Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Ján Majdan

Implementace vybraných databázových operací v paralelním prostředí

Katedra softwarového inženýrství

Vedoucí diplomové práce:	RNDr. David Bednárek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové systémy

Praha 2013

Na tomto mieste by som sa rád poďakoval vedúcemu mojej diplomevej práce RNDr. Davidovi Bednárkovi, Ph.D. za cenné rady a čas, ktorý mi venoval. Ďalej by som sa chcel poďakovať Vlaste Poliačkovej za osobnú podporu a pomoc pri revízii slovenčiny v texte.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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 v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 11.4.2013 Podpis autora

Název práce: Implementace vybraných databázových operací v paralelním prostředí

Autor: Bc. Ján Majdan

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek, Ph.D.

Abstrakt: Předložená diplomová práce se zabývá možnostmi implementace databázových operací v paralelním prostředí **Bobox.** Popisuje potřebné teoretické znalosti z oblasti databází, a též algoritmy potřebné pro implementaci databázových operací v hlavní paměti počítače. V textu jsou popsané možnosti paralelní realizace databázových operací pomocí hašovacích tabulek. Představené jsou principy založené na sdílení hašovací tabulky, a taky předzpracování dat (lokální hašovací tabulka). Součástí diplomové práce je i paralelní implementace vybraných operací, která je založená na postupech představených v textu práce. Realizace zahrnuje taky měření výkonnosti implementace při různých stupních paralelizace. Naměřené výsledky jsou přehledně prezentované a analyzované z pohledu škálovatelnosti paralelizace v prostředí **Bobox.**

Klíčová slova: databázové operace, Bobox, paralelizace, hašování, předspracování dat

Title: Implementation of selected database operations in parallel environment

Author: Bc. Ján Majdan

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D.

Abstract: This thesis describes several design possibilities of database operations in a parallel environment called Bobox. First, the thesis covers theory of databases as well as algorithms needed to implement database operations in main memory. Next part discusses the usage of hash tables to implement parallel database operations. The key principles discussed contain hashing approaches based on a shared hash table, as well as principles of data preprocessing (local hash Table). The thesis then describes practical usage of selected database operations, which were implemented based on the principles described in the first theory sections. The implementation also involves performance measurement at different levels of parallelism. Finally, acquired results are analyzed and discussed in terms of scalability of parallelism and performance in the Bobox environment.

Keywords: database operations, Bobox, parallelism, hashing, data preprocessing

Názov práce: Implementácia vybraných databázových operácií v paralelnom prostredí

Autor: Bc. Ján Majdan

Katedra: Katedra softwarového inženýrství

Vedúci diplomovej práce: RNDr. David Bednárek, Ph.D.

Abstrakt: Predložená diplomová práca sa zaoberá možnosťami implementácie databázových operácií v paralelnom prostredí **Bobox.** Popisuje potrebné teoretické znalosti z oblasti databáz, a tiež algoritmy potrebné pre implementáciu databázových operácií v hlavnej pamäti počítača.V texte sú popísané možnosti paralelnej realizácie databázových operácií pomocou hašovacích tabuliek. Predstavené sú princípy založené na zdieľaní hašovacej tabuľky, a tiež predspracovania dát (lokálna hašovacia tabuľka). Súčasťou diplomovej práce je aj samotná paralelná implementácia vybraných operácií, ktorá je založená na postupoch predstavených v texte práce. Realizácia zahŕňa tiež aj merania výkonnosti implementácie pri rôznych stupňoch paralelizácie. Namerané výsledky sú prehľadne prezentované a analyzované z pohľadu škálovateľnosti paralelizácie v prostredí **Bobox.**

Kľúčové slová: databázové operácie, Bobox, paralelizácia, hašovanie, predspracovanie dát

Obsah

Kapitola 1

Úvod

Myšlienka paralelizácie počítačových výpočtov nie je v informatike žiadnou novinkou. Prvé snahy použiť princípy paralelizácie sa objavili už na prelome päťdesiatych a šesťdesiatych rokov minulého storočia. Cieľom paralelizácie je vykonať požadovanú úlohu rýchlejšie, spracovať väčšie množstvo dát, vypočítať presnejší výsledok a podobne. Je zrejmé, že cieľ paralelizácie zavisí na riešenom probléme, avšak je možné povedať, že vždy sa jedná o snahu zvýšiť výkon aktuálneho riešenia.

Pod paralelizáciou výpočtu v počítači je možné si predstaviť viacero rôznych postupov ako napríklad paralelizáciu na úrovni inštrukcií procesoru, súbežné spúšťanie programov na úrovni procesov operačného systému (kde programy môžu byť stále sekvenčné), alebo tiež programy skladajúce sa z viacerých paralelne bežiacich výpočtov (vlákien). Je zrejmé, že sa jedná o rôzne úrovne abstrakcie práce počítača. V ďalšom texte bude pod paralelizáciou vždy myslený posledný zmienený prípad, tj. paralelizácia na úrovni programov.

Programovanie paralelných programov je v súčasnosti komplikovanejšie ako vytváranie klasických sekvenčných programov. Programátor musí explicitne zabezpečiť samotnú paralelizáciu, tj. vytvoriť a spustiť vhodný počet vlákien s paralelným výpočtom [1], ošetriť ich prípadnú synchronizáciu a zabezpečiť synchronizáciu prístupu k zdieľaným dátam. Tieto činnosti nie sú súčasťou úlohy, ktorú programátor rieši, ale jedná sa len o pomocné činnosti, ktoré by v ideálnom prípade mal zabezpečiť prekladač. Požadovanú automatickú paralelizáciu majú zabudovanú prekladače/interpretre jazykov[2], ktoré boli s týmto zámerom navrhnuté, avšak ich použitie v praxi je málo časté. Dôvodom je problematické ladenie chýb, a tiež špecifická doména použitia týchto jazykov.

V praxi sa pre vývoj paralelných aplikácií skôr osvedčilo používanie nadstavieb pre komerčne úspešné jazyky. Určite za zmienku stoja technológie ako napríklad **OpenMP**, **MPI** a **TBB**.

Jednou z technológii, ktoré poskytujú podporu pre paralelizmus, je aj framework **Bobox**, ktorý je vyvíjaný na Katedre Softwarového Inžinierstva Matema-

[1] Obecne nie je možné určiť vhodný počet vlákien počas prekladu, resp. písania kódu, nakoľko je toto rozhodnutie závislé od konfigurácie HW (počet procesorov), na ktorom program pobeží.

[2] Jedná sa napríklad o jazyky Axum, High Performance Fortran (HPF), Id, LabVIEW, NESL, SISAL, Parallel Haskell, SystemC.

ticko-fyzikálnej fakulty Univerzity Karlovej. Jedná sa o nástroj pre programovací jazyk C++. Použitím **Boboxu** je programátor de facto oslobodený od vyžšie spomenutých „nízko úrovňových“ problémov paralelizácie. Práca **Boboxu** je riadená modelom výpočtu, ktorý musí programátor sám navrhnúť. Na základe predloženého modelu je naplánovaná a realizovaná vhodná paralelizácia.

Bobox vznikol ako prostredie pre paralelné spracovanie dát. Možnosti spracovania dát boli overené experimentami s *XQuery* [1], a tiež pri implementácii jazyka *SPARQL* [2]. Klasické relačné dotazy však zatiaľ v postredí **Bobox** implementované neboli.

Táto diplomová práca sa zaoberá možnosťami implementácie paralelných relačných[3] databázových operácií pomocou frameworku **Bobox**, čím môže poskytnúť základy pre realizáciu relačných dotazov. V neposlednej rade práca poskytne informácie o škálovateľnosti paralelizácie databázových operácií v prostredí **Bobox**.

V ďalšom texte budú rozobrané nasledujúce témy:

Paralelné prostredie Bobox

Keďže **Bobox** nie je všeobecne známy a používaný nástroj, tak druhá kapitola je koncipovaná ako úvod do fungovania použitého frameworku **Bobox.** Po prečítaní by mal byť čitateľ schopný porozumieť celej problematike týkajúcej sa **Boboxu**, ktorá je použitá v tejto práci.

Databázové operácie

Ďalšia časť práce sa zaoberá teóriou z oblasti databázových operácií. Čitateľ tu nájde vysvetlenie viacerých základných pojmov z prostredia databáz, ako napríklad plán dotazu alebo definície jednotlivých databázových operácií. Ďalej sú tu popísané sekvenčné algoritmy databázových operácií.

Databázové operácie v prostredí Bobox

Štvrtá kapitola je zameraná na vlastný popis návrhu modelov databázových operácií. Súčasne je tu popísaná aj implementácia boxov, ktoré realizujú potrebnú logiku.

Analýza implementovaných databázových operácií

Súčasťou práce je aj overenie vlastností implementovaných databázových operácií, ktoré spolu s popisom vykonaných experimentálnych meraní tvorí piatu kapitolu. V kapitole je zhrnutý popis a analýza nameraných dát, čo vedie k výsledným vyskúmaným záverom.

[3] V ďalšom texte budú vždy uvažované relačné databázové operácie, a preto nebude tento fakt explicitne uvádzaný.

Kapitola 2

Paralelné prostredie Bobox

Účelom tejto kapitoly je v dostatočnej miere oboznámiť čitateľa so základnými princípmi práce s frameworkom **Bobox** tak, aby bol schopný pochopiť ďalej popisovanú problematiku. Cieľom práce však nie je ponúknuť vyčerpávajúci popis práce s frameworkom **Bobox**, ktorý by poskytol čitateľovi po prečítaní dostatočné znalosti pre samostatnú prácu v tomto prostredí. Ďalšie informácie ohľadom inštalácie, popisu použitia či implementačných detailov je možné nájsť v dokumentácii frameworku **Bobox** [3]. Okrem iného je možné nájsť ďalšie informácie o použitých postupoch a vnútornej logike frameworku v článkoch [4] a [5].

Kapitola je zameraná na výklad priebehu návrhu aplikácie bežiacej pod frameworkom **Bobox**, základnej terminológie a priebehu výpočtu.

2.1 Bobox

Framework **Bobox** je paralelizačný nástroj, ktorého úlohou je v najväčšej možnej miere uľahčiť prácu programátorovi pri paralelizácii programu. Pričom je samozrejme požadovaný čo najlepší výkon týmto spôsobom vytváraných programov. **Bobox** je postavený na základe štyroch rozhodnutí, ktoré značne determinujú jeho vlastnosti:

1. Synchronizácia je pred programátorom skrytá. O každú synchronizáciu sa stará framework a kód užívateľa by mal byť striktne sekvenčný.

2. Technické detaily[1] riešenia paralelizácie sú tiež skryté pred programátorom.

3. Komunikácia v rámci výpočtu prebieha len pomocou vysokoúrovňových správ. To znamená, že programátor sa musí naučiť len minimum nových nutných znalostí.

4. Základné paradigma pre výpočet **Boboxu** je paralelizmus úloh[2] a nelineárna pipeline.

[1] NUMA, cache, architektúra CPU ...
[2] task parallelism

Zmienený paralelizmus úloh je jeden z možných prístupov paralelizácie. Princíp paralelizácie pomocou úloh spočíva v rozdelení riešeného problému na rôzne podúlohy, ktoré sa môžu vykonávať samostatne-paralelne. V kontraste s týmto postupom je dátová paralelizácia[3], ktorá spočíva v rozdelení dát na časti, nad ktorými sa paralelne spúšťa tá istá úloha.

Pod pojmom pipeline sa rozumie zreťazené spracovanie úloh. Úlohy v pipeline sú zreťazené tak, že výstup jednej je vstupom ďalšej úlohy v reťazci, čím je dané poradie vykonávania. Nelineárna pipeline umožňuje vytvárať spätné spojenia v reťazci spracovania, prípadne v zreťazení poslať dáta do viacerých výstupov. Tento prístup umožňuje oveľa väčšie možnosti ako iba lineárne zapojenie, avšak nesie so sebou aj nebezpečenstvo vytvorenia cyklu, ktorý môže zapríčiniť nedeterministicé chovanie alebo deadlock. Tento problém je nutné zohľadniť pri tvorbe návrhu výpočtu pre **Bobox**.

2.2 Výpočet v prostredí Bobox

Ako už bolo spomenuté, v **Boboxe** je ako základné paralelizačné paradigma zvolený paralelizmus úloh. Takže samotný program je v **Boboxe** zložený z niekoľkých úloh, ktoré môžu bežať paralelne. V terminológii **Boboxu** sa označujú tieto úlohy ako boxy. Box nesie jednovláknový, t.j. sekvenčný kód naprogramovaný užívateľom, ktorý na vstupe očakáva nejaké dáta a na výstup posiela výsledky svojej práce. Celý výpočet je vykonávaný len pomocou zreťazenia boxov, ktorých spúšťanie, resp. plánovanie má na starosti **Bobox**. Zreťazenie boxov presne určuje poradie vykonávania kódu jednotlivých boxov, a tiež určuje dátové vstupy a výstupy boxov. To znamená, že výstupy jedného boxu môžu byť vstupmi iného boxu, pokiaľ medzi týmito boxami existuje priame spojenie v zreťazení určujúcom výpočet.

Celý výpočet si je možné predstaviť ako orientovaný graf, v ktorom sú vrcholmi grafu boxy a hranami sú spojenia určujúce poradie vykonávania. Ako už bolo spomenuté, hrany/spojenia môžu niesť aj dáta.

Vizualizácia zmieneného grafu je názorná pomôcka pri prezentácii modelu výpočtu v **Boboxe**. V skutočnosti je **Bobox** riadený pomocou textovej podoby modelu výpočtu, ktorá popisuje všetky zmienené aspekty grafu. Textová podoba modelu výpočtu je ešte pred spustením predaná behovému prostrediu **Bobox**, ktoré na základe získaného modelu riadi celý výpočet. Vytvorenie správneho modelu výpočtu je ponechané na programátora. **Bobox** je zodpovedný len za dodržiavanie pravidiel uvedených v modele.

Programátor je teda zodpovedný za naprogramovanie sekvenčného kódu boxov, a tiež aj za vytvorenie modelu výpočtu. **Bobox** na základne poradia boxov, resp. závislostí boxov v modele, vytvára a spúšťa konkrétne boxy, ktorým prípadne predáva dáta na spracovanie.

[3]data parallelism

2.3 Model výpočtu

Textový popis modelu výpočtu je zložený z dvoch častí. V prvej časti sú zadefinované boxy, ktoré sa počas výpočtu budú používať. **Bobox** na základe tejto časti dokáže vytvoriť mapovanie s užívateľskými implementáciami boxov, ktoré bude pri výpočte používať. V druhej časti modelu sú určené spojenia medzi týmito boxami, podľa ktorých sa boxy radia do pipeline pre spracovanie. Takto určené spojenie môže mať tiež definované typy dát, ktoré je možné posielať medzi spojenými boxami.

Textový zápis modelu má definovanú gramatiku, avšak pre potreby pochopenia tejto práce nie je nutná znalosť zápisu, nakoľko všetky modely popisované v tomto texte budú vizualizované pomocou názorných diagramov.

V ďalšom texte budú prezentované vlastnosti modelov a ich vizualizácie na príklade modelu, ktorý predstavuje program vypisujúci na štandardný výstup text „*Hello World!*“. Zvolený príklad nie je úplne vhodný pre demonštráciu paralelizácie, nakoľko prístup k štandardnému výstupu nie je synchronizovaný, a tak reálne by paralelný program nemusel fungovať korektne. Účelom nasledujúcich príkladov je ale prezentovať fungovanie modelov v **Boboxe**, a teda implementácia vnútornej logiky samotného boxu nie je podstatná. Pre dosiahnutie čo najjednoduchšej implementácie boxu bude použitý príklad s výpisom na štandardný výstup s upozornením, že sa nejedná o reálne riešenie.

Obr. 2.1: Model výpočtu, ktorý vypisuje na štandardný výstup *Hello World!*

Na obrázku 2.1 je znázornený model programu *Hello World!*, tj. model typického učebnicového programu vypisujúceho na štandardný výstup pozdrav. Na tomto modele je demonštrovaný práve jeden užívateľom implementovaný box *hello_box* a dva povinné **Boboxom** poskytované boxy *input_box* a *output_box*. Každý model musí obsahovať počiatočný vstupný bod výpočtu a koncový výstupný bod výpočtu. Tieto dva body sú v modele zastúpené práve povinnými boxami *input_box* a *output_box*, ktorých názvy sú vyhradené. Týmito boxami je určené miesto, kde začne tok výpočtu a miesto, kde program ukončí výpočet, keď sem vstúpi tok výpočtu.

```
class HelloBox : public bobox::group_box {
public:
  virtual void init_impl() {
    // Pridanie vstupnej skupiny.
    add_group(group_index_type(0), column_index_type(0),
      get_input_descriptors(input_index_type(0)));

    // Zahajenie cakania na vstup vypoctu do
    // boxu skrz prvu vstupnu skupinu.
    receive(input_index_type(0), group_index_type(0));
  }
```

```
  void synch_mach_etwas() {
    // Toto je miesto pre funkcnu logiku boxu.
    std::cout << 'Hello World!' << std::endl;

    // Odoslanie vypoctu do dalsieho boxu.
    send_poisoned(input_index_type(0));
  }
};
```

Zdrojový kód 2.1: Príklad možnej implemntácie boxu, ktorý zodpovedá boxu *hello_box* z modelu, ktorý je znázornený v diagrame 2.1.

Okrem vytvorenia modelu musí programátor implementovať box *hello_box*, ktorý bude v tomto prípade triviálne vypisovať pozdrav na štandardný výstup.

V ukážke zdrojového kódu 2.1 je predvedený príklad možnej implementácie boxu *hello_box*. Metóda *init_impl()* slúži na inicializáciu boxu, a to predovšetkým jeho vstupov a výstupov. Vstupy a výstupy sú určené pomocou indexov, ktoré sú definované poradím zápisu v modele. Box sprístupňuje vstupy a výstupy pomocou abstrakcie skupín. V kóde metódy *init_impl()* je možné vidieť založenie vstupnej skupiny a následne zahájenie čakania na vstup. Metóda *init_impl()* je automaticky volaná **Boboxom** v období medzi konštrukciou boxu a spustením výpočtu boxu, ktorý je vykonávaný metódou *synch_mach_etwas()*.

Po obdržaní správy na definovanom čakajúcom vstupe je spustená metóda *synch_mach_etwas()*, ktorá má na starosti hlavnú časť výpočtu. V zobrazenom príklade je vykonaný len výpis pozdravu „*Hello World!*“. Nakoniec box posiela správu, ktorá prenesie výpočet na ďalší box podľa definície v modele výpočtu.

Pre ďalšie pochopenie textu nie je potrebné rozumieť technickým detailom práce so vstupnými a výstupnými skupinami boxu[4]. Dôležité je však vedieť, že box po obdržaní správ na určených vstupoch vykoná prácu a môže odoslať správy výstupmi pre ďalšie boxy. Takto posielané správy okrem toho, že riadia tok programu môžu niesť aj dáta.

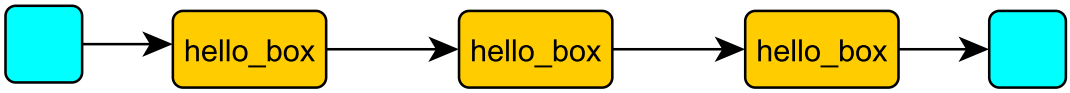
Je zrejmé, že klasický učebnicový kód programu *Hello World* je jednoduchší ako zobrazené prevedenie pomocou **Boboxu**. Príklad však demonštruje dostatočnú jednoduchosť vytvárania boxov pri schopnosti jednoducho paralelizovať výpočet, ktorá bude prezentovaná ďalej.

Predvedený príklad modelu na obrázku 2.1 je tak triviálny, že ho nie je možné paralelizovať. Celý výpočet prebehne v rámci jedného volania boxu *hello_box*. Pre ukážku paralelizácie bude teda požadované aby program vypisoval pozdrav na štandardný výstup tri krát.

Ako už bolo zmienené, zvolený príklad nie je síce úplne šťastnou voľbou, nakoľko prístup k štandardnému výstupu nie je synchronizovaný a výpisy sa môžu pri paralelizácii prelínať. **Bobox** zabezpečuje synchronizáciu všetkých dát a prostriedkov, ktoré sám spravuje a poskytuje. Avšak dáta a prostriedky, ktoré sú mimo **Bobox** podliehajú bežným problémom paralalného programovania. Pokiaľ by aj bol prístup k štandardnému výstupu synchronizovaný, tak paralelizácia by nemala žiaden prínos, pretože boxy len zapisujú do výstupu. Takže by bo-

[4]V prípade záujmu sú bilžšie podrobnosti uvedené v [3].

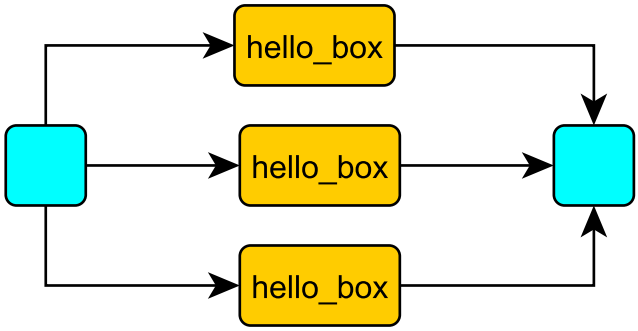
xy čakali a postupne zapisovali na výstup rovnako ako v sekvenčnom prevedení. Pre jednoduchosť demonštrácie paralelizácie bude implementácia boxu ponechná v nesynchronizovanom prevedení ako je naznačené v ukážke kódu 2.1 a výpisy sa tak budú môcť na výstupe prelínať.



Obr. 2.2: Sekvenčný model výpočtu, ktorý vypisuje na štandardný výstup „*Hello World! Hello World! Hello World!*“

Pre trojnásobný výpis je nutné použiť *hello_box* v modele tri krát. Jednoduchým zreťazením troch boxov *hello_box* za sebou vznikne sekvenčný model, ktorý bude vykonávať požadovaný trojitý výpis použitím troch nezávislých inštancií boxov. Diagram tohoto modelu výpočtu je znázornený na obrázku 2.2. Priebeh výpočtu tohto programu však stále neprebehne paralelne, nakoľko v modele sú boxy zapojené tak, ako keby vzájomne ich výpočty na sebe záviseli, tj. vstup druhého boxu je výstup prvého boxu a vstup tretieho boxu je výstupom druhého boxu.

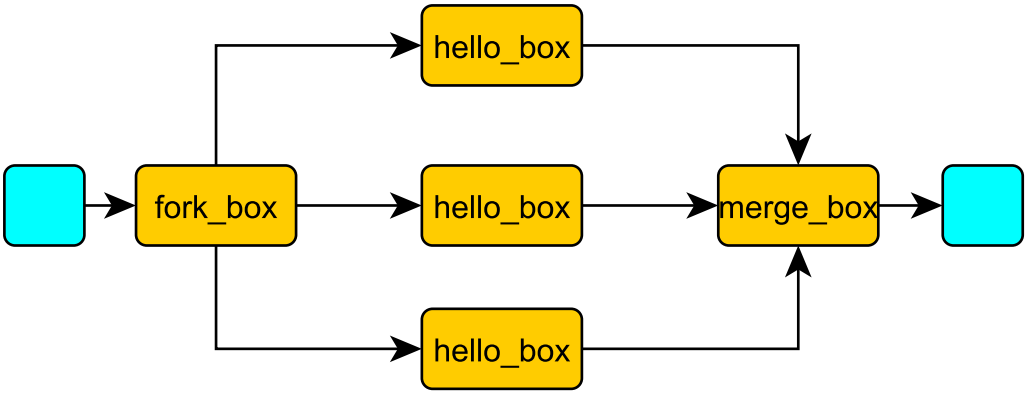
Pokiaľ sú boxy v modele zapojené nezávisle, tak **Bobox** môže vykonať paralelizáciu. Paralelné zapojenie boxov *hello_box* pre trojitý výpis pozdravu je znázornené na obrázku 2.3.



Obr. 2.3: Paralelný model výpočtu, ktorý vypisuje na štandardný výstup „*Hello World! Hello World! Hello World!*“

Kvôli technickému obmedzeniu frameworku **Bobox** však nie je možné vetviť výpočet priamo zo vstupného boxu *input_box* a nie je možné ani zlievať výpočet do výstupného boxu *output_box*. Riešením tohoto obmedzenia je rozšíriť model o dva nové boxy, ktoré budú požadované vetvenie a zlievanie výpočtu realizovať.

Nasledujúci model výpočtu 2.4 je už úplne korektný a výpočet prebehne paralelne. Pretože vstavaný vstupný a výstupný box pracujú len s jedným výstupom, resp. vstupom, tak do modelu pribudli naviac boxy pre rozštiepenie a spojenie výpočtu. Samozrejme, nové boxy je tiež potrebné implementovať. Implementácia týchto boxov je však pomerne triviálna a pozostáva len z odoslania správ a prijatia správ v správnych počtoch.



Obr. 2.4: Paralelizovateľný model Hello World!

```
class ForkBox : public bobox::group_box {
public:
  virtual void init_impl() {
    // Pridanie vstupnej skupiny.
    add_group(group_index_type(0), column_index_type(0),
      get_input_descriptors(input_index_type(0)));

    // Pridanie vystupnych skupin
    for (int i = 1; i <= 3; ++i) {
      add_group(group_index_type(i), column_index_type(0),
        get_output_descriptors(output_index_type(i)));
    }

    // Zahajenie cakania na vstup vypoctu do
    // boxu skrz prvu vstupnu skupinu.
    receive(input_index_type(0), group_index_type(0));
  }

  void synch_mach_etwas() {
    // Odoslanie spravy do vsetkych vystupov.
    for (int i = 1; i <= 3; ++i) {
      send_poisoned(group_index_type(i), outut_index_type(0));
    }
  }
};
```

Zdrojový kód 2.2: Box realizujúci rozštiepenie výpočtu do práve troch vetiev.

```
class MergeBox : public bobox::group_box {
public:
  virtual void init_impl() {

    // Pridanie vstupnych skupin
    for (int i = 0; i < 3; ++i) {
      add_group(group_index_type(i), column_index_type(0),
        get_input_descriptors(input_index_type(i)));
    }
```

```
    // Pridanie vystupnej skupiny.
    add_group(group_index_type(3), column_index_type(0),
      get_output_descriptors(output_index_type(0)));

    // Zahajenie cakania na vstupy vypoctu do
    // boxu skrz vsetky vstupne skupiny.
        for (int i = 0; i < 3; ++i) {
          receive(input_index_type(i), group_index_type(0));
    }

  }

  void synch_mach_etwas() {
    // Odoslanie spravy do vystupu.
      send_poisoned(group_index_type(3), outut_index_type(0));
  }
};
```

Zdrojový kód 2.3: Box realizujúci zlievanienie práve troch vetiev výpočtov do jednej vetvy.

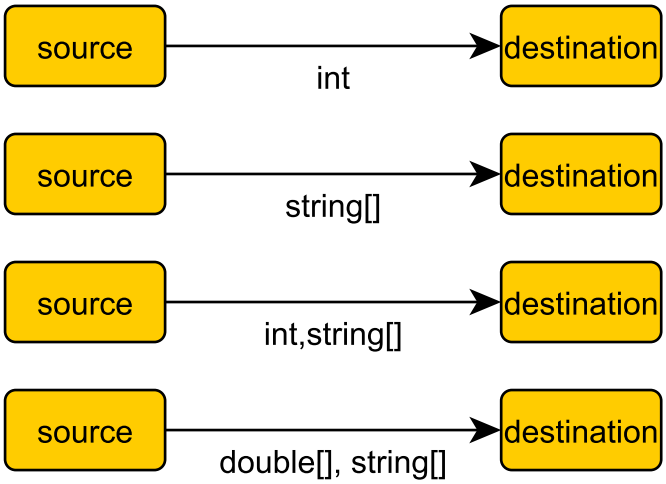
Príklady implementácií boxov starajúcich sa o vetvenie a zlievanie výpočtu sú prakticky málo využiteľné, nakoľko sú obmedzené na konkrétny počet spracovávaných vetiev výpočtu a neošetrujú všetky možné použitia. Pre demonštračný model sú však tieto implementácie postačujúce. Vetvenie a zlievanie výpočtu je v modeloch pomerne časté, preto je vhodné predpripraviť si boxy, ktoré budú dostatočne variabilné. Prípadne framework poskytuje bázu základných vstavaných boxov, medzi ktorými sú aj zmienené boxy.

Doposiaľ prezentované modely výpočtu neobsahovali žiadne prenosy dát medzi boxami. Boxy si medzi sebou posielajú správy, ktorými sa riadi tok programu. Naviac môžu správy obsahovať dáta, ktoré chce predať jeden box inému. Vo svojej podstate je prenos dát spolu s predaním toku výpočtu typickejší ako len predávanie toku výpočtu. Posielané dáta môžu byť skaláry alebo vektory hodnôt. Pre zaznačenie prenášanej hodnoty sa bude do diagramu modelu zapisovať k spojeniu typ prenášanej hodnty. Využívať sa bude notácia C++, tj. skaláry budu napr. *int* alebo *string* a vektory budú *int*[] a *string*[]. Špeciálnym prípadom bude typ *void*, ktorý bude značiť, že sa neprenášajú žiadne dáta. Význam zápisu *void* je len zvýraznenie-potvrdenie absencie prenosu dát a môže byť v diagrame vynechaný. Modely, a teda aj diagramy, budú často použiteľné pre rôzne typy hodnôt. V tomto prípade budú použité zástupné označenia *type* či *type*[], za ktoré budú v realizácii dosadené skutočné typy.

Na obrázku 2.5 sú znázornené spojenia, ktoré prenášajú rôzne kombinácie dát od jedného skaláru nesúceho reálne číslo až po dvojicu vektorov nesúcich reálne čísla a textové reťazce.

Samotná práca s dátami v boxoch je pomerne priama. Potrebné je však poznať typ prenášaných dát a ich poradie v prichádzajúcej, resp. odchádzajúcej správe, ktoré je určené zápisom v modele. V ukážke kódu 2.4 je predvedené načítanie hodnoty z vektoru typu *int*[], ktorý vstupuje do boxu ako prvý, teda na indexe 0. Načítaná je konkrétne prvá hodnota vektoru. Následne je vyko-

naný aj zápis do vektoru. Zapísaný je dvojnásobok načítanej hodnoty, pričom sa zapisuje opäť na prvú pozíciu prvého vektoru.



Obr. 2.5: Výseky modelov znázorňujúce prenos dát medzi boxami.

```
void synch_mach_etwas() {
  // Ziskanie hodnoty prvej bunky z prveho
  // vektoru vstupujuceho do boxu.
  int value = get_data<int>(column_index_type(0))[0];
  // Zapis hodnoty do prvej bunky prveho vektoru boxu.
  get_data<int>(column_index_type(0))[0] = value * 2;
  // A dalsi kod ...
}
```

Zdrojový kód 2.4: Čítanie a zápis dát prenášaných medzi boxami.

Kapitola 3

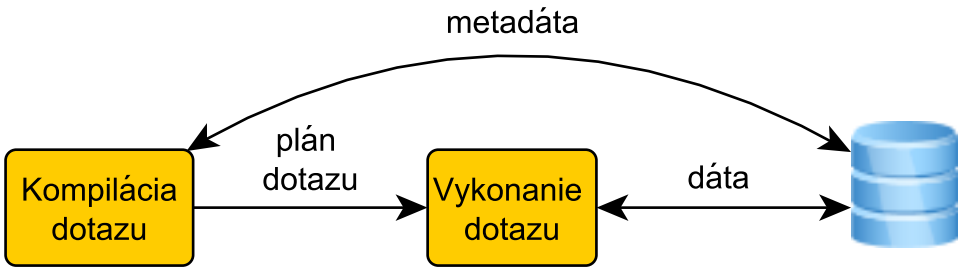
Databázové operácie

V tejto kapitole budú popísané databázové operácie a ich rôzne prevedenia. Ďalej sa čitateľ oboznámi s niektorými základnými pojmami, ktoré bude potrebovať pri čítaní ďalšieho textu. V neposlednej rade kapitola obsahuje popisy rôznych sekvenčných algoritmov vybraných databázových operácií, na základe ktorých budú v ďalšej časti diplomovej práce vytvorené paralelné prevedenia pre spracovanie vo frameworku **Bobox**.

3.1 Spracovanie dotazu

Spracovanie dotazu do databázy je zložené z dvoch hlavných fáz:

1. kompilácia dotazu
2. vykonanie dotazu



Obr. 3.1: Spracovanie dotazu.

Proces kompilácie dotazu je pomerne komplikovaný, podrobný popis je možné nájsť napríklad v [6] a [7]. Pre pochopenie ďalšieho textu je postačujúce vedieť, že výstupom kompilácie je fyzický plán pre vykonanie dotazu.

Fyzický plán je optimalizovaná sada úkonov, ktoré je potrebné vykonať, aby bol realizovaný celý dotaz. Pod základnými stavebnými blokmi fyzického plánu je často možné si predstaviť operácie, ktoré zodpovedajú operáciám relačnej algebry (podrobnosti je možné naštudovať v [8]). Naviac je sada operácií rozšírená

o pomocné činnosti ako napríklad prenesenie dát z vedľajšej do hlavnej pamäte a podobne.

Pre jednotlivé kroky plánu existuje niekoľko možných prevedení/realizácií, ktoré majú rôzne vlastnosti a ich použitie je vhodné v rôznych situáciách. Voľba konkrétnej realizácie, resp. algoritmu, pre každý použitý úkon je základnou časťou transformácie logického plánu dotazu na fyzický plán dotazu počas kompilácie. Súčasťou fyzického plánu dotazu je teda aj určenie najvhodnejšieho algoritmu pre vykonanie jednotlivých krokov. Voľba správneho prevedenia konkrétneho kroku je ovplyvnená mnohými faktormi[1], ktoré je možné zistiť z metadát.

Algoritmy používané pre realizáciu operácií vo fyzickom pláne dotazu je možné rozdeliť do nasledujúcich kategórií:

1. algoritmy založené na triedení
2. algoritmy založené na hašovaní
3. algoritmy založené na použití indexu

Prípadne je tiež možné rozdeliť jednotlivé algoritmy podľa počtu priechodov dátami, ktoré sú potrebné pre vykonanie algoritmu:

1. jedno-priechodové algoritmy
2. dvoj-priechodové algoritmy
3. viac-priechodové algoritmy

V nasledujúcej časti tejto kapitoly budú popísane jedno-priechodové algoritmy, pre ktoré budú v ďalšej kapitole špecifikované vytvorené paralelné verzie algoritmov. Popis bude obsahovať aj základné myšlienky dvoj-priechodových algoritmov, ktoré budú tiež využité pri implementácii paralelizácie databázových operácií. Konkrétne sa jedná o predspracovanie dát. K dvoj-priechodovým algoritmom, ako takým, nebudú poskytnuté paralelné verzie algoritmov, a to hlavne z dôvodu, že dvoj-priechodové algoritmy intenzívne využívajú k svoje práci sekundárnu pamäť, čo by bolo v rozpore s prúdovým spracovaním dát, ktorým sa **Bobox** vyznačuje. Avšak budú vytvorené paralelné verzie algoritmov kombinujúce prvky jedno a dvoj-priechodových metód.

Viac-priechodovými algoritmami sa táto práca nezaoberá, nakoľko sa jedná o zovšeobecnenie dvoj-priechodových algoritmov, a teda trpia rovnakými obmedzeniami. Z popisu algoritmov sú vynechané algoritmy založené na použití indexu, pretože nie sú dobre implementovateľné s dostupnými možnosťami prúdového spracovania a ani neobsahujú žiadne vlastnosti, ktoré by sa použili pri paralelizácii a neboli zmienené už v predošlých algoritmoch. O nepopísaných algoritmoch sa je možné dočítať napríklad v [9],[10] a [11].

Uvedené algoritmy nepredpokladajú žiadne špeciálne vlastnosti, ktoré by boli kladené na vstupné dáta a sú tak obecne použiteľné. Jediným faktorom

[1] Najzákladnejšie faktory, ktoré majú vplyv sú napr. vlastnosti dát (veľkosť dát, rozsah uložených hodnôt, indexy ...) ale aj vlastnosti hardware.

vplývajúcim na použiteľnosť algoritmov je mohutnosť dát. V prípade znalosti vhodnej vlastnosti dát (napr. zotriedenie) by bolo možné navrhnúť efektívnejšie algoritmy. Zámerom práce je však zaoberať sa možnosťami paralelizácie a následným preskúmaním jej vlastností. Pre tento zámer sú zvolené algoritmy postačujúce.

3.2 Základné pojmy

Základné kroky fyzického plánu - databázové operácie, ktoré budú ďalej popísané sú:

1. selekcia
2. projekcia
3. funkcie na stĺpcoch
4. agregačné funkcie
5. zoskupovanie záznamov
6. eliminácia duplicít
7. zjednotenie
8. prienik
9. rozdiel
10. karteziánsky súčin
11. spojenie

Všetky vymenované operácie pracujú s reláciami, resp. operandy operácií sú relácie.

Definícia 1. *(relácia, doména relácie)*
Nech $D_1, D_2, D_3 \ldots D_n, n \geq 1$ sú množiny hodnôt (domény), tak reláciou rozumieme ľubovolnú podmnožinu karteziánskeho súčinu $D_1 \times D_2 \times D_3 \times \ldots \times D_n$, kde karteziánsky súčin $D_1 \times D_2 \times D_3 \times \ldots \times D_n$ nazveme doménou relácie.

Zmienené databázové operácie sa delia podľa arity na unárne a binárne. Operácie vymenované pod očíslovaním 1.-6. spracovávajú jednu reláciu a zvyšné operácie 7.-11. dve relácie.

Dôležitým faktorom pri práci algoritmov realizujúcich uvedené operácie je forma vstupných a výstupných dát. Uvedené algoritmy budú pripúšťať dve rôzne formy:

- množinovú a

- s opakovaním záznamov[2].

Dáta v podobe množiny nepripúšťajú žiaden alebo pripúšťajú práve jeden výskyt záznamu a dáta s opakovaním pripúšťajú aj viacnásobný/opakovaný výskyt záznamu. V kontexte tejto informácie treba chápať aj definíciu relácie (definícia 1), ktorá sa pre dáta s opakovaním záznamov zmení tak, že množiny vymení za multi-množiny (množiny pripúšťajúce opakovanie prvkov).

Vstupné dáta algoritmov budú vždy v podobe dát s opakovaním záznamov. Algoritmy pre množinové dáta na vstupe sú totožné s algoritmami pre dáta s opakovaním záznamov, ale naviac je možné optimalizovať výpočet skorším ukončením spracovávania záznamu po prvom spracovaní, nakoľko o dátach je známe, že obsahujú maximálne jednu inštanciu záznamu. Prípadne nie je žiaden algoritmus potrebný, pretože dáta sú už v požadovanej podobe (eliminácia duplicít, zoskupovanie záznamov).

Algoritmy pre elimináciu duplicít a zoskupovanie záznamov umožňujú výstup len v podobe množiny, čo plynie priamo z povahy požadovaného výsledku. Algoritmy selekcie, projekcie, funkcie na stĺpcoch a agregačných funkcií sa formou výstupných dát nezaoberajú sú aplikované priamo na príchodzie dáta. Pokiaľ je požadovaný množinový výstup, je to možné docieliť aplikáciou postupov pre elimináciu duplicít, či zoskupovanie záznamov.

Definícia 2. *(početnosť záznamu)*
Nech D je doména relácie R tak, potom je definovaná funkcia $p(R, x)$, ktorá pre $\forall x \in D$ vráti počet výskytov záznamu x v relácii R.

Pre ostatné operácie zjednotenie, prienik a rozdiel sú možné obe formy výstupných dát.

Definícia 3. *(výstup s opakovaním záznamov operácií $\cup$,$\cap$ a $\setminus$)*
Nech relácie R a S majú spoločnú doménu relácie D, potom výstup s opakovaním prvkov bude pre operácie $R \cup S$, $R \cap S$ a $R \setminus S$ následovný:

$R \cup S$**:** *Vo výsledku sa objaví každý záznam, ktorý sa nachádza aspoň v jednej z relácií R a S. Početnosť každého záznamu vo výsledku je určená ako $p(R, x) + p(S, x)$.*

$R \cap S$**:** *Výsledok operácie prieniku je tvorený záznamami, ktoré splnia podmienku $p(R, x) > 0 \ \wedge \ p(S, x) > 0$. Početnosť každého záznamu vo výsledku je určená vzťahom $min(p(R, x), p(S, x))$.*

$R \setminus S$**:** *Výsledok je tvorený záznamami, pre ktoré platí $p(R, x) - p(S, x) > 0$ a početnosť záznamu vo výsledku je daná ako $p(R, x) - p(S, x)$.*

Definícia 4. *(množinový výstup operácií $\cup$,$\cap$ a $\setminus$)*
Nech relácie R a S majú spoločnú doménu relácie D, potom množinový výstup bude pre operácie $R \cup S$, $R \cap S$ a $R \setminus S$ nasledovný:

[2] V anglickej literatúre sa používajú výstižnejšie výrazy pre dáta pripúšťajúce opakovanie záznamov, a to multiset alebo bag.

$R \cup S$: *Vo výsledku sa objaví každý záznam, ktorý sa nachádza aspoň v jednej z relácií R a S. Početnosť každého záznamu vo výsledku je presne* 1.

$R \cap S$: *Výsledok operácie prieniku je tvorený záznamami, ktoré splnia podmienku $p(R, x) > 0 \ \wedge \ p(S, x) > 0$. Početnosť každého záznamu vo výsledku je* 1.

$R \setminus S$: *Výsledok je tvorený záznamami, pre ktoré platí $p(R, x) > 0 \wedge p(S, x) = 0$ a početnosť záznamu vo výsledku je* 1.

Karteziánsky súčin a spojenie produkujú na výstup dáta s opakovaním prvkov. Pokiaľ je požadovaný množinový výstup, tak je potrebné aplikovať operáciu eliminácie. Pre operáciu spojenia bude používaný znak $\bowtie$, takže operácia spojenia pre relácie R a S bude vyzerať $R \bowtie S$.

Definícia 5. *(θ-join, equi-join, natural-join)*
Výsledkom operácie spojenia $R \bowtie S$ je relácia, ktorá obsahuje záznamy vytvorené spojením záznamu z R a záznamu z S. Spojenie záznamov sa vykonáva na základe splnenia podmienok kladených na určené atribúty záznamov oboch relácií R a S. Podľa typu podmienok je možné rozlíšiť:

- *θ-join*
  Na vymenované atribúty relácií sa aplikuje ľubovolná podmienka resp. porovnávací operátor ($<, \leq, =, \neq, \geq, >$). Do výsledného spojenia prispejú len tie záznamy, ktorých hodnoty v určených atribútoch splnia podmienku.

- *equi-join*
  equi-join je špeciálny prípad θ-join, v ktorom sa aplikuje len operátor rovnosti.

- *natural-join*
  Obdoba equi-join, kde atribúty, na ktoré sa aplikuje podmienka, sú vybrané automaticky na základe zhody mena atribútu v reláciách R a S.

V ďalšom texte diplomovej práce bude pod spojením vždy myslené prirodzené spojenie, tj. *natural-join*. K implementácii spojenia typu *equi-join* je možné použiť postupy platné pre *natural-join*, pretože sa jedná len o vhodné premenovanie atribútov. Operácia *θ-join* môže byť realizovaná pomocou aplikácie operácie selekcie na *equi-join* alebo operáciu karteziánskeho súčinu. Operáciu *θ-join* je možné implementovať aj efektívnejšími postupmi, tie však prekračujú rámec práce. Popisy nezmienených možností spojení je možné nájsť v [12].

Požadovaný výstup algoritmov spojenia a karteziánskeho súčinu bude s opakovaním prvkov, pokiaľ by bol žiaduci množinový výstup, je to možné docieliť použitím eliminácie duplicít.

3.3 Jedno-priechodové algoritmy

Táto časť textu je zameraná na vysvetlenie algoritmov, ktoré dokážu vykonať požadovanú operáciu jedným priechodom dát, resp. jedným čítaním dát z disku. Podmienkou použitia väčšiny algoritmov v tejto skupine je, že sa vstupné

dáta musia zmestiť do operačnej pamäte. Presnejšie je možné rozdeliť algoritmy do štyroch skupín podľa požiadaviek na vyhradený priestor v pamäti:

1. jeden záznam - unárna operácia
2. jedna relácia - unárna operácia
3. jedna relácia - binárna operácia
4. dve relácie - binárna operácia

Do prvej skupiny patria algoritmy pre selekciu, projekciu, funkcie na stĺpcoch, zjednotenie s opakovaním záznamov, a tiež aj algoritmus pre agregačné funkcie, keď sa neaplikuje spoločne so zoskupovaním. V prípade zmienených algoritmov je možné načítať záznamy jednotlivo a jednotlivo ich aj spracovávať, preto je nutné mať k dispozícii priestor v pamäti aspoň pre jeden záznam.

Načítanie dát po jednom zázname z disku je veľmi nevhodné, pretože je veľmi neefektívne a degraduje praktickú použiteľnosť algoritmu. Jedná sa len o teoretickú medzu použiteľnosti algoritmu, v praxi je nutné dáta čítať po väčších blokoch, ktoré obsahujú podstatne viac záznamov ako jeden.

Druhá skupina je zastúpená algoritmami pre zoskupovanie, elimináciu duplicít, a tiež algoritmom pre agregačné funkcie, ktorý je spojený s aplikáciou zoskupovania. Jedná sa o unárne operácie a v najhoršom prípade je potrebné zmestiť do pamäte celú reláciu, tj. všetky vstupné dáta operácie.

Do tretej skupiny náležia algoritmy pre prienik, rozdiel, karteziánsky súčin a algoritmus spojenia. V tomto prípade je nutné do pamäte vtesnať vždy celú jednu reláciu a druhú reláciu je v najhoršom prípade pri nedostatku pamäte možné spracovávať po jednotlivých záznamoch. Tento postup v medznom prípade je však značne neefektívny a ako už bolo popísane dáta je vhodné načítať po blokoch.

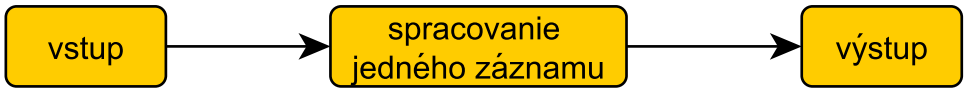
Posledná skupina je tvorená jediným algoritmom zjednotenia s množinovým výstupom, ktorý vždy vyžaduje načítať obe relácie súčasne do pamäte.

Jedno-priechodové algoritmy nevyužívajú k svojej práci externú pamäť[3]. Keďže vstupno/výstupné operácie sekundárnej pamäte sú pomerne drahé v porovnaní s prácou v primárnej pamäti, tak sa jedno-priechodové algoritmy radia k najefektívnejším riešeniam databázových operácií. Na druhej strane, ich obmedzenie je dané väčšinou veľkosťou operačnej pamäte, a teda použitie je možné len pre obmedzenú veľkosť dát. Výnimku z tohto pravidla tvoria algoritmy, ktoré sú schopné vykonať operáciu na jedinom zázname.

3.3.1 Selekcia, projekcia a funkcie na stĺpcoch

Algoritmy selekcie, projekcie a funkcie na stĺpcoch, ktoré pripúšťajú opakovanie záznamov pre vstup aj výstup, patria medzi najjednoduchšie databázové operácie. V prípade požadovaného množinového výstupu je potrebné aplikovať postupy eliminácie duplicít popísnane v 3.3.3.

[3]Pokiaľ nie je uvažované načítanie vstupných dát z disku a zápis výsledku na disk ako súčasť algoritmu.



Obr. 3.2: Priebeh operácie selekcie, projekcie a funkcie na stĺpci.

Na obrázku 3.2 je znázornený jednoduchý priebeh vykonania všetkých algoritmov tejto sekcie. V prvom kroku sa načíta jeden záznam, ktorý sa v ďalšom kroku spracuje a následne môže byť priamo predaný na výstup. Spracovanie záznamu sa samozrejme odlišuje podľa typu vykonávanej operácie:

Selekcia
: Na záznam sa aplikujú podmienky a pokiaľ sú splnené, môže byť presunutý na výstup. *(Za zmienku stojí, že selekciu je možné (vhodné) realizovať aj pomocou indexu, čo môže ušetriť zbytočné načítanie dát z disku. Algoritmami s použitím indexu sa však práca nezaoberá, pretože realizácia algoritmov tejto práce nepracuje s načítaním dát z disku priamo, ale dáta sú realizované ako prúd vstupujúci do a vystupujúci z výpočtu[4].)*

Projekcia
: Zo záznamu sú vybrané požadované hodnoty, ktoré tvoria požadovaný výstup.

Funkcia
: Požadované hodnoty zo záznamu sú predané do funkcie a návratová hodnota funkcie je predaná na výstup.

Do skupiny funkcií sa zaraďujú aj aritmetické operácie vykonané na hodnotách v stĺpcoch.

3.3.2 Agregačné funkcie

Algoritmy z tejto kapitoly potrebujú k svojmu výpočtu kontextovú informáciu o už spracovaných dátach. Spravidla sa však jedná o jednu hodnotu a nie o všetky doposiaľ spracované údaje, ako tomu bude v ďalších predstavených algoritmoch. Agregačné funkcie akumulujú hodnotu konkrétneho atribútu záznamu naprieč všetkými spracovávanými záznamami. Akumulovaná hodnota je pre agregačné funkcie zrejmá:

COUNT: Pre každý spracovaný záznam sa pripočíta jednotka k akumulovanej hodnote. Nakoniec akumulovaná hodnota obsahuje počet spracovávaných záznamov. Iniciálna hodnota je 0.

MIN a MAX: V akumulovanej hodnote je uložená najmenšia, resp. najväčšia zatiaľ spracovaná hodnota atribútu záznamu. Iniciálna hodnota je maximum, resp. minimum, použitého typu.

[4]Opačný prístup s prispôsobením algoritmov pre prácu z diskom je možné nájsť v [14]

SUM: Hodnota atribútu spracovávaného záznamu sa pripočíta k aktuálnej akumulovanej hodnote. Výsledkom je súčet hodnôt atribútu skrz všetky záznamy. Počiatočná hodnota akumulovanej hodnoty je 0.

AVG: V podstate akumuluje dve hodnoty, a to rovnakým spôsobom ako agregačné funkcie SUM a COUNT, nech zodpovedajúce akumulované hodnoty sú *sum* a *count*. Po spracovaní všetkých záznamov sa spočíta priemer ako podiel *sum*/*count*. Iniciálne hodnoty sú rovnaké ako pri SUM a COUNT.

Spracovanie záznamov môže prebiehať obdobne ako v 3.3.1, kde je potrebné mať k dispozícii jeden spracovávaný záznam. Agregačné funkcie naviac vyžadujú priestor pre jednu (resp. dve) premennú, v ktorej sa akumuluje výpočet.

Uvedený postup je platný pre výpočet nad reláciou ako celkom. Je však potrebné uvedomiť si, že agregačné funkcie častejšie pracujú nad skupinami záznamov vytvorených zoskupením záznamov ako nad celou reláciou. Postup pre vykonanie zoskupovania s aplikáciou agregačných funkcií je možné nájsť v sekcii 3.3.4.

3.3.3 Eliminácia duplicít

Pri eliminácii duplicít je už potrebné udržiavať v pamäti viac ako jednu hodnotu. Algoritmus je naznačený v ukážke kódu 3.1, ktorá používa vyhľadávaciu štruktúru T pre detekciu duplicitného výskytu záznamov v relácii R.

```
foreach r in R do
  if r is not in T then
    insert r into T
    output r;
  fi
done
```

Zdrojový kód 3.1: Pseudokód algoritmu eliminácie duplicít. R predstavuje reláciu, ktorej duplicitné záznamy sa eliminujú a T je vyhľadávacia štruktúra.

V najhoršom prípade budú všetky záznamy z R unikátne a vyhľadávacia štruktúra T bude nakoniec obsahovať všetky záznamy z relácie R. Takže dostupný priestor v hlavnej pamäti pre realizáciu algoritmu musí byť dostatočný pre všetky záznamy z relácie R plus réžiu vyhľadávacej štruktúry.

Vyhľadávacia štruktúra by mala podporovať pridávanie záznamu a zisťovanie výskytu záznamu v najlepšom možnom čase[5]. Pre takéto použite sú vhodné hašovacie tabuľky, či vyvážené vyhľadávacie stromy. V ďalšom texte už nebude explicitne zmieňovaná réžia vyhľadávacej štruktúry. Vždy, keď bude zmienené, že je potrebné mať dostupnú nejakú pamäť pre záznamy a ukladanie bude prebiehať do vyhľadávacej štruktúry, tak bude implicitne uvažovaná aj potrebná pamäť pre réžiu vyhľadávacej štruktúry. V tomto svetle má algoritmus eliminácie pamäťovú zložitosť $O(|R|)$.

[5] V ideálnom prípade s konštantnou zložitosťou.

3.3.4 Zoskupovanie záznamov

Zoskupovanie záznamov prebieha podľa určených atribútov relácie. Vykonanie zoskupenia môže prebiehať v spojení s aplikáciou agregačných funkcií alebo samostatne.

V prípade prevedenia zoskupenia bez aplikácie agregačných funkcií sa jedná de facto o elimináciu duplicít 3.3.3, v ktorej sa ako kľúč pre vyhľadávanie nepoužije celý záznam, ale len atribúty určujúce zoskupovanie.

```
foreach r in R do
  a := get grouping attributes from r
  if a is  in T then
    v := get accumulated value from T for a
    v := accumulate v and r
    update accumulated value in T for a with v
  else
    v := init_accumulated_value
    v := accumulate v and r
    insert (a, v) into T
  fi
done

foreach (a, v) in T do
  output a, v
done
```

Zdrojový kód 3.2: Pseudokód algoritmu zoskupenia záznamov s aplikáciou agregačnej funkcie. R je relácia, nad ktorou sa vykonáva zoskupovanie, a T je vyhľadávacia štruktúra. Samotná akumulácia a iniciálna hodnota (*init_accumulated_value*) sú závislé na type agregačnej funkcie (popis viď sekcia 3.3.2).

Zoskupovanie s použitím agregačných funkcií prebieha v dvoch fázach. Najskôr sa akumulujú hodnoty podľa typu agregačnej funkcie obdobne ako v 3.3.2. Rozdiel je v tom, že sa počíta naraz viacero akumulovaných hodnôt, pre každú skupinu jedna. Akumulované hodnoty sa udržujú vo vyhľadávacej štruktúre, ktorá ako kľúč používa atribúty, podľa ktorých prebieha zoskupovanie záznamov. V druhej fáze prebieha posielanie výsledkov na výstup, ktorý je vytvorený z akumulovaných hodnôt a hodnôt atribútov, podľa ktorých sa vykonalo zoskupovanie.

V prípade, že pre každý záznam vstupnej relácie R vznikne zoskupením samostatná skupina, tak je potrebná pamäť pre chod algoritmu $|R| * size(record)$, kde $size(record)$ je veľkosť jedného záznamu v T, ktorý je tvorený atribútmi zlučovania a akumulovanou hodnotou. Pamäťová zložitosť je opäť $O(|R|)$.

3.3.5 Operácia zjednotenia

Pri operácii zjednotenia je už potrebné rozlíšiť typy výstupných sád - klasické množinové prevedenie a prevedenie, ktoré umožňuje opakovanie prvkov. Všetky ďalšie uvedené jedno-priechodové algoritmy budú obsahovať dve prevedenia pre typy výstupov, ktoré už boli zmienené.

Zjednotenie s opakovaním prvkov

Vlastnosť opakovania prvkov umožňuje jednoduché riešenie zjednotenia. Pre zjednotenie $R \cup S$ stačí poslať na výstup R a následne S, pričom povolenie opakovania záznamov zaručuje korektnosť tohto postupu.

Klasické množinové zjednotenie

Algoritmus pre množinové zjednotenie $R \cup S$ je opakovane aplikovaný algoritmus pre elimináciu duplicít 3.3.3. Najskôr sa algoritmus aplikuje na reláciu R a následne na reláciu S, pričom vyhľadávacia štruktúra T sa zdieľa pri oboch „elimináciách“ (viď ukážka kódu 3.3).

```
foreach x in R do
  if x is not in T then
    insert x into T
    output x;
  fi
done

foreach x in S do
  if x is not in T then
    insert x into T
    output x;
  fi
done
```

Zdrojový kód 3.3: Pseudokód algoritmu zjednotenia, ktorého výsledok je množina. R a S sú zjednocované relácie a T je vyhľadávacia štruktúra.

Pamäťová zložitosť eliminácie pre reláciu Q je $O(|Q|)$, takže pri dvojnásobnej postupnej aplikácii pre dve relácie R a S je pamäťová zložitosť $O(|R| + |S|)$. Pokiaľ pre $R \cup S$ majú obe relácie všetky záznamy unikátne (aj vzájomne), tak je potrebné do vyhľadávacej štruktúry umiestniť všetky záznamy z oboch relácií súčasne, čo zodpovedá najhoršiemu prípadu.

3.3.6 Operácia prieniku

Pri operácii prieniku $R \cap S$ je vhodné rozlíšiť, ktorá z relácií je menej početná a tú načítať do vyhľadávacej štruktúry. Nech pre $R \cap S$ platí, že $|R| \leq |S|$. Pamäťovým obmedzením algoritmu prieniku je načítanie do pamäte jednej celej (menej mohutnej) relácie, pretože druhú je možné v najhoršom prípade načítať po jednotlivých záznamoch[6]. Pamäťová zložitosť je $O(|R|)$, ak platí vyššie uvedená podmienka pre veľkosti relácií v $R \cap S$.

Prienik s opakovaním prvkov

Pri riešení algoritmu s vyhľadávacou štruktúrou, ktorá udržuje samostatne aj duplicitné záznamy, sú vždy pamäťové požiadavky na maximálnej možnej hra-

[6]Načítanie po jednom zázname však znamená značnú réžiu a je vhodné sa v takomto prípade poohliadnuť po inom riešení. Teoreticky to je však možné.

nici, tj. $O(|R|)$ pri ukladaní R do vyhľadávacej štruktúry. Pokiaľ je však vyhľadávacia štruktúra „rozumnejšia“ a skladuje len jednu inštanciu záznamu a počítadlo duplicít tak, ako je uvedené aj v pseudokóde algoritmu 3.4, potom má algoritmus v určitých prípadoch (častý výskyt duplicít) značne menšie pamäťové požiadavky.

```
foreach x in R do
  if x is in T then
    increment counter of x in T
  else
    insert x into T
  fi
done

foreach x in S do
  if x is in T then
    counter := get counter of x in T
    if counter > 0 then
      decrement counter of x in T
      output x;
    fi
  fi
done
```

Zdrojový kód 3.4: Pseudokód algoritmu pre $R \cap S$ (s opakovaním záznamov), kde R a S sú vstupné relácie a T je vyhľadávacia štruktúra.

Klasický množinový prienik

Prienik s množinovým výstupom umožňuje skladovať len jednu inštanciu záznamu a duplicity ignorovať, preto má podobné vlastnosti v spotrebe pamäti, ako varianta algoritmu prieniku s opakovaním prvkov realizovaná s počítadlom duplicít vo vyhľadávacej štruktúre uvedená v predošlom odstavci. Algoritmus je znázornený v ukážke kódu 3.5.

```
foreach x in R do
  if x is not in T then
    insert x into T
  fi
done

foreach x in S do
  if x is in T then
    remove x from T
    output x;
  fi
done
```

Zdrojový kód 3.5: Pseudokód algoritmu pre $R \cap S$ (množinový výstup), kde R a S sú vstupné relácie a T je vyhľadávacia štruktúra.

3.3.7 Operácia rozdielu

Nech pre obe prevedenia operácie rozdielu (množinové prevedenie a prevedenie s opakovaním záznamov) platí predpoklad, že $|R| \leq |S|$. Pri operácii rozdielu je potrebné brať do úvahy okrem veľkostí $|R|$ a $|S|$ aj nekomutatívnosť tejto operácie. V prevedení algoritmov je teda rozdiel či požadujeme $R \setminus S$ alebo $S \setminus R$.

Ďalej uvedené prevedenia algoritmov s opakovaním záznamov používajú vyhľadávaciu štruktúru s počítaním duplicít, tak ako algoritmy pre prenik. Všetky úvahy o požiadavkách na pamäť uvedené v sekcii 3.3.6 sú platné aj pre algoritmy operácie rozdielu.

Rozdiel $R \setminus S$ s opakovaním prvkov

Algoritmus pre $R \setminus S$ s opakovaním prvkov, ktorý ukladá reláciu R do vyhľadávacej štruktúry pracuje v troch krokoch:

1. Uloženie relácie R do vyhľadávacej štruktúry.
2. Dekremntácia početnosti duplicít záznamov vo vyhľadávacej štruktúre podľa početnosti záznamov v relácii S.
3. Odoslanie záznamov na výstup podľa početnosti, ktorá je zaznamenaná vo vyhľadávacej štruktúre.

```
foreach x in R do
  if x is in T then
    increment counter of x in T
  else
    insert x into T
  fi
done

foreach x in S do
  if x is in T then
    decrement counter of x in T
  fi
  c := get counter of x in T
  if c = 0 then
    remove x from T
  fi
done

foreach x in T do
  c := get counter of x in T
  output x (c times)
done
```

Zdrojový kód 3.6: Pseudokód algoritmu pre $R \setminus S$ (výstup s opakovaním prvkov), kde R a S sú vstupné relácie a T je vyhľadávacia štruktúra.

Tretí krok bude v najhoršom prípade (tj. pokiaľ $R \cap S = \emptyset$) vykonávať priechod celou reláciou R. Pre prípad, keď sa zmestí do pamäte tiež druhý operand operácie rozdielu, je vhodnejšie voliť verziu algoritmu s vkladaním druhého operandu do vyhľadávacej štruktúry, pretože tá je vykonávaná bez dodatočného tretieho kroku.

Rozdiel $S \setminus R$ s opakovaním prvkov

V prípade, že je požadované zistiť výsledok operácie $S \setminus R$, tak sa rovnako ako v predošlom prípade načíta do vyhľadávacej štruktúry R, keďže platí $|R| \leq |S|$. Dôležité je uvedomiť si, že R je druhý operand operácie rozdielu.

```
foreach x in R do
  if x is in T then
    increment counter of x in T
  else
    insert x into T
  fi
done

foreach x in S do
  if x is in T then
    decrement counter of x in T
    c := get counter of x in T
    if c = 0 then
      remove x from T
    fi
  else
    output x
  fi
done
```

Zdrojový kód 3.7: Pseudokód algoritmu pre $S \setminus R$ (výstup s opakovaním prvkov), kde R a S sú vstupné relácie a T je vyhľadávacia štruktúra.

Klasický množinový rozdiel $R \setminus S$

Algoritmus pre množinový rozdiel $R \setminus S$ s ukladaním prvého operandu do vyhľadávacej štruktúry je obdobou už uvedeného algoritmu pre rozdiel $R \setminus S$ s opakovaním záznamov a s ukladaním prvého operandu do vyhľadávacej štruktúry. Množinové prevedenie tiež prebieha v troch krokoch, v ktorých sa však ignorujú početnosti záznamov (pre porovnanie viď pseudokód 3.8 a 3.6).

Opäť sa v algoritme objavuje problém tretieho kroku - priechodu vyhľadávacou štruktúrou, kvôli zisteniu výsledku. Vyhnúť sa tomuto problému je možné použitím algoritmu s ukladaním druhého operandu do vyhľadávacej štruktúry, pokiaľ to jeho rozmer umožní.

```
foreach x in R do
  if x is not in T then
    insert x into T
  fi
done
```

```
foreach x in S do
  if x is in T then
      remove x from T
  fi
done

foreach x in T do
  output x
done
```

Zdrojový kód 3.8: Pseudokód algoritmu pre $R \setminus S$ (množinový výstup), kde R a S sú vstupné relácie a T je vyhľadávacia štruktúra.

Klasický množinový rozdiel $S \setminus R$

Ukladaním druhého operandu do vyhľadávacej štruktúry sa situácia oproti predošlej variante zjednoduší a je možné detekovať výsledky bez potreby dodatočného priechodu (viď 3.9).

```
foreach x in R do
  if x is not in T then
    insert x into T
  fi
done

foreach x in S do
  if x is not in T then
      insert x into T
      output x
  fi
done
```

Zdrojový kód 3.9: Pseudokód algoritmu pre $S \setminus R$ (množinový výstup), kde R a S sú vstupné relácie a T je vyhľadávacia štruktúra.

3.3.8 Karteziánsky súčin

Algoritmus pre karteziánsky súčin posiela na výstup spojené záznamy vytvorené tým spôsobom, že každý záznam z R je spojený s každým záznamom z S. Tento algoritmus nutne nepotrebuje mať v pamäti celú reláciu, avšak opakované čítanie záznamov (blokov záznamov) znižuje výkon, respektíve opakované čítanie dát z disku by znamenalo, že sa nejedná o jedno-priechodový algoritmus. Nech pre $R \times S$ platí, že $|R| \leq |S|$, potom je vhodné mať v pamäti celú reláciu R, ktorá bude prechádzaná opakovane vo vnútornom cykle (viď 3.10).

```
foreach s in S do
  foreach r in R do
    output r, s
  done
done
```

Zdrojový kód 3.10: Pseudokód algoritmu pre $R \times S$, kde R a S sú vstupné relácie.

Dáta náležiace relácii S môžu byť načítané postupne podľa potreby. Pri použití karteziánskeho súčinu treba mať tiež na pamäti, že algoritmus produkuje mohutný výstup.

3.3.9 Operácia spojenia

V tejto časti bude predvedená operácia spojenia, presnejšie povedané operácia prirodzeného spojenia označovaná tiež ako *natural-join*.

Nech pre relácie v spojení $R \bowtie S$ platí $|R| \leq |S|$. Samotný algoritmus pre prirodzené spojenie je založený na načítaní menej mohutnej relácie R do vyhľadávacej štruktúry, kde sa ako kľúč použijú atribúty, skrz ktoré prebieha spojenie. Vo vyhľadávacej štruktúre však musí byť uložená celá hodnota záznamu, nakoľko vo výsledku sú požadované aj hodnoty atribútov, ktoré nie sú súčasťou spojenia. Tiež nie je postačujúce ukladanie záznamov s počítaním duplicít, pretože dva záznamy pri zhode v atribútoch spojenia môžu byť rôzne v ostatných atribútoch. Z tohto dôvodu sú vo vyhľadávacej štruktúre uložené zoznamy záznamov tak, aby bolo možné uložiť viacero rôznych záznamov s rovnakými atribútmi spojenia. Priebeh celého algoritmu je znázornený v ukážke kódu 3.11.

```
foreach r in R do
  a := get join attributes from r
  if a is in T then
    l := get list of records from T for a
    insert r into l
    upadte list of records in T for a by l
  else
    l := create list of records
    insert r into l
    insert l into T for a
  fi
done

foreach s in S do
  a := get join attributes from s
  if a is in T then
    l := get list of records from T for a
    foreach r in l do
      output s, r
    done
  fi
done
```

Zdrojový kód 3.11: Pseudokód algoritmu $R \bowtie S$.

3.4 Dvoj-priechodové metódy

Princíp fungovania dvoj-priechodových metód je založený na predspracovaní dát a následnom vykonávaní logiky konkrétnej databázovej operácie. Vďaka predspracovaniu dát je možné dvoj-priechodové algoritmy použiť na pomerne rozsiah-

le sady dát. V praxi sa skoro nevyskytujú dáta rozmerov, ktoré by presahovali možnosti dvoj-priechodového spracovania.

V predošlom texte boli vstupno/výstupné operácie sekundárnej pamäte prezentované trochu vágne a len vo výnimočných prípadoch bolo spomenuté načítanie dát po blokoch. Avšak pre správne a efektívne fungovanie nasledujúcich algoritmov je práca s blokmi dát zásadná, pretože dáta sa počas predspracovania zapisujú na disk a následne sa opätovne načítajú vo fáze vykonania hlavnej logiky databázovej operácie. Dáta budú načítané a zapisované po blokoch[7]. V ďalšom texte bude počet blokov potrebných pre uloženie relácie R označovaný ako B_R a počet blokov, ktoré je možné umiestniť do hlavnej pamäte bude ďalej označovaný ako B_M.

Prvý krok dvoj-priechodových metód je fáza predspracovania dát, ktorá spočíva v zotriedení alebo zahašovaní dát.

Predspracovanie využívajúce triedenie funguje tak, že sa do pamäte načíta najväčší možný úsek dát, ktorý sa následne zotriedi nejakým vhodným triediacim algoritmom. Takto vzniknutý zotriedený úsek dát sa zapíše na disk. Tento postup sa opakuje, kým sa neprejdú všetky dáta. Na disku vznikne niekoľko zotriedených úsekov s dátami. Počet zotriedených úsekov dát na disku nesmie byť väčší ako $B_M - 1$, pretože v druhej fáze dvoj-priechodových algoritmov je potrebné načítať aspoň jeden blok dát pre každý zotriedený úsek a jeden blok je potrebné ponechať voľný pre výsledky.

Pre fázu predspracovania hašovaním je možné v pamäti vytvoriť $B_M - 1$ blokov, do ktorých hašovacia funkica *h(x)* distribuuje záznamy. Jeden blok je potrebné vyhradiť pre vstup dát pri predspracovaní. Týmto spôsobom je možné vytvoriť na disku maximálne $B_M - 1$ úsekov dát. Každý úsek dát na disku obsahuje záznamy, pre ktoré nadobúda *h(x)* rovnakú hodnotu.

```
foreach block B in R do
  foreach r in B do
    i := h(r)
    insert r into T[i]
    if T[i] is full then
      write T[i] to disk
      erase T[i]
    fi
  done
done

foreach t in T do
  if t is not empty then
    write t to disk
  fi
done
```

Zdrojový kód 3.12: Pseudokód predspracovania dát pomocou hašovania. R je predpracovávaná relácia, $h()$ je hašovacia funkcia a T je vektor $B_M - 1$ blokov, do ktorých sa hašujú záznamy. Jeden blok (B) sa používa na načítanie dát relácie R.

[7] Je samozrejmé, že aj v/v operácie pre jedno-priechodové algoritmy pracujú s blokmi a nie s jednotlivými záznamami, avšak pri ich popise nebolo tak dôležité tento fakt zdôrazniť.

Oba uvedené spôsoby predspracovania dát (triedenie a hašovanie) boli prezentované pre unárne operácie. Pre binárne operácie je potrebné predspracovávať záznamy (resp. bloky záznamov) pre oba operandy separovane. Jedná sa o triviálnu úpravu popísaných postupov a nebude teda ďalej popisovaná.

V druhej fáze pri predspracovaní dát triedením sa do hlavnej pamäte načíta aspoň jeden blok pre každý zotriedený úsek dát z disku. Nad blokmi sa postupne vykonávajú operácie, pričom je možné využiť zotriedenie záznamov. Po vyčerpaní všetkých záznamov z nejakého bloku, sú dáta do bloku doplnené z disku zo zodpovedajúceho zotriedeného úseku dát. Postup sa opakuje, kým nie sú spracované všetky dáta. V druhej časti algoritmov teda dochádza k priebežnému spracovaniu dát zo všetkých zotriedených úsekov dát na disku.

Pre dáta predspracované hašovaním je postup druhej fázy odlišný. Využíva sa fakt, že dáta v rôznych úsekoch na disku spolu vôbec nesúvisia. Spracovanie prebieha po celých úsekoch nezávisle na sebe a je možné jednoducho aplikovať jedno-priechodové algoritmy (viď 3.3).

Pre účely paralelizácie budú použité postupy odvodené od práve prezentovaného predspracovania dát. Samotné dvoj-priechodové algoritmy jednotlivých databázových operácií nebudú ďalej v práci použité a preto nie sú uvedené. K dohľadaniu sú popisy dvoj-priechodových algoritmov napríklad v [13] a [14].

Kapitola 4

Databázové operácie v prostredí Bobox

Nasledujúca kapitola sa zaoberá popisom realizácie vybraných databázových operácií v paralelnom prostredí **Bobox**. Súčasťou výkladu sú názorné diagramy, ktoré znázorňujú modely použité pre riadenie výpočtu databázových operácií. Detailne sú tiež vysvetlené funkcie konkrétnych boxov a ich vzájomných väzieb tak, ako sú aj skutočne implementované.

4.1 Realizácia databázových operácií

Ako súčasť diplomovej práce boli realizované v prostredí **Bobox** nasledujúce databázové operácie:

1. selekcia
2. funkcie na stĺpcoch
3. agregačné funkcie
4. eliminácia duplicít
5. prienik
6. rozdiel
7. spojenie

Vstupné a výstupné dáta algoritmov zodpovedajú definíciám z 3 a pokiaľ nebude výslovne povedané, o ktorý druh výstupu sa jedná, budú implicitne myslené obe možné prevedenia (množinové a aj s opakovaním záznamov). Databázové operácie, ktorých algoritmy zostali zatiaľ neimplementované, sú nasledovné:

1. projekcia
2. zoskupovanie záznamov

3. zjednotenie
4. karteziánsky súčin
5. triedenie záznamov

Chýbajúce realizácie zmienených databázových operácií však z pohľadu paralelizovateľnosti neprinášajú nič nové. Projekcia je podľa požiadaviek v kapitole 3, vykonateľná v konštantnom čase, pretože sa jedná len o výber konkrétnych atribútov z relácie (množinové prevedenie nie je požadované) a nie je potrebný žiaden priechod dátami. Takže paralelizácia by bola v tomto prípade skôr kontraproduktívna. Operácia zjednotenia s množinovým výstupom je tiež prevediteľná v konštantnom čase, viď sekcia 3.3.5. Prevedenie zjednotenia s opakovaním prvkov je možné nahradiť pomocou realizovanej databázovej operácie eliminácie tak, že oba operandy zjednotenia sú spoločne odoslané ako jeden operand do eliminácie. Pre zvyšné nerealizované databázové operácie by realizácia paralelizácie bola prínosom.

Triedenie pomocou **Boboxu** už bolo implementované [15]. Duplicitná implementácia triedenia pre databázové operácie by bola teda zbytočná.

S ohľadom na skúmanie možností paralelizácie databázových operácií v prostredí **Bobox**, nedostupnosť zmienených paralelných realizácií nie je zásadným nedostatkom, nakoľko sa jedná o opätovnú aplikáciu postupov, ktoré sú použité v realizáciách implementovaných databázových operácií. V ďalšom texte budú uvažované už len skutočne implementované databázové operácie.

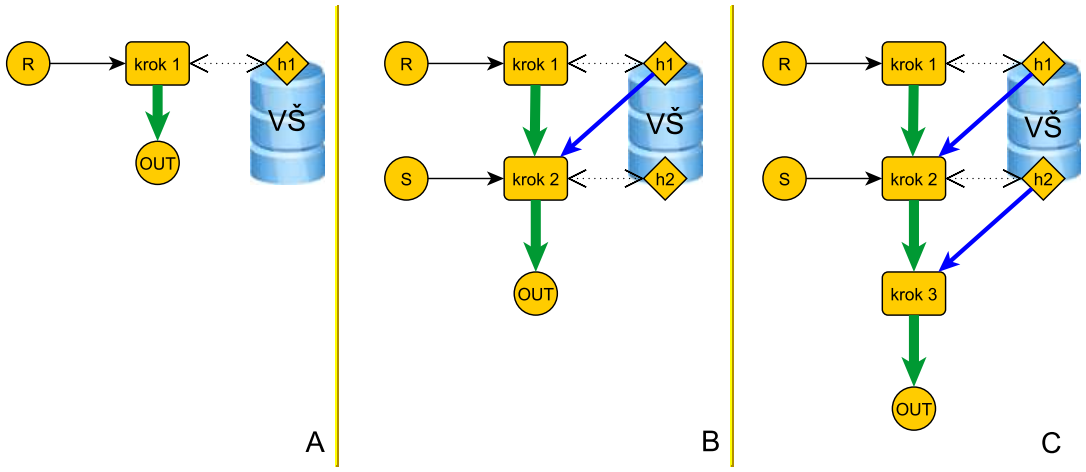
Realizácie jedno-priechodových algoritmov je tiež možné detailnejšie rozdeliť do nasledovných skupín:

1. výpočet bez kontextu
2. výpočet s agregovaným kontextom
3. výpočet s vyhľadávacím kontextom
   (a) spracovanie v jednom kroku
   (b) spracovanie v dvoch krokoch
   (c) spracovanie v troch krokoch
4. výpočet s vyhľadávacím kontextom a predspracovaním dát

Výpočet bez kontextu realizuje databázovú operáciu selekcie a aplikácie funkcie na stĺpce, konkrétne sa jedná o aritmetické operácie $(+, -, *, /)$ na stĺpcoch. Výpočet s agregovaným kontextom zastupuje agregačné funkcie (COUNT, SUM, MIN, MAX a AVG). Implementácia operácií eliminácie, zjednotenia, prieniku, rozdielu a spojenia náleží do skupiny výpočtov vyžadujúcich vyhľadávací kontext. Práve popísané rozdelenie realizovaných operácií vyplýva z potrieb už uvedených sekvenčných algoritmov pre jednotlivé databázové operácie.

Rozdelenie realizácií výpočtov s vyhľadávacím kontextom na ďalšie skupiny je učinené na základe povahy prístupu k spracovaniu dát. Krokom sa rozumie časť výpočtu, kde sa na dáta (časť dát) opakovane aplikuje jedna úprava.

Spracovanie v jednom kroku umožňuje len databázová operácia eliminácie duplicít. Na obrázku 4.1 (časť A) je znázornený priebeh výpočtu v jednom kroku. Jedná sa o realizáciu unárnej operácie, preto vstupuje do výpočtu len jedna relácia R. Po skončení (eliminácie) kroku výpočtu je vyhľadávacia štruktúra v stave $h1$, čo však nie je dôležité, pretože výpočet po jednom kroku skončí.



Obr. 4.1: Kroky vo výpočte s vyhľadávacím kontextom. Výpočet v jednom kroku je zobrazený v časti (A), výpočet v dvoch krokoch je znázornený v (B) a v (C) je možné nájsť troj-krokový výpočet. R a S sú vstupné relácie, $VŠ$ označuje vyhľadávaciu štruktúru, $h1$ a $h2$ predstavujú stavy vyhľadávacej štruktúry v určitom bode výpočtu a OUT je grafické znázornenie konca výpočtu.

Dvoj-krokové spracovanie dát (obrázok 4.1 časť B) vykonáva binárne databázové operácie - prienik, spojenie a rozdiel. Databázová operácia rozdiel je spracovateľná v dvoch krokoch len v prípade, že sa v prvom kroku výpočtu spracováva druhý operand (tj. v kontexte obrázku 4.1 verzia B sa jedná o rozdiel $S \setminus R$). V prvom kroku sa všetky dáta náležiace do relácie R vložia do vyhľadávacej štruktúry, stav štruktúry po prvom kroku je označený ako $h1$. Druhý krok výpočtu môže byť spustený až po skončení prvého kroku, pretože požaduje, aby vyhľadávacia štruktúra obsahovala všetky dáta z R, tj. bola v stave $h1$. V druhom kroku sa spracovávajú dáta z relácie S podľa konkrétnej logiky príslušnej realizovanej databázovej operácie s pomocou informácií uložených vo vyhľadávacej štruktúre. Druhý krok po skončení zanechá vyhľadávaciu štruktúru v stave $h2$[1], čo je už pre dvoj-krokový výpočet irelevantné, nakoľko výsledok operácie už je vypočítaný.

Výsledky databázovej operácie rozdielu, ktorá je realizovaná tak, že sa v prvom kroku ukladá do vyhľadávacej štruktúry prvý operand (obrázok 4.1 časť C zodpovedá rozdielu $R \setminus S$), nie je možné zistiť počas druhého kroku, preto je nutné použiť ďalší krok. Tretí krok slúži na dohľadanie výsledkov vo vyhľadávacej štruktúre, ktorá je zanechaná v stave $h2$ po druhom kroku. Pri dohľadávaní

[1]Pre databázové operácie spojenia a rozdielu s množinovým výsledkom sú stavy vyhľadávacej štruktúry po oboch krokoch výpočtu totožné, tj. $h1 = h2$.

výsledkov sa z vyhľadávacej štruktúry len načítajú dáta, takže sa nemení jej obsah (stav).

Pozorný čitateľ si určite všimol, že všetky zmienené databázové operácie, resp. realizácie databázových operácií, už boli zatriedené do skupín a aj napriek tomu ostáva ešte jedna skupina výpočtov - výpočty s vyhľadávacím kontextom a predspracovaním dát. Realizácie databázových operácií s vyhľadávacím kontextom vytvorené na základe jedno-priechodových algoritmov zdieľajú vyhľadávaciu štruktúru naprieč paralelnými vetvami výpočtu, čo môže byť úzkym hrdlom ich výkonu. Pomocou princípu predspracovania, ktorý je prevzatý z dvoj-priechodových metód (viď sekcia 3.4), je možné zdieľanú vyhľadávaciu štruktúru nepoužiť. V poslednej skupine sa nachádzajú alternatívne realizácie databázových operácií k dvoj-krokovým realizáciám výpočtu s vyhľadávacím kontextom.

4.2 Prostredie realizácie

Súčasťou realizácie databázových operácií sú aj pomocné činnosti[2], ktoré slúžia k tomu, aby bolo možné spúšťať databázové operácie samostatne a vykonávať merania. Pomocné činnosti sú znázornené v každom diagrame, ktorý znázorňuje model výpočtu. Popis pomocných úkonov je súčasťou prvej analýzy realizácie databázovej operácie, tj. aplikácie funkcií na stĺpce. Časť diagramu s logikou databázovej operácie bude vždy zvýraznená pre uľahčenie orientácie.

Všetky realizované databázové operácie sú implementované tak, aby boli pohodlne integrovateľné navzájom. Tento fakt je nesmierne dôležitý pre efektívne vykonávanie celého fyzického plánu, pretože ten je bežne zložený z väčšieho množstva databázových operácií. Databázové operácie môžu byť v pláne nezávislé jedna na druhej, ale aj navzájom závislé, keď výstup jednej je vstupom inej.

V prípade závislých databázových operácií v rámci fyzického plánu dochádza k postupnému vykonávaniu týchto operácií v rade. Pri naivnom spájaní paralelných realizácií databázových operácií je potrebné vykonať naviac množstvo režijnej práce pri rozdeľovaní a spájaní dát[3]. Podrobnejšie informácie o vykonávaní dotazov v paralelnom, resp. distribuovanom prostredí viď [16] a [17].

Naivné zapojenie je zobrazené na obrázku 4.2 a alternatívne optimalizované zreťazenie je znázornené na obrázku 4.3. Pre zjednodušenie popisu oboch zapojení na obrázkoch nech sú znázornené napríklad nasledovné databázové operácie:

Zelená databázová operácia: $(A_1 \cap A_2 \rightarrow A_3)$, ďalej bude táto operácie označovaná ako G

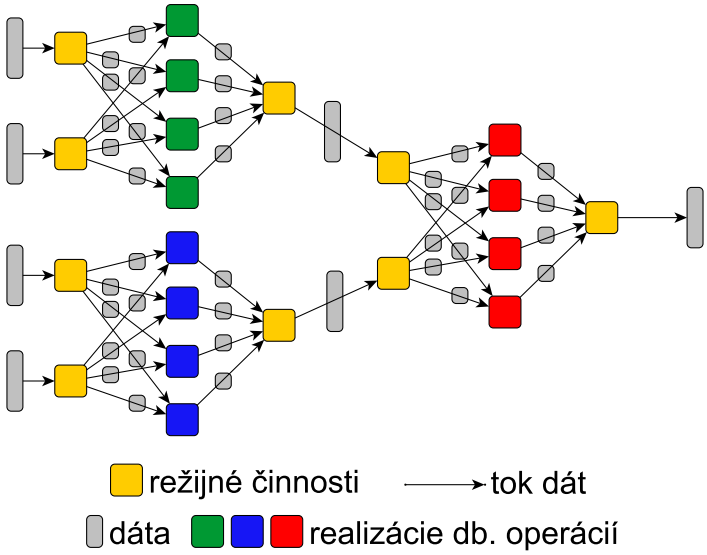
Modrá databázová operácia: $(A_4 \setminus A_5 \rightarrow A_6)$, ďalej bude táto operácie označovaná ako B

Červená databázová operácia: $(A_3 \cap A_6 \rightarrow A_7)$, ďalej bude táto operácie označovaná ako R

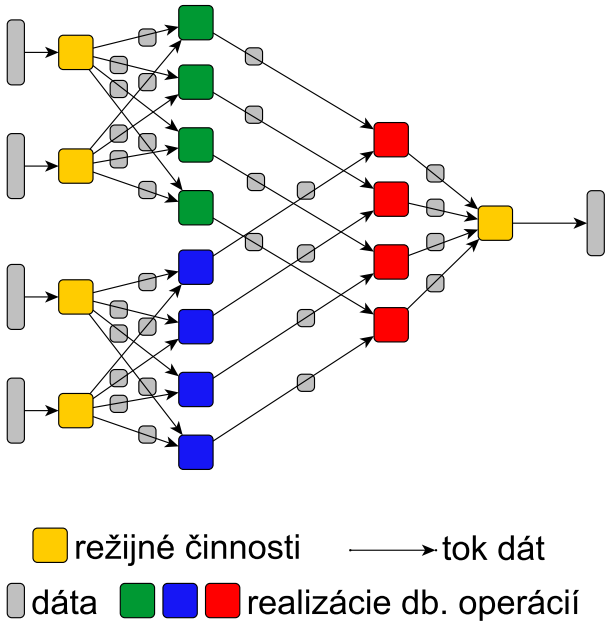
[2]Pomocné činnosti sú prvotné rozvetvenie výpočtu, načítanie vstupných dát, meranie času výpočtu, výstup výsledkov (ak je požadovaný) a záverečné zliatie výpočtu.

[3]Réžiou nie je len samotné rozdelenie a spojenie dát ale aj vytváranie, spúšťanie a plánovanie boxov, ktoré delenie dát realizujú. Naviac sú vykonávané aj zbytočné prenosy dát.

Takže zobrazovaná časť fyzického plánu dotazu prezentuje nasledovný zápis $(A_1 \cap A_2) \cap (A_4 \setminus A_5)$.



Obr. 4.2: Spájanie paralelných realizácií databázových operácií s réžiou rozdeľovania a spájania dát pre paralelné vetvy výpočtu pre každú databázovú operáciu samostatne.



Obr. 4.3: Spájanie paralelných realizácií databázových operácií so zdieľanou réžiou rozdeľovania a spájania dát pre paralelné vetvy výpočtu.

Pri oddelenom zapojení (obr. 4.2) dochádza k zbytočnému zlievaniu výsledkov operácií G a B. Následne sú práve spojené dáta rozdelné pre potreby paralelného spracovania operáciou R. Tieto kroky v rámci vykonávania fyzického plánu sú už

zbežným pohľadom redundantné. V optimalizovanom zreťazení môžu realizácie databázových operácií posielať výsledky priamo do nadväzujúcich databázových operácií, tj. vstupy realizácií operácií G a B sú priamo napojené, ako vstup do databázovej operácie R (obr. 4.3).

Z tohoto dôvodu budú diagramy znázorňujúce modely, použité pre riadenie výpočtu databázových operácií, obsahovať viacnásobný výstup do pomocných boxov [4]. Schopnosťou prijímať rozvetvený vstup a posielať rovnako rozvetvené výsledky umožňuje zmienenú integráciu.

4.3 Výpočet bez kontextu

V tejto skupine sa nachádzajú realizácie algoritmov, ktoré sú popísané v sekcii(3.3.1). Databázové operácie z tejto skupiny je možné vykonať priechodom nad dátami bez ďalšej doplňujúcej informácie, tj. bez kontextu.

Algoritmy s týmito vlastnosťami sú triviálne paralelizovateľné, pretože každý jeden výpočet je absolútne nezávislý na ostatných výpočtoch. Pre študovanie vlastností paralelizácie a hlavne pre porovnanie s komplikovanejšími prípadmi boli zvolené, ako zástupcovia tejto skupiny, aritmetické operácie na stĺpcoch a selekcia. Výpočet je vykonaný podľa modelu, ktorý vznikne na základe diagramu 4.4, dosadením konkrétneho boxu (ktorý vykonáva selekciu podľa určenej podmienky alebo aritmetickú operáciu) za box *operation*.

Rozdiel v realizácii selekcie a aritmetickej operácie je, že selekcia neposiela dáta vždy na výstup, ale len keď je splnená podmienka, ktorú realizuje. Naopak, aritmetické operácie pre každý jeden vstup vyprodukujú práve jeden výstup. Samotný paralelný algoritmus pre databázové operácie selekcie a funkcie na stĺpcoch je triviálny viď 4.1.

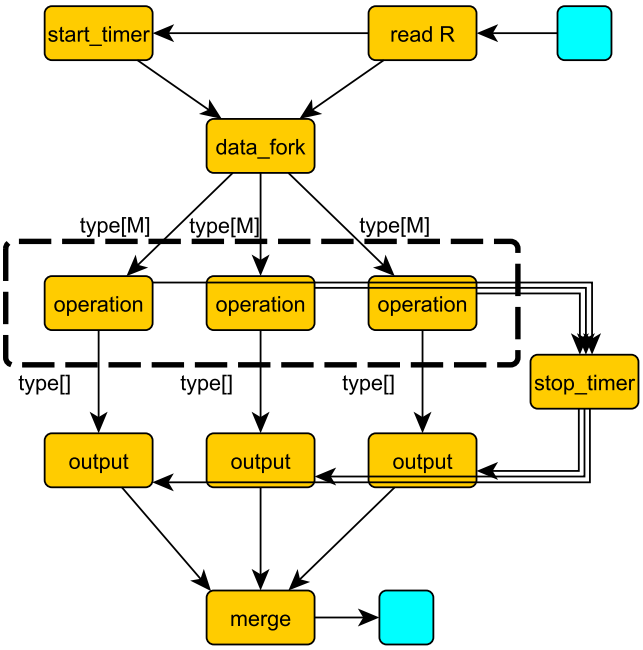
```
C := split up R to chunks
foreach c in C do_parallel
  foreach r in c do
    operation(r);
  done
parallel_done
```

Zdrojový kód 4.1: Paralelný algoritmus pre realizáciu databázovej operácie bez kontextu. Logika db. operácie je vykonaná vo funkcii *operation*(), napr. súčet dvoch atribútov relácie.

Diagramy modelov uvedené v texte práce presne zodpovedajú implementáciám databázových operácií, ktoré sú použité na meranie vlastností a následnú analýzu možností paralelizácie. Každý diagram modelu znázorňuje okrem časti realizácie algoritmu databázovej operácie aj pomocné časti ako načítanie dát, meranie času či výstup výsledkov výpočtu. Hlavná časť modelu realizujúca samotný algoritmus databázovej operácie je v diagrame zakaždým zvýraznená ohraničením hrubou čiarkovanou čiarou.

[4] Pre diagram znázorňujúci výpočet agregačnej funkcie to však neplatí, pretože výsledkom výpočtu je len jediná hodnota.

K meraniu výkonnosti (času vykonania výpočtu) je potrebné ešte zdôrazniť, že všetky dáta sú počas merania stále v hlavnej pamäti, tj. včetne vstupných aj výstupných dát.



Obr. 4.4: Šablóna výpočtu pre aritmetické operácie. Reálne obsahuje realizácia namiesto boxov *operation* konkrétne boxy pre operácie či selekciu. Počet paralelných vetiev je konfigurovateľný, v znázornenom diagrame sú tri paralelné vetvy vykonávajúce výpočet.

Boxy

V realizácii databázovej operácie, ktorá zodpovedá diagramu 4.4, sa vyskytujú boxy, ktorých funkcia je nasledovná:

1. *fork* - vetvenie výpočtu z 1 vetvy do n vetiev.
2. *read R* - načítanie vstupných dát/relácie R do výpočtu (implementácie sú dostupné pre vstup z csv súboru, relačnej databázy, a tiež pre dáta zadané priamo v zdrojovom kóde). V ďalšom texte diplomovej práce bude použitý aj box *read S*, ktorého funkcia je analogická.
3. *start_timer* - spustenie merania času výpočtu.
4. *data_fork* - box, ktorý očakáva na vstupe dáta a následne rozvetví výpočet do n vetiev, kde do každej novej vetvy pošle časť dát, ktorá je veľká úmerne k pomeru veľkosti pôvodných dát a počtu novovzniknutých vetiev.
5. *operation* - predstavuje šablónový box, za ktorý sa dosadí box realizujúci podmienku selekcie alebo vykonávajúci konkrétnu aritmetickú operáciu.

6. *stop_timer* - zastavenie merania čau, ktoré bolo spustené pomocou boxu *start_timer*.
7. *write_result* - zápis výsledku.
8. *merge* - zliatie výpočtu z n paralelných vetiev do 1 vetvy.

Výpočet

Súčasťou popisu priebehu výpočtu databázovej operácie selekcie (resp. aritmetickej operácie na stĺpcoch) bude aj vysvetlenie pomocných krokov v testovacom prostredí. V popisoch realizácií ostatných databázových operácií bude pomocným častiam venovaný len striedmy alebo žiadny popis, pretože majú rovnakú povahu vo všetkých realizáciách. Priebeh výpočtu:

1. Načítaj dáta vstupnej relácie do výpočtu. (*read R*).
2. Vstupné data sú po načítaní odoslané do boxu *data_fork*, ktorý však čaká na ďalší vstup, ktorý má definovaný z boxu *start_timer*.
3. Box *read R* po načítaní a odoslaní celej relácie *R* pošle signál[5] do boxu *start_timer*.
4. Box *start_timer* spustí časomieru a predá tok programu od boxu (*data_fork*), ktorý má na starosti rozdelenie dát do paralelných vetiev výpočtu.
5. Box *data_fork* po obdržaní dát z boxu *read R* a signálu z boxu *start_timer* na definované vstupy prepočíta veľkosti dát podľa stupňa paralelizácie a rozošle naporcované dáta k boxom vykonávajúcim funkciu na stĺpcoch, resp. selekciu.
6. Box *operation* vykonáva operáciu nad všetkými dátami, ktoré dostal pridelené. Výsledky priebežne posiela do výstupných boxov.
7. Po skončení výpočtu (po spracovaní všetkých vstupných dát) pošle box *operation* signál na zastavenie časomiery do boxu *stop_timer*.
8. Box *stop_timer* čaká na signály od všetkých boxov vykonávajúcich hlavnú logiku (boxy *operation*). Keď obdrží požadované signály, zastaví meranie času a rozošle signály boxom výstupu - *output*.
9. Boxy výstupu *output* v prostredí s meraním času, ako je popísané v predošlých bodoch, čakajú až na ukončenie merania času výpočtu. Potom začnú spracovávať svoj vstup.
10. Výstupné boxy zapíšu[6] výsledok výpočtu a predajú tok programu do boxu *merge*.

[5] Signál je v podstate správa bez dát.

[6] Pri spusteniach, kvôli meraniu času sa výstup nikde nezapisoval. Výstupné boxy len prijali dáta ale nikde ich nezapísali.

11. Box *merge* čaká, kým neobdrží signál od všetkých boxov *output* a potom ukončí výpočet.

Je potrebné si uvedomiť, že pomocné kroky (1-5 a 7-11) by boli pri vykonávaní reálneho plánu dotazu nahradené výstupom inej databázovej operácie, resp. vstupom, tak ako bolo popísané v úvode kapitoly.

4.4 Výpočet s agregovaným kontextom

Modely s agregovaným kontextom realizujú algoritmy, ktoré sú popísane v kapitole 3.3.2. Konkrétne sa jedná o algoritmy pre agregačné funkcie, ktoré potrebujú k svojmu výpočtu informáciu o predošlom výpočte.

Pri naivnej snahe previesť sekvenčnú variantu algoritmov na paralelnú sa môže javiť zdieľanie kontextovej informácie naprieč paralelne bežiacimi vetvami výpočtu nevyhnutné. Synchronizácia zdieľanej premennej sa tak môže stať úzkym hrdlom paralelizácie. Pri hlbšom zamyslení sa nad algoritmami pre agregačné funkcie je badateľné, že je možné spracovávané dáta rozdeliť na úseky, nad ktorými je možné nezávisle od ostatných úsekov vypočítať agregovaný medzivýsledok. Konečný výsledok sa nakoniec dopočíta z jednotlivých medzivýsledkov. Uvedený postup je korektný z dôvodu, že kumulované operácie sú komutatívne. Popísaný postup vo forme pseudokódu je možné nájsť v ukážke 4.2.

```
C := split up R to chunks
A := create container of aggregated values (by count of chunks)
foreach c in C do_parallel
  foreach r in c do
    x := aggregate(r);
    update value in A for c by x
  done
parallel_done

foreach a in A do
  v := aggregate(a)
done

output v
```

Zdrojový kód 4.2: Paralelný algoritmus pre realizáciu databázovej operácie s agregovaným kontextom. Pre operáciu **AVG** je potrebné kód doplniť a nakoniec z agregovaných hodnôt ešte dopočítať výsledok.

Zmienené prevedenie nepotrebuje žiadne zdieľanie dát medzi vetvami výpočtu medzivýsledkov a tým je možné plne využiť paralelizáciu. Problematický sa môže javiť záver algoritmu, ktorý nepracuje paralelne, avšak počet paralelne bežiacich vetiev je malý, potom aj počet medzivýsledkov je nízky, a teda záverečný priechod bude rýchly[7].

Diagram šablóny pre modely agregačných funkcií je znázornený na obrázku 4.5.

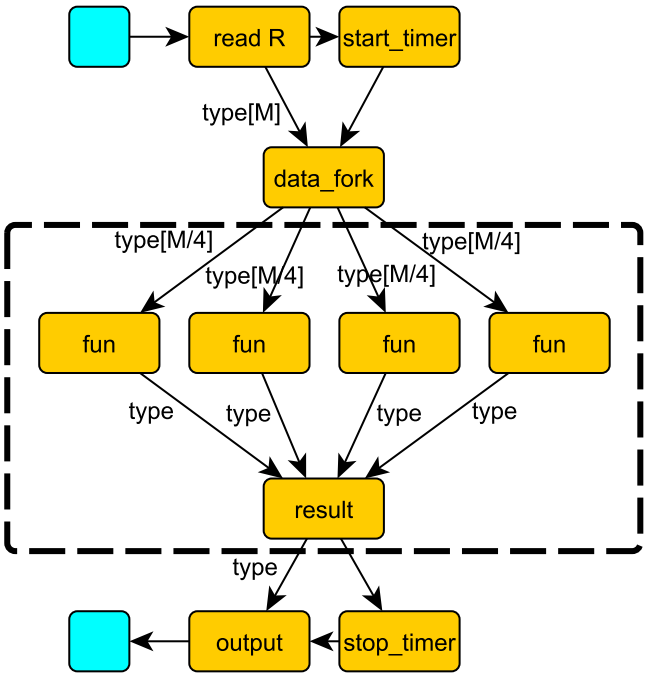
[7]Rýchly v porovnaní s časom potrebným na spracovanie veľkých dát v predošlom kroku.

Boxy

V diagrame modelu 4.5 sú opätovne použité niektoré boxy z modelu 4.4 a ich funkcionalita je zachovaná aj v tomto modele. Do modelu pribudli naviac ďalšie boxy:

1. *fun* - realizácia výpočtu konkrétnej agregačnej funkcie (medzivýsledky).
2. *result* - dopočítanie výsledku z medzivýsledkov.

Rozdelenie výpočtu do dvoch rôznych typov boxov zodpovedá predvedenému algoritmu v ukážke kódu 4.2. Boxy *fun* sa starajú o paralelnú časť algoritmu a box *result* vykonáva záverečný cyklus dopočítavajúci výsledok.



Obr. 4.5: Šablóna modelu výpočtu agregačných funkcií, kde za boxy *fun* je v konkrétnych modeloch doplnený box implementujúci logiku konkrétnej agregačnej funkcie.

Výpočet

Výpočet podľa diagramu modelu 4.5 prebieha v pomocných častiach z veľkej časti obdobne ako v prípade diagramu 4.4. Každá pomocná činnosť je v tomto modele síce vykonávaná len jedným boxom[8], ale analógia postupu je zrejmá. Naviac ako bolo popísané v predošlom texte, jedná sa o kroky, ktoré sú potrebné len pre samostatné vykonanie databázovej operácie a pri vykonávaní databázovej operácie ako súčasti plánu nejakého dotazu by boli vynechané. Z týchto dôvodov bude popísané len jadro výpočtu modelu vykonávané boxami *fun* a *result*.

[8] V predošlom prípade bolo výstupných boxov viac.

Po obdržaní úseku dát na spracovanie vykoná box *fun* svoju prácu, a to bez kontextu výsledkov ostatných vetiev vykonaných nad inými časťami dát. V podstate prebehne úplne klasický algoritmus pre sekvenčný výpočet uvedený v kapitole 3.3.2.

Po spočítaní hodnoty agregovaného medzivýsledku pre zvolený úsek dát box *fun* prikročí k odoslaniu získanej hodnoty do boxu *result*, ktorý na základe typu agregačnej funkcie dopočíta konečný výsledok:

COUNT a SUM - medzivýsledky sa navzájom spočítajú,

MIN a MAX - z medzivýsledkov sa vyberie minimum resp. maximum,

AVG - medzivýsledky pre súčet a počet sa oddelene spočítajú a výsledok výpočtu sa získa ako podiel agregovaného súčtu a počtu.

4.5 Výpočet s vyhľadávacím kontextom

Výpočty popísané v predošlom texte boli jednoducho prevedené sekvenčné algoritmy na paralelný výpočet. Nasledujúce popisy výpočtov sú vytvorené z popísaných algoritmov tiež dostatočne priamo, avšak dochádza ku komplikácii prístupu k vyhľadávacej štruktúre. Paralelné algoritmy je možné komponovať pomerne jednoducho so zdieľaním a následnou synchronizáciou vyhľadávacej štruktúry alebo komplikovanejšie, kde je potrebné najskôr dáta predspracovať, ale následne je možné využívať vyhľadávaciu štruktúru bez synchronizácie, keďže nedochádza ku konfliktným situáciám.

Je obtiažne rozhodnúť, ktorý prístup je efektívnejší, preto budú pre niektoré operácie použité oba prístupy a na základe experimentálnych meraní bude prípadne možné rozhodnúť.

Najskôr budú uvedené všetky postupy, ktoré využívajú zdieľanú vyhľadávaciu štruktúru a následne výpočty s predspracovaním dát.

4.6 Vyhľadávacia štruktúra

Pre zvyšné algoritmy, ktoré budú uvedené v tejto kapitole, je nutné použiť vyhľadávaciu štruktúru. Do úvahy prichádzajú rôzne prevedenia vyhľadávacích stromov či hašovacie tabuľky. Dôležité je si pri voľbe vyhľadávacej štruktúry uvedomiť požiadavky vznikajúce paralelizáciou algoritmov. **Bobox** je síce navrhnutý tak, aby umožnil spracovanie skalárnych či vektorových dát v paralelnom prostredí a v čo najväčšej miere oslobodil programátora od problémov synchronizácie prístupu k zdieľaným dátam, avšak zdieľaný prístup k dátam vo vyhľadávacej štruktúre presahuje zmienené možnosti. Takže pokiaľ bude vyžadovaná synchronizácia prístupov k štruktúre, je ju potrebné dodatočne zabezpečiť.

Ako vyhľadávacia štruktúra bola zvolená hašovacia tabuľka[9], ktorá je implementovaná s ohľadom na požiadavky realizovaných algoritmov. Schopnosť

[9]Viac o možnostiach hašovania v datbázových systémoch sa je možné dočítať v [18].

synchronizácie bola zabudovaná priamo do tabuľky a v záujme zachovania čo najmasívnejšej paralelizácie boli uplatnené dve opatrenia:

1. Tabuľka poskytuje synchronizované a nesynchronizované rozhranie. To znamená, že v prípade, keď nie je hašovacia tabuľka zdieľaná, tak nemusí výpočet zbytočne niesť záťaž, ktorú synchronizácia prináša. Dokonca je možné rozhrania dynamicky meniť počas výpočtu, čo je dobre využiteľné vďaka povahe implementovaných algoritmov. Nesynchronizovaný prístup je vhodný v prípadoch, keď je postačujúce len čítanie. V algoritmoch s viacerými prístupmi k vyhľadávacej štruktúre (hašovacej tabuľke) dochádza k oddeleným prístupom rôzneho typu.

   Napríklad v prvej sekvencii prístupov sa do tabuľky len zapisuje a až keď skončia všetky zápisy, tak sa v druhej sekvencii prístupov z tabuľky len číta. V tomto prípade je možné ľahko rozoznať aplikáciu synchronizovaného a nesynchronizovaného rozhrania.

2. V prípade synchronizovaného prístupu nie je synchronizácia realizovaná na úrovni celej tabuľky, ale až na úrovni riadkov. Teda pokiaľ nedôjde ku kolízii pri hašovaní, tak prebieha prístup k tabuľke stále paralelne a bez čakania. Tento spôsob synchronizácie dostatočne zabezpečuje korektnosť prístupu k dátam.

Pre úplnosť je potrebné ešte zmieniť, že prípady kolízií záznamov v hašovacej tabuľke sú riešené pomocou reťazenia záznamov.

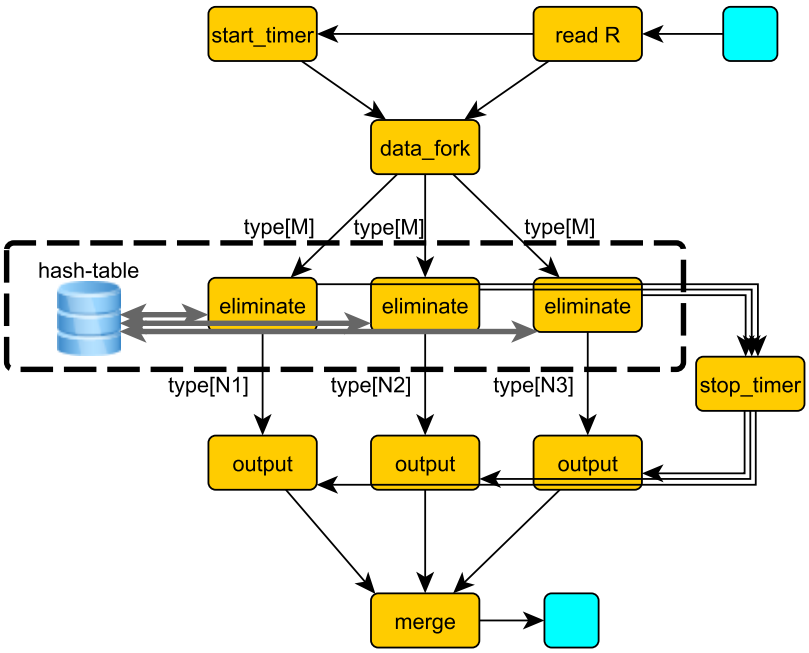
4.6.1 Jedno-krokový výpočet

Jediný algoritmus, ktorý je možné realizovať jedno-krokovým výpočtom je algoritmus pre elimináciu duplicít 3.3.3. Všetky vstupné dáta sa spracujú v jedinom kroku a pre každý spracovávaný záznam sa použije totožný postup, pri ktorom sa kontroluje výskyt záznamu vo vyhľadávacej štruktúre (hašovacej tabuľke). Prvý výskyt záznamu, ktorý je indikovaný neprítomnosťou záznamu v hašovacej tabuľke, sa pošle na výstup a uloží do hašovacej tabuľky. Neskoršie výskyty záznamu v spracovávaných dátach sa ignorujú. Detekcia násobného výskytu záznamu je vykonávaná pomocou hašovacej tabuľky, ktorá je zdieľaná medzi paralelnými vetvami výpočtu - eliminácie.

```
C := split up R to chunks
foreach c in C do_parallel
  // Nasleduje pseudokód práce boxu eliminate.
  foreach r in c do
    if r is not in T then
      insert r into T
      output r;
    fi
  done
  // Koniec práce boxu eliminate.
parallel_done
```

Zdrojový kód 4.3: Pseudokód algoritmu paralelnej eliminácie duplicít. R predstavuje reláciu, ktorej duplicitné záznamy sa eliminujú a T je zdieľaná haš. tabuľka.

Preklopenie sekvenčného algoritmu 3.1 do paralelnej verzie 4.3 je priame. Vstupné dáta sa rozdelia na úseky, ktoré sú následne spracovávané v paralelných vetvách výpočtu, ktoré vlastne realizujú klasický sekvenčný algoritmus. Každá paralelná časť výpočtu však používa zdieľanú hašovaciu tabuľku, čo zabezpečí, že prvý výskyt nejakého prvku v jednej vetve výpočtu bude zohľadnený aj v ostatných.



Obr. 4.6: Znázornenie modelu výpočtu eliminácie duplicít so zdieľanou hašovacou tabuľkou.

Boxy

V modele 4.6 sa vyskytuje len jeden nový typ boxu:

1. *eliminate* - eliminácia duplicitných záznamov. (V pseudokóde 4.3 je pomocou komentárov vyznačená časť kódu, ktorá zodpovedá boxu *eliminate*)

Výpočet

Pomocné časti výpočtu (diagram 4.6) sú obdobné ako v predchádzajúcich prípadoch (diagramy 4.4 a 4.5). Logika eliminácie je umiestnená v boxoch *eliminate* a bola uvedená už skôr.

4.6.2 Dvoj-krokové výpočty

Medzi dvoj-krokové výpočty sa radia realizácie algoritmov pre prienik (kap. 3.3.6), rozdiel[10] (kap. 3.3.7) a spojenie (kap. 3.3.9). Každý z dvoch krokov výpočtu

[10] Jedná sa len o rozdiel, ktorý je ralizovaný ukladaním druhého operandu do vyhľadávacej štruktúry v prvom kroku výpočtu.

je paralelizovaný samostatne. Tým pádom dochádza k dvom separátnym paralelizovaným činnostiam:

1. Uloženie dát do vyhľadávacej štruktúry (hašovacej tabuľky) - spracovanie dát náležaicich k jednému operandu.

2. Vykonanie logiky operácie nad naplnenou hašovacou tabuľkou - spracovanie dát náležiacich k druhému operandu.

Obe zmienené činnosti je však potrebné vykonať v poradí ako sú uvedené. Postup je naznačený v ukážke kódu 4.4, v ktorej je konkrétne znázornený paralelizovaný algoritmus pre databázovú operáciu $S \setminus R$ s množinovým výstupom.

```
CR := split up R to chunks
foreach cr in CR do_parallel
  // Nasleduje pseudokód pre box hash.
  foreach r in cr do
    if r is not in T then
      i := hash(r)
      insert r into T[i]
    fi
  done
  // Koniec boxu hash.
parallel_done

CS := split up S to chunks
foreach cs in CS do_parallel
  // Nasleduje pseudokód pre box realizujúci logiku operácie.
  foreach s in cs do
    i := hash(r)
    if s is not in T[i] then
      insert s into T[i]
      output s
    fi
  done
  // Koniec boxu s logikou operácie.
parallel_done
```

Zdrojový kód 4.4: Pseudokód pre paralelný algoritmus dvoj-krokových databázových oprácií so zdieľanou hašovacou tabuľkou, konkrétne sa jedná o databázovú operáciu $S \setminus R$ s množinovým výstupom.

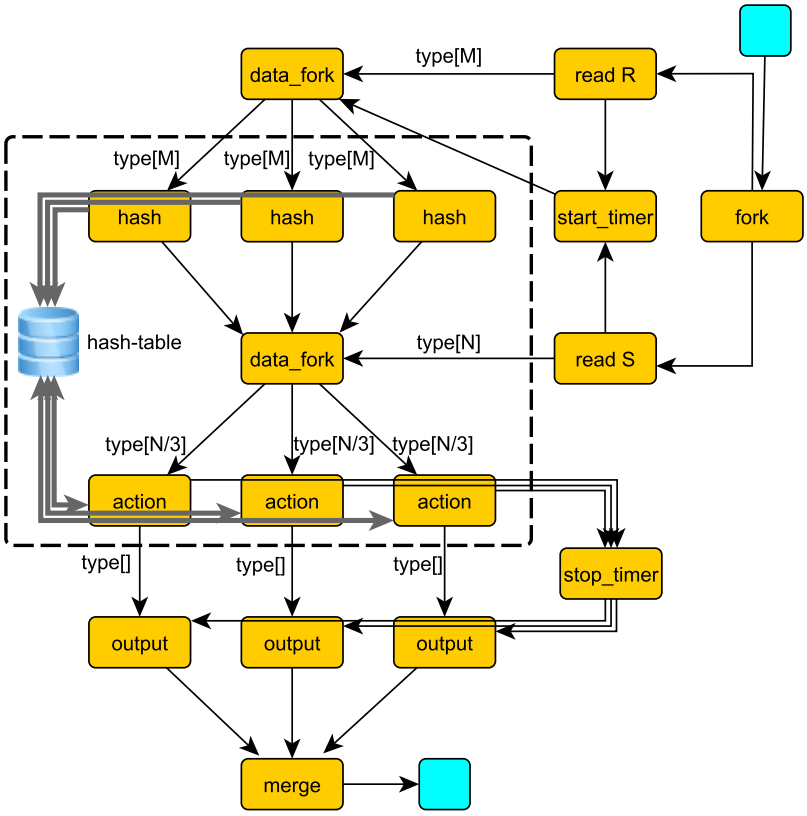
V ukážke kódu sú pomocou komentárov vyznačené aj časti, ktoré zodpovedajú boxom z nákresu modelu 4.7.

Pre operácie prienik, spojenie a rozdiel s opakovaním prvkov je priebeh paralelného algoritmu podobný, ako je v prípade rozdielu s množinovým výstupom (pseudokód 4.4). Sekvenčné algoritmy uvažovaných databázových operácií pozostávajú tiež z dvoch krokov, ktoré zodpovedajú popisu paralelných krokov vyššie. Príslušné paralelné algoritmy vo forme príkladu 4.4 nie je problém zostrojiť na základe ukážok sekvenčných algoritmov z predošlej kapitoly.

Diagram modelu 4.7 pokrýva všetky prevedenia dvoj-krokových algoritmov pre zmienené operácie, tj. množinové prevedenie a aj prevedenie s opakovaním prvkov, samozrejme s výnimkou rozdielu s hašovaním prvého operandu, ktorý je realizovateľný len v troch krokoch.

Pridelenie dát pre jednotlivé kroky dvoj-krokového modelu zodpovedá operandom realizovanej binárnej operácie. Operácie spojenia a prieniku sú komutatívne, takže pridelenie dát jedného operandu prvému kroku, alebo opačne druhému kroku je zameniteľné. Vhodná voľba operandu, resp. dát pre prvý krok je voľba menej mohutných dát, pretože dáta spracovávané v prvom kroku je nutné mať počas celého priebehu výpočtu v pamäti.

Jedine operácia rozdielu nie je komutatívna a teda nie je možné si v rámci dvoj-krokového modelu zamieňať operandy. Ako však bolo zmienené vyššie, operácia rozdielu v dvoj-krokovom prevedení vyžaduje hašovanie dát náležiacich druhému operandu.



Obr. 4.7: Šablóna dvoj-krokového modelu pre operácie spojenia, prieniku a rozdielu. V prípade boxu *hash* dochádza len k zápisom do hašovacej tabuľky (znázornené jednostrannou šípkou v smere toku dát). Boxy typu *action* v niektorých prevedeniach algoritmov nepotrebujú možnosť zápisu do hašovacej tabuľky (znázornená je verzia so zápisom aj čítaním - obojsmerná šípka).

Boxy

Do diagramu 4.7 pribudli nasledovné nové typy boxov:

1. *hash* - hašovanie záznamov do hašovacej tabuľky.
2. *action* - vykonanie logiky operácie.

Box *action* je šablónový box, ktorý je v reálnom modele nahradený za box s konkrétnou implementáciou logiky realizovanej databázovej operácie.

Výpočet

Výpočet v prostredí **Bobox** zodpovedá v pomocných častiach už popísaným postupom. Odlišnosťou je vstup dvoch sád dát, nakoľko sa jedná o realizáciu binárnych operácií. Box *read* R posiela dáta do jedného boxu *data_fork* a *read* S do druhého boxu *data_fork*. Box *data_fork* zodpovedný za delenie dát relácie R je štandardne spustený po príchode signálu, ktorý je iniciovaný spustením časomiery tak, ako v ostatných doposiaľ predstavených výpočtoch. Tým je spustený prvý krok výpočtu.

Narozdiel od prvého, tak druhý *data_fork* zodpovedný za delenie dát relácie S nie je napojený na časomieru, ale na boxy *hash* vykonávajúce prvý krok výpočtu. Pokiaľ boxy *hash* nepošlú signál o skončení svojej práce, tak *data_fork* náležiaci k S neprepošle dáta ďalej. Tak je druhý krok spustený po skončení prvého kroku.

Všetky ostatné aspekty výpočtu sú analogické ako v predošlých popisoch.

4.6.3 Troj-krokový výpočet s dohľadaním výsledku

Troj-krokový výpočet je vlastne práve popísaný dvoj-krokový výpočet doplnený o tretí krok - dohľadanie výsledku vo vyhľadávacej štruktúre. Dohľadanie výsledku je potrebné pre algoritmus rozdielu $R \setminus S$, ktorý využíva na ukladanie do vyhľadávacej štruktúry prvý operand, tj. R. Pre opačné prevedenie hašovania operandu S je postačujúci dvoj-krokový model predstavený v predošlej sekcii 4.6.2.

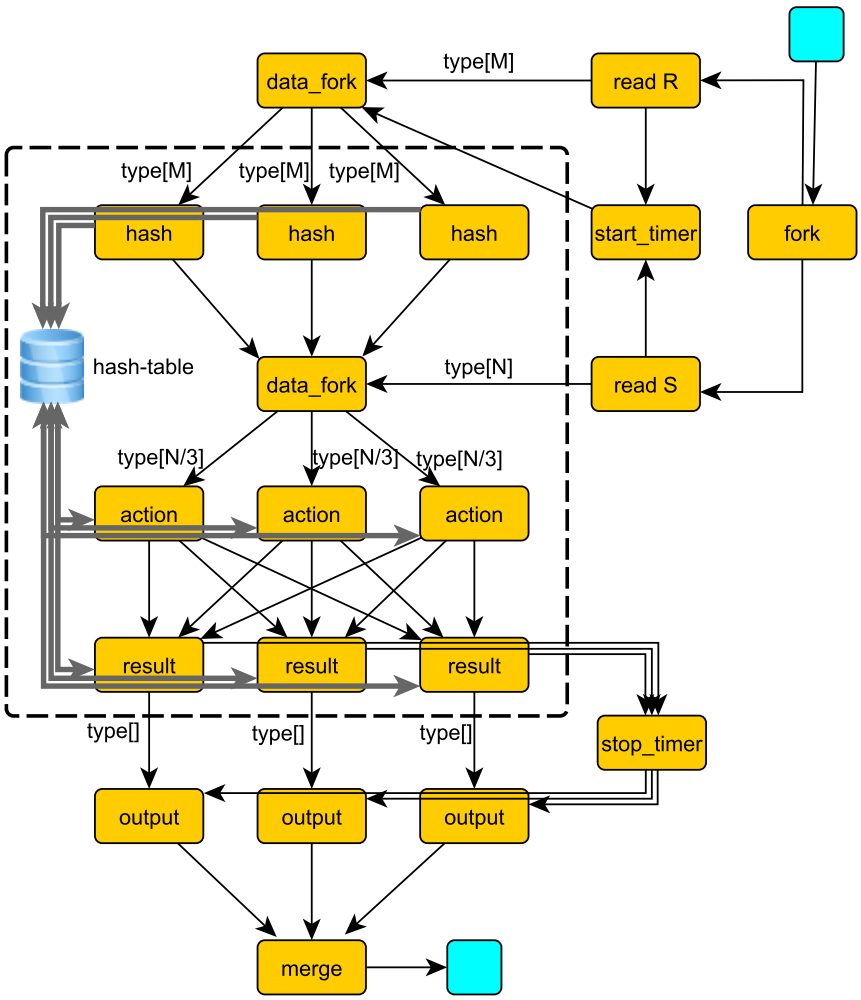
```
CR := split up R to chunks
foreach cr in CR do_parallel
  // Nasleduje pseudokód pre box hash.
  foreach r in cr do
    i := hash(r)
    if r is not in T[i] then
      insert r into T[i]
    fi
  done
  // Koniec boxu hash.
parallel_done

CS := split up S to chunks
foreach cs in CS do_parallel
  // Nasleduje pseudokód pre box realizujúci logiku operácie.
  foreach s in cs do
    i := hash(r)
    if s is in T[i] then
      remove s from T[i]
    fi
  done
  // Koniec boxu s logikou operácie.
parallel_done
```

```
CT := split up T to chunks
foreach ct in CT do_parallel
  // Pseudokód detekcie výsledkov.
  foreach x in ct do
    output x
  done
  // Koniec detekcie výsledkov.
parallel_done
```

Zdrojový kód 4.5: Pseudokód pre paralelný algoritmus troj-krokových databázových oprácií so zdieľanou hašovacou tabuľkou, konkrétne sa jedná o databázovú operáciu $R \setminus S$ s množinovým výstupom.

Paralelný algoritmus reprezentovaný pseudokódom 4.5 znázorňuje operáciu $R \setminus S$ s množinovým výstupom. Pre výstup s opakovaním prvkov je možné algoritmus jednoducho upraviť na základe sekvenčného algoritmu (kapitola 3.3.7), pretože postup paralelizácie je totožný.



Obr. 4.8: Diagram modelu pre operáciu rozdielu s hašovaním dát prvého operandu.

Boxy

Do diagramu modelu pribudol jeden nový box:

1. *result* - detekcia výskytu záznamu v hašovacej tabuľke a jeho odoslanie na výstup.

Výpočet

Výpočet určený diagramom modelu 4.8 je veľmi podobný dvoj-krokovému výpočtu (diagram 4.7). Dá sa povedať, že sa jedná len o rozšírenie o ďalší krok. Z tohoto dôvodu je postačujúce popísať len odlišné časti realizácie - tretí krok.

1. Boxy *action* neposielajú výsledky na výstup (nakoľko ich ešte nepoznajú), ale len zapisujú zmeny do hašovacej tabuľky.

2. Po skončení práce boxy *action* pošlú signál do všetkých boxov *result*.

3. Boxy *result* prehľadajú hašovaciu tabuľku, po ukončení práce všetkých boxov *action*. Každý z boxov typu *result* má určenú[11] svoju časť tabuľky, v ktorej vyhľadáva výsledky. Každý záznam, ktorý sa vyskytne v hašovacej tabuľke, je prenesený na výstup.

4. Po ukončení prehľadávania hašovacej tabuľky boxy *result* pošlú signál, ktorý determinuje ukončenie merania času. Výpočet ďalej pokračuje, resp. končí rovnako ako v predošlých popisoch.

4.6.4 Výpočty s predspracovaním dát

Pri paralelizovaní algoritmov pre niektoré databázové operácie bola vo výpočtoch, ktoré sú popísané v predošlom texte, použitá zdieľaná hašovacia tabuľka, nakoľko sa táto možnosť javila ako jediná cesta paralelizácie jedno-priechodových sekvenčných algoritmov.

V tejto časti bude, pomocou princípov predspracovania dát, nutnosť zdieľania hašovacej tatbuľky odstránená. Predpracovanie dát pre realizáciu databázových operácií je predstavené v kapitole 3.4, kde sú uvedené dve formy predspracovania (triedením a hašovaním).

Predspracovanie dát v úprave pre jedno-priechodové algoritmy je realizované na princípoch hašovania. Vstupné dáta sú rozdelené do skupín, ktorých počet je rovnaký ako stupeň vetvenia výpočtu. Delenie dát zaručuje to, že výpočet realizovaný nad jednou skupinou dát nepotrebuje žiadne informácie o dátach v iných skupinách. Takže výpočet nad každou skupinou prebehne absolútne nezávisle od výpočtu ostatných skupín. Tento postup nápadne pripomína prístup k spracovaniu dát v distribuovaných systémoch, označovaný ako *partioning* (viac informácií viď [16]).

[11] Určenie oblasti prebieha už v definícii boxu v modele výpočtu tak, že každý box *result* má určené poradové číslo svojho úseku a informáciu o počte úsekov. Veľkosť tabuľky si boxy zistia za behu, a tak si dopočítajú na základe dostupných faktov úsek tabuľky, ktorý majú prehľadať.

Proces delenia prebieha pomocou aplikácie masky. Ako maska je zvolený typ *int*, ktorý má na cieľovej platforme štyri bajty. Avšak pre potreby masky sa využíva len päť prvých menej významných bitov[12].

```
GR := create N groups
CR := split up R to chunks
GS := create N groups
CS := split up S to chunks
foreach (G,C) in [(GR, CR),(GB,CB)] do_parallel
  foreach c in C do_parallel
    // Pseudokód predspracovania - delenia dát do skupín.
      foreach g in G do
        m := get mask of group g
        if m & r then
          insert r into g
          continue
        fi
      done
    done
    // Koniec predspracovania.
  parallel_done
parallel_done


for i := 1 to N do_parallel
  // Nasleduje pseudokód pre box realizujúci logiku operácie.
  foreach r in GR[i] do
    i := hash(r)
    if s is not in T[i] then
      insert s into T[i]
      output s
    fi
  done

  foreach s in GS[i] do
    i := hash(s)
    if s is not in T[i] then
      insert s into T[i]
      output s
    fi
  done
  // Koniec boxu s logikou operácie.
parallel_done
```

Zdrojový kód 4.6: Pseudokód pre paralelný algoritmus databázovej operácie zjednotenia s množinovým výstupom, ktorý je realizovaný pomocou predspracovania dát.

Algoritmus naznačený v ukážke kódu 4.6 (v ktorom sú komentármi vyznačené pseudokódy ďalej popísaných boxov) prebieha v dvoch fázach[13]. Priebeh algo-

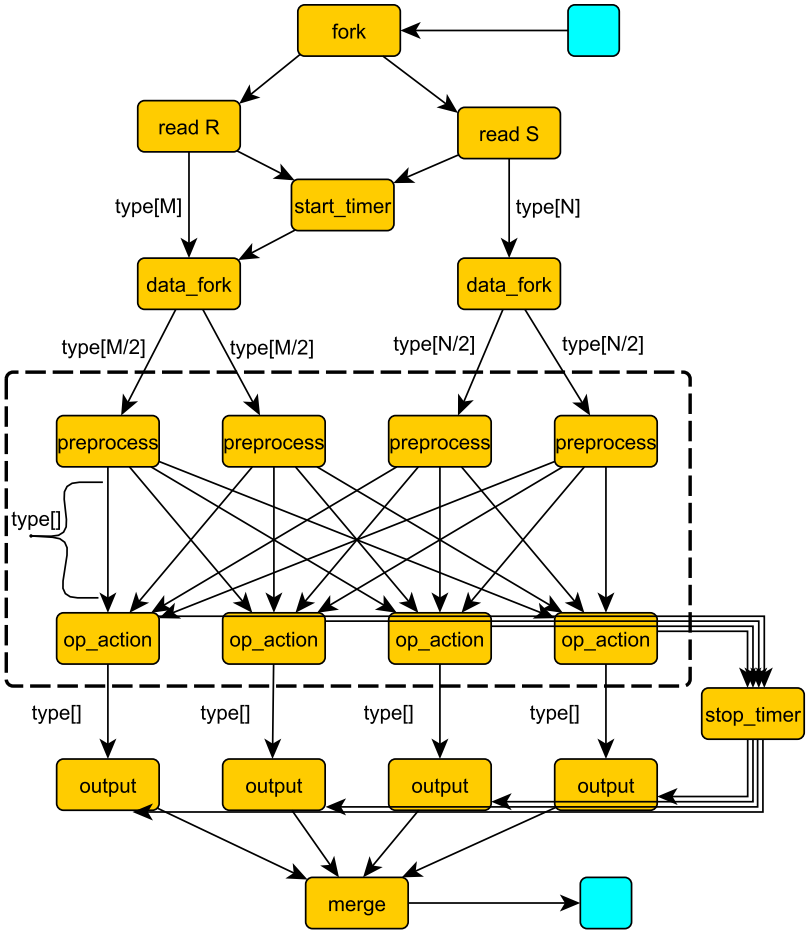
[12]Použitie šiestich bitov na masku plynie z potreby maximálne 32 skupín v sprievodnom programe. Využitie všetkých 32 dostupných bitov pre masku je možné a dosiahne sa jednoduchým nastavením konštanty v zdrojovom kóde.

[13]Nezamieňať si fázy s krokmi.

ritmu s predspravovaním dát je znázornený pre binárne databázové operácie[14], avšak úprava pre unárne operácie je triviálna a nebude preto v texte explicitne uvedená.

V prvej fáze algoritmu dochádza k deleniu vstupných dát do skupín na základe aplikácie masky. Prezentovaný pseudokód 4.6 vytvára dojem, že je potrebné synchronizovať zápis delených dát do skupín. Pri realizácii v **Boboxe** je kód delenia vykonávaný separátnymi boxami, ktoré si držia skupiny delenia lokálne a dáta posielajú následne ďalej k spracovaniu pomocou štandardných rutín v **Boboxe**, takže synchronizácia nie je potrebná.

V druhej fáze dochádza k vykonaniu logiky databázovej operácie nad konkrétnou skupinou dát. Vďaka povahe rozdelených dát je možné na každú skupinu aplikovať separátne sekvenčný algoritmus. Takže z pseudokódu 4.6 je možné jednoduchou zámenou časti pre logiku operácie za iný sekvenčný algoritmus vytvoriť pseudokód pre inú databázovú operáciu.



Obr. 4.9: Diagram modelu realizujúci binárnu databázovú operáciu pomocou predspracovania dát.

[14]Realizácia bola uskutočnená tiež len pre binárne databázové operácie.

Boxy

V diagrame 4.9 sa objavujú nové typy boxov:

1. *preprocess* - rozdelenie dát do skupín.
2. *op_action* - vykonanie databázovej operácie vcelku pomocou sekvenčného algoritmu.

Box *op_action* je šablónový box, ktorý je v reálnom modele nahradený za box s konkrétnou implementáciou celého sekvenčného algoritmu realizovanej databázovej operácie. V predošlých realizáciách boli databázové operácie implementované viacerými boxami, z dôvodu synchronizácie krokov pri viac-krokových výpočtoch. Realizácia databázových operácií za pomoci predspracovania dát nepotrebuje synchronizovať kroky výpočtu viacerých boxov, nakoľko sa jedná o separátnu realizáciu v každom boxe. Dokonca je rozdelenie nežiaduce, pretože v prípade viacerých boxov by bolo nutné riešiť zdieľanie hašovacej tabuľky viacerými boxami.

Výpočet

Výpočet predspracovaním prebieha, ako bolo popísané, v dvoch fázach. Prvá fáza je realizovaná boxami *preprocess*. Boxy *preprocess* sú rozdelené na skupinu boxov, ktoré spracovávajú dáta prvého operandu a skupinu boxov určených pre spracovanie dát druhého operandu. Doposiaľ boli všetky realizácie databázových operácií koncipované tak, aby používaný stupeň rozvetvenia výpočtu bol vo všetkých častiach výpočtu rovnaký a toto pravidlo je dodržané aj pre výpočet s predspracovaním dát. Keďže predspracovanie dát oboch operandov je realizované súčasne, tak počet všetkých použitých boxov *preprocess* je rovný stupňu vetvenia výpočtu.

Každý box *preprocess* rozďeľuje dáta do skupín, ktorých počet je určený stupňom vetvenia výpočtu. Každá skupina zodpovedá jednému boxu *op_action*. Boxy *op_action* najskôr spracujú vstupy z boxov *preprocess*, ktoré predspracovávajú dáta prvého operandu. Následne sú spracované aj dáta reprezentujúce druhý operand. Postup spracovania teda zodpovedá sekvenčným algoritmom jednotlivých databázových operácií.

Kapitola 5

Analýza implementovaných databázových operácií

Nasledujúca kapitola sa sústredí na popísanie vlastností implementovaných databázových operácií v prostredí **Bobox** na základe uskutočnených meraní. Skúmané sú možnosti rôznych stupňov vetvenia výpočtov jednotlivých implementácií. Záujem je hlavne orientovaný na analýzu možností zrýchlenia výpočtu.

5.1 Prostriedky a zdroje

Experimentálne merania prebiehali na stroji s 64-bitovým operačným systémom Windows 8 Pro. Počítač disponoval štyrmi fyzickými jadrami Intel(R) Core(TM) i7-3770 o frekvencii 3,4GHz. Naviac zmienený procesor podporuje technológiu Hyper-Threading, čo znamená, že testovací systém má k dispozícii osem logických jadier.

Pre uskutočnenie meraní boli použité dve vygenerované sady dát, ktoré boli obdržané ako výstup zo štatistického software R vo forme CSV súborov. Vygenerované dáta majú rovnomerné rozdelenie a sú tvorené jedným miliónom záznamov. Dáta sú celé čísla v rozsahu od -2^{31} (tj. -2147483648) po $2^{31} - 1$ (tj. 2147483647), čo zodpovedá rozsahu typu *int* na zvolenej platforme, nakoľko program je preložený ako 32 bitový.

Pri meraní unárnych databázových operácií bola použitá ako vstup jedna sada dát a pre binárne operácie obe sady dát, jedna pre jeden operand a druhá pre druhý operand.

5.2 Hašovacia tabuľka

Veľký vplyv na vlastnosti algoritmov má veľkosť použitej hašovacej tabuľky, resp. presnejšie povedané, záleží na faktore naplnenia. Pokiaľ by veľkosť spracovávaných dát značne presahovala veľkosť tabuľky, tak by dochádzalo k neustálym konfliktom pri ukladaní dát, čo by vyžadovalo reťazenie konfliktných záznamov, a to by spôsobilo značné spomalenie. Tento problém však nie je podmienený iba uvedenou podmienkou, ale môže nastávať aj pri dátach, ktoré sa

pohodlne zmestia do hašovacej tabuľky.

Dôležitú úlohu v popísanom probléme zohráva hašovacia funkcia, ktorá distribuuje záznamy po tabuľke. Pre potreby implementácie bola použitá hašovacia funkcia z knižníc Boost, ktorá vykazuje dobré vlastnosti pri hašovaní testovacích dát.

Pre kontrolu rozloženia dát v hašovacej tabuľke bola vykonaná analýza. Výstupom boli rôzne pohľady na naplnenie tabuľky, ktorých definície sú ďalej uvedené.

Definícia 6. *(faktor naplnenia)*
Nech s je počet riadkov hašovacej tabuľky T a nech n je počet všetkých uložených záznamov. Potom faktor naplnenia $f(T)$ je vyjadrený ako $f(T) = n/s$.

Definícia 7. *(faktor naplnenia bez duplicít)*
Nech s je počet riadkov hašovacej tabuľky T a nech n_u je počet všetkých uložených záznamov bez duplicít, tj. opakované výskyty záznamov v hašovacej tabuľke sa do počtu n_u nezapočítavajú. Potom faktor naplenia bez duplicít $f_u(T)$ je vyjadrený ako $f_u(T) = n_u/s$.

Definícia 8. *(faktor naplnenia podľa hašovacej funkcie)*
Nech s je počet riadkov hašovacej tabuľky T a nech n_h je počet všetkých záznamov zahašovaných na unikátnu pozíciu, tj. dva záznamy (duplicitné alebo rôzne), ktoré sú zahašované na rovnakú pozíciu v tabuľke, sú započítané do počtu len raz. Potom faktor naplnenia podľa hašovacej funkcie $f_h(T)$ je vyjadrený nasledovne $f_h(T) = n_h/s$.

Pre uvedené faktory naplnenia vždy platí nasledújúca nerovnosť:

$$f(T) \geq f_u(T) \geq f_h(T)$$

Faktor naplnenia $f(T)$ hašovacej tabuľky je možné ľahko dopočítať z veľkosti dát a samotnej tabuľky. Veľkosť dát je známa (1 milión záznamov) a veľkosť tabuľky je konfigurovateľná. Keďže záujem je kladený hlavne na analýzu vlastností paralelizácie a nie na študovanie vplyvu naplnenia tabuľky na výkon algoritmov, tak je zvolená veľkosť tabuľky 3 milióny riadkov. Hodnota $f(T)$ je potom rovná 1/3.

Zvyšné dva typy faktorov naplnenia hašovacej tabuľky $f_u(T)$ a $f_h(T)$ sú závislé na vlastnostiach hašovacej funkcie. Hodnoty, ktoré boli namerané pre zahašovanú testovaciu sadu dát sú následovné:

- $f_u(T) = 0.333255$
- $f_h(T) = 0.283566$

Zo všetkých doposiaľ uvedených hodnôt je možné pomocou jednoduchej trojčlenky vyvodiť dva závery:

1. Z faktu, že $f_u(T) = 0.333255$ je možné dopočítať, že v dátach sa vyskytuje $0,0235\%$ duplicitných záznamov. Táto informácia nie je prekvapujúca v kontexte s informáciami o generovaní dát.

2. Z druhého merania ($f_h(T) = 0.283566$) je zrejmé, že v $14,93\%$ prípadov dôjde ku kolízii záznamov v hašovacej tabuľke.

5.3 Realizácia experimentov

Pre každý testovaný model boli vykonané merania pre šesť rôznych prevedení rozvetvenia výpočtu. Rozvetvenie výpočtu určuje stupeň možnej paralelizácie, tj. rozvetvenie do jednej vetvy zodpovedá sekvenčnému prevedeniu a rozvetvenie do štyroch vetiev je nutnou podmienkou pre paralelizáciu stupňa štyri (nie však postačujúcou podmienkou). Framework má paralelizáciu v svojej réžii a zaleží len na konfigurácii behového prostredia ako bude s možnosťami paralelizácie naložené.

Testované rozvetvenia sú 1, 2, 4, 8, 16 a 32 vetiev, pričom **Bobox** nie je limitovaný žiadnym obmedzujúcim nastavením a prostredie môže využiť všetkých osem logických jadier, ako uzná za vhodné. Na prvý pohľad sa to môže zdať pre prípady nižších rozvetvení zbytočné. Napríklad rozvetvenie výpočtu v modele do dvoch vetiev neumožňuje väčšiu paralelizáciu ako stupňa dva, ale **Bobox** môže využiť dostupné jadrá pre interné potreby, napr. plánovanie použitia boxov, prenos dát a podobne.

Pre každý stupeň vetvenia výpočtu bolo vykonaných päťdesiat meraní, z ktorých boli následne odstránené odľahlé pozorovania. Zo zvyšných hodnôt boli dopočítané priemerné časy, ktoré sú znázornené v grafoch a tabuľkách v ďalšom texte. Ďalej sú dopočítané absolútne zrýchlenia (tj. zrýchlenie oproti sekvenčnému prevedeniu) a relatívne zrýchlenia (tj. zrýchlenie z predchádzajúceho stupňa rozvetvenia výpočtu).

V grafoch sú absolútne aj relatívne zrýchlenia zobrazované v percentách. Pri absolútnom zrýchlení je braný výsledok sekvenčného prevedenia ako 100% a zrýchlenie sa vyjadruje vzhľadom k tejto hodnote, tj. pokiaľ je výpočet jeden a pol krát rýchlejší ako sekvenčná varianta potom je znázornený ako 150% (100% základ plus 50% zrýchlenie). Relatívne zrýchlenie vyjadruje prírastok medzi dvomi bezprostredne susediacimi vetveniami, tj. $1 \rightarrow 2, 2 \rightarrow 4, 4 \rightarrow 8, 8 \rightarrow 16$ a $16 \rightarrow 32$. Takže pokiaľ zrýchli výpočet pri zmene vetvenia $1,8$ krát tak je vyjadrený len čistý prírastok, tj. 80%. Všetky zrýchlenia sú nakoniec zhrnuté v prehľadnej tabuľke, kde sú však znázornené ako číselné koeficienty dopočítané na základe referenčného (základného) času t_b a času referovaného (zmeneného) t_c, ako podiel t_b/t_c. Koeficient teda presne vyhovuje intuitívnej predstave zrýchlenia, napríklad zrýchliť dva krát zodpovedá koeficientu dva.

5.4 Vyhodnotenie výsledkov

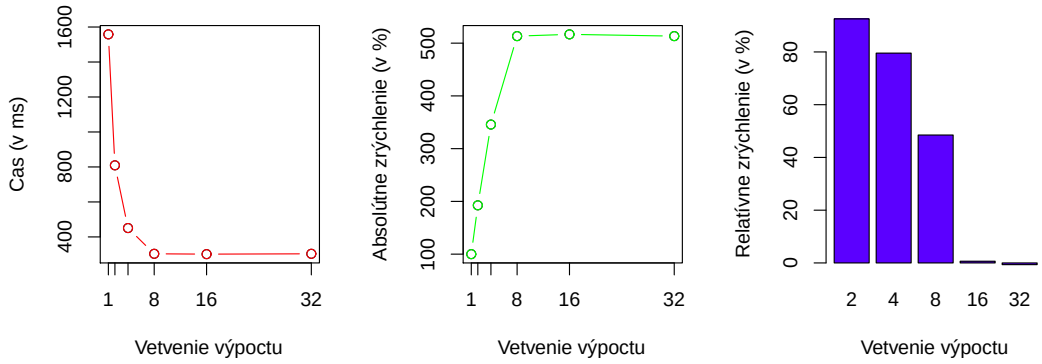
Žiadne z vykonaných meraní nevykázalo signifikantné odlišnosti od ostatných. Z tohto dôvodu je v tejto kapitole prezentovaná len analýza databázovej operácie spojenia a uvedené úvahy sú všeobecne platné pre všetky uskutočnené merania. Výstupy ostatných meraní sú dohľadatelné v prílohe A.

Okrem prevedenia analýzy jednotlivých realizácií databázových operácií je zaujímavá aj otázka porovnania realizácií so zdieľanou hašovacou tabuľkou a relizácií s predspracovaním dát (s lokálnou hašovacou tabuľkou). Databázová operácia spojenia je realizovaná oboma spôsobmi, a teda vyhovie ako vzorová analýza pre ostatné databázové operácie aj v tomto prípade.

5.4.1 Výpočty so zdieľanou hašovacou tabuľkou

Merania preukazujú prínos paralelizácie pri vykonávaní databázových operácií. Skrátenie doby výpočtu je markantné a je znázornené v prvom grafe na obrázku A.8. Značne prínosné sa ukazujú paralelizácia výpočtov so stupňom vetvenia dva a štyri, ktoré sa blížia optimálnemu zrýchleniu, tj. zdvojnásobenie vetiev výpočtu by optimálne malo priniesť dvojnásobné zrýchlenie.

Pri následných stupňoch vetvenia už k tak zásadnému zisku nedochádza, čo je však logické, nakoľko systém má len štyri fyzické jadrá. Technológia Hyper Threading však umožňuje spúšťať dve úlohy súčasne na jednom fyzickom jadre, nedosahuje však výkonu ako spustenie dvoch úloh na dvoch jadrách (podrobnosti viď [19] a [20]). Zmienenému faktu zodpovedá aj zrýchlenie výpočtu pre rozdelenie výpočtu do ôsmich vetiev, ktoré dosahuje necelých 50%. Pre zvyšné dve vetvenia už nie je dosiahnuté žiadne badateľné zrýchlenie. To však je očakávané vzhľadom na počet jadier (či už fyzických alebo logických) v testovacom systéme. Je však možné sa na vlastnosti vetvenia do šestnásť a tridsaťdva vetiev pozrieť z opačnej strany, a to, že pri ich behu nedochádza k badateľnému spomaleniu. Pri navyšovaní počtu vetiev výpočtu by mohlo dochádzať k väčšej réžii výpočtu a pri nezískaní zrýchlenia z väčšieho paralelizmu by sa mal tento fakt prejaviť negatívne v koncovom čase. Merania však ukazujú, že k zásadnému zníženiu výkonu nedochádza.



Obr. 5.1: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

Vetvenie výpočtu	Priemerný čas	Počet meraní
1	1558.08	49
2	809.27	41
4	450.71	48
8	303.51	49
16	301.55	49
32	303.51	49

Tabuľka 5.1: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.93	3.46	5.13	5.17	5.13	1.80	1.48	1.01	0.99

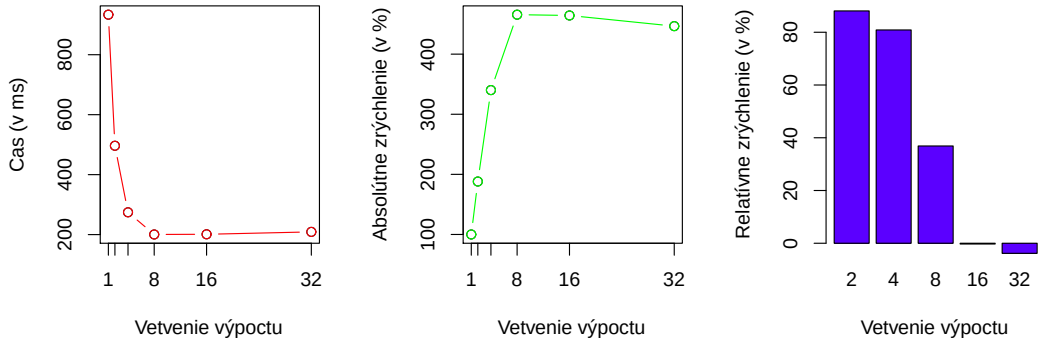
Tabuľka 5.2: Zrýchlenie pri zmene stupňa vetvenia.

5.4.2 Výpočty s predspracovaním dát

Pre merania vykonané na realizáciách databázových operácií, ktoré sú implementované s pomocou predspracovania dát (resp. s lokálnou hašovacou tabuľkou), platí všetko, čo bolo napísané o ostatných meraniach.

Jediná drobná odlišnosť je pokles výkonu pri vetvení stupňa tridsaťdva, ten je však tiež pomerne malý (cca. 4 %). Dôvod, pre ktorý dochádza k poklesu výkonu, sa zatiaľ nepodarilo objasniť. Podozrenie padá na časť predspracovania dát a možnú neefektívnosť v implementácii. Pre podloženie či vyvrátenie tohto tvrdenia je však potrebné vykonať ďalšie kroky.

Zaujímavé je porovnanie zodpovedajúcich si prevedení, kde sú jednoznačne výkonnejšie prevedenia s predspracovaním dát.



Obr. 5.2: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

Vetvenie výpočtu	Priemerný čas	Počet meraní
1	933.54	50
2	496.33	49
4	274.41	49
8	200.47	38
16	201.05	41
32	209.04	50

Tabuľka 5.3: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.88	3.40	4.66	4.64	4.47	1.81	1.37	1.00	0.96

Tabuľka 5.4: Zrýchlenie pri zmene stupňa vetvenia.

5.4.3 Zdieľaná hašovacia tabuľka vs. lokálna hašovacia tabuľka

Napriek potrebe predspracovania dát sú realizácie databázových operácií pomocou lokálnej hašovacej tabuľky výkonnejšie pri všetkých meraných stupňoch rozvetvenia výpočtu. Rozdiel vo výkonnosti je značný, spotreba času pre totožný výpočet sa pohybuje v rozmedzí približne o 40 % − 70 % času naviac, kde ako 100 % je braný výsledok rýchlejšieho prevedenia databázovej operácie. Konkrétne rozdiely pre databázovú operáciu spojenia sú uvedené v tabuľke 5.5. Najmarkantnejší rozdiel je medzi sekvenčnými realizáciami a postupne sa zmenšuje.

Vetvenie výpočtu	zdieľaná haš.t.	lokálna haš.t. haš.t.	percentuálny rozdiel v rýchlosti výpočtu
1	1558.08	933.54	66,90 %
2	809.27	496.33	63,05 %
4	450.71	274.41	64,24 %
8	303.51	200.47	51,39 %
16	301.55	201.05	49,98 %
32	303.51	209.04	45,19 %

Tabuľka 5.5: Porovnaie času potrebného na výpočet databázovej operácie pomocou zdieľanej a lokálnej hašovacej tabuľky. Percentuálny rozdiel je vyjadrenie podielu času, ktorý je potrebný naviac pre výpočet so zdieľanou hašovacou tabuľkou, pričom čas potrebný pre výpočet s lokálnou tabuľkou je braný ako 100%.

Dôvod výkonnostného rozdielu v neprospech realizácií so zdieľanou hašovacou tabuľkou je pomerne prozaický, a to samotná zdieľaná hašovacia tabuľka. Tabuľku je totiž potrebné synchronizovať. Spomalenie však nie je spôsobené čakaním na prístup k tabuľke, pretože čakanie je nutné len pri súčasnom konfliktnom hašovaní (viď 4.6) a konfliktných dát je len cca. 15 %. Spomalenie je vytvorené len samotnou réžiou zamknutia a odomknutia zámku, ktorým je riadený prístup k synchronizovaným dátam.

Presnejšie povedané, problém je vytvorený častým prístupom k zámku z rôznych vlákien, čo spôsobuje premiestňovanie dát z cache jedného procesoru do cache druhého. Realizácia premiestňovania dát medzi jednotlivými chache môže byť vykonávaná dokonca pomocou hlavnej pamäte.

Pre podloženie tohto tvrdenia bola synchronizácia natvrdo vypnutá a meranie času bolo vykonané na sekvenčnom prevedení algoritmu so zdieľanou tabuľkou, kde ostatne nie je synchronizácia ani potrebná[1]. Vykonané merania potvrdili domnienku, nakoľko priemerný čas výpočtu sa znížil z $1558.08ms$ na $806,24ms$, čo už je porovnateľný čas, keď uvážime, že sekvenčné prevedenie s lokálnou hašovacou tabuľkou vykonáva naviac priechod dátami pri predspracovaní[2].

[1]Povaha implementácie jednotlivých boxov je však taká, že boxy o svojom prostredí nič nevedia, a teda samé nemôžu riadiť zamykanie.

[2]Predspracovanie dát pri sekvenčnom prevedení je tiež zbytočné, keďže dáta sa delia len do jednej skupiny. Dôvod je však opäť realizácia boxov, ktoré nevedia nič o svojom okolí a kontexte výpočtu.

Je teda zrejmé, že úzkym hrdlom výpočtov so zdieľanou hašovacou tabuľkou je použitie zámkov. Nezodpovedanou otázkou zostáva, či existuje efektívnejšia možnosť synchronizácie prístupu k zdieľanej premennej ako využíva použitá verzia prostredia **Bobox**, pretože pri synchronizácii prístupu k hašovacej tabuľke bol použitý zámok z **Boboxu**.

Kapitola 6

Záver

Cieľom diplomovej práce bolo preskúmať možnosti paralelizácie databázových operácií a vybrané operácie implementovať v paralelnom prostredí **Bobox**. Ďalším zámerom bolo implementovanú paralelnú verziu porovnať so sekvenčným prevedením a posúdiť limity škálovateľnosti zvoleného riešenia.

V úvodnej časti práce sú predstavené sekvenčné algoritmy databázových operácií, ktoré poskytujú základné východiská neskoršej paralelizácie. Teoretická časť poskytuje dostatok znalostí potrebných pre samotnú implementáciu databázových operácií.

Následne sú v diplomovej práci popísané postupy paralelizácie výpočtov databázových operácií, ktoré sú prezentované už v kontexte prostredia **Bobox**. Uvedené postupy nie sú využiteľné len programami, ktoré využívajú paralelné prostredie **Bobox**, ale jedná sa všeobecne použiteľné návody.

Záverečná časť práce obsahuje analýzu meraní výkonu a zhodnotenie škálovateľnosti paralelizácie implementovaných databázových operácií. Získané merania ukazujú na slušné vlastnosti prevedených realizácií. Zaujímavé je hlavne porovnanie dvoch odlišných prístupov k realizácii niektorých databázových operácií[1]. Rozdiel vo výkone bol kvantifikovaný na základe meraní. Vykonané merania ukazujú očakávaný výsledok, a to, že implementácia pomocou zdieľanej hašovacej tabuľky vykazuje značne slabší výkon, ako prístup, ktorý je realizovaný pomocou predspracovania dát (lokálna hašovacia tabuľka).

Predložená diplomová práca naplnila stanovené ciele a otvorila aj ďalšie otázky. Vhodné by bolo podrobne preskúmať spôsob a možnosti synchronizácie pri algoritmoch so zdieľanou hašovacou tabuľkou, či dokonca implementovať vlastný prístup. Veľmi prospešné by bolo opakovane vykonať merania na systéme, ktorý by umožnil masívnejšiu paralelizáciu. Z pohľadu implementácie by bolo vhodné doplniť ďalšie realizácie databázových operácií a rozhodne tiež venovať nejaký čas optimalizácii už realizovaných operácií.

[1] Konkrétne sa jedná o operácie spojenia, prieniku a rozdielu.

Zoznam použitej literatúry

[1] Bednárek D.: Output Driven XQuery Evaluation, in 2nd International Symposium on Intelligent Distributed Computing, Springer-Verlag, ISBN: 978-3-540-85256-8, ISSN: 1860-949X, pp. 55-64, September 2008

[2] Falt Z., Bednárek D., Čermák M., Zavoral F., : On Parallel Evaluation of SPARQL Queries, in The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications DBKDA 2012, Saint Gilles, Reunion Island, Xpert Publishing Services, ISBN: 978-1-61208-185-4, pp. 97-102, 2012

[3] Bednárek D., Dokulil J., Yaghob J., Zavoral F., Čermák M., Falt Z., Grafnetter M., Kruliš M.: http://www.ksi.mff.cuni.cz/bobox/

[4] Bednárek D., Dokulil J., Yaghob J., Zavoral F.: The Bobox Project - A Parallel Native Repository for Semi-structured Data and the Semantic Web, in ITAT 2009 - IX. Informacné technológie - aplikácie a teória, PONT Slovakia, ISBN: 978-80-970179-1-0, pp. 44-59, September 2009

[5] Falt Z., Yaghob J.: Task Scheduling in Data Stream Processing, in Dateso 2011, VSB - Technical University of Ostrava FEECS, Department of Computer ScienceI, ISBN: 978-80-248-2391-1, pp. 85-96, 2011

[6] Chaudhuri S.: An overview of query optimization in relational systems, Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems, 1998

[7] Graefe G.: Query evaluation techniques for large databases, Computing Surveys, 1993

[8] Pokorný J.: Dotazovací Jazyky, Karolinum, ISBN: 978-80-246-0497-8, 2007

[9] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: Database systems - the complete book (international edition), Pearson Education, ISBN: 978-0-13-098043-4, pp. I-XXVII, 1-1119, 2002

[10] Sciore E.: Database Design and Implementation, Wiley, ISBN: 978-0471757160, 2008

[11] DeWitt D. J., Katz R. H., Olken F., Shapiro L. D., Stonebraker M., Wood D.: Implementation techniques for main-memory database systems, Proc. ACM SIGMOD Intl. Conf. Management of Data, 1984

[12] Gotlieb L. R.: Computing joins of relations, Proc. ACM SIGMOND Intl. Conf. on Management of Data, 1975

[13] Blasgen M. W., Eswaran K. P.: Storage access in relational databases, IBM Systems, 1977

[14] Chou H. T., DeWitt D. J.: An evaluation of buffer management strategies for relational database systems, Proc. Intl. Conf. on Very Large Databases, 1985

[15] Falt Z., Bulánek J., Yaghob J.: On Parallel Sorting of Data Streams, in Advances in Databases and Information Systems, Poznan, Poland, Springer, ISBN: 978-3-642-32740-7, ISSN: 2194-5357, pp. 69-77, 2012

[16] Taniar D., Leung C. H. C., Rahayu W., Goel S.: High Performance Parallel Database Processing and Grid Databases, Wiley, ISBN: 978-0470107621, 2008

[17] Kossman D.: The state of the art in distributed query processing, Computing Surveys, 2000

[18] Kitsuregawa M., Tanaka H., Moto-oka T.: Application of hash to data base machine and its architecture, New generation computing, 1983

[19] Shawn Casey: http://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application/

[20] Garrett Drysdale, Antonio C. Valles, Matt Gillespie: http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/

Zoznam obrázkov

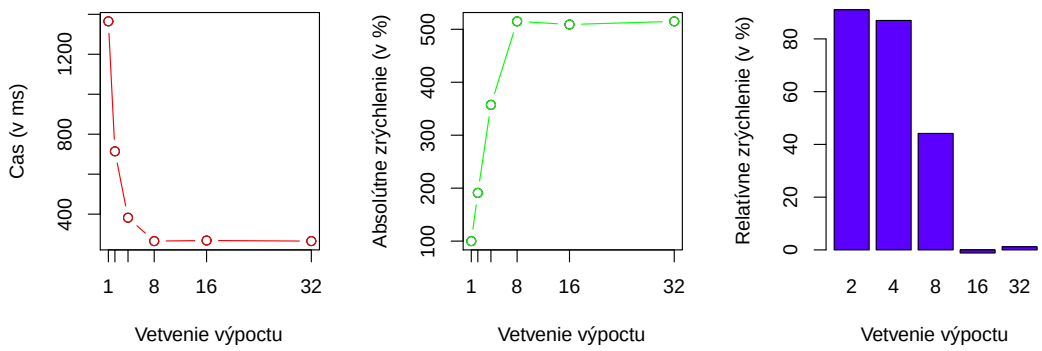
Zoznam zdrojových kódov

Zoznam tabuliek

Dodatok A

Merania vlastností jednotlivých realizácií

A.1 Eliminácia duplicít (zdieľaná haš. tab.)



Obr. A.1: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

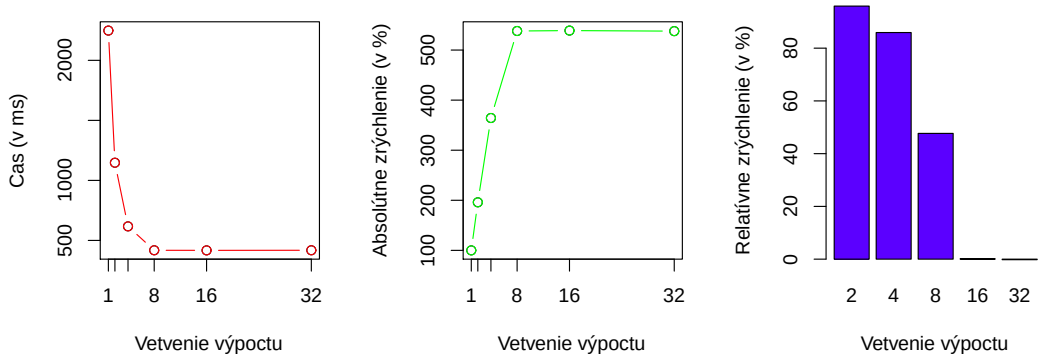
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	1364.83	47
2	714.35	48
4	382.02	43
8	265.00	40
16	268.13	46
32	265.02	40

Tabuľka A.1: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.91	3.57	5.15	5.09	5.15	1.87	1.44	0.99	1.01

Tabuľka A.2: Zrýchlenie pri zmene stupňa vetvenia.

A.2 Prienik (zdieľaná haš. t., bag)



Obr. A.2: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

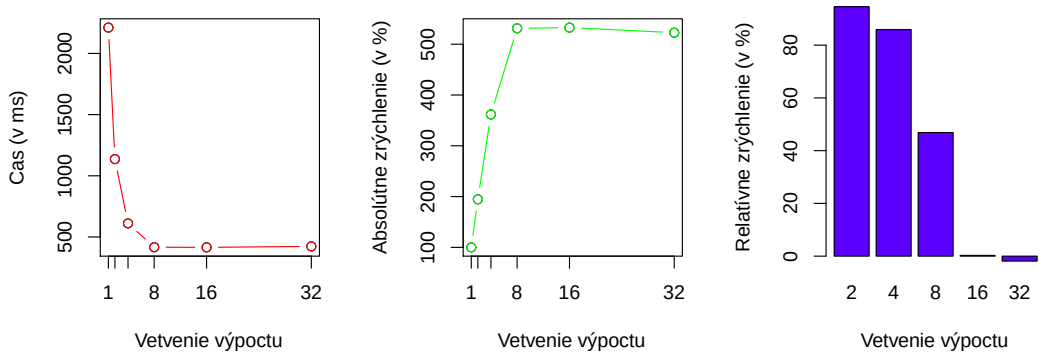
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	2247.92	48
2	1147.19	42
4	617.02	45
8	417.81	47
16	417.15	46
32	418.02	49

Tabuľka A.3: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.96	3.64	5.38	5.39	5.38	1.86	1.48	1.00	1.00

Tabuľka A.4: Zrýchlenie pri zmene stupňa vetvenia.

A.3 Prienik (zdieľaná haš. t., set)



Obr. A.3: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

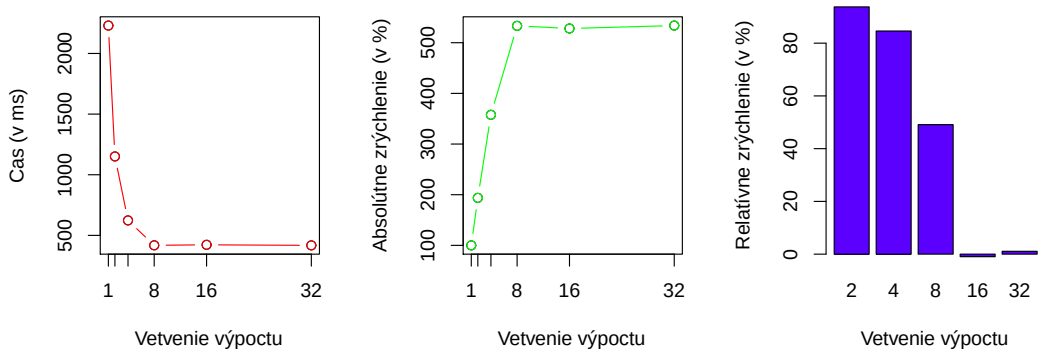
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	2211.37	49
2	1136.62	40
4	611.32	47
8	416.24	50
16	415.08	48
32	423.04	45

Tabuľka A.5: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.95	3.62	5.31	5.33	5.23	1.86	1.47	1.00	0.98

Tabuľka A.6: Zrýchlenie pri zmene stupňa vetvenia.

A.4 Rozdiel (zdieľaná haš. t., bag, 2-step)



Obr. A.4: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

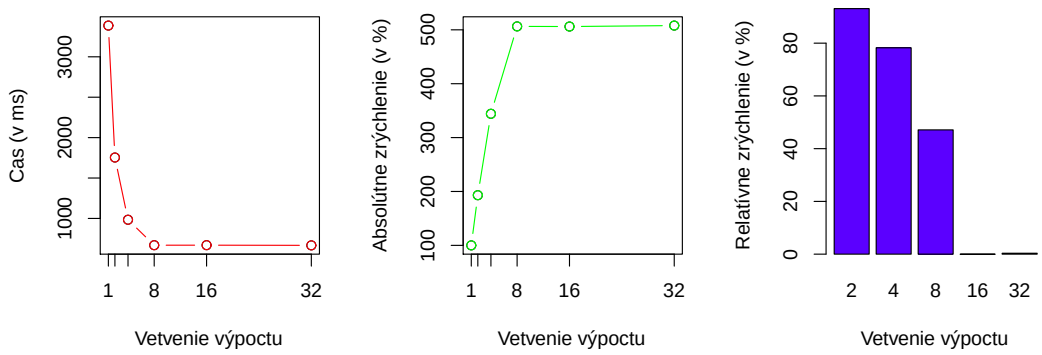
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	2229.83	46
2	1150.86	50
4	623.42	48
8	418.14	42
16	422.22	45
32	417.71	41

Tabuľka A.7: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.94	3.58	5.33	5.28	5.34	1.85	1.49	0.99	1.01

Tabuľka A.8: Zrýchlenie pri zmene stupňa vetvenia.

A.5 Rozdiel (zdieľaná haš. t., set, 2-step)



Obr. A.5: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

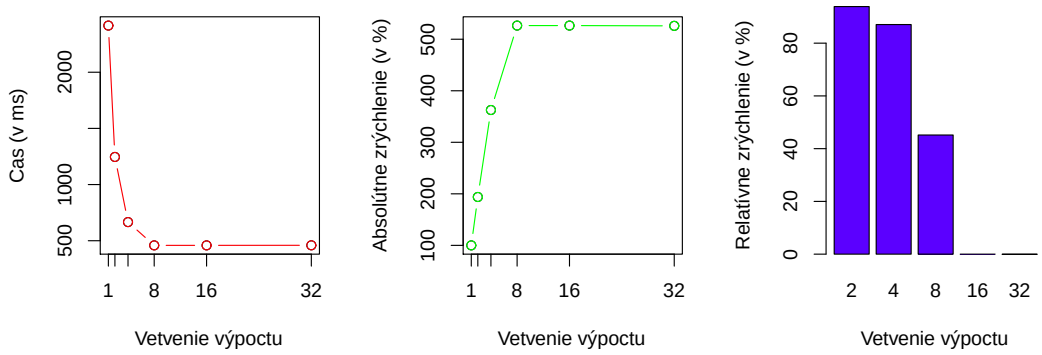
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	3386.35	48
2	1753.74	47
4	983.89	46
8	668.75	40
16	669.08	39
32	666.77	39

Tabuľka A.9: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.93	3.44	5.06	5.06	5.08	1.78	1.47	1.00	1.00

Tabuľka A.10: Zrýchlenie pri zmene stupňa vetvenia.

A.6 Rozdiel (zdieľaná haš. t., bag, 3-step)



Obr. A.6: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

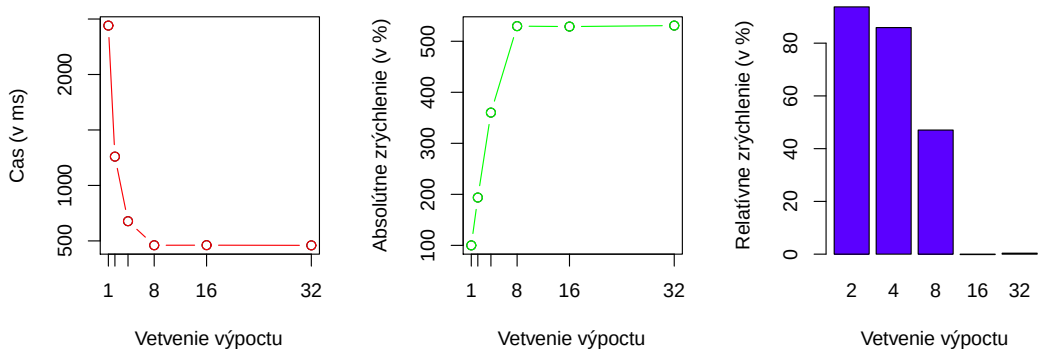
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	2415.34	47
2	1245.81	42
4	666.00	44
8	458.77	47
16	458.77	47
32	459.25	48

Tabuľka A.11: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.94	3.63	5.26	5.26	5.26	1.87	1.45	1.00	1.00

Tabuľka A.12: Zrýchlenie pri zmene stupňa vetvenia.

A.7 Rozdiel (zdieľaná haš. t., set, 3-step)



Obr. A.7: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

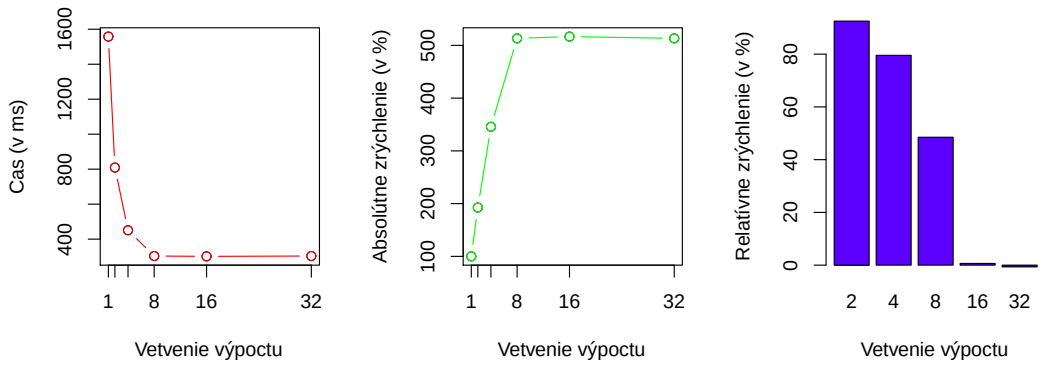
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	2440.98	47
2	1259.84	44
4	677.68	44
8	460.85	46
16	461.57	49
32	459.90	48

Tabuľka A.13: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.94	3.60	5.30	5.29	5.31	1.86	1.47	1.00	1.00

Tabuľka A.14: Zrýchlenie pri zmene stupňa vetvenia.

A.8 Spojenie (zdieľaná haš. t.)



Obr. A.8: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

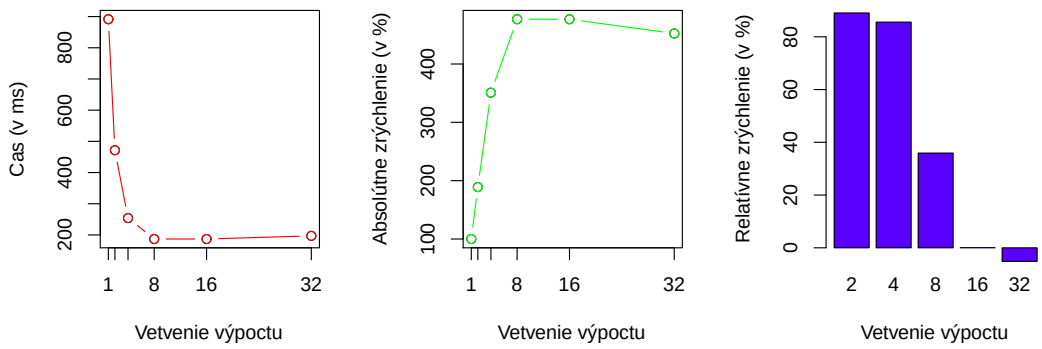
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	1558.08	49
2	809.27	41
4	450.71	48
8	303.51	49
16	301.55	49
32	303.51	49

Tabuľka A.15: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.93	3.46	5.13	5.17	5.13	1.80	1.48	1.01	0.99

Tabuľka A.16: Zrýchlenie pri zmene stupňa vetvenia.

A.9 Prienik (lokálna haš. t., bag)



Obr. A.9: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

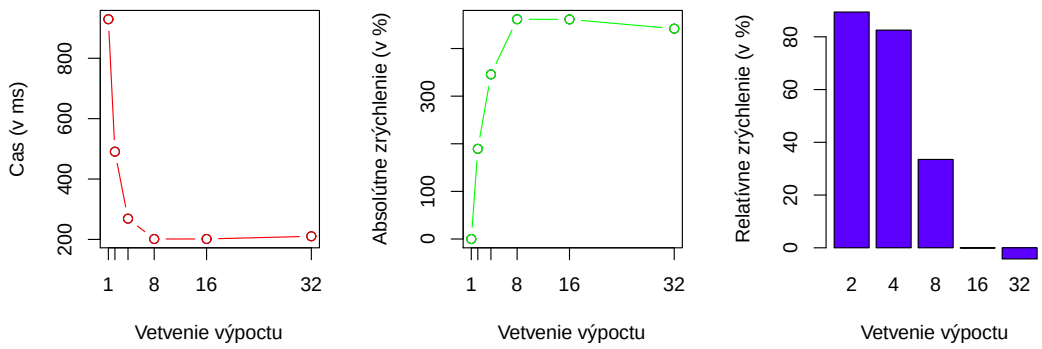
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	891.50	50
2	471.60	50
4	254.15	47
8	187.00	44
16	187.00	39
32	197.10	49

Tabuľka A.17: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.89	3.51	4.77	4.77	4.52	1.86	1.36	1.00	0.95

Tabuľka A.18: Zrýchlenie pri zmene stupňa vetvenia.

A.10 Prienik (lokálna haš. t., set)



Obr. A.10: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

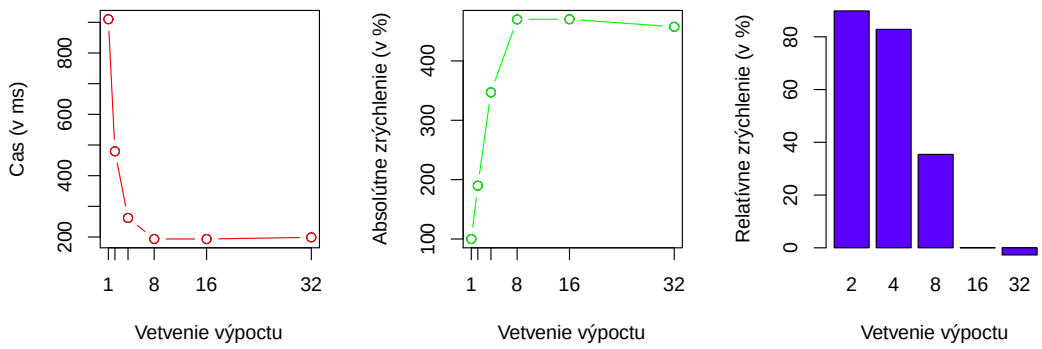
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	929.34	50
2	490.68	50
4	268.82	49
8	201.36	39
16	201.51	43
32	210.39	49

Tabuľka A.19: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.89	3.46	4.62	4.61	4.42	1.83	1.34	1.00	0.96

Tabuľka A.20: Zrýchlenie pri zmene stupňa vetvenia.

A.11 Rozdiel (lokálna haš. t., bag, 2-step)



Obr. A.11: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

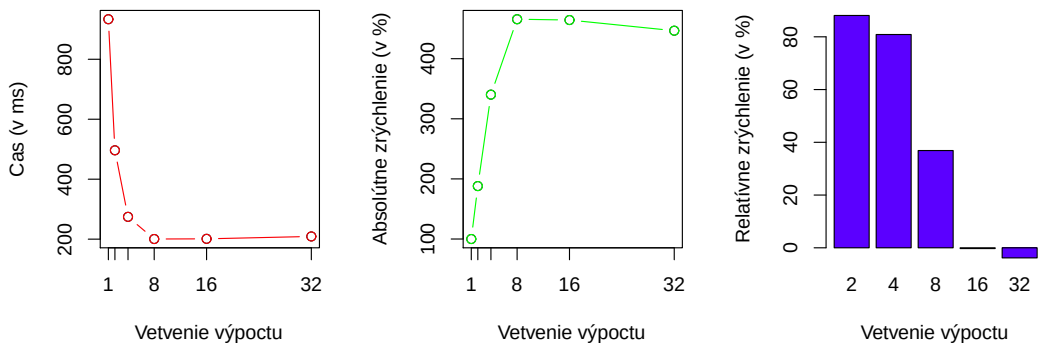
Vetvenie výpočtu	Priemerný čas	Počet meraní
1	910.08	51
2	479.43	49
4	262.23	47
8	193.65	48
16	193.53	49
32	198.91	47

Tabuľka A.21: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.90	3.47	4.70	4.70	4.58	1.83	1.35	1.00	0.97

Tabuľka A.22: Zrýchlenie pri zmene stupňa vetvenia.

A.12 Spojenie (lokálna haš. t.)



Obr. A.12: Vývoj priemerného času výpočtu, absolútneho zrýchlenia a relatívneho zrýchlenia v závislosti od vetvenia výpočtu.

Vetvenie výpočtu	Priemerný čas	Počet meraní
1	933.54	50
2	496.33	49
4	274.41	49
8	200.47	38
16	201.05	41
32	209.04	50

Tabuľka A.23: Priemerné časy výpočtu pre rôzne stupne vetvenia.

1→2	1→4	1→8	1→16	1→32	2→4	4→8	8→16	16→32
1.88	3.40	4.66	4.64	4.47	1.81	1.37	1.00	0.96

Tabuľka A.24: Zrýchlenie pri zmene stupňa vetvenia.

Dodatok B

CD

Na priloženom CD je možné nájsť:

- Text diplomobej práce vo formáte pdf.
- Zdrojové súbory textu diplomovej práce pre LaTeX.
- Zdrojové súbory implementácie databázových operácií, prostredia pre testovanie, generátory modelov a ďalšie pomocné funkcie.
- Zdrojové súbory paralelného prostredia **Bobox**.
- Súbory s dátami, na ktorých boli realizované merania.
- Súbory s výsledkami meraní.
- Skripty na generovanie dát a analyzovanie dát pre program R.
- Súbor README.txt

Podrobnosti o obsahu CD sú popísané v súbore README.txt.