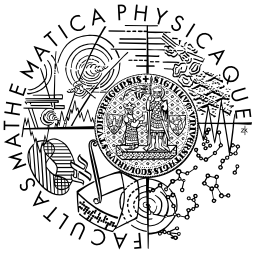


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

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

1.3 Paralelismus

2. Analýza

V této kapitole se pokusíme analyzovat problémy výstavby a úpravy dotazovacího enginu dle zadání práce. Zároveň poskytneme možná řešení daných problémů. Budeme zde postupovat 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. V momentě načtění graf bude pouze statický, tedy nebude docházet k žádným změnám. Nad grafem se pak vykoná uživatelsky definovaný dotaz. Dané omezení jsme zvolili, protože hlavním cílem je testovat pouze části Group by a Order by. Vytvořit reálnou grafovou databázi by zabralo netriviální časové období.

Při obecném pohledu na engine jsme lokalizovali hlavní bloky výstavby, které musíme uvážit. Jsou to: reprezentace grafu, parsování uživatelského dotazu, výrazy (expressions) a dotaz/vykonání dotazu. Graf nám bude simulovat grafovou databázi. Samotně pak určuje formát objektů, nad kterými je vykonán uživatelský dotaz. Při parsování se načítá uživatelký dotaz do interní reprezentace. Expressions slouží k výpočtu hodnot z uživatelsky zadaných výrazů. Například v části order by x.PropOne, musíme vědět, jak reprezentovat výraz x.PropOne a získat jeho hodnotu. Na základě interní reprezentace se musí vytvořit struktury dotazu a definovat 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. Graf bude simulovat grafovou databázi. Z části 1 jsou hlavními faktory námi zvolená podmnožina jazyku PGQL a že se jedná o Property graf. Pro případy nejednoznačnosti označíme `elType` jako typ elementu v Property grafu a `propType` jako typ Property.

2.2.1 Elementy grafu a jejich typ

Musíme zvažovat reprezentaci elementů grafu a jejich `elType`. V našem případě jsou elementy pouze vrcholy a orientované hrany. `elType` definuje seznam Properties na elementu. Properties jsou také typované. Vrchol a hrana musí mít rozdílný `elType`, ale samotné Properties se mohou opakovat pro oba druhy elementů. 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 hodnotám jeho Properties.

V naší představě je řešení následovné. Elementy budou třídy. 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 `ID`, abychom elementy dokázali rozlišit. Předek navíc bude znát svůj `elType`. Bude se jednat o ukazatel na třídu. Daná třída by reprezentovala pouze jeden `elType` a bude společná všem elementům majících daný `elType`. V třídě by byl obsažen seznam `IDs` elementů daného typu, jejich pořadí (např: dle vkládání do seznamu) a Properties v podobě polí s hodnotami. Property musí být přístupná skrze mapu/slovník, protože může nastat situace, kdy daná Property na elementu neexistuje. Pro náš případ nebudeme uvažovat situaci, kdy Property pro nějaký element nemá definovanou hodnotu. Properties tedy budou 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`. Samotný přístup pak může být realizován například generickou funkcí.

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 držíme odkaz na 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 grafu, chceme být schopni přistoupit k jeho sousedním elementů v obsajující struktuře za předpokladu, že víme, jestli se jedná o hranu nebo vrchol.

K vyřešení daných problému v naší představě bychom použili tři pole. Pole vrcholů, pole out hran a in hran. Zde by bylo vhodné vytvořit nové potomky obecné hrany: out hrana a in hrana. 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 vrcholu, stačí procházet pole out/in hran pomocí rozsahů uložených v daném vrcholu. Tedy čtyř indexů. Skrze indexy můžeme pak pole 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 grafová data

Vstupní soubory musí obsahovat informace nutné pro Property graf. Budeme očekávat dva druhy souborů. Soubory schémat typů elementů a jejich Properties. Datové soubory pak budou obsahovat konkrétní data elementů.

Protože každý element grafu má svůj `elType`, budeme mít na vstupu dva soubory schémat pro hrany a vrcholy. Schéma bude obsahovat informace o všech `elType` a `propType` vyskytujících se v grafu. Pro `elType` je důležitý název a výčet Properties. Properties pak musí nést svůj název a `propType`. Vidíme, že se jedná jen o výčet `(name/value)` dvojic (např. `(PropertyOne, integer)`). V tomto případě se nám jeví nejvhodnější zvolit pro reprezentaci schémat formát JSON. `elType` bude reprezentován JSON objektem. Bude obsahovat položku `Kind`, jejíž hodnota udává jméno `elType`. Za ní bude následovat výčet Properties. Properties budou reprezentovány dvojicí `(propName/propType)`. Záznamy pak budou obsaženy v JSON poli:

```
Soubor schéma vrcholů:
[    { "Kind": "BasicNode" },
     { "Kind": "BasicNodeTwo", "PropertyOne": "integer" } ]
Soubor schéma hran:"
[    { "Kind": "BasicEdge" },
     { "Kind": "BasicEdgeTwo", "PropertyOne": "integer" } ]
```

Jako určující `propTypes` pro náš případ enginu bychom chtěli zvolit dva druhy. První by byla číselná hodnota značená integer (32-bit integer). Další by představoval řetězec značený string. Práce s řetězci je obecný problém a existuje mnoho znakových sad, proto bychom chtěli zvolit vstupní řetězce pouze se znaky ASCII. Dané dva druhy představují základní typy použitých v komerčních sférách.

Samotná data budou obsažena opět ve dvou separatních souborech pro hrany a vrcholy. Chtěli bychom reprezentovat konkrétní data pomocí jednoduchého .csv souboru. Každý řádek reprezentuje jednu hranu/vrchol. V první řadě řádek musí obsahovat unikatní `ID` elementu a jeho `elType`. Za `elType` následuje seznam hodnot Properties v pořadí určených schématem. Pro hrany existuje na řádku navíc záznam `ID` vrcholů, které spojuje. Oddělovače mezi daty jsou implementační

detail. Pro naše účely se jedná o dostačující formát a poskytuje nám jednoduché možnosti parsování. Pokud by docházelo v budoucnu k rozšířením, například více slovné Property nebo XML Property, musí dojít k úpravě daných formátů.

Pro výše zmíněné schéma by datové soubory mohly vypadat následovně:

```
Soubor hran:
ID elType fromID toID Properties // bez této hlavičky
50 BasicEdge 0 0
51 BasicEdgeTwo 0 1 44
...
Soubor vrcholů:
ID elType Properties // bez této hlavičky
0 BasicNode
1 BasicNodeTwo 42
...
```

Analyzovali a navrhli jsme způsob reprezentace grafu společně s formátem datových souborů. Samotné načítání je už implementační detail. Nyní musíme analyzovat způsob získání informací z uživatelem zadaného dotazu.

2.3 Parsování uživatelského dotazu

Uživatelský dotaz se pohybuje v rozsahu definovaném v sekci PGQL 1.2. Nicméně, je zde nutné přemýšlet i nad možnými rozšířeními. Například může dojít k přidaní částí Where a Having spolu s nutností porovnávání (Where x.PropOne >= 10). Proto se budeme snažit držet základních principů object oriented programming a volit vhodné návrhové vzory.

K načtení uživatelského dotazu se nám jeví jako nejvhodnější způsob použít techniky známé z překladačů programovacích jazyků. Budeme vycházet ze základních principů knihy o překladačích (Aho a kol., 2006). V prvním kroku dojde k lexikální analýze uživatelsky zadaného řetězce. Dojde k vytvoření tokenů. V druhém kroku dojde k syntaktické a semantické analýze tokenů. Metodou top-down parsing (Aho a kol., 2006, str. 217) se vytvoří stromová struktura reprezentující daný dotaz. Poslední krok provede vytvoření tříd reprezentující dotaz pomocí iterace stromové struktury. Iterace a sběr dat ze stromové struktury budou implementovány návrhovým vzorem Visitor (Gamma a kol., 1994, str. 331). V naši představě bychom chtěli vygenerovat stromovou strukturu pro každou hlavní část dotazu (Match, Select, Order by a Group by). Nyní bychom mohli sestavit Visitor pro každou část separatně a vyřadit tak nutnost jednoho globálnáho Visitoru. Dané postupy nám pak umožní jednoduše pracovat s naší podmnožinou jazyka PGQL (sekce 1.2).

2.3.1 Match a proměnné

Každá hlavní část dotazu po sesbírání informací pomocí Visitoru vygeneruje určité struktury. Pro Match se přímočaře naskytuje reprezentovat posloupnosti vrcholů a hran pomocí polí. Každá posloupnost oddělená čárkou bude obsažena v samostatném poli. Pole bude obsahovat třídy. Třída si musí pamatovat jakou

proměnou reprezentuje, `elType` pokud je definován a jde-li o hranu (in/out/any) nebo vrchol. Jedná se o všechny nutné informace, které můžeme následně využít k vytvoření vzoru prohledávání grafu. Všimout si musíme faktu, že Match část definuje proměnné ve zbytku dotazu. Během parsování musíme určit zda se jedná o validní proměnnou a při vypočtech hodnot výrazů je nutné vědět přesně k jaké proměnné musíme přistoupit. Problém se dá řešit vytvořením mapy/slovníku přistupných proměnných pro zbytek dotazu. Proměnným pak můžeme přiřadit `ID`.

2.3.2 Select, Order/Group by

Ostatní části Group by, Order by a Select obsahují výrazy proměnných (např: order by x), přístup k Properties proměnných (např: select x.PropOne) nebo volání agregačních funkcí (`min`, `max`, `avg`, `sum` a `count`). Proměnné zde představují elementy grafu. Várazy se však musí evaluovat za běhu programu. Dalším problémem je, že výrazy mají různorodé návratové hodnoty. Výraz x (`ID` vrcholu) lze chápat jako integer. Váraz x.PropOne má návratovou hodnotu dle `propType`, který je definovám ve vstupním schématu. Agregační funkce `min`, `max` mají návratovou hodnotu definovanou na základě jejich vstupních argumentů. Funkce `sum` a `count` by měli ideálně pracovat s typem, který by předešel přetečení. U `avg` se očekává hodnota s desetinnou čárkou. V budoucnu však může dojít k rozšířením a vyvstanou složitější výrazy, například infixová notace x.PropOne + y.PropOne nebo zmiňované porovnání z Where/Having části. Problém nám usnadňuje fakt, že Properties nesoucí stejné jméno mají stejný `propType`. Pokud ne, je nutné určit vhodnou návratovou hodnotu. Navíc musíme brát v potaz, že daný výraz se nemusí vyhodnotit, například absence Property na vrcholu. Proto jsme byli nuceni vymyslet systém výrazů (expressions).

2.3.3 Expressions

Systém vytváření a vyhodnocování výrazů efektivně za běhu je obecně složitý problém. Omezíme se pouze na případy: přístup k proměnné, přístup k hodnotě Property proměnné a agregační funkce (`min`, `max`, `avg`, `sum` a `count`).

Základní myšlenka je reprezentovat výraz pomocí stromové struktury. Každý vrchol stromové struktury bude reprezentovat určitou akci. Vrcholy budou výše vypsané výrazy. Na struktuře bude existovat metoda pro vyhodnocení. Její návratová hodnota bude dvojice úspěch vyhodnocení + vypočtená hodnota. Dané struktury musí být read-only, protože se budou využívat v paralelním prostřední. Metody by se mohli libovolně dodávat při nutnosti použití nových struktur.

Následuje ukázka možného kódu v jazyce C#:

```
// Base classes
abstract class Expression { }
abstract class ExpressionReturnValue<T>: Expression {
  public abstract bool TryEvaluate(Element[] elms, out T retVal);
}

abstract class VariableAccess<T>: ExpressionReturnValue<T> {
     readonly int accessedVariableID;
}
```

Třída reprezentující přístup k `ID` proměnné:

```
class VariableIDAccess: VariableAccess<int> {
  public override bool TryEvaluate(Element[] elms, out int retVal) {
     returnValue = elms[accessedVariableID].ID;
     return true;
  }
}
```

Třída `VariableAccess` nám poskytuje abstrakci pro přístup k proměnné. Položka `accessedVariableID` určuje k jaké proměnné se má přistoupit. Zde předpokládáme, že pole `Element[]` obsahuje proměnné přesně v pořadí, jak se vyskytly v části Match. Tedy pokud je roven Match (x) -> (y), tak jeden výsledek hledání by bylo pole obsahující dva elementy x a y. Jedná se pouze o ilustrační příklad.

Případ přístupu k Property by mohl vypadat následovně:

```
class VariablePropertyAccess<T>: VariableAccess<T> {
  reaonly int accessedPropertyID;
  public override bool TryEvaluate(Element[] elms, out T retVal) {
    return elms[accessedVariableID].
                GetPropertyValue<T>(accessedPropertyID, out retVal);
  }
}
```

Zde dojde k volání metody na elementu grafu, který přistoupí k třídě reprezentující jeho `elType`. Třída pak na základě existence Property vrátí hodnotu nebo neuspěje. Timto dokážeme vyřešit základní definované problémy.

Zbává uvažovat, jakým způsobem reprezentovat agregační funkce. Agregační funkce představují několik problémů. Funkce je vypočtená pouze pro skupiny. Skupiny jsou vytvářeny v části Group by. Jejich návratové hodnoty jsou finální pouze po dokončení Group by. Vstupem funkcí je výstupní hodnota uživatelem zadané expression. Argument, dle kterého se aktualizuje agregovaná hodnota, je nutný znát pouze v době vykonání Group by. Dle naši představy je ideální vytvořit dva separátní koncepty. První koncept bude zahrnovat výpočet hodnot argumentu společně s logikou agregační funkce. Koncept bude reprezentován třídou, která vlastní stromovou strukturu dle předchozího příkladu. Zároveň bude obsahovat logiku počítané funkce. Například logika funkce `min` je porovnat dvě hodnoty a vybrat menší. Na vstupu dané funkce pak bude úložiště hodnoty dané skupiny. Všechny počítané agregační funkce zadané uživatelem označíme pomocí `ID`. Druhý koncept představuje nový potomek třídy expression. Daný potomek si pamatuje `ID` přistupované agregační funkce a na vstupu očekává strukturu reprezentující skupinu. K hodnotě počítané agregace přistoupíme pomocí její `ID`.

Následuje ukázka druhého konceptu:

```
class GroupAggValueAccess<T>: ExpressionReturnValue<T> {
  reaonly int accessedAggregationFuncID;
  public override bool TryEvaluate(Group group, out T retVal) {
    retVal = group.GetAggValue<T>(accessedAggregationFuncID);
    return true; }}
```

`Group` reprezentuje výsledky jedné skupiny. `accessedAggregationFuncID` je `ID` vypočítané agregační funkce. Hodnota funkce se vrací pomocí `GetAggValue<T>`.

Následuje ukázka prvního konceptu:

```
abstract class Aggregation { }
abstract class Aggregation<T>: Aggregation {
  public ExpressionReturnValue<T> expr; // Argument of the agg. func.
  public abstract void Apply(ValueStorage storage, Element[] elms);
}

public class Sum<T>: Aggregation<T>{
  public override void Apply(ValueStorage storage, Element[] elms) {
    if (expr.TryEvaluate(elms, out T retVal)) {
      storage.value += retVal;
    }
  }
}
```

Zde vidíme položku `expr`, která reprezentuje vstupní expression agregační funkce. Funkce `Apply` je logikou funkce. Vidíme funkci `Sum`. Logikou je přičtení vypočítané hodnoty do poskytnutého úložište, pokud dojde k úspěšné evaluaci výrazu. Pomocí našeho návrhu pak můžeme vyřešit i budoucí rozšířešení, jako třeba aritmetické operátory nebo porovnání.

Třídy pro binární sčítání mohou vypadat takto:

```
class ExpressionBinOperation<T>: ExpressionReturnValue<T> {
  public ExpressionReturnValue<T> expr1;
  public ExpressionReturnValue<T> expr2;
}

class ExpressionIntegerAdd: ExpressionBinOperation<int>{
  public override bool TryEvaluate(Element[] elms, out int retVal) {
    if (expr1.TryEvaluate(elms, out int expr1Val) &&
        expr2.TryEvaluate(elms, out int expr2Val)) {
      retVal = expr1Val + expr2Val;
      return true;
    } else {
      retVal = default;
      return false;
    }
  }
}
```

Tímto jsme vyřešeli problémy parsování a reprezentace expressions pro náš engine. Kód je pouze ilustrační a není finální. Použili jsme jej, protože poskytoval lepší možnosti vysvětlení konceptu než obrázek. Nyní přistoupíme k problémům vykonávání dotazu.

2.4 Vykonání dotazu

Máme připravené obecné podklady. Víme jak reprezentovat graf a jak budeme získávat informace z uživatelem zadaného dotazu. Dotaz bude vykonán nad naší reprezentací grafu. Abychom splnili zadání, tak Group/Order by musí být vykonány po dokončení vyhledávání vzoru. Vylepšená řešení budou dané části vykonávat v průběhu vyhledávání. V ideálním případě chceme dosáhnout toho, aby dotazovací engine poskytoval dva módy. Mód zde reprezentuje způsob vykonání dotazu. Uživatel enginu si při spuštění vybere chtěný mód. Tudíž, módy musí v programu koexistovat. V této části analyzujeme obecný model vykonání, který posléze v sekci 2.8 upravíme, aby vykonával Group/Order by v průběhu hledání.

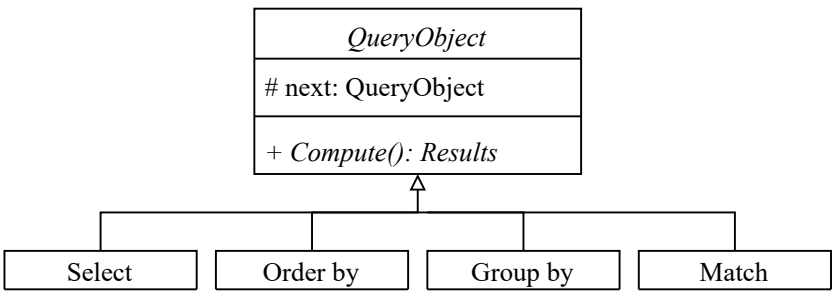
Při zkoumání vlastností hlavních částí PGQL (sekce 1.2) jsme si uvědomili jejich separaci logiky z hlediska vykonávání dotazu. Match prohledává graf a produkuje výsledky. Select výsledky vypisuje. Order/Group by třídí/seskupuje vyprodukované výsledky. Filtrace výsledků je prováděna v části Where a Having. Tedy dané části se mohou vyvíjet nezávisle na sobě a následně propojit dle priorit.

Priority (zleva největší):

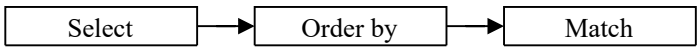
```
Match > Where > Group by > Having > Order by > Select
```

Match vyprodukuje výsledky, Where je profiltruje, Group by je seskupí, Having opět profiltruje, následně jsou setříděny a jako poslední krok se provede výpis uživateli. Propojení pak tvoří primitivní exekuční plán. Při podrobnějším zkoumání jsme zjistili, že dané schéma značně připomíná návrhový vzor Pipes and Filters (Buschmann a kol., 1996, str. 53). Ačkoliv v naší práci použitá podmnožina PGQL (sekce 1.2) neobsahuje Having a Where, jsme si vědomi provázanosti částí Match/Where a Group by/Having. Pokud by byli v budoucnu doimplementovány, tak mohou být propojeny pro docílení lepší výkonnosti.

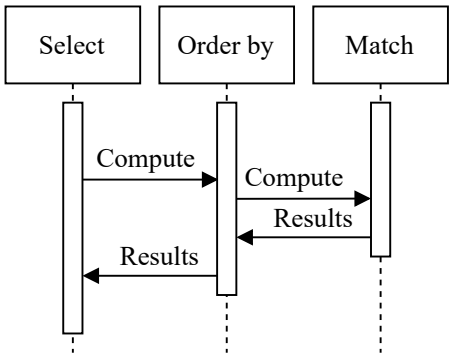
Poznatky jsme se rozhodli aplikovat v našem návrhu. Každá část dotazu bude reprezentována objektem (obrázek 2.1). Metoda `Compute` implementuje logiku objektu. Propojení se realizuje pomocí položky `next`. Dle uživatelského dotazu se vytvoří objekty a propojí dle priorit výše (obrázek 2.2). Propojejí bude realizováno od nejmenší po největší, protože práce s nejvyšší prioritou je vykonána první a po dokončení její práce ji už nepotřebujeme. Tedy můžeme uvólnit její zdroje. Při vykonání se provede rekurzivní voláni pro vyhodnocení objektů v položce `next` (obrázek 2.3).



Obrázek 2.1: UML class diagram objektů představující části dotazu.



Obrázek 2.2: Propojení objektů pomocí položky `next` pro dotaz select x match (x) order by x.



Obrázek 2.3: UML activity diagram rekurzivního volání metody `Compute` pro dotaz select x match (x) order by x.

2.4.1 Paralelizace vykonání dotazu

Poslední věc nutnou ošetřit je způsob paralelizace dotazu. Existuje několik možností. Paralelizace bude provedena pouze interně pro každy objekt nebo dojde k vypracování složitějšího modelu. V naší představě bychom volili první variantu. Pomocí ní zaručíme nezávislost objektů a nebudeme nutni vytvářet závislosti mezi objekty. Při analýze jednotlivých částí budeme vycházet z předpokladu, že existuje `ThreadPool` vláken. Do `ThreadPool` budeme zařazovat úkony k vykonání (`Tasks`).

2.4.2 Formát výsledků

Mezi částmi dochází k předávání výsledků. Výsledky musí mít definovaný formát, aby každá část dokázala správně provádět svou logiku. V části Match dochází ke generování obecných výsledků. V části Group by dojde k vytvoření skupin a výpočtů agregačních funkcí. Pokud je v uživatelském dotazu zahrnuto Group by, tak zbylé části musí očekávat jiný formát výsledků. Daný formát musí obsahovat skupiny a výsledné hodnoty agregačních funkcí.

Obecné výsledky hledání budeme ukládat do tabulky. Jeden řádek tabulky bude reprezentovat jeden výsledek hledání. Nyní musíme rozhodnout, jaké informace do tabulky uložíme. V části Match se pracuje s elementy grafu. Jeden výsledek hledání obsahuje seznam elementů, které odpovídají vzoru. Máme dvě možnosti, jak daný výsledek zpracovat. První varianta v části Match při jeho nalezení vypočte hodnoty všech výrazů obsažených ve zbytku dotazu. Sloupeček představuje hodnoty jednoho výrazu. Tato varianta nám nepřišla vhodná, protože by objekt Match musel znát informace o výrazech v celém dotazu. Navíc výrazy v částech mohou být rozdílné. Tedy se vytváří nutnost vytvářet sloupečky hodnot v moment, kdy se nepotřebují a tím se navyšuje spotřeba paměti.

Příkladem může být dotaz:

```
select x.P1, ..., x.Pn match (x) -> (y) order by y.P1, ..., y.Pm
n, m belongs to N and n, m > 1
```

Zde vidíme množství výrazů. Pokud bychom aplikovali první variantu, tak v Match části musíme vygenerovat $m+n$ sloupců hodnot. Z tohoto důvodu chceme zvolit jinou variantu. Do tabulky budeme ukládat elementy grafu. Sloupeček tabulky bude reprezentovat jednu proměnnou z dotazu Match. Tedy tabulka má počet sloupečku rovný počtu unikátních proměnných. Pro stejný dotaz vytvoříme pouze dva sloupečky. Abychom zamezili stejnému problému i z opačné strany (tj. mnoho proměnných a málo výrazů), tak budeme ukládat pouze proměnné, ke kterým se přistupuje v ostatních částech. Tímto jsme vyřešili paměťový problém, ale nastal problém výkonnosti. Vyvstává totiž nutnost vypočítávat hodnoty výrazů znova, přestože jsme je už v minulosti počítali (např. při třídění se několikrát porovnává stejný výsledek s jinými). Problém se dá částečně řešit cachováním výsledků, ale ten ponecháme na analůzu zbylých částí. V tento moment jsme se rozhodli jít cestou menší paměťové náročnosti na úkor výkonnosti. Pro výsledky Group by můžeme opět v představě volit tabulku. Jediný rozdíl bude, že řádek zde bude reprezentovat jednu skupinu (opět uložené elementy grafu) společně s vypočtenými hodnotami agregačních funkcí.

2.4.3 Proxy třída jako řádek tabulky

Musíme být schopni pracovat s řádky tabulek. Pomocí řádků v tabulce se musí vyhodnotit výrazy v částech Group/Order by a Select. Vstupním argumentem expression by měl být pouze jeden řádek. Přesouvání řádku o více sloupcích je drahé. Pro docílení efektivní práce s řádky budeme řádek reprezentovat proxy třídou. Proxy třída bude návratová hodnota funkce indexeru tabulky (`ResultTable[i]`, kde `i` je index řádku). Třída poskytne metody pro přístup k elementům nebo výsledkům agr. funkcí ve sloupečcích pro daný řádek tabulky. V ideálním případě si bude pamatovat pouze index reprezentujícího řádku a odkaz na tabulku. Nyní pokud budeme chtít vyhodnotit výraz pro i-tý řádek tabulky, tak zavoláním indexeru dostaneme proxy třídu a tu použijeme k vyhodnocení výrazu.

Analyzovali a navrhli jsme obecně způsob vykonání dotazu společně s formátem předávaných výsledků mezi částmi. Nyní přejdeme k analýze jednotlivých částí dotazu. V analýze jsme se rozhodli vychenat část Select, protože není podstatná pro naši práci.

2.5 Match a prohledávání grafu

Match část má za úkol najít všechny podgrafy v grafu odpovídající zadanému vzoru. Vlastnosti hledání jsou definované jazykem PGQL (sekce 1.2). Vzor se vždy skládá z posloupnosti vrcholů a hran. Na každý prvek posloupnosti se můžeme dívat jako na placeholder nějakého elementu grafu.

Vlastnosti hledání:

- Výsledky hledání jsou podgrafy homomorfní se zadaným vzorem.

- Hrana se může opakovat několikrát v rámci jednoho vzoru.
- Proměnná hrany ve vzoru se může použít pouze jednou.
- Dvě rozdílné proměnné mohou obsahovat stejný element.
- Shodnost elementů se ověřuje pouze na opakující se proměnné.

2.5.1 BFS vs DFS

Hledání podgrafu v grafu je obecně složitý problém. Cílem této práce není navrhnout algoritmus pro vyhledávání vzoru, proto jsme se rozhodli inspirovat a použít obecný postup k řešení daného problému. Mezi základní postupy vyhledávání vzoru patří prohledávání do šířky (BFS) a prohledávání do hloubky (DFS) (Needham a Hodler, 2019, kap. 4). Na základě 1. a 2. kapitoly článku Roth a kol. (2017) jsme vybrali algoritmus DFS, jelikož v průběhu prohledávání generuje menší množství mezivýsledů. To je dáno chováním BFS. BFS v každém kroku musí prozkoumat všechny sousedy políček z předešlého kroku. Toho se docílí vložením nových sousedů do fronty (obecná struktura `queue` first in first out). Fronta se tak rychle zvětšuje. DFS naopak potřebuje znát sousedy pouze aktuálně prohledávaných vrcholů.

2.5.2 Hledaný vzor

K aplikaci algoritmu musíme vytvořit strukturu (vzor) představující hledaný podgraf. V sekci 2.3.1 jsme uvedli, že posloupnosti oddělené čárkou budou reprezentovány jako pole tříd obsahující informace o proměnných. Jedno pole je ekvivalentní jedné posloupnosti. Pro zřetelnost budeme hovořit o jednom poli jako o řetězci.

Příklad dotazu se dvěma řetězci:

```
match (x) -> (y), (x) -> (q)
```

Řetězce nám nyní budou sloužit k vytvoření vzoru. Abychom mohli efektivně hledat daný vzor, potřebujeme z řetězců vytvořit souvislé komponenty. Souvislé komponenty vytvoříme propojením řetězců pomocí opakujících se proměnných. Tedy dva řetězce jsou propojeny právě tehdy, obsahují-li stejnou proměnnou. Propojením všech takových řetězců vytvoříme souvislé komponenty. Souvislá komponenta představuje hledaný vzor. Problém vyvstane, pokud nám vzniknou dvě separátní komponenty z jednoho dotazu. Tento případ nastene právě tehdy, když pro dvě komponenty neexistuje proměnná, která by je propojila. V takovém případě se můžeme dívat na dotaz, jako na skalární součin výsledků hledání dvou komponent. Stejný princip aplikujeme, pokud existuje vícero separátních komponent.

Příklad dotazu separátních komponent:

```
select x, y match (x), (y)
```

2.5.3 Průběh hledání

K nalezení všech podgrafů v grafu pořebujeme z každého vrcholu spustit DFS. Při DFS se bude kontrolovat, jestli průchod odpovídá hledanému vzoru. Procházení vždy začíná vrcholem, následně přístupem k hranám daného vrcholu a pak koncovému vrcholu hrany. K procházení grafu máme navrhnutou strukturu z sekce reprezentace grafu 2.2. Pokud dojde k nalezení podgrafu, tak výsledek bude uložen způsobem z sekce 2.4.2.

Vyhledávání separátních komponent vyřešíme následovně. V momentě, kdy nalezneme podgraf odpovídající jedné komponentě, tak se spustí DFS vyhledávání pro komponentu další. Teprve až projdeme všechny komponenty, výsledek se uloží do tabulky. V tuto chvíli budeme vlastnit finální výsledek hledáná, který můžeme uložit do tabulky. Zbavíme se tak nutnosti uchovávat mezivýsledky a následnému tvoření skalárního součinu.

2.5.4 Paralelizace hledání

Nyní přistoupíme k analýze paralelizace hledání. V paralelním řešení chceme použít co nejmenší počet synchronizačních primitiv. Ukládání výsledků do společné struktury by způsobilo značny overhead za synchronizaci. V ideálním případě bude probíhat hledání lokálně, následně pak dojde k efektivnímu mergi.

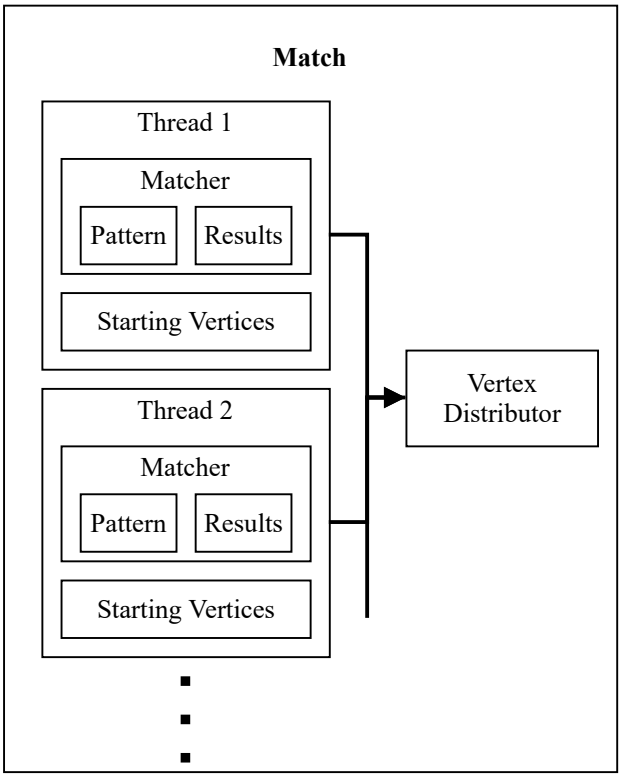
Jako řešení jsme zvolili jeden ze základních způsobů. Budeme paralelizovat prohledávání ze startovních vrcholů. Vyhledávání bude reprezentováno objektem (`Matcher`). `Matcher` vlastní strukturu reprezentující hledaný vzor (`Pattern`). Každé vlákno bude vlastnit lokálně svůj `Matcher`, `Pattern` a svou tabulku výsledků. Všechna vlákna budou sdílet thread-safe objekt, který přiděluje části vrcholů grafu (`VertexDistributor`). Vlákno vždy zažádá `VertexDistributor` o určitý počet vrcholů, ze kterých spustí lokálně prohledávání a výsledky uloží do své tabulky. Nikdy nenastave situace, kdy dva vlákna mají stejný startovní vrchol. Po vyčerpání všech vrcholů grafu prohledávání končí. V Single-thread řešení jsou přiděleny všechny vrcholy najednou danému vláknu. Rozložení objektů mezi vlákny je zobrazeno na obrázku 2.4.

`VertexDistributor` je zde velice důležitý. Musí rozdělovat malé části vrcholů. Kdyby rozděloval velké části vrcholů může se stát, že některá vlákna budou mít mnohem více práce. Je to protože reálné grafy nemají obecně rovnoměrné rozložení hran. Jedno vlákno by mohlo dostat vrholy nacházející se v oblasti s množstvím hran, zatímco jiné vlákno by procházelo řídkou oblastí. Jelikož se rychle vyčerpaly startovní vrcholy, tak vlákno z řídké oblasti ukončí svou práci mnohem dříve něž vlákno první. Nyní se musí čekat na dokončení práce prvního vlákna.

2.5.5 Merge výsledků hledání

Po dokončení hledání je nutno vyřešit mergování výsledků jednotlivých vláken. Kdybychom ponechali výsledky bez úpravy, tak nedokážeme rovnoměrně rozdělit práci mezi vlákna v paralelních řešeních Order/Group by. Cílem je vytvořit jednu tabulku obsahující všechny výsledky. K vyhnutí překopírovávání všech výsledků vláken využijeme následující princip. Sloupeček tabulky bude tvořen polem polí fixní délky. V jazyce C# `List<Element[FixedArraySize]>`. V kroku mergování

nyní pouze překopírujeme odkazy na pole místo samotných výsledků. Avšak, pořád nám zůstává nutnost překopírovat výsledky posledních polí sloupečků, která jsou nezaplněná. Volbou vhodné hodnoty `FixedArraySize` se bude překopírovávat pouze malé množství výsledků. Konkrétní volba hodnoty je heuristická a vyplývá z vlastností grafu a počtu nalezených výsledků. Pro naše učely během implementace zkusíme zvolit prvně $n/\log_2 n$ ($n = \#$výsledků hledání) pro single-thread a pro paralelní zpracování $(n/\log_2 n)/\#threads$. $log_2 n$ odpovídá počtu přealokování při plnění dynamického pole n položkami. Mergování bude probíhat opět paralelně. Nabízí se dva způsoby. Vlákno merguje pouze jeden sloupeček nebo dojde k dvoucestnému mergi výsledků vláken. Nyní je obtížné odhadnou správné řešení a proto jej ponecháme na dobu implementace.



Obrázek 2.4: Diagram paralelizace prohledávání grafu.

Zbývá nám analyzovat a navrhnout Group by a Order by. Je nutné si uvědomit potřebu analýzy daných částí. Analýza nám pomůže pochopit jejich fungování, což nám umožní je lépe porovnat s vylepšenými řešeními. Zároveň také tvoří odrazový můstek pro návrh vylepšených řešení.

2.6 Order by

Order by si klade za cíl setřídit vyhledané výsledky z části Match pomocí zadaných klíčů. Pořadí klíčů určuje pořadí porovnání. Výsledky se porovnávají zleva doprava. To znamená, pokud jsou dva klíče stejné, postoupí se k porovnání s klíčem dalším. Rovnost dvou výsledků nastává právě tehdy, když mají stejné hodnoty pro všechny klíče. Pro klíč se také definuje, jestli má třídění probíhat

v rostoucím nebo klesajícím pořadí. Defaultní pořadí je chápáno jako rostoucí. Potřebujeme určit, jakým způsobem budou výsledky hledání setříděny. Musíme vybrat algoritmus a následně navrhnout způsob efektivního třídění tabulky výsledků.

2.6.1 Výběr algoritmů a paralelizace

Existuje mnoho algoritmů pro třídění. Chtěli bychom zvolit již prozkoumané a zároveň běžně používané třídící algoritmy. Mezi náš výběr padli Merge sort nebo Quick sort. Jsou ideální volba, protože pro ně již existuje paralelní verze. Při implementaci chceme ideálně použít již existující knihovny nebo implementace.

2.6.2 Quick sort vs Merge sort

Uvedeme krátké porovnání na základě 3. kapitoly průvodce algoritmů (Mareš a Valla, 2017). Merge sort má časovou složitost $\Theta(n \log n)$ i v nejhorším případě, zatímco Quick sort stejné složitosti dociluje pouze v průměrném případě. V nejhorším případě má $\Theta(n^2)$. Merge sort potřebuje $\Theta(n)$ pomocné paměti a je to stabilní třídící algoritmus. Quick sort pouze $\Theta(\log n)$, ale nejedná se stabilní třídění. K implementaci stabilního Quick sortu je potřeba $\Theta(n)$ pomocné paměti. Quick sort značně závisí na výběru pivota. V našem případě bude obtížné ho volit správně, protože nedokážeme říci nic o rozložení tříděných dat. Z tohoto důvodu bychom volili raději Merge sort.

2.6.3 Třídění pomocí indexů

Hlavním problémem Order by je třídění tabulky výsledků. V sekci 2.4.2 jsme definovali formát výsledků a navrhli proxy třídy pro výpočet výrazů. Setřídit tabulku v našem smyslu znamená setřídit řádky pomocí klíčů zadaných uživatelem. Pro zjištění shodnosti dvou řádků musíme vypočítat hodnoty jejich klíčů pomocí výrazů a následně je porovnat. Přesouvání řádků v tabulce, která má více než jeden sloupec, by představovalo značné zpomalení. Uvedli jsme, že výrazy pro řádky dostanou na vstupu proxy třídu řádků. Proxy třída se v naší představě získá voláním funkce indexeru na tabulce výsledků. Daný princip nám umožní vytvořit pole indexů v rozsahu počtu řádek tabulky. Indexy budou setříděny namísto pravých řádků vybraným algoritmem a při porovnání dojde k získání proxy tříd a výpočtu výrazů. Přistup nám umožní vyřadit přesouvání řádků, ale zase potřebujeme lineární pamět na pole indexů. Po dokončení třídění se pole použije jako indexační struktura.

2.6.4 Optimalizace porovnání Property hodnot

Třídění obvykle představuje opakované porovnání jednoho prvku s ostatními v řadě za sebou. Pro pokaždé takové porovnání v řadě počítáme stejnou hodnotu výrazu opakovaně. Porovnání pomocí `ID` pouze přistupuje k položce elementu grafu. Problém nastane, když budeme porovnávat pomocí Property. V takovém případě musíme přistoupit k tabulce `elType` (sekce 2.2), zjistit existenci přistupované Property a následně přistoupit k její hodnotě. Danou situaci můžeme vyřešit částečně cachování výsledků výrazů. Budeme si pamatovat poslední porovnané

řádky a jejich hodnoty. Pokud dojde k porovnání řádků pro který byl výraz již vypočítán, tak použijeme zachovanou hodnotu.

2.6.5 Optimalizace porovnání stejných elementů

Další možná optimalizace porovnání může nastat v případě, pokud pro použitý graf platí #Vrcholů << #Hran. Daná vlastnost může mít za následek, že porovnávané řádky budou často obsahovat stejné elementy pro dotazy typu:

```
select x.PropOne match (x) -> (y) order by x.PropOne;
```

V takovém dotazu by hledání mělo vygenerovat několik výsledků se stejným elementem v proměnné x. Počet takových výsledků zde odpovídá počtu hran vrcholů x. Abychom předešli opakovanému porovnávání hodnot Property stejných elementů, tak než přistoupíme k výpočtu výrazů porovnáme `ID` přistupovaných elementů. Tedy využijeme dvou optimalizací. Budeme si cachovat poslední výsledky výrazů a navíc omezíme porovnání výsledků se stejnými elementy.

2.6.6 Optimalizace v paralelním prostření

Vymysleli jsme optimalizace porovnání. Doteď jsme však předpokládali, že porovnání probíhá v jednom vlákně. Druhá optimalizace funguje i v paralelním prostřední, protože se jedná pouze o čtení statických hodnot. Avšak první optimalizace vytváří problém. V prvním případě dochází k uchovávání výsledků lokálních pro vlákno a existence sdíleného úložiště by vytvořilo race condition. Vlákna by se snažila číst a ukládat výsledky ze sdíleného úložiště a docházelo by k nedefinovanému chování. Problém se dá vyřešit například tak, že každé vlákno bude vlastnit svoje objekty s cachí výsledků. V průběhu implementace budeme muset najít vhodnou techniku ukládání, aby došlo ke zrychlení třídění.

2.7 Group by

Group by seskupuje výsledky hledání podle uživatelem zadaných klíčů. Dva výslekdy patří do stejné skupiny právě tehdy, když se shodují ve všech hodnotách klíčů. Musíme být schopni vypočítat agregační funkce pro skupiny. K tomu již máme navržené způsoby z sekce Expressions (2.3.3). Zbývá nám vymyslet způsob ukládání mezivýsledku a algoritmy k vykonání.

2.7.1 Módy Group by

Group by představuje dle našeho pohledu dva módy vykonání. První mód obsahuje v dotazu část Group by a libovolné množství agregačních funkcí. Dojde k vytvoření skupin a výpočtu výsledků funkcí pro každou skupinu. Tento mód budeme označovat **Group by**. Druhý mód nemá v dotazu část Group by, ale pouze agregační funkce v ostatních částech. Zde automaticky dochází k předpokladu, že všechny výsledky patří do stejné skupiny. Tedy je vytvořena pouze jedna skupina a pro ni se vypočtou hodnoty agregačních funkcí. Mód nazveme **Single group Group by**.

2.7.2 Úložiště mezivýsledků agregačních funkcí

V sekci Expressions (2.3.3) jsme navrhli objekt, který obsahuje logiku výpočtu funkce a na vstupu dostává úložiště výsledku. Očekává se, že pro každou skupinu bude existovat úložiště. Úložiště musí obsahovat prostor pro výsledky všech počítaných funkcí. Vymysleli jsme dva způsoby:

- **Bucket** - každá skupina bude vlastnit pole objektů. Pole bude mít délku rovnou počtu počítaných agregačních funkcí. Objekty představují úložiště výsledků funkcí.
- **List** - výsledky všech skupin jsou uchovávány v dvoudimenzionálním poli, tj. primitivní tabulce. Sloupeček představuje výsledky jedné agregační funkce. Jedné skupině pak přináleží řádek. Každé skipině bude přidělen index řádku. Nyní pokud budeme chtít výsledky funkcí skupiny, stačí přistoupit skrze přidělený index ke chtěnému výsledku.

Následuje ukázka možné implementace v jazyce C#:

```
Bucket:
BucketResult[] groupAggFuncResults;
class BucketResult {}
class BucketResult<T>: BucketResult { T value; }

List:
ListResults groupsAggFuncResults;
class ListResults { ListHolder[] holders; }
class ListHolder {}
class ListHolder<T> : ListHolder { List<T> values }
```

První způsob je náročnější na paměť oproti druhému způsobu, neboť je zde nutnost vytvářet objekty a pole pro každou skipinu. Avšak, v druhém způsobu jsme nuceni přistupovat k výsledkům pomocí indirekce. Výhoda Bucket spočívá v jeho jednoduchém přemisťování (chápeme-li pole jako referenci) a izolaci od ostatních výsledků skupin. V takovém případě jsme schopni přesouvat výsledky skupiny aniž bychom museli kopírovat jejich hodnoty. Předpokládáme, že bucket bude výhodnější pro paralelní zpracování díky izolaci od ostatních výsledků, jelikož počet skupin není dopředu znám. V List budeme muset dynamicky rozšiřovat pole. Při rozšíření nastane race condition, který budeme muset ošetřit.

2.7.3 Logika agregačních funkcí

Ještě než přistoupíme k výběru zpracování musíme analyzovat logiku agregačních funkcí. Logika pak ovlivňuje co a jak se bude ukládat v úložišti výsledků.

- `Min/Max`: výsledek funkce je minimum/maximum pro skupinu. Při výpočtu dojde k porovnání a následnému uložení hodnoty. V objektu výsledku musíme znát aktuální minimum/maximum a jestli byla hodnota již nastavena.
- `Count/Sum`: výsledek je počet výsledků/suma výsledků. Při výpočtu musíme přičítat hodnotu k hodnotě v úložišti.

- `Avg`: výsledek je aritmetický průměr. Chtěli bychom volit inkrementální metodu. Budeme si ukládat počet výsledků a sumu hodnot. Jako finální krok dojde k vypočtení podílu hodnot.

2.7.4 Single thread zpracování

Po celou dobude budeme pracovat s tabulkou výsledků Match části. Pro vykonání Group by se nám nabízí několik možností. První možnost je řádky tabulky setřídit a následně při iteraci vytvářet skupiny a počítat agregační funkce. Myslíme, že možnost přináší zbytečný overhead za porovnání při třídění, protože pro každé porovnání musíme vypočítat hodnotu výrazu. Z tohoto důvodu chceme využít strukturu, která bude interně používat hashovací tabulku (C++ `std::map<Key, Value>` nebo C# `Dictionary<Key, Value>`). Záznam v tabulce bude dvojice `key/value`. `key` je zde index do tabulky výsledků z Match části (nikoliv proxy třída). Chceme ukládat pouze index abychom ušetřili paměť za pointr na tabulku. Porovnání vyvolá získání proxy třídy a následné evaluaci hodnot výrazů. `value` obsahuje strukturu pro výsledky agregačních funkcí. Pro Bucket to bude pole objektů a pro List to bude index do tabulky výsledků agregačních funkcí. Pro výsledek bude vypočítáná hash na základě hodnot klíču a vložena do hashovací tabulky. Pokud už je obsažen v tabulce, tak se pouze pro získanou `value` (pole nebo index) aktualizují hodnoty výsledků agregačních funkcí. V opačném případě bude vložen nový záznam s novou `value`. Pro Single group Group by stačí pouze iterovat skrze tabulku a počítat agregační funkce.

2.7.5 Optimalizace při výpočtu hash hodnoty

Minulý přístup nám nabízí jednu optimalizaci k ušetření opakovaného výpočtu hodnoty výrazu klíčů. Hashovací tabulka při vložení dvojice v prvním kroku vypočte hodnoty klíčů a vypočte jejich hash. Výsledek se vloží do patřičné přihrádky. Pokud nastala kolize, tak se prvky musí porovnat. Při tomto porovnání se opět musí vypočíst hodnoty výrazů vkládané dvojice. Zde mužeme využít chvíle výpočtu hash hodnoty. Budeme cachovat hodnoty výrazů a následně je znovu použít při porovnání. Nicméně, pravděpodobně budeme nutni vytvořit závislost mezi objektem počítajícím hash hodnotu a objektem provádějícím porovnání. Stejný princip jsme použili při optimalizaci porovnání v sekci 2.6.4. Nastávají pro něj stejné problémy v paralelním prostředí.

2.7.6 Paralelní zpracování

Pro paralelní zpracování jsme vymysleli několik přístupů: **Global**, **Two-step** a **Local + 2-way merge** (všechny budou schopny pracovat s úložišti Bucket i List). Chceme zvolit několik řešení, abychom byli schopni lépe porovnat vylepšená řešení. Každý z nich začíná rozdělením tabulky výsledků Match na ekvivalentní části. To si můžeme dovolit, neboť máme tabulku obsahující všechny výsledky hledání. Každé vlákno dostane danou část ke zpracování. Tím zaručíme rovnoměrnost práce mezi vlákny. Samotná práce vláken pak zavisí na zvoleném přístupu.

2.7.7 Thread-safe agregační funkce

Ještě než přistoupíme k řešením, je důležité si uvědomit nutnost vytvořit thread-safe verzi objektů implementující logiku agregačních funkcí. Pro každou funkci bude existovat ekvivalent thread-safe `Apply` metody. Samotně pak vyvstává nutnost implementovat thread-safe metody pro mergování výsledků agregačních funkcí. Problém vyřešeníme přidáním metod do objektů zpravující logiku agregačních funkcí, protože při mergi stále musíme vědět, jak s výsledky správně zacházet. Pro úpravu logiky agr. funkcí na thread-safe verze volíme tyto varianty:

- `Min/Max:` princip Compare and Exchange. V moment porovnání mohlo dojít ke změně hodnoty v úložišti. Musíme znovu provést porovnání.
- `Sum/Count:` tyto metody implementují pouze přičítání. V ideálním případě chceme použít atomické operace přičtění.
- `Avg:` tuto funkci lze implementovat iterativně. Budeme si ukládat součet zpracovaných výsledků a jejich počet. Finální hodnota pak bude podíl součtu a počtu hodnot. Tímto můžeme implementovat thread-safe funkci pomocí atomických operací, jako u funkcí `Sum/Count`.

2.7.8 Global Group by

Všechna vlákna budou provádět ekvivalent single-thread seskupování pomocí thread-safe paralelní mapy/slovníku. Skupiny se vytváření globálně. Všimněme si nutnosti dvojité synchronizace. Prvě musí dojít k synchronizaci vytváření záznamů v mapě. Druhý krok synchronizace je nutný při zpracování agregačních funkcí. Paralelní mapa nám vyřadí race-condition při vytváření nových záznamů skupin. V moment zpracování agregačních funkcí musí dojít k volání thread-safe verzí. Výhoda této metody je, že nedochází k mergi, tj. po dokončení práce vlákny jsou výsledky finální.

Bucket reprezentace úložiště má zde značnou výhodu vůči List. List musí dynamicky rozšiřovat tabulku svých výsledků. Místo abychom použili dvojí synchronizace jako u Bucket, tak zde bude nutné představit ještě třetí krok synchronizace. Při přístupu do tabulky je nutné kontrolovat, jestli se nemusí rozšířit. V ten moment se musí zabránit přístupu ostatních vláken a teprve pak rozšířit tabulku. To se dá implementovat například semaforem omezující pohyb vláken v kritické sekci. Vlákno se na přístup pokusí projít skrze semafor, pokud je mu vstup zabráněn tak dochází k rozšiřování tabulky. Vlákno v moment nutnosti rozšíření zablokuje vstup přes semafor. Postup s List může být však značně pomalý a proto budeme uvažovat, jestli přístup raději implementovat pouze pro Bucket. Global Group by je znázorněn na obrázku 2.5.

2.7.9 Two-step Group by

Tento přístup probíhá ve dvou krocích. V prvník kroku pro každé vlákno běží kompletně identický ekvivalent single thread řešení. Druhý krok nastave v moment dokončení této práce. Místo aby vlákno ukončilo běh, tak provede merge svých výsledků do thread-safe paralelní mapy. Tento přístup kombinuje single-thread řešení společně s přístupem Global. V druhém kroku platí vše, co jsme

zmínili u Global řešení. Mínus je, že zde musí docházet mergování, ale v prvním kroku se využívají metody, které nemusí řešit synchoronizaci. Problematická část je zde mergování výsledků s List úložištěm výsledků agregačních funkcí. První fáze nepředstavuje problém. Druhá představuje problém jako u Global řešení. Můžeme zkusit implementovat stejné řešení jako u Global pomocí semaforu. Two-step Group by je znázorněn na obrázku 2.6.

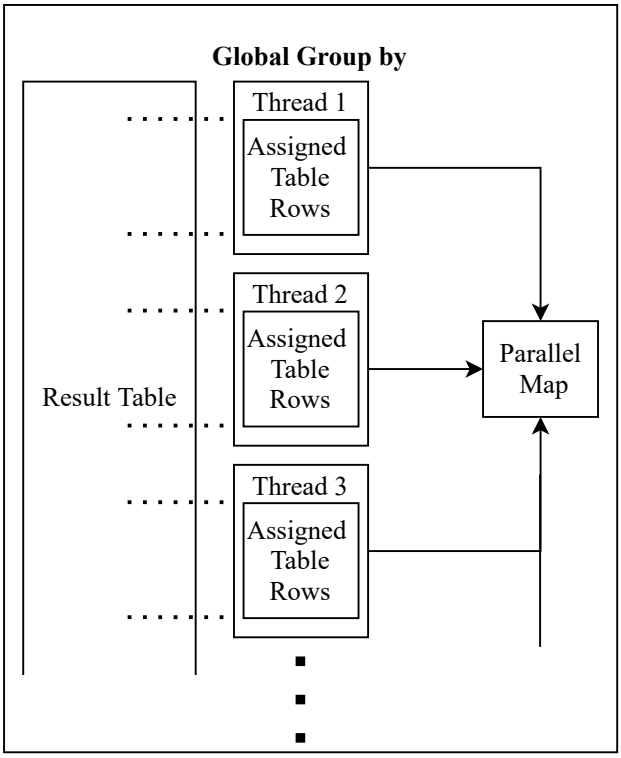
2.7.10 Local + 2-way merge Group by

Přístup nepoužívá thread-safe metody ani paralelní mapu. Opět rozdělíme přístup na dvě části. V prvník kroku pro každé vlákno běží identický ekvivalent single thread řešení. Po dokončení se spustí dvoucestný merge na výsledky vláken. Mergování bude připomínat binární strom. Listy představují lokální seskupování vláken. Vnitřní vrcholy stromu představují mergování výsledků. Finální výsledek je vytvořen v kořeni. U mergování využijeme paralelního zpracování. Při mergi vždy jedno vlákno ukončí běh, druhé z dvojice provede merge a postoupí ke kořeni. Hlavní výhody byly už zmíněny. Používají se metody bez synchronizace. Přístup je znázorněn na obrázku 2.7.

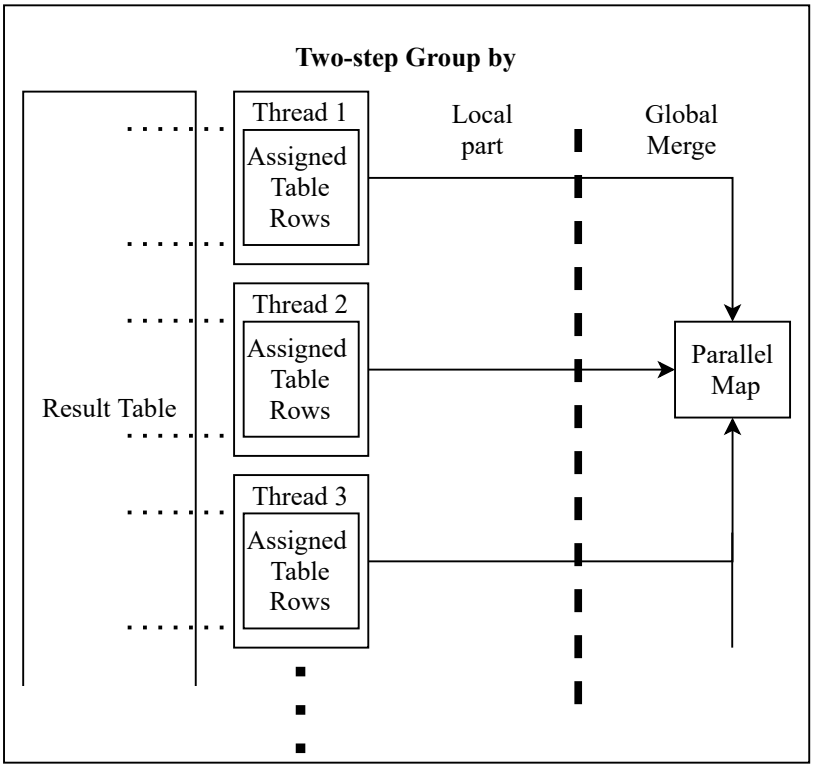
2.7.11 Paralelizace Single group Group by

K paralelizaci Single group Group by se nám poskytuje využít jíž zmíněných principů. Konkrétně zde mají využití všechny tři. Mergování všech výsledků do jednoho úložiště výsledků agregačních funkcí z principu Global. Problémem tohoto přístupu je synchronizace mnoha vláken na jednom místě. Ideálnější je využít lokálního zpracování zbylých dvou metod. Opět řešení rozdělíme na dva kroky. V prvním kroku každé vlákno provádí ekvivalent single thread řešení bez vytváření skupin, tj. počítá pouze agregační funkce pro jednu skupinu. Výsledky se následně musí mergovat. K rozhodnutí, které řešení finálně použít použijeme fakt, že počet výsledků k mergování je roven počtu vláken. V takovém případě nebudeme implementovat paralelní mergování, ale pouze jedno vybrané vlákno provede sjednocení všech výsledků.

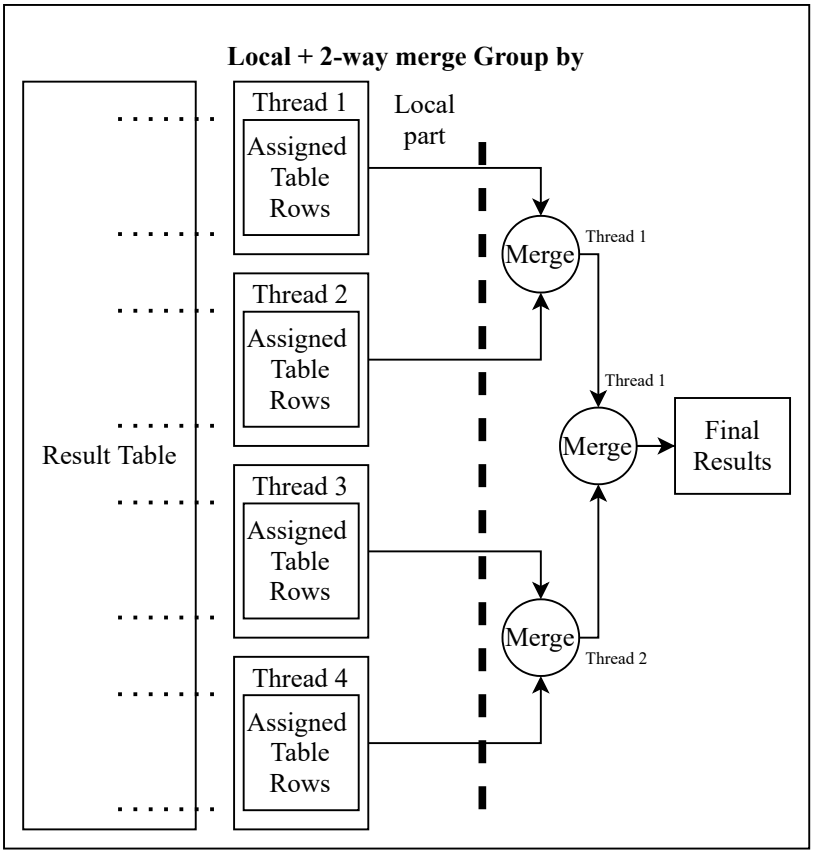
Analyzovali jsme a navrhli řešení vykonání částí Match, Group by a Order by. Daná analýza a návrh nám poskytnou odrazový můstek při analýze úprav pro vykonání částí Group by a Order by v průběhu hledání.



Obrázek 2.5: Diagram paralelizace Global Group by.



Obrázek 2.6: Diagram paralelizace Two-step Group by. Local part provádí ekvivalent single thread řešení. V Merge části dochází k mergování výsledků vláken pomocí paralelní mapy.



Obrázek 2.7: Diagram paralelizace Local + 2-way merge Group by pro 4 vlákna.

2.8 Vylepšení částí Group/Order by

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.

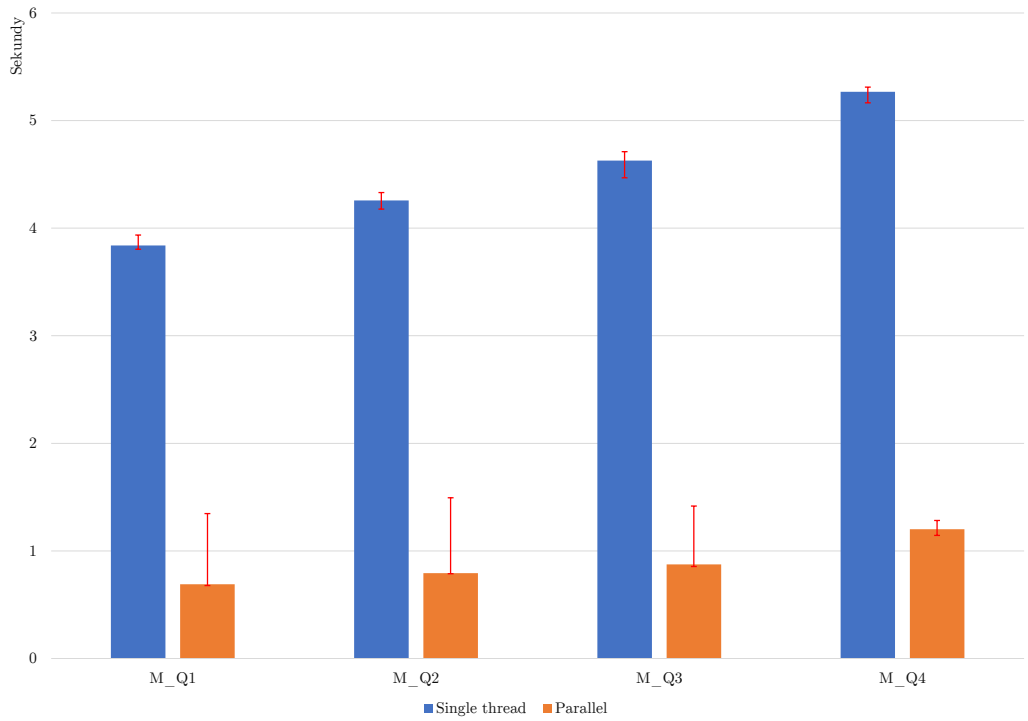
4.4 Výsledky

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

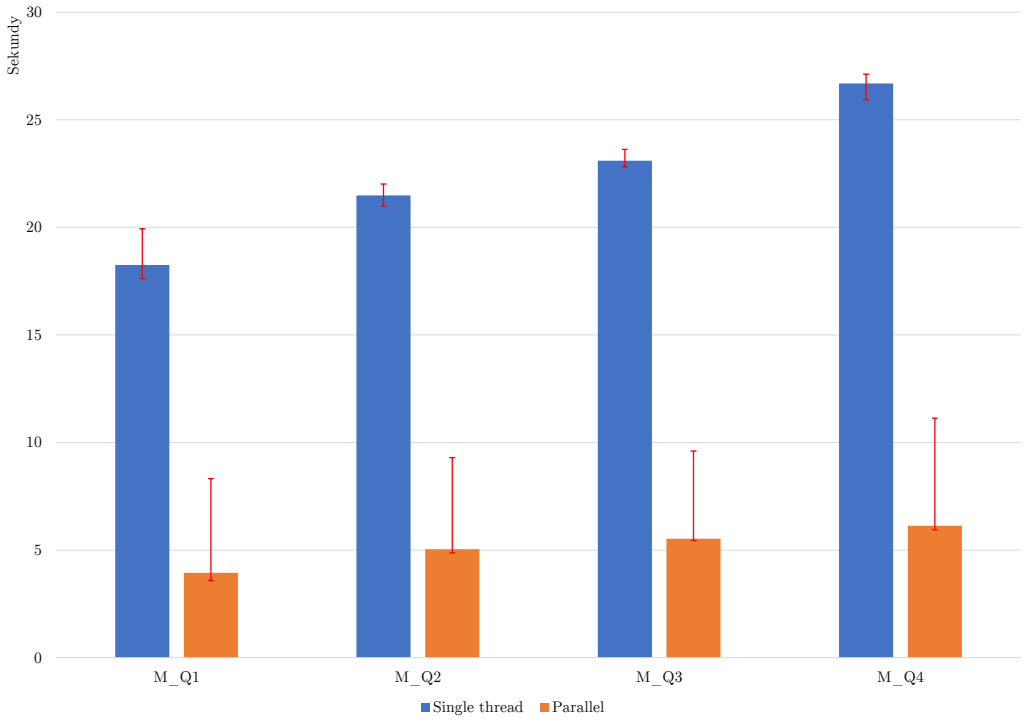
U grafů Group/Order by se držíme značení odpovídající z kapitoly implementace (tj. ve tvaru (mód enginu): (Název řešení) (způsob ukládání výsledků u Group by)?). Pokud řešení obsahuje kombinaci módů, pak řešení pro dané módy jsou totožná. Pro připomenutí zmíníme, že mód Normal vykonává Group/Order by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

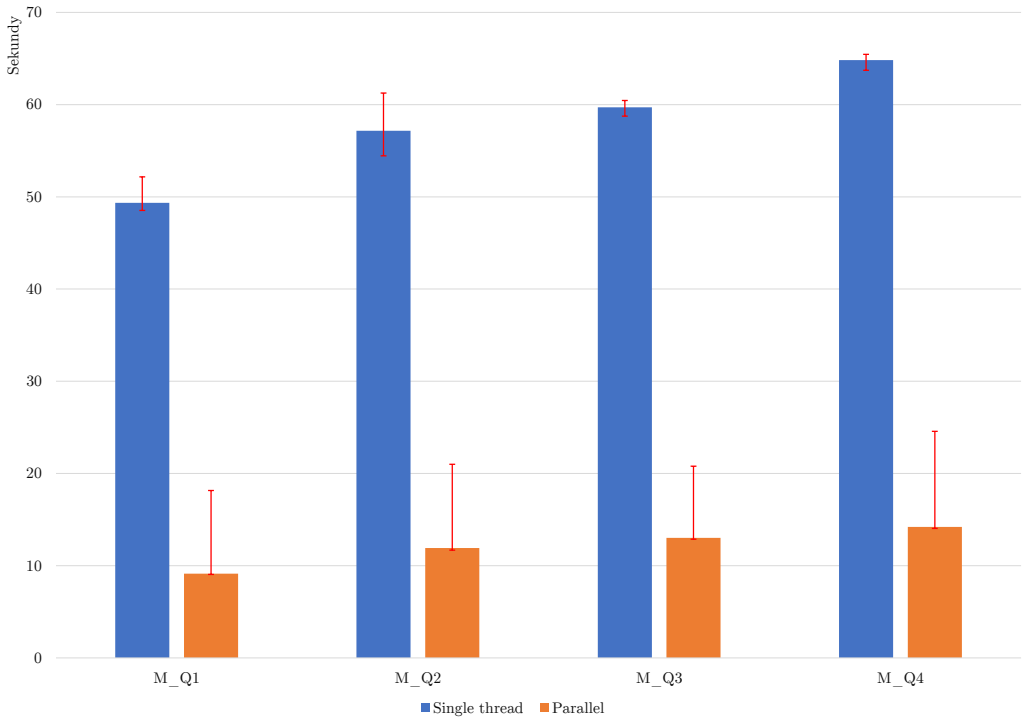
Stávající a vylepšené verze Group/Order by jsou značně ovlivněny vyhledáváním vzoru. Proto uvádíme výsledky a analýzu dotazů Match zvlášť, aby bylo možné sledovat čas výhradně strávený vyhledáváním a uložením všech nalezených výsledků do tabulky.



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

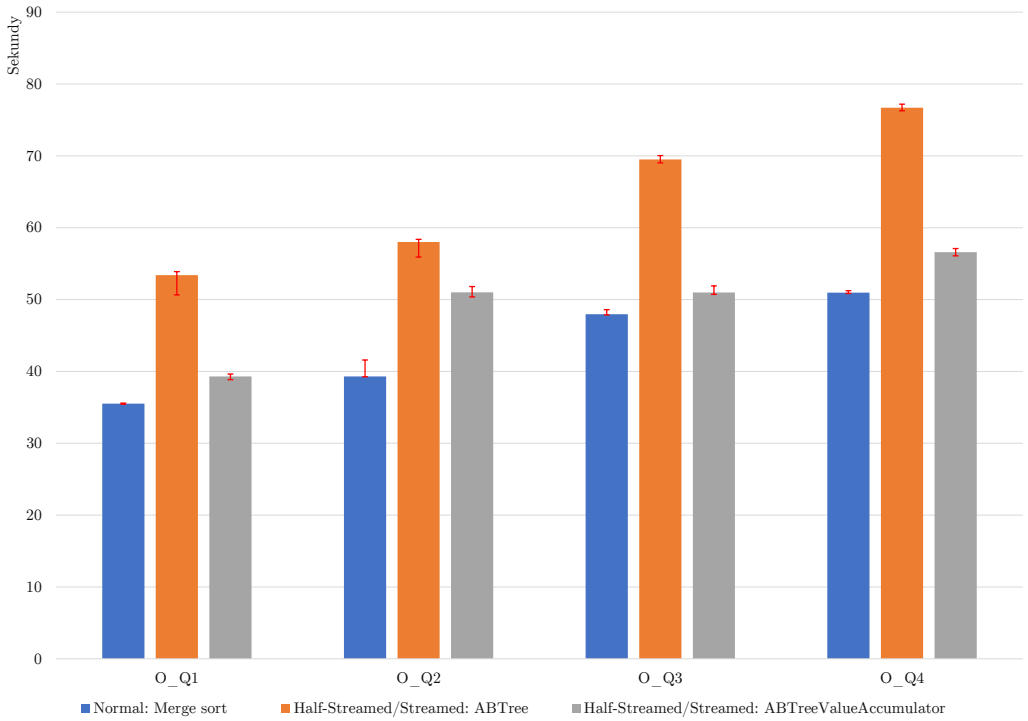
Z důvodu časové a prostorové složitosti třídění na grafu As-Skitter jsme se rozhodli jej vynechat pro Order by dotazy.

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].

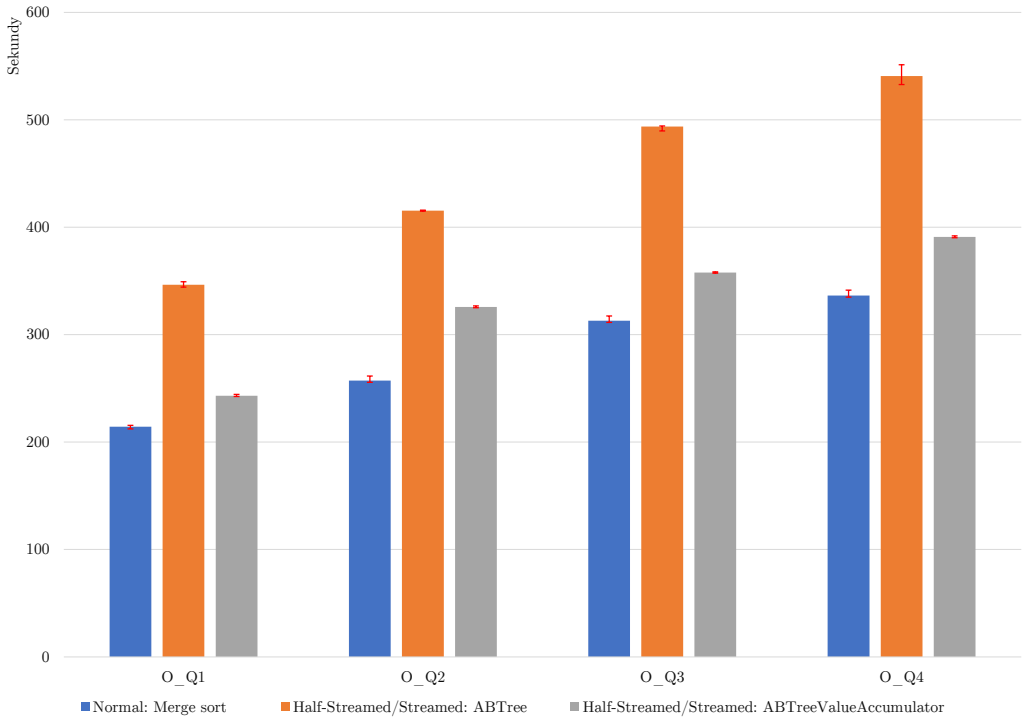
Začneme řešením běžícím v jednom vlákně, tj. obrázky 4.4 a 4.5. Můžeme si všimnout, že výsledky vypadají v rámci daných grafů konzistentně pro každý dotaz. Ani jedno z vylepšených řešení nedokázalo porazit mód Normal, což odpovídá našim předpokladům. Je to protože daný strom podléhá režii za `Insert` $\Theta(\log n \cdot (a/\log a))$ (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 Property existuje na daném elementu a následném čtení hodnoty ze struktury obsahující ji.

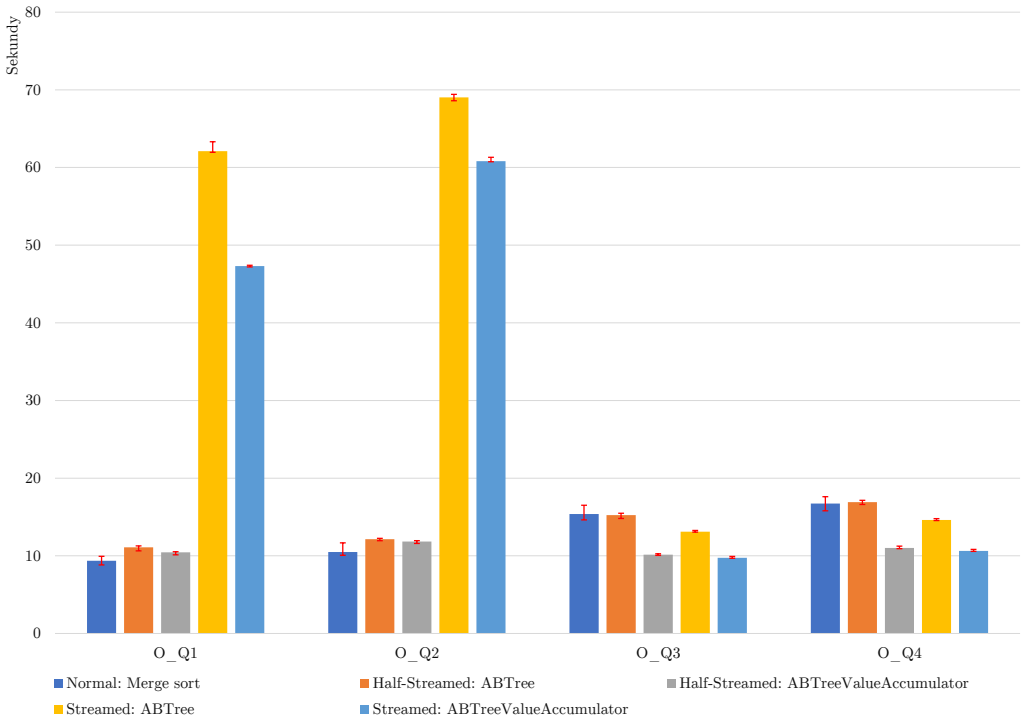
Paralelní zpracování aplikuje použité verze (a, b)-stromů ze zpracování pro jedno vlákno. Half-Streamed řešení obsahuje lokální tabulku a indexační strom pro každé běžící vlákno. Po dokončení vyhledávání se obsahy stromů překopírují do pole a dojde k paralelnímu 2-way merge používající stejnou funkci 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.

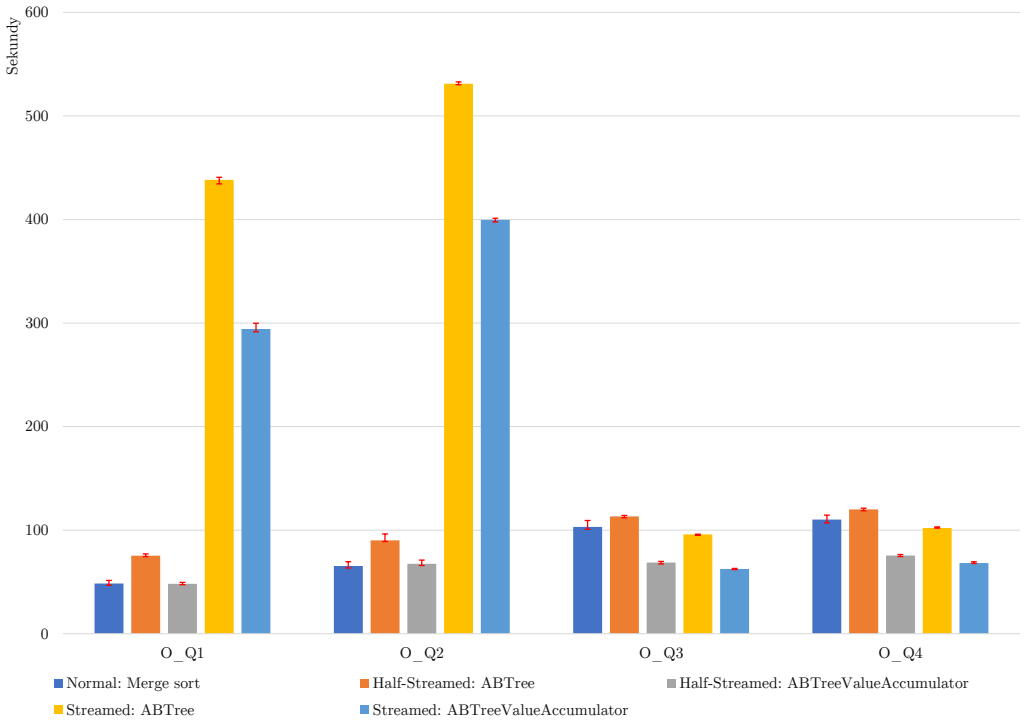
Half-Streamed řešení se přibližuje Normal řešení v prvních dvou dotazech a překonává jej ve třetím i čtvrtém dotazu pro řešení používající ABTreeValueAccumulator. U třetího a čtvrtého dotazu se porovnává pomocí Properties. V single thread zpracovávání jsme viděli overhead za dané porovnání. V druhém 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.



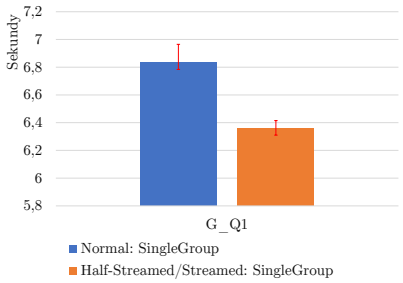
Obrázek 4.7: Doba vykonání dotazů Order by pro graf WebBerkStan (sekce 4.1). Běh osmi vláken. Počet tříděných výsledků je 222498869.

	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

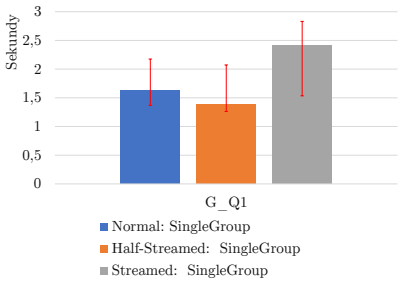
Tabulka 4.8: Rozsah zrychlení paralelizovaných řešení pomocí osmi vláken v ramci grafů pro dotazy Order by.

4.4.3 Group by

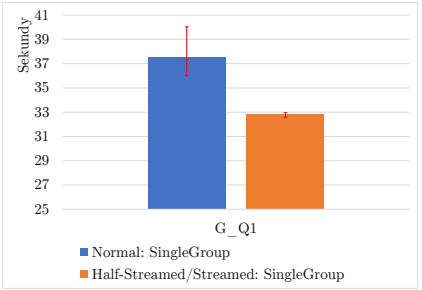
Analýzu výsledků dotazu G_Q1 uvádíme samostatně, protože testuje pouze agregační funkce a nikoliv seskupování. Po levé straně je běh v jednom vlákně a na pravé straně běh osmi vláken.



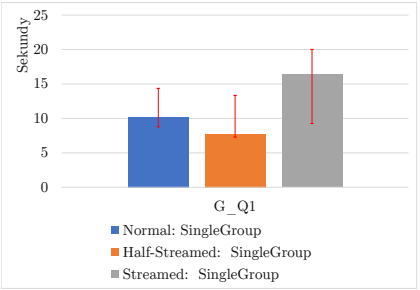
Obrázek 4.8: Doba vykonání dotazu G_Q1 pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 32373599.



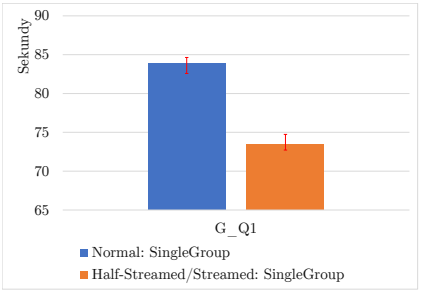
Obrázek 4.9: Doba vykonání dotazu G_Q1 pro graf Amazon0601 (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 32373599.



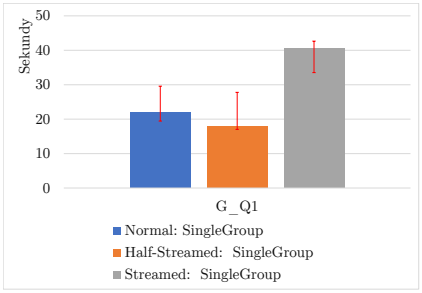
Obrázek 4.10: Doba vykonání dotazu G_Q1 pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 222498869.



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.

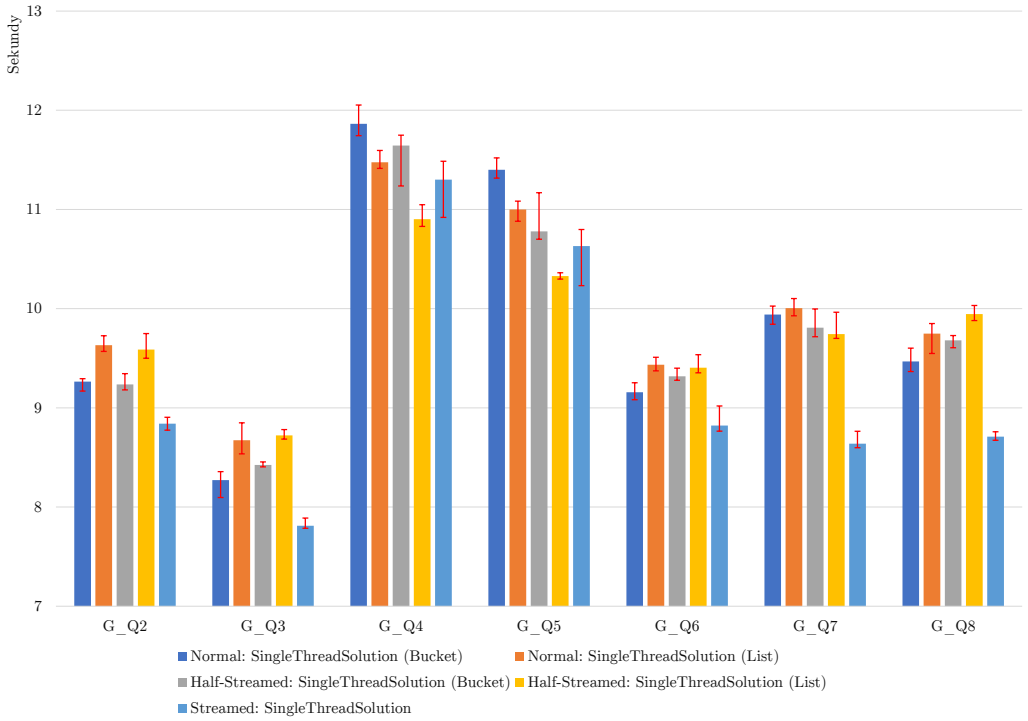


Obrázek 4.13: Doba vykonání dotazu G_Q1 pro graf As-Skitter (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 453674558.

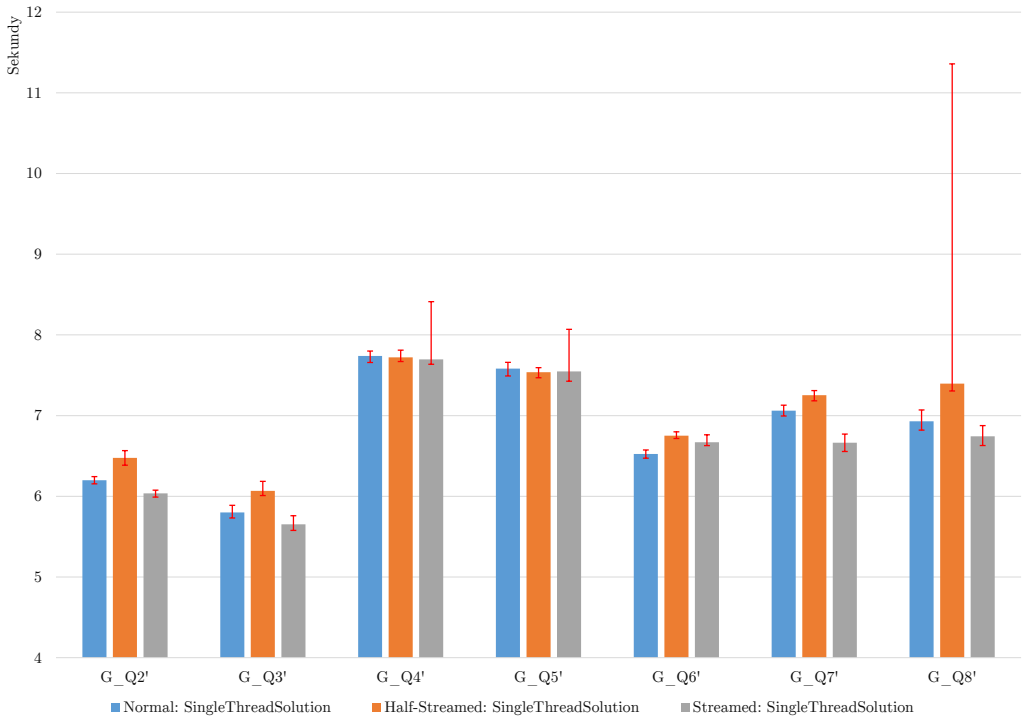
Na obrázcích 4.8 až 4.13 lze vidět značnou konzistenci mezi výsledky testování při nárustu počtu výsledků vyhledávání. Half-Streamed a Streamed řešení zde neukládá výsledky vyhledávání do tabulky, ale pouze na aktuální výsledek aplikuje agregační funkce a následně jej zahodí. To způsobuje značnou výhodu 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:

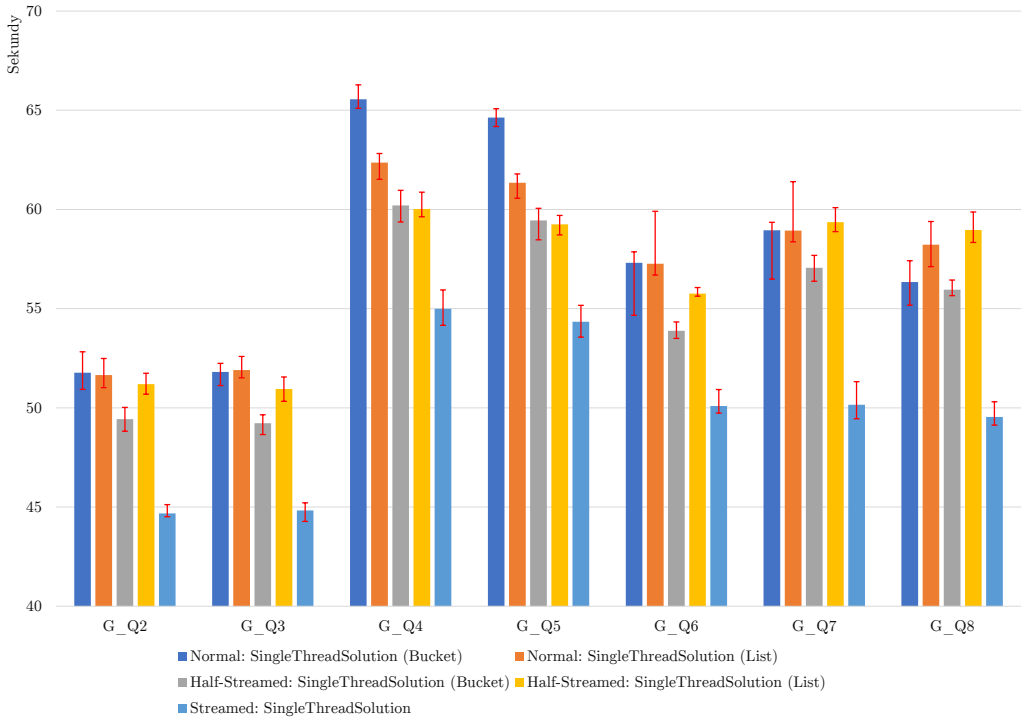
```
Bucket:
Dictionary<key, BucketResult[]> map; // Used map.
class BucketResult {}
class BucketResult<T>: BucketResult { T value; }

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 }
```



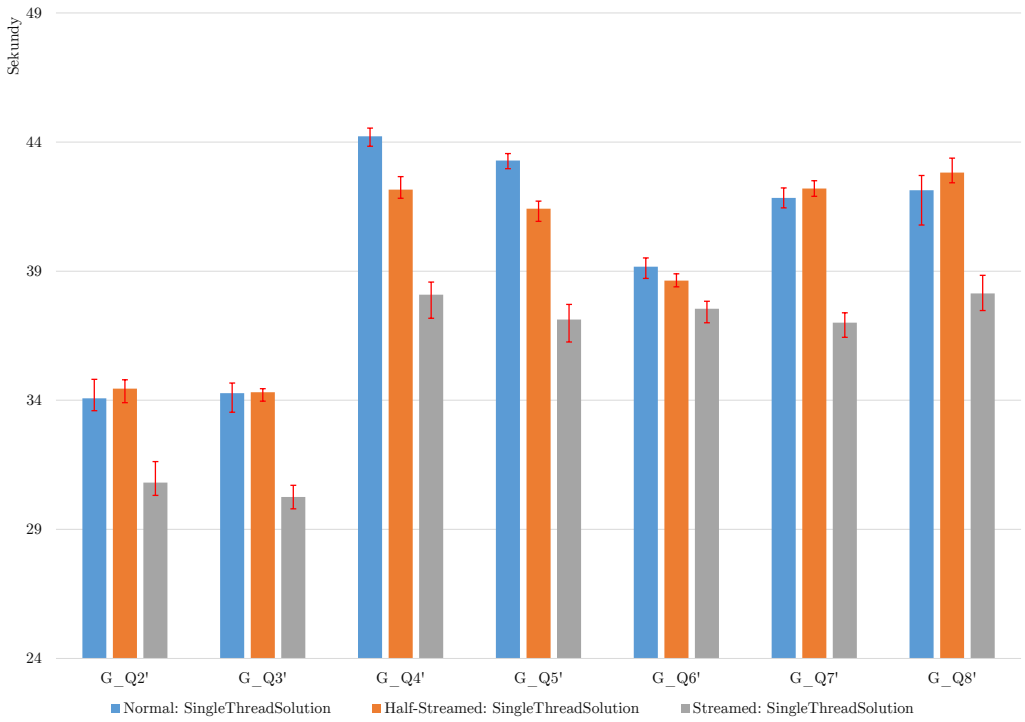
Obrázek 4.14: Doba vykonání dotazů Group by pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 32373599.



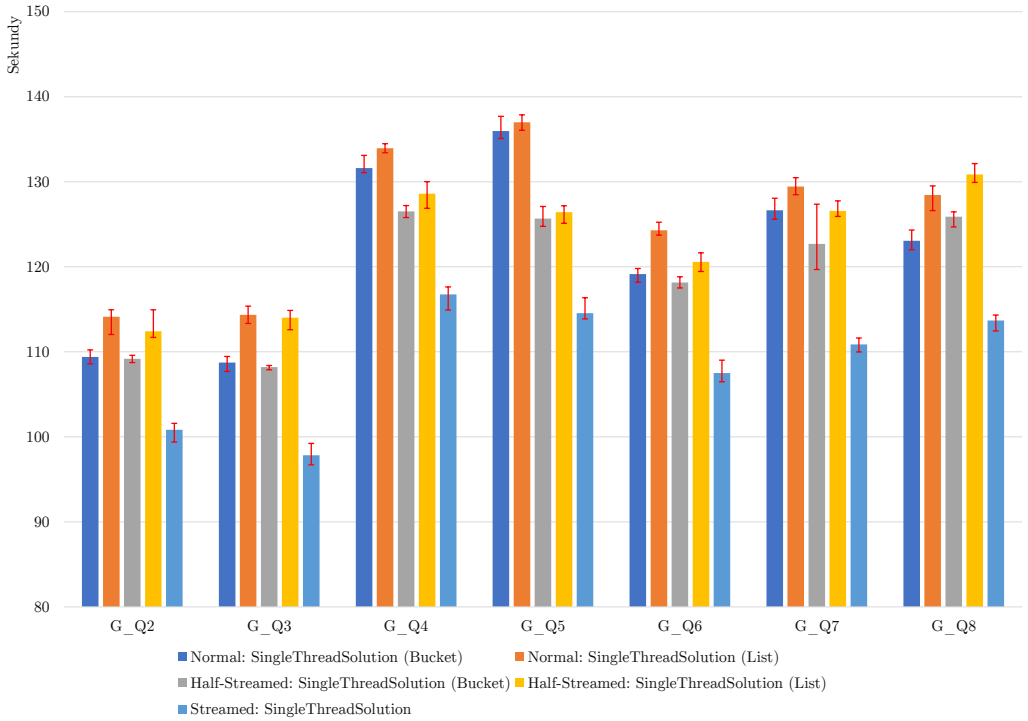
Obrázek 4.15: Doba vykonání dotazů Group by bez agr. funkcí pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 32373599.



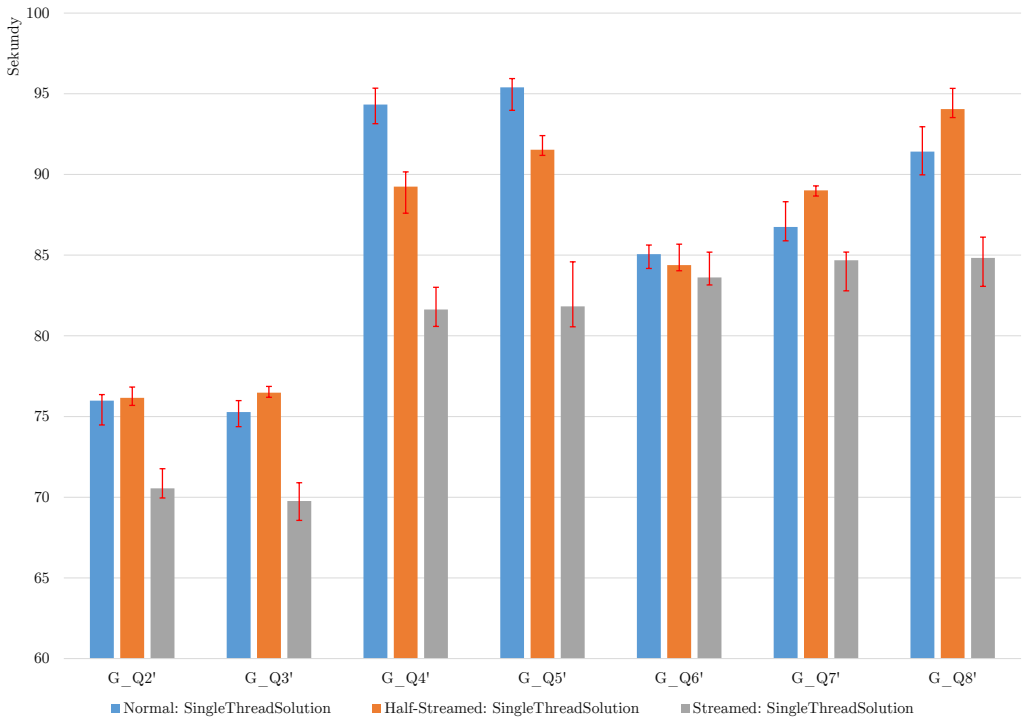
Obrázek 4.16: Doba vykonání dotazů Group by pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu. Počet seskupovaných výsledků je 222498869.



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.

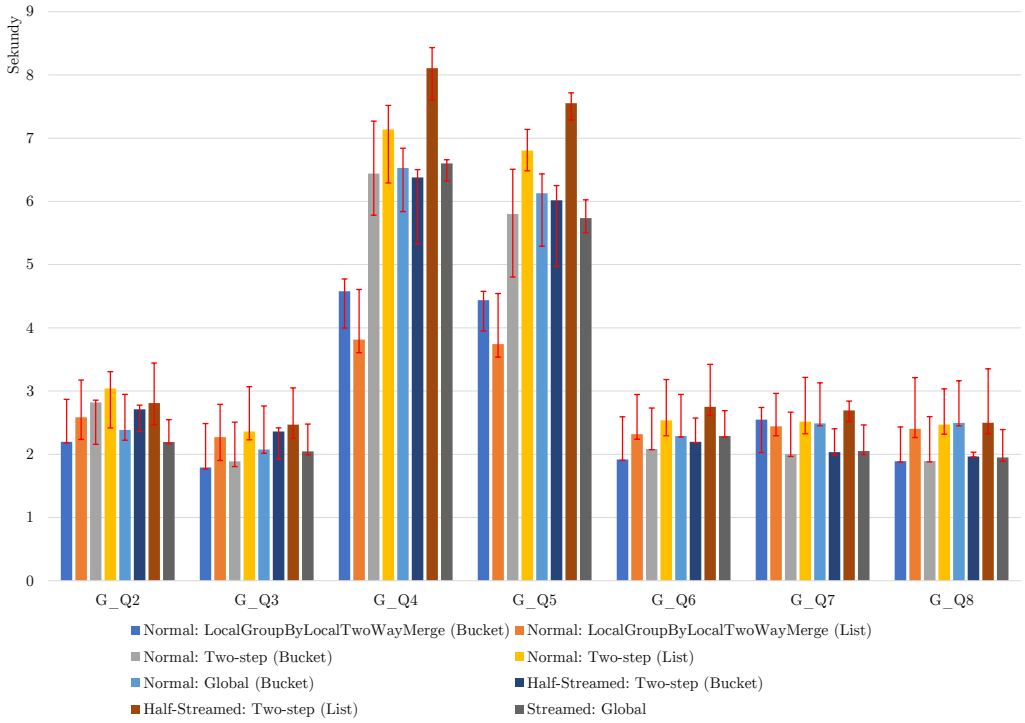
Na obrázcích 4.14, 4.16 a 4.18 vidíme výsledky Group by pro běh v jednom vlákně. Výsledky na obrázcích 4.15, 4.17 a 4.19 představují dotazy bez agregačních funkcí v části Select. Dvojice dotazů G_Q4/G_Q5, G_Q2/G_Q3 a G_Q6/G_Q7/G_Q8 jsou pouze mírně rozličná a můžeme u nich vidět konzistenci výsledků 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.

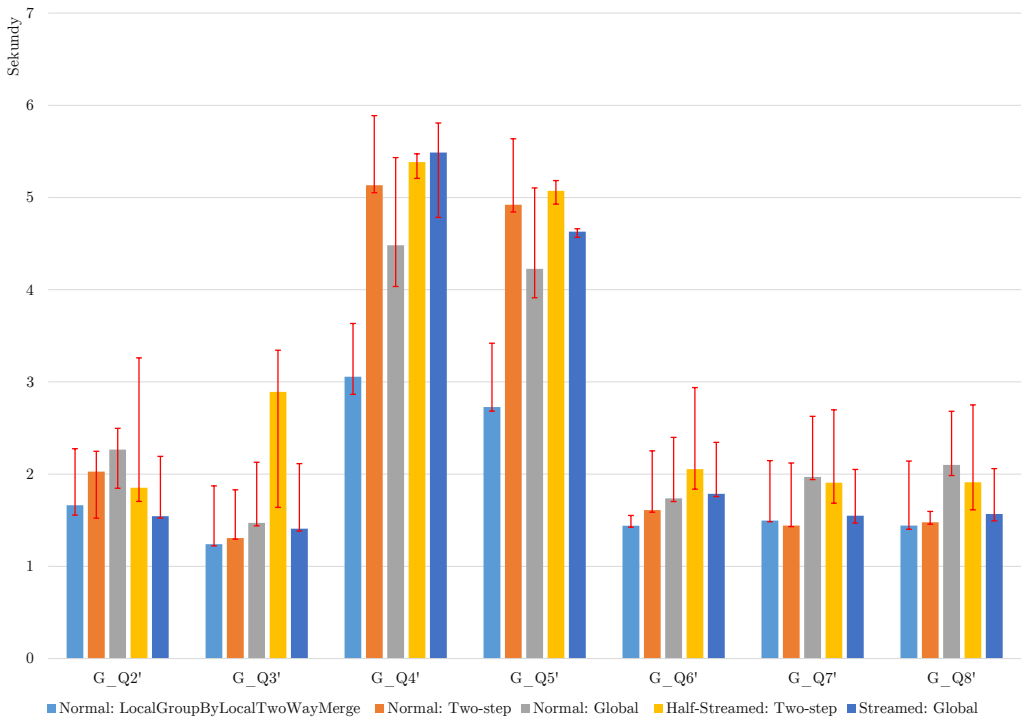
Na největším grafu platí, že použité ukládání List je pomalejší než Bucket, kvůli indirekci navíc. Na menších grafech rozdíly ustupují a dokonce nastavají situace, kdy je List rychlejší. Přesněji u dotazů G_Q4 a G_Q5. Přehození rolí si u nich vysvětlujeme overheadem za množství vytvářených polí (tj. 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.

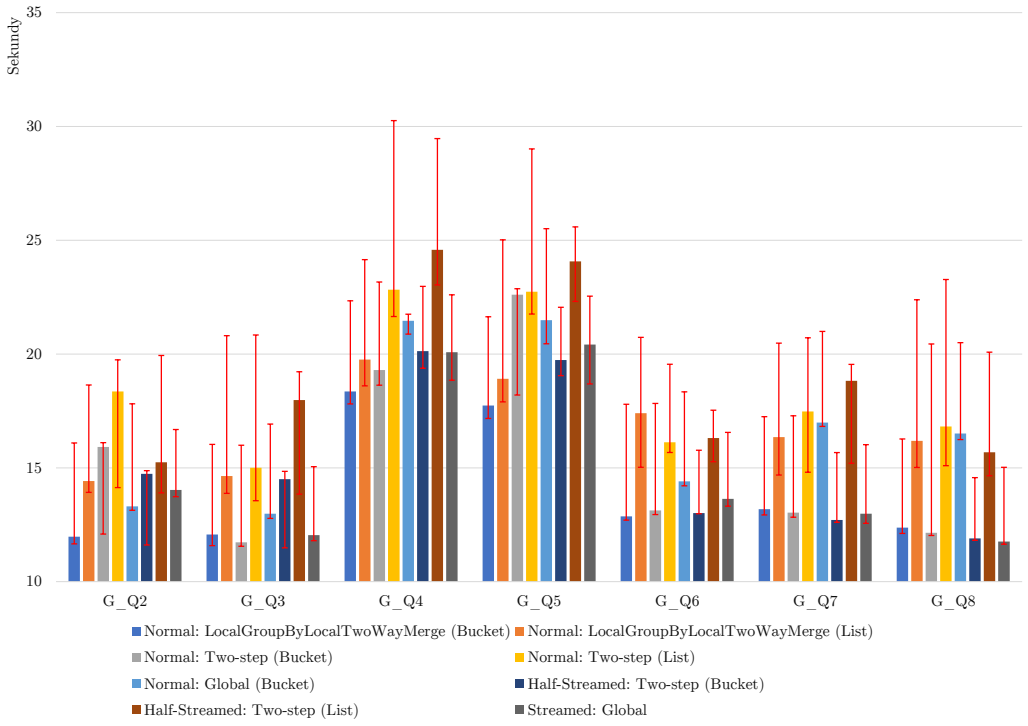
Paralelní řešení používají doposud zmíněná značení. Global řešení seskupuje výsledky globálně pomocí paralelní mapy (`ConcurrentDictionary`). Two-Step řešení seskupuje výsledky nejdříve lokálně pomocí mapy a následném mergi do paralelní mapy. LocalGroupByLocalTwoWayMerge řešení seskupuje lokálně a následně 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.



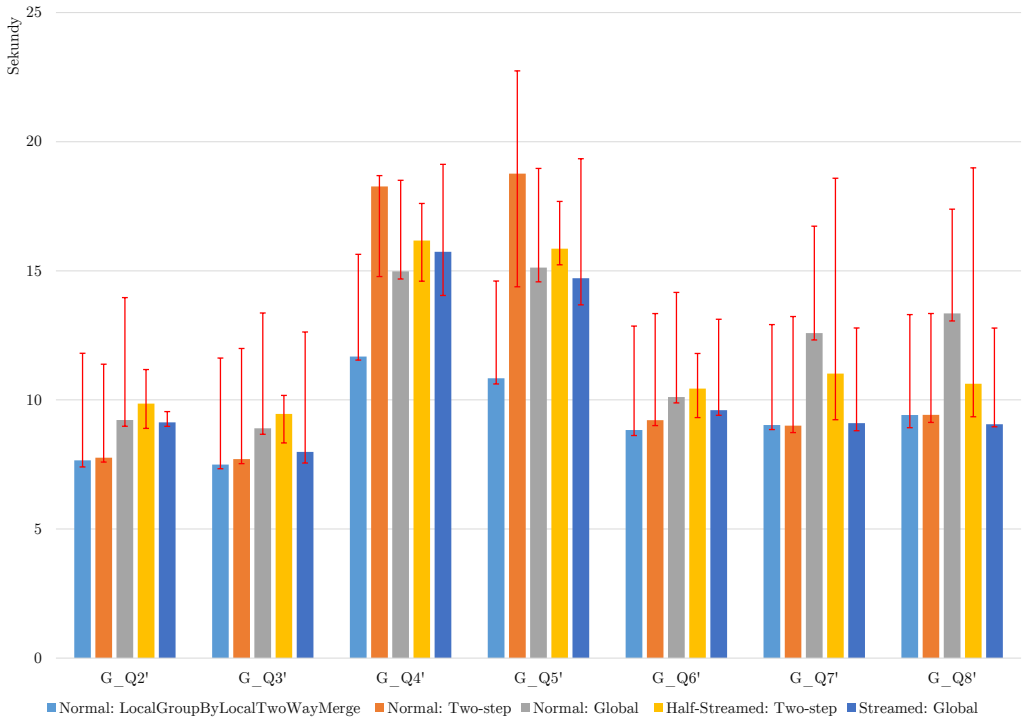
Obrázek 4.20: Doba vykonání dotazů Group by pro graf Amazon0601 (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 32373599.



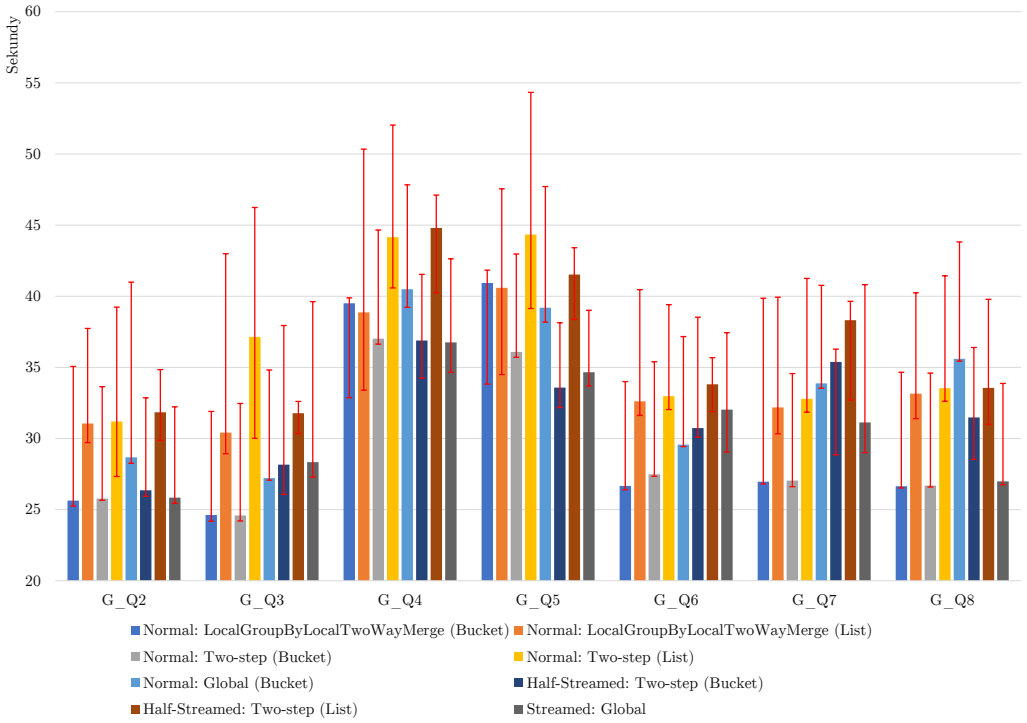
Obrázek 4.21: Doba vykonání dotazů Group by bez agr. funkcí pro graf Amazon0601 (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 32373599.



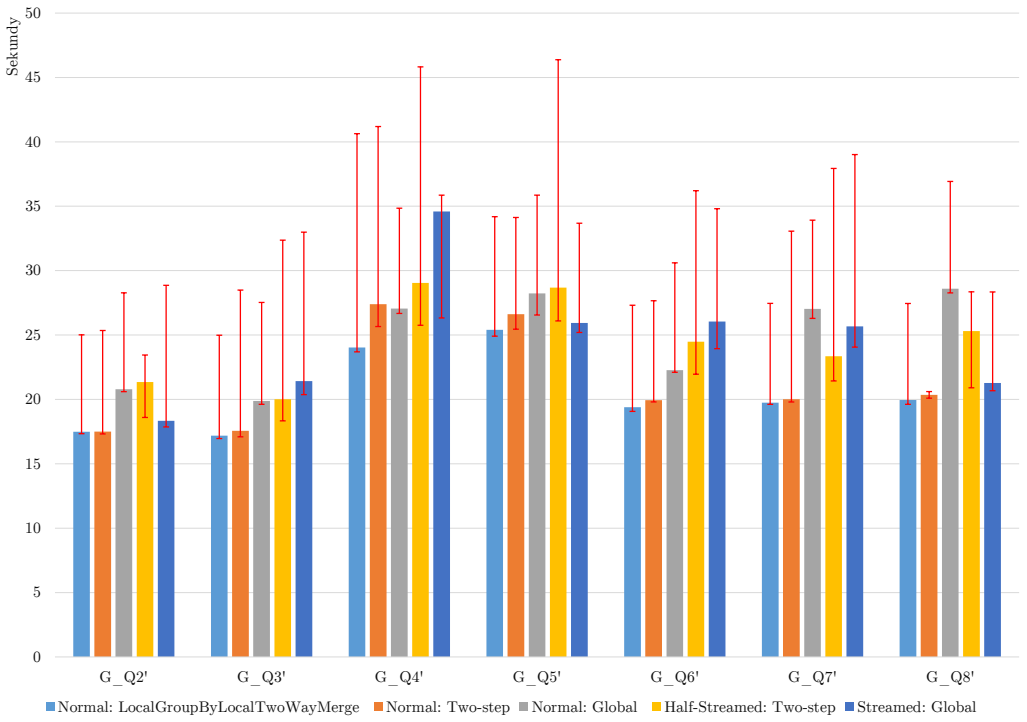
Obrázek 4.22: Doba vykonání dotazů Group by pro graf WebBerkStan (sekce 4.1). Běh osmi vláken. Počet seskupovaných výsledků je 222498869.



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.

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

5. Závěr

Tady ma byt text

5.1 Budoucí výzkum

1. Rozšíření enginu o možnost zadat Order by a Group by společně. Agregování v průběhu hledání se dá rozdělit na dvě hlavní části. V první části lze navrhnout řešení pro dotazy, které obsahují třídění pouze podle klíčů skupin. V druhé části je nutné vyřešit problém třídění pomocí výsledků agregačních funkcí. Zde je největší problém fakt, že výsledky agregačních funkcí jsou známy pouze po dokončení Group by.

2. Testování daných řešení na grafech s reálnými Properties. V našem testování jsme sice volili reálne grafy, ale jejich Properties jsme uměle vygenerovali.

3. Sledování obecného problému rozdělení dat při paralelizaci vylepšených řešení. Normal přistup má vždy všechna data připravená v paměti a při zpracování je rovnoměrně rozděluje mezi vlákny. Vlákna tedy mají vždy stejný počet výsledků pro zpracování. Navíc díky kompletnosti dat lze data optimálněji zpracovávat a použít větší množství obecných algoritmů. Například při třídění jsme použili základní algoritmu Merge sort, který není možný aplikovat při třídění v průběhu vyhledávání. Rozdělení práce vylepšených řešení závísí na počtu vyhledaných výsledků v každém vlákně. Mohou nastávát případy, kdy jedno vlákno má více výsledků ke zpracování než ostatní. Daný problém jsme se v našem řešení prohledávání snažili vyřešit pomocí přidělování malých skupin vrcholů vláknum. Vlákno po prohledání daných vrcholů zažádalo o další. Nicméně, dané řešení nemůže zaručit stoprocentně rovnoměrné rozdělení práce. Bylo by vhodné prozkoumat, jak daná situace ovlivňuje naše řešení.

4. U Order by řešení jsme viděli značné zrychlení při třídění pomocí Properties v paralelizovaných řešeních. Bylo by vhodné prozkoumat možnosti vytvoření globálních statistik pro každou Property a podrobněji zjistit možnosti rozdělení rozsahů přihrádek. Samotné rozdělení přihrádek jsme pro řetězce zpracovali pouze s předpokladem, že se jedná o ASCII znaky. V budoucí práci je možné zkoumat rozdělování i pro složitější znakové sady.

5. V paralelních Group by řešeních by bylo vhodné prozkoumat podrobněji skalabilitu daných řešení pro rozličné počty vláken. Pokud možno, také možnosti jiných paralelních map.

Seznam použité literatury

Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley. ISBN 0321486811. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321486811.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. a Stal, M. (1996). *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0-471-95869-7.

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

Gamma, E., Helm, R., Johnson, R. a Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition. ISBN 0201633612. URL http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

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].

Mareš, M. a Valla, T. (2017). *Průvodce labyrintem algoritmů*. CZ.NIC, z.s.p.o. ISBN 978-80-88168-19-5.

Needham, M. a Hodler, E. A. (2019). *Graph Algorithms*. O'Reilly Media, Inc. ISBN 9781492047681.

Roth, P. N., Trigonakis, V., Hong, S., Chafi, H., Potter, A., Motik, B. a Horrocks, I. (2017). Pgx.d/async: A scalable distributed graph pattern matching engine. *GRADES'17: Proceedings of the Fifth International Workshop on Graph Data-management Experiences and Systems*, (7), 1–6. doi: https://doi.org/10.1145/3078447.3078454.

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