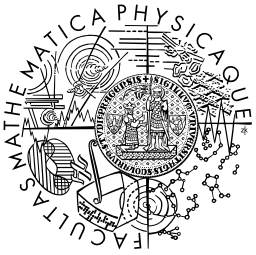


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

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. Dále 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

Grafové databáze

Grafové databáze zažívají v dnešní době rozkvět, například kvůli nutnosti analyzovat data na sociálních sítích. Pro grafové databáze existují dva hlavní modely dat: RDF [1] a Labeled-property (sekce 1.1). RDF je převážně používán k popisu informací na internetu. Labeled-property model se používá k obecnému popisu grafových dat. Grafové databáze poskytují dvě základní možnosti analýzy dat:

- První možnost je **spouštění algoritmů** nad uloženými grafovými daty. Samotná skupina používaných algoritmů obsahuje od jednoduchých algoritmů jako hledání nejkratší cesty až po komplexní algoritmy jako Community detection [2, str. 115]. Program, který poskytuje tuto možnost analýzy dat, budeme označovat jako **analytický engine**.
- Druhá možnost je **vykonávání dotazů** pomocí dotazovacího jazyka. Dotazy jsou zde podobné SQL dotazům. Pro RDF model to je například jazyk SPARQL [3]. Pro Labeled-property to je PGQL [4] nebo openCypher [5]. Dotazy obsahují základní části se stejnou logikou z SQL jako *Select*, *Order by*, *Group by*, *Having* a podobně. Hlavní rozdíl těchto dotazovacích jazyků od SQL je ten, že specifikují výběr dat pomocí sekvence vrcholů a hran. Pro PGQL a openCypher to je část dotazu *Match* a pro SPARQL to je část dotazu *Where*. Tyto sekvence tvoří podgraf (vzor), který bude vyhledáván v grafu. V moment nalezení všech podgrafů jsou na výsledcích aplikovány další části jako *Group by*, *Order by*, a podobně. Tomuto se říká vyhledávání vzoru, nebo-li pattern matching. Program, který poskytuje tuto možnost analýzy dat, budeme označovat jako **dotazovací engine**.

Proudové systémy

Na druhé straně množství dat ke zpracování poslední dobou značně roste a nastává problém při jejich zpracování, protože se všechny nevejdou do paměti nebo dochází k jejich velmi častým změnám. Z tohoto důvodu začaly vznikat takzvané **proudové systémy** [6]. Proudové systémy obecně pracují s daty, která jsou potencionálně nekonečná a nevlezou se do paměti. V takovém případě jsou data reprezentovány formou tekoucího „**proudu**“ (stream). Obecně uvnitř proudu mohou být jakákoliv data, například prováděné akce, události změn nebo transakce. Programy pracující s proudem dat sledují jen jeho určitý bod a zpracovávají data v moment kdy protékají daným bodem. Výsledně program vidí jen útržky všech dat. Zpracování takových dat pak probíhá po částech.

Existují dva hlavní modely zpracování [7]. První model udržuje kondenzovaný stav při zpracování dat a nazývá se **synopsis**. Dobrým příkladem může být počítání sumy hodnot prvků v proudu. V takovém případě si program udržuje pouze sumu výsledků. Data v proudu pak nepotřebuje uchovávat. Druhý model si udržuje okno určitého počtu posledních prvků z proudu a nazývá se **windowing**.

Prvky v okně tvoří skupinu, která se v určitý moment zpracuje najednou. Proudové systémy se uchytili v distribuovaných systémech, například Apache Flink [8].

Proudové systémy s grafovými daty

Nově se objevují proudové systémy pracující s grafovými daty. V proudu jsou pak obsaženy hrany nebo vrcholy. Nad daným proudem vzniká nutnost analýzy grafových dat jako u grafových databází[1]. Mezi proudové systémy s grafovými daty patří například rozšíření pro Apache Flink Gelly [10].

Proudové systémy s grafovými daty mají výhodu při agregaci prvků vůči grafovým databázím, protože při použití modelu synopsis jim stačí uchovávat pouze agregované prvky. Agregací zde rozumíme akt seskupování prvků podle určitých klíčů (obecně část *Group by* v SQL, PGQL nebo SPARQL). Součástí agregace je i výpočet statických funkcí pro vzniklé skupiny. Tyto funkce označujeme jako agregační funkce. Mezi nejčastější agregační funkce patří funkce:

- `min(...)`, která vrací minimum z hodnot ve skupině..
- `max(...)`, která vrací maximum z hodnot ve skupině..
- `count(...)`, která vrací počet prvků ve skupině.
- `avg(...)`, která vrací aritmetický průměr z hodnot ve skupině..
- `sum(...)`, která vrací sumu z hodnot ve skupině.

Problémem je, že proudové systémy nedovolují vykonávat dotazy obsahující vyhledávání vzoru (část *Match* pro PGQL nebo *Where* pro SPARQL), protože grafová data čteme po částech a tak je vyhledávání vzoru prakticky nemožné. Jediným možným způsobem je vytvořit přehledovou strukturu grafu [11], na které vyhledávání proběhne. Avšak, tím se ztratí všechny výhody proudových systémů.

Spojení dotazovacího enginu s proudovým systémem

Na základě poznatků z předchozí sekce v této práci provedeme úpravu dotazovacího enginu po vzoru proudových systémů. Konkrétně upravíme části zabývající se agregací dat, tj. části *Group by* a *Order by*. Ačkoliv *Order by* nespadá konkrétně pod definici agregace, tak z našeho pohledu provádí „primitivní seskupení“. V takovém případě výsledky se stejnými hodnotami klíčů třídění ve výstupu tvoří souvislou posloupnost (tj. leží u sebe), protože taková posloupnost nemůže obsahovat prvek, který má rozdílné hodnoty klíčů od hodnot klíčů prvků v dané posloupnosti. Kdyby takový prvek existoval, pak by se nejednalo o setříděnou posloupnost. Z tohoto důvodu v naší práci budeme upravovat i část *Order by*. Výsledně rozdíl mezi *Group by* a *Order by* je ten, že v *Group by* jsou výše

[1] Přehledovou práci základních algoritmů a principů zpracování je možno nalézt v článku „Graph Stream Algorithms: A Survey“ [9].

zmíněné posloupnosti sloučeny do jednoho prvku (tj. skupiny) a lze pro ně vypočítat agregační funkce. Avšak, zaměříme se pouze na jejich separátní použití a nebudeme uvažovat situaci, ve které jsou obě části zadány společně. Dále také budeme uvažovat, že dotaz vždy bude obsahovat pouze části *Select*, *Match* (PGQL a openCypher)/ *Where* (RDF) a následně *Group/Order by*.

Při úpravě dotazovacího enginu využijeme principů zpracování dat proudových systémů. Místo abychom výsledky prohledávání grafu agregovali až v moment získání všech výsledků, tak nalezené výsledky budeme agregovat již v moment jejich nalezení. V takovém případě se na část *Match*/ *Where* můžeme dívat jako na proud dat, který obsahuje nalezené výsledky prohledávání grafu. Prvek je vložen do proudu v moment jeho nalezení. Následně části *Order by* a *Group by* zpracovávají data v proudu. Cílem a motivací této práce je zjistit, zda danou úpravou docílíme zrychlení vykonávání dotazů provádějící agregaci dat (tj. dotazy obsahující část *Group/Order by*).

Výběr dotazovacího enginu

K provedení této práce potřebujeme upravit dotazovací engine. Při našem průzkumu jsme zjistili, že využití již existujících dotazovacích enginů v naší práci by bylo netriviálně náročné. Děje se tak ze dvou hlavních důvodů. Komerční dotazovací enginy, například Neo4j [12] nebo Neptune [13], jsou volně přístupné. Nalezené dotazovací enginy s veřejným zdrojovým kódem, například dgraph [14], jsou značně komplexní a pouze pochopení všech důležitých aspektů k vykonání zmíněných úprav by zabralo netriviální časové období. Protože cílem této práce je pouze otestovat obecný koncept propojení dotazovacího enginu s proudovým systémem, rozhodli jsme se vytvořit vlastní dotazovací engine s jednoduchou grafovou databází.

Námi vytvořená grafová databáze bude využívat Labeled-property (sekce 1.1) datový model, protože nám umožní pracovat s obecnými grafovými daty. Samotně bude pouze statická a bude celá obsažena v hlavní paměti, jelikož návrhnout komplexní databázi by opět zabralo netriviální časové období. Statickou databází zde rozumíme databázi, ve které nedochází za běhu k úpravě uložených dat (např. nelze přidat nebo odebrat nový vrchol). Dotazovací engine bude zpracovávat dotazy nad danou grafovou databází. K dotazování využijeme podmnožinu jazyka PGQL [4], jelikož obsahuje jednodušší skladbu jazyka na rozdíl od jazyka openCypher [5] a umožní nám tak jednodušší načítání dotazů. Daný jazyk má spoustu funkcí, které v naši práci nepovažujeme za vitální, proto jsme vybrali jen jeho určitou podmnožinu (podmnožina je blíže popsána v sekci 1.2). Samotný vývoj dotazovacího enginu bude probíhat ve dvou krocích. V prvním kroku musíme vyvinou řešení, která vykonávají agregace po dokončení prohledávání grafu. V tomto bodě se budeme snažit využít již existujících a hojně využívaných algoritmů (např. pro třídění algoritmus Merge sort). V druhém kroku dojde k úpravě vykonávání dotazů dle zadání práce. Při vývoji budeme klást důraz na jednovláknové i paralelní zpracování. Nakonec provedeme experiment, který otestuje vytvořená řešení v prvním kroku vůči nově navrženým řešením v druhém kroku.

Cíl práce

Hlavním cílem práce je určit zda úprava částí *Group by* a *Order by* dotazovacího enginu po vzoru proudových systémů dociluje rychlejšího zpracování dotazů než zpracování, které by dané části vykonávalo po získání všech výsledků prohledávání grafu. Kroky k dosažení cíle jsou následovné:

1. Navrhnout dotazovací engine, který bude sloužit jako výchozí prostředí pro naši práci. Obsahuje dvě části:
   (a) První část obsahuje jednu **statickou grafovou databázi**, která pracuje s **Labeled-property** modelem (sekce 1.1) grafových dat. Celá grafová databáze bude obsažena v hlavní paměti.
   (b) Druhá část obsahuje algoritmy a struktury pro zpracování dotazu **podmnožiny jazyka PGQL** (sekce 1.2) nad danou grafovou databází:
      i. Podmnožina jazyka PGQL bude obsahovat pouze hlavní částí dotazu: *Select*, *Match*, *Group by* a *Order by*. **Nebude možno** zadat *Group by* a *Order by* společně.
      ii. Všechna data v průběhu zpracování dotazu budou obsažena v hlavní paměti.
      iii. Zpracování částí *Group by* a *Order by* bude provedeno **po dokončení prohledávání grafu** v části *Match*. To znamená, že nejdříve dojde k nalezení všech výsledků prohledávání a teprve pak dojde k seskupení nebo setřídění daných výsledků.
      iv. Obecně engine bude schopen vykonat dotaz jednovláknově. Části *Match*, *Group by* a *Order by* bude schopen vykonat i paralelně.
2. Upravit druhou část výše definovaného dotazovacího enginu po vzoru proudových systémů tak, aby vykonávání částí *Group by* a *Order by* probíhalo v průběhu prohledávání grafu časti *Match*:
   (a) V moment nalezení jednoho výsledku prohledávání grafu v části *Match* bude daný výsledek zpracován. Zpracováním zde rozumíme zatřídění do již setříděné posloupnosti výsledků nebo přiřazením výsledku do patřičné skupiny.
   (b) Engine po úpravě musí být schopen zpracovávat dotazy původním řešením i upraveným řešením.
   (c) Upravená část bude pracovat se stejnou podmnožinou jazyka PGQL jako původní neupravená část.
   (d) Všechna data v průběhu zpracování budou obsažena v hlavní paměti.
   (e) Obecně upravená část bude schopna vykonat dotaz jednovláknově. Části *Match*, *Group by* a *Order by* bude schopna vykonat i paralelně.
3. Vykonat experiment, který testuje původní zpracování (odrážka 1.b) vůči upravené části (odrážka 2.) v rychlosti vykonání dotazů. Cílem experimentu je určit, zda úprava dotazovacího enginu po vzoru proudových systémů způsobí zrychlení vykonávání dotazu:

(a) Experiment bude proveden na několika reálných grafech s reálnými nebo uměle vygenerovanými vlastnostmi.

(b) V experimentu otestujeme všechna navržená řešení.

(c) Každé řešení bude otestováno na několika vybraných dotazech.

(d) Experiment bude zakončen prezentací výsledků a diskuzí.

1. Předpoklady

V úvodu jsme uvedli, že budeme pracovat s Labeled-property grafovým modelem a dotazy nad daným modelem budou provedeny v jazyku PGQL. V této kapitole popíšeme konkrétněji daný model a jazyk.

1.1 Labeled-property datový model

Při zpracování této práce uvažujeme grafovou databázi, která pracuje s grafovým modelem Labeled-property[1]. V této sekci popíšeme daný model a jeho vlastnosti z pohledu grafové databáze. Překvapivě neexistuje standardní definice daného modelu, ačkoliv je značně využíván v moderních systémech. Proto budeme vycházet z definice modelu systému Neo4j [12] a jazyka PGQL [4]. Na základě těchto definic určíme klíčové vlastnosti modelu, které budeme uvažovat v naší práci.

Obecný popis Labeled-property modelu

Obecně grafová databáze pracující s daným modelem organizuje grafová data do tří částí:

- První část obsahuje množinu **vrcholů** (Vertices). Vrcholy zde představují entity nebo doménové komponenty grafu. Vrchol tedy reprezentuje objekt, osobu, myšlenku, dílo a podobně.
- Druhá část obsahuje množinu **hran** (Edges). Hrany jsou zde vždy orientované a spojují dva vrcholy. Hrany zde představují vztahy mezi vrcholy. Hrana tedy může být například vztah kamarádství dvou lidí (Karel *se kamarádí s* Jirkou.), vztah zaměstnání člověka se zaměstnavatelem (Karel *pracuje pro Google.*) a podobně. Vrcholy a hrany zde označíme za **elementy** grafu.
- Poslední třetí část představuje **vlastnosti** (Properties) a **štítky** (Labels), které jsou uloženy v elementech grafu.
  - Štítek představuje označení určité skupiny nebo chování. Každému elementu může být přiděleno několik štítků. Na elementy, kterým byl přidělen stejný štítek, se můžeme dívat jako na skupinu.
  - Vlastnosti definují dvojice (`název, hodnota`). První z dvojice `název` udává název vlastnosti a `hodnota` udává hodnotu dané vlastnosti. Hodnota vlastnosti má také vždy definován svůj skalární **typ** (například číselná hodnota nebo řetězec). Každému elementu může přináležet několik vlastností. Avšak, jeden element nemůže mít dvě dvojice sdílející název. Navíc pokud dva různé elementy sdílejí vlastnost (tj. mají stejný název), pak typy hodnoty vlastnosti musí být rovněž totožné.

[1] `https://en.wikipedia.org/wiki/Graph_database` [dostupnost ověřena k datu 8.5.2021]

Omezení Labeled-property modelu

Popsali jsme obecně Labeled-property model. Daný popis modelu je značně abstraktní, proto si určíme další zpřísňující vlastnosti, které budeme využívat v naši práci.

1. Každý element bude mít unikátní identifikátor (`ID`), abychom dokázali rozlišovat dané elementy. `ID` zde nechápeme jako vlastnost elementů, ale jako interní značení uvnitř grafu.

2. Každý element (vrchol/hrana) bude mít právě jeden štítek. Myslíme, že počet štítků není určující pro naši práci, protože primárně ovlivňuje prohledávání grafu a ne části *Order by* nebo *Group by*.

3. Budeme uvažovat, že štítky vrcholů jsou vždy rozdílné od štítků hran.

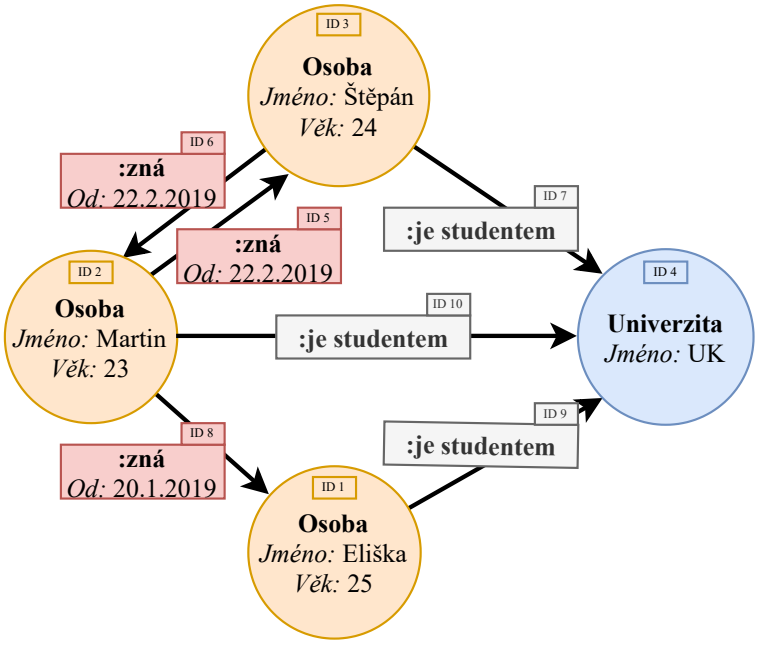
4. Každému štítku přiřadíme výčet vlastností. To znamená, že štítek nemusí mít žádnou vlastnost nebo jich má určitý počet.

5. Štítky mohou sdílet vlastnosti, ale vlastnosti musí mít stejný typ hodnoty. To znamená, že i štítek hrany se štítkem vrcholu může mít stejnou vlastnost, pokud mají stejný typ hodnoty.

Samotná grafová data nebudeme nijak omezovat. To znamená, že v grafu mohou být cykly, dva vrcholy mohou být propojeny několika hranami, nemusí existovat žádná hrana ani vrchol, mohou existovat nepropojené vrcholy, hrana může mít totožný počáteční vrchol s koncovým vrcholem a podobně.

Příklad grafu splňující Labelel-property model

Nyní uvedeme příklad jednoduchého grafu splňující náš model:



Obrázek 1.1: Příklad Property grafu.

Na obrázku 1.1 vidíme tři vrcholy se štítkem **Osoba**, které reprezentující fyzické osoby, a jeden vrchol se štítkem **Univerzita** reprezentující vysokou školu. Štítku **Osoba** byly přiděleny dvě vlastnosti: *Jméno* (řetězec) a **Věk** (číselná hodnota). Štítku **Univerzita** byla přidělena jedna vlastnost: *Jméno*, která odpovídá stejné vlastnosti štítku **Osoba** a mají stejný typ hodnoty (řetězec). Mezi vrcholy fyzických osob existuje vztah **zná**, který určuje zda jedna osoba (počáteční vrchol) zná druhou osobu (koncový vrchol). Vlastnost *Od* definuje datum, kdy došlo k poznání osoby. Mezi vrcholy fyzických osob a vysokou školou existuje vztah **je studentem**, který říká zda osoba studuje na dané vysoké škole. Tento vztah nemá žádnou vlastnost. Samotně pak každý element má přiřazeno `ID`. Příklad vznikl na základě prvního grafu z dokumentace níže popisovaného PGQL.

V průběhu práce budeme graf splňující daný model označovat jako **Property graf**.

1.2 Jazyk PGQL

V naši práci budeme používat dotazovací jazyk PGQL [4] verze 1.2 k dotazování nad Property grafem. Jazyk je značné obsáhlý a k realizaci naši práce nepotřebujeme všechny jeho vlastnosti, proto jsme se rozhodli využít jeho podmnožinu. V naši práci budeme pracovat jen s částmi *Select*, *Match*, *Group by* a *Order by*. Ostatní části nebudeme používat. V každém dotazu musí být vždy přítomná část *Select* a *Match*. S části *Match* se pojí část *From* specifikující název grafu, nad kterým se má vykonat dotaz. Nicméně, v naši práci vždy budeme pracovat s jedním grafem a tedy jsme část vyřadili. Zároveň v naši práci uvažujeme, že *Order by* a *Group by* nejsou zadány nikdy společně.

- *Match:* definuje podgraf, který bude vyhledán v grafu. Součástí definice podgrafu jsou proměnné, které jsou přístupné v ostatních částech dotazu. Proměnná zde reprezentuje jeden element podgrafu.
- *Select:* určuje vrácená data ve formě tabulky jako u tabulkových databází. K jednomu výsledku tak můžeme vrátit výčet informací. Každá položka výčtu pak definuje jeden sloupeček tabulky.
- *Group by:* dovoluje seskupit výsledky dle zadaných klíčů a vypočítat agregační funkce pro výsledné skupiny.
- *Order by:* vykoná setřídění výsledků dle zadaných klíčů.

Nyní krátce popíšeme omezení a klíčové vlastnosti, s kterými budeme pracovat. Pro komplexní popis odkazujeme na citovanou dokumentaci.

Match

Část *Match* se skládá z posloupností elementů definující hledaný podgraf. Posloupnosti jsou odděleny čárkou. Položky v posloupnosti jsou vrcholy a hrany. Položky vrcholů značíme pomocí `()` a položky hran pomocí šipek `->` (dopředná hrana), `<-` (zpětná hrana) a `-` (libovolná hrana).

Každé položce je možno přidělit jméno (definice proměnné) a predikát štítku. Vrcholu přidělíme proměnnou vložením jména do kulatých závorek: `(jméno)`.

Hraně přidělíme proměnnou vložením jména do hranatých závorek: `-[jméno]->`, `<-[jméno]-` a `-[jméno]-`. V části *Match* musí být vždy přítomna alespoň jedna proměnná. Pokud se v posloupnostech opakují proměnné znamená to, že v průběhu hledání musí dané položky obsahovat elementy se shodným `ID`. Nicméně, proměnné hran se nesmí opakovat. Pokud dvě posloupnosti nemají společnou proměnnou, tak výsledek je skalární součin daných posloupností.

Predikát je vložen do závorek za jméno, pokud jméno chybí tak jen do závorek: `(jméno:štítek)`, `(:štítek)`, `-[jméno:štítek]->` a `-[:štítek]->`. Predikát určuje výběr elementů pouze s daným štítkem. Pokud se v posloupnostech opakuje proměnná se štítkem, pak štítek musejí mít i opakované definice proměnných. Protože v našem modelu elementy mají pouze jeden štítek, tak element posloupnosti může mít pouze jeden predikát. Navíc jsme vyřadili možnost aplikovat v predikátu operátor „|“ (logická operace nebo), tj. nelze použít `(jméno:štítek|štítekDva)`. Následují příklady *Match* části na grafu z obrázku 1.1 (bez *Select*) a uvedené výsledky hledání rovněž odpovídají danému obrázku:

1. ```
   match (x:Osoba) -> (y:Osoba)
   ```

   Jsou zde dva vrcholy s názvy proměnných `x` a `y`. Oba mají predikát štítku `:Osoba`. To znamená, že do položek posloupnosti budou vybrány pouze elementy s daným štítkem. Část *Match* v průběhu hledání využívá pouze hrany vedoucí z proměnné `x`. Výsledky hledání jsou dvojice vrcholů `x-y`: 2-1, 2-3, 3-2. Hodnoty jsou zde `ID` elementů.

2. ```
   match (x:Osoba) <- (y:Osoba)
   ```

   Část *Match* v průběhu hledání využívá pouze hrany vedoucí do proměnné `x`. Výsledky hledání jsou dvojice vrcholů `x-y`: 1-2, 2-3, 3-2. Hodnoty jsou zde `ID` elementů.

3. ```
   match (x:Osoba), (x:Osoba)
   ```

   Zde vidíme dvě posloupnosti sdílející proměnnou `x`. Protože první měla predikát, tak i druhé opakování musí mít stejný predikát. Výsledky hledání jsou dvojice vrcholů `x-x`: 1-1, 2-2, 3-3. Hodnoty jsou zde `ID` elementů.

4. ```
   match (x:Osoba), (y:Osoba)
   ```

   Zde vidíme dvě separátní posloupnosti bez sdílené proměnné, proto dojde ke vzniku skalárního součinu výsledků posloupností. Výsledky hledání jsou dvojice vrcholů `x-y`: 1-1, 1-2, 1-3, 2-1, 2-2, 2-3, 3-1, 3-2, 3-3. Hodnoty jsou zde `ID` elementů.

Výrazy

Zbylé části se skládají převážně z výrazů (např. `select x, y ...`, kde `x` a `y` jsou funkce vracející `ID` elementů). Samotné výrazy v jazyku PGQL mohou být značně komplikované, proto jsme vybrali tři základní výrazy, které budeme využívat.

- Funkce vracející `ID` elementu. V dotazu se reprezentuje použitím názvu proměnné. Lze zadat v jakékoliv části kromě *Match*. Například:

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

  Dotaz obsahuje dva výrazy `x` a `y` v části *Select*. *Select* část vytváří tabulku se dvěma sloupečky `x` a `y`, které obsahují `ID` elementů v daných proměnných.

- Přístup k hodnotě vlastnosti elementu. Lze zadat v jakékoliv části kromě *Match*. V dotazu se reprezentuje použitím názvu proměnné s názvem vlastnosti sloučených pomocí tečky. Například:

```
select x.VlastnostJedna, y.VlastnostDva match (x) -> (y)
```

  Dotaz obsahuje dva výrazy `x.VlastnostJedna` a `y.VlastnostDva` v části *Select*. *Select* část vytváří tabulku se dvěma sloupečky `x.VlastnostJedna` a `y.VlastnostDva`, které obsahují hodnoty vlastností elementů v daných proměnných. Tento výraz se například nemusí povést vyhodnotit, protože vlastnost nemusí existovat na daném elementu. V takovém případě budeme předpokládat, že výraz vrací nějakou konstantu definující selhání výpočtu.

- Agregační funkce. Agregační funkce vypočítají výslednou hodnotu pro skupiny vytvořených při seskupování (*Group by*). Pokud nebylo zadáno *Group by* a je použita agregační funkce, pak ve zbylých částech kromě *Match* mohou být pouze agregační funkce. V takovém případě všechny výsledky patří do jedné skupiny. Agregační funkce povolíme pouze v části *Select*, protože *Order by* a *Group by* nemůžou být nikdy zadány společně. Nedovolíme rekurzivní používání daných funkcí, tj. agregační funkce nemůže mít na vstupu další agregační funkci. Funkce na vstupu mají hodnotu vypočteného výrazu (tj. `ID` nebo přístup k vlastnosti). Musíme brát v potaz, že při výpočtu výrazu může dojít k selhání a tedy funkce musí umět pracovat s konstantou definující selhání výpočtu. PGQL definuje pět funkcí:

  1. Funkce minima vracející minimum z hodnot na vstupu. Značená `min(...)`. Pracuje s číselnými typy i řetězci.
  2. Funkce maxima vracející maximum z hodnot na vstupu. Značená `max(...)`. Pracuje s číselnými typy i řetězci.
  3. Funkce aritmetického průměru vracející aritmetický průměr hodnot na vstupu. Značená `avg(...)`. Pracuje pouze s číselnými typy.
  4. Funkce součtu vracející součet hodnot na vstupu. Značená `sum(...)`. Pracuje pouze s číselnými typy.
  5. Funkce počtu vracející počet prvků na vstupu, značená `count(...)`. Pracuje s číselnými typy i řetězci. Funkce má dva módy. První mód je použití zápisu `count(výraz)`, kde `výraz` je přístup k `ID` nebo vlastnosti elementu. Druhý mód je použití zápisu `count(*)`. Tento zápis nepotřebuje vyhodnotit výrazy. Pokud byl například zadán dotaz: `select count(*) match (x)`. V tomto dotaze pak nepotřebujeme znát elementy z části *Match*, ale stačí nám jejich počet.

Select

Select část může obsahovat pouze výše zmíněné výrazy oddělené čárkami. V této části lze přistoupit k proměnným, které jsou definované v *Match* části, nebo zadat výrazy agregačních funkcí. Výsledkem části *Select* je tabulka. Sloupečky v tabulce představují výrazy zadané v dané části. Řádky tabulky jsou pak hodnoty výrazů pro výsledky z části *Match* nebo *Group/Order by*. Existuje ještě speciální operátor `*`, který lze zadat pouze v případě, pokud je v dotazu jen část *Select* a *Match*. V takovém případě se `*` nahradí všemi proměnnými z části *Match*. Následují příklady použití *Select* části:

- Použití obyčejných výrazů:

```
select x, y, x.Vlastnost match (x) -> (y)
```

  Dojde k návratu tabulky se třemi sloupečky. Sloupečky obsahují hodnoty daných výrazů.

- Použití operátoru `*`:

```
select * match (x) -> (y)
select x, y match (x) -> (y)
// dotazy jsou ekvivalentní
```

- Použití agregačních funkcí v *Select* bez části *Group by*:

```
select min(x.Vlastnost), avg(x.Vlastnost) match (x) -> (y)
```

  V takovém případě je výsledkem dotazu pouze jedna dvojice výsledných hodnot dvou funkcí, protože všechny výsledky jsou seskupeny do jedné skupiny. Část *Select* zde tedy může zadávat pouze agregační funkce.

- Použití části *Select* zároveň s částí *Group by*:

```
select avg(x.Vlastnost), x match (x) -> (y) group by x
```

  Část *Select* může obsahovat pouze výrazy agregačních funkcí nebo výrazy zadaných v části *Group by*. Výše zmíněný dotaz bez výrazu `x` v *Select* části a ponecháním pouze agregační funkce je správně. Rovněž odstraněním agregační funkce z *Select* části a zanecháním pouze druhého výrazu je správně. Výsledná tabulka má počet řádků rovný počtu vytvořených skupin.

Order by

Order by definuje klíče (výrazy), podle kterých budou výsledky z části *Match* setříděny. *Order by* může přistupovat pouze k proměnným z *Match* části. *Order by* a *Select* části nemusí mít shodné výrazy. Výrazy v této části jsou opět odděleny čárkami. Operátor `asc`/`desc` za výrazem definuje zda jsou hodnoty tříděny vzestupně/sestupně. Pokud operátor chybí, tak je jako výchozí operátor použit `asc`. Následují příklady použití *Order by* části:

- Použití jednoho klíče:

```
select x, y match (x) -> (y) order by x
```

  Výsledky z *Match* jsou setříděny podle hodnot `x` vzestupně.

- Použití dvou klíčů:

```
select x, y match (x) -> (y) order by x, y
```

  Výsledky z části *Match* jsou setříděny podle hodnot `x` vzestupně. Pokud se dva výsledky shodují v dané hodnotě, pak je použito porovnání hodnotou `y`. Obecně se aplikuje stejný postup, pokud je tříděno podle více než dvou klíčů.

- Použití operátoru `desc`:

```
select x, y match (x) -> (y) order by x desc
```

  Výsledky z části *Match* jsou setříděny podle hodnot `x` sestupně.

Group by

Group by definuje klíče seskupení (výrazy). Výsledkem *Group by* je množina skupin. Skupiny obsahují výsledky prohledávání z části *Match*. Všechny výsledky v jedné skupině mají stejné hodnoty klíčů seskupení. Výrazy jsou odděleny čárkami. *Select* část může obsahovat výrazy agregačních funkcí a výrazy klíčů seskupení. Agregační funkce se vypočítají pro každou skupinu zvlášť. Pokud není zadána část *Group by* a dotaz obsahuje výrazy agregačních funkcí, pak se jedná o seskupení všech výsledků do jedné skupiny. Následují příklady použití *Group by* části:

- Použití jednoho klíče:

```
select x match (x) -> (y) group by x
```

  Výsledky z části *Match* jsou seskupeny podle hodnot `x`.

- Použití dvou klíčů:

```
select x, y match (x) -> (y) group by x, y
```

  Výsledky z části *Match* jsou seskupeny dle uspořádané dvojice (`x`, `y`). Pokud obecně existuje více klíčů, pak jsou seskupeny dle uspořádané n-tice.

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 prohledávání grafu v *Match* části. 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. Pokud v nějaké ukázce použijeme jazyk C# míníme tím jazyk C# pro .NET Framework 4.8.

2.1 Obecný pohled na engine

V naší představě je dotazovací engine určen pro práci nad grafem, který je celý obsažen v paměti a to včetně vlastností elementů grafu. Graf bude načten v definovaném formátu a následně na něm budou vykonávány dotazy. Graf po načtení bude pouze statický, tedy nebude v něm 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, načítání uživatelského dotazu, výrazy (expressions) a dotaz/vykonání dotazu. Graf nám bude představovat grafovou databázi. Samotně pak určuje formát objektů, nad kterými je vykonán uživatelský dotaz. Uživatelský dotaz se musí načíst 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. Blok dotaz/vykonání dotazu pak musí navíc obsahovat dva módy. A to upravené řešení a původní řešení. Uživatel si při spuštění enginu bude moct vybrat požadovaný mód.

Samotná představa vykonávání je následovná. Uživatel při spuštění aplikace vybere chtěný mód. Aplikace po zapnutí načte graf z datových souborů. Uživatel pomocí příkazové řádku zadá dotaz k vykonání. Dotaz se vykoná a po dokončení uživatel může opět zadat další dotaz. Myslíme, že zde není nutné složité grafické rozhraní a proto budeme engine považovat za konzolovou aplikaci.

2.2 Reprezentace grafu

Musíme uvážit, jak reprezentovat graf. Graf bude představovat 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 štítku a `propType` jako typ hodnoty vlastnosti.

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 vlastností na elementu. Vlastnosti jsou také typované. Vrchol a hrana musí mít rozdílný `elType`, ale samotné vlastnosti se mohou opakovat pro oba druhy elementů. Každá hodnota vlastnosti 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 vlastností.

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 vlastností v podobě polí s hodnotami. V daných polích budou hodnoty vlastností každého elementu ležet na pozicích určených dle jejich pořadí. Pro náš případ nebudeme uvažovat situaci, kdy vlastnost pro nějaký element nemá definovanou hodnotu. Vlastnost musí být přístupná skrze mapu/slovník, abychom k ní mohli rychle přistoupit. Každé vlastnosti tedy přidělíme unikátní identifikátor, který bude klíčem mapy/slovníku. Nyní pokud držíme element grafu, můžeme přistoupit k hodnotě jeho vlastnosti skrze ukazatel na danou třídu. Samotný přístup pak může být realizován například generickou funkcí.

2.2.2 Struktury obsahující elementy

Nyní musíme analyzovat jaké struktury by byly ideální 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 prohledávání grafu se z vrcholu musí projít skrze všechny jeho hrany.
- Pokud držíme hranu, musíme být schopni přistoupit ke koncovému vrcholu. V průběhu prohledávání grafu 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 obsahující 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 (ukazují na koncový vrchol hrany) a `in` hran (ukazují na počáteční vrchol hrany). 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. Tedy pro jednu hranu na vstupu budou existovat dva záznamy v daných polích, které se liší pouze koncovým vrcholem. Abstraktní předek všech elementů by si měl nově pamatovat i svou pozici v daných polích pro rychlý přístup k jeho sousedům. Každé pole tedy bude mít unikátní typ, který nám pomůže rozlišit k jaké situaci má dojít v průběhu prohledávání grafu.

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 vrcholů. Řešením by mohlo být vytvořit samostatné pole `out/in` hran pro každý vrchol. Daný přístup nám připadá výrazně náročnější z hlediska paměti, protože musíme vytvářet pole pro každý vrchol zvlášť.

2.2.3 Návrh vstupních grafových dat

Vstupní soubory musí obsahovat informace nutné pro Property graf. Budeme očekávat dva druhy souborů. Soubory schémat `elType` a jejich vlastností. Datové soubory pak budou obsahovat konkrétní data grafových elementů. (Jedná se o návrh a finální specifikaci uvedeme v rámci implementace.)

Soubory schémat

Protože každý element grafu má svůj `elType`, budeme mít na vstupu dva soubory schémat. Jeden pro hrany a jeden pro vrcholy. Schéma bude obsahovat informace o všech `elType` a jejich vlastností. Pro `elType` je důležitý název a výčet vlastností. Vlastnosti pak musí nést svůj název a `propType`. Vidíme, že se jedná jen o výčet `(Název/Hodnota)` dvojic. V tomto případě se nám jeví nejvhodnější zvolit pro reprezentaci schémat formát JSON [15]. `elType` bude reprezentován JSON objektem. První položka objektu je `Kind`, která představuje `Název` a její `Hodnota` udává jméno `elType`. Za ní bude následovat výčet vlastností. Vlastnosti budou opět reprezentovány dvojicí `(NázevVlastnosti/propType)`. Každý JSON objekt tedy definuje jeden `elType`. Všechny objekty pak budou obsaženy v JSON poli:

```
Soubor schéma vrcholů:
[    { "Kind": "NodeX" },
     { "Kind": "NodeY", "Vlastnost1": "integer" }
     { "Kind": "NodeZ", "Vlastnost2": "string",
       "Vlastnost1": "integer" }  ]
```

```
Soubor schéma hran:
[    { "Kind": "EdgeX" },
     { "Kind": "EdgeY", "Vlastnost1": "integer" } ]
```

Pro vrcholy zde vidíme tři `elType` s názvy `NodeX`, `NodeY` a `NodeZ`. První `elType` nemá definované vlastnosti, ale druhá má vlastnost `Vlastnost1` s typem `integer`. Poslední definuje dvě vlastnosti. `Vlastnost2` s typem `string` a za ní následuje opět `Vlastnost1` s typem `integer`. Samotné vlastnosti se mohou opakovat, ale musí mít vždy stejný `propType`. Následně identicky pro schéma hran. Ačkoliv hrana i vrchol by mohli mít stejný `elType`, tak budeme vždy uvažovat, že jsou rozdílná. Nicméně dovolíme, aby vrchol i hrana mohli mít stejnou vlastnost. Nyní musíme říct, co konkrétně znamenají `integer` a `string`. Jedná se o dva různé `propType`. První budeme chápat jako 32-bitovou číselnou hodnotu (v C# `Int32`). A druhý jako řetězec (v C# `string`). Práce s řetězci je obecný problém a existuje mnoho znakových sad, proto bychom chtěli omezit vstupní řetězce na základní znaky ASCII v rozsahu hodnot 0 až 127. Dané dva druhy představují základní typy používané i v komerčních sférách, proto chceme omezit vstupní typy pouze na tyto dva.

Datové soubory

Samotná data budou obsažena opět ve dvou separátních souborech pro hrany a vrcholy. Chtěli bychom reprezentovat konkrétní data pomocí jednoduchého textového souboru. Každý řádek bude reprezentovat jednu hranu/vrchol. V první řadě řádek musí obsahovat unikátní `ID` elementu a jeho `elType`. Za `elType` následuje seznam hodnot vlastností v pořadí určených schématem. To znamená, že první vlastnost v JSON objektu má hodnotu jako první v daném seznamu. Pro hrany existuje na řádku navíc záznam `ID` vrcholů, které spojuje. Tedy `ID` počátečního vrcholu (`fromID`) a `ID` koncového vrcholu (`toID`). Oddělovače mezi daty jsou implementační detail. Pro naše účely se jedná o dostačující formát a poskytuje nám jednoduché možnosti načítání dat. Pokud by docházelo v budoucnu k rozšířením, například víceslovné vlastnosti nebo XML vlastnosti, musí dojít k úpravě daných formátů. Pro výše zmíněné schéma by datové soubory mohly vypadat následovně:

```
Soubor hran (bez hlavičky a komentáře):
ID elType fromID toID Vlastnosti...
50 EdgeX 0 0    // EdgeX nemá vlastnosti.
51 EdgeY 0 1 44 // EdgeY má vlastnost Vlastnost1
                // s číselnou hodnotou.
52 EdgeX 1 2
...
Soubor vrcholů:
ID elType Vlastnosti...
0 NodeX    // NodeX nemá vlastnosti.
1 NodeY 42 // NodeY má jednu vlastnost.
2 NodeZ Martin 22 // Dvě vlastnosti v pořadí
                  // definované schématem.
...
```

2.3 Načítání uživatelského dotazu

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. 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 objektově orientovaného programování 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 [16]. V prvním kroku dojde k lexikální analýze uživatelsky zadaného řetězce. Dojde k vytvoření tokenů. V druhém kroku dojde k syntaktické a sémantické analýze tokenů. Metodou top-down parsing [16, 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 [17, 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 separátně a vyřadit tak nutnost jednoho globálního Visitoru. Dané postupy nám pak umožní jednoduše pracovat s naší podmnožinou jazyka PGQL.

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 nebo vrchol. Jedná se o všechny nutné informace, které můžeme následně využít k vytvoření vzoru prohledávání grafu. Všimnout si musíme faktu, že *Match* část definuje proměnné ve zbytku dotazu. Během načítání dotazu musíme určit zda se jedná o validní proměnnou a při výpočtech hodnot výrazů je nutné vědět přesně k jaké proměnné musíme přistoupit. Problém se dá řešit vytvořením mapy/slovníku přístupných 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 vlastnostem proměnných (např `select x.PropOne`) a volání agregačních funkcí (`min`, `max`, `avg`, `sum` a `count`). Proměnné zde představují elementy grafu. Výrazy se však musí vyhodnotit za běhu programu. Dalším problémem je, že výrazy mají různorodé návratové hodnoty. Výraz `x` (`ID` vrcholu) lze chápat jako číselnou hodnotu. Výraz `x.PropOne` má návratovou hodnotu dle `propType`, který je definován 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 aritmetické operace nebo zmiňované porovnání z *Where/Having* části. Problém nám usnadňuje fakt, že vlastnosti 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 vlastnosti 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 vlastnosti proměnné a agregační funkce (`min`, `max`, `avg`, `sum` a `count`).

Základní myšlenka je reprezentovat výraz pomocí stromové struktury. Obecně struktura bude využívat návrhový vzor Composite [17, str. 163]. 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. Úspěch je zde důležitý, protože například můžeme přistupovat k neexistující vlastnosti. Dané struktury musí být pouze ke čtení, protože se budou využívat v paralelním prostřední. Metody by se mohli libovolně dodávat při nutnosti použít novou strukturu k vyhodnocení výrazu.

Následuje ukázka možného kódu v jazyce C#:

```
// Základní rodičovské třídy
abstract class Expression { }
abstract class ExpReturnValue<T>: Expression {
  public abstract bool TryEvaluate(Element[] elms, out T retVal);
}
abstract class VariableAccess<T>: ExpReturnValue<T> {
     readonly int accessedVariableID; }
```

Třída reprezentující přístup k `ID` proměnné:

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

Typ návratu funkce je definován pomocí `T` parametru třídy `ExpReturnValue`. Volající tedy musí znát typ návratové hodnoty, aby funkci mohl vyhodnotit. 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 zadán `match (x) -> (y)`, tak jeden výsledek prohledávání by bylo pole obsahující dva elementy `x` a `y`. `Element` je zde chápán jako element grafu, tj. vrchol nebo hrana. Na elementu existuje položka `ID` s unikátním identifikátorem elementu. Funkce pak vrací hodnotu uloženou v `retVal` a úspěch

vyhodnocení v návratové hodnotě. Jedná se pouze o ilustrační příklad. Výsledný formát definujeme v implementační části.

Případ přístupu k vlastnosti 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`. `accessedPropertyID` je identifikátor přistoupené vlastnosti. Třída pak na základě existence vlastnosti vrátí hodnotu nebo neuspěje. Tímto dokážeme vyřešit základní definované problémy.

Agregační funkce

Zbývá uvažovat, jakým způsobem reprezentovat agregační funkce. Agregační funkce představují několik problémů. Funkce je vypočtena pouze pro skupiny. Skupiny jsou vytvářeny v části *Group by*. Jejich návratové hodnoty jsou finální pouze po dokončení *Group by*. Vstupem funkcí jsou výstupní hodnoty uživatelem zadaných výrazů. 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` přistoupené agregační funkce a na vstupu očekává strukturu reprezentující skupinu. K hodnotě počítané agregace přistoupíme pomocí její `ID`.

Následuje ukázka prvního konceptu:

```
abstract class Aggregation { }
abstract class Aggregation<T>: Aggregation {
  public ExpReturnValue<T> expr; // Argument ag. funkce.
  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;
    }
  }
}
```

Na ukázce výše vidíme položku `expr`, která reprezentuje vstupní výraz agregační funkce. Funkce `Apply` je logikou funkce. Vidíme funkci `Sum`. Logikou je přičtení vypočítané hodnoty do poskytnutého úložiště, pokud dojde k úspěšnému vyhodnocení výrazu.

Následuje ukázka druhého konceptu:

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

`Group` reprezentuje výsledky jedné skupiny. `accessedAggFuncID` je identifikátor vypočítané agregační funkce. Hodnota funkce se vrací pomocí `GetAggValue<T>`. Obecně ale vždy nemusí dojít k úspěchu vyhodnocení. Při výpočtu se například vždy přistoupilo k neexistující vlastnosti u všech elementů. Pomocí našeho návrhu pak můžeme vyřešit i budoucí rozšíření (např. zmíněné 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;
    }
  }
}
```

Zde `ExpressionIntegerAdd` rozšiřuje obecný binární operátor a určuje návratovou hodnotu výrazu. V těle funkce `TryEvaluate` dojde k pokusu o vyhodnocení dvou podvýrazů. Při úspěchu dojde k vypočtení finální hodnoty a v opačném případě dojde k selhání vyhodnocení.

Tímto jsme vyřešili problémy načítání 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 prohledávání grafu v *Match* části. 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 tak, aby vykonával *Group/Order by* v průběhu prohledávání grafu.

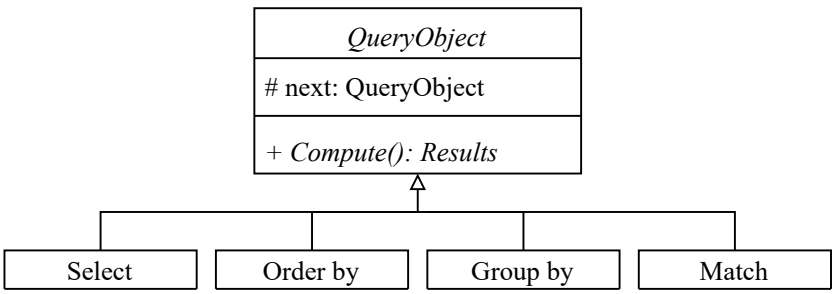
Při zkoumání vlastností hlavních částí PGQL (sekce 1.2) jsme si uvědomili jejich separaci logiky z hlediska vykonávání dotazu. Match prohledává graf a produkuje výsledky. *Select* výsledky vypisuje. *Order/Group by* třídí/seskupuje vyprodukované výsledky. Filtrace výsledků je prováděna v části *Where*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

Priority (zleva největší):

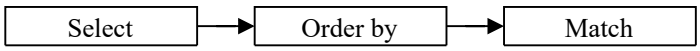
```
Match > Where > Group by > Having > Order by > Select
```

Match vyprodukuje výsledky, *Where* je filtruje, *Group by* je seskupí, *Having* opět filtruje, 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 [18, str. 53]. Ačkoliv v naší práci použitá podmnožina PGQL (sekce 1.2) neobsahuje *Having* a *Where*, tak jsme si vědomi provázanosti částí *Match/Where* a *Group by/Having*. Pokud by byly v budoucnu implementovány, pak mohou být propojeny pro docílení lepší výkonnosti.

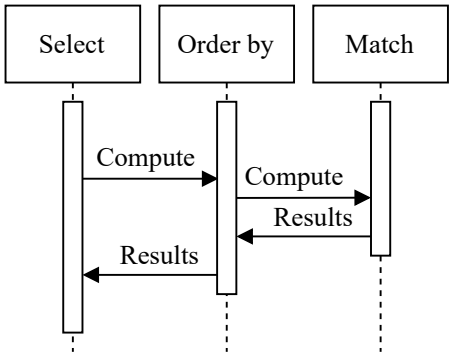
Poznatky jsme se rozhodli aplikovat v našem návrhu. Každá část dotazu bude reprezentována objektem (obrázek 2.1). Metoda `Compute` implementuje logiku objektu. Propojení se realizuje pomocí položky `next`. Dle uživatelského dotazu se vytvoří dané objekty a propojí dle priorit výše (obrázek 2.2). Propojení bude realizováno od nejmenší po největší, protože práce s nejvyšší prioritou je vykonána první a po dokončení její práce ji už nepotřebujeme. Tedy můžeme uvolnit její zdroje. Při vykonání se provede rekurzivní voláni funkce `Compute` na objektech 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. Existují dvě možnosti. Paralelizace bude provedena pouze interně pro každý objekt nebo dojde k vypracování složitějšího modelu. V naší představě bychom volili první variantu, díky které nebudeme potřebovat vytvářet závislosti mezi objekty.

2.4.2 Formát výsledků

Mezi částmi dochází k předávání výsledků. Výsledky musí mít definovaný formát, aby každá část dokázala správně provádět svou logiku zpracování. 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 prohledávání grafu budeme ukládat do tabulky. Jeden řádek tabulky bude reprezentovat jeden výsledek prohledávání. Nyní musíme rozhodnout, jaké informace do tabulky uložíme. V části *Match* se pracuje s elementy grafu. Jeden výsledek prohledávání obsahuje seznam elementů, které odpovídají hledanému 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 a m jsou přirozená čísla větší než 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 části *Match*. Tedy tabulka má počet sloupečků 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 uchováváním výsledků, ale konkrétní řešení ponecháme na analýzu 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 hranatých závorek na třídě tabulky (`tabulka[i]`, kde `i` je index řádku). Třída poskytne metody pro přístup k elementům nebo výsledkům agregačních 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 funkce hranatých závorek na objektu tabulky dostaneme proxy třídu řádku 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 vynechat 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 prohledávání grafu jsou definované jazykem PGQL (sekce 1.2). Vzor se vždy skládá z posloupností vrcholů a hran. Na každý prvek posloupnosti se můžeme dívat jako na úložiště nějakého elementu grafu.

Vlastnosti prohledávání grafu:

- Výsledky prohledávání jsou podgrafy izomorfní 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) [2, kap. 4]. Na základě 1. a 2. kapitoly článku o distribuovaném zpracování PGQL dotazů [19] jsme vybrali algoritmus DFS, jelikož v průběhu prohledávání generuje menší množství mezivýsledků. 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` FIFO). Fronta se tak rychle zvětšuje. DFS naopak potřebuje znát sousedy pouze aktuálně prohledávaných vrcholů.

2.5.2 Hledaný vzor a finální výsledek

K aplikaci algoritmu musíme vytvořit strukturu (vzor) představující hledaný podgraf. Strukturu budeme chápat jako propojené posloupnosti tříd. Třída obsahuje informace specifikující vhodný element grafu, který ji má náležet. Cílem DFS je nalézt elementy grafu, které budou odpovídat hledaným posloupnostem. V průběhu DFS prohledávání si vzor pamatuje aktuální třídu, pro kterou DFS hledá vhodný element. V moment průchodu DFS přes nějaký element se ověří zda se jedná o vhodný element pro aktuální třídu. Pokud ano, třída nyní reprezentuje daný element, DFS z něj pokračuje v prohledávání a dojde k přestupu na další třídu v posloupnosti. Pokud ne, DFS se vrací na předchozí element v prohledávání, ze kterého vybere ještě neprozkoumaný element procházení.

Tvorba vzoru

Nyní musíme strukturu vytvořit. V sekci 2.3.1 jsme uvedli, že posloupnosti oddělené čárkou v *Match* části budou reprezentovány jako pole tříd obsahující informace o proměnných. Tedy 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 souvislou komponentu (popis tvorby vzoru a komponent je blíže popsán v sekci 3.4.5). Obecně ji vytvoříme propojením řetězců pomocí opakujících se proměnných. Tedy dva řetězce jsou propojeny právě tehdy, obsahují-li stejnou proměnnou. Propojením všech

takových řetězců vytvoříme souvislou komponentu. Souvislá komponenta představuje hledaný vzor. Strukturu tedy chápeme jako abstrakci procházení DFS. Problém vyvstane, pokud nám vzniknou dvě separátní komponenty z jednoho dotazu. Tento případ nastane 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 prohledávání dvou komponent. Stejný princip aplikujeme, pokud existuje vícero separátních komponent.

Příklad dotazu separátních komponent:

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

Finální výsledek

V *Match* části dotazu je součástí definice hledaného vzoru i specifikace proměnných. Proměnné jsou přístupné v ostatních částech dotazu. Finálním výsledkem prohledávání grafu bude pole elementů představující proměnné dotazu. Toto pole bude obsaženo ve struktuře vzoru. Jedna položka v poli odpovídá právě jedné proměnné. Opakující se proměnné nemají více položek v poli. Příkladem může být `match (x) -> (y) -> (x), (k) <- (p)`. Tento hledaný vzor obsahuje čtyři proměnné: `x`, `y`, `k` a `p`. Pole tedy obsahuje čtyři položky v pořadí daném prvním výskytem proměnné v dotazu. Výsledně první položka v poli představuje proměnnou `x`, druhá položka proměnnou `y`, třetí `k` a poslední `p`.

Bude existovat pouze jedno pole pro jeden vzor. Nebudou se vytvářet další, ale budou se měnit pouze vnitřní elementy, protože chceme omezit režii za tvorbu polí. Pole bude sloužit k ověření, jestli držíme totožné elementy v moment opakující se proměnné v průběhu prohledávání grafu. V moment nalezení celého podgrafu bude pole proměnných zaplněné a bude chápáno jako finální výsledek prohledávání, protože pouze proměnné jsou přístupné v jiných částech dotazu.

2.5.3 Průběh prohledávání grafu

K nalezení všech podgrafů v grafu potřebujeme z každého vrcholu spustit DFS. Při DFS se bude kontrolovat, jestli průchod odpovídá hledanému vzoru. Procházení vždy začíná vrcholem, následně se přistoupí k hraně daného vrcholu a pak koncovému vrcholu hrany. K procházení grafu máme navrženou strukturu z sekce reprezentace grafu 2.2. Pokud dojde k nalezení podgrafu, tak výsledek bude uložen způsobem z sekce 2.4.2. V našem případě tedy překopírování elementů z pole proměnných náležící vzoru.

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 se výsledek uloží do tabulky, protože tuto chvíli budeme vlastnit finální výsledek prohledávání. Zbavíme se tak nutnosti uchovávat mezivýsledky a následnému tvoření skalárního součinu.

2.5.4 Paralelizace prohledávání grafu

Nyní přistoupíme k analýze paralelizace prohledávání grafu. V paralelním řešení chceme použít co nejmenší počet synchronizačních primitiv. Ukládání vý-

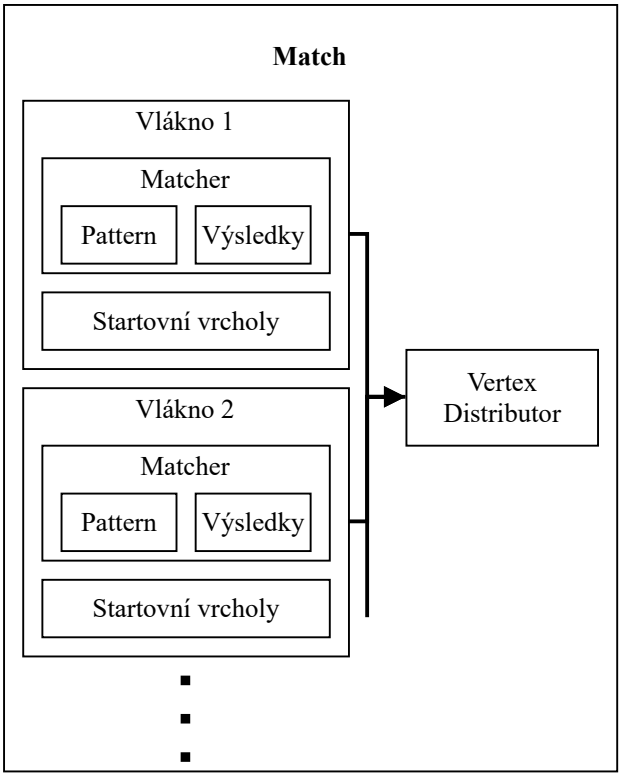
sledků do společné struktury by způsobilo značnou režii za synchronizaci. V ideálním případě bude probíhat prohledávání grafu lokálně, následně pak dojde k efektivnímu slévání výsledků.

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 (`VertexDistributor`), který jim bude přidělovat vrcholy grafu. Vlákno vždy zažádá `VertexDistributor` o určitý počet vrcholů, ze kterých spustí lokálně prohledávání a výsledky uloží do své tabulky. Nikdy nenastane situace, kdy dvě vlákna mají stejný startovní vrchol. Po vyčerpání všech vrcholů grafu prohledávání končí. V jednovláknovém 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. Děje se tak, protože reálné grafy nemají obecně rovnoměrné rozložení hran. Jedno vlákno by mohlo dostat vrcholy nacházející se v oblasti s množstvím hran, zatímco jiné vlákno by procházelo řídkou oblastí. Jelikož se rychle vyčerpaly startovní vrcholy, tak vlákno v řídké oblasti ukončí svou práci mnohem dříve něž vlákno první. Nyní se musí čekat na dokončení práce prvního vlákna.

2.5.5 Slévání výsledků prohledávání

Po dokončení prohledávání grafu je nutno vyřešit slévání 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 vyřazení překopírovávání všech výsledků vláken využijeme následující princip. Sloupeček tabulky bude tvořen polem polí fixní délky. Jeden sloupeček v programovacím jazyce C# může vypadat takto: `List<Element[FixedArraySize]>`. V kroku slévání nyní pouze překopírujeme odkazy na pole místo samotných výsledků. Avšak, pořád nám zůstává nutnost překopírovat výsledky v posledních polích 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 účely během implementace zkusíme zvolit prvně $n/\log_2 n$ (n = #výsledků prohledávání) pro jednovláknové zpracování a pro paralelní zpracování $(n/\log_2 n)/\#vláken$. $log_2 n$ odpovídá počtu alokací při plnění dynamického pole n položkami. Slévání bude probíhat opět paralelně. Nabízí se dva způsoby. Vlákno slévá pouze jeden totožný sloupeček všech výsledků vláken nebo dojde k dvoucestnému slévání výsledků vláken. Správné řešení ponecháme na dobu implementace.



Obrázek 2.4: Diagram paralelizace prohledávání grafu.

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. Výchozí pořadí je chápáno jako rostoucí. Potřebujeme určit jakým způsobem budou výsledky prohledávání 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 existují 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 [20]. 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 hranatých závorek na objektu tabulky 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 paměť na pole indexů. Po dokončení třídění se pole použije jako indexační struktura.

2.6.4 Optimalizace porovnání vlastností 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 vlastností. V takovém případě musíme přistoupit k tabulce `elType` (sekce 2.2), zjistit existenci přistoupené vlastnosti a následně přistoupit k její hodnotě. Danou situaci můžeme vyřešit částečně uchováváním hodnot výrazů. Budeme si pamatovat poslední porovnané řádky a jejich hodnoty výrazů. 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 prohledávání grafu mělo vygenerovat několik výsledků se stejným elementem v proměnné `x`. Počet takových výsledků zde odpovídá počtu hran vrcholů `x`. Pokud bychom porovnávali dané výsledky, museli bychom pro ně vždy vypočítat hodnoty výrazů, ačkoliv výsledky obsahují stejné elementy. Abychom předešli opakovanému výpočtu výrazů pro dané výsledky, vymysleli

jsme optimalizaci. Výsledky nejdříve porovnáme pomocí `ID` jejich elementů a teprve pak pomocí výrazů. Tímto vyřadíme opakovaný výpočet výrazů pro výsledky se stejnými elementy. Tedy využijeme dvou optimalizací. Budeme si uchovávat poslední hodnoty 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. Dochází zde k uchovávání výsledků lokálních pro vlákno a existence sdíleného úložiště by vytvořila souběh. 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 hodnotami výrazů. V průběhu implementace budeme muset najít vhodnou techniku ukládání, aby došlo ke zrychlení třídění.

2.7 Group by

Group by seskupuje výsledky prohledávání grafu podle uživatelem zadaných klíčů. Dva výsledky 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 ve dvou-dimenzionálním poli, tj. primitivní tabulce. Sloupeček představuje výsledky agregační funkce.

Jedné skupině pak přináleží řádek v daném poli. Nyní pokud budeme chtít výsledky funkcí skupiny, stačí přistoupit skrze přidělený index ke chtěnému výsledku.

Následuje ukázka možné implementace dle popisu výše v jazyce C#:

```
Bucket:
\\ Pole výsledků agregačních funkcí.
BucketResult[] groupAggFuncResults;
\\ Objekt úložiště.
class BucketResult {}
\\ Objekt definující typ ukládané hodnoty.
class BucketResult<T>: BucketResult { T value; }

List:
\\ Jednoduchá tabulka výsledků agregačních funkcí.
ListResults groupsAggFuncResults;
\\ Objekt primitivní tabulky výsledků agregačních funkcí.
class ListResults { ListHolder[] holders; }
\\ Objekt jednoho sloupečku tabulky.
class ListHolder {}
\\ Objekt definující typ sloupečku.
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 skupinu. Avšak, v druhém způsobu jsme nuceni přistupovat k výsledkům pomocí indirekce. Výhoda Bucket spočívá v jeho jednoduchém přemisťování (chápeme-li pole jako odkaz) 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 souběh, 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. Dále, byla-li hodnota již nastavena, protože musíme být schopni rozeznat prázdné úložiště po jeho vytvoření.

- `count/sum`: výsledek je počet výsledků/suma hodnot výsledků. Při výpočtu musíme přičítat hodnotu k hodnotě v úložišti.

- `avg`: výsledek je aritmetický průměr. Při výpočtu si budeme ukládat počet výsledků a sumu hodnot výsledků. Dochází zde tedy k navýšení počtu výsledků a přičtení nové hodnoty ke stávající sumě. Pro získání finální hodnoty dojde k vypočtení podílu sumy a počtu výsledků.

2.7.4 Jednovláknové zpracování

Po celou dobu 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 zbytečnou režii za porovnání při třídění, protože pro každé porovnání musíme vypočítat hodnotu výrazu. Z tohoto důvodu chceme využít strukturu, která bude interně používat hašovací 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 ukazatel na tabulku. Porovnání vyvolá získání proxy třídy a následně vyhodnocení 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 vypočítána haš na základě hodnot klíčů a následně použita při vložení do hašovací tabulky. Pokud už je výsledek obsažen v tabulce, tak se pro `value` (pole nebo index) pouze 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 haš hodnoty

Minulý přístup nám nabízí jednu optimalizaci k ušetření opakovaného výpočtu hodnoty klíčů. Hašovací tabulka při vložení dvojice v prvním kroku vypočte hodnoty klíčů a vypočte jejich haš. 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 můžeme využít předchozího výpočtu haš hodnoty. Budeme uchovávat hodnoty výrazů a následně je znovu použijeme při porovnání. Nicméně, pravděpodobně budeme potřebovat vytvořit závislost mezi objektem počítajícím haš hodnotu a objektem provádějícím porovnání. Stejný princip jsme použili při optimalizaci porovnání v sekci 2.6.4. Nastávají pro něj také stejné problémy v paralelním prostředí.

2.7.6 Paralelní zpracování

Chceme zvolit několik řešení s rozdílnou úrovní synchronizace, protože nevíme, které bude dosahovat nejrychlejších výsledků. Navíc nám větší počet řešení umožní lépe porovnat vylepšená řešení. Pro paralelní zpracování jsme vymysleli tři přístupy: **Global**, **Two-step** a **Local + dvoucestné slévání** (všechny budou schopny pracovat s úložišti Bucket i List). Každý z nich začíná rozdělením tabulky výsledků *Match* na ekvivalentní části. To si můžeme dovolit, neboť máme tabulku obsahující všechny výsledky prohledávání grafu. Každé vlákno dostane danou část ke zpracování. Tím zaručíme rovnoměrnost práce mezi vlákny. Samotná práce vláken pak závisí na zvoleném přístupu. Specifika vykonání jsou popsána v následujících sekcích.

2.7.7 Thread-safe agregační funkce

Ještě než přistoupíme k popisu jednotlivých řešení, je důležité si uvědomit nutnost vytvořit thread-safe verzi objektů implementující logiku agregačních funkcí. Pro každou funkci bude existovat ekvivalent thread-safe metody zpracovávající logiku funkce. Samotně pak vyvstává nutnost implementovat thread-safe metody pro slévání výsledků agregačních funkcí. Problém vyřešíme přidáním metod do objektů logiky agregačních funkcí. Pro úpravu logiky agregačních 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 přičtení.
- `avg`: tuto funkci budeme implementovat stejně jako původní funkci. Budeme si ukládat součet hodnot zpracovaných výsledků a jejich počet. Navýšení hodnot provedeme opět atomickým přičtením jako u funkcí `sum` a `count`. Finální hodnota pak bude podíl součtu a počtu hodnot.

2.7.8 Global Group by

Všechna vlákna budou provádět ekvivalent jednovláknového zpracování pomocí thread-safe paralelní mapy/slovníku (C# `ConcurrentDictionary`). Skupiny se vytváření globálně. Všimněme si nutnosti dvojité synchronizace. Prvně musí dojít k synchronizaci vytváření záznamů v mapě. Druhý krok synchronizace je nutný při zpracování agregačních funkcí. Paralelní mapa nám vyřadí souběh při vytváření nových záznamů skupin. V moment zpracování agregačních funkcí musí dojít k volání thread-safe verzí. Výhoda tohoto přístupu je, že zde nedochází ke slévání výsledků.

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 (C# `Semaphor`) omezující pohyb vláken v kritické sekci. Vlákno v moment nutnosti rozšíření zablokuje vstup přes semafor. Vlákno se na přístup pokusí projít skrze semafor. Pokud je mu vstup zabráněn, tak dochází k rozšiřování tabulky. V opačném případě aktualizuje výslednou hodnotu agregační funkce. 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 na konci této sekce.

2.7.9 Two-step Group by

Tento přístup probíhá ve dvou krocích. Lokální část následovaná globální částí. V lokální části pro každé vlákno běží kompletně identický ekvivalent jednovláknového zpracování. Globální část nastane v moment dokončení této práce. Místo

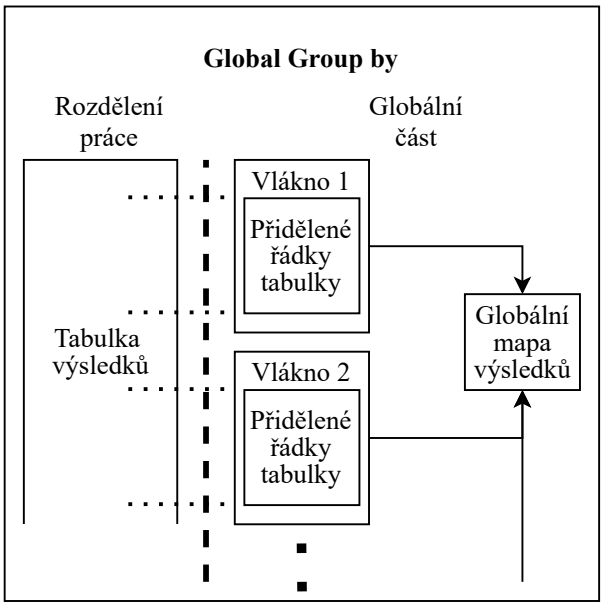
aby vlákno ukončilo běh, tak rovnou provede slévání svých výsledků do thread-safe paralelní mapy. To znamená, že vlákno nečeká na dokončení práce ostatních vláken, ale rovnou slévá své výsledky do paralelní mapy. Tento přístup kombinuje jednovláknové řešení společně s přístupem Global. Mínus je, že zde musí docházet ke slévání. Plus je, že v prvním kroku se využívají metody, které neobsahují synchronizaci. Problematická část je zde slévání výsledků s List úložištěm výsledků agregačních funkcí. První fáze nepředstavuje problém. Druhá představuje problém jako u Global řešení, tedy platí vše, co jsme pro něj zmínili. Můžeme zkusit implementovat stejné řešení jako u Global pomocí semaforu. Two-step *Group by* je znázorněn na obrázku 2.6 na konci této sekce.

2.7.10 Local + dvoucestné slévání Group by

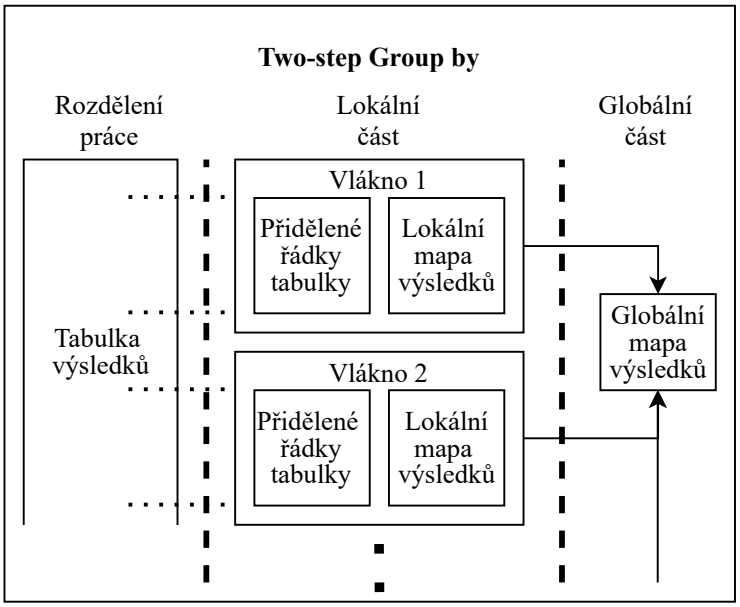
Přístup nepoužívá thread-safe metody ani paralelní mapu. Opět rozdělíme přístup na dvě části. V prvním kroku pro každé vlákno běží identický ekvivalent jednovláknového zpracování. Po dokončení se spustí dvoucestné slévání výsledků vláken. Slévání bude připomínat binární strom. Listy představují lokální seskupování vláken. Vnitřní vrcholy stromu představují slévání výsledků. Finální výsledek je vytvořen v kořeni. U slévání využijeme paralelního zpracování. Při slévání vždy jedno vlákno ukončí běh, druhé z dvojice provede slévání a postoupí ke kořeni. Hlavní výhody byly už zmíněny. Nevýhoda řešení je, že vlákna musejí čekat při slévání na dokončení práce druhého vlákna a teprve až pak může dojít ke slévání. Další nevýhoda nastane v případě velkého počtu vláken. Strom slévání v tuto chvíli bude hluboký a tedy výsledky vláken se budou často překopírovávat. Přístup je znázorněn na obrázku 2.7 na konci této sekce.

2.7.11 Paralelizace Single group Group by

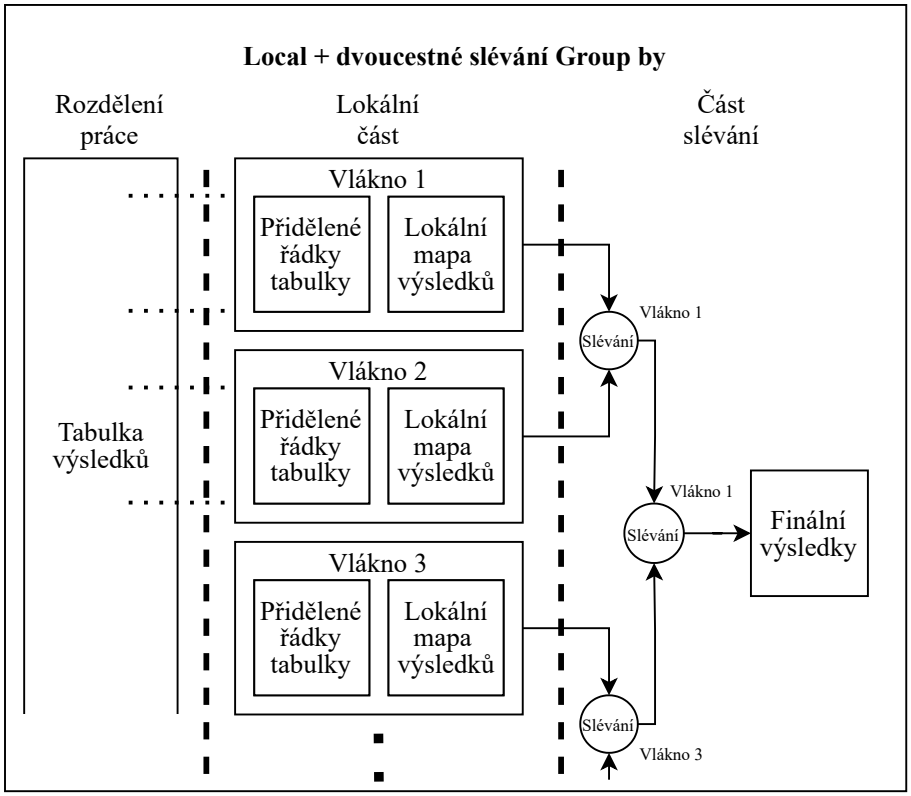
K paralelizaci *Single group Group by* využijeme jíž zmíněných principů. Konkrétně zde mají využití všechny tři. Problémem Global přístupu je synchronizace mnoha vláken na jednom místě. Ideálnější je využít lokálního zpracování zbylých dvou přístupů. Opět řešení rozdělíme na dva kroky. V prvním kroku každé vlákno provádí ekvivalent jednovláknového zpracování bez vytváření skupin, tj. počítá pouze agregační funkce pro jednu skupinu. Výsledky se následně musí slévat. K rozhodnutí, které řešení finálně použít použijeme fakt, že počet výsledků k slévání je roven počtu vláken. V takovém případě nebudeme implementovat paralelní slévání, ale pouze jedno vybrané vlákno provede sjednocení všech výsledků.



Obrázek 2.5: Diagram paralelizace Global *Group by*.



Obrázek 2.6: Diagram paralelizace Two-step *Group by*.



Obrázek 2.7: Diagram paralelizace Local + dvoucestné slévání *Group by* pro 4 vlákna.

Analyzovali jsme a navrhli řešení vykonání částí *Match*, *Group by* a *Order by*. Tímto jsme dokončili analýzu a návrh dotazovacího enginu. Daná analýza a návrh nám poskytnou odrazový můstek při analýze úprav pro vykonání částí *Group by* a *Order by* v průběhu prohledávání grafu.

2.8 Úprava enginu

Cílem úprav je poskytnout enginu schopnost provádět části *Group by* a *Order by* v průběhu prohledávání grafu části *Match*. Obecně to znamená, že v moment nalezení jednoho výsledku jej musíme okamžitě zpracovat. Pro *Order by* to znamená výsledek správně zatřídit do již setříděné posloupnosti výsledků. Pro *Group by* to znamená výsledek přidat do správné skupiny nebo pro něj skupinu vytvořit, navíc musíme pro něj zpracovat agregační funkce. Čili, výsledky v průběhu prohledávání *Match* části nebudeme ukládat do tabulky, která se po nalezení všech výsledků předá k dalšímu zpracování. Namísto toho navrhneme postup, kterým docílíme zpracování výsledků v moment jeho nalezení. Postupy ověříme vůči stávajícím řešením v kapitole Experiment 4 z hlediska doby vykonání. V ideálním případě docílíme zrychlení nebo pouze vyrovnání vůči řešením vykonávající *Order by* a *Group by* po dokončení prohledávání grafu. Z hlediska implementace to také znamená naprogramovat vylepšení tak, abychom byli schopni jednoduše přepínat mezi způsobem vykonání dotazu. Řešení tedy musí fungovat nezávisle na sobě. V prvním kroku úprav musíme získat obecný pohled na pozměněný způ-

sob vykonání dotazu. Následně v dalších krocích budeme navrhovat části dotazu konkrétněji.

2.8.1 Pohled na pozměněný způsob vykonání dotazu

V následujících sekcích popíše náš obecný pohled na zpracování dotazu. Budeme vycházet z sekce 2.4. V dané sekci jsme si definovali prioritu částí dotazu. Priorita určovala pořadí vykonání:

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

Match se provedl jako první. Následně se nalezené výsledky předávali dalším částem ve směru klesající priority. Nejvyšší priorita se nachází nalevo a nejnižší napravo. Pro naše potřeby úprav budeme nyní uvažovat pouze části *Match*, *Group by*, *Order by* a *Select*. Opět budeme o částech uvažovat jako o separátních objektech. Potřebujeme, aby *Match* část v moment nalezení výsledku jej předala k zatřídění části *Order by* nebo k seskupení části *Group by*. Po zatřídění/seskupení se opět pokračuje v prohledávání grafu, dokud se nenajde další výsledek a ten se zase předá k dalšímu zpracování. Můžeme si všimnout značné podobnosti s předchozím návrhem. Jediný rozdíl je ten, že místo předání všech výsledku najednou další části se předá výsledek pouze jeden. Na *Match* část se můžeme dívat jako na kontinuální generátor výsledků. V momentě nalezení se výsledek pošle dalším částem. Zbylé části pak pouze čekají na moment příchozího výsledku. Finální výsledky budou uchovány v objektech částí *Group by* nebo *Order by*.

Otázkou je, co je zde předávaný výsledek. Budeme předpokládat, že předávaný výsledek ke zpracování je pole proměnných definované v sekci hledaného vzoru 2.5.2. To znamená, že předané pole se nesmí měnit, protože pole náleží struktuře vzoru. Tedy pokud s ním chceme pracovat přímo, tak musíme vytvořit kopii. Když budeme hovořit o výsledku prohledávání, tak máme na mysli dané pole proměnných.

2.8.2 Order/Group by část jako bariéra

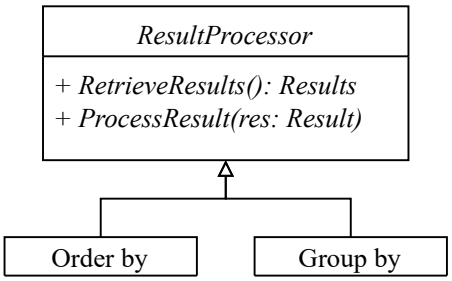
Problematická sekce návrhu je část *Select*. Pokud dotaz neobsahuje další části, pak v době nalezení výsledku jej stačí pouze vypsat. Nicméně, pokud je v dotazu obsažen *Group by* nebo *Order by*, pak se výsledky mohou vypsat až po dokončení třídění nebo seskupování. Tedy dané dvě části nám tvoří bariéru, skrze kterou se nemůže posílat výsledky dále. Jelikož máme navrhnout pouze vykonání částí *Order by* a *Group by*, tak budeme uvažovat pro *Match* a *Select* stejný návrh, jako v minulých sekcích. Tedy dotaz bude tvořen původní částí *Select* propojenou s části *Match*. Část *Match* pak propojíme s částmi *Order/Group by* a upravíme tak, aby jim byla schopna předávat jeden výsledek v moment jeho nalezení.

2.8.3 Změna objektů reprezentující části Order/Group by

Řekli jsme, že budeme opět považovat části dotazu za separátní objekty. Vytvoříme nové objekty částí *Group/Order by* reprezentující nový způsob zpracování. Abychom byli schopni pracovat souběžně i s původními objekty, tak potřebujeme upravit objekt *Match*. Reprezentanty částí *Order by* a *Group by* budou nyní

nové objekty (obrázek 2.8). Budou si implementovat logiku zpracování jednoho výsledku v metodě `ProcessResult`. Objekt zároveň obsahuje finální výsledky dotazu, proto potřebujeme metodu `RetrieveResults` na jejich získání.

Dotaz bude reprezentován opět řetězcem jako před úpravou zpracování. *Select* a *Match* budou tvořit propojení objetků na základě UML diagramů 2.1 a 2.2. Propojení objektu *Match* s novými objekty *Group by* a *Order by* navrhneme v dalších částech.



Obrázek 2.8: UML class diagram nových objektů reprezentující části *Group by* a *Order by*.

2.8.4 Propojení nových objektů s objektem Match

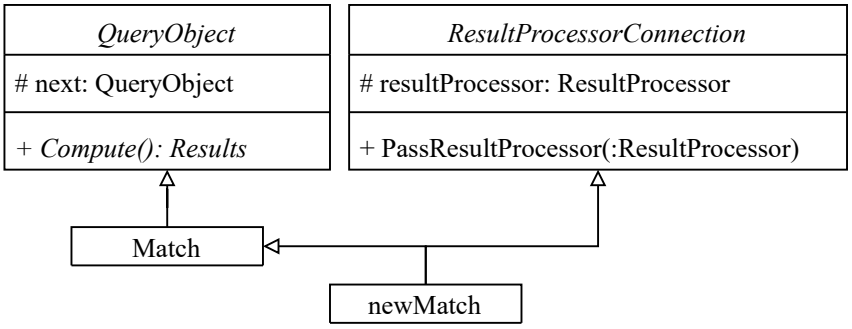
Vytvořili jsme nové objekty a nyní je musíme propojit s objektem *Match*. Objekty budou propojeny na dvou úrovních. První úroveň bude přímé spojení nových objektů s objektem *Match* a druhá úroveň bude propojení vyhledávání a zpracování výsledku.

Přímé spojení objektů

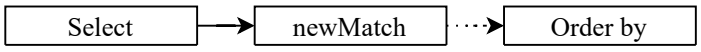
Popíšeme realizaci první úrovně propojení, tj. objekt *Match* drží odkaz na objekt *Group/Order by*. Abychom realizovali tuto úroveň, tak musíme upravit objekt *Match*. Objekt *Match* rozšíříme o chápání logiky nových objektů (obrázek 2.9). Stávající objekt *Match* jsme v naši představě pouze rozšířili a nevytvářeli kompletně nový objekt, protože propojení s *Select* objektem je realizováno stále původním způsobem. Finální propojení objektů je pak znázorněno na obrázku 2.10.

Propojení vyhledávání a zpracování výsledku

Musíme ještě navrhnout způsob předávání výsledků částem *Order/Group by* v průběhu prohledávání grafu. Způsob musí být použitelný pro paralelní zpracování, proto budeme rovnou přemýšlet nad paralelním řešením. V naši představě budeme vycházet přímo z původního návrhu z sekce 2.5. Definovali jsme si, že každé vlákno vlastní lokálně strukturu vzoru `Pattern`, objekt vyhledávání `Matcher` a tabulku výsledků. Finální výsledky ukládá do tabulky a po dokončení prohledávání grafu dojde k slévání tabulek. Zde upravíme část, kdy dochází k ukládání výsledků do tabulky. Vlákno bude držet odkaz na objekt *Group/Order by* a v moment nalezení výsledek předá pomocí volání metody `ProcessResult`. Metoda výsledek



Obrázek 2.9: UML class diagram rozšířeného objektu *Match*. Objekt nyní drží přímý odkaz na `ResultProcessor` a zároveň implementuje původní logiku objektu.



Obrázek 2.10: Diagram objektů upraveného vykonání představující části dotazu `select x match (x) order by x`. Plná šipka znázorňuje původní propojení a tečkovaná šipka představuje nové nové propojení.

zpracuje a po návratu z metody se pokračuje ve vyhledávání. To se opakuje dokud prohledávání grafu neskončí.

Zpracování dotazu bude finálně vypadat následovně. Na řetězci objektů se zavolá metoda `Compute`. První objekt v řetězci je *Select*. *Select* rekurzivně zavolá metodu na části *Match* a ta spustí vyhledávání. Část *Match* prohledává graf a výsledky předává části *Order/Group by* pomocí metody `ProcessResult`. Po dokončení prohledávání grafu volaná metoda `Compute` na objektu *Match* získá zpracované výsledky z další části pomocí volání metody `RetrieveResults`. Výsledky se tímto předají části *Select* k vypsání uživateli.

2.8.5 Alternativní řešení

Při výběru tohoto řešení jsme uvažovali ještě nad možností, ve které by vlákna vyhledávání pouze předávali pouze do fronty. Další část by pouze zpracovávala výsledky z fronty. Myslíme, že tohle řešení by bylo neefektivní v našem případě, protože by muselo docházet k synchronizaci fronty, tj. problém producent a spotřebitel. V našem řešení jsme synchronizaci při předávání výsledků zcela vynechali.

Vytvořili jsme nové objekty částí *Order by* a *Group by*. Propojili jsme původní objekty s novými a nyní přejdeme k analýze a návrhu samotných způsobů vykonání *Order by* a *Group by*.

2.8.6 Obecný model vykonání Order/Group by

Pokud bychom realizovali pouze jednovláknové vykonávání, tak bychom z předchozích sekcí měli již kompletní návrh. Pomocí volání metody `ProcessResult` dojde k předání výsledků a následnému zpracování. Problematická část je paralelizace vykonání. Musíme být schopni zpracovávat výsledky z množství vláken

v jeden okamžik. Je nutné, aby docházelo k synchronizaci. Ještě než přistoupíme ke konkrétním návrhům algoritmů zpracování, tak navrhneme obecný model paralelního zpracování. Vymysleli jsme dva modely. Modely **Half-Streamed** a **Streamed**. Modely jsme vybrali na základě množství využívané synchronizace a množství slévání výsledků. Chceme vytvořit větší množství řešení, abychom mohli lépe porovnat výsledky v kapitole Experiment 4.

Návrh Half-Streamed

Prvním modelem vykonání je model **Half-Streamed**. Myšlenkou tohoto návrhu je příchozí výsledky zpracovat ve dvou krocích. V prvním kroku každé vlákno výsledky zpracovává lokálně. Po dokončení vyhledávání dojde ke slévání výsledků vláken. Vycházíme z řešení **Two-step** a **Local + dvoucestné slévání** *Group by* z sekcí 2.7.9 a . Výhodou tohoto přístupu je využití zpracování bez nutnosti synchronizace vláken v prvním kroku. Nevýhoda je, že zde dochází ke slévání výsledků po dokončení zpracování.

Nyní jsme si uvědomili problém vznikající voláním metody `ProcessResult`. Logika zpracování náleží objektům *Group by* a *Order by*. Objekty zde musí vědět, že dochází nejdříve k lokálnímu zpracování. Proto jsme rozšířili metodu o další formální parametr `MatcherID`. Parametr symbolizuje `ID` vyhledávače (`Matcher`) vlákna. Objekty *Group by* a *Order by* pak budou vlastnit lokální výsledky vláken, které budou přístupné skrze `MatcherID`. Při volání metody `ProcessResult` dojde ke zpracování výsledku nad danými lokálními výsledky. Po dokončení dojde ke slévání výsledků vláken.

Návrh Streamed

Druhým modelem vykonání je model **Streamed**. Myšlenkou tohoto návrhu je příchozí výsledky zpracovat globálně. Vycházíme z Global zpracování *Group by* z sekce 2.7.8. Vytvoříme strukturu, která nám poskytne metody se synchronizací. Všechna vlákna v moment nalezení výsledku volají metodu `ProcessResult`, uvnitř které dojde k volání synchronních metod na struktuře. Výhoda tohoto přístupu je odstranění nutnosti slévání výsledků vláken. Nevýhoda je, že zde dochází k volání metod se synchronizací.

2.9 Úprava Order by

Cílem *Order by* by je setřídit výsledky prohledávání. V původním řešení jsme měli značnou výhodu, protože v moment začátku třídění jsme vlastnili všechny výsledky prohledávání (tj. kompletní tabulka výsledků). Místo třídění samotných řádků tabulky jsme třídili pole indexů. Na pole indexů pak stačilo aplikovat základní třídicí algoritmus. Nyní nevlastníme všechny výsledky. Výsledky jsou generovány a zpracovány postupně. V následujících sekcích navrhneme způsob zpracování *Order by* v průběhu prohledávání grafu.

2.9.1 Obecný princip zpracování

V naši představě zpracování budeme udržovat setříděnou posloupnost výsledků prohledávání. Při nalezení výsledku výsledek zatřídíme do již setříděné

posloupnosti. Obecně jsme se rozhodli použít následující princip. Bude existovat totožná tabulka výsledků prohledávání jako v původním řešení společně s indexační strukturou tabulky, protože nechceme přesouvat řádky tabulky. Indexační struktura bude opět obsahovat indexy řádků v tabulce. Výsledek *Order by* pak bude tabulka výsledku s indexační strukturou.

Výsledek prohledávání v moment nalezení je vložen na nový řádek tabulky výsledků. Následně je index nového řádku tabulky zatříděn do indexační struktury. Z tohoto principu budeme vycházet v našem návrhu. Nejdříve budeme analyzovat způsob jednovláknového zpracování a následně jednotlivé módy paralelního zpracování.

2.9.2 Jednovláknové zpracování

Určili jsme, že tabulka výsledků je totožná s původní. Výsledek je vložen na nový řádek tabulky a index je zatříděn do indexační struktury. Potřebujeme jen navrhnout způsob zatřídění indexu do indexační struktury. Uvažovali jsme nad dvěma základními přístupy. První kombinuje pole indexů s binárním vyhledáváním. Druhý přistup využívá vyhledávací stromy. Přístupy popíšeme a následně provedeme malý experiment.

Pole indexů + binární vyhledávání

Myšlenka přístupu je udržovat setříděné pole indexů. V moment nalezení výsledku je využito binární vyhledávání [20, str. 26] k nalezení vhodného místa vložení do pole. Prvek je vložen do pole. Pokud se na místě nacházel již nějaký prvek, prvek je posunut doprava. Pokud v poli není dostatek místa, tak je rozšířeno. Jedná se vlastně o algoritmus Insert sort (třídění vkládáním). Jediným rozdílem je, že k nalezení místa vložení se používá binární vyhledávání.

Problém zde představuje posouvání prvků v poli. Předpokládáme-li, že prvky posouváme napravo, tak v nejhorším případě vložením na začátek pole posuneme všechny prvky. V naši představě bychom chtěli problém řešit rozmístěním mezer mezi prvky v poli. Mezerou zde rozumíme prázdný záznam, tj. neobsahuje žádný index tabulky výsledků. Mezi každými dvěma sousedními prvky by existoval stejný počet prázdných záznamů. Vložení by bylo opět realizováno binárním vyhledáváním. V situaci posouvání prvků nyní stačí posouvat prvky do první mezery. Řešení však naskýtá spoustu otázek. Například neznáme ideální počet mezer mezi prvky a nevíme jak optimálně navrhnout binární vyhledávání na takovém poli. Navíc, zvyšováním počtu mezer se pole značně zvětšuje, tj. roste paměťová složitost.

Vyhledávací stromy

Druhý přístup využívá vyhledávací stromy [20, str. 177]. Chtěli bychom využít základní druhy vyhledávacích stromů jako binární vyhledávací stromy nebo (a, b)-stromy. Výhoda (a, b)-stromů je ta, že každý vrchol stromu obsahuje vícero klíčů, zatímco binární vyhledávací stromy drží pouze jeden klíč ve vrcholu. Pokud bychom implementovali řešení s vyhledávacími stromy, tak budeme ideálně chtít využít již stávajících knihoven.

Experiment pole vůči vyhledávacím stromům

K porovnání dvou přístupů jsme se rozhodli provést jednoduchý experiment. Cílem testu je otestovat rychlost vkládání prvků do vybraných datových struktur. Test bude částečně simulovat reálnou činnost zatřiďování prvků do struktury. Experiment jsme naprogramovali v jazyce C# jako konzolovou aplikaci Visual Studia 2019. Kód je součástí příloh A.5. Kód jsme přeložili pro platformu Windows 10 x64 využívající .NET Framework 4.8. Samotný hardware testovacího stroje je rozepsán v sekci metodiky 4.3.3. K otestování jsme vybrali dvě nativní struktury C#:

1. `List`: reprezentuje případ zatřiďování do pole binárním vyhledáváním.
2. `SortedSet`: reprezentuje případ binárního vyhledávacího stromu.

Struktury jsme zaplnili n náhodně generovanými prvky. Prvky byly generovány nativní třídou `Random` s inicializační hodnotou 100100 v rozsahu hodnot [10 000; `Int32.MaxInt`−10 000]. Struktury byly setříděny. Následně jsme do struktur vkládali m náhodně generovaných prvků (totožnou třídou `Random`) z dvou různých rozsahů:

1. Rozsah $[0; 10\,000]$ a je označen $front$. Umožní nám vkládat prvky na začátek setříděné posloupnosti. Sledujeme nejhorší případ pro pole, kdy dochází k posouvání všech prvků doprava.
2. Rozsah [10 000; `Int32.MaxInt`−10 000] a je označen $random$. Umožní nám sledovat vkládání prvků do náhodných pozic struktury.

Měřili jsme pouze část vkládání m prvků do struktur pomocí třídy `Stopwatch`. Měření jsme opakovali desetkrát pro každý rozsah a vybrali průměr hodnot. Při každém opakování byly struktury znovu sestaveny. Parametry n a m jsme volili tak, abychom byli schopni sledovat chování při zvyšování počtu vložených a vkládaných prvků.

Výsledky

	$n = 10^6$		$n = 10^7$		$n = 10^8$	
	`List`	`SSet`	`List`	`SSet`	`List`	`SSet`
random 10^2	0,0335	0,0001	0,464	0,0001	4,327	0,0001
random 10^3	0,3312	0,002	4,470	0,0011	44,148	0,0016
random 10^4	3,1092	0,006	43,999	0,0117	440,254	0,0155
front 10^2	0,883	0,0001	0,879	0,0001	8,673	0,0001
front 10^3	0,8729	0,0001	8,793	0,001	87,396	0,001
front 10^4	9,6823	0,0049	87,856	0,0119	885,589	0,0117

Pozn: `SSet` = `SortedSet`

Tabulka 2.1: Výsledky testu vkládání v sekundách `List` vůči `SortedSet`. Hodnota za názvem testu představuje parametr m.

Výsledky se nacházejí v tabulce 2.1. Z tabulky vidíme, že vkládání prvků do pole pomocí binárního vyhledávání značně zaostává, proto jsme se rozhodli upustit od myšlenky pole s mezerami.

Experiment (a, b)-strom vůči SortedSet

Na základě výsledků jsme se rozhodli naimplementovat (a, b)-strom [20, str. 190], u kterého jsme upravili definici na $b = 2a$. Pro experiment jsme určili parametr $a = 128$, protože jsme díky němu dosahovali nejrychlejších výsledků. Strom jsme porovnali vůči struktuře `SortedSet` při stejných testech, ale pouze pro nejvyšší řády počtu prvků stavby a vkládání.

Výsledky

	$n = 10^8$	
	`SortedSet`	$(128, 256)$-`strom`
random 10^4	0,0155	0,014
random 10^5	3,992	1,338
front 10^4	0,0117	0,002
front 10^5	1,483	0,483

Tabulka 2.2: Výsledky testu vkládání v sekundách `SortedSet` vůči $(128, 256)$-strom. Hodnota za názvem testu představuje parametr m.

	`SortedSet`	$(128, 256)$-`strom`
build $n = 10^8$	59,890	101,421

Tabulka 2.3: Výsledky testu stavby struktur v sekundách `SortedSet` vůči $(128, 256)$-strom.

Výsledky jsou zobrazeny v tabulce 2.2. Z tabulky vidíme, že při vkládání prvků je rychlejší (a, b)-strom. Pro finální rozhodnutí, kterou strukturu zvolíme k zatřiďování, jsme rozhodli provést poslední experiment. Tentokrát budeme měřit první část vkládání n prvků do prázdné struktury (test je označen *build*). Výsledky jsou v tabulce 2.3. Vidíme, že v experimentu je rychlejší `SortedSet`. Situaci si vysvětlujeme režií za operaci vložení do (a, b)-stromu, kdy dochází k častému překopírovávání prvků v moment vytváření nového vrcholu. Nicméně, v průběhu experimentu jsme sledovali paměťové vytížení procesu pomocí nativního nástroje `Memory Usage` Visual Studia 2019. Ukázalo se, že v průběhu testu `SortedSet` dosahovalo využití paměti 6,8 GB, zatímco (a, b)-strom pouze 2,1 GB. Daný jev je způsoben vytvářením nové třídy pro každý vložený prvek do struktury `SortedSet`. Finálně jsme se rozhodli zatřiďování realizovat pomocí (a, b)-stromu.

2.9.3 Ukládání prvků v (a, b)-stromu

V minulé sekci jsme rozhodli, že zatřiďování nalezených prvků prohledávání provedeme pomocí (a, b)-stromu. Musíme definovat chování, kdy nějaké dva prvky sdílí stejnou hodnotu třídění. Předpokládejme, že ve stromě existuje index s určitou hodnotou klíče. Prohledávání nalezne výsledek a uloží jej do tabulky a následně index řádku vloží k zatřídění do stromu. V takovém případě dva indexy se budou jevit jako shodné. Avšak, stále potřebuje mít indexy setříděné, čili v

moment shodnosti klíčů třídění budeme třídit prvky pomocí samotných hodnot indexů. To můžeme, protože každý index řádku existuje pouze jednou v tabulce. Čili nedojde k porušení setříděnosti.

Nyní jsme si uvědomili jednu možnou optimalizaci. Optimalizace bude fungovat v případech, kdy se hodnoty třídění často opakují. Každý záznam ve stromě bude obsahovat dvojici `(index, pole)`. `index` je index řádku tabulky. `pole` je datová struktura pole. Pokud vkládáme prvek, který ještě není ve stromě, tak se pro prvek vytvoří daná dvojice. Pole bude prázdné a položka `index` je index právě vkládaného řádku. Pokud vkládáme prvek, který už se nachází ve stromě, tak prvek pouze vložíme do příslušného pole. Tímto způsobem omezíme počet prvků ve stromě a tím i počet porovnání.

Nyní navrhneme způsob paralelizace třídění pro naše módy zpracování. V průběhu návrhu se budeme snažit využít jíž získané informace z předchozí sekce.

2.9.4 Řešení Half-Streamed

Samotný model zpracování **Half-Streamed** nám nabízí využít zatřiďování pomocí zmíněných stromů. V prvním kroku každé vlákno bude lokálně vytvářet svou tabulku výsledků a indexační strom. Problematická část je zde slévání výsledků. Informace zde jsou uloženy ve stromech. Všechny struktury bychom mohli průběžně iterovat a udržovat si minimum z aktuálních výsledků. Budeme vlastnit separátní pole, do kterého na jeho konec ukládáme minima v každém kroku iterace. Po dokončení iterace pole bude obsahovat výsledek slévání. Bohužel jsme nepřišli na jednoduchý způsob paralelizace tohoto řešení, proto jsme se rozhodli od něj upustit.

Místo toho využijeme již několikrát zmiňované dvoucestné slévání (použili jsme jej při paralelizaci *Group by* v sekci Local + dvoucestné slévání 2.8.6). V moment dokončení prohledávání všemi vlákny se vytvoří jedno pole a každé vlákno do něj překopíruje své výsledky. Zde budeme muset překopírovat proxy třídy řádků a nikoliv pouze jejich indexy, protože existuje zde mnoho tabulek a slévání pouhých indexů nedokážeme rozeznat, jaké tabulce přináleží. To znamená, že pole obsahuje posloupnosti výsledků vláken. Následně se posloupnosti slévají po dvojicích, jako ve zmiňovaném řešení *Group by*. Nevýhoda tohoto řešení je nutnost čekat na dokončení práce prohledávaní grafu všech vláken.

2.9.5 Řešení Streamed

Zatřiďování do globální struktury je mnohem komplikovanější problém. Avšak, stále zde využijeme již nevržené principy tabulek a stromů. Připravíme určité množství přihrádek. Každá přihrádka bude obsahovat tabulku výsledků prohledávání a indexační strom. Přihrádka bude přístupná pouze skrze zámek.

Třídění probíhá na základě klíčů zadaných uživatelem. Klíče jsou vlastnosti elementů grafu. Vlastnosti mají svůj typ (např. číselná hodnota `integer` nebo řetězec `string`). V moment implementace enginu typ vlastnosti je definován konkrétním typem programovacícho jazyka. Pro jazyk C# číselná hodnota může být typ `Int32` a řetězec typ `string`. Dané typy mají své rozsahy hodnot. Například pro `Int32` je rozsah hodnot [`Int32.MinValue`; `Int32.MaxValue`]. Rozsahem hodnot zde míníme uzavřený interval na množině celých čísel. Interval budeme

označovat anglickým značením, tj. $[a; b]$ pro uzavřený, kde a a b jsou celá čísla a $a \leq b$. Rozsah typu **prvního** klíče rozsekáme na části obsahující ideálně shodný počet prvků. Nyní každé přihrádce přiřadíme jednu část rozsahu. Přihrádky tedy budou tvořit ostré uspořádání.

Dále bude existovat objekt sdílený všemi vlákny, který pro každý výsledek určí přihrádku na základě jeho hodnoty prvního klíče. Tedy vlákno při zatřiďování přistoupí k sdílenému objektu. Ten na základě hodnoty prvního klíče výsledku určí jeho přihrádku. Vlákno uzamkne zámek dané přihrádky, následně vloží výsledek prohledávání do tabulky a index nového řádku uloží do indexačního stromu. Zámek se po zatřídění odemkne a vlákno pokračuje v prohledávání grafu. Jsme si vědomi, že rozdělování celého rozsahu typu programovacího jazyka není ideální, protože nám nic neříká o konkrétních hodnotách vlastností v grafu. Nicméně, daný postup chceme vyzkoušet, protože se dané hodnoty dají jednoduše pro graf vygenerovat. Pokud bychom zjistili, že daný přístup je dostatečně rychlý, tak bylo vhodné v budoucích rozšířeních zvážit vytvoření statistik rozsahu konkrétních hodnot vlastností.

Nyní musíme zodpovědět několik otázek. Musíme určit, kolik přihrádek budeme vytvářet. Jak budeme rozdělovat rozsahy klíčů a jak určíme správnou přihrádku pro výsledek prohledávání. Pro jednoduchost nebudeme uvažovat případ, kdy existuje pouze jedna přihrádka nebo práci vykonává pouze jedno vlákno. Nejdříve si definuje formálně základní zmíněná označení:

- P je množina všech přihrádek.
- $|P|$ je počet přihrádek, kde $|P| \geq 2$.
- p_i je přihrádka i, kde $i = 0, 1, 2..., |P| - 1$.
- t je počet vláken prohledávající graf, kde $t \geq 2$, $t \leq 64$ a t je mocnina dvou.
- $R = [a; b]$, kde a a b jsou celá čísla a $a \leq b$, je uzavřený interval rozsahu hodnot typu klíče třídění.
- R_i je uzavřený podinterval intervalu R, kde $i = 0, 1, 2, ..., |P| - 1$. Přihrádce p_i náleží podinterval R_i. Mějme podintervaly $R_i = [a_i; b_i]$ $a_i \leq b_i$ a $R_j = [a_j; b_j]$ $a_j \leq b_j$, pak $\forall i$ a $\forall j$, takové že $i < j$ a $i \neq j$ platí $a_i < a_j$, $b_i < a_j$, $a_i < b_j$ a $b_i < b_j$. Uvažujeme, že každý podinterval má alespoň jeden prvek, tedy R obsahuje dostatek prvků.

Počet přihrádek

Ideální počet přihrádek je těžké odhadnout. Obecně platí, čím více přihrádek tím je menší šance, že se dva vlákna setkají u jednoho zámku. Proto počáteční odhad přihrádek určíme jako $|P| = t^2$. Abychom nemuseli pracovat s obecným počtem přihrádek, omezili jsme počet přihrádek na 4096 ($t = 64$).

Rozdělování rozsahu

Rozhodli jsme se, že rozdělení přihrádek budeme provádět pouze podle prvního klíče třídění. Cílem je rozdělit rozsah typu programovacího jazyka na části stejné velikosti tak, aby tvořili ostré uspořádání dle poznámek výše. Pro jednoduchost

budeme přetvářet konkrétní rozsahy na rozsah $R = [0, n]$ $n \geq 0$, s kterým dále budeme pracovat. Dále přihrádce p_i přináleží po rozdělení část rozsahu R_i. Jako základní typy vlastností jsme v sekci vstupních dat 2.2.3 zvolili číselné hodnoty (`integer`) a řetězce (`string`). Nyní se pokusíme demonstrovat způsob rozdělení rozsahu typů v jazyce C# pro .NET Framework 4.8. Typ `integer` jsme zvolili jako typ `Int32`. Typ `string` jsme zvolili jako typ `string`. Musíme zmínit, že jsme v sekci vstupních dat omezili znaky vstupních řetězců pouze na hodnoty ASCII [0; 127].

Rozdělování typu Int32

Nejdíve vytvoříme rozsah $R = [0;\ n]$ pomocí úpravy rozsahu hodnot typu `Int32`. Následně dle upraveného rozsahu vytvoříme rozdělení přihrádek. Finálně pak sestavíme konkrétní rozsah náležící přihrádce. `Int32` je 32-bitový typ s rozsahem $I =$ [`Int32.MinValue`; `Int32.MaxValue`]. Označme `Int32.MinValue` jako I_{min} a `Int32.MaxValue` jako I_{max}. Abychom vytvořili rozsah $R = [0;\ n]$, tak přičteme ke každé hodnotě rozsahu I kladnou hodnotu I_{min}. Hodnotu označme I_{+min}. Tímto vytvoříme rozsah $R = [0;\ I_{+min} + I_{max}] = [0;\ 4\ 294\ 967\ 295]$ a maximální hodnotu tohoto intervalu označme R_{max}. Z tohoto rozsahu nyní jednoduše určíme velikosti částí k rozdělení původního rozsahu I. Počet hodnot v intervalu je $R_{max} + 1$. Velikost podintervalů je $d = (R_{max} + 1)/t^2$, za předpokladu, že máme t vláken. Tedy každá přihrádka bude obsahovat právě d hodnot, protože $R_{max} + 1$ je dělitelné námi definovanými t^2. Nyní zbývá určit jak budou vypadat samotné rozsahy přihrádek. Pro každou přihrádku p_i je rozsah $R_i = [i \cdot d; d + (i \cdot d) - 1]$, což odpovídá $I_i = [I_{min} + (i \cdot d); I_{min} + d + (i \cdot d) - 1]$.

K nalezení indexu i správné přihrádky nyní stačí použít celočíselné dělení. Předpokládejme, že nalezený výsledek má hodnotu prvního klíče třídění k. Pomyslná nula je zde hodnota I_{+min}. Pokud je $k \leq 0$, pak index přihrádky je $i = (I_{+min} - (-k))/d$. V opačném případě je $i = (I_{+min} + k)/d$.

Rozdělování typu string

Nyní musíme rozdělit rozsah typu `string`. Pracujeme pouze se základními znaky ASCII, ordinální hodnoty jsou v rozsahu [0; 127]. Ordinální hodnotu jednoho znaku pak budeme značit $[x]$, kde x je ordinální hodnota znaku, zároveň $0 \leq x \leq 127$. Pokud budeme mluvit o hodnotě znaku, tak máme namysli ordinální hodnotu znaku. Dále budeme postupovat jako v minulé sekci.

Abychom byli schopni definovat nějaký vhodný rozsah R jako v předchozí sekci, rozhodli jsme se provádět porovnání ordinální. Při porovnávání znaků budeme tedy uvažovat pouze jejich ordinální hodnoty a ne abecední pořadí. Znaky [0; 31] a [127] jsou řídící kódy, které se nevykreslují. Proto budeme považovat za adekvátní práci pouze s $128 - 33 = 95$ znaky, čili rozsah hodnot je $K = [32;\ 126]$. K získání rozsahu $R = [0;\ n]$ odečteme hodnotu znaku [32] od každé hodnoty rozsahu K. Výsledný rozsah je $K' = [0;\ 94]$. Znak [32] budeme tedy chápat jako pomyslnou nulu. Kdybychom vytvářeli přihrádky na základě jednoho znaku, tj. aktuální K', tak bychom měli problém rozdělit rozsah pro větší počty vláken. Proto jsme se rozhodli vytvořit rozsah na základě ordinálních hodnot dvou prvních znaků. Vytváříme vlastně pomyslný skalární součin $K' \times K'$. Uvažujeme tedy rozsah $R = [0;\ 95^2 - 1]$, protože máme 95 znaků a děláme skalární součin.

Hodnoty znaků jsou ale stále číslovány od 0, tedy -1. Označme počet hodnot v rozsahu jako $R_{max} = 95^2$.

Nyní můžeme postupovat stejným způsobem jako v minulé sekci. Velikost částí je $d = R_{max}/t^2$. Přihrádka p_i má rozsah $R_i = [d \cdot i; d + (d \cdot i) - 1]$. Určení přihrádky řetězce pak bude vypadat následovně. Označme ordinální hodnoty prvních dvou znaků řetězce jako $[x]$ a $[y]$, kde první je $[x]$. Pro získání indexu i náležité přihrádky stačí od obou hodnot znaků odečíst hodnotu [32]. Pokud znaky chybí přiřadíme jim hodnotu 0. Hodnoty po odečtení označme x' a y'. Jejich hodnotu q z rozsahu R vypočteme jako $q = ((x' \cdot 95) + y')$, protože každý znak se kombinoval s každým znakem. Následně $i = q/d$.

Nastává tady problém pro velký počet vláken. Uvažujme $t = 64$, pak $d = R_{max}/4096 = 2$. Jenže v moment výpočtu $i = q/2$ bychom pro hodnoty $q > 2 \cdot 4096$ získali indexy větší než počet přihrádek. Všimněme si, že počet problematických hodnot je zde vždy menší než počet přihrádek. Tedy řešením daného problému je hodnoty spadající za danou hranici rozmístit do prvních n přihrádek, které mají $i = 0, 1, ..., n - 1$. Tyto přihrádky budou mít $d' = d + 1$ a zbytek d. Výpočet i nyní bude vypadat následovně. Označme hodnotu horní meze rozsahu poslední přihrádky mající d' jako D_{max}. Pokud $q \leq D_{max}$, pak $i = q/d'$, protože prvních n rozsahů má o jednu hodnotu navíc. V opačném případě $i = n + ((q - D_{max})/d)$, jelikož od D_{max} jsou rozsahy původní velikosti a předcházelo jim n přihrádek. Přičítáme n, protože přihrádky jsou číslované od nuly.

Tímto jsme dokončili návrh vykonávání paralelního *Order by*. Dělení popsané v předchozích kapitolách jsme určili na základě jazyka C#. Dané principy se dají aplikovat i v dalších jazycích.

2.10 Úprava Group by

Group by má za úkol seskupit výsledky dle zadaných klíčů a vypočíst hodnoty zadaných agregačních funkcí. Při úpravě budeme využívat původní úložiště (List a Bucket) a logiku agregačních funkcí. Opět budeme při zpracování chtít použít hašovací tabulku jako původní řešení. Přejdeme rovnou k návrhu zpracování módů **Streamed** a **Half-Streamed**. Budeme uvažovat paralelní řešení, protože jednovláknové zpracování je určitý případ paralelního.

2.10.1 Řešení Half-Streamed

V tomto módě chceme pouze upravit původní řešení Two-step z sekce 2.7.9. Má totiž stejný průběh vykonání jako samotný model **Half-Streamed**. Oba pracují s myšlenkou práce ve dvou krocích. V prvním kroku dochází k lokálnímu vytváření skupin a po dokončení prohledávání grafu dojde ke slévání výsledků vláken. Vlákno po dokončení prohledávání nečeká na dokončení práce ostatních vláken, ale rovnou výsledky slévá do sdíleného úložiště. Zmiňovali jsme, že se jedná o paralelní mapu/slovník (C# `ConcurrentDictionary`). Nepoužijeme zde řešení Local + dvoucestné slévání, protože samotné slévání vytváří nutnost vláken čekat na dokončení práce jiného vlákna. Pomocí řešení Two-step se problému kompletně vyhneme.

Ukládání pouze reprezentantů skupin

K upravení původního řešení musíme pouze vyřešit problém chybějící tabulky výsledků, protože v moment nalezení výsledku je výsledek předán části *Group by* a není tak uložen do tabulky. Mohli bychom výsledky opět ukládat do tabulky. V moment nalezení je výsledek překopírován do tabulky na nový řádek a proxy třídu řádku využijeme k jeho zpracování. Daný způsob nám přijde neefektivní, protože cílem *Group by* je vytvořit skupiny. K vytvoření skupin nám stačí znát pouze reprezentant skupiny, tj. klíč v hašovací tabulce (proxy třída řádku v tabulce výsledků prohledávání). Nepotřebujeme znát všechny výsledky prohledávání, tedy nemusíme všechny výsledky kopírovat do tabulky. Abychom mohli pouze minimálně upravit vykonávání řešení Two-step, tak upravíme původní tabulku výsledků na základě tohoto poznatku. Tabulka bude nyní kromě samotných hodnot obsahovat položku držící odkaz na výsledek prohledávání. V moment nalezení výsledku se odkaz aktualizuje na daný výsledek a tabulka bude tento akt chápat jako přidání nového imaginárního řádku. Při přístupu k novému řádku se vytvoří obvyklá proxy třída řádku. Následný přístup k hodnotám řádku skrze proxy třídu vyvolá čtení hodnot z odkazu a nikoliv hodnot z tabulky. Proxy třída se použije ke zpracování výsledku prohledávání. Pokud byla proxy třída použita k vytvoření nové skupiny, tj. je vytvořen nový záznam v hašovací tabulce, tak je celý výsledek držený v odkazu překopírován do tabulky. Tímto si zachováme pouze reprezentanty skupin a díky používání odkazu nemusíme každý výsledek prohledávání kopírovat do tabulky.

Zbytek zpracování bude totožný s původním řešením Two-step. Úložiště hodnot agregačních funkcí bude implementováno rovněž totožně. Platí pro něj obdobné problémy, které jsme již zmínili v sekci popisu daného zpracování 2.7.9.

2.10.2 Řešení Streamed

Při tomto řešení budeme vycházet z přístupu Global *Group by* z sekce 2.7.8, protože zpracovává výsledky stejným způsobem jako model **Streamed**. Existuje zde jedna sdílená thread-safe struktura, ke které přistupují všechna vlákna. Jedná se opět o paralelní mapu/slovník. Předpokládejme nyní, že budeme chtít stále používat tabulky výsledků a stejná úložiště agregačních funkcí.

Problém synchronizace tabulky výsledků

Vyvstává zde opět problém neexistující tabulky výsledků. Opět chceme ukládat pouze reprezentanty skupin. Nicméně, problém je zde komplikovanější, protože v minulé sekci jsme tabulky vytvářeli lokálně. Tedy v moment slévání byly tabulky již finální. Aplikujeme-li nyní přístup lokálních tabulek, tak vyvstává další problém. Vlákna v průběhu vyvolávají porovnaní klíčů v hašovací tabulce. Klíč je zde proxy třída řádku v tabulce. Skrze proxy třídu vlákno přistoupí k elementům na řádku tabulky. V tuto chvíli však můžou přistupovat k hodnotám v tabulce i jiná vlákna a zároveň může docházet k rozšíření tabulky. Pokud by docházelo k rozšiřování tabulky, tak zde dojde k souběhu a přistoupení k nevalidní tabulce. Problém se dá řešit například použitím zámku. Při každém přístupu k tabulce by došlo k získání zámku, následnému vyvolání porovnání nebo rozšíření a nakonec

uvolnění zámku. Řešení však přináší nutnou režii za synchronizaci tabulky, proto jsme se od řešení pomocí tabulky rozhodli kompletně upustit.

Nový přístup ukládání výsledků

Úložiště hodnot agregačních funkcí budou pořád totožná. Místo ukládaní výsledků do tabulky vypočteme konkrétní hodnoty klíčů *Group by*. Hodnoty uložíme do pole, které následně použijeme jako klíč paralelní mapy. Samotné pole elementů použijeme pouze k výpočtu hodnot klíčů a k aktualizaci hodnot agregačních funkcí. Následně jej zahodíme. Paralelní mapa bude ve výsledku obsahovat pole hodnot klíčů a pole hodnot agregačních funkcí. Samotné hodnoty klíčů jsou v moment vytvoření statické, tedy nikdy nedojde ke změně hodnot klíče. Vlákna mohou jednoduše vyvolat porovnání klíčů bez nutnosti synchronizace.

Ačkoliv se může zdát, že vytvářením mnoha malých polí povede k značné paměťové zátěži. Musíme si uvědomit, že budeme uchovávat pouze ta pole, která byla vložena do paralelní mapy. V typickém případě platí, že počet finálních skupin je mnohonásobně menší než počet výsledků prohledávání. Tedy bude zde existovat pouze malá množina polí. Otázka může být, proč jsme nezvolili daný postup i u ostatních řešení. V původním řešení jsme drželi všechny výsledky v paměti a vytváření dalších polí spolu s překopírováváním výsledků by ještě víc vytížil paměťovou spotřebu. V **Half-Streamed** řešení jsme způsob nepoužili, protože výsledky vláken se vytvářeli lokálně. Kdyby každé vlákno našlo kompletně stejné skupiny, pak by existovala totožná pole pro všechna vlákna.

Další otázka je proč jsme se rozhodli uchovávat hodnoty výrazů a ne elementy grafu. Hlavní příčina ukládání elementů jsme popsali v sekci návrhu tabulky výsledků prohledávání 2.4.2. Tento problém zde odpadá, protože ukládáme malé množství výsledků. Navíc, v ostatních částech dotazu se mohou vyskytnout pouze výrazy klíčů *Group by* a agregační funkce. Tedy v moment výpočtu daných výrazů a agregačních funkcí jsme získali všechny možné hodnoty, ke kterým se může přistoupit v jiných částech dotazu.

Sloučení pole klíčů a pole úložiště agregačních funkcí

Nabízí se zde malá optimalizace za předpokladu, že budeme používat Bucket úložiště agregačních funkcí. Bucket úložiště je pole hodnot. Můžeme zde propojit pole hodnot klíčů a pole hodnot úložiště, protože oba pole pouze obsahují konkrétní hodnoty a logika zpracování agregačních funkcí je obsažena v objektu mimo úložiště. Finálně bude existovat pouze jedno pole, ve kterém prvních n hodnot představuje hodnoty klíčů a zbytek hodnot je považován za výsledky agregačních funkcí. Tímto dokážeme zmenšit počet vytvářených polí. Protože vycházíme z principu Global *Group by* 2.7.8, tak je zde problémem úložiště List. Rozhodli jsme se výsledně dané řešení s úložištěm List neimplementovat.

2.11 Úprava Single group Group by

Single group Group by je mód, ve kterém uživatel zadal v dotazu výpočet agregačních funkcí, ale nezadal část *Group by*. V tomto případě všechny výsledky prohledávání náleží pouze do jedné skupiny, pro kterou se počítají dané agregační

funkce. Stačí nám tedy navrhnout způsob vykonání pro naše dva módy **Streamed** a **Half-Streamed**. Budeme uvažovat pouze paralelní řešení, protože jednovláknové bude pouze určitý případ paralelního. Obecně si můžeme všimnout, že výsledky se nemusí ukládat do tabulky. Po nalezení výsledku dojde k aktualizaci hodnot agregačních funkcí a výsledek není dále potřeba. Ušetříme tedy spoustu paměti za tabulku výsledků. Samotný výpočet hodnot funkcí a úložiště hodnot budou totožné s původním řešením.

2.11.1 Řešení Half-Streamed

V tomto módu každé vlákno počítá hodnoty agregačních funkcí lokálně. Po nalezení výsledku dojde k aktualizování lokálních hodnot agregačních funkcí. V moment dokončení prohledávání grafu všemi vlákny dojde ke slévání výsledků vláken. Vlákno však nebude čekat na ukončení práce ostatních vláken, ale rovnou výsledky slévá do globální struktury. Výhoda tohoto řešení je vyřazení nutnosti používat thread-safe logiku agregačních funkcí v lokální části.

2.11.2 Řešení Streamed

V tomto módu existuje pouze jedno sdílené úložiště hodnot agregačních funkcí. Vlákna v moment nalezení výsledku aplikují thread-safe agregační funkce. Výhoda tohoto řešení je, že zde nedochází ke slévání výsledků. Problém zde může nastat ve chvíli, kdy existuje velké množství přistupujících vláken. Tyto přístupy mohou mít za následek značné zpomalení zpracování, kvůli synchronizaci.

Všimněme si také, že v jednovláknovém zpracování jsou módy totožné. V obou případech se nemusí použít thread-safe logika agregačních funkcí a existuje pouze jedno úložiště hodnot. Módy se tedy liší jen způsobem paralelního zpracování.

3. Implementace

Dokončili jsme analýzu, návrh a návrh úprav dotazovacího enginu. V této kapitole popíšeme implementaci. Začneme výběrem jazyka, obecným rozložením aplikace, popisu hlavních bloků a skončíme konkrétnějším pohledem na vybrané části aplikace.

3.1 Výběr jazyka

Aplikaci jsme se rozhodli implementovat v jazyce C# pro .NET Framework 4.8. K výběru jazyka jsme měli několik důvodů. Framework nabízí množství knihoven, modulů a základních datových struktur. Dále také poskytuje nástroje pro práci ve vícevláknovém prostředí. Uvažovali jsme ještě o jazyku C++, který nabízí množství technik a možností optimalizace k získání rychlosti při vykonávání aplikace. Nyní zmíníme, že hlavním cílem práce není vyvinou co nejrychlejší dotazovací engine, ale otestovat obecný koncept vykonávání *Group by* a *Order by* v průběhu prohledávání grafu. Myslíme, že tento koncept se dá implementovat v každém jazyce. Navíc, v průběhu analýzy jsme si zkoušeli již implementovat určité koncepty v daném jazyce C#, abychom měli lepší přehled o způsobech vykonání. Z tohoto důvodu jsme měli určité části již implementovány. Výsledně jsme se rozhodli z výše zmíněných důvodu použít C# pro .NET Framework 4.8.

3.2 Značení módů

Určili jsme, že engine bude pracovat v několika módech, které uživatel bude moct měnit. Jsou to módy:

- **Normal:** reprezentuje původní způsob vykonávání. V prvním kroku dojde k prohledání grafu a uložení výsledků do tabulky. Teprve po dokončení dojde vykonání *Group by* a *Order by*.
- **Half-Streamed:** Reprezentuje upravené vykonávání. Tento mód v paralelním vykonání ukládá výsledky nejdříve lokálně a po dokončení dojde ke slévání.
- **Streamed:** Reprezentuje upravené vykonávání. Tento mód v paralelním vykonání zpracovává výsledky globálně.

Samotná individuální řešení se pro upravené módy **Half-Streamed** a **Streamed** liší pouze v paralelním vykonávání. V určitých případech nastane, že i jednovláknová řešení jsou rozdílná. V takovou chvíli na to upozorníme. V průběhu této kapitoly se budeme držet tohoto značení.

3.3 Rozložení aplikace

Při implementaci aplikace jsme vycházeli z analýzy a návrhu předchozí kapitoly. Aplikaci jsme vyvíjeli jako projekt konzolové aplikace pro .NET Framework

4.8 v prostředí Visual Studia 2019. Samotný projekt je rozdělený na tři hlavní řešení:

- `QueryEngine`, který představuje implementaci dotazovacího enginu.
- `HPCsharp` [21] je doprovodné řešení, které poskytuje sadu mnoha výkonných jednovláknových i paralelních algoritmů.
- `Benchmark`, který jsme implementovali pro porovnání upraveného a původního způsobu zpracování dotazu.

V průběhu celé kapitoly se budeme věnovat pouze řešení `QueryEngine`, protože obsahuje hlavní část práce. Řešení `Benchmark` je popsáno podrobněji v kapitole Experiment 4., ve které se věnujeme porovnání implementovaných řešení. Z `HPCsharp` řešení jsme využili pouze určité algoritmy při implementaci části *Order by*. Samotná implementace algoritmů lze nalézt v odkazu citovaného zdroje. Nyní popíšeme podrobněji rozložení řešení `QueryEngine`.

3.3.1 Rozložení řešení QueryEngine

Řešení `QueryEngine` neobsahuje další podřešení, ale pouze adresáře. Níže popsané adresáře rozdělují engine na hlavní části výstavby z kapitoly analýzy. Hlavní adresáře jsou:

- *DB*: obsahuje objekty grafových elementů, struktury reprezentace grafu, objekty vlastností elementů a objekty k načítání grafových dat.
- *DataFiles*: obsahuje datové soubory, které se při překladu projektu ve Visual Studiu překopírují k binárním souborům.
- *Parser*: obsahuje metody načítání uživatelského dotazu, definované tokeny, objekty syntaktického stromu a objekty k procházení daného stromu.
- *Query*: obsahuje objekty zpracování dotazu. Obecně představuje část, která vykonává *Group by* a *Order by* po dokončení prohledávání grafu.
- *QueryStreamed*: obsahuje objekty upraveného zpracování dotazu. Obecně představuje část, která vykonává *Group by* a *Order by* v průběhu prohledávání grafu. Adresář částečně kopíruje strukturu adresáře *Query*. Pokud jsou názvy složek stejné znamená to, že objekty ve složce rozšiřují právě objekty ze stejnojmenné složky uvnitř *Query*. Obsahuje řešení pro **Streamed** i **Half-Streamed** módy.

3.4 Programátorská dokumentace

V této sekci popíšeme postupně klíčové objekty, položky a způsoby vypracování aplikace.

3.4.1 Reprezentace grafu

V této sekci popíšeme implementaci hlavních objektů, které jsme použili k reprezentaci grafových dat. Při implementaci jsme postupovali dle návrhu z sekce analýzy 2.2.

Třída Element

Všechny druhy elementů v grafu (vrchol a hrana) dědí od abstraktní třídy `Element`. Představuje obecný základ všech elementů. Hlavní vlastnosti jsou:

- `int ID` je unikátní identifikátor elementu. Definuje jej uživatel ve vstupním souboru. Není to vlastnost v Property grafu.
- `Table Table` odkaz na typ v Property grafu.
- `int PositionInList` pozice v `List<T>`, kde `T` je potomek třídy `Element`, protože každý potomek je obsažen ve vlastním poli. Určili jsme v analýze, kvůli jednoduché možnosti iterace elementů. Pole jsou pak obsažená v třídě `Graph`.

Potomek abstraktní třídy je třída `Vertex` (vrchol) a abstraktní třída `Edge` (hrana). `Edge` přidává novou položku `Vertex EndVertex`, která odkazuje na koncový vrchol hrany. Z dané třídy vznikají konkrétní potomci `InEdge` a `OutEdge`. Hrany jsou orientované a spojují dva vrcholy. Mějme hranu z vrcholu x do y. Položka `EndVertex` pro `OutEdge` zde odkazuje na vrchol y. Pro `InEdge` to je x. Tedy pro každou definovanou hranu v grafu existují obě instance tříd. Instance sdílejí `ID`, ale záznam v `Table` je pouze jeden, tedy sdílejí i hodnoty vlastností. `Vertex` přidává položky dvou dvojic indexů, o kterých jsme mluvili v analýze. První dvojice je `int OutEdgesStartPosition` a `int OutEdgesEndPosition`, které označují rozsah hran v poli `OutEdge` náležících vrcholu. Ekvivalentně pro typ `InEdge`. Celkově tedy třída `Graph` bude obsahovat pole `List<T>` pro každého neabstraktního potomka třídy `Element`.

Třída Table

Třída `Table` určuje typ štítku. Hlavní vlastnosti jsou:

- Položka `string IRI` je název typu.
- Položka `Dictionary<int, int> IDs`, kde klíčem je `ID` elementu a hodnota je pozice hodnot vlastností ve struktuře, která je obsahuje (tj. třídě `Property`).
- Položka `Dictionary<int, Property> Properties`, kde klíč je `ID` vlastnosti a hodnota je třída reprezentující vlastnost.
- `bool TryGetPropertyValue<T>(int id,int propID, out T retValue)` je metoda, která se pokusí získat hodnotu vlastnosti `propID` pro daný element s daným `id`. Hodnota se vrací v `retValue`. Úspěšně dokončená vrací `true`, jinak `false`. Pro získání hodnoty vlastnosti tedy musíme znát její typ.

Třída Property

Samotné vlastnosti jsou reprezentovány abstraktní třídou `Property`, která má svůj název `string IRI`. Z třídy vznikají abstraktní generické třídy `Property<T>`, kde `T` je typ hodnot vlastnosti. Třída obsahuje pole hodnot `List<T> propHolder`. Z třídy pak vznikají konkrétní třídy:

- `StringProperty`, kde `T` je `string`. Odpovídá typu `string` ve vstupním JSON schématu.
- `IntProperty`, kde `T` je `Int32`. Odpovídá typu `integer` ve vstupním JSON schématu.

V analýze jsme se rozhodli implementovat pouze tyto dva typy. Což znamená, že v celém enginu bude možno pracovat **pouze s těmito dvěma typy**. Tyto třídy mají metodu `void ParsePropFromStringToList(string strProp)`, která se používá při načítání hodnot ze vstupních souborů. Převede hodnotu `strProp` na svůj typ `T` a uloží na konec pole `propHolder`. Každý element v `Table` má svou hodnotu vlastnosti v poli na pozici `IDs[Element.ID]`. Třídy vlastností se vytvářejí pomocí třídy `PropertyFactory`, která implementuje Factory metodu [17, str. 107] `Property CreateProperty(string token, string name)`. Kde `token` je typ vlastnosti a `name` je její název.

Třída Graph

Třída `Graph` pak reprezentuje celý graf. Můžeme se dívat na ni spíše jako na objekt držící data grafu a ne objekt se složitou logikou. Načítá grafová data během inicializace. Zároveň dělá kontrolu načtených vlastností a jejich typů během načítání. Obsahuje pole všech typů elementů. Tedy `List<Vertex>`, `List<InEdge>` a `List<OutEdge>`. Obsahuje dále:

- `Dictionary<string, Table> nodeTables` všechny typy vrcholů Property grafu. To samé pro hrany v položce `edgeTables`.
- `Dictionary<string, Tuple<int, Type> > labels` mapa, kde klíč je název vlastnosti a `int` je její přiřazený unikátní identifikátor, abychom nemuseli v enginu používat řetězce jako `ID` vlastnosti. `Type` je pak typ vlastnosti, slouží pro kontrolu, protože dvě stejnojmenné vlastnosti musí mít stejný typ.

3.4.2 Čtení vstupních souborů

Máme implementovaný graf a nyní jej potřebujeme načíst. V této sekci popíšeme implementaci načítaní vstupních grafových dat. Budeme vycházet z kapitoly analýzy sekce 2.2.3.

Vstupní soubory

Rozhodli jsme se použít totožný vstupní formát dat jako při analýze. Při spuštění program očekává čtyři soubory ve složce *DataFiles*. Dva soubory se zmiňovaným JSON formátem *EdgeTypes.txt* (schéma hran) a *NodeTypes.txt* (schéma

vrcholů), které definují typy Property grafu a jejich vlastnosti. Další dva soubory *Edge.txt* (data hran) a *Nodes.txt* (data vrcholů) obsahují samotná data s mezerami jako oddělovače. Soubory musí mít přesně stejné názvy. Jediná úprava formátu je ta, že hrany v datovém souboru musí být seřazeny podle `ID` počátečního vrcholu hrany tak, jak jsou vrcholy seřazeny v jejich datovém souboru. To znamená, že pokud máme v souboru vrcholů za sebou vrcholy s `ID` 1, 2 a 3, tak soubor hran musí nejdříve obsahovat hrany začínající vrcholem s `ID` 1, pak s `2` apod. Důvodem je zjednodušení procesu načítání.

Načítání vstupních souborů

Načítání je implementováno následovně. `interface Creator<T>` s metodou `T Create()` je rozhraní pro tvorbu objektů `T`. Rozhraní bude implementovat třída `CreatorFromFile<T>`, která symbolizuje tvorbu objektu `T` postupným čtením souboru. Třída očekává při inicializaci rozhraní `IReader` čtoucí vstupní soubor metodou `string Read()`, která přečte vždy určitý úsek souboru. Dále rozhraní `IProcessor` vytvářející iterativně `T` na základě poskytnutých částí souboru. Protože text je čtený po částech, tak `IProcessor` implementuje návrhový vzor State [17, str. 305]. V našem případě nejdříve zpracujeme soubory schémat pomocí třídy `TableDictProcessor`, která vytváří třídy `Table`. Následně čtením datového souboru *Nodes.txt* vytvoříme třídou `VerticesListProcessor` pole vrcholů. A posledně čtením datového souboru *Edges.txt* vytvoříme dva pole `InEdge` a `OutEdge` třídou `EdgeListsProcessor`. Rozhraní `IProcessor` implementují všechny tyto třídy. Samotné čtení souborů je vyvoláno při inicializaci třídy `Graph`.

3.4.3 Načítání uživatelského dotazu

V této sekci popíšeme implementaci načítání uživatelského dotazu. Při načítání dotazu jsme vycházeli z sekce analýzy 2.3. Nejdříve dojde k tokenizaci a následně vytváření syntaktického stromu.

Třída Tokenizer

V prvním kroku dojde k tokenizaci uživatelského dotazu třídou `Tokenizer`. Uživatel zadá aplikaci svůj dotaz a třída provede tokenizaci. Výstupem tokenizace je pole `List<Token>`, které obsahuje všechny nalezené tokeny. Tokenem zde myslíme `struct Token`, který obsahuje dvě položky. První je `TokenType type`, což je typ tokenu. Druhá je `string strValue`, která obsahuje hodnotu tokenu, pokud se jedná o token `Identifier`. Tomu odpovídá například token názvu proměnné. Pole tokenů se předá statické třídě `Parser`.

Třída Parser

Třída postupně z tokenů vytváří syntaktické stromy každé hlavní části dotazu (*Match*, *Select*, *Order by* a *Group by*). Syntaktická analýza se vyvolá veřejnou metodou `Parse(List<Token> tokens)` a probíhá po částech. Každá hlavní část dotazu má svou separátní metodu. Například `ParseMatch(ref int position, List<Token> tokens)` parsuje část *Match*. Během syntaktické analýzy tokenů se

rekurzivně volají další metody. Při rekurzi se používá parametr `position`, který udržuje pozici aktuálně zpracovávaného tokenu.

Syntaktické stromy jsou tvořeny potomky abstraktní třídy `Node`, která implementuje návrhový vzor Visitor [17, str. 331], tj. definuje metodu `Accept<T>(...)`. Rozhraní `IVisitor<T>` implementuje druhou část vzoru, tj. metodu `Visit(...)`, kde vstupem ja každý neabstraktní potomek třídy `Node`. Každá část dotazu má svůj objekt implementující `IVisitor<T>`, například části *Match* přináleží objekt `MatchVisitor`. Parametr `T` je zde návratová hodnota procházení syntaktického stromu. Výstupem třídy `Parser` je množina všech vzniklých syntaktických stromů. Samotná tokenizace a syntaktická analýza se provádí při inicializaci třídy `Query`. Procházení syntaktických stromů je rovněž prováděno při inicializaci dané třídy.

3.4.4 Reprezentace dotazu

Celý dotaz jsme reprezentovali třídou `Query`. Exekuční plán a samotné zpracování pak bude odpovídat popisu z sekce analýzy 2.4.

Třída Query

Třída reprezentuje celý dotaz. Obsahuje všechny struktury, které se využívají pro vykonání dotazu. Objekt je používán uživatelem. Poskytuje statické veřejné metody `Query Create(...)` a `void Compute()`. Metoda `void Compute()` spustí vykonání dotazu. Metoda `Query Create(...)` vytváří dotaz. Daná metoda dostává množství argumentů, vypíšeme ty hlavní:

- `string/TextReader inputQuery`: definuje dotaz uživatele, který se má vykonat. `string` zde reprezentuje vstup jako řetězec. `TextReader` představuje vstup z konzole.
- `QueryMode mode`: definuje mód vykonávání, který se má provádět (sekce 3.2).
- `Graph graph`: definuje graf, nad kterým se má dotaz provést.
- `ThreadCount threadCount`: definuje počet vláken, které se mají využít při vykonávání.
- `GrouperAlias grouperAlias`: definuje řešení, které se má použít při vykonávání *Group by*.
- `SorterAlias sorterAlias`: definuje řešení, které se má použít při vykonávání *Order by*.

Při zavolání metody dojde k tokenizaci `inputQuery` a kontrole všech argumentů metodou `CheckArgs`. Zkontrolované argumenty a pole `List<Token>` se předají privátnímu konstruktoru třídy `Query`. Upravené módy sdílí konstruktor. Mód **Normal** má separátní konstruktor. Uvnitř obou konstruktorů dochází k syntaktické analýze pole tokenů třídou `Parser`. Výstupem jsou stromové struktury hlavních částí dotazu, které jsou dále použity k inicializaci privátních položek a exekučního plánu:

- `VariableMap variableMap` je seznam proměnných vyskytujících se v dotazu. Seznam obsahuje jejich přidělený identifikátor a typ, pokud byl definován.
- `QueryObject query` je exekuční plán. Obsahuje řetězec objektů, které postupně vykonávají dotaz.
- `QueryExecutionHelper qEhelper` obsahuje informace o způsobu vykonání dotazu.
- `QueryExpressionInfo exprInfo` obsahuje všechny výrazy (expressions) v dotazu.

Po inicializaci, v moment volání metody `Compute()` dojde k vyvolání metody `Compute(...)` na položce `query`. Tímto se spustí vykonávání dotazu.

Třída QueryObject

Jedná se o abstraktní třídu. Každá hlavní část dotazu je reprezentována potomkem dané třídy (`MatchObject`, `SelectObject`, ...). Třída definuje rozhraní exekučního plánu. Obsahuje:

- Položku `QueryObject next`, která propojuje objekt s dalším objektem.
- Metodu `void Compute(out ITableResults r, out GroupByResults g)`, která rekurzivné volá stejnou metodu na dalším objektu v `next`. Každý potomek třídy si implementuje vlastní logiku zpracování této metody. Všimněme si, že tato metoda definuje rozhraní pro předávání výsledků zpracování. `ITableResults` definuje obecné rozhraní tabulky výsledků prohledávání, pokud není zadáno *Group by*. `GroupByResults` definuje formát výsledků *Group by*.
- Metodu `void AddToEnd(QueryObject queryObject)`, která připojí poskytnutý objekt na konec řetězce.

Konkrétní potomci třídy jsou vytvářeny v konstruktoru třídy `Query`. Navíc, každý potomek očekává v konstruktoru syntaktický strom. Uvnitř konstruktoru dojde k vytvoření adekvátního `IVisitor<T>` objektu. Návratová hodnota `T` se využije ke konstrukci privátních objektů pro vykonání dotazu.

3.4.5 Match

V této sekci popíšeme implementaci části *Match*. Prvně popíšeme konstrukci vzoru a následně objekty algoritmu prohledávání. Budeme vycházet z sekce analýzy 2.5.

Třída MatchObject

Třída `MatchObject` reprezentuje *Match* část dotazu. Je potomkem abstraktní třídy `MatchObjectBase`, která dědí z `QueryObject`. Obecně třída `MatchObject` obsahuje odkaz na tabulku výsledků prohledávání `MatchFixedResults` a odkaz

na objekt DFS algoritmu prohledávání `DFSParallelPatternMatcher`. V konstruktoru třídy se vytváří vzor a objekt prohledávání. Objekt prohledávání obdrží vzor a tabulku pro ukládání výsledků. Třída dědí z `MatchObjectBase`, protože objekt části *Match* upraveného zpracování používá stejnou metodu ke kontrole vzoru uživatele (metoda `ParsedPatternCorrectness`). Metoda kontroluje uživatelem zadaný vzor, jestli splňuje podmínky jazyka PGQL. Metoda očekává na vstupu pole tříd, které je výstupem procházení syntaktické stromové struktury objektem `MatchVisitor`.

Třída ParsedPattern

`MatchVisitor` vytváří procházením stromu pole tříd `List<ParsedPattern>`. Třída `ParsedPattern` reprezentuje jednu vyhledávací posloupnost *Match* části (např. `(x) -> (y)`). V poli je tolik tříd, kolik je v dotazu posloupností oddělených čárkou. Třída obsahuje:

- Pole abstraktních tříd `List<ParsedPatternNode> pattern`. Abstraktní třída reprezentuje jeden hledaný element v posloupnosti, tj. hrana (`-[e]>`) nebo vrchol (`(x)`). Obsahuje položky `string Name`, pokud je element označen proměnnou (např. vrchol `(x)`), a `Table Table` pokud je definován typ elementu (např. vrchol `(:Type)`). Potomci specifikují konkrétní případy vrcholů a hran. Například pro posloupnost výše `(x) -> (y)` bude pole obsahovat tři třídy. `pattern[0]` je třída vrcholu `x`, `pattern[1]` je třída `OutEdge` hrany a `pattern[2]` je třída vrcholu `y`.

- Položku `string splitBy`, která označuje jméno proměnné, podle které posloupnost budeme dělit na dvě posloupnosti v průběhu vytváření objektu vzoru. Výsledkem dělení budou tedy dvě třídy `ParsedPattern`. Část před proměnnou, podle které dělíme, bude tvořit posloupnost převrácenou. To znamená, pokud máme posloupnost tříd `ParsedPatternNode (x) -> (y) -> (z)` a dělíme podle `(y)`, pak výsledkem dělení jsou posloupnosti `(y) <- (x)` a `(y) -> (z)`.

- Metodu `bool TryFindEqualVariable(ParsedPattern p, out string n)`, která vrátí název první sdílené proměnné s posloupností `p`, pokud nějaká existuje. Pokud existuje, metoda vrací `true` a název v `n`.

- Metodu `void TrySplitParsedPattern()`, která zkusí provést rozdělení posloupnosti. Rozdělení se nemusí provést, pokud je rozdělováno podle první položky `List<ParsedPatternNode>`. Pokud je rozdělováno podle poslední, posloupnost se pouze převrátí.

Z daného formátu se během vytváření struktur *Match* části vytvoří finální hledaný vzor `DFSPattern` s pomocí zmíněný metod a položek.

Třída DFSPattern

Třída reprezentuje hledaný vzor (obrázek 2.4 blok `Pattern`). Konstruktor dostane pole `ParsedPattern`. V analýze jsme řekli, že se z posloupností vytvoří

souvislé komponenty. Aplikováním výše zmíněných metod nalezneme sdílené proměnné. Pole, která sdílejí proměnnou, seskupíme k sobě a samotné posloupnosti rozdělíme pomocí položky `splitBy`. Výsledkem bude pole posloupností a souvislé komponenty v něm budou obsaženy postupně za sebou. Příklad seskupení a rozdělení:

```
Původní List<ParsedPattern> a uvnitř List<ParsedPatternNode>:
[[(x), ->, (z)], [(r), ->, (q)], [(y), ->, (x), ->, (w)]]
Pole po zpracování:
[[(x), ->, (z)], [(x), <-, (y)], [(x), ->, (w)], [(r), ->, (q)]]
```

Zde vidíme, že první tři posloupnosti po zpracování tvoří souvislou komponentu, protože sdílejí proměnnou `x`, ale původně nebyli v poli za sebou. Nulté a druhé pole sdílejí proměnnou `x`, podle které jsme druhou posloupnost dělili. V průběhu prohledávání grafu budeme vždy iterovat po daných posloupnostech. V moment nalezení vhodného elementu se posuneme na další prvek posloupnosti (doprava) nebo na novou posloupnost. To symbolizuje DFS krok zanoření. Opačně se posouváme doleva a zkoušíme ještě neprohledané elementy grafu. Díky rozdělení můžeme při přesouvání na začátek další posloupnosti vždy navázat již nalezenou proměnnou, pokud existuje a jedná o součást aktuální komponenty. Z pole posloupnosti nyní vytvoříme pole `DFSBaseMatch[][] patterns`. `patterns[i]` znamená přístup k i-té posloupnosti a `patterns[i][j]` přístup k j-té položce i-té posloupnosti.

Vzor kromě `patterns` obsahuje pole `Element[] scope`. Pole obsahuje každou proměnnou prohledávání právě jednou. Pokud ve vzoru není žádná proměnná, tak je pole vždy prázdné. Každá proměnná má svou pozici. Tyto pozice budeme chápat jako `ID` proměnných v celém dotazu. Toto pořadí je uchováváno v třídě `QueryVariableMap`. V moment nalezení vhodného elementu proměnné se element uloží do daného pole na pozici proměnné. V moment vynořování z DFS se element z pozice smaže. Položka se používá ke kopírování do tabulky nebo k dalšímu zpracování.

Vzor dále implementuje rozhraní `IDFSPattern` spolu s `IPattern`, které slouží k posouvání po posloupnosti. Mají množství metod, ale vypíšeme pouze hlavní:

- Položka `int CurrentPatternIndex` je index aktuální posloupnosti procházení. `patterns[CurrentPatternIndex]` vrátí aktuální posloupnost.
- Položka `int CurrentMatchNode` je aktuální objekt `DFSBaseMatch` v posloupnosti. `patterns[CurrentPatternIndex][CurrentMatchNode]` je aktuální objekt `DFSBaseMatch`.
- Metoda `void PrepareNextSubPattern()` připraví k procházení následující posloupnost.
- Metoda `void PreparePreviousSubPattern()` připraví k procházení předchozí posloupnost.
- Metoda `void PrepareNextNode()` připraví k procházení následující objekt `DFSBaseMatch`.
- Metoda `void PreparePreviousNode()` připraví k procházení předchozí objekt `DFSBaseMatch`.

- Metoda `Element[] GetMatchedVariables()` vrátí `scope` vzoru.
- Metoda `bool Apply(Element[] e)` slouží ke zjištění, zda držený element je použitelný pro aktuální pozici ve vzoru. Vyvolá stejnojmennou funkci na třídě `DFSBaseMatch`.

Nyní popíšeme třídu vytvářející pole posloupností.

Třída DFSBaseMatch

`DFSBaseMatch` je abstraktní třída, reprezentující jeden element procházení grafu. Třída obsahuje:

- Položku `bool isAnonnymous` která říká, jestli se jedná o proměnnou.
- Položku `bool isFirstAppearance` která říká, pokud se jedná o proměnnou, jestli už je to její první nález.
- Položku `int positionOfRepeatedField` která říká, pokud to není první nález proměnné, tak kde v `scope` se nachází.
- Metodu `bool Apply(Element element, Element[] map)`, která ověřuje jestli element `element` se dá použit v aktuálním kroku prohledávání. `map` je pak `scope` vzoru, kde se případně ověří rovnost elementů opakující se proměnné. Při úspěchu vrací `true`.

Potomci pak specifikují, jestli se jedná o `Vertex` (vrchol), `InEdge` (hrana vedoucí do vrcholu), `OutEdge` (hrana vedoucí z vrcholu) nebo `AnyEdge` (jakýkoliv druh hrany). Ještě než popíšeme struktury algoritmu prohledávání, tak popíšeme struktury pro ukládání výsledků vláken v průběhu prohledávání grafu.

Třída MatchFixedResultsInternal

Máme implementován vzor a metody k procházení vzoru. Nyní popíšeme implementaci ukládání výsledků prohledávání. Třída obsahuje lokální tabulku výsledků vlákna při prohledávání grafu (obrázek 2.4 blok `Výsledky`). Každé vlákno má odkaz na vlastní třídu v průběhu prohledávání grafu. Za finální výsledek prohledávání se považují hodnoty v poli `scope`. Sloupečky zde tedy odpovídají unikáním proměnným v grafu. Řádek je pak kopie daného pole. Ukládáme pouze elementy grafu. Uvnitř je:

- Položka `int ColumnCount`, která udává počet sloupečků tabulky.
- Položka `int FixedArraySize`, která udává velikost bloků uvnitř tabulky.
- Položka `List<Element[][]> ResTable` je tabulka výsledků. Je složená z bloků `Element[][FixedArraySize] block` konstantní velikosti. `block[i]` přistupuje k i-tému sloupečku a `block[i][j]` přistupuje k j-té pozici i-tého sloupečku.
- Položka `Element[][FixedArraySize] LastBlock` je odkaz na poslední nezaplněný blok.

- Položka `int CurrentPosition` je odkaz na první volný index v posledním bloku.
- Metoda `void AddRow(Element[] row)` přidá nový výsledek do tabulky. Pokud je poslední blok zaplněn, vytvoří se nový. Při vytváření nového bloku nemusí docházet k překopírovávání výsledků prohledávání, protože pole `ResTable` drží odkazy na zmíněné bloky. Rozšířením pak překopíruje pouze odkazy na pole. Očekává se, že `row` je položka `scope` vzoru.

Třída MatchFixedResults

Třída obsahuje lokální tabulky všech vláken v položce `matcherResults`. Takto budeme moct přistoupit k lokálním výsledkům po dokončení prohledávání grafu. Třída zároveň poskytuje rozhraní pro slévání sloupečků tabulek vláken v metodě `void MergeColumn(int columnIndex)`, která slévá jeden sloupeček. Výsledky jsou slévány do položky `List<Element[]>[] FinalMerged`.

Třída DFSPatternMatcherBase

Implementovali jsme vzor a ukládání výsledků. Nyní popíšeme implementaci algoritmu procházení (obrázek 2.4 blok `Matcher`). Abstraktní třída reprezentuje základní jednovláknový algoritmus DFS prohledávání grafu. Prohledání jsme implementovali přesně podle analýzy. Samotný algoritmus nebudeme popisovat, ale popíšeme jen základní položky. Třída v konstruktoru očekává vzor `DFSPattern` a graf `Graph`. Obsahuje dva indexy `int startVerticesIndex` a `int startVerticesEndIndex`. Dva indexy určují rozsah vrcholů z pole vrcholů `List<Vertex>`, ze kterých se bude spouštět prohledávání. Dané položky jsou inicializovány na celý rozsah pole, což odpovídá jednovláknovému zpracování. Třída má rozhraní `void SetStartingVerticesIndeces( int start, int end)` nastavující dané indexy rozsahu. Daná metoda se používá v paralelním zpracování, kdy je přidělováno malé množství vrcholů z grafu k prohledání. Metodou `void Search()` se spustí prohledávání grafu. Prohledávání prochází všechny vrcholy v definovaném rozsahu a pak se ukončí. V tento moment lze nastavit zmíněnou metodou další rozsah a opět spustit prohledávání grafu. V moment nalezení finálního výsledku se zavolá abstraktní metoda `void ProcessResult()`, která zpracuje výsledek. Rozhraní prohledávání neposkytuje návrat nalezených výsledků. Pokud se výsledky mají ukládat, tak potomek dané třídy musí dostat v konstruktoru objekt úložiště a přepsat metodu `void ProcessResult()`. Způsob zpracování výsledku si definují potomci.

Třída DFSPatternMatcher

Třída je potomkem třídy výše. Reprezentuje algoritmus prohledávání grafu, který ukládá výsledky do tabulky v moment jejich nalezení. V konstruktoru dostane tabulku `MatcherFixedResultsInternal`, kam ukládá své výsledky prohledávání. Metoda `ProcessResult` tedy ukládá výsledky do dané tabulky. V průběhu volání této metody taky dochází k počítání nalezených výsledků v položce `NumberOfMatchedElements`. Pokud uživatel zadal pouze `count(*)` v části *Select*, tak nedochází k ukládání výsledků do tabulky.

Třída DFSParallelPatternMatcher

Třída představuje paralelní prohledávání grafu, které ukládá výsledky v moment nalezení do tabulky. Paralelizaci jsme popsali v sekci 2.5.4. V konstruktoru dostává počet vláken `ThreadCount`, vzor `DFSPattern`, graf `Graph` a instanci úložiště výsledků `MatchFixedResults`. K paralelizaci prohledávání grafu využívá instance třídy `DFSPatternMatcher` a počet instancí odpovídá hodnotě `ThreadCount`. Tedy v konstruktoru vytvoří tyto instance a každé přidělí kopii vzoru, graf a její lokální úložiště. Výsledně obsahuje položky:

- `MatchFixedResults results` obsahuje lokální tabulky výsledků vláken.
- `DFSPatternMatcher[] matchers` obsahuje instance tříd algoritmů prohledávání grafu. Každá instance má svou lokální kopii vzoru a odkaz na tabulku výsledků.

Paralelní prohledávání je spuštěno metodou `void Search()`. Zde využíváme nativní `ThreadPool`. Je vytvořeno `ThreadCount` instancí `Task` vykonávají práci `WorkMultiThreadSearch(object o)`, kde `o` je lokální `JobMultiThreadSearch`. Ten obsahuje instanci prohledávání a objekt `VertexDistributor`. Objekt jsme definovali v analýze (obrázek 2.4 blok `VertexDistributor`). Drží odkaz na pole vrcholů grafu a index posledního přiděleného vrcholu. Množství přidělených vrcholů je definováno v položce `int verticesPerRound`. Vlákna jej žádají o rozsah vrcholů metodou `void DistributeVertices(out int start, out int end)` a následně spustí prohledávání z daných vrcholů.

Po dokončení prohledávání dojde k paralelnímu slévání tabulek. Rozhodli jsme se použít metodu slévání po sloupečcích tabulek, protože slévání celých tabulek po dvojicích bylo pomalejší. Paralelizace probíhá podobným způsobem jako paralelizace prohledávání. Je vytvořeno n instancí `Task`, kde n je počet sloupečků v tabulce, které vykonávají práci `ParallelMergeColumnWork(object o)`, kde `o` je lokální `ParallelMergeColumnJob`. Ten obsahuje odkaz na `MatchInternalResults` a `ColumnDistributor`. Vlákna žádají objekt `ColumnDistributor` o sloupeček ke slévání. Interně objekt funguje totožně jako `VertexDistributor` s rozdílem, že si objekt pamatuje index prvního nepřiděleného sloupečku. Slévání probíhá voláním metody `MergeColumn` na instanci `results`. V moment dokončení slévání jsou finální výsledky v položce `results.FinalMerged`. Tato položka je předána konstruktoru `TableResults`, která implementuje rozhraní `ITableResults`. Třída představuje tabulku výsledků, kterou si objekty částí dotazu (`QueryObjekt`) předávají ke zpracování. Pokud dotaz obsahoval pouze `count(*)` v části *Select*, tak se výsledky neukládaly do tabulku a místo slévání výsledků dojde pouze ke slévání položek instancí DFS algoritmů `NumberOfMatchedElements`.

3.4.6 Tabulka výsledků

V této sekci popíšeme implementaci tabulky výsledků, kterou si předávají objekty dotazu. Tabulku jsme implementovali dle sekce analýzy 2.4.2.

Třída TableResults

Objekty exekučního plánu si předávají tabulku výsledků prohledávání rozhraním `ITableResults`. Ačkoliv jsme navrhli rozhraní, tak obecně používáme pouze

jednu třídu `TableResults`, která rozhraní implementuje. Zároveň jsme ji upravili rovnou k možnosti použití pro ukládání pouze reprezentantů skupin popsaném v sekci 2.10.1. Hlavní vlastnosti jsou:

- `int ColumnCount` je počet sloupečků v tabulce. Sloupeček zde odpovídá jedné proměnné dotazu, tj. na jeden řádek se díváme jako na hodnoty v poli `scope` z průběhu prohledávání. Tedy počet sloupečků je roven počtu unikátních proměnných v dotazu. Pokud chceme přistoupit k proměnné výsledku prohledávání, musíme znát její pozici v poli `scope`.
- `int RowCount` je počet řádků v tabulce.
- `int FixedArraySize` je velikost bloků v tabulce. Tabulka opět využívá princip ukládání výsledků do bloků fixní délky. Stejný princip jsme použili při ukládání výsledků v průběhu prohledávání grafu. To nám umožní v moment dokončení slévání pouze přesunou výsledky do konstruktoru této třídy.
- `List<Element[FixedArraySize]>[] resTable` je tabulka výsledků prohledávání. `resTable[i]` je i-tý sloupeček. `resTable[i][j]` je j-tý blok ve sloupečku. `resTable[i][j][k]` je k-tý výsledek v bloku.
- `Element[] temporaryRow` představuje odkaz na pole elementů, ke kterému lze přistoupit skrze rozhraní tabulky, aniž by bylo pole elementů v tabulce.
- Metoda `void StoreRow(Element[])` překopíruje hodnoty do tabulky. Funguje ekvivalentně jako `AddRow` ve třídě `MatcherFixedResultsInternal`.
- Metoda `void StoreTemporaryRow()` vyvolá metodu výše pro uložení pole `temporaryRow` do tabulky, tj. zavolá metodu `StoreRow(temporaryRow)`.
- Funkce hranatých závorek `RowProxy this[int i]`, která vrátí proxy třídu řádku.
- Položka `int[] order` umožní definovat vlastní pořadí řádků v tabulce bez nutnosti přesouvat řádky.
- Položka `int NumberOfMatchedElements`, která udává počet nalezených výsledků, pokud bylo v části dotazu *Select* zadáno pouze `count(*)`. V takovém případě nedocházelo k ukládání výsledků do tabulky, ale jen počítání jejich počtu.

V průběhu analýzy jsme navrhli způsob práce s řádky. Místo abychom pracovali konkrétně s řádky, tak jsme implementovali proxy třídu řádku pomocí struktury `struct RowProxy`. Struktura obsahuje pouze dvě položky. První je index řádku `int index`, který reprezentuje, a druhá je odkaz na tabulku `TableResults resTable`. Získá se voláním funkce hranatých závorek na třídě tabulky. Na získané struktuře se pak voláním metody `Element this[int c]` přistoupí k proměnné na řádku tabulky (`c` je zde sloupeček tabulky). Danou třídu pak budeme používat k vyhodnocení výrazů a výpočtům agregačních funkcí. Kdykoliv budeme předávat strukturu do funkce, tak ji budeme předávat pomocí parametru `in`, který vyvolá předání odkazem. Tím vyloučíme zbytečné kopírování položek struktury. Na struktuře existuje statická metoda `AreIdenticalVars`, která porovná dvě proxy třídy řádků, zda obsahují stejné elementy grafu na základě jejich `ID`.

3.4.7 Expressions

Než přistoupíme k implementaci *Order by* a *Group by* musíme implementovat způsob vyhodnocení výrazů. K výpočtu výrazů jsme implementovali vlastní jednoduchý systém. Postupovali jsme přesně podle sekce analýzy 2.3.3, proto popíšeme jen tvorbu a základní objekty.

Tvorba výrazů

V části načítání dotazu 3.4.3 jsme uvedli, že výstupem načítání jsou syntaktické stromy každé části dotazu. Tyto stromy obsahují i podstromy výrazů. Každá část dotazu má svůj objekt implementující rozhraní `IVisitor<T>`, kterým se sbírají důležitá data. Pokud v průběhu procházení dojde k nalezení podstromu výrazu, tak dojde k vytvoření speciálního objektu `ExpressionVisitor`. Tento objekt procházením podstromu vytvoří stromovou strukturu výrazu, pomocí které se bude vyhodnocovat daný výraz v průběhu vykonávání dotazu. Objekt implementuje `IVisitor<ExpressionBase>`, kde `ExpressionBase` reprezentuje výsledný výraz. Všechny výrazy se globálně udržují v třídě `QueryExpressionInfo`, která jim přiděluje `ID` na základě jejich pořadí vytvoření. Třída si také pamatuje výrazy agregačních funkcí a přiděluje jim rovněž `ID` na základě pořadí vzniku.

Třídy výrazů

Implementace tříd výrazů je totožná jako v analýze. `ExpressionBase` je abstraktní třída, která definuje základní rozhraní struktur výrazů. Definuje:

- Položku `int ExprPosition` je `ID` výrazu dle pořadí vytvoření.
- Metodu `Type GetExpressionType()`, která vrací návratový typ výrazu.
- Metodu `void CollectUsedVars(ref List<int> v)`, která vrátí `ID` proměnných potřebných k vyhodnocení stromové struktury výrazu.

Z třídy vzniká abstraktní potomek `ExpressionReturnValue<T>`, který definuje návratovou hodnotu funkce v parametru `T`. Třída definuje rozhraní výpočtu výrazu metodami `bool TryEvaluate( ... x, out T returnValue)`. Kde `x` jsou položky nutné k vyhodnocení výrazu, což jsou zde řádky z tabulky výsledků. Mezi hlavní patří `Element[]` a `RowProxy`. Z třídy následně dědí nová abstraktní třída `VariableReference<T>`, která představuje přístup k proměnné. `ID` přistoupené proměnné je uloženo v položce `int VariableIndex`. Z třídy finálně dědí dvě třídy. Konkrétní třída `VariableIDReference`, která vrací `ID` elementu grafu představující proměnnou. Druhá třída je `VariablePropertyReference<T>`, která představuje přístup k vlastnosti elementu grafu představující proměnnou a `T` je zde typ vlastnosti. `ID` vlastnosti je uloženo v položce `int PropertyID`.

Výraz agregační funkce

Při implementaci výrazu agregační funkce jsme postupovali jako v analýze. Vytvořili jsme dva koncepty. První koncept představuje logiku agregační funkcí. Tu představuje abstraktní třída `Aggregate`, jejíž potomci implementují specifickou logiku výpočtu funkcí. Druhý koncept představuje odkaz na vypočtenou

hodnotu funkce, ke které se může přistupovat v jiných částech dotazu. Tento koncept představuje třída `AggregateReference<T>: ExpressionReturnValue<T>`. Třída reprezentuje hodnotu již vypočtené agregační funkce a typ hodnoty je parametr `T`. `ID` funkce, kterou reprezentuje, je uloženo v položce `int AggrPosition`.

Obecně jsme vytvořili třídu `ExpressionHolder`, která udržuje odkaz na třídu `ExpressionBase`. Tato třída je používaná k předávání výrazů do funkcí a konstruktorů.

3.4.8 Order by

Implementovali jsme části nutné ke zpracování *Order by* a *Group by*. Nyní popíšeme implementaci jejich řešení. Začneme popisem části *Order by*. U implementace jsme vycházeli z sekce analýzy 2.6.

Třída OrderByObject

Část *Order by* je reprezentována objektem `OrderByObject: QueryObject`. *Order by* musí setřídit tabulku `TableResults` pomocí výrazů zadaných uživatelem (klíčů třídění). Klíče jsou vytvořeny v konstruktoru objektu při procházení syntaktického stromu objektem `OrderByVisitor<List<ExpressionComparer> >`. Výstup procházení obsahuje pole porovnávačů, které se musí použít k porovnaní řádků tabulky.

Třída ExpressionComparer

`ExpressionComparer` je abstraktní třída definující rozhraní porovnávače jednoho klíče třídění. Jejím úkolem je vypočíst a porovnat hodnoty jednoho klíče třídění dvou řádků. Definuje:

- Abstraktní metodu `int Compare(in RowProxy x, in RowProxy y)` porovnávající dva řádky tabulky. `in` parametr zde představuje předání argumentu odkazem, aby nedocházelo ke zbytečnému kopírování struktur.
- Položku `int[] usedVars`, která obsahuje `ID` proměnných nutných k výpočtu hodnot klíče.
- Položku `bool isAscending`, která určuje, jestli se třídí sestupně nebo vzestupně.
- Položku `ExpressionHolder expr`, která obsahuje výraz porovnávání.
- Položku `bool CacheResults`, která určuje, zda se má se má použít optimalizace. `true` označuje využití optimalizace.

Potomky třídy jsou třídy `ExpressionComparer<T>`, které představují porovnání konkrétních typů návratových hodnot výrazů. Potomci budou také implementovat zmíněné optimalizace z sekce analýzy 2.6.

Třída ExpressionComparer<T>

Třída konkretizuje návratovou hodnotu výrazu porovnání v parametru `T`. Obsahuje výraz `ExpressionReturnValue<T>`, který se vyhodnotí porovnávanými řádky. Implementuje metodu `Compare( ... x, ... y)` rodiče. Nyní popíšeme způsob porovnání společně s optimalizacemi:

1. Dojde k porovnání proxy tříd pomocí statické metody `RowProxy.AreIdenticalVars(in x, in y, this.usedVars)`, která porovnává elementy na řádcích tabulky dle jejich `ID`. Avšak, porovnáváme zde jen elementy proměnných, které se používají k výpočtu výrazu (položka `usedVars`). Pokud jsou totožné, nemusíme vypočítávat hodnotu výrazu. Tím vyřešíme optimalizaci porovnání stejných elementů z sekce 2.6.5.

2. Následuje vypočtení hodnot výrazů řádků a jejich porovnání. Pokud byla `CacheResults == false`, tak nic dalšího se neprovádí. Opačně, při výpočtu hodnot si uložíme pozici řádku `x.index` do `int lastXRow`, zda se výraz vyhodnotil správně do `bool lastXSuccess` a hodnotu výrazu do `T lastXValue`. To samé pro řádek `y`: `int lastYRow`, `bool lastYSuccess` a `T lastYValue`. Pokud při příštím porovnání budou `x` nebo `y` totožné řádky, tak již máme vypočtené hodnoty výrazů. Tím vyřešíme optimalizaci porovnání stejných vlastností z sekce 2.6.4.

Optimalizace v druhém kroku je nefunkční v paralelním prostředí. Protože by se vlákna snažila ukládat výsledky na sdílené pozice a došlo by k souběhu. Snažili jsme se problém vyřešit třídou `ThreadLocal` (lokální úložiště vlákna). Každá ukládaná položka by byla obsažena v dané třídě a vlákna by tak měla svá úložiště. Dalším zkoušeným řešením bylo uzavřít celou třídu porovnávače do `ThreadLocal`. Avšak, tyto způsoby způsobili zpomalení vykonávání a proto jsme se rozhodli optimalizaci využívat pouze v jednovláknovém prostředí.

Třída RowComparer a IndexToRowProxyComparer

Všechny třídy porovnávače využívané k porovnání dvou proxy tříd jsme uzavřeli do třídy `RowComparer`, která postupně zkouší vykonat porovnání porovnávači, než dojde ke stanovení rovnosti řádků. Protože jsme v analýze stanovili, že se budou porovnávat pouze indexy řádků, tak jsme vytvořili třídu, která obalí `RowComparer` a umožní porovnávat řádky pomocí indexů. Třída se jmenuje `IndexToRowProxyComparer`. Drží odkaz na tabulku `ResultTable` a zmíněnou `RowComparer`. Volání metody `int Compare(int x, int y)` na třídě dojde k získání proxy tříd řádků `x` a `y`. Následně dojde k jejich porovnání pomocí `RowComparer`. Implementaci třídy jsme dále rozšířili dle návrhu porovnávání řádků tabulky při vkládání řádků do vyhledávacího stromu z sekce analýzy 2.9.3. Existuje zde nově položka `bool allowDuplicities`. Pokud položka je nastavená na `false`, tak při určení shodnosti hodnot klíčů dvou řádků se výsledně použije k porovnání jejich index v tabulce (což jsou zde hodnoty `x` a `y`). Čili, položka nastavená na `true` porovnává pouze podle klíčů třídění. `true` je nastaveno tedy vždy, pokud je třída používá s vyhledávacími stromy.

Řešení Normal: Merge sort

Připravili jsme všechny nutné podklady pro vykonání třídění. Samotné vykonání třídění je implementováno v třídě `MultiColumnTableSorter` v metodě `Sort`. Třída drží odkaz na `IndexToRowProxyComparer`, který bude sloužit jako porovnávač při třídění. Metoda vytvoří pole indexů `int[]` velikosti odpovídající počtu řádků v tabulce. K třídění indexů v poli používáme knihovnu HPCsharp [21]. Konkrétně využíváme algoritmus Merge sort pro jednovláknové zpracování a pro paralelní používáme paralelní verzi Merge sort. Výsledně se pole indexu předá tabulce na položku `ResultTable.order`. Pole nyní slouží jako indexační struktura tabulky. Toto řešení budeme označovat v průběhu testování paralelního i jednovláknového zpracování jako **Normal: Merge sort**. První slovo určuje mód, kterému řešení přináleží.

3.4.9 Group by

V této sekci popíšeme implementaci *Group by*. U implementace *Group by* jsme vycházeli z sekce analýzy 2.7.

Třída GroupByObject

Část *Group by* je reprezentována objektem `GroupByObject: QueryObject`. *Group by* musí seskupit výsledky v tabulce prohledávání `TableResults` pomocí výrazů zadaných uživatelem (klíčů seskupení). Zároveň musí provést výpočet agregačních funkcí. Klíče jsou vytvořeny v konstruktoru objektu při procházení syntaktického stromu objektem `GroupByVisitor<List<ExpressionHolder> >`. Výsledné pole obsahuje výrazy, které se musí použít k seskupování. V průběhu sestavování dalších částí dotazu dochází k vytváření objektů logiky výpočtu agregačních funkcí. Logika je obsažena v potomcích objektu `Aggregate`. Výsledný objekt části *Match* tedy obsahuje pole výrazů a pole objektů logiky agregačních funkcí. Nejdříve popíšeme implementaci seskupování a následně implementaci agregačních funkcí.

Dictionary a ConcurrentDictionary

Abychom pochopili implementaci seskupování musíme se podívat na způsob práce s hašováním výrazů. V analýze jsme určili, že výsledky prohledávání budeme seskupovat pomocí hašovací tabulky, ať už v jednovláknovém nebo paralelním zpracování. Jazyk C# obsahuje hašovací tabulku `Dictionary<Key, Value>` pro jednovláknové zpracování a `ConcurrentDictionary<Key, Value>` pro paralelní zpracování. `ConcurrentDictionary` je thread-safe verze `Dictionary`. `Key` zde chápeme jako řádek tabulky a `Value` úložiště hodnot agregačních funkcí. `Key` zde nebude proxy třída, ale pouze index řádku. Porovnání vyvolá získání proxy třídy jako v předchozí sekci u třídění. Obvyklé vkládání prvků do `Dictionary` vypadá následovně:

```
Dictionary<int, int> dict = new Dictionary<int, int>();
int x = 5;
if (!dict.TryGetValue(x, out int y)) dict.Add(x, 42);
```

Nejdříve se v podmínce otestuje, jestli vkládaný prvek ve struktuře existuje. To znamená, že se vypočte haš prvku a nalezne se místo vložení. Jestli na místě nějaký prvek již leží, tak dojde k porovnání hodnot a jinak funkce vrací `false`. Funkce vrátí `true` při úspěchu porovnání prvků a při neúspěchu vrátí `false`. Při vrácení `false` dojde k vložení funkcí `Add`, která opět vypočte haš a případně porovná prvky. Obvyklé vkládání prvku pro `ConcurrentDictionary` vypadá následovně:

```
ConcurrentDictionary<int, int> dict =
    new ConcurrentDictionary<int, int>();
int x = 5;
var retVal = dict.GetOrAdd(x, 42);
```

Nyní neuvažujme žádné synchronizační koncepty. Funkce `GetOrAdd` vypočte haš, získá místo vložení a pokud na místě jiný prvek není rovnou ho vloží. Zde tedy dochází k hašování a porovnání pouze jednou. V prvním případě `Dictionary` bychom byli nuceni vyhodnotit ten samý výraz 4-krát. Obecně obě struktury pro dva rozdílné prvky se stejnou haš hodnotou mohou vytvářet spojový seznam daných prvků. Výsledně bychom museli vypočítávat výrazy při každém porovnání v seznamu. Navíc, pokud by docházelo ke slévání výsledků dvou hašovacích tabulek do jedné, tak bychom opět museli počítat haš a porovnávat. Interně obě struktury používají k výpočtu haš hodnoty a porovnání objekt s rozhraním `IEqualityComparer<T>`. Kde `T` je `Key`. Objekt má metodu `int GetHashCode(T o)` a `bool Equals(T x, T y)`. První vypočte haš vkládaného objektu a při porovnání se vyvolá metoda `Equals`. Strukturám se dá v konstruktoru poskytnou vlastní implementaci rozhraní. Díky této implementaci jsme vymysleli dvě optimalizace:

1. V moment kdy dochází k použití `Dictionary` nebo aktu slévání hašovacích tabulek budeme jako `Key` používat strukturu `GroupDictKey`. Struktura obsahuje dvě položky `int hash` a `int position`. `hash` je zde haš řádku a `position` je index řádku v tabulce výsledků. Tedy jsme rozšířili `Key` o haš hodnotu řádku, kterou můžeme znovu použít, pokud to bude nutné. Budeme implementovat vlastní `IEqualityComparer<T>`, který jen použije haš hodnotu ze struktury. Tímto vypočteme haš pouze jednou za celý cyklus vkládání nebo slévání. Pokud používáme `ConcurrentDictionary` a nedochází ke slévání, tak stačí jako `Key` volit index řádku.

2. Při výpočtu haš hodnoty se vypočítávají hodnoty výrazů. Abychom nepočítali stejné výrazy opětovně při porovnání, tak budeme používat dvě třídy. V prvním případě vytvoříme `ExpressionHasher`, který počítá haš hodnotu jednoho výrazu. Třídu propojíme s již existující `ExpressionComparer<T>`. Třída `ExpressionHasher` při výpočtu výrazu pak jen aktualizuje vnitřní hodnoty položek (`lastYRow`, `lastYValue` a `lastYSuccess`) uvnitř porovnávače a ten je následně využije. Nastavujeme položky argumentu `y` (dle metody `Compare(x, y)`), protože vkládaný prvek při porovnání je vždy uchováván v `y`. Toto odpovídá navržené optimalizaci z sekce analýzy 2.7.5.

Problémem u druhé optimalizace je opět sdílení položek v paralelním prostředí. Jelikož jsme se v předchozí sekci *Order by* rozhodli nevyužívat při paralelním

zpracování optimalizaci ukládání výsledků výrazů. Tak ani zde ji nebudeme implementovat pro paralelní zpracování.

Třída ExpressionHasher a třídy ExpressionHasher<T>

`ExpressionHasher` je abstraktní třída reprezentující jeden klíč seskupení. Definuje:

- Položku `ExpressionHolder expr`, která obsahuje výraz seskupení.
- Abstraktní metodu `int Hash(in RowProxy row)`, která vypočte haš výrazu pro řádek tabulky. V průběhu výpočtu dojde k nastavení položek porovnávače.
- Abstraktní metodu `SetCache(ExpressionComparer cache)`, ve které si potomci nastaví odkaz na `ExpressionComparer<T>`. To umožní následně při výpočtu hodnoty výrazu přiřadit výsledky dané třídě.

Potomky třídy jsou třídy `ExpressionHasher<T>`. Implementují výpočet výrazu pro konkrétní návratový typ `T` a výpočet haše výrazu. Třída obsahuje položku `ExpressionComparer<T> exprComp`, která je odkaz na porovnávač. Skrze odkaz se třídě přepíšou položky při výpočtu výrazu. Dále obsahuje výraz seskupení `ExpressionReturnValue<T> exprR`. Všechny klíče seskupení jsou obsaženy v třídě `RowHasher`, která vypočte finální haš kombinací všech haš hodnot výrazů.

Rozhraní IEqualityComparer<T>

V aplikaci jsme vytvořili dva objekty:

- `RowEqualityComparerInt: IEqualityComparer<int>` definuje porovnání a výpočet haš hodnot pomocí indexu řádku tabulky. Třída drží odkaz na tabulku `TableResults` pro získání proxy třídy řádku indexem. Dále má porovnávače `ExpressionComparer[]` pro zjištění rovnosti dvou prvků v hašovací tabulce a finálně `RowHasher` pro výpočet haše. Položka `bool cacheResults` určuje zda se má použít druhá optimalizace.
- `RowEqualityComparerGroupDictKey: IEqualityComparer<GroupDictKey>`, který je totožný s předchozím, ale neobsahuje `RowHasher`, protože haš je uložen v objektu porovnání.

Logika agregačních funkcí

Implementovali jsme způsob seskupování. Nyní budeme implementovat objekty logiky agregačních funkcí. V sekci implementace výrazů jsme řekli, že logika je reprezentována abstraktní třídou `Aggregate`. Tyto třídy jsou tvořeny v průběhu procházení syntaktických podstromů výrazů a jsou uloženy v poli `List<Aggregate>` uvnitř třídy `QueryExpressionInfo`. Indexy funkcí v daném poli slouží jako `ID` daných funkcí. Každá funkce potřebuje úložiště hodnot. Úložiště pak musí být používány pro správnou funkci na základě jejich `ID`.

Každá uživatelem zadaná funkce je tedy reprezentována třídou `Aggregate`. Třída obsahuje položku `ExpressionHolder expr`, která představuje výraz, jehož

hodnota se použije k výpočtu funkce. Dále třída definuje metody, které slouží k aplikování logiky pro specifické úložiště. Popíšeme je obecně:

- Metoda `void Apply(x, druh_úložiště y)` vypočte hodnotu výrazu `expr` pomocí `x` (`RowProxy` nebo `Element[]`), hodnotu zpracuje definovanou logikou a výsledek uloží do úložiště `y`. Pro každé úložiště pak existuje daná metoda. Zároveň pro úložiště existují thread-safe verze metod opatřených příponou `ThreadSafe`.

- Metoda `void Merge(druh_úložiště_X x, druh_úložiště_Y y)` sloučí výsledky dvou úložišť `x` a `y`. Výsledek sloučení je uložen v úložišti `x`. Pro každé úložiště pak existuje daná metoda a opět i thread-safe verze opatřených příponou `ThreadSafe`.

Konkrétní implementace logiky je přesunuta na potomky. Z dané třídy dědí abstraktní třída `Aggregate<T>`, která konkretizuje návratovou hodnotu výrazu parametrem `T`. Obsahuje položku `ExpressionReturnValue<T>` pro výpočet hodnoty výrazu. Z třídy následně vznikají už konkrétní implementace logiky. Funkce `sum` a `avg` mohou na vstupu obsahovat pouze číselnou hodnotu. To v našem případě je pouze `int`, protože jsme omezili typy vlastností ve vstupních souborech. Funkce `count`, `min` a `max` mohou navíc mít i řetězec:

- Třída `IntSum: Aggregate<int>` reprezentuje funkci `sum`. Obecně metoda `Apply` vypočte hodnotu výrazu a přičte ji k hodnotě v úložišti. Úložiště musí obsahovat sumu již vypočtených výrazů. Thread-safe verze používá k přičtení atomickou operaci přičtení `Interlocked.Add(...)`. `Merge` pak pouze přičte hodnotu v `y` do `x` stejným způsobem.

- Třída `IntAvg: Aggregate<int>` reprezentuje funkci `avg`. Úložiště musí obsahovat sumu již vypočtených výrazů společně s jejich počtem. Zpracování pak pouze přičte hodnotu výrazu k sumě a navýší jejich počet. Přičítání je implementováno shodně jako v `IntSum`.

- Třída `Count<T>: Aggregate<T>` představuje funkci `count`. Přebírá v definici parametr `T`, protože v případě nepoužití `count(*)` musí být schopna pracovat se všemi druhy návratových hodnot výrazů. Úložiště musí obsahovat počet správně vyhodnocených výrazů a v případě `count(*)` to je pouze počet prvků skupiny. V případě `count(*)` se nevyhodnocují žádné výrazu uvnitř metody zpracování, ale pouze navyšuje interní hodnota úložiště. Implementace metod je stejná jako u `IntSum`.

- Třída `MinMaxBase<T>: Aggregate<T>` představuje logiku funkcí `min` a `max`. Očekává se, že úložiště bude mít `bool`, který říká zda již byla hodnota inicializována. Dále má aktuální minimum nebo maximum hodnot výrazu. V threa-safe verzích metod je využito `Interlocked.CompareExchange(..)` jako princip *Compare and Exchange*.

Úložiště hodnot agregačních funkcí

Objekty `Aggregate` pracují s úložišti. V analýze jsme definovali dva druhy úložišť. První bylo úložiště Bucket a druhé List. V aplikaci jsme implementovali

řešení reprezentující úložiště Bucket a řešení úložiště List přesně podle analýzy. Samotná logika tříd je společná všem řešením, proto popíšeme pouze podrobněji řešení Bucket. Zbylá řešení implementují stejnou logiku s tím, že místo jedné hodnoty obsahují pole hodnot. Samotná úložiště jsou použita jako `value`, při vkládání do `Dictionary<key, value>` nebo `ConcurrentDictionary<key, value>`. To znamená, že každá skupina má své úložiště.

Třída AggregateBucketResult

Představuje abstraktní třídu používající způsob ukládání Bucket. Potomci definují typ ukládané hodnoty. Každý potomek je využívaný určitou agregační funkcí. Třída definuje:

- Metodu
  `AggregateBucketResult Factory(Type type, string funcName)`,
  která implementuje návrhový vzor Factory metoda [17, str. 107]. `type` zde určuje typ ukládané hodnoty v úložišti a `funcName` představuje druh agregační funkce, která s úložištěm bude pracovat.

- Metodu
  `AggregateBucketResult[] CreateBucketResults(Aggregate[] ags)`,
  která vytváří výsledné úložiště hodnot na základě počítaných agregačních funkcí. Pro Bucket jsme úložiště definovali jako pole tříd, ve kterém každá třída obsahuje hodnotu počítané funkce. Výsledek funkce označme `x`. Přístup `x[i]` odpovídá úložišti funkce `ags[i]`.

Z třídy dále dědí `AggregateBucketResult<T>`, která definuje ukládanou hodnotu `T aggResult`. Třída se používá při výpočtech funkcí `count` a `sum`. Z této třídy dědí dvě třídy:

- První je `AggregateBucketResultWithSetFlag<T>`, která přidává položku `bool isSet`. Položka určuje zda byla hodnota již inicializována. Třída se použije při výpočtu funkcí `min` a `max`.

- Druhá je `AggregateBucketAvgResult<T>`, která se použije při výpočtu `avg` a definuje počet vypočtených výrazů `int eltsUsed`.

Pro řešení List je návrh totožný. Názvy místo `Bucket` obsahují `List`. A definované položky jsou pole místo jedné hodnoty (např. `List<T> aggResults` místo `T aggResult`). Výsledně při používání úložiště Bucket vkládáme jako `value` do hašovací tabulky odkaz na pole vytvořené metodou `CreateBucketResult(..)`. Při použití List nemůžeme použít odkazy na pole, protože hlavní myšlenka List je mít úložiště mimo hašovací tabulku, abychom nemuseli vytvářet spoustu tříd jako u Bucket. V tomto případě budeme vkládat jako `value` pozici hodnot v úložišti pro danou skupinu. Každá skupina tedy obdrží unikátní index (typ `int`). Výsledky agregačních funkcí pro danou skupinu jsou tedy uloženy na daném indexu v úložišti.

Group by zpracování

Implementovali jsme všechny potřebné objekty k vykonání seskupení. Nyní krátce popíšeme řešení zpracování, protože jsme zde postupovali dle analýzy. Každému řešení rovněž přiřadíme název, kterým řešení označíme v průběhu testování. Název bude opět obsahovat název módu, za kterým následuje název řešení. Zde navíc uvedeme za název řešení do závorky druh úložiště, tj. Bucket nebo List. Začneme jednovláknovým zpracováním (sekce analýzy 2.7.4):

- Řešení **Normal: SingleThreadSolution (Bucket)** odpovídá jednovláknovému zpracování *Group by* pomocí úložiště Bucket. Řešení je v třídě `GroupByWithBuckets`. K seskupení je použit `Dictionary<key, value>`, proto je zde využit `GroupDictKey` jako `key`. `value` je zde odkaz na pole úložiště `AggregateBucketResult[]`. Pole úložiště hodnot agregačních funkcí se vytváří pouze v momentě, kdy je vytvářen nový záznam do hašovací tabulky. `RowEqualityComparerGroupDictKey` je zde rozhraní pro určení rovnosti skupin. Využívají se obě optimalizace.

- Řešení **Normal: SingleThreadSolution (List)** odpovídá jednovláknovému zpracování *Group by* pomocí úložiště List. Řešení je obsaženo v třídě `GroupByWithList`. Řešení využívá `Dictionary<key, value>`, tudíž je zde opět `GroupDictKey` jako `key`. Úložiště výsledků agregačních funkcí je vytvořeno na začátku v položce `aggResults`. `value` je `int`, protože úložiště hodnot agregačních funkcí leží mimo hašovací tabulku. Při přidávání nové skupiny do hašovací tabulky je určena nová pozice pro skupinu v úložišti `aggResults` pomocí `Dictionary.Count`. Tato hodnota se vkládá s klíčem do hašovací tabulky a skrze ni se přistoupí k výsledkům funkcí pro skupinu.

Všechna paralelní řešení využívají nativní třídu `ThreadPool`. `Tasks` vykonávají statickou metodu `SingleThreadGroupByWork(object o)`, kde `o` je objekt obsahující lokální položky vláken. Obecně pro všechna řešení daný objekt obsahuje odkaz na pole `Aggregate`, odkaz na tabulku výsledků prohledávání `TableResults` a dva indexy určující rozsah výsledků z tabulky ke zpracování. Každé řešení je reprezentováno třídou, která implementuje vlastní logiku zpracování.

- Řešení **Normal: Global (Bucket)** odpovídá paralelnímu zpracování z sekce 2.7.8. Řešení je obsaženo v třídě `GlobalGroupByBucket`. K seskupení vlákna využívají sdílenou hašovací tabulka `ConcurrentDictionary<key, value>`. Není zde druhá optimalizace a `key` je zde pouze index řádku (`int`). `value` je zde odkaz na pole úložiště `AggregateBucketResult[]`. Metoda `GetOrAdd(key, value)` je použita k vkládání. Při úspěšném vložení nového záznamu vrátí odkaz na námi vkládané pole úložišť (`value`). Při neúspěchu vrací odkaz na již dříve vložené pole. Zda k tomu došlo ověříme pomocí rovnosti referencí na vkládané a vrácené pole. Samotná logika funkcí vykonává jejich thread-safe verze. Rozhraní pro určení rovnosti řádků je zde `RowEqualityComparerInt`. Úložiště List jsme se snažili implementovat, ale nepodařilo se nám vytvořit efektivní řešení. Hlavní problémem byla synchronizace při přístupu k úložišti hodnot agregačních funkcí. Zde jsme zkoušeli využít nativní třídu `Semaphor` k omezení vstupu do kritické sekce. Ta však při přístupu pracuje se zámky a ve výsledku řešení bylo značně pomalé. Řešení proto nebudeme dále uvádět.

- Řešení **Normal: Two-tep (Bucket)** odpovídá paralelnímu zpracování z sekce 2.7.9 s úložištěm Bucket. Řešení je v třídě `TwoStepGroupByBucket`. První část je implementována stejně jako jednovláknové řešení s Bucket. Každé vlákno drží odkaz na svou lokální tabulku `Dictionary`. Druhá část funguje jako Global řešení výše s rozdílem, že klíčem je zde `GroupDictKey`. Zde vlákna sdílejí `ConcurrentDictionary`.

- Řešení **Normal: Two-step (List)** odpovídá paralelnímu zpracování z sekce 2.7.9 s úložištěm List. Řešení je v třídě `TwoStepGroupByList`. První část je implementována stejně jako jednovláknové řešení s List. Druhá část byla problematická, jelikož se jedná o stejný problém s úložištěm List jako u Global řešení. Proto jsme se rozhodli v druhé části výsledky z List přeložit do Bucket. To znamená, že pro záznamy v List vytvoříme nová pole `AggregateBucketResult[]`, které vložíme do `ConcurrentDictionary`. Výsledně je druhá část implementována stejně jako Global řešení výše s rozdílem, že klíčem je zde `GroupDictKey`.

- **Normal: LocalGroupByLocalTwoWayMerge (Bucket)/(List)** odpovídají paralelním zpracováním z sekce 2.8.6 s úložištěm Bucket/List. Řešení jsou v třídách `LocalGroupByLocalTwoWayMerge` s příponou `Bucket` nebo `List`. V prvním kroku jsou implementovány stejně jako jednovláknová řešení. Vlákna zde drží odkaz na svou lokální tabulku `Dictionary`. Při slévání dvou tabulek dochází k překopírování záznamů z jedné hašovací tabulky do druhé. První je poté zahozena. Výsledně existuje pouze jedna tabulka. Zde má značnou výhodu Bucket řešení, protože při slévání dochází k přesunutí odkazu na pole. Zatímco u List dochází překopírovávání prvků mezi poli.

- **Normal: SingleGroup** odpovídájí módu *Single Group Group by* zpracování z sekce 2.7.11. Řešení je v třídě `SingleGroupGroupBy`. Třída je využívaná pro jednovláknové i paralelní zpracování. Každé vlákno drží odkaz na lokální pole `AggregateBucketResult[]`, do kterého ukládá hodnoty funkcí. Není zde úložiště List, protože pro každé vlákno existuje jen jedna skupina.

3.4.10 Úprava propojení

Implementovali jsme řešení, která zpracovávají výsledky po dokončení prohledávání grafu. V této sekci popíšeme implementaci úprav dotazovacího enginu dle zadání práce. K realizaci zpracování v průběhu prohledávání grafu jsme postupovali podle sekce analýzy 2.8.

Zahrnutí úprav do třídy Query

Celé upravené zpracování dotazu je reprezentováno již existující třídou `Query`. Samotný způsob vykonání se definuje v metodě `Create(...)`, kde jeden vstupní argument je mód. Při vybrání upraveného módu se spustí rozdílný privátní konstruktor. V konstruktoru dojde k vytvoření exekučního plánu. V analýze jsme určili, že objekt části *Select* s novým objektem *Match* části `MatchObjectStreamed` je propojen původním způsobem. Nyní pouze vytvoříme objekt `ResultProcessor` a nový objekt části *Match* `MatchObjectStreamed`.

Třída ResultProcessor

Části dotazu *Order/Group by* budou nyní reprezentovány potomky nové třídy `ResultProcessor`. Metody jsou použity ke zpracování výsledků v průběhu prohledávání grafu. Třída definuje metody dle obrázku z analýzy 2.8. Definuje:

- Abstraktní metodu `void Process(int matcherID, Element[] result)`, která implementuje logiku zpracování jednoho výsledku prohledávání. První parametr `matcherID` je zde `ID` instance algoritmu prohledávání grafu, které se použije při přístupu k lokálním výsledkům v paralelních řešeních. Očekává se, že instance algoritmu prohledávání grafu v moment dokončení prohledávání signalizují situaci pomocí této metody s `result == null`. Po této signalizaci může dojít k finálním úpravám výsledků.
- Abstraktní metodu `RetrieveResults(out ITableResults t, out GroupByResults g)`, která získá finální zpracované výsledky v podobě tabulek.

Z dané třídy dědí dvě abstraktní třídy. První je třída `OrderByResultProcessor` představující část dotazu *Order by*. Díváme se na ni jako na ekvivalent třídy `OrderByObject`. Druhá je třída `GroupByResultProcessor` představující část dotazu *Group by*. Díváme se na ni jako na ekvivalent třídy `GroupByObject`. Samotná konstrukce daných tříd je rovněž totožná s původními. To znamená, že při konstrukci opět dochází k procházení syntaktických stromů a tvorbě totožných výsledných struktur. Rozdíl je ten, že potomci tříd specifikují algoritmy zpracování v metodě `Process`.

Třída MatchObjectStreamed

Třída dědí z třídy `MatchBaseObject`. Třídě jsme přidali položku držící odkaz na třídu zpracovávající výsledky v `ResultsProcessor resultProcessor`. Dále jsme třídě přidali metodu `PassResultProcessor`, která uloží odkaz na objekt zpracování do dané položky. To odpovídá návrhu z obrázku 2.9. Abychom mohli vykonávat zpracování v průběhu, tak nyní musíme upravit i samotné objekty prohledávání grafu. Řekli jsme, že původní třída části *Match* obsahuje odkaz na objekt prohledávání grafu a tabulku výsledků. Nyní vlastní pouze odkaz na objekt zpracování `ResultsProcessor` a odkaz na nový objekt algoritmu prohledávání grafu `DFSParallelPatternMatcherStreamed`. Objekt algoritmu musíme také upravit a předat mu odkaz na objekt zpracování.

Třída DFSParallelPatternMatcherStreamed

Tato třída představuje upravené chování třídy `DFSParallelPatternMatcher`. V konstruktoru se nepředává tabulka výsledků `MatchFixedResults` a nedochází zde ke slévání výsledků. Při inicializaci objektů jednovláknového prohledávání grafu se vytvářejí nově instance třídy `DFSParallelPatternMatcherStreamed`. Tyto instance obdrží nově `ID` na základě jejich pořadí vzniku. Samotná paralelizace zpracování prohledávání funguje totožně jako v původní třídě. Třídu jsme dále rozšířili o metodu `PassResultProcessor`, která předá odkaz na objekt zpracování instancím jednovláknového prohledávání.

Třída DFSPatternMatcherStreamed

Třída reprezentuje jednovláknový DFS algoritmus prohledávání. Třída je potomkem třídy `DFSPatternMatcherBase`. V konstruktoru očekává nově `ID` objektu `int matcherID`. To znamená, že implementuje stejný algoritmus prohledávání, ale přepisuje metodu zpracování výsledků `ProcessResult()`. Nově třída drží odkaz na objekt zpracování *Order by* a *Group by* v položce `ResultProcessor resultProcessor`. Třídě jsme opět přidali metodu `PassResultProcessor`, která uloží odkaz objektu zpracování do zmíněné položky. Finálně upravená metoda `ProcessResult()` vyvolá předání nalezeného výsledku (pole `Element[] scope` vzoru) spolu se svým `ID` pomocí metody
`ResultProcessor.Process(Element[] result, int matcherID)`.

Toto propojení vyplývá z sekce 2.8.4. Tímto jsme dokončili úpravu objektu části *Match* pro pozměněné zpracování. Nyní přistoupíme ke konkrétním implementacím zpracování a potomkům třídy `resultProcessor`.

3.4.11 Úprava Order by

V této sekci popíšeme implementaci úprav části dotazu *Order by*. Potomci specifikují navržené přístupy zpracování dle sekce analýzy 2.9. Část dotazu *Order by* je reprezentována třídou `OrderByResultProcessor`. Nejdříve popíšeme implementaci použitých vyhledávacích stromů a následně konkrétní třídy zpracování.

Implementace (a, b)-stromu

Implementovali jsme vlastní (a, b)-strom, protože jsme nedokázali nalézt již existující knihovnu. Obecně při zpracovávání budeme využívat dva druhy stromů. První je obecný (a, b)-strom a druhý je optimalizovaný strom z sekce 2.9.3, který seskupuje totožné prvky do pole místo aby je vložil do vrcholu. Oba stromy implementují rozhraní `IABTree<T>`, který definuje metodu vložení prvku `void Insert(T key)` a položku počtu prvků ve stromě `itn Count`. Toto rozhraní použijeme pro vytvoření obecného algoritmu zpracování, kterému budeme jednoduše moct změnit druh stromu. Při zpracování očekáváme, že vkládané prvky jsou indexy řádků tabulky. Potřebujeme setřídit i prvky se shodnými klíči, proto u všech řešení je použit `IndexToRowProxyComparer` s `allowDuplicities == false`. To způsobí porovnání při shodnosti klíčů pomocí indexů řádků a navíc `cacheResults == true`, protože se stromem vždy bude pracovat jedno vlákno.

Třída ABTree<T>

Třída představuje (a, b)-strom [20, str. 190], u kterého jsem upravili definici na $b = 2a$. `T` je zde typ vkládaných prvků. V konstruktoru obdrží parametr b. Struktura definuje:

- Metodu `void Insert(T key)`, která vloží prvek `key` do stromu. Prvky, které již jsou ve stromě nejsou vloženy.
- Položku `int Count`, která určuje počet prvků ve stromě.

Strom implementuje pouze metodu vkládání `Insert`, protože pro naše zpracování funkce mazání `Delete` není potřebná. Při vkládání je hledání správného

podstromu realizováno binárním vyhledáváním. Strom je interně složen z tříd `ABTreeNode<T>`, reprezentující vrcholy stromu. Vrcholy obsahují odkaz na rodiče, abychom při iterování a procházení stromu nemuseli zaznamenávat cestu z kořene. Vrchol obsahuje:

- `List<ABTreeNode<T> > children` je pole potomků. Každý potomek je zde chápán jako podstrom.
- `List<T> keys` je pole vkládaných hodnot.
- `ABTreeNode<T> parent` je odkaz na rodiče vrcholu.
- `int index` je pozice vrcholu v jeho rodičovském vrcholu.

Třída implementuje rozhraní `IABTree<T>` a `IEnumerable<T>`. Rozhraní procházení `IEnumerable<T>` iteruje prvky stromu od nejmenšího po největší.

Třída ABTreeAccumulator<T>

Třída představuje optimalizovaný (a, b)-strom z sekce analýzy 2.9.3. Třída interně funguje totožně jako třída `ABTree<T>` a implementuje `IABTree<T>`. Vrchol stromu `ABTreeNode<T>` nově vlastní položku `List<List<T> > accumulations`. Každému prvku v poli `keys` na indexu `i` náleží právě jedno pole v `accumulations` na stejném indexu. Metoda `Insert` chybějící prvek vloží do pole `keys` na index `i` a vytvoří prázdné pole v `accumulations` na indexu `i`. Pokud se v budoucnu vkládaný prvek rovná již vloženému prvku v poli `keys` na indexu `i`, tak je vkládaný prvek vložen do pole na pozici `i` v poli `accumulations`. Třída implementuje rozhraní `IEnumerable<ValueAccumulation<T> >`. `ValueAccumulation<T>` je struktura, která obsahuje prvek z pole `keys` (`T value`) a jeho náležité pole v `accumulations` (`List<T> accumulation`).

Řešení Half-Streamed

Řešení je obsaženo v třídě `ABTreeSorterHalfStreamed` a odpovídá sekci analýzy 2.9.4. Třída je abstraktním potomkem `OrderByResultProcessor`. Potomci této třídy jsou řešení specializované na druh vyhledávacího stromu. Definuje:

- Položku `SortJob[] sortJobs`, která obsahuje lokální výsledky vláken. Každému vláknu v průběhu prohledávání grafu náleží objekt algoritmu DFS `DFSPatternMatcherStreamed` s identifikátorem v položce `int matcherID`. Tyto `ID` vznikly na základě pořadí vytvoření objektů. Tedy vlákno vždy při volání metody `Process` přistupuje k objektu `sortJobs[matcherID]`. Objekt interně obsahuje `ITableResults resTable` (lokální tabulka výsledků prohledávání) a `IABTree<int> tree` (lokální indexační struktura).
- Položku `int sortJobsFinished`, která určuje kolik vláken již dokončilo prohledávání grafu. Vlákna při dokončení prohledávání grafu vyvolají atomické navýšení `Interlocked.Increment` na této položce. Poslední vlákno zahájí paralelní slévání výsledků.
- Metodu `SortJob CreateJob(..)`, kterou implemetují potomci a vytváří v ní specializovaný druh vyhledávacího stromu.

- Implementaci metody `Process(Element[] result, int matcherID)`. V první části vlákno přistoupí k `sortJobs[matcherID]`. Následně překopíruje výsledek `result` do tabulky (`resTable.StoreRow(result)`) a zatřídí index nového řádku tabulky (`tree.Insert(resTable.RowCount-1)`). V této části fungují obě optimalizace porovnání. Po dokončení prohledávání všech vláken dochází k paralelnímu dvoucestnému slévání. Slévání je implementováno v objektu `MergeObject<T>` v metodě `T[] Merge()`. V jednovláknovém zpracování nedochází ke slévání.

- Objekt `MergeObject<T>`, který implementuje paralelní dvoucestné slévání. Interně používá metodu knihovny HPCsharp [21]. Rozhraní metody pracuje s dvěma pol. `T[] source` a `T[] destination`. V prvním kroku slévání dochází k překopírování výsledků uvnitř stromů do pole `source`. Na pole se tedy díváme jako na několik posloupností výsledků vláken. Posloupnosti jsou slévány knihovní metodou do pole `destination`. V této části nefunguje druhá optimalizace z důvodu paralelizace.

Toto je implementace obecné části. Z třídy `ABTreeSorterHalfStreamed` dědí dvě třídy, které představují dva řešení. Specifikují pouze implementaci `IABTree` a typ `T` třídy `MergeObject<T>`. Budeme se držet definovaného značení:

- Řešení **Half-Streamed: ABTree**: využívá `ABTree` jako `IABTree`. `T` třídy `MergeObject` je `RowProxy`, protože sléváme několik tabulek najednou a tedy použitím pouze `int` bychom nedokázali rozeznat, které tabulce index patří. Řešení je obsaženo v třídě `ABTreeGenSorterHalfStreamed`.

- Řešení **Half-Streamed: ABTreeAcccumulator**: využívá interně strom `ABTreeAccumulator` jako `IABTree`. `T` třídy `MergeObject` je `RowProxyAccum`. Jedná se o strukturu shodnou s `ValueAccumulation<int>`. Jediný rozdíl je ten, že místo indexu řádku tabulky (`int value`) obsahuje proxy třídu daného řádku (`RowProxy row`). Takto sléváme pouze akumulované hodnoty a při porovnání dvou struktur porovnáváme pouze jejich položky `row`, protože indexy v poli struktury mají stejnou hodnotu klíčů porovnání. Řešení je obsaženo v třídě `ABTreeAccumSorterHalfStreamed`.

Třídy řešení se využívají i k jednovláknovému zpracování. V takovém případě existuje pouze jedna tabulka a jedna stromová struktura.

Řešení Streamed

Řešení je obsaženo v třídě `ABTreeStreamedSorter<T>` a odpovídá řešení z sekce analýzy 2.9.5. `T` je zde typ prvního klíče třídění, podle kterého budeme rozdělovat rozsahy přihrádkám. Potomci této třídy jsou řešení specializované na druh vyhledávacího stromu. Definuje:

- Položku `RangeBucket[] rangeBuckets`, která odpovídá definovaným přihrádkám. `RangeBucket` interně obsahuje `ITableResults resTable` (lokální tabulka výsledků prohledávání) a `IABTree<int> tree` (lokální indexační struktura).

- Položku `ExpressionReturnValue<T> firstKeyExpression` představující výraz prvního klíče třídění. Výraz se vypočte a následně se jeho hodnota použije k zjištění správné přihrádky.
- Položku `ExpressionComparer<T>[] firstKeyComparers`, která obsahuje odkaz na porovnávače prvních klíčů použitých ve stromech uvnitř přihrádek. V moment výpočtu prvního klíče nastavíme výslednou hodnotu výrazu položkám porovnávače, aby nedocházelo k opětovnému výpočtu výrazu.
- Metodu `RangeBucket CreateBucket()`, kterou implementují potomci a vytváří v ní specializovaný druh vyhledávacího stromu.
- Položku `TypeRangeHasher<T> firstKeyHasher`, která na základě hodnoty prvního klíče třídění určí správnou přihrádku výsledku (volání metody `int Hash(T value)`).
- Implementaci metody `Process(Element[] result, int matcherID)`. V moment nalezení výsledku vlákno vypočte hodnotu `firstKeyExpression` pomocí `result` a určí přihrádku voláním metody `Hash` s vypočtenou hodnotou výrazu. Uzamkne přihrádku a vloží výsledek do tabulky v přihrádce. Nastaví položky porovnávače uvnitř stromu a následně vloží index řádku do stromu.

Výpočet indexu přihrádky je zpracován potomky třídy `TypeRangeHasher`. Třída pouze definuje Factory metodu [17, str. 107] `Factory(int threadCount, Type type)`. `threadCount` říká kolik vláken bude přistupovat k přihrádkám a je omezen dle analýzy na 64. `type` určuje jaký typ hodnoty třída zpracovává. Z třídy dědí abstraktní třída `TypeRangeHasher<T>`, která definuje abstraktní metodu `int Hash(T value)`. `T` je zde `type` z metody `Factory`. Dále definuje položku `int BucketCount`, který určuje počet existujících přihrádek. Potomek dané třídy je `IntRangeHasher`, která představuje rozdělení typu `Int32` dle analýzy. Dále třídě je potomkem třída `AsciiStringRangeHasher`, která představuje rozdělení typu `string` dle analýzy.

Výsledně potomci `ABTreeSorterStreamed<T>` specializují použitý strom.

- Řešení **Streamed: ABTree**: využívá `ABTree` jako `IABTree`. Řešení je obsaženo v třídě `ABTreeGenSorterStreamed<T>`.
- Řešení **Streamed: ABTreeAcccumulator**: využívá interně strom `ABTreeAccumulator` jako `IABTree`. Řešení je obsaženo v třídě `ABTreeAccumSorterStreamed<T>`.

Třídy řešení se využívají i k jednovláknovému zpracování. V takovém případě existuje pouze jedna tabulka a jedna stromová struktura. Zároveň, v jednovláknovém zpracování se řešení shodují i s řešeními **Half-Streamed**. Při prezentaci výsledků v kapitole 4 je na to důležité brát zřetel.

3.4.12 Úprava Group by

V této sekci popíšeme implementaci úprav části dotazu *Group by*. Část dotazu *Group by* je reprezentována třídou `GroupByResultProcessor`. Potomci specifikují navržené přístupy zpracování dle sekce analýzy 2.10.

Řešení Half-Streamed

V analýze jsme se rozhodli využít seskupování způsobem Two-step *Group by*. Vytvořili jsme dva ekvivalenty daného řešení, které se liší využitým úložištěm:

- Řešení **Half-Streamed: Two-step (Bucket)**, které využívá Bucket jako úložiště a je obsaženo v třídě `TwoStepHalfStreamedBucket`. Je ekvivalentem řešení **Normal: Two-step (Bucket)**.
- Řešení **Half-Streamed: Two-step (List)**, které využívá List jako úložiště a je obsaženo v třídě `TwoStepHalfStreamedListBucket`. Je ekvivalentem řešení **Normal: Two-step (List)**.

Oba řešení ve výsledcích pro jednovláknové zpracování mají v názvu **SingleThreadSolution** místo **Two-step**. Řešení obecně fungují totožně jako jejich ekvivalenty a rovněž používají stejná úložiště agregačních funkcí, proto popíšeme krátce rozložení třídy a hlavní rozdíly. Obě třídy definují:

- Položku `GroupJob[] groupJobs`, která představuje lokální výsledky vláken. Každému vláknu v průběhu prohledávání grafu náleží objekt algoritmu DFS `DFSParallelPatternMatcherStreamed` s identifikátorem v položce `int matcherID`. Tyto `ID` vznikly na základě pořadí vytvoření objektů. Tedy vlákno vždy při volání metody `Process` přistupuje k objektu `groupJobs[matcherID]`. Objekt obsahuje `ITableResults resTable` (lokální tabulka výsledků prohledávání) a `Dictionary<GroupDictKey, ...> groups` s úložištěm agregačních funkcí (lokální výsledky seskupování).
- Položku
  `ConcurrentDictionary<GroupDictKeyFull, ...> globalGroups`,
  do které se slévají výsledky, když vlákno dokončí prohledávání grafu. Zde je první rozdíl s řešeními módu **Normal**. V původním řešení jsme použili při slévání výsledků do paralelní hašovací tabulky strukturu `GroupDictKey` jako klíč seskupení. Nyní využíváme `GroupDictKeyFull`. Struktura rozšiřuje `GroupDictKey` tak, že místo indexu řádku obsahuje proxy třídu řádku. Využíváme danou strukturu, protože při slévání výsledků je sléváno několik tabulek `ITableResults`. Kdybychom použili původní strukturu, tak bychom nedokázali rozeznat, které tabulce řádek náleží.
- Implementaci metody `Process(Element[] result, int matcherID)`, ve které dochází v první části k lokálnímu seskupování. V moment kdy vlákno dokončí prohledávání grafu začne slévat své lokální výsledky do položky `globalGroups`. Slévání vykonává metoda `MergeResults(GroupJob job, int matcherID)`. V jednovláknovém zpracování nedochází ke slévání.

Hlavní rozdíl, kromě jiné struktury klíče při slévání, je způsob ukládání výsledků prohledávání do tabulky `ITableResults` v první části. Místo aby se výsledek `result` překopíroval prvně do tabulky (`resTable.StoreRow(result)`), tak je uložen do položky `resTable.temporaryRow`. Porovnání v hašovací tabulce vyvolá přístup k dané položce a ne hodnotám v tabulce. Pokud položka neexistuje v hašovací tabulce, tak je pole `temporaryRow` překopírováno do tabulky voláním metody `resTable.StoreTemporaryRow()`. V opačném případě je položka nastavena na `null`. To odpovídá návrhu z sekce analýzy 2.10.

Řešení Streamed

Řešení vychází z řešení Global *Group by* a pojmenovali jsme jej **Streamed: Global** a je implementováno třídou `GlobalGroupByStreamed`. Jednovláknové zpracování je rovněž obsaženo v dané třídě. Pokud je řešení použito v jednovláknovém kontextu označíme jej **Streamed: SingleThreadSolution**. V tomto řešení vlákna seskupují výsledky pomocí `ConcurrentDictionary<key, value>`. V jednovláknovém řešení je seskupováno pomocí `Dictionary<key, value>`. Hlavním rozdílem od původních řešení je typ `key` a `value`. Zde jsou oba rovny typu `AggregateBucketResult[]`. Metody vkládání jsou volány s odkazem na stejné pole v `key` i `value`. Ačkoliv je to zbytečné, nenašli jsme ekvivalent thread-safe struktury, který by umožnil práci pouze s jedním záznamem. Pro `Dictionary` existuje ekvivalent s jedním záznamem `HashSet`, ale protože již používáme původní paralelní hašovací tabulku, tak využijeme i `Dictionary`.

Hlavní myšlenka z analýzy byla mít pole obsahující v prvních n položkách hodnoty klíčů seskupení a ve zbylých m výsledky agregačních funkcí. To jsme realizovali zmíněným polem `AggregateBucketResult[]`. Objekty úložiště agregačních funkcí jsou totožná s již navrhnutými. Pro objekty hodnot klíčů jsme vytvořili nové objekty rozšířením třídy `AggregateBucketResult<T>`.

Třída AggregateBucketResultStreamed<T>

Třída reprezentuje hodnotu výrazu jednoho klíče seskupení a dědí z třídy `AggregateBucketResult<T>`. Hodnota klíče je uložena ve zděděné položce `T aggResult`. Definuje:

- Položku `bool isSet`, která určuje, zda výpočet výrazu byl úspěšný. Při neúspěchu `aggResult` obsahuje hodnotu `default`.
- Imlementaci metody `int GetHashCode()`, která vyvolává stejnojmennou metodu na zděděném prvku `aggResult`. Metoda se používá k získání haše výsledku prohledávání při vkládání do hašovací tabulky.

Při vkládání pole do hašovací tabulky musíme prvky porovnat. Implementovali statickou třídu `AggregateBucketResultStreamedComparers`. Třída má metodu `bool Equals(Type type, aggBucket x, aggBucket y)`, kde `aggBucket` je roven `AggregateBucketResult`. Na základě `type` se přetypují prvky `x` a `y` na `AggregateBucketResultStreamed<T>`. Následně se vyvolá porovnání hodnot.

Třída BucketsKeyValueFactory a třída BucketKeyFactory

Třída vytváří pole `AggregateBucketResult[]`, které se vkládá do hašovací tabulky. Obsahuje:

- Položku `int keysCount`, která obsahuje počet klíčů v poli.
- Položku `Aggregate[] aggregates`, na základě které se vytváří úložiště hodnot agregačních funkcí.
- Položku `AggregateBucketResult[] lastBucketsKeyValue`, která je odkaz na poslední vytvořené pole.

- Metodu `AggregateBucketResult[] Create(Element[] result)`, která vytváří pole objektů hodnot klíčů a úložišť agregačních funkcí.
- Položku `bool lastWasInserted`, která říká, zda poslední vytvořené pole bylo vloženo do hašovací tabulky. Pokud bylo vloženo, tak se při volání metody `Create` vytváří pole úplně nové. Pokud nebylo, tak se nevytváří pole nové, ale pouze se přepíšou hodnoty klíčů v prvních `keysCount` objektech uvnitř pole `lastBucketsKeyValue`. Tímto vytváříme pole jen když je to nutné.
- Položku `BucketKeyFactory[] factories`, je pole objektů, které vytvářejí objekty klíčů (`AggregateBucketResultStreamed<T>`). Každý objekt v tomto poli vytváří jeden objekt hodnoty klíče pro výsledné pole uvnitř metody `Create(...)`.

Objekt `BucketKeyFactory` definuje metodu `AggregateBucketResult Create(bool lastWasInserted, Element[] ..)`, která vytváří jeden objekt úložiště klíče. Z třídy dědí `BucketKeyFactory<T>`, která obsahuje výraz `ExpressionReturnValue<T> expr` použitý k získání hodnoty jednoho klíče. Dále obsahuje odkaz na poslední vytvořený objekt hodnoty klíče `AggregateBucketResultStreamed<T> lastCreatedBucket`. Tento objekt je obsažen v poli `lastBucketsKeyValue` uvnitř třídy `BucketsKeyValueFactory`. Při volání metody `Create(..)` se buď vytvoří nový objekt nebo se přepíší jen jeho vnitřní hodnoty na základě položky `lastWasInserted`.

Třída GlobalGroupByStreamed

Třída obsahuje implementaci řešení **Streamed: Global**. Třída definuje:

- Položku `ConcurrentDictionary<..., ...> parGroups`, která je použita k seskupování v paralelním řešení.
- Položku `Dictionary<..., ...> groups`, která je použita k seskupování v jednovláknovém řešení.
- Položku `BucketKeyFactory[] matcherBucketFactories`, kde každý záznam odpovídá třídě vytvářející pole vkládané do hašovací tabulky jednoho vlákna. To znamená, že každé vlákno má svůj objekt opět přístupný pomocí `matcherID`. Každé vlákno vlastní svůj objekt, protože při použití sdíleného objektu by vlákna přepisovala objekty uvnitř pole.
- Implementaci metody `Process(Element[] result, int matcherID)`. Vlákno volající metodu vytvoří vkládané pole pomocí svého objektu `matcherBucketFactories[matcherID]` a vloží jej do hašovací tabulky.

3.4.13 Úprava Single group Group by

V této sekci popíšeme implementaci úprav *Single group Group by*. Při úpravě *Single group Group by* jsme postupovali přesně podle sekce analýzy 2.11. Vytvořili jsme dva řešení. Jedno řešení pro mód **Half-Streamed** a jedno pro mód **Streamed**. Řešení jsou:

- Řešení **Half-Streamed: SingleGroup** představuje *Single group Group by* **Half-Streamed** módu. Je implementováno v třídě `SingleGroupGroupByHalfStreamed`.
- Řešení **Streamed: SingleGroup** představuje **Streamed** *Single group Group by*. Je implementováno v třídě `SingleGroupGroupByStreamed`.

Řešení se liší pouze paralelním zpracování. V jednovláknovém zpracování jsou totožná. U obou řešení se používá úložiště Bucket, protože se vždy jedná o jednu skupinu. Obě třídy dědí z třídy `GroupByResultProcessor`.

Třída SingleGroupGroupByHalfStreamed

Třída definuje hlavní položky:

- Položku `AggregateBucketResult[][] matcherNonAsterixResults` představující lokální úložiště agregačních funkcí vláken. Každé vlákno ukládá výsledky funkcí při volání metody `Process` do svého lokálního úložiště `matcherNonAsterixResults[matcherID]`. Tato položka neobsahuje úložiště pro funkci `count(*)`, abychom nemuseli vyvolávat metody logiky této funkce.
- Položku `int[] numberOfMatchedElements`, která obsahuje lokální úložiště pro výpočet funkce `count(*)`. Vlákna tedy při zpracování této funkce vykonají pouze vykonají navýšení této položky.
- Položku `AggregateBucketResult[] finalResults`, do kterého se budou výsledky vláken slévat. Tato položka obsahuje již úložiště pro `count(*)`.
- Implementaci metody `Process(Element[] result, int matcherID)`. Metoda přistoupí k lokálnímu úložišti vlákna. Vlákno zpracuje výsledek logikou tříd `Aggregate` a následně výsledek zahodí, tj. neukládá ho do tabulky. Volají se zde normální funkce zpracování, protože se jedná o lokální výsledky. Vlákno po dokončení prohledávání slévá výsledky do položky `finalResults`, tj. nečeká až ostatní vlákna dokončí prohledávání. Při slévání se využívají thread-safe verze zpracování.

Třída SingleGroupGroupByStreamed

Třída definuje hlavní položky:

- Položku `AggregateBucketResult[] nonAsterixResults`, která představuje globální úložiště hodnot agregačních funkcí všech vláken. Neobsahuje úložiště pro funkci `count(*)`, abychom nemuseli vyvolávat metody logiky této funkce.
- Položku `int numberOfMatchedElements`, která představuje globální úložiště hodnot funkce `count(*)`. Vlákna tedy při zpracování této funkce vykonají pouze atomické navýšení této položky.
- Položku `AggregateBucketResult[] finalResults`, která obsahuje finální výsledky agregačních funkcí. Tyto výsledky již obsahují úložiště pro funkci `count(*)`.

- Položku `int matchersFinished` udávající počet vláken, která dokončila prohledávání grafu.
- Implementaci metody `Process(Element[] result, int matcherID)`. Vlákno zpracuje výsledek do úložiště `nonAsterixResults` pomocí thread-safe verzí logiky funkcí. Výsledek se neukládá do tabulky, ale pouze zahodí. Poslední pracující vlákno vyvolá uložení výsledků z `nonAsterixResults` do položky `finalResults`.

3.5 Překlad a spuštění enginu

Překlad

Překlad se vykonává z prostředí aplikace Visual Studia 2019. K překladu je nutné volit Release mód a platformu x64. Předpokládáme, že aplikace se používá na systému Windows 10. Vstupní soubory v adresáři *DataFiles* se při překladu překopírují do složky s přeloženou aplikací.

Vstupní soubory

Aplikace očekává čtyři vstupní soubory definované v sekcích 2.2.3 a 3.4.2 ve složce *DataFiles*. Schéma hran v souboru *NodeTypes.txt*. Schéma vrcholů v souboru *EdgeTypes.txt*. Pro schémata jsme definovali JSON formát v sekci 2.2.3. Obecně hrana a vrchol nemůžou mít stejný štítek. Typy vrcholů a hran mohou však sdílet vlastnost. Typy vlastností jsme definovali na `integer` a `string`. Datový soubor hran v souboru *Edges.txt*. Datový soubor vrcholů v souboru *Nodes.txt*. Soubory mají formát definovaný formát v sekci 2.2.3 a 3.4.2. Očekává se, že oddělovače sloupečků jsou mezery. Každé `ID` elementu musí být unikátní. Názvy vlastností a typů elementů v Property grafu nesmí obsahovat čísla. Nedodržení formátu má za následek nedefinované chování.

Argumenty programu

Aplikace má celkově 9 argumentů. Prvních 7 je povinných a zbylé dva jsou povinné jen v určitých situacích. Následuje výpis argumentů. Pořadí výpisu určuje pořadí zadání při spuštění.

1. Povinný argument **Mode:** argument definuje mód zpracování dotazu. Ve zbytku výpisu budeme využívat dané zkratky k označení módů.

Hodnota argumentu	**Mód**
n	Normal
hs	Half-Streamed
s	Streamed

Tabulka 3.1: Tabulka argumentu **Mode**.

2. Povinný argument **SorterAlias:** argument definuje způsob zpracování části *Order by*. Každý mód má svá vlastní řešení. Názvy řešení odpovídají názvům z implementace. Hodnota ve sloupečku **Argument** udává hodnotu argumentu.

Mód	Argument	Řešení
n	mergeSort	Normal: Merge sort
hs	abtreeHS	Half-Streamed: ABTree
hs	abtreeAccumHS	Half-Streamed: ABTreeAcccumulator
s	abtreeS	Streamed: ABTree
s	abtreeAccumS	Streamed: ABTreeAcccumulator

Tabulka 3.2: Tabulka argumentu **SorterAlias**.

3. Povinný argument **GrouperAlias:** argument definuje způsob zpracování části *Group by*. Každý mód má svá vlastní řešení. Názvy řešení odpovídají názvům z implementace. Hodnota ve sloupečku **Argument** udává hodnotu argumentu. Pokud bylo zadáno „refL“ nebo „refB“, pak je *Group by* vykonáno jedním vláknem i když bylo definováno paralelní zpracování v argumentu **ThreadCount**. Výběr zpracování *Single group Group by* závisí pouze na výběru módu, protože každý mód má pravě jedno takové řešení.

Mód	Argument	Řešení
n	refB	Normal: SingleThreadSolution (Bucket)
n	refL	Normal: SingleThreadSolution (List)
n	localB	Normal: LocalGroupByLocalTwoWayMerge (Bucket)
n	localL	Normal: LocalGroupByLocalTwoWayMerge (List)
n	globalB	Normal: Global (Bucket)
n	twpstepB	Normal: Two-step (Bucket)
n	twostepL	Normal: Two-step (List)
hs	twostepHSB	Half-Streamed: Two-step (Bucket)
hs	twostepHSL	Half-Streamed: Two-step (List)
s	globalS	Streamed: Global

Tabulka 3.3: Tabulka argumentu **GrouperAlias**.

4. Povinný argument **FixedArraySize:** definuje velikost bloků fixní velikost uvnitř tabulky výsledků (viz sekce 2.5.5, 3.4.6 a 3.4.5). Argument má číselnou hodnotu. Minimální hodnota je 1 a maximální je `Int32.MaxValue`.

5. Povinný argument **ThreadCount:** definuje počet použitých vláken k zpracování dotazu. Argument má číselnou hodnotu. Minimální počet je 1 a maximální počet je 64.

6. Povinný argument **Printer:** definuje místo výpisu výsledků dotazu. Existují dvě možnosti: „console“ a „file“. První vypíše výsledky dotazu uživateli do konzolového okna aplikace. Druhý vypíše výsledky dotazu uživateli do souboru. Název souboru je definován jako poslední argument programu.

7. Povinný argument **Formater:** definuje formát výpisu výsledků dotazu ve formě tabulky. Existují dvě možnosti: „simple“ a „markdown“. Hlavička tabulky obsahuje výrazy s *Select* části dotazu. První vypíše tabulku výsledků ve formátu, ve kterém jsou sloupečky odděleny mezerami. Druhý vypíše tabulku výsledků ve formátu Markdown[1] tabulky.

8. Nepovinný argument **VerticesPerThread:** argument je povinný, pokud je argument **ThreadCount** větší než jedna. Argument má číselnou hodnotu. Minimální hodnota je 1 a maximální je `Int32.MaxValue`. Argument udává počet vrcholů přidělených vláknům k prohledání v části *Match* (viz sekce 2.5.4 a 3.4.5 (`VertexDistributor`)).

9. Nepovinný argument **File:** argument je povinný, pokud je argument **Printer** roven „file“. Argument definuje jméno souboru, do kterého se vypíšou výsledky dotazu. Součástí jména není koncovka souboru (např. .txt nebo .cs). Koncovka souboru je určena na základě argumentu **Formater**. Při hodnotě „simple“ se vytvoří soubor s koncovkou .txt a při „markdown“ se vytvoří soubor s koncovkou .md.

Následuje příklad argumentů programu:

```
Jednovláknové spuštění s výpisem do konzole:
./QueryEngine.exe n globalB mergeSort 4194304 1 console simple
Paralelní spuštění s výpisem do konzole:
./QueryEngine.exe n globalB mergeSort 4194304 8 console simple 512

Spuštění s výpisem do souboru:
./QueryEngine.exe n globalB mergeSort 4194304 1 file simple newfile
Spuštění s výpisem do souboru + paralelní zpracování:
./QueryEngine.exe n globalB mergeSort 4194304 8 file simple 512 newfile
```

Zadání dotazu

Po spuštění aplikace se načtou vstupní soubory. Po načtení je uživatel vyzván pomocí zprávy, aby zadal vstupní dotaz k vykonání:

```
Enter Query:
```

Uživatel zadá dotaz pomocí jazyka PGQL z sekce 1.2. Názvy proměnných, elementů Property grafu a vlastností jsou citlivé na velikost písmene. Samotné názvy částí dotazu a funkcí nejsou citlivé na velikost písmene. Názvy proměnných nesmí obsahovat čísla. Každý dotaz musí být ukončen znakem „;“. Stisknutím klávesy „Enter“ dojde k zahájení zpracování dotazu. Po dokončení je uživatel vyzván pomocí zprávy, aby zadal další dotaz nebo ukončil aplikaci:

```
Do you want to continue with another query?
    y/n (single character answer):
```

Program očekává odpověď jedním znakem. „y“ vyzve uživatele k zadání nového dotazu. „n“ ukončí aplikaci. Stisknutím klávesy „Enter“ se vykoná daná akce.

[1] `https://www.markdownguide.org/`

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 náležící 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 [22]. Grafy jsme primárně vybírali na základě počtu nalezených výsledků z testovaného dotazu *Match* části (sekce 4.2.1). Počet vygenerovaných výsledku je zobrazen v sekci výsledků 4.4.1. Cíl byl nalézt grafy, které vygenerují počet výsledků v řádech 10^7 až 10^8, abychom mohli sledovat chování řešení při zpracování velkého množství dat. Zároveň jsme vybrali grafy pro které platí, že mají minimálně dvojnásobný počet výsledků vůči již vybraným grafům, což nám umožní sledovat chování řešení při zvyšování počtu zpracovaných výsledků. Výsledky v řádu 10^9 bychom měli problém zpracovat na testovaném hardwaru (sekce 4.3.3), kvůli nedostatku paměti. Samotná databáze SNAP nám poskytla datový formát, který jsme byli schopni jednoduše transformovat na náš vstupní datový formát.

	#Vrcholů	**#Hran**
Amazon0601	403 394	3 387 388
WebBerkStan	685 230	7 600 595
As-Skitter	1 696 415	11 095 298

Tabulka 4.1: Vybrané grafy pro experiment

- **Amazon0601:** Jedná se o graf vytvořený procházením webových stránek Amazonu na základě funkce „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 vlastnosti 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
```

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" }
]
```

Soubory obsahují datové schéma v JSON formátu definovaném v sekci 2.2.3. Soubor *EdgeTypes.txt* definuje jeden druh hrany `BasicEdge` bez vlastností. Soubor *NodeTypes.txt* definuje jeden druh vrcholu `BasicNode` bez vlastností.

Generování souborů *Nodes.txt*/*Edges.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 první sloupeček v datových souborech *Edges.txt* a *Nodes.txt* odpovídá unikátnímu `ID` v rámci celého grafu. Výsledný soubor *Nodes.txt* definuje vrcholy grafu. Řádek představuje jeden vrchol. Soubor má dva sloupečky. První obsahuje `ID` vrcholů a druhý obsahuje název typu. Soubor *Edges.txt* je ekvivalentní, ale obsahuje `ID` hran a jejich typ. Zároveň pak obsahuje dva sloupečky navíc, které definují směr hrany. První sloupeček udává `ID` počátečního vrcholu a druhý sloupeček určuje `ID` koncového vrcholu. Formát je přesněji definován v sekci 2.2.3.

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 vlastností vrcholů

Posledním krokem přípravy dat pro experiment je vygenerování vlastností vrcholů. Rozhodli jsme se pro generování vlastních hodnot. Budeme mít tak lepší znalost použitých dat. Znalost využijeme k vhodnějšímu otestování vybraných případů. Navíc, 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 vlastností.

Vlastnost	**Typ**	**Popis**
PropOne	integer	`Int32` s rozsahem $[0; 100\ 000]$
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 vlastností vrcholů

- **PropOne** hodnoty jsou generovány pouze v rozsahu $[0; 100\ 000]$. Neobsahují negativní hodnoty.
- **PropTwo** hodnoty jsou generovány střídavě kladně a záporně, aby nastal rovnoměrný počet záporných a kladných hodnot.
- **PropThree** hodnoty jsou pouze ASCII znaky z rozsahu $[33; 126]$. Dané omezení vyplývá z vlastností dotazovacího enginu, aby bylo možné bez obtíží načíst datový soubor.
- **PropFour** hodnoty jsou generovány pouze v rozsahu $[0; 1000]$. Neobsahují negativní hodnoty.

Vlastnosti **PropOne**, **PropTwo** a **PropFour** jsme vybrali primárně k otestování části *Group by*. Použijeme je jako klíč *Group by*. Tímto omezíme počet vytvářených skupin během zpracování. Detailnější vysvětlení je podáno v následující sekci výběru dotazů *Group by* 4.2.3. **PropThree** využijeme v části *Order by*, abychom mohli sledovat rozdílnost třídění řetězců vůči číselným hodnotám.

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

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

Výsledné hodnoty vlastností 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*. K inicializaci generátoru náhodných čísel pro každý graf jsme použili různé hodnoty. Zvolené inicializační hodnoty byly vygenerovány rovněž náhodně.

	Inicializační hodnota
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)
    };
```

Tímto jsme dokončili poslední nutný krok k vygenerování platných vstupních dat pro dotazovací engine. 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 uváděny 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 režii 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).

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

4.2.3 Dotazy Group by

Pro výpočet agregačních funkcí jsme zvolili funkce `min` a `avg`, protože představují netriviální práci na rozdíl od funkcí `sum`/`count` (jedno přičtení proměnné). Funkce `min` porovná a prohodí výsledek. Thread-safe verze používá mechanismus *Compare and Exchange*. Funkce `avg` provádí dvě 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_Q7$'$ je `select x.PropOne match (x) -> (y) -> (z) group by x.PropOne`).

U vybraných dotazů je nutné si uvědomit, co znamenají klíče v části *Group by*. Každý klíč má svůj rozsah hodnot. Konkrétní hodnoty klíčů jsou uloženy v našem případě ve vrcholech. Pokud by každý vrchol měl unikátní hodnotu klíče, pak počet vytvářených skupin v části *Group by* je shora omezen počtem vrcholů v grafu. To se děje pro dotazy G_Q2, G_Q3 a G_Q6. Zvolíme-li dva takové klíče za klíče *Group by*, pak maximální počet skupin je shora omezen skalárním součinem jejich rozsahů. To nastává pro dotazy G_Q4 a G_Q5. Posledním případem jsou klíče vlastností **PropOne** a **PropFour**. Dané vlastnosti obsahují hodnoty dle tabulky rozsahů 4.2. Pro testované grafy platí, že počet vrcholů (tabulka 4.1) je vždy větší než rozsah hodnot vlastností. Odtud vyplývá, že při generování vlastností vrcholů v minule sekci určitě nastala situace, při které dva vrcholy obsahují stejnou hodnotu dané vlastnosti. Tímto jsme dokázali shora omezit počet skupin v dotazech G_Q7 a G_Q8 horní hranicí intervalu rozsahu hodnot daných vlastností.

Další nutnou znalostí je způsob paralelizace prohledávání grafu (sekce 2.5.4). Každé vlákno provádí prohledávání grafu z určité části vrcholů v grafu. Platí, že každé vlákno neprovede prohledávání ze stejného vrcholu jako vlákno jiné. Tedy vrcholy představující proměnnou x jsou unikátní pro každé vlákno, taktéž hodnoty jejich vlastností. Při paralelizaci se provádí paralelní slévání výsledků do globální struktury. V dotazech G_Q3 a G_Q6 pak vlákna při slévání výsledků vytvářejí vždy nové záznamy v dané struktuře, jelikož výsledky vláken jsou vždy rozdílné. To nám umožní sledovat situaci, kdy vlákna vytvářejí navzájem rozdílné záznamy. Obecně dotazy G_Q4 až G_Q8 nám umožní otestovat chování řešení při snižování počtu vytvářených skupin. Pro G_Q3 a G_Q6 bude navíc vidět režie za porovnání vlastnosti vůči `ID`.

Následuje shrnutí:

- 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 přidávají režii za ukládání a zpracovávání 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 $[0; 100\,000]$ `Int32` (viz 4.2). Dojde k rozprostření několika stejných hodnot v grafu. Maximální počet skupin je 100 000.
- 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.

4.3 Metodika

Pro provedení experimentu jsme připravili jednoduchý benchmark v jazyce C# pro .NET Framowork 4.8, 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);  \\ Inicializace struktur dotazu.
    timer.Restart();  \\ Začátek měření.

    q.Compute();      \\ Vykonání dotazu.

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

Hlavní smyčka benchmarku se skládá z 5-ti opakování warm up fáze následovanou 15-ti opakováními měřené části. Warm up fáze vykonává identickou práci jako část měřená, tj. úplně stejný dotaz. 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* (vyhledávání vzoru) společně s částí *Group/Order by*.

Před měřením dochází vždy k úklidu haldy.

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

4.3.1 Měření uběhlého času

Protože benchmark je implementován v jazyce C# pro .NET Framework 4.8, rozhodli jsme se využít nativních možností měření uběhlé doby. Existují dvě hlavní metody měření. První využívá třídu `Stopwatch`. Třída měří uběhlý čas mezi voláním metod `Start` a `Stop` na instanci bez ohledu v jakém vlákně byly metody volány. Měření probíhá pomocí počítání taktů časovače v podkladovém mechanismu hardwaru. Pokud hardware a operační systém podporují časovač s vysokým rozlišením, pak je využit tento. V opačném případě je využit pouze systémový časovač. Druhá možnost využívá přístup k vlastnosti `Process.TotalProcessorTime` a měří dobu strávenou vykonáváním procesu aplikace procesorem. Tato doba obsahuje čas zpracování aplikační části společně s časem stráveným v jádru operačního systému. Tudíž doba obsahuje pouze čas běhu procesoru a nezapočítává dobu nečinnosti vláken v paralelních řešeních (např. čekání na zámek).

Rozhodli jsme se využít první variantu, protože lépe odráží reálný běh programu. Navíc, náš hardware a operační systém umožňují využití časovače s vysokým rozlišením.

4.3.2 Volitelné argumenty konstruktoru dotazu

Pro měření argumenty `FixedArraySize` a `VerticesPerThread` (sekce 3.5) jsme volili následovně:

	FixedArraySize	**VerticesPerThread**
Amazon0601	4 194 304	512
WebBerkStan	4 194 304	512
As-Skitter	8 388 608	1024

Tabulka 4.7: Výběr argumentů konstruktoru dotazu pro grafy

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

4.3.3 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

4.3.4 Příprava hardwaru

Každému testování předcházel studený restart 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.

4.3.5 Překlad

Benchmark společně s dotazovacím enginem a potřebnými knihovnami byl přeložen v `Release` módu Visual Studia 2019 pro platformu x64 využívající 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. Značení se skládá ze tří částí. Prvních dvě jsou obsaženy vždy a poslední je použit popuze v paralelní části *Group by*. Značení vypadá následovně:

```
[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 prohledávání grafu a vylepšené módy **Streamed**/**Half-Streamed** je vykonávají v průběhu prohledávání grafu. U paralelního řešení **Streamed** jsou výsledky zpracovány globálně, zatímco u **Half-Streamed** řešení dochází k lokálnímu zpracování, které je zakončené sléváním. Zopakování hlavních konceptů řešení a způsob ukládání ponecháme jako úvod jednotlivých částí.

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. Počet nalezených výsledků při prohledávání jednotlivých grafů je zobrazen v tabulce 4.8. Všechny výsledky měření prohledávání grafu jsou zobrazeny na obrázcích na konci této sekce.

	Počet nalezených výsledků
Amazon0601	32 373 599
WebBerkStan	222 498 869
As-Skitter	453 674 558

Tabulka 4.8: Počet nalezených výsledků pro dotazy obsahující vzor (x) -> (y) -> (z) nad jednotlivými grafy.

Paralelizace prohledávání grafu

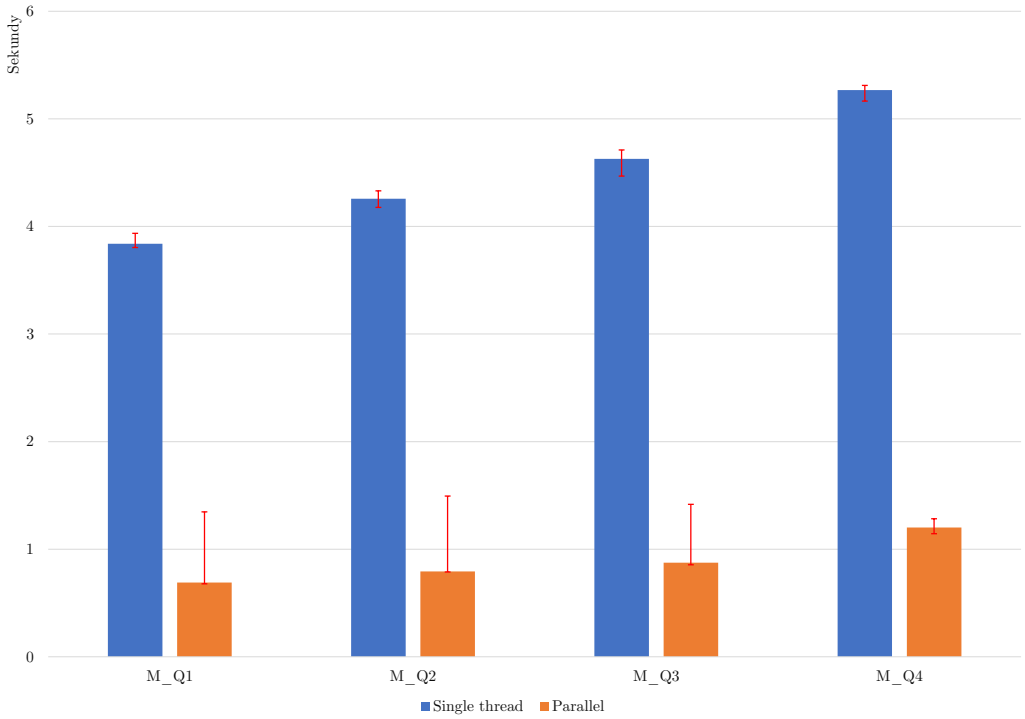
Zbytek sekce věnujeme popisu obrázků 4.1, 4.2 a 4.3. Paralelizace startovního prohledávacího vrcholu (tj. každé vlákno dostává opakovaně množství vrcholů k prohledání určené argumentem `VerticesPerThread`, dokud se nevyčerpají všechny vrcholy grafu) dociluje zrychlení v rozmezí [4,17; 5,56]-krát pro 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. U jednovláknového zpracování si navíc můžeme všimnout až lineárního zpomalení při navýšení počtu ukládaných proměnných. Pro představu, každá proměnná (element grafu) je uložena do vlastního sloupečku, který je lokální pro vlákno (`List<Element[FixedArraySize]>`).

Paralelizace slévání výsledků

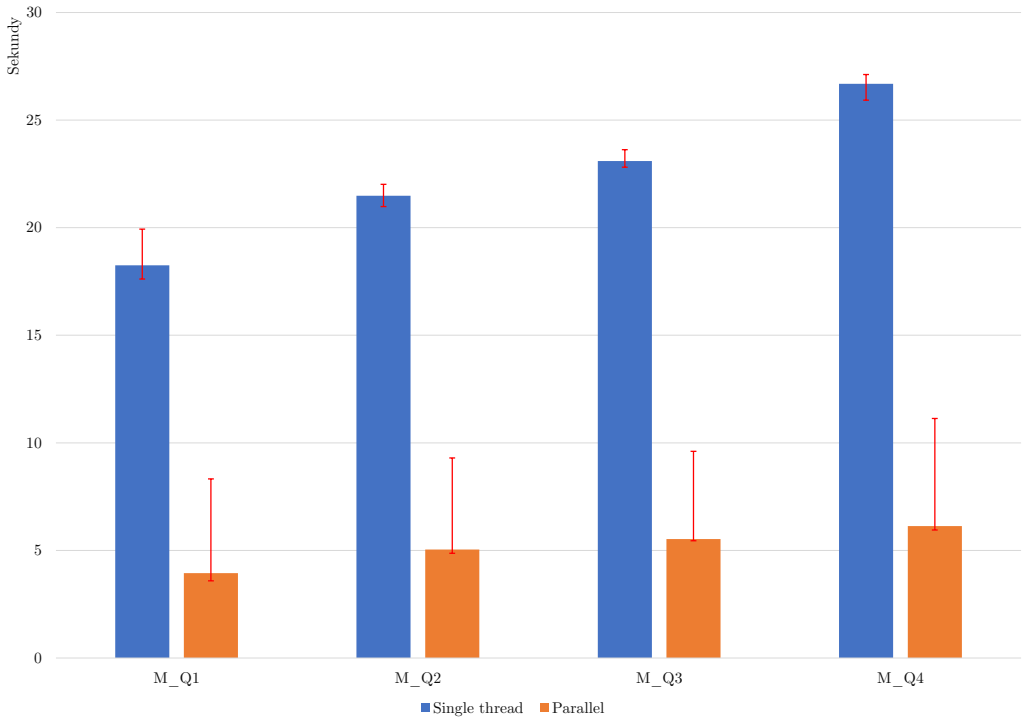
Lokalita sloupečků vede na potřebu slévání výsledků vláken. Nicméně, díky ukládání do polí fixní délky nastává nutnost pouze zarovnat poslední nezaplněná pole, zbytek práce slévání je jen přesunutí několika ukazatelů na pole. Tento proces je paralelizovaný pouze přes sloupečky. Poslední nezaplněná pole jednoho sloupečku zarovná vždy jedno vlákno. Pokud je sloupeček jen jeden, zarovnání probíhá jednovláknově. Pokud je sloupečků více, zarovnání probíhá paralelně. V tomto případě je každý sloupeček přidělen jednomu vláknu, které následně provede zarovnání.

Zpomalení paralelních řešení slévání

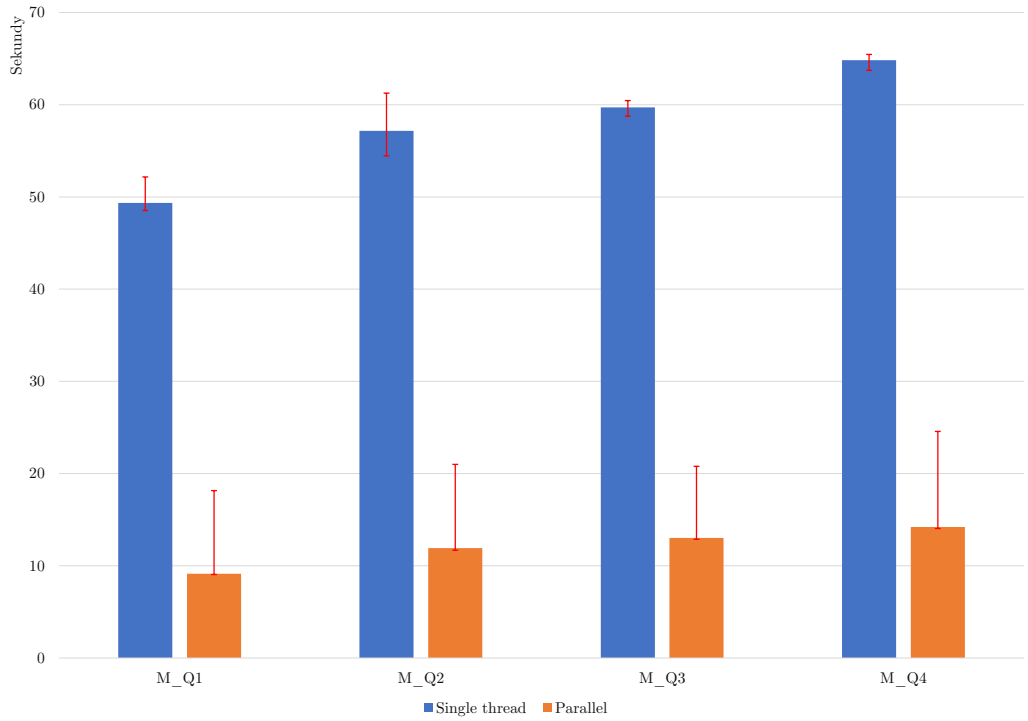
Již jsme zmínili, že nárůst počtu ukládaných proměnných zpomaluje vykonání. Nyní se podíváme ještě na zpomalení z pohledu paralelizace slévání. V paralelním řešení u dotazu M_Q2 vůči M_Q1 můžeme vidět značný skok nárůstu doby vykonání. V M_Q1 se neukládá žádná proměnná, zatímco v M_Q2 se ukládá jedna proměnná a zároveň zarovnání sloupečku probíhá pouze v jednom vlákně. Při navýšení počtu ukládaných proměnných vůči M_Q2, tj. dotazy M_Q3 a M_Q4, nedochází k tak velkému skoku jako mezi dotazy M_Q1 a M_Q2. To je právě způsobeno již zmíněnou paralelizaci zarovnání. Každý sloupeček je zde zarovnán jedním vláknem.



Obrázek 4.1: Doba vykonání dotazů *Match* pro graf Amazon0601 (sekce 4.1). Jedno vlákno vůči osmi vláknům.



Obrázek 4.2: Doba vykonání dotazů *Match* pro graf WebBerkStan (sekce 4.1). Jedno vlákno vůči osmi vláknům



Obrázek 4.3: Doba vykonání dotazů *Match* pro graf As-Skitter (sekce 4.1). Jedno vlákno vůči osmi vláknům.

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. Všechny výsledky měření třídění jsou zobrazeny na obrázcích na konci této sekce.

Obecné shrnutí jednovláknových řešení

Nejprve shrneme základní koncepty použitých řešení. Každé řešení pracuje s tabulkou výsledků, kterou musí setřídit. Nikdy netřídíme samotné řádky tabulky, ale pouze indexy řádků. Výsledek třídění je indexační struktura nad tabulkou. Při porovnávání je nutné mít na paměti, že jednovláknově bežící části používají optimalizace popsané v sekcích 2.6.4, 2.6.5 a 3.4.8.

Mód **Normal** využívá k třídění algoritmus Merge sort. Algoritmus je implementován v knihovně HPCsharp [21]. Třídění probíhá po dokončení prohledávání grafu a třídí celou tabulku výsledků pomocí pole indexů.

Vylepšená řešení zpracovávají vyhledané výsledky v moment jejich nalezení. Nalezený prvek se nejdříve vloží do tabulky na nový řádek. Následně se index daného řádku vloží do indexační struktury. Jako indexační strukturu nad tabulkou používáme (a, b)-strom [20, str. 190], u kterého jsme upravili definici na $b = 2a$. V našem případě $b = 256$. V řešení **ABTree** se jedná o obecný (a, b)-strom, zatímco řešení **ABTreeValueAccumulator** výsledky (indexy) mající stejnou hodnotu klíčů třídění, jako již vložené prvky, uloží do `List<int>`. Tedy místo vytvoření nového záznamu ve stromě dojde pouze k vložení do patřičného pole.

Výsledky jednovláknového zpracování

Začneme řešením běžícím v jednom vlákně, tj. grafy 4.4 a 4.6. 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 způsobeno značnou režií za metodu vložení (`Insert`) do stromu, ve které dochází k častému alokování nových vrcholů a překopírování prvků. 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 zatřiďování výsledků se stejnou hodnotou a tím navyšování velikosti stromu společně s počtem porovnání na `Insert`. Celý problém jsme vyřešili v řešení **ABTreeValueAccumulator**, ve kterém se duplicitní hodnoty ukládají do zmíněného pole a tím omezujeme velikost výsledného stromu. Jak vidíme na obrázcích, řešení se přibližuje rychlosti řešení **Normal**. Problém by nastal v případě, pokud by množství hodnot odpovídalo počtu nalezených výsledků. V tomto případě bychom zbytečně navyšovali režii za vzniklá pole, která se nevyužijí.

Třídění pomocí vlastnosti vůči `ID`

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

Obecné shrnutí paralelních řešení

Opět zde platí, že tabulka výsledků je tříděna pomocí indexů. Mód **Normal** používá paralelní Merge sort [21].

Vylepšená paralelní zpracování aplikují použité verze (a, b)-stromů ze zpracování pro jedno vlákno. Každé běžící vlákno v **Half-Streamed** řešení obsahuje lokální tabulku a indexační strom. Po dokončení vyhledávání se obsahy stromů překopírují do pole a dojde k paralelnímu dvoucestnému slévání používající stejnou funkci jako paralelní Merge sort. U **Streamed** řešení jsme navrhli sdílenou strukturu pro zatřiďování výsledků. Struktura rozdělí rozsah hodnot prvního klíče třídění na rovnoměrné části (konkrétněji v sekci 2.9.5) Počet částí je heuristicky zvolen jako $m = t^2$, kde $t = \#vláken$. Pro každou část rozsahu vytvoříme přihrádku obsahující zámek, tabulku a indexační strom. Při příchozím výsledku se získá hodnota prvního klíče třídění a určí se jeho patřičná přihrádka. Do přihrádky vlákno přistoupí pomocí přiloženého zámku. Následně výsledek vloží do tabulky a index nového řádku tabulky do stromu. Rozdíl v řešeních je pak jen využitá stromová struktura. Při porovnávání je nutné mít na paměti, že lokálně bežící části používají optimalizace popsané v sekcích 2.6.4, 2.6.5 a 3.4.8.

Výsledky paralelního zpracování

Nyní budeme prezentovat výsledky paralelizace (obrázky 4.5 a 4.7). Na první pohled jsme si všimli mnohonásobného zpomalení **Streamed** řešení pro dotazy O_Q1 a O_Q2. Výše jsme zmínili základní princip **Streamed** řešení. Rozsah hodnot prvního klíče třídění se rozdělí na rovnoměrné části a pro každý takový

rozsah existuje přihrádka přístupná pomocí zámku. Vlákno v moment nalezení výsledku přistupuje k přihrádce pomocí zámku a vloží do ní daný výsledek. Při implementaci řešení jsme se omezili pouze na celý rozsah hodnot C# (.NET Framework 4.8) typu vlastnosti. To znamená, že pokud vlastnost má typ `integer` (v C# `Int32`), ačkoliv hodnoty vlastnosti v grafu mohou být v rozsahu $[0; 100]$, tak rozdělení přihrádek se vytvářejí z celého rozsahu C# typu. Tedy přihrádky se vytvářejí rozdělením rozsahu [`Int32.MinValue`; `Int32.MaxValue`] a nikoliv $[0; 100]$. To má za následek při třídění pomocí vlastnosti s rozsahem hodnot v grafu $[0; 100]$, že vlákna budou přistupovat v nejhorším případě k pouze jedné přihrádce. Vlákna pak budou vždy čekat na uvolnění jednoho zámku. Přesně tato situace nastala u **Streamed** řešení pro dotazy O_Q1 a O_Q2. V nich třídíme pomocí vlastnosti `ID`. `ID` je typu `integer`. Vlastnost typu `integer` je v C# typ `Int32`. Vytvoří se přihrádky rozdělením rozsahu [`Int32.MinValue`; `Int32.MaxValue`]. Rozsah se rozdělí na 64 dílů, protože jsme určili počet přihrádek jako $m = t^2$, kde $t = \#vláken$ a všechny testy běž9 v osmi vláknech. Ale problém je, že hodnoty `ID` vrcholů jsou z omezeného rozsahu $[0; \#\text{vrcholů v grafu}]$, zatímco pouze jedna přihrádka obsahuje posloupnost 67 108 863 hodnot. A vždy platí, že 67 108 863 $>$ #vrcholů v grafu (z tabulky 4.1). Čili vlákna vždy přistupují pouze k přihrádce, kde se synchronizují pomocí zámku. Výsledná doba zpracování pak odpovídá jednovláknovému zpracování s přidanou režií za přístup k zámku.

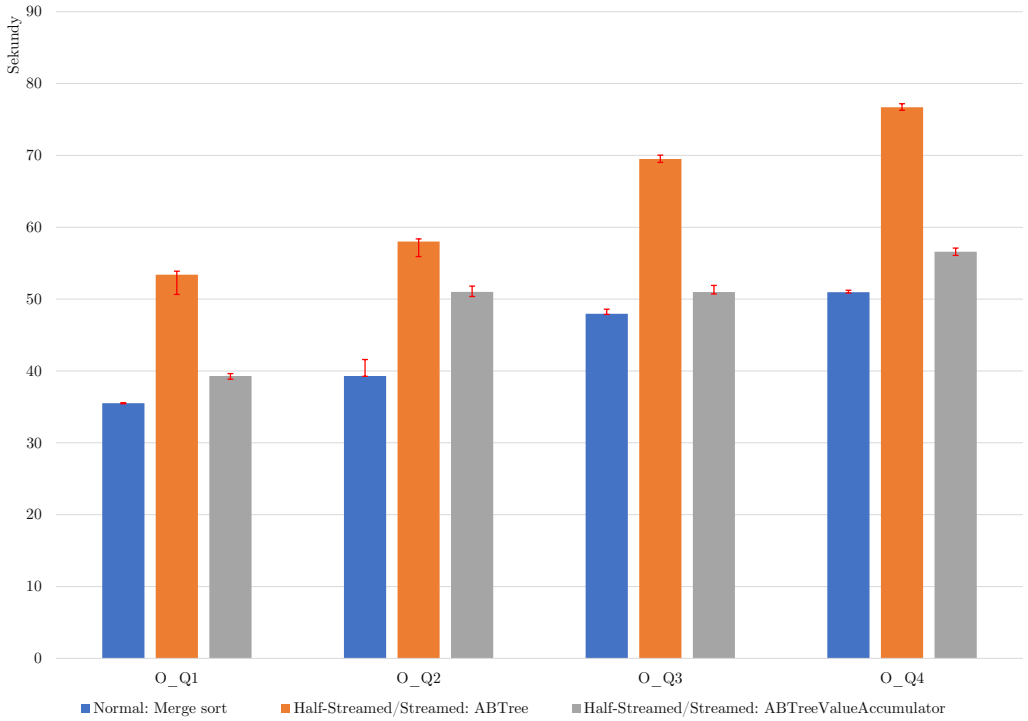
Naopak u dotazů O_Q3 a O_Q4 je tříděno pomocí hodnot vygenerovaných náhodně spadající do celého rozsahu typu klíče a zde **Streamed** řešení předčilo všechna ostatní. Pro budoucí rozšíření by bylo nutné zvážit vytvoření statistik rozsahů jednotlivých vlastností, aby bylo možné lépe vytvořit rozdělení přihrádek.

Zrychlení paralelních řešení pro O_Q3 a O_Q4

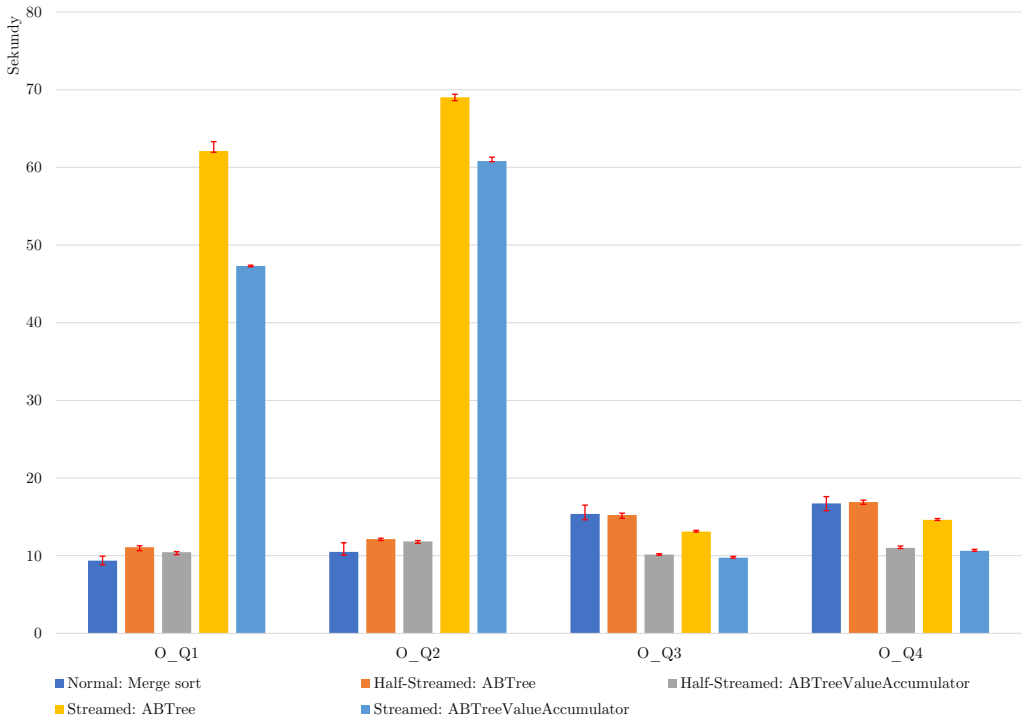
Half-Streamed řešení se přibližuje **Normal** řešení v prvních dvou dotazech a překonává jej ve třetím i čtvrtém dotazu pro řešení **ABTreeValueAccumulator**. U třetího a čtvrtého dotazu se porovnává pomocí vlastností. V jednovláknovém zpracovávání jsme viděli režii za dané porovnání. V druhém kroku u daného **Half-Streamed** řešení dochází k slévání pouze akumulovaných skupin, což rapidně sníží počet porovnávání při slévání a odtud výhoda oproti **Normal: Merge sort** řešení. To samé platí u **Streamed** řešení, protože použití přihrádek způsobí vkládání do mnohonásobně menší skupiny výsledků. Celá situace je navíc umocněna zmíněnými optimalizacemi.

Rozsah zrychlení paralelních řešení

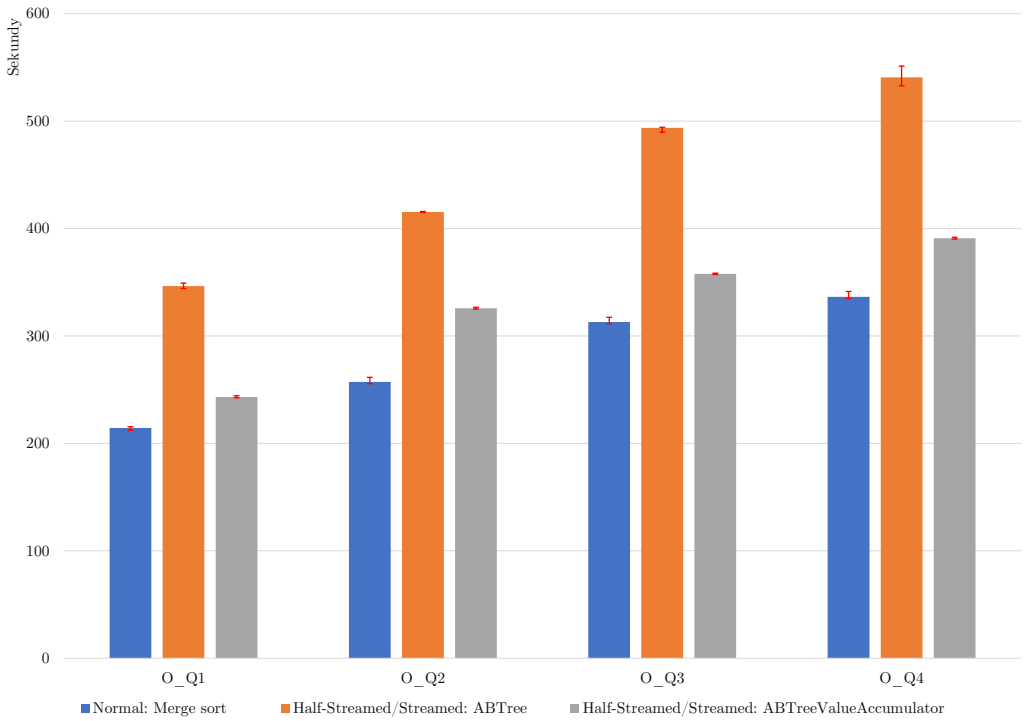
Zajímavý výsledek testování je rozsah zrychlení vylepšených módů, tj. tabulky 4.9 a 4.10). Zrychlení Merge sortu zaostává. Maximální zrychlení u ostatních řešení je až pětinásobné. Danou situaci si vysvětlujeme následovně. Implementace paralelního algoritmu Merge sort funguje na principu postupného rekurzivního rozdělování, při kterém se vytváří nové `Tasks` pro `ThreadPool`. U vylepšených řešení běží jedna metoda pro každé vlákno po dobu celého zpracování.



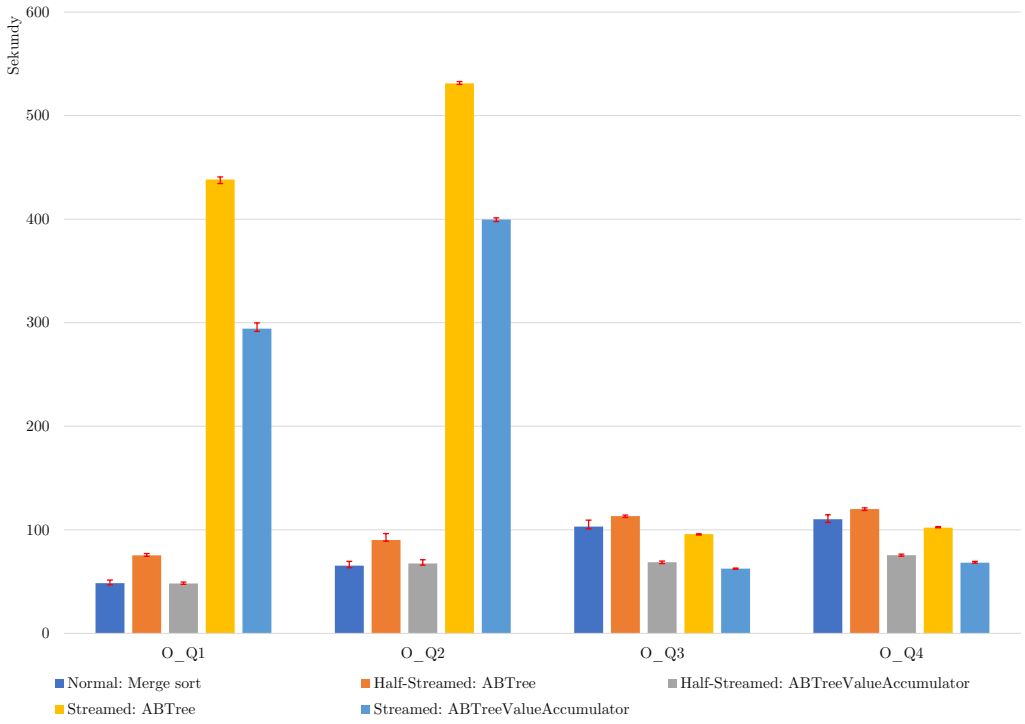
Obrázek 4.4: Doba vykonání dotazů *Order by* pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu.



Obrázek 4.5: Doba vykonání dotazů *Order by* pro graf Amazon0601 (sekce 4.1). Běh osmi vláken.



Obrázek 4.6: Doba vykonání dotazů *Order by* pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu.



Obrázek 4.7: Doba vykonání dotazů *Order by* pro graf WebBerkStan (sekce 4.1). Běh osmi vláken.

	O_Q1	O_Q2	O_Q3	O_Q4
Normal: Merge sort	3,79	3,75	3,12	3,05
Half-Streamed: ABTree	4,81	4,78	4,56	4,54
Half-Streamed: ABTreeValueAccumulator	3,76	4,31	5,03	5,14
Streamed: ABTree	0,83	0,84	5,23	5,33
Streamed: ABTreeValueAccumulator	0,86	0,84	5,30	5,25

Tabulka 4.9: Rozsah zrychlení paralelizovaných řešení pomocí osmi vláken pro dotazy *Order by* nad grafem Amazon0601 (sekce 4.1). Tabulka zobrazuje podíl jednovláknového vůči paralelnímu zpracování.

	O_Q1	O_Q2	O_Q3	O_Q4
Normal: Merge sort	4,43	3,94	3,03	3,05
Half-Streamed: ABTree	4,59	4,61	4,36	4,51
Half-Streamed: ABTreeValueAccumulator	5,05	4,83	5,21	5,18
Streamed: ABTree	0,83	0,82	5,72	5,72
Streamed: ABTreeValueAccumulator	0,79	0,78	5,15	5,29

Tabulka 4.10: Rozsah zrychlení paralelizovaných řešení pomocí osmi vláken pro dotazy *Order by* nad grafem WebBerkStan (sekce 4.1). Tabulka zobrazuje podíl jednovláknového vůči paralelnímu zpracování.

Jako důsledek testování můžeme konstatovat, že třídění během prohledávání grafu v našich vybraných případech nepřináší předpokládané výhody. Zrychlení nastává pouze u paralelizace řešení při dostatečně náhodném rozložení dat třídění, pokud je porovnáváno pomocí vlastností. Zmínili jsme, že pro stromy nastává zpomalení kvůli režii za metodu `Insert`. V našich vylepšeních jsme neimplementovali předalokování vrcholů stromu. Jestli by pomocí předalokování nastalo zrychlení, které by předčilo řešení **Normal**, ponecháme jako budoucí rozšíření.

4.4.3 Group by

Obecné shrnutí jednovláknových řešení

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 prohledávání 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 vypočtou 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. Podrobnější vysvětlení je v kapitole Analýzy 2.7.2. Výsledky jednovláknového seskupování jsou zobrazeny na obrázcích před shrnutím paralelních výsledků seskupování v sekci 4.4.3.

Výsledky jednovláknového zpracování

Na obrázcích 4.8, 4.10 a 4.12 vidíme výsledky *Group by* pro běh v jednom vlákně. Výsledky na obrázcích 4.9, 4.11 a 4.13 představují dotazy bez agregačních funkcí v části *Select*. Dvojice G_Q4/G_Q5, G_Q2/G_Q3 a trojice G_Q6/G_Q7/G_Q8 dotazů jsou pouze mírně rozličné 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.

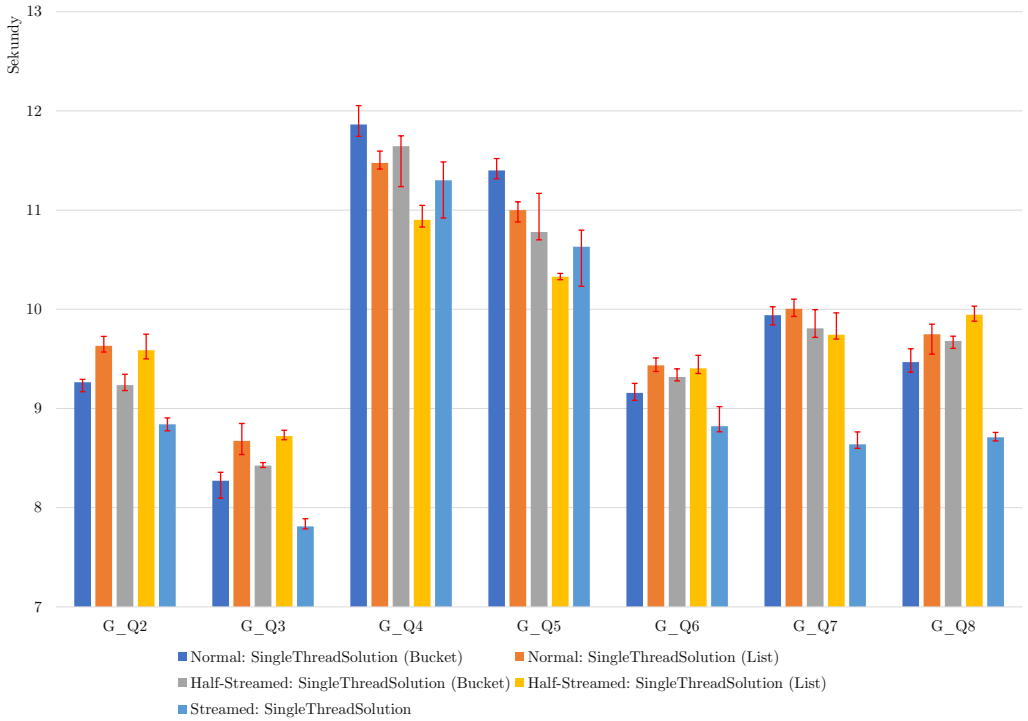
Zpomalení jednovláknového Half-Streamed řešení

Velice mírné zrychlení můžeme vidět u **Half-Streamed** řešení, které ukládá jen reprezentanty skupiny. U všech dotazů bez agregačních funkcí, kromě dvojice 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 a vysvětlujeme si ji následovně. **Half-Streamed** řešení používá při zpracování výsledku položku tabulky `temporaryRow`, do které přesouvá ukazatel 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 vidíme 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 režie **Normal** řešení je značně pomalejší, protože musí ukládat vždy dvě proměnné, zatímco **Half-Streamed** jen přesouvá ukazatel. 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.

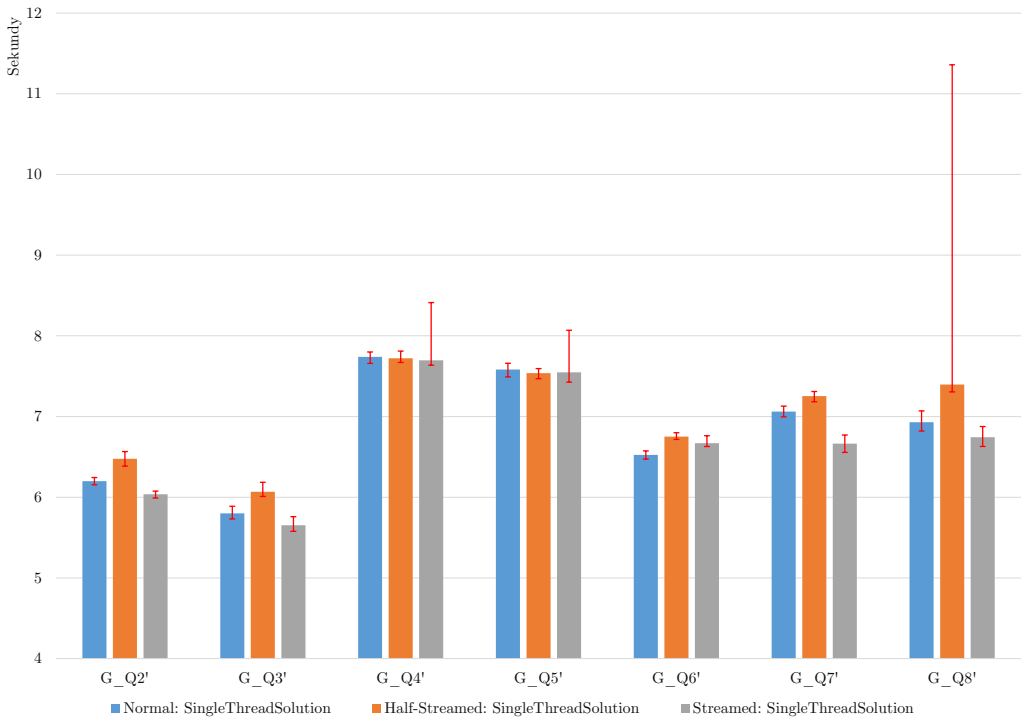
Porovnání výsledků úložišť List/Bucket jednovláknového zpracování

Na největším grafu platí, že použité ukládání List je pomalejší než Bucket, kvůli indirekci navíc. Na menších grafech rozdíly ustupují a dokonce nastávají situace, kdy je List rychlejší. Přesněji u dotazů G_Q4 a G_Q5. Přehození rolí si u nich vysvětlujeme režií 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** s úložištěm List, protože také vytváří pole jako Bucket řešení (viz implementace 3.4.12). U dalších grafů je pak počet vytvářených polí mnohonásobně menší než počet přístupů k nim.

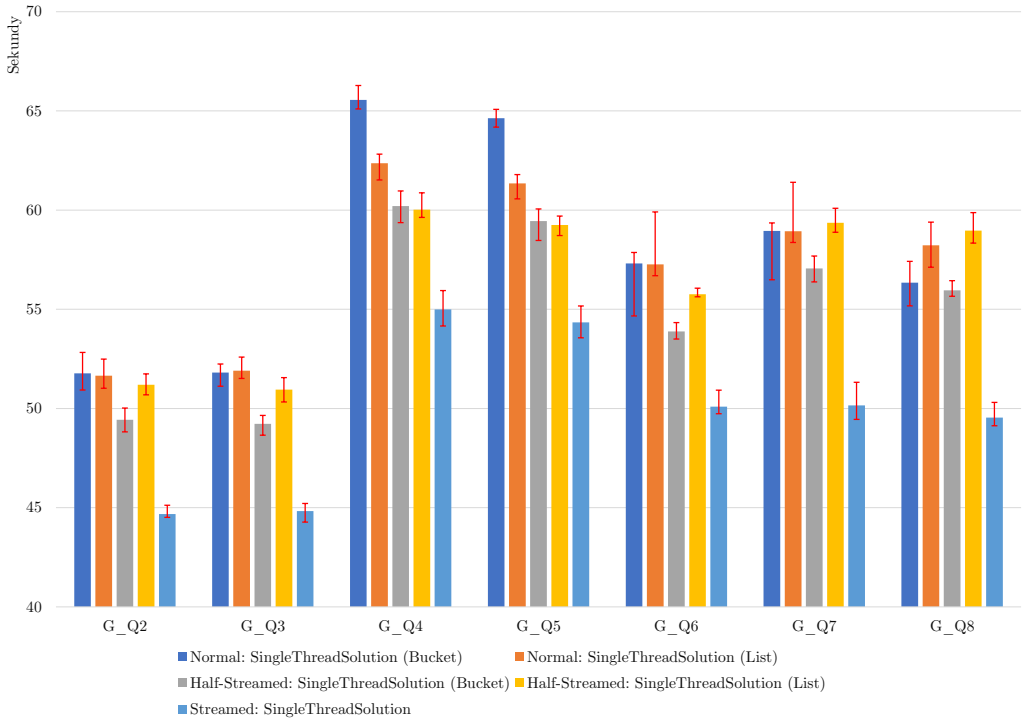
Z výsledků můžeme vyvodit, že vylepšená jednovláknová **Streamed** řešení *Group by* jsou výhodnější z hlediska rychlosti vykonávání než řešení **Normal**. Nyní přejdeme k výsledkům paralelizace.



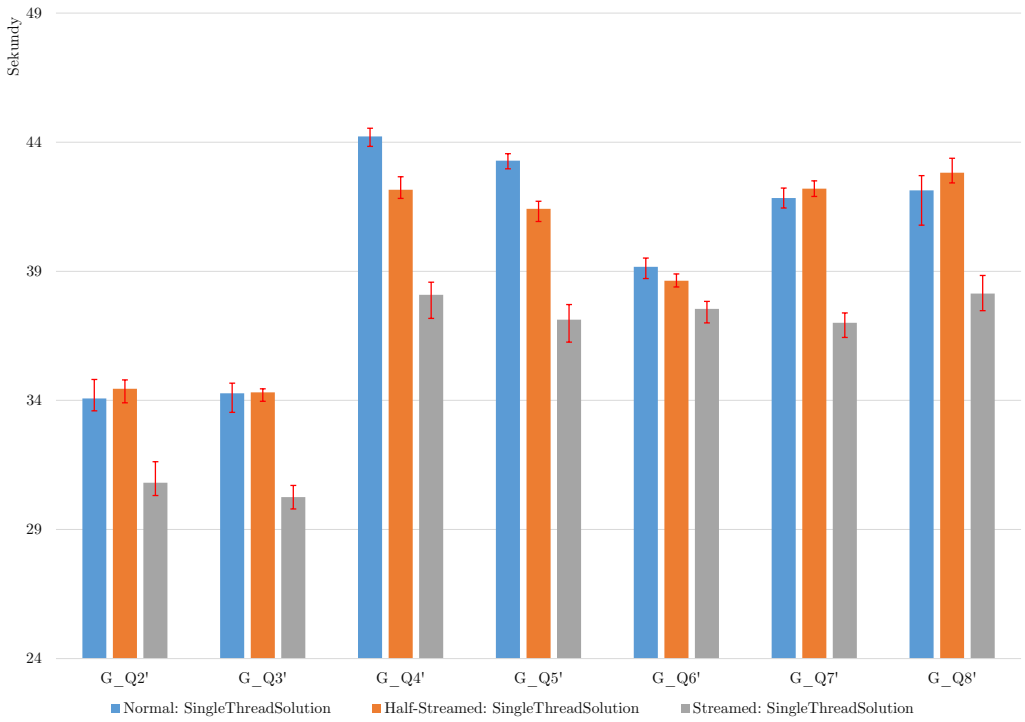
Obrázek 4.8: Doba vykonání dotazů *Group by* pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu.



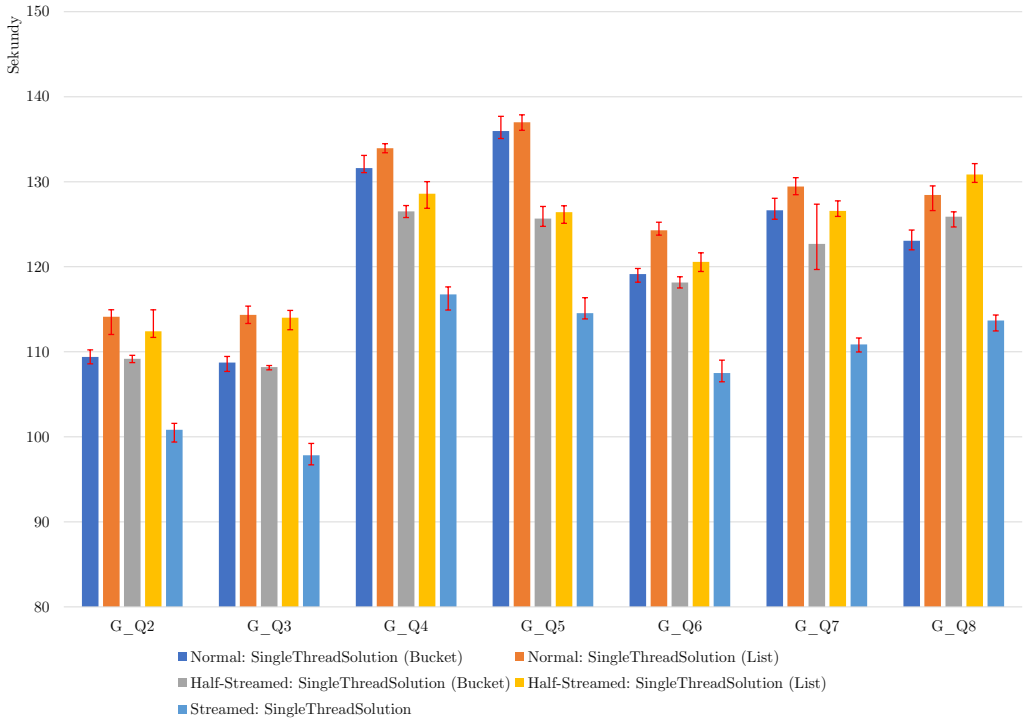
Obrázek 4.9: Doba vykonání dotazů *Group by* bez agr. funkcí pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu.



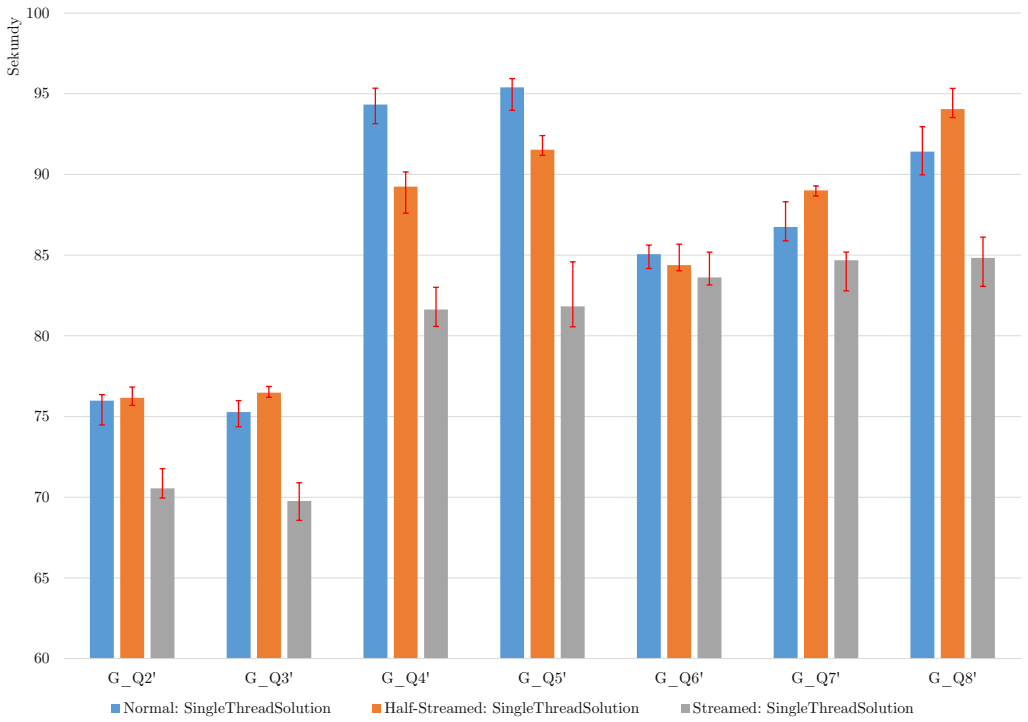
Obrázek 4.10: Doba vykonání dotazů *Group by* pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu.



Obrázek 4.11: Doba vykonání dotazů *Group by* bez agr. funkcí pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu.



Obrázek 4.12: Doba vykonání dotazů *Group by* pro graf As-Skitter (sekce 4.1). Běh v jednom vláknu.



Obrázek 4.13: Doba vykonání dotazů *Group by* bez agr. funkcí pro graf As-Skitter (sekce 4.1). Běh v jednom vláknu.

Obecné shrnutí paralelních řešení

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 sléváním do paralelní mapy. **LocalGroupByLocalTwoWayMerge** řešení seskupuje lokálně a následně slévá výsledky vláken po dvojicích. Toto slévání si můžeme představit jako binární strom. Listy jsou výsledky vláken a vnitřní vrcholy jsou akce slévány. Výsledky paralelního seskupování jsou zobrazeny na obrázcích před shrnutím výsledků *Single group Group by* v sekci 4.4.3.

Výsledky paralelního zpracování

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. První vrstva synchronizace nastává u přístupu 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 4.21, 4.23 a 4.25 (paralelní **Streamed** řešení) v sekci 4.4.3 je pak vidět značnou režii za synchronizaci za thread-safe agregační funkce při přístupu osmi vláken.

Test režie paralelní mapy (jedno vlákno)

Pro představu pouhé režie paralelní mapy jsme otestovali režii zvlášť čtení a vložení `ConcurrentDictionary` vůči `Dictionary` pro jedno vlákno.

Následuje příklad kódu použitého při testu (označíme jej *I/Rcode*):

```
Random ran = new Random(100100);
Dictionary<int, int> map = new Dictionary<int, int>();
ConcurrentDictionary<int, int> parMap =
    new ConcurrentDictionary<int, int>();
...
// Insert test. Slovník je prázdný.
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. Slovník obsahuje hodnoty od 0 do 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);
}
```

Výsledek testu režie paralelní mapy (jedno vlákno)

Test	**Dict**	**ConDict**	**ConDict/Dict**
Insert 10^6	165	667	4,04
Insert 10^7	2476	11 272	4,55
Read 10^8	19 002	21 516	1,13

Tabulka 4.11: Výsledky testování map v milisekundách.

Měření dle kódu *I/Rcode* výše. Zkratka **Dict** označuje `Dictionary` a zkratka **ConDict** označuje `ConcurrentDictionary`. Běh v jednom vlákně. Generování prvků pomocí třídy `Random` s inicializační hodnotou 100100. Měřeno pomocí třídy `Stopwatch`. Výsledek byl 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 1 000 000. Poměr je roven podílu času paralelní mapy a mapy.

Tabulka naměřených hodnot testování (4.11) 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.

Test režie paralelní mapy (osm vláken)

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*. Výsledek porovnáme s výsledky režie normální mapy z tabulky 4.11.

Následuje příklad kódu použitého při testu (označíme jej *ParIcode*):

```
ConcurrentDictionary<int, int> parMap =
    new ConcurrentDictionary<int, int>();
// Insert test. Slovníky jsou prázdné. (příklad testu 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); }
```

Výsledek testu režie paralelní mapy (osm vláken)

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

Tabulka 4.12: Výsledky testování map v milisekundách.

Měřeno dle kódu *ParIcode* výše. Zkratka **Dict** označuje `Dictionary` a zkratka **ConDict** označuje `ConcurrentDictionary`. `Dictionary` vykoná práci v jednom vlákně. `ConcurrentDictionary` běží paralelně v osmi vláknech. Generování prvků pomocí třídy `Random` s inicializační hodnotou 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 4.12 vidíme, že samotná paralelizace je pro náš počet vkládání prvků pomalejší. Tedy obecné zpomalení je zřetelné u řešení používající paralelní mapu.

Paralelní Streamed řešení vs paralelní Normal: Global řešení

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 vlastností skrze tabulku výsledků.

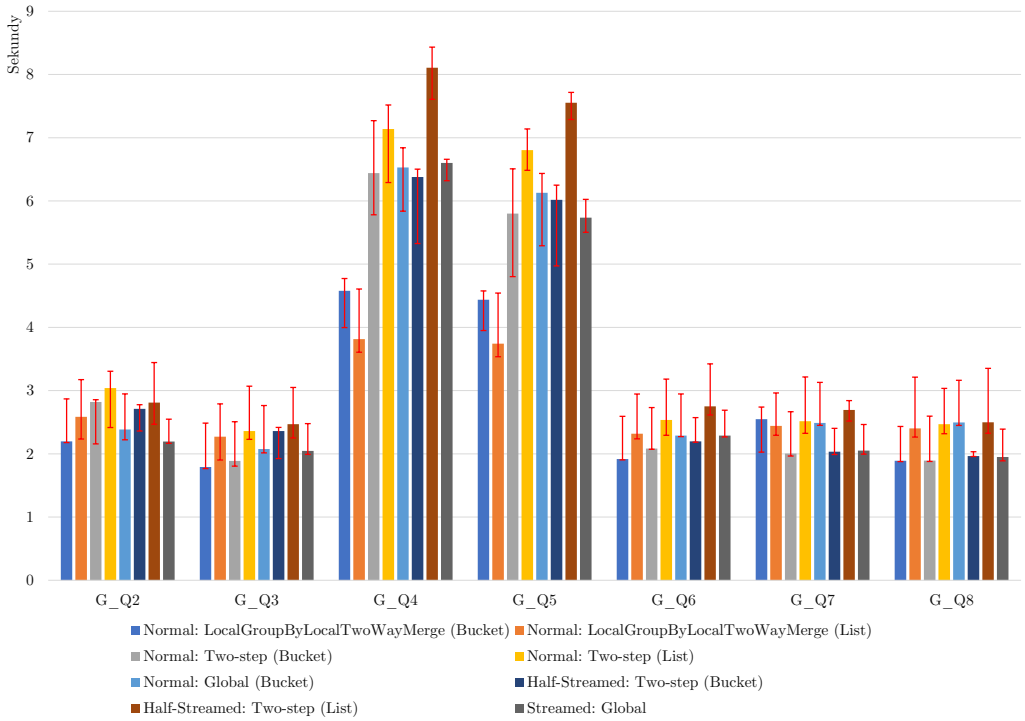
Aplikace poznatků jednovláknového zpracování při paralelním zpracování

Můžeme zde aplikovat poznatky z výsledků jednovláknových řešení pro **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 jednovláknových 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 slévání překopírovává větší množství dat, proto u něj daný jev už nenastává.

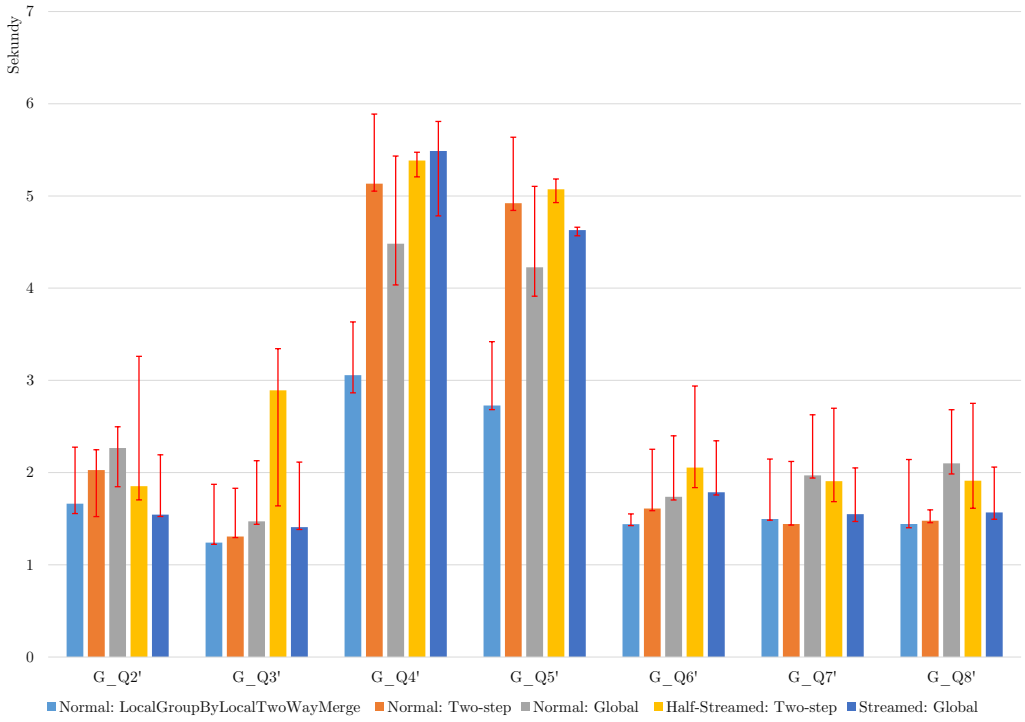
Nejrychlejší paralelní řešení

Všem řešením dominuje**Normal: LocalGroupByLocalTwoWayMerge**, které provádí vše lokálně a ujišťuje nás v předpokladu, že hlavní důvod zpomalení je synchronizace. Například dané řešení vůči **Normal: Two-Step**. Je zde vidět režie za použití paralelní mapy vůči lokálnímu slévání 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.

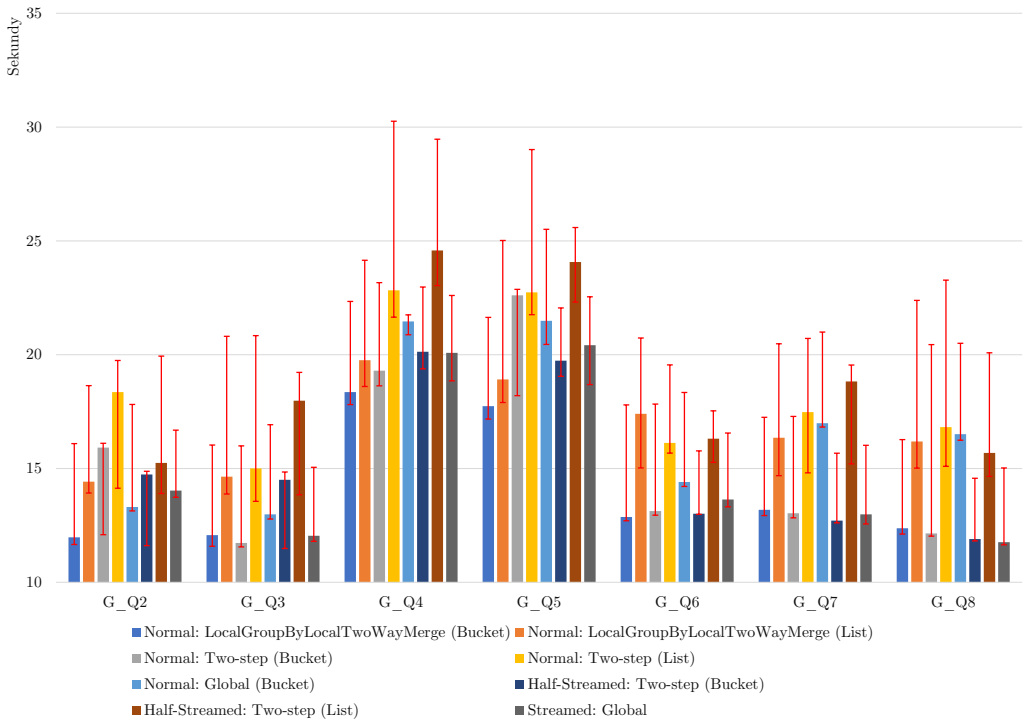
Z výsledků usuzujeme, že vylepšená řešení pro náš případ paralelizace neposkytují z hlediska rychlosti vykonání znatelné výhody oproti stávajícím řešením.



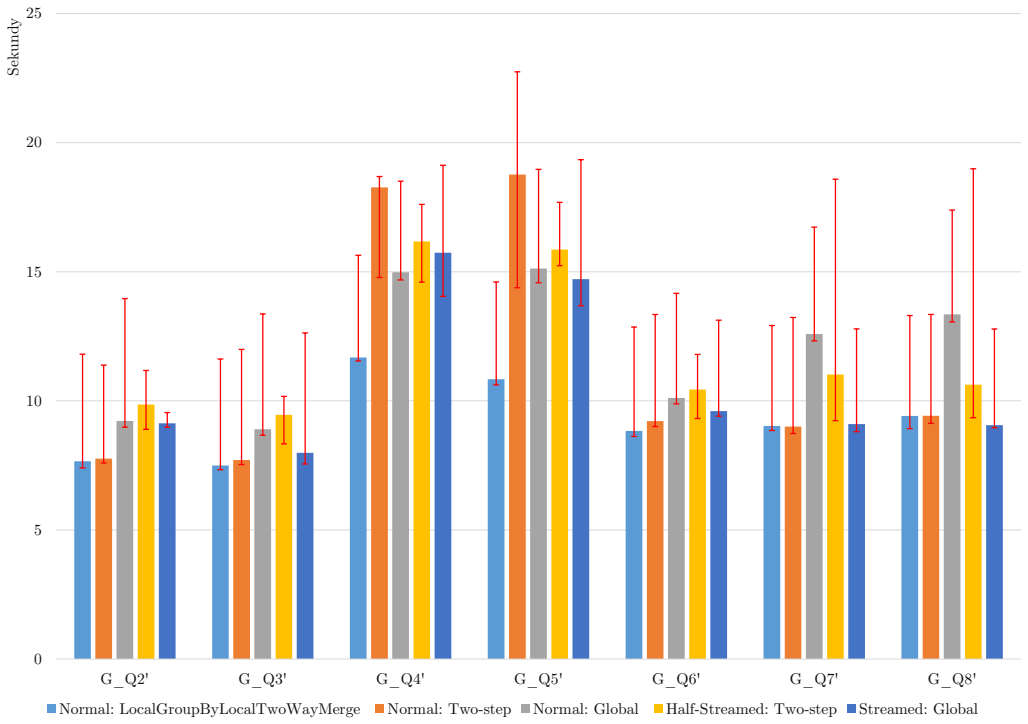
Obrázek 4.14: Doba vykonání dotazů *Group by* pro graf Amazon0601 (sekce 4.1). Běh osmi vláken.



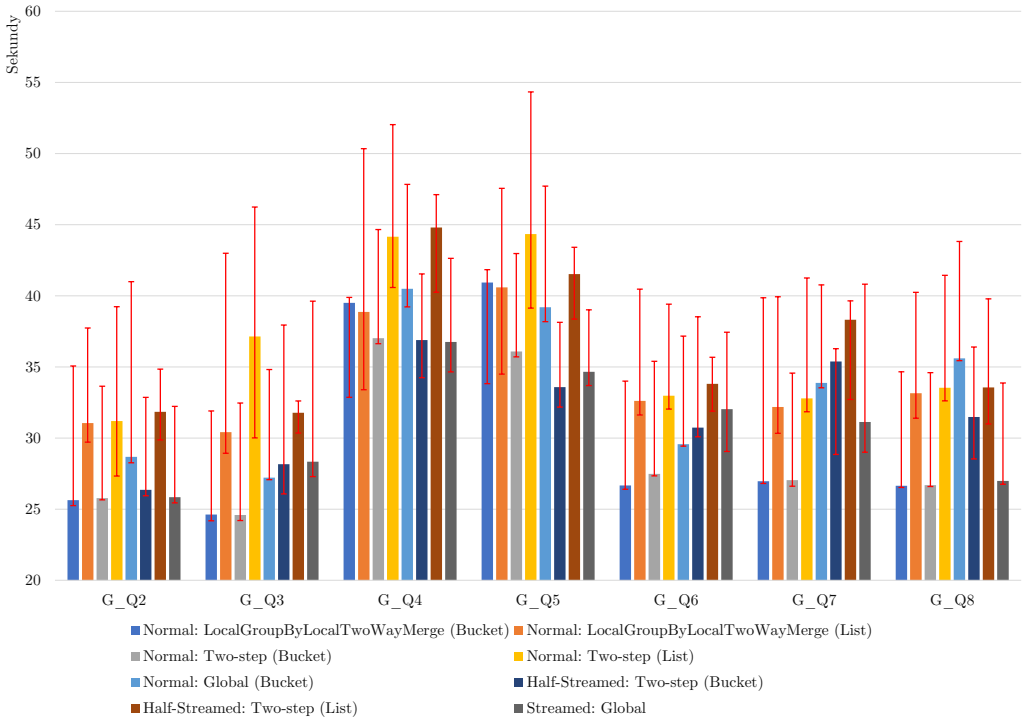
Obrázek 4.15: Doba vykonání dotazů *Group by* bez agr. funkcí pro graf Amazon0601 (sekce 4.1). Běh osmi vláken.



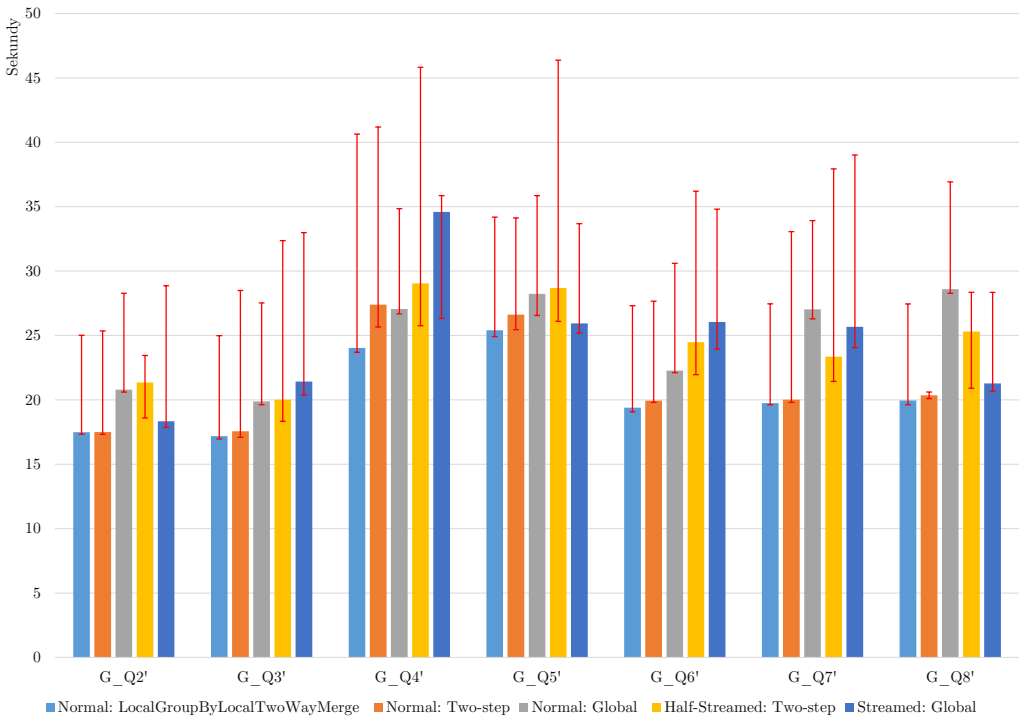
Obrázek 4.16: Doba vykonání dotazů *Group by* pro graf WebBerkStan (sekce 4.1). Běh osmi vláken.



Obrázek 4.17: Doba vykonání dotazů *Group by* bez agr. funkcí pro graf WebBerkStan (sekce 4.1). Běh osmi vláken.



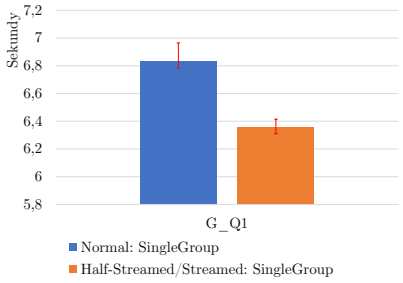
Obrázek 4.18: Doba vykonání dotazů *Group by* pro graf As-Skitter (sekce 4.1). Běh osmi vláken.



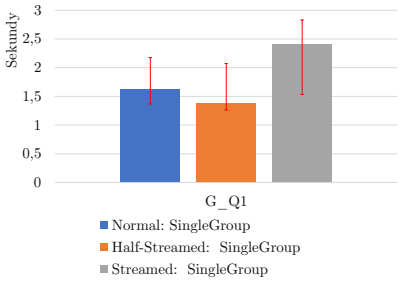
Obrázek 4.19: Doba vykonání dotazů *Group by* bez agr. funkcí pro graf As-Skitter (sekce 4.1). Běh osmi vláken.

Obecné shrnutí Single group Group by řešení

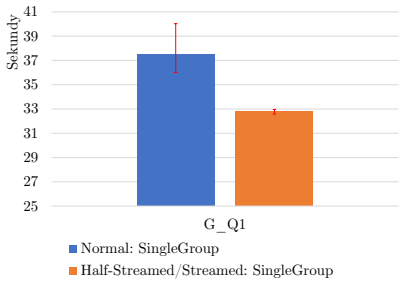
Analýzu výsledků dotazu G_Q1 uvádíme samostatně, protože testuje pouze agregační funkce a nikoliv seskupování. Daný mód *Group by* předpokládá, že všechny výsledky patří do stejné skupiny a tedy nedochází k seskupování ale pouze k výpočtu agregačních funkcí. Jednovláknové **Normal** řešeni iteruje skrze všechny výsledky po dokončení vyhledávání a vypočte agregační funkce. Jednovláknové **Half-Streamed** a **Streamed** řešení jsou totožná. V průběhu prohledávání aktualizují hodnotu agregační funkce a výsledek zahodí. Paralelní **Normal** řešení všem vláknům přidělí stejný počet výsledků prohledávání a každé lokálně spočte hodnoty agregačních funkcí. Po ukončení práce všech vláken jedno vybrané vlákno sloučí výsledky všech ostatních vláken. Řešení **Half-Streamed** provádí stejnou práci jako v jednovláknovém zpracování. Po dokončení prohledávání dojde ke sloučení všech lokálních výsledků. **Streamed** řešení pracuje se sdíleným úložištěm výsledků a používá thread-safe funkce k aktualizaci výsledků. Obě řešení výsledky neukládají.



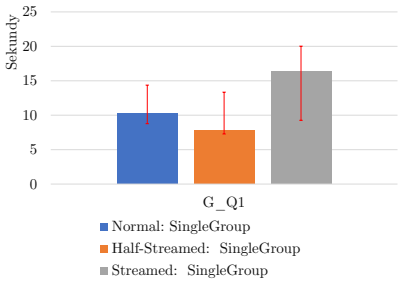
Obrázek 4.20: Doba vykonání dotazu G_Q1 pro graf Amazon0601 (sekce 4.1). Běh v jednom vláknu.



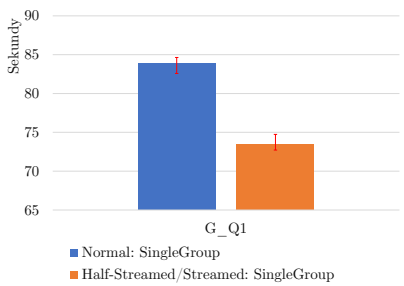
Obrázek 4.21: Doba vykonání dotazu G_Q1 pro graf Amazon0601 (sekce 4.1). Běh osmi vláken.



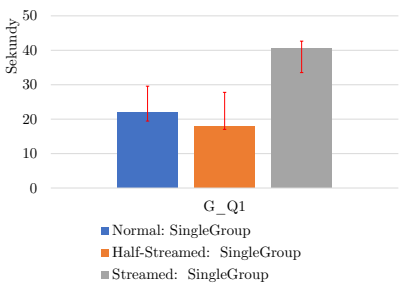
Obrázek 4.22: Doba vykonání dotazu G_Q1 pro graf WebBerkStan (sekce 4.1). Běh v jednom vláknu.



Obrázek 4.23: Doba vykonání dotazu G_Q1 pro graf WebBerkStan (sekce 4.1). Běh osmi vláken.



Obrázek 4.24: Doba vykonání dotazu G_Q1 pro graf As-Skitter (sekce 4.1). Běh v jednom vláknu.



Obrázek 4.25: Doba vykonání dotazu G_Q1 pro graf As-Skitter (sekce 4.1). Běh osmi vláken.

Výsledky Single group Group by zpracování

Na obrázcích 4.20 až 4.25 lze vidět značnou konzistenci mezi výsledky testování při nárůstu 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. Problém 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.

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

Závěr

Vytvořili jsme dotazovací engine dle zadaných požadavků. Engine se skládá ze dvou částí. První část je statická grafová databáze využívající Labeled-property model grafových dat. Druhá část obsahuje struktury a algoritmy pro vykonání uživatelských dotazů nad grafovou databází. Dotazy jsou zadány ve stanovené podmnožině jazyka PGQL. Vykonávání je možno provádět jednovláknově i paralelně. Všechna grafová data a data v průběhu zpracování dotazu jsou obsažena v hlavní paměti. Druhou část jsme rozdělili na dva bloky:

1. V prvním bloku jsme navrhli řešení vykonávající *Group/Order by* po dokončení prohledávání grafu v části *Match*. Výsledky prohledávání byly ukládány do tabulky. Teprve po získání všech výsledků byly na výsledcích aplikovány algoritmy *Group/Order by*.

    - V části *Order by* jsme použili standartní algoritmus Merge sort pro jednovláknové i paralelní zpracování.
    - V části *Group by* jsme výsledky seskupovali pomocí hašovací tabulky. Pro paralelní zpracování jsme zvolili tři řešení dle úrovně synchronizace (sekce 2.7.6). V módu *Single group Group by* (dotaz obsahuje agregační funkce, ale neobsahuje část *Group by*) jsme využili lokálního zpracování, které je zakončeno sléváním.

2. V druhém bloku jsme upravili první blok a navrhli nová řešení zpracování dle zadání práce. Propojili jsme prohledávání grafu se zpracováním *Group/Order by* tak, aby engine seskupoval/setřiďoval výsledky prohledávání v moment jejich nalezení. Pro paralelní zpracování jsme navrhli dva módy: **Streamed** a **Half-Streamed**. **Streamed** zpracovával výsledky globálně. **Half-Streamed** zpracovával výsledky lokálně, následně docházelo ke slévání.

    - V části *Order by* jsme pro jednovláknové zpracování použili standartní $(a, 2a)$-strom a upravenou verzi $(a, 2a)$-stromu, která omezila počet zatříďovaných výsledků. **Half-Streamed** řešení zpracovávalo lokálně výsledky pomocí navržených stromů a následně došlo k dvoucestnému paralelnímu slévání. **Streamed** řešení vytvořilo přihrádky přistupné skrze zámek a každé přihrádce přiřadilo specifický rozsah hodnot.
    - V části *Group by* jsme vycházeli z řešení prvního bloku. Jednovláknové a paralelní zpracování **Half-Streamed** používalo tabulku z prvního bloku. Obecně jsme u něj omezeli ukládané výsledky pouze na reprezentanty skupin. Jednovláknové a paralelní zpracování **Streamed** nepoužívalo žádnou tabulku. Zpracování *Single group Group by* neukládalo žádné výsledky, kromě výsledků agregačních funkcí.

Dva dané bloky jsme porovnali v rychlosti vykonávání dotazů pomocí experimentu. Experiment obsahoval množství dotazů, které byly vykonány nad třemi reálnými grafy s uměle vygenerovanými vlastnostmi. Otestovali jsme jednovláknové zpracování a paralelní zpracování využívající osm vláken. Při experimentu

jsme měřili práci pouze *Match* části s *Group/Order by*, tj. výsledky neobsahují práci *Select* části. Následuje shrnutí výsledků:

- *Order by:* z výsledků *Order by* vyplynulo, že řešení druhého bloku využívající $(a, 2a)$-stromy jsou obecně pomalejší než řešení v prvním bloku využívající Merge sort. Nicméně, námi navržená paralelní řešení druhého bloku předčila paralelní Merge sort prvního bloku při třídění pomocí vlastností.

- *Single group Group by:* zde byla řešení druhého bloku rychlejší. Problematické zde bylo paralelní **Streamed** řešení, ve kterém docházelo ke zpomalení, kvůli značné režii za synchronizaci.

- *Group by:* u řešení *Group by* druhého bloku nedocházelo ke znatelnému zrychlení. Jedinou výjimkou bylo **Streamed** řešení druhého bloku v jednovláknovém zpracování, které bylo nejrychlejší.

Z výsledků experimentu můžeme konstatovat, že naše předpoklady o zpracovávání *Group by* a *Order by* v průběhu prohledávání grafu nabízí v určitých případech zrychlení vykonávání dotazů.

Budoucí výzkum a rozšíření

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, ve kterých část *Order by* obsahuje pouze výrazy seskupování z části *Group by*. Zde lze například aplikovat podobný přístup řešení **ABTreeValueAccumulator** části *Order by*, ve kterém se výsledky se stejnými klíči třídění seskupovaly do pole. Nyní místo seskupování do pole bude stromová struktura obsahovat pouze reprezentanty skupin a jejich úložiště hodnot agregačních funkcí. V druhé části je nutné vyřešit dotazy, které v části *Order by* obsahují výrazy 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 vlastnostmi. V našem experimentu jsme sice volili reálne grafy, ale jejich vlastnosti 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 algoritmus Merge sort, který není možný aplikovat při třídění v průběhu vyhledávání. Rozdělení práce vylepšených řešení závísí na počtu vyhledaných výsledků v každém vlákně. Mohou nastávat 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 vláknům. 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. *Order by*:

   (a) U paralelního řešení jsme viděli značné zrychlení při třídění pomocí vlastností. Bylo by vhodné prozkoumat možnosti vytvoření globálních statistik pro každou vlastnost a podrobněji zjistit možnosti rozdělení rozsahů použitým přihrádkám.

   (b) V našem řešení jsme rozdělení přihrádek pro řetězce zpracovali pouze s předpokladem, že se jedná o ASCII znaky. V budoucí práci je možné zkoumat rozdělování i pro složitější znakové sady.

   (c) Obecně *Order by* řešení využívaly implementaci $(a, 2a)$-stromu. Daná implementace má značnou režii za metodu `Insert`, při které dochází k časté alokaci nových vrcholů. V budoucích rozšířeních je možné vyzkoušet předalokovat určitou množinu vrcholů stromu, které se následně využijí v dané metodě.

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

6. Dotazy v našem dotazovacím enginu jsou zadávány pomocí podmnožiny jazyka PGQL. Při implementaci jsme se snažili oddělit načítání dotazu od zpracování dotazu. K tokenizaci jsme použili třídu `Tokenizer`, pro vytvoření syntaktických stromů jsme použili třídu `Parser` a k procházení vzniklých stromů jsme implementovali rozhraní `IVisitor`. Procházením stromů vznikají struktury, které se následně využijí při zpracování dotazu. Obecně tokenizace a vytvoření syntaktických stromů je odděleno od zpracování dotazu. Při použití jiného jazyka stačí implementovat dané dvě části separátně. Problém nastave v rozhraní `IVisitor` a vzniku struktur dotazu, protože konstruktory tříd `Query` a `QueryObject` očekávají stávající formát. Nicméně, dané konstruktory a rozhraní `IVisitor` se dají jednoduše upravit k použití nového formátu. Přínosem je pak možnost využít i jiné dotazovací jazyky za cenu malých úprav. Dalším přínosem je také možnost využít třídu `Tokenizer` a `Parser` jako separátní knihovnu k načítání PGQL dotazu.

Seznam použité literatury

[1] Yves Raimond and Guus Schreiber. RDF 1.1 primer. W3C note, W3C, June 2014. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[2] Mark Needham and E. Amy Hodler. *Graph Algorithms*. O'Reilly Media, Inc., May 2019.

[3] Andy Seaborne and Steven Harris. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[4] *Dokumentace jazyka PGQL 1.2*. `https://pgql-lang.org/spec/1.2/` [Dostupnost ověřena k datu 15.4.2021].

[5] *Dokumentace jazyka openCypher*. `https://www.opencypher.org/` [Dostupnost ověřena k datu 15.4.2021].

[6] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. O'Reilly Media, Inc., 1st edition, 2018.

[7] János Dániel Bali. Streaming graph analytics framework design. Master's thesis, KTH, School of Information and Communication Technology (ICT), 2015. `http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A830662&dswid=-2589`, OAI=oai:DiVA.org:kth-170425, URN=urn:nbn:se:kth:diva-170425, diva=diva2:830662.

[8] *Apache Flink proudový systém*. `https://ci.apache.org/projects/flink/flink-docs-release-1.13` [Dostupnost ověřena k datu 15.4.2021].

[9] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

[10] *Apache Flink Gelly proudový systém pro grafová data*. urlhttps://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/libs/gelly/overview/ [Dostupnost ověřena k datu 15.4.2021].

[11] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1481–1496, New York, NY, USA, 2016. Association for Computing Machinery.

[12] *Neo4j grafová databáze*. `https://neo4j.com/developer/graph-database/` [Dostupnost ověřena k datu 15.4.2021].

[13] *Neptune grafová databáze*. `https://aws.amazon.com/neptune/` [Dostupnost ověřena k datu 15.4.2021].

[14] *Dgraph grafová databáze*. `https://github.com/dgraph-io/dgraph` [Dostupnost ověřena k datu 15.4.2021].

[15] *Dokumentace jazyka JSON.* `https://www.json.org/` [Dostupnost ověřena k datu 15.4.2021].

[16] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994.

[18] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[19] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. Pgx.d/async: A scalable distributed graph pattern matching engine. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences and Systems*, GRADES'17, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Martin Mareš and Tomáš Valla. *Průvodce labyrintem algoritmů.* CZ.NIC, z.s.p.o., 2017.

[21] Victor J. Duvanenko. Hpcsharp. `https://github.com/DragonSpit/HPCsharp`, 2018.

[22] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[23] *Dokumentace jazyka C# pro .NET Framework 4.8.* `https://docs.microsoft.com/en-us/dotnet/csharp/` [Dostupnost ověřena k datu 15.4.2021].

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

A. Přílohy

A.1 Zdrojové kódy

Přílohou této bakalářské práce jsou zdrojové kódy dotazovacího enginu, benchmarku a použité knihovny HPCsharp. Vše zmíněné je přiloženo v rámci jednoho projektu Visual Studia 2019, 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 (kapitola 4) jsou vloženy do odpovídajících složek dle názvu grafu.

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 Benchmark stromy vůči polím

Součástí této přílohy jsou zdrojové kódy benchmarku použitého k testování vybraných indexačních struktur. Benchmark je použit v sekci 2.9.2.