlivňuje vyhledávání v části Match. V průběhu vyhledávání v určitý moment vždy vlastníme nějaký element grafu. Na základě daného elementu musíme provést akci:

- Pokud držíme vrchol, musíme být schopni přistoupit k jeho hranám. Hranám z/do něj. Daný přístup by měl být co nejrychlejší a neměl by obsahovat žádné iterace. V průběhu hledání se z vrcholu musí projít skrze všechny jeho hrany. Ideálně by měly být hrany přístupné skrze index.
- Pokud držíme hranu, musíme být schopni přistoupit ke koncovému vrcholu. V průběhu hledání vždy vlastníme vrchol než přistoupíme k jeho hraně a následně k jejímu koncovému vrcholu. Tímto můžeme vyloučit nutnost, aby hrana znala informaci o svém původu.
- Pokud držíme element, chceme být schopni přistoupit k jeho sousedním elementů v obsajující struktuře. Například budeme chtít prohledávát graf jen z určeného množství vrcholů bez vytváření pomocných struktur.

K vyřešení daných problému v naší představě bychom použili tři pole. Pole vrcholů, pole out hran a in hran. Zde by bylo vhodné rozšířit potomky hran na dva nové typy. Hrany by si pamatovali svůj koncový vrchol. Pro in hranu by to byl vrchol odkud vychází, aby bylo možné v moment držení vrcholu projít skrze ni na vrchol další. Každé pole tedy bude mít unikátní typ, který nám pomůže rozlišit k jaké situaci má dojít v průběhu prohledávání. Abstraktní předek všech elementů by si měl nově pamatovat i svou pozici v daných polích pro rychlý přístup k jeho sousedům. Zbývá vyřešit vztah hran a vrcholů. Řešení, které bychom chtěli zvolit, je mít hrany v polích seskupeny podle: vrcholů odkud vycházejí (pole out hran), vrcholů kam směřují (pole in hran). Vrchol by si pak pamatoval rozsah svých hran v příslušných polích. Chceme-li procházet hrany vycházejích z vrcholu, stačí držet pole out hran a rozsah náležící vrcholu. Tedy dva indexy. Skrze indexy můžeme pak libovolně iterovat.

Uvažovali jsme nad různými alternativami. Mít jeden typ hrany obsahující všechny nutné informace. Řešení je paměťově přijatelnější, ale nastává problém s přístupem k in hranám vrcholu. Řešením by mohlo být vytvořit pole in/out

hran pro každý vrchol. Daný přístup nám případá výrazně náročnější z hlediska paměti, protože musíme vytvářet pole pro každý vrchol zvlášť.

2.2.3 Vstupní formát dat

Vstupní soubory musí obsahovat informace nutné pro Property graf. Soubory schémat typů elementů a jejich Properties. Datové soubory pak obsahující konkrétní data elementů.

Protože každý element má svůj typ, budeme mít na vstupu dva soubory schémat pro hrany a vrcholy. Schéma bude obsahovat informace o všech typech vyskytujících se v grafu. Pro typ je důležitý název a výčet Properties. Properties pak musí nést svůj název a typ. Vidíme, že se jedná jen o výčet (key, value) dvojic. Například (PropOne, integer). V tomto případě zvolíme formát JSON. Záznamy pak budou obsaženy v JSON poli:

```
Soubor schéma vrcholů:
[    { "Kind": "BasicNode" },
{ "Kind": "BasicNodeTwo", "PropOne": "integer" } ]

Soubor schéma hran:
[    { "Kind": "BasicEdge" },
{ "Kind": "BasicEdgeTwo", "PropOne": "integer" } ]
```

Jeden typ je reprezentován jednou třídou JSON. Kind zde představuje jméno typu. Properties obsahují název v položce key a typ Property je uložen v položce value.

Samotná data budou obsažena opět ve dvou separatních souborech. Každý řádek by reprezentoval jednu hranu/vrchol. V první řadě řádek musí obsahovat unikatní ID elementu a jeho typ. Za typem by následoval seznam hodnot Properties v pořadí určených schématem. Pro hrany by existoval na řádku navíc záznam ID vrcholů, které spojuje. Oddělovače mezi daty jsou implementační detail. Pro výše zmíněné schéma by datové soubory mohly vypadat následovně:

```
Soubor hran:
ID TYPE FROMID TOID PROPERTIES
50 BasicEdge 0 0
51 BasicEdgeTwo 0 1 44
...
Soubor vrcholů:
ID TYPE PROPERTIES
0 BasicNode
1 BasicNodeTwo 42
...
```

2.2.4 Načítání vstupní dat

asdasd as

2.3 Parser

2.4 Expressions a Aggregates

2.5 Reprezentace dotazu

2.6 Group by

2.7 Order by

2.8 Návrh vylepšení

3. Implementace

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. Kapitolu zakončíme prezentací výsledků.

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 pouze snižuje počet nalezených výsledků, což 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 čtyř 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 $[33; 126]$
PropFour	integer	`Int32` s rozsahem $[0; 1000]$

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",
"PropFour": "integer"
}
]
```

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),
    new Int32Generator(0, 1_000, false)
};
```

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(x.PropOne), avg(x.PropOne) M group by x;
G_Q4	select min(y.PropOne), avg(y.PropOne) M group by y, x;
G_Q5	select min(x.PropOne), avg(x.PropOne) M group by x, y;
G_Q6	select min(x.PropOne), avg(x.PropOne) M group by x.PropTwo;
G_Q7	select min(x.PropOne), avg(x.PropOne) M group by x.PropOne;
G_Q8	select min(x.PropOne), avg(x.PropOne) M group by x.PropFour;

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` (jedno přičtení proměnné). Funkce `min` porovná a prohodí výsledek. Thread-safe verze používá mechanismus `CompareExchange`. Funkce `avg` provádí dva přičtení proměnné. Thread-safe verze používá atomická přičtení. Otestujeme i samotné seskupování na dotazech G_Q2 až G_Q8. V dotazech nahradíme Select část prvním klíčem Group by. Dané dotazy značíme symbolem $'$ (např. G_Q2$'$).

- G_Q1 testuje single group Group by. Vše je agregováno pouze do jedné skupiny.
- G_Q2 a G_Q3 testuje vytváření skupin podle `ID` vrcholů. Rozdíl mezi nimi je ten, že proměnná x je při paralelním zpracování přístupná pouze jednomu vláknu za celý běh vyhledávání. Maximální počet skupin je ze shora omezen počtem vrcholů v grafu.

- G_Q4 a G_Q5 přidávájí overhead za ukládání a zpracovávání (hash + compare) další proměnné. Počet skupin je ze shora omezen počtem hran v grafu. Tyto dotazy obsahují nejvíce skupin mezi zbylými dotazy.
- G_Q6 testuje vytváření skupin náhodně vygenerovaných hodnot z celého rozsahu `Int32` (viz 4.2). Počet skupin je ze shora omezen počtem vrcholů v grafu.
- G_Q7 testuje vytváření skupin náhodně vygenerovaných hodnot z rozsahu $[0; 100000]$ `Int32` (viz 4.2). Dojde k rozprostření několika stejných hodnot v grafu. Maximální počet skupin je 100000.
- G_Q8 testuje vytváření skupin náhodně vygenerovaných hodnot z rozsahu $[0; 1000]$ `Int32` (viz 4.2). Dojde k rozprostření mnoha stejných hodnot v grafu. Maximální počet skupin je 1000.

Dotazy G_Q4/G_Q5, G_Q6, G_Q7 a G_Q8 nám umožní sledovat chování řešení při snižování počtu vytvářených skupin. Pro G_Q4 a G_Q6 bude vidět overhead za porovnání Property vůči `ID`.

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

FixedArraySize udává fixní velikost polí v tabulce použité pro ukládání výsledků (tj. sloupeček je pole polí). VerticesPerThread určuje počet přidělovaných vrcholů k prohledání v průběhu paralelní Match části. 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.

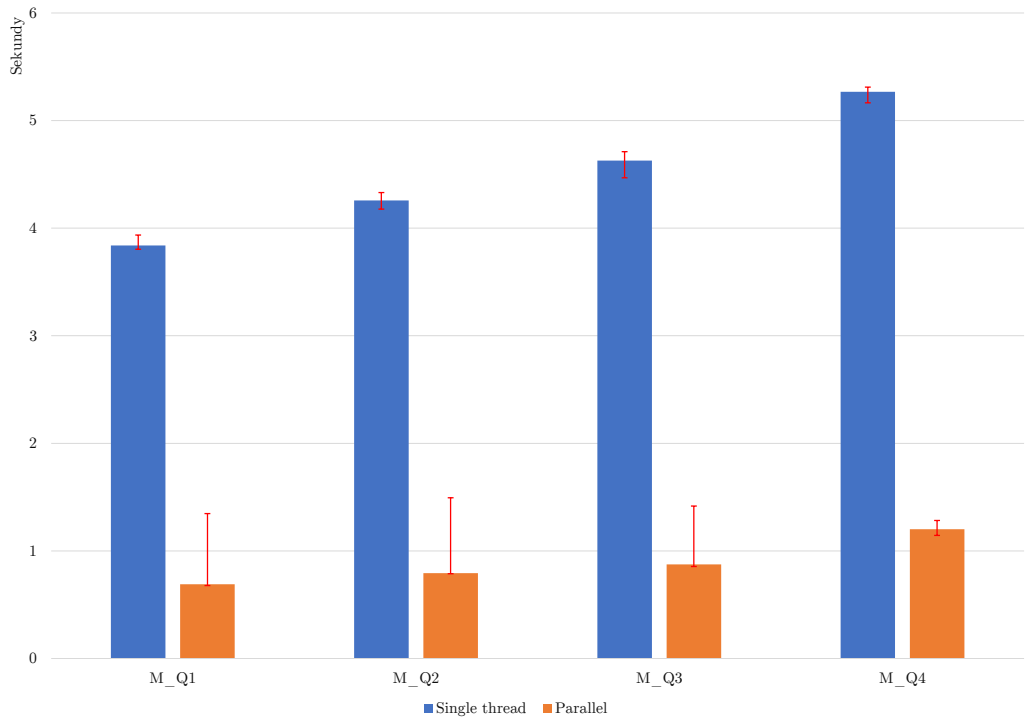
4.4 Výsledky

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

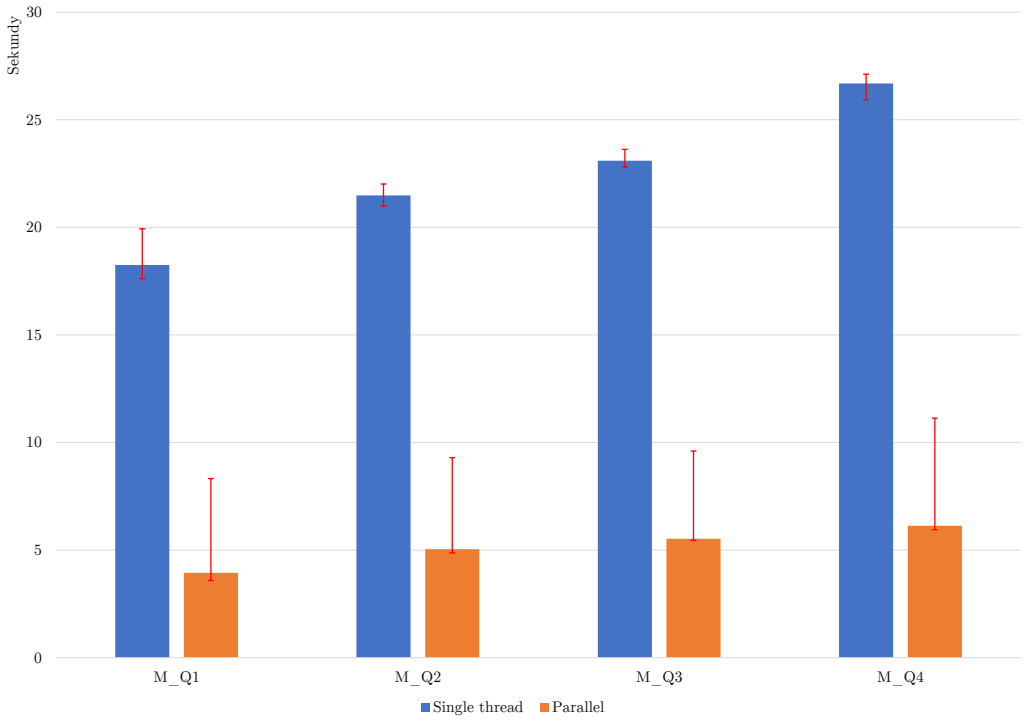
U grafů Group/Order by se držíme značení odpovídající z kapitoly implementace (tj. ve tvaru (mód enginu): (Název řešení) (způsob ukládání výsledků u Group by)?). Pokud řešení obsahuje kombinaci módů, pak řešení pro dané módy jsou totožná. Pro připomenutí zmíníme, že mód Normal vykonává Group/Order by 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 Match

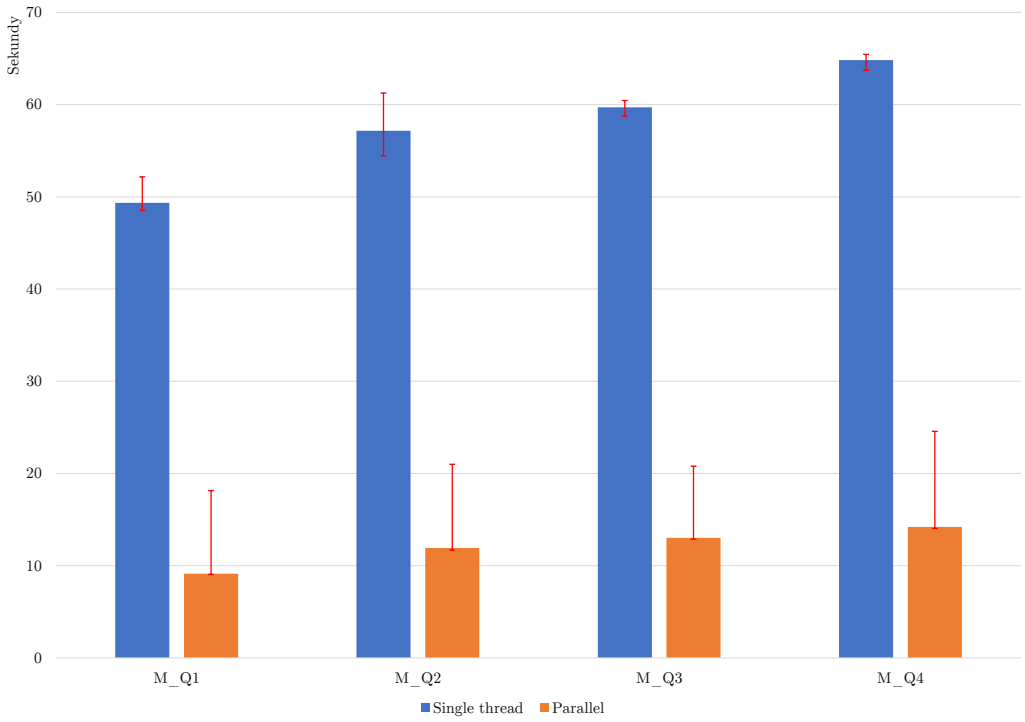
Stávající a vylepšené verze Group/Order by jsou značně ovlivněny vyhledáváním vzoru. Proto uvádíme výsledky a analýzu dotazů Match zvlášť, aby bylo možné sledovat čas výhradně strávený vyhledáváním a uložením všech nalezených výsledků do tabulky.



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; 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 Order by

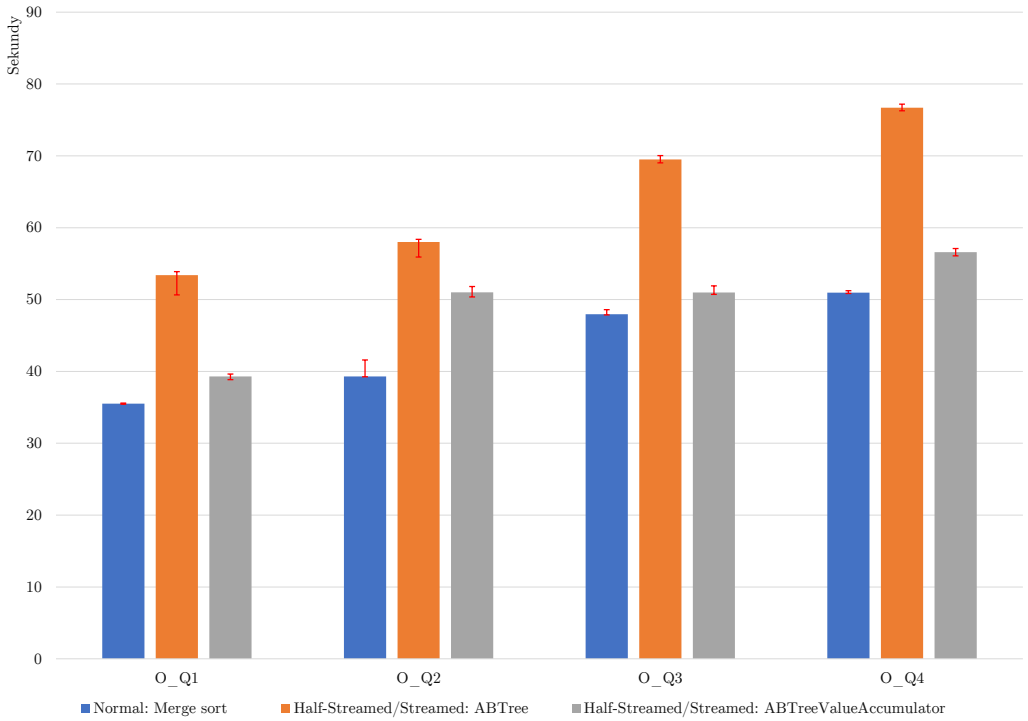
Z důvodu časové a prostorové složitosti třídění na grafu As-Skitter jsme se rozhodli jej vynechat pro Order by dotazy.

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].

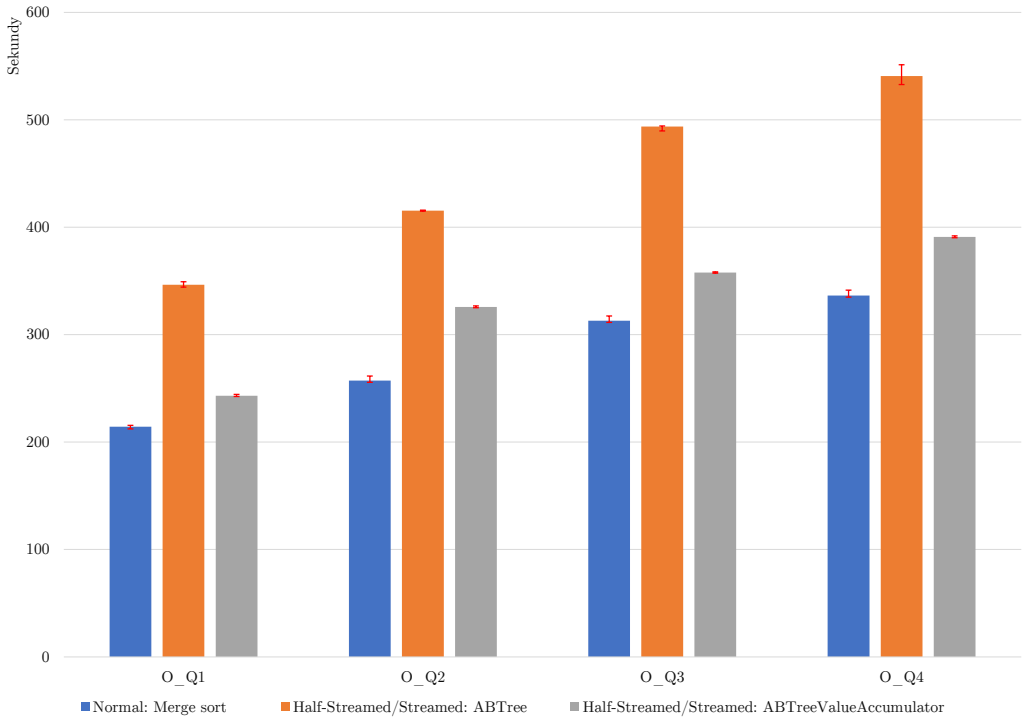
Začneme řešením běžícím v jednom vlákně, tj. obrázky 4.4 a 4.5. Můžeme si všimnout, že výsledky vypadají v rámci daných grafů konzistentně pro každý dotaz. Ani jedno z vylepšených řešení nedokázalo porazit mód Normal, což odpovídá našim předpokladům. Je to protože daný strom podléhá režii za `Insert` $\Theta(\log n \cdot (a/\log a))$ (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 opakovanému zatřizování výsledků se stejnou hodnotou a tím navyšování velikosti stromu společně s počtem porovnání na `Insert`. 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.

Problém by nastal, pokud by množství hodnot odpovídalo počtu nalezených výsledků. V tomto případě bychom vytvářeli zbytečný overhead za režii pole. 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.

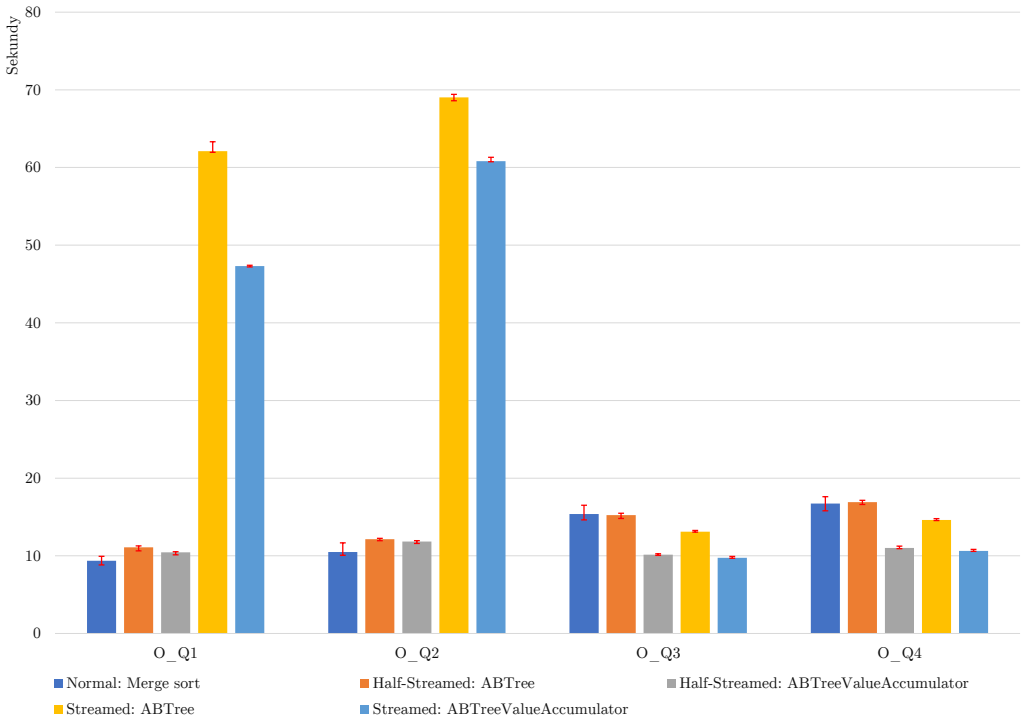
Paralelní zpracování aplikuje použité verze (a, b)-stromů ze zpracování pro jedno vlákno. Half-Streamed řešení obsahuje lokální tabulku a indexační strom pro každé běžící vlákno. Po dokončení vyhledávání se obsahy stromů překopírují do pole a dojde k paralelnímu 2-way merge používající stejnou funkci 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 spadají do rozsahu `ID` vrcholů grafu, což představuje rozsah $\approx [0; \#vrcholů]$. Hodnota třídění spadá vždy do jedné přihrádky a výsledná doba je rovna době single thread řešení s overheadem za přistupovaný zámek. Pro O_Q1 je to zpomalení o [20,51; 21]% a pro O_Q2 [17,64; 18,9]%. 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.

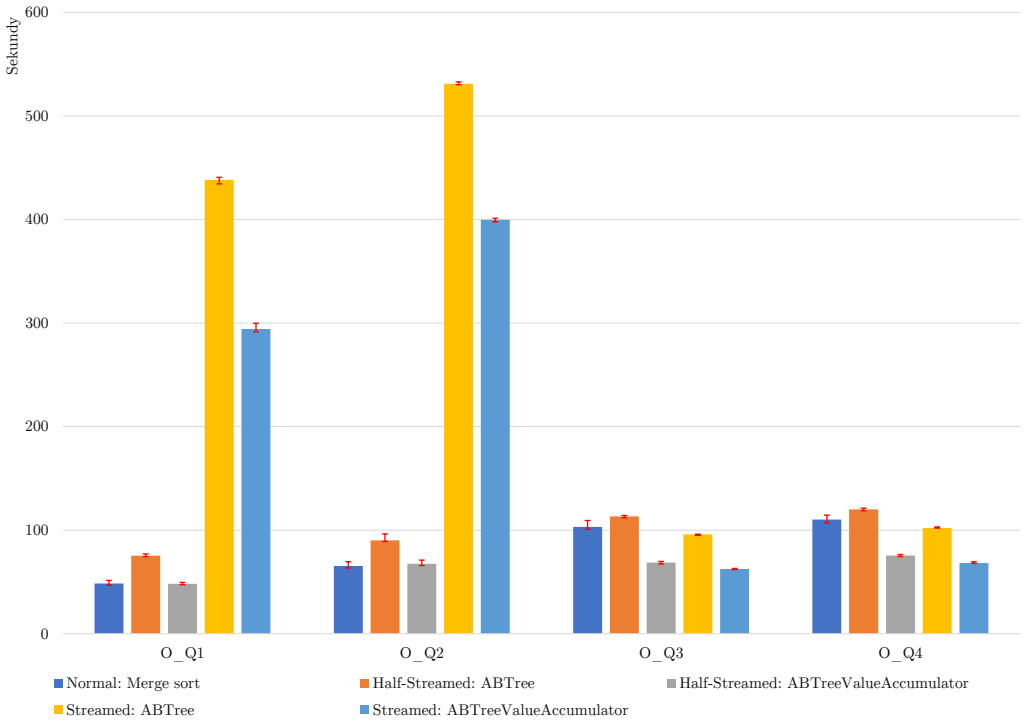
Half-Streamed řešení se přibližuje Normal řešení v prvních dvou dotazech a překonává jej ve třetím i čtvrtém dotazu pro řešení používající ABTreeValueAccumulator. U třetího a čtvrtého dotazu se porovnává pomocí Properties. V single thread zpracovávání jsme viděli overhead za dané porovnání. V druhém 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 rozhašování způsobí vkládání do mnohonásobně menší skupiny výsledků. Celá situace je navíc umocněna zmíněným cachovaním porovnávaných hodnot.

Zajímavý výsledek testování je rozsah zrychlení vylepšených módů (tabulka 4.8), který jsme neočekávali. Zrychlení Merge sortu zaostává. Maximální zrychlení u ostatních řešení je až pětinásobné. Implementace paralelního Merge sortu funguje na principu postupného rekurzivního rozdělování, při kterém 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 třídění v průběhu vyhledávání nepřináší předpokládané výhody. Zrychlení nastává pouze u paralelizace řešení při dostatečně náhodném rozložení dat třídění, pokud samotná porovnání jsou drahá.



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



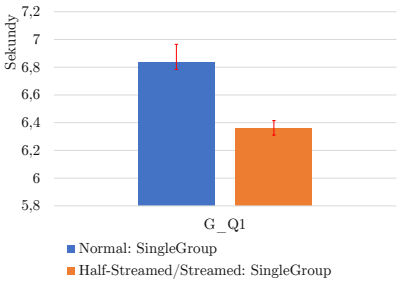
Obrázek 4.7: Doba vykonání dotazů Order by pro graf WebBerkStan (sekce 4.1). Běh osmi vláken. Počet tříděných výsledků je 222498869.

	Zrychlení
Merge sort	[3,33; 4,42]-krát
Half-Streamed: ABTree	[4,36; 4,81]-krát
Half-Streamed: ABTreeValueAccumulator	[3,76; 5,18]-krát
Streamed: ABTree	[0,78; 5,3]-krát
Streamed: ABTreeValueAccumulator	[0,81; 5,72]-krát

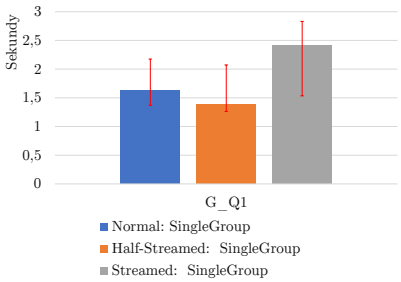
Tabulka 4.8: Rozsah zrychlení paralelizovaných řešení pomocí osmi vláken v ramci grafů pro dotazy Order by.

4.4.3 Group by

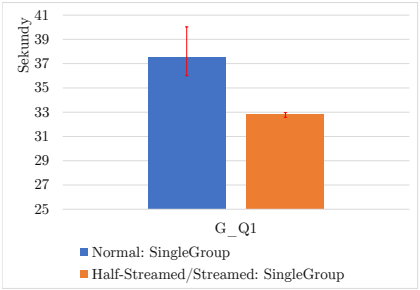
Analýzu výsledků dotazu G_Q1 uvádíme samostatně, protože testuje pouze agregační funkce a nikoliv seskupování. Po levé straně je běh v jednom vlákně a na pravé straně běh osmi vláken.



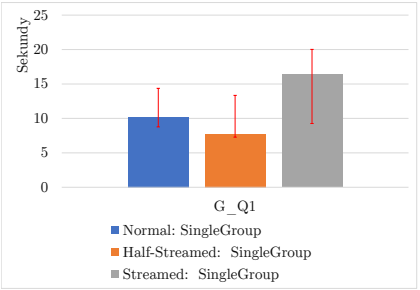
Obrázek 4.8: Doba vykonání dotazu G_Q1 pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 32373599.



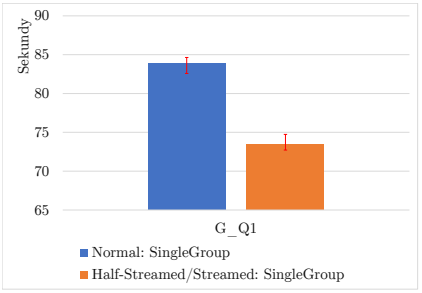
Obrázek 4.9: Doba vykonání dotazu G_Q1 pro graf Amazon0601 (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 32373599.



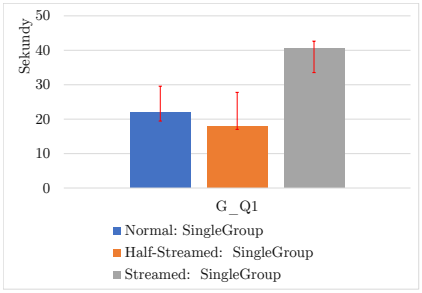
Obrázek 4.10: Doba vykonání dotazu G_Q1 pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 222498869.



Obrázek 4.11: Doba vykonání dotazu G_Q1 pro graf WebBerkStan (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 222498869.



Obrázek 4.12: Doba vykonání dotazu G_Q1 pro graf As-Skitter (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 453674558.

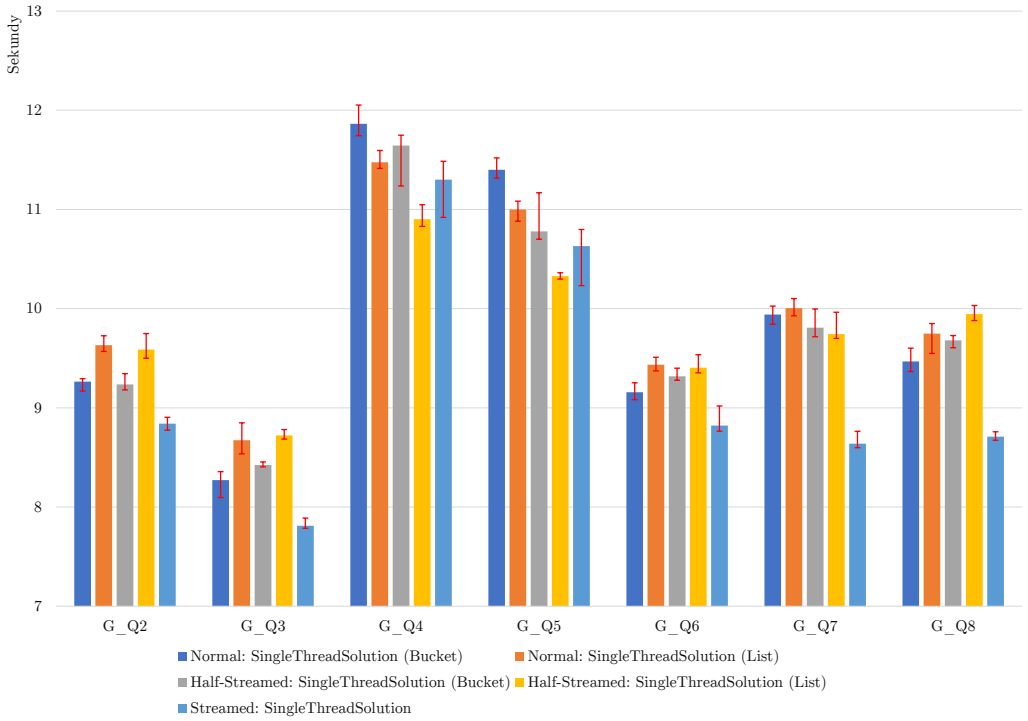


Obrázek 4.13: Doba vykonání dotazu G_Q1 pro graf As-Skitter (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 453674558.

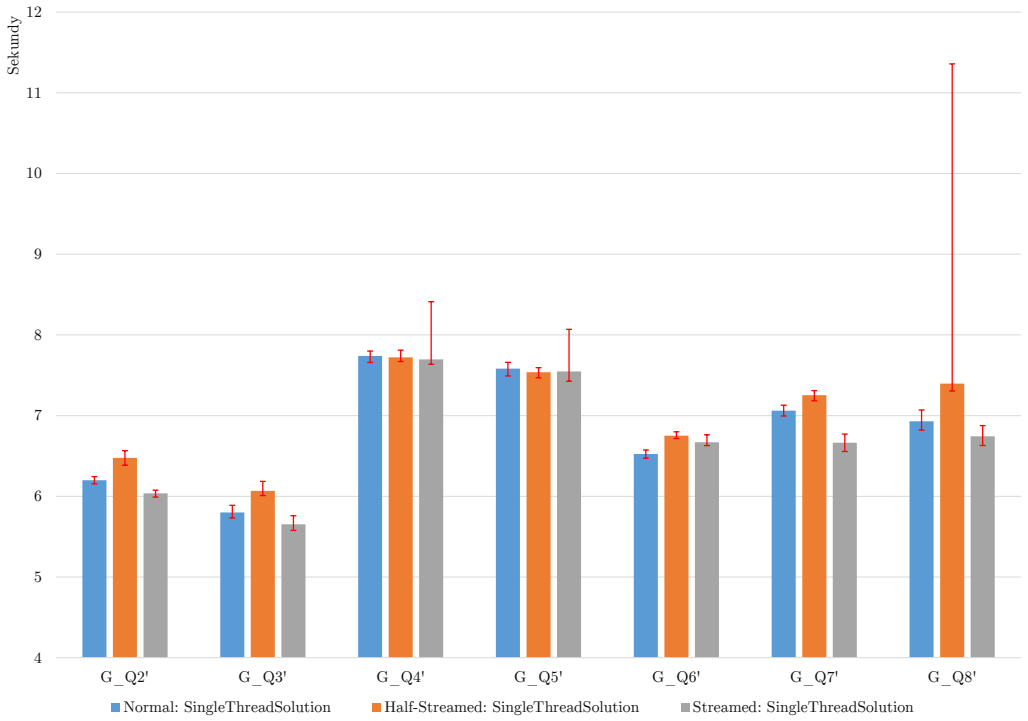
Na obrázcích 4.8 až 4.13 lze vidět značnou konzistenci mezi výsledky testování při nárustu počtu výsledků vyhledávání. Half-Streamed a Streamed řešení zde neukládá výsledky vyhledávání do tabulky, ale pouze na aktuální výsledek aplikuje agregační funkce a následně jej zahodí. To způsobuje značnou výhodu oproti Normal řešení, které drží všechny výsledky v paměti. Použijeme-li poznatky z sekce 4.4.1 o zpomalení způsobeném ukládáním výsledků do tabulky zjistíme (v našem případě jedné proměnné), že rozdíl mezi Normal a Half-Streamed řešením se pohybuje právě v rozsahu onoho zpomalení. To platí pro běh jednoho vlákna i běhu osmi vláken. Problem představuje paralelní Streamed řešení, jelikož k jednomu výsledku přistupuje osm vláken najednou, což způsobuje značné zpomalení kvůli nutné synchronizaci při výpočtu funkcí `min` a `avg`. Zrychlení je zde pouze v rozsahu $[1{,}81; 2{,}64]$-krát, zatímco u zbylých řešení je $[3{,}67; 4{,}61]$-krát.

Než postoupíme dál připomeneme hlavní rozdíly řešení a značení u zobrazených grafů. Každé řešení používá k Group by mapu (`Dictionary<key, value>`). Normal řešení ukládá všechny výsledky vyhledávání vzoru do tabulky a po dokončení vykoná Group by. Half-Streamed řešení vykonává Group by v průběhu hledání a ukládá do tabulky pouze výsledky, pro které ještě neexistuje skupina v použité mapě. Pro zmíněná řešení se jako `key` používá index do tabulky a skrze něj se následně výpočtou hodnoty klíče. Streamed řešení nepoužívá tabulku, ale hodnoty klíče ukládá rovnou do mapy. Objevující se značení Bucket a List určuje způsob ukládání výsledků agregačních funkcí (`min`, `avg`...) jako `value` záznam v mapě:

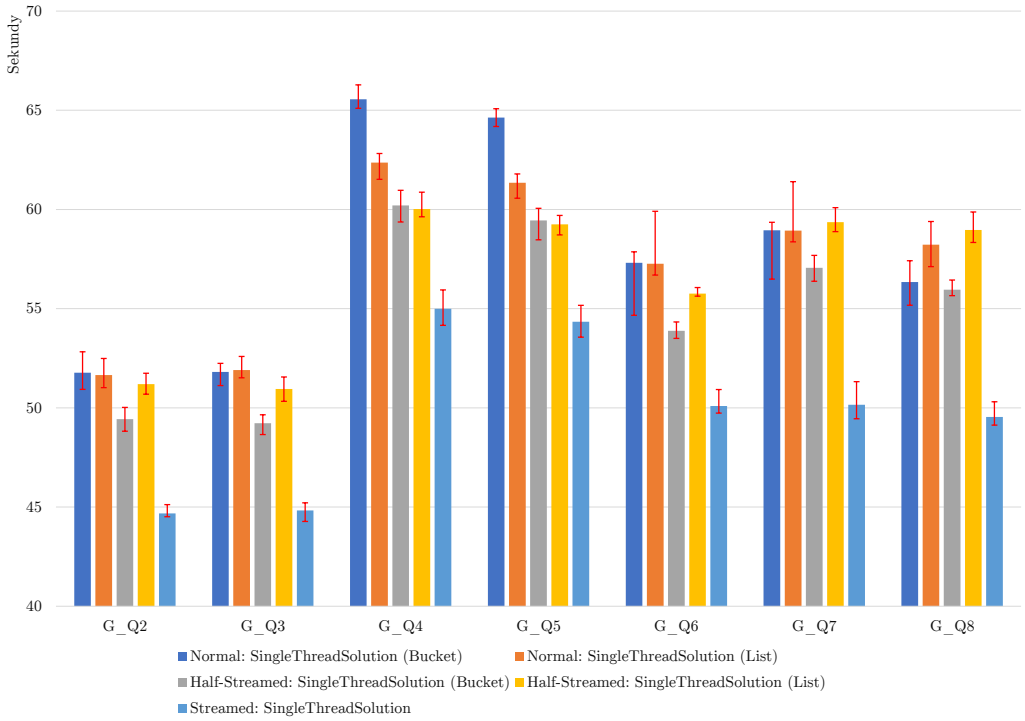
```
Bucket:
Dictionary<key, BucketResult[]> map; // Used map.
class BucketResult {}
class BucketResult<T>: BucketResult { T value; }

List:
Dictionary<key, tableIndex> map; // Used map.
ListResults aggResults; // Agg. func. values of a group
                        // are accessed via tableIndex.
class ListResults { ListHolder[] holders; }
class ListHolder {}
class ListHolder<T> : ListHolder { List<T> values }
```



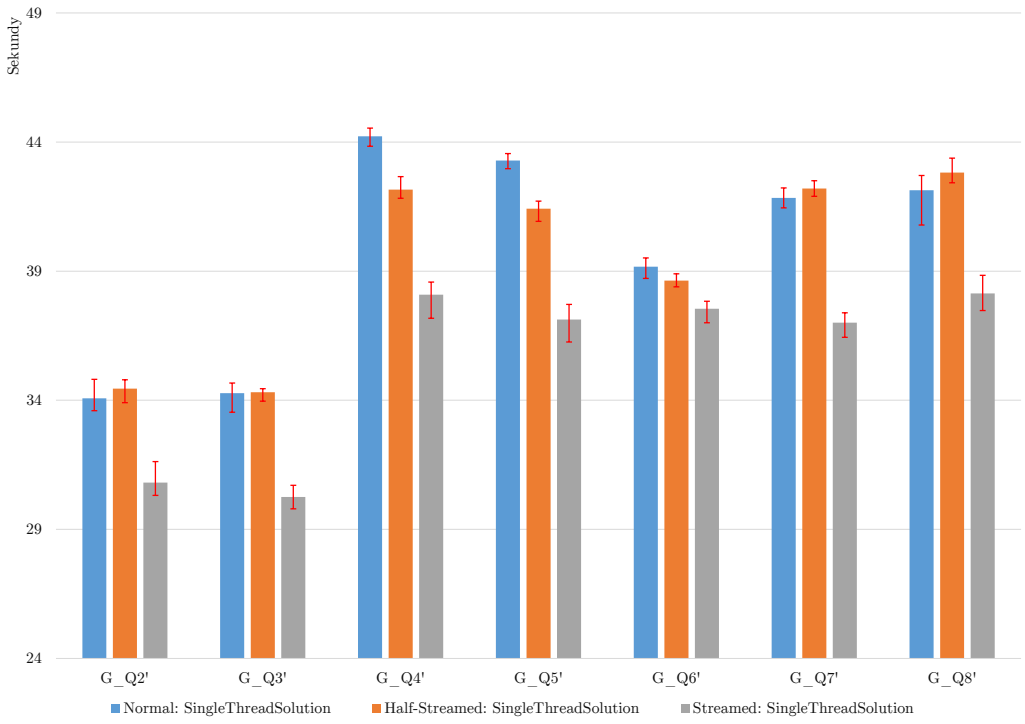
Obrázek 4.14: Doba vykonání dotazů Group by pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 32373599.



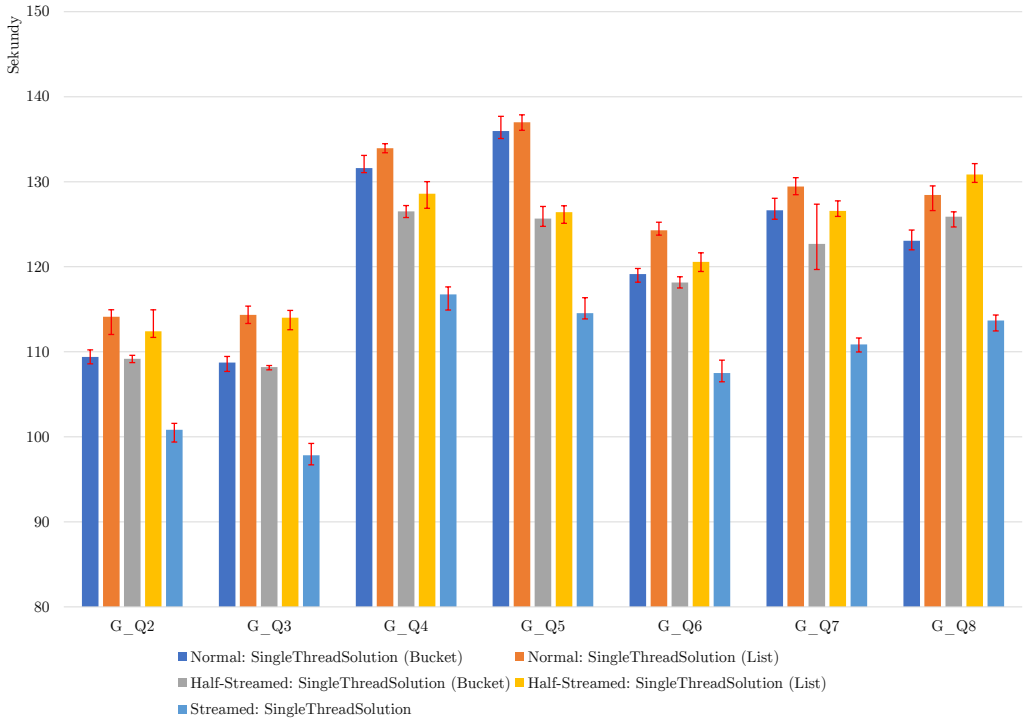
Obrázek 4.15: Doba vykonání dotazů Group by bez agr. funkcí pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 32373599.



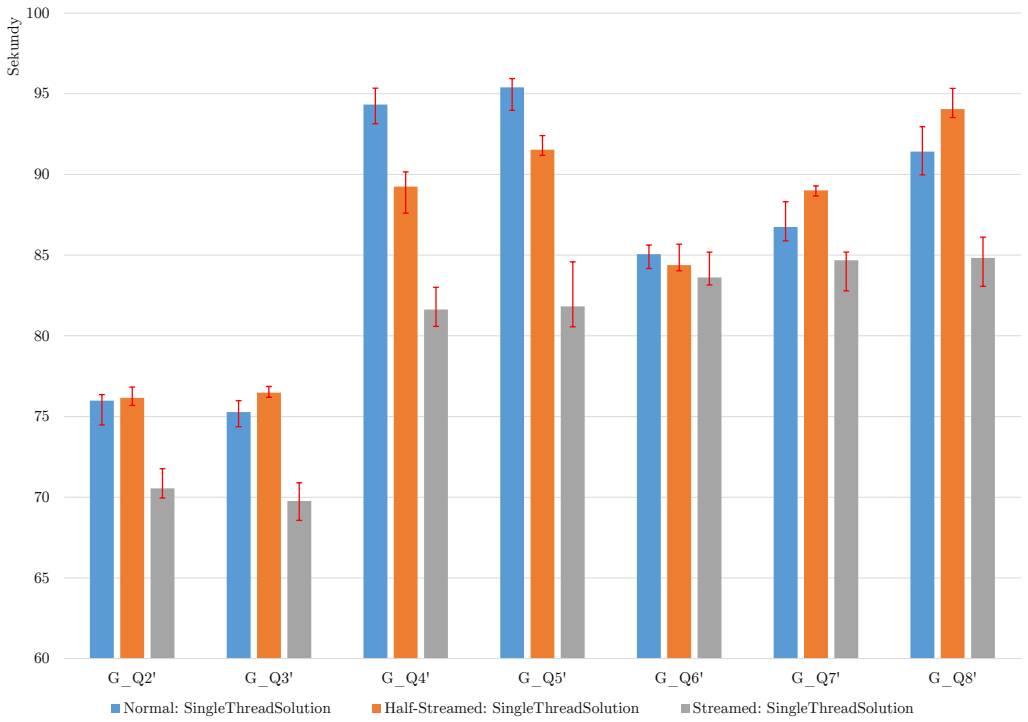
Obrázek 4.16: Doba vykonání dotazů Group by pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 222498869.



Obrázek 4.17: Doba vykonání dotazů Group by bez agr. funkcí pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 222498869.



Obrázek 4.18: Doba vykonání dotazů Group by pro graf As-Skitter (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 453674558.



Obrázek 4.19: Doba vykonání dotazů Group by bez agr. funkcí pro graf As-Skitter (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 453674558.

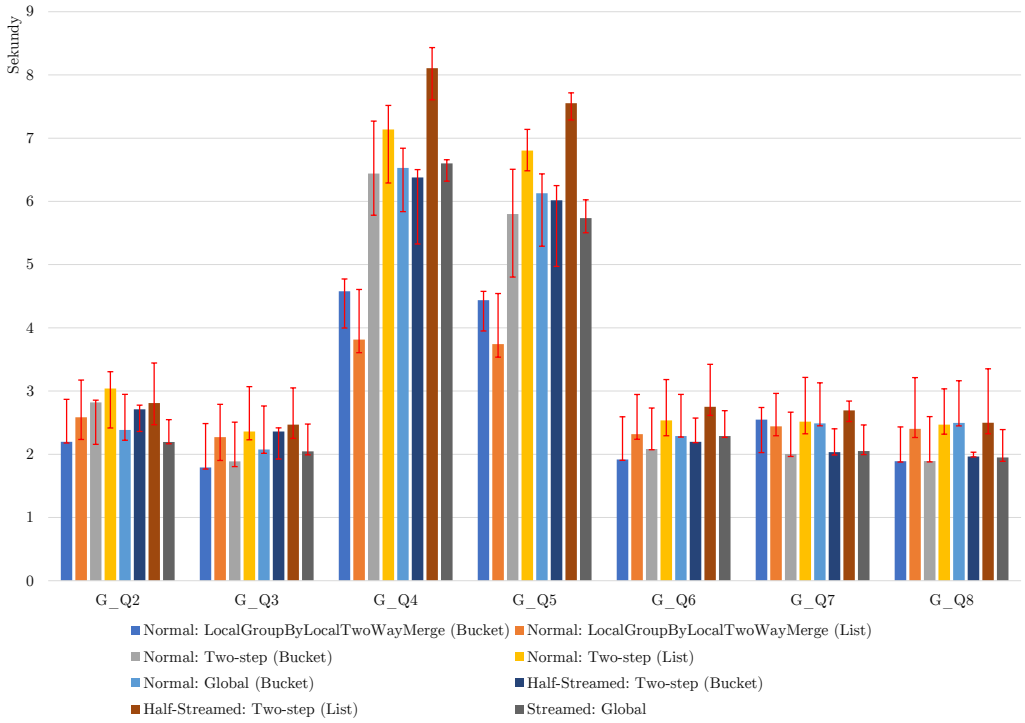
Na obrázcích 4.14, 4.16 a 4.18 vidíme výsledky Group by pro běh v jednom vlákně. Výsledky na obrázcích 4.15, 4.17 a 4.19 představují dotazy bez agregačních funkcí v části Select. Dvojice dotazů G_Q4/G_Q5, G_Q2/G_Q3 a G_Q6/G_Q7/G_Q8 jsou pouze mírně rozličná a můžeme u nich vidět konzistenci výsledků pro použité grafy. Řešení vykonávající Group by v průběhu vyhledávání překonávají Normal řešení. S růstem počtu výsledku se rozdíly mezi módy prohlubují. Například, zrychlení Streamed řešení je znatelnější u grafu As-Skitter než u grafu Amazon0601.

Obecně nejznačnější zrychlení nastává u Streamed řešení, kdy není použita tabulka výsledků. Velice mírné zrychlení můžeme vidět u Half-Streamed řešení, které ukládá jen reprezentanty skupiny. U všech dotazů bez agregačních funkcí, kromě G_Q4′/G_Q5′, nastávají situace, kdy Half-Streamed řešení je pomalejší než Normal. U grafu Amazon0601 nastává stejná situace pro G_Q3, G_Q6 a pro každý graf G_Q8. Situaci jsme neočekávali. Vysvětlujeme si ji následovně. Half-Streamed řešení používá při zpracování výsledku položku tabulky `temporaryRow`, do které přesouvá pointr na pole výsledků. Skrze danou položku pak následně přistupuje k výsledku při vkládání do mapy. Na dané přesouvání se můžeme dívat jako na kopírovaní jedné proměnné do tabulky. Při úspěšném vložení nastane navíc překopírování výsledků do pravé tabulky. Což odpovídá větší režii na zpracování výsledku než u Normal řešení. Proto vídíme pokles rychlosti a celkově jen mírné zrychlení u Half-Streamed řešení v jiných případech. Největší skok pak právě nastává v dotazech G_Q4/G_Q4′ a G_Q5/G_Q5′, kdy se ukládají dvě proměnné. Tedy samotná řežie Normal řešení je značně pomalejší, protože musí ukládat vždy dvě proměnné, zatímco Half-Streamed jen přesouvá pointr. Zajímavé je, že dané situace nastávají u dotazů bez agregačních funkcí, přestože všechna řešení pro jejich reprezentaci používají stejné funkce a struktury (List/Bucket). Předpokládali bychom tedy stejnou situaci i na ostatních (větších) grafech. Absenci jevu neumíme plně objasnit.

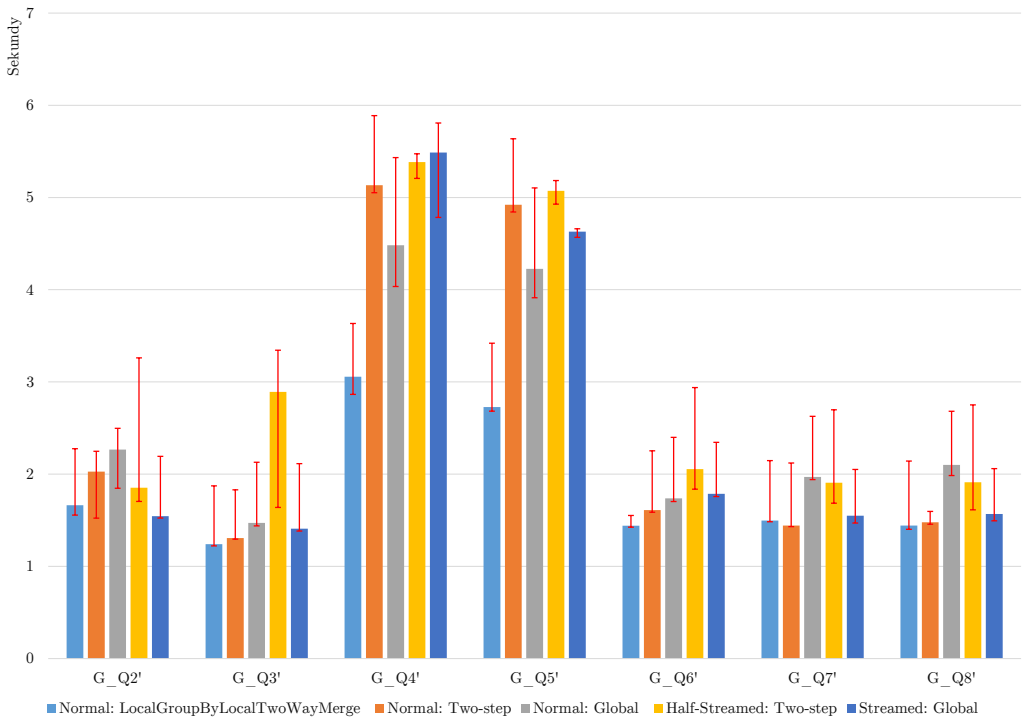
Na největším grafu platí, že použité ukládání List je pomalejší než Bucket, kvůli indirekci navíc. Na menších grafech rozdíly ustupují a dokonce nastavají situace, kdy je List rychlejší. Přesněji u dotazů G_Q4 a G_Q5. Přehození rolí si u nich vysvětlujeme overheadem za množství vytvářených polí (tj. hodně alokací, málo přístupů), které se vyrovná použité indirekci. Na grafu Amazon0601 u G_Q4 a G_Q5 dotazů je Streamed řešení pomalejší než Half-Streamed (List), protože také vytváří pole jako Bucket řešení (viz implementace TODO). U dalších grafů je pak počet vytváření polí mnohonosábně menší než počet přístupů k nim.

Z výsledků můžeme vyvodit, že vylepšená řešení pro single thread Group by jsou výhodnější z hlediska rychlosti vykonávání než řešení Normal. Nyní přejdeme k výsledkům paralelizace.

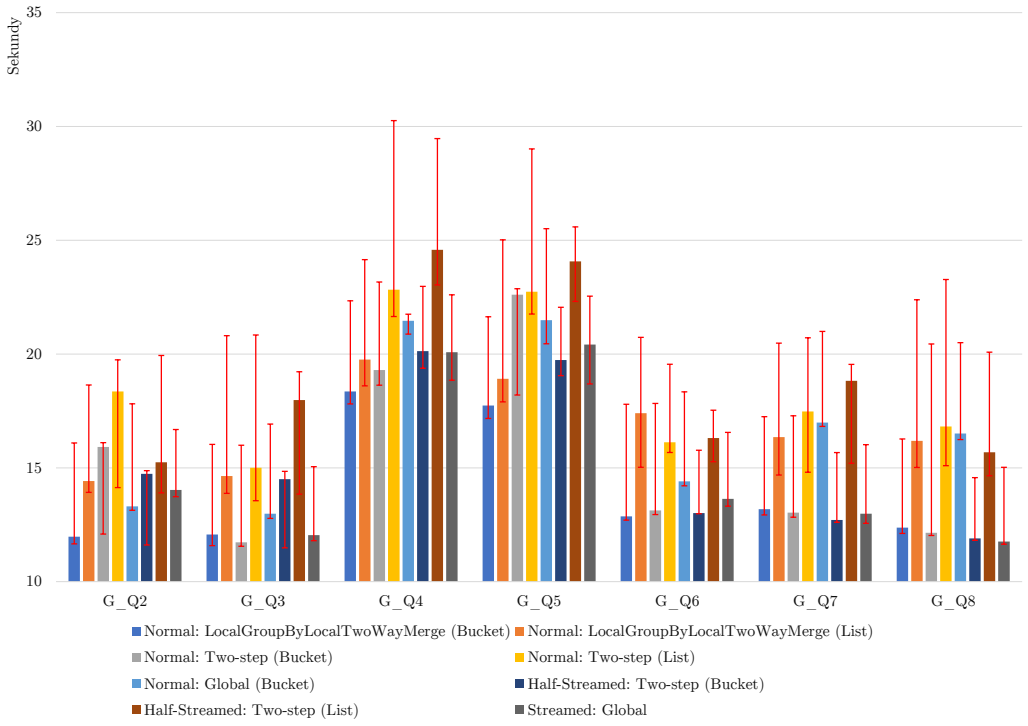
Paralelní řešení používají doposud zmíněná značení. Global řešení seskupuje výsledky globálně pomocí paralelní mapy (`ConcurrentDictionary`). Two-Step řešení seskupuje výsledky nejdříve lokálně pomocí mapy a následném mergi do paralelní mapy. LocalGroupByLocalTwoWayMerge řešení seskupuje lokálně a následně merguje výsledky vláken po dvojicích. Toto Mergování si můžeme představit jako binární strom. Listy jsou výsledky vláken a vnitřní vrcholy jsou akce mergování. Výsledky paralelizování jsou zobrazeny na obrázcích 4.20, 4.22 a 4.24. Obrázky 4.21, 4.23 a 4.25 obsahují výsledky dotazů bez agregačních funkcí.



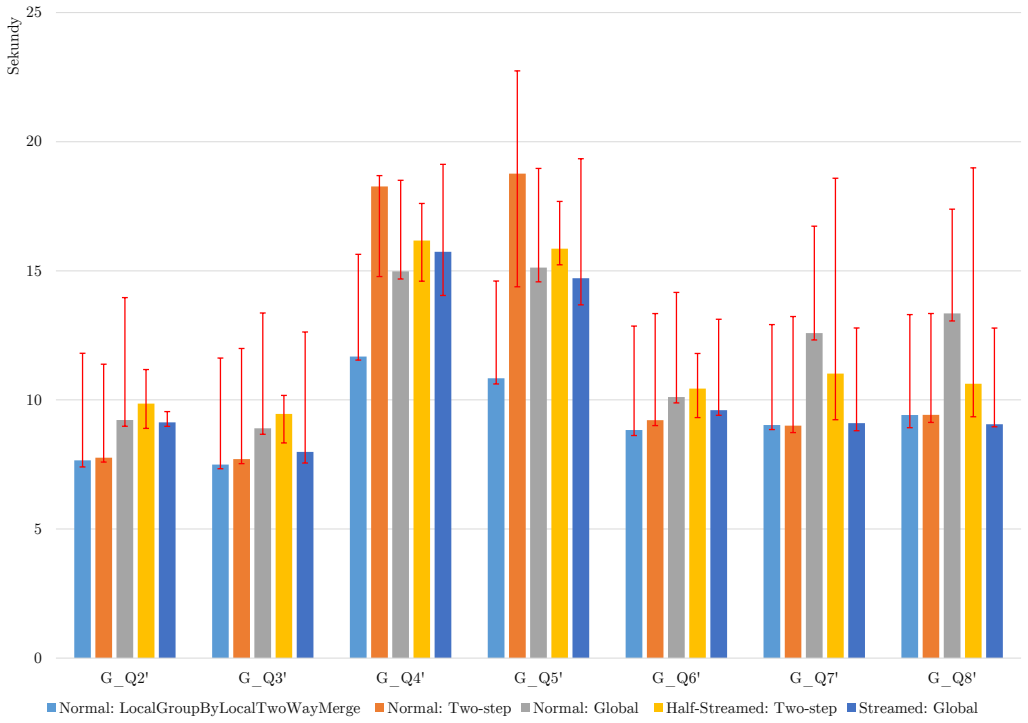
Obrázek 4.20: Doba vykonání dotazů Group by pro graf Amazon0601 (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 32373599.



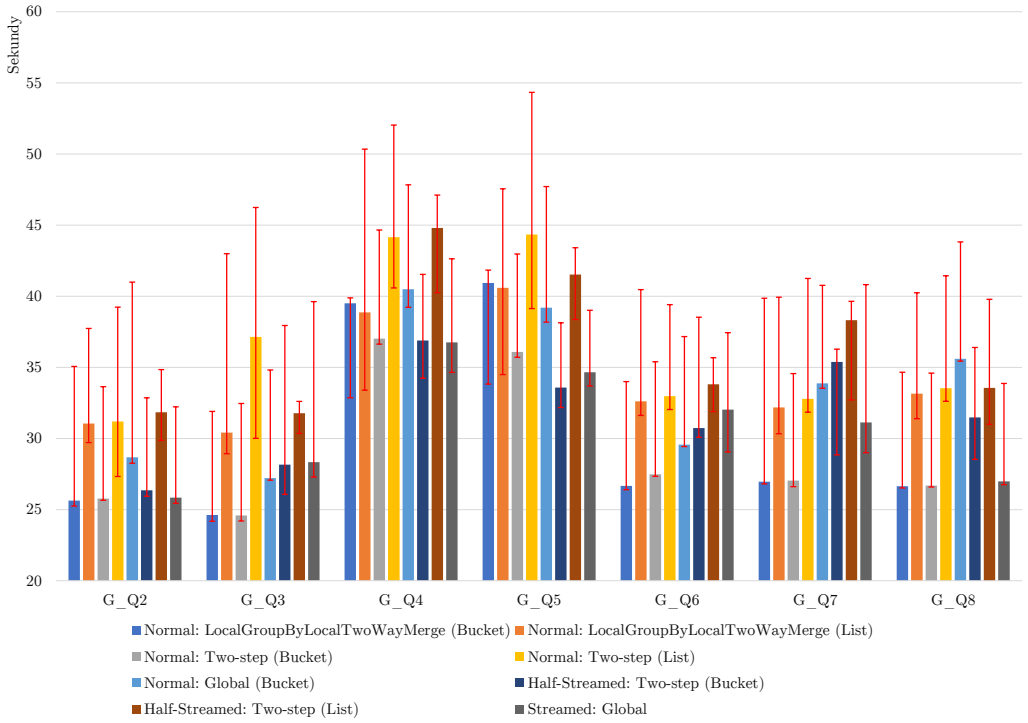
Obrázek 4.21: Doba vykonání dotazů Group by bez agr. funkcí pro graf Amazon0601 (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 32373599.



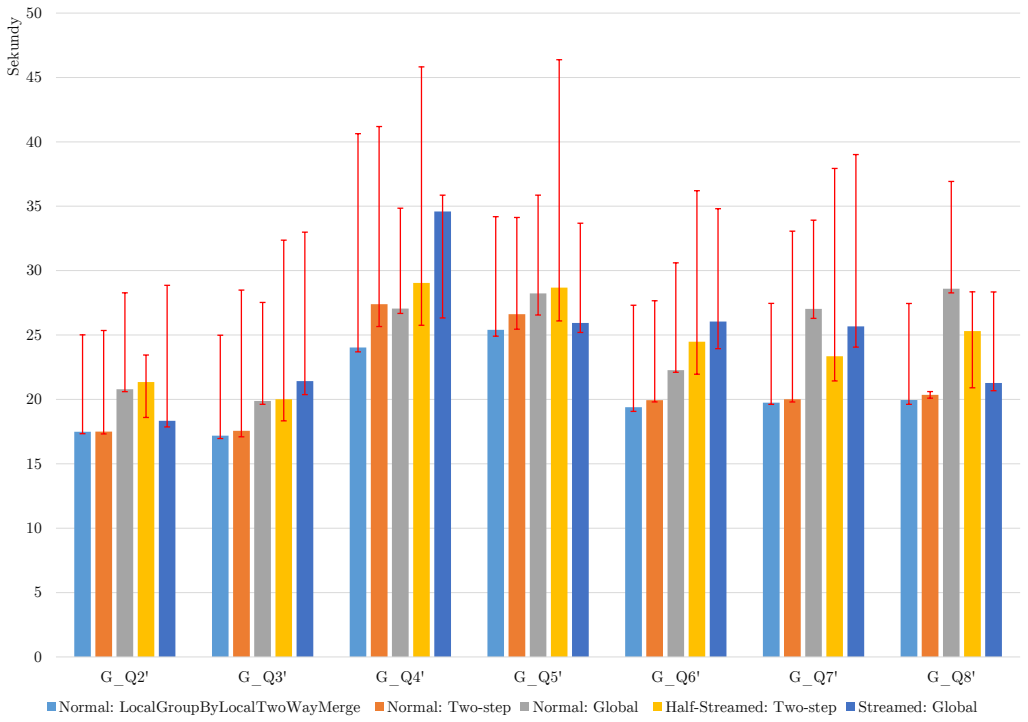
Obrázek 4.22: Doba vykonání dotazů Group by pro graf WebBerkStan (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 222498869.



Obrázek 4.23: Doba vykonání dotazů Group by bez agr. funkcí pro graf WebBerkStan (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 222498869.



Obrázek 4.24: Doba vykonání dotazů Group by pro graf As-Skitter (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 453674558.



Obrázek 4.25: Doba vykonání dotazů Group by bez agr. funkcí pro graf As-Skitter (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 453674558.

Byli jsme překvapeni, že vylepšená řešení se mnohdy nevyrovnala původním řešením. Streamed řešení, ačkoliv bylo nejrychlejší v jednovláknovém běhu, tak zde se pouze vyrovnalo Normal řešením a nebo bylo pomalejší. Danou situaci si vysvětlujeme synchronizací. Prvni vrstva synchronizace nastává u přistupu k paralelní mapě a čtení/vložení záznamu. Po získání `value` z mapy následuje druhá vrstva, která obsahuje volání thread-safe funkcí pro výpočet agregovaných hodnot pro danou skupinu. Z obrázků 4.9, 4.11 a 4.13 (Streamed řešení) jsme viděli cenu za synchronizaci na agregačních funkcích při přístupu osmi vláken. Pro představu pouhé režie paralelní mapy jsme otestovali overhead zvlášť čtení a vložení `ConcurrentDictionary` vůči `Dictionary` pro jedno vlákno. Následuje příklad kódu použitého při testu:

```
Random ran = new Random(100100);
Dictionary<int, int> map = new Dictionary<int, int>();
ConcurrentDictionary<int, int> parMap =
    new ConcurrentDictionary<int, int>();
...
// Insert test. Assuming the maps are empty.
for (int i = 0; i < 1_000_000; i++)
{
    var val = ran.Next();
    // Based on the used map choose (1) or (2).
    (1) if (!map.TryGetValue(val, out int value)) map.Add(val, i);
    (2) var tmp = parMap.GetOrAdd(val, i);
}
...
// Read test. Assuming the maps contain keys from 0 to 1_000_000.
for (int i = 0; i < 100_000_000; i++)
{
    var val = ran.Next(0, 1_000_000);
    // Based on the used map choose (1) or (2).
    (1) if (!map.TryGetValue(val, out int value));
    (2) var tmp = parMap.GetOrAdd(val, val);
}
```

Test	**Dict**	**ConDict**	**ConDict/Dict**
Insert 10^6	165	667	4,04
Insert 10^7	2476	11272	4,55
Read 10^8	19002	21516	1,13

Tabulka 4.9: Výsledky testování map v milisekundách. Běh v jednom vlákně. Měření dle kódu výše. Dict = Dictionary a ConDict = ConcurrentDictionary. Generování prvků pomocí třídy Random se seedem 100100. Měřeno pomocí třídy Stopwatch. Výsledek zvolen jako průměr pěti měření. Test Insert n provádí vkládání n náhodně vygenerovaných prvků do prázdné mapy. Test Read n provádí n čtení z rozsahu 0 až 1000000. Poměr je roven podílu času paralelní mapy a mapy.

Tabulka naměřených hodnot testování (4.9) ukazuje, že pouhé vkládání náhodně generovaných prvků do paralelní mapy trvá průměrně 4x déle. Samostatné čtení náhodně generovaných hodnot, které existují v mapě, je průměrně o 13% pomalejší. Provedli jsme další test. Test bude simulovat vkládání náhodných prvků do paralelní mapy osmi vlákny. Daná situace je velmi podobná naší situaci v Group by.

```
ConcurrentDictionary<int, int> parMap =
    new ConcurrentDictionary<int, int>();
...
// Insert test. Assuming the maps are empty. (case Insert 10^6)
Parallel.Invoke(
() => Insert(parMap, 0, 125_000, new Random(100100)),
() => Insert(parMap, 125_000, 250_000, new Random(100100)),
...
public static void Insert(ConcurrentDictionary<int, int> dict,
                          int start, int end, Random ran) {
    for (int i = 0; i < (end - start); i++)
        var v = dict.GetOrAdd(ran.Next(start, end), i);
}
```

Test	**Dict (1 thread)**	**ConDict (8 threads)**	**ConDict/Dict**
Insert 10^6	165	406	2,601
Insert 10^7	2476	4714	1,903

Tabulka 4.10: Výsledky testování map v milisekundách. Měřeno dle kódu výše. Dict = Dictionary a ConDict = ConcurrentDictionary. Dictionary vykoná práci v jednom vlákně. ConcurrentDictionary běží paralelně v osmi vláknech. Generování prvků pomocí třídy Random se seedem 100100. Měřeno pomocí třídy Stopwatch. Výsledek zvolen jako průměr pěti měření. Test Insert n provádí vkládání n náhodně vygenerovaných prvků do prázdné mapy. Poměr je roven podílu času paralelní mapy a mapy. Každé vlákno vkládá stejný počet náhodně generovaných prvků z určitého rozsahu.

Z tabulky porovnání vkládání 4.10 vidíme, že samotná paralelizace je pro náš počet vkládání prvků pomalejší. Tedy obecné zpomalení je zřetelné u řešení používající paralelní mapu. Streamed řešení vůči jeho protějšku Normal: Global je místy pomalejší. Děje se tak ve dvou grafech. První je graf As-Skitter u dotazů G_3/G_3′ a G_6/G_6′. Druhý graf je Amazon0601 na dotazech G_4/G_4′ a G_5/G_5′. Myslíme si, že se jedná o specifické situace pro dané grafy a nedokážeme je plně zodpovědět, jelikož navzájem a pro graf WebBerkStan nenastávají. Obecně pro dotazy G_6/G_6′ až G_8/G_8′ vidíme mírné zrychlení Streamed řešení, protože šetří drahou režii za vytváření nových skupin. Pro Normal: Global nastává zpomalení, jelikož se při vkládání do mapy častěji vyvolá drahé porovnání klíčů pomocí Properties skrze tabulku výsledků.

Můžeme zde aplikovat výsledky z single-thread řešení pro řešení Normal: Two-Step proti Half-Streamed: Two-Step. Half-Streamed řešení je zde opět pomalejší

než Normal řešení nebo jsou vyrovnané. Dále zde opět vidíme zpomalení implementace List vůči Bucket, které jsme viděli u single-thread řešení. Situace při které je List rychlejší nastávala pro dotazy G_Q4 a G_Q5 na grafu Amazon0601 a WebBerkStan. Nyní nastává pouze pro graf Amazon0601 s řešením Normal: LocalGroupByLocalTwoWayMerge. Two-step (List) řešení při mergi překopírovává větší množství dat, proto u něj daný jev už nenastává.

Všem řešením dominuje Normal: LocalGroupByLocalTwoWayMerge, které provádí vše lokálně a ujišťuje nás v předpokladu, že hlavní overhead je způsoben synchronizací. Například dané řešení vůči Normal: Two-Step. Je zde vidět overhead za použití paralelní mapy vůči mergování lokálně po dvojicích, jelikož samotný první krok je totožný pro obě řešení. Zpomalení Normal: Two-Step je ještě znatelnější pro dotazy G_Q4/G_Q4′ a G_Q5/G_Q5′, kdy se vkládá množství skupin do paralelní mapy.

Z výsledků usuzujeme, že vylepšená řešení pro náš případ paralelizace neposkytují z hlediska rychlosti vykonání znatelné výhody oproti stávajícím řešením. Pro budoucí práci by bylo vhodné udělat podrobnější testování pro rozličné počty použitých vláken a sledovat skalabilitu daných řešení. Další možný budoucí výzkum může sledovat obecný problém rozdělení dat vláken během vyhledávání. Normal přistup rovnoměrně rozděluje množství výsledků pro každé vlákno. Rozdělení práce vylepšených řešení závísí na počtu vyhledaných výsledků v každém vlákně. Aby došlo k rovnoměrnému rozdělení práce tak se každému vláknu přidělují malé skupiny vrcholů k prohledání. Nicméně, dané řešení nemůže zaručit stoprocentně rovnoměrné rozdělení práce.

Tímto jsme zakončili prezentaci výsledků. Všechna nasbíraná data použitá k tvorbě grafů je možné nalézt v příloze výsledků benchmarku (A.4).

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 Výsledky benchmarku pro jednotlivé grafy

Součástí této přílohy je výstup benchmarku při vykonaném experimentu (kapitola 4). Soubory jsou rozděleny do složek podle názvu grafů. Samotné výstupy nejsou nijak setříděny.

A.5 druha priloha

Priloha po prvni strance priloh