Vysoké učení technické v Brně
Fakulta informačních technologií

Modelování a simulace - 6. Počítačové služby

Porovnávání SQL a JAVA přístupů do databáze

Vojtěch Meluzín - xmeluz04
Matěj Mlejnek - xmlejn04

31. ledna 2018

Obsah

1 Úvod

V této práci je řešen projekt do předmětu **IMS - Modelování a simulace** [1] vyučovaném na Fakulta informačních technologií Vysokého učení technického v Brně [2]. Konkrétně se jedná o zadání **6. Počítačové služby** [3].

Tato práce se věnuje problematice doby vyhledávání v databázi. Zaměříme se na porovnávání přístupu do databáze výhradně přes SQL dotazy a přístupu do databáze spojeného s cacheováním (jednotlivých řádků vyhledávané tabulky) na počítači klienta.

V našich experimentech se zaměřujeme zjištění za jakých podmínek je efektivnější pro vyhledávání použit spíše databázový server a nebo ve veškerých datech vyhledávát až na straně klienta. Vzhledem k tomu, že na tyto časy hraje roli několik faktorů výběr nemusí být na první pohled hned jasný. V kapitole Experimenty (viz. Experimenty) jsou sepsány jednotlivé krajní i více obecné (realističtější) případy při práci s databází.

2 Zdroje faktů

Jako model jsme si vybrali databázi Postgresql ve verzi 9.5.10 [7]. Pro přístup do této databáze jsme zvolili naprogramování aplikace v jazyce JAVA [8] ve verzi JDK-1.8.0_151 [9], ve které jsme si naprogramovali komunikaci se serverem. Programy pro sběr dat z této komunikace běželi na virtualním stroji Ubuntu 16.04.3 LTS [10] a samotné posílání jednotlivých dotazů bylo zautomatizované pomocí scriptu psaném v GNU Bash version 4.3.48(1)-release (x86_64-pc-linux-gnu) [11].

2.1 Průběh sběru dat

Pro přístup k datům z databáze a měření doby přístupu k datům, jsme se rozhodli, že bude vhodné, aby se vyhledávací dotazy vytvářeli na virtuální počítači odděleného od virtuálního počítače s databázovým serverem. Vytvořili jsme tedy 2 virtuální počítače s těmito parametry:

VM1SERVER	
CPU	2 jádra - 4,2 GHz
RAM	2048 MB
HDD	25 GB
OS	Ubuntu - 16.04.3

VM2CLIENT	
CPU	2 jádra - 4,2 GHz
RAM	8192 MB
HDD	25 GB
OS	Ubuntu - 16.04.3

Parametry byli vybrány, tak aby splňovali minimální systémové požadavky a zároveň bylo co nejvíce místa pro nacachování prohledávaných tabulek.

Tyto parametry jsme zvolili, tak aby splňovali požadavky operačního systému Ubuntu [13], databázového serveru Postgresql [14] a zároveň aby se na straně klienta spustila JAVA s námi vybranými velikostmi RAM pro měření.

OS parametry Operační systém ubuntu [10] jsme si vybrali z důvodu jednoduché instalace jednotlivých aplikací potřebných pro tento sběr dat, nízké náročnosti na hardwarové požadavky a jednoduchou obsluhu. Jako databázový systém nám posloužil PostgreSQL [7]. Tento systém jsme zvolili z důvodu jednoduché instalace, nízkých nároků na hardware a The PostgreSQL Licence(mirně modifikované

Open Source licence) [15]. Při výběru s jakou databází budeme pracovat jsme si nezvolili Oracle [16] z důvodu, že již není volně k dispozici pro komerční použití a naše výsledky by nebyli dostatečně využitelné.

Na prvním virtuálním počítači běžel server (dále jen VM1SERVER) s databází PostgreSQL a měl za úkol zpracovávat přijaté SQL dotazy a odpovídat na ně. Druhý virtuální počítač znázorňující klienta (dále jen VM2CLIENT) se postupně připojoval na databázový server a posílal dotazy.

Na VM2CLIENT tedy běžel BASH [11] script *functions.sh*, který automaticky spouštěl námi naprogramované Javové dotazy z */dist/testApp.jar*.

V rámci automatických testů se také před každým měřením musel zaslat požadavek pro vymazání cache paměti v databázi na straně serveru, toto je vyřešeno připojením přes OpenSSH rozhrani (OpenSSH_7.2p2 Ubuntu-4ubuntu2.4, OpenSSL 1.0.2g 1 Mar 2016) [12] *sshclear.sh* připojením se na VM1SERVER a odtud zavoláním scriptu *clearcache.sh*.

2.1.1 Přesnost vyhledávání

Jako hodnotu pro vyhledávání jsme si zvolili index(od 1 do velikosti tabulky po 1). Rozhodli jsme se filtraci provádět podle začátku řetězce jeho hodnoty. Tato filtrace byla zvolena z toho důvodu, že se nejvíce podobá přístupu do databáze k vyhledání určité položky, aneb jak s databází pracuje normální uživatel. Při práci s databázi byla tato filtrace prováděna příkazem LIKE 'prefix%' [20] a v JAVA startsWith(String prefix) [19] (dále se na obě funkce budeme zároveň odkazovat jako na pseudofunkci LIKEE(prefix)).
Příklady LIKEE() nad řetězci:

$$LIKEE(\text{"}AH\text{"}) > (\text{"}AHOJ\text{"}) => True$$

$$LIKEE(\text{"1"}) > (\text{"10"}) => True$$

$$LIKEE(\text{"0"}) > (\text{"10"}) => False$$

$$LIKEE(\text{"1"}) > (\text{"1"}) => True$$

Jelikož v našem projektu pracujeme s indexovanými tabulkami, tak můžeme vyhledávat počet výskytů indexů se stejným začátkem:

$$LIKEE(\text{"1"}) > TABLE(100) => 12$$

$$LIKEE(\text{"1"}) > TABLE(1000) => 111$$

$$LIKEE(\text{"10"}) > TABLE(1000) => 2$$

$$LIKEE(\text{"10"}) > TABLE(500) => 11$$

$$LIKEE(\text{"100"}) > TABLE(100) => 1$$

$$LIKEE(\text{"100"}) > TABLE(30000) => 111$$

$$LIKEE(\text{"1000"}) > TABLE(500) => 0$$

LIKEE V našich experimentech jsme pracovali s LIKEE(0), LIKEE(1), LIKEE(10), LIKEE(100), LIKEE(1000). Kde **LIKEE(0)** znamená výběr všech hodnot z tabulky.

2.1.2 RAM pamět

Pro výběr s jakými hodnoty RAM paměti budeme pracovat jsme byli limitování fyzickým hardwarem, proto jsme na VM1SERVER usoudili, že nebude potřeba větší než minimální Systémem a Databází požadovaná viz: **OS parametry** 2.1. Pro VM2CLIENT jsme měli 8GB paměti. Nyní stačilo vyhledat pro jaké RAM paměti nám JAVA běžící na VM2CLIENT dovolí s maximální velikostí tabulek. Zjistili jsme, že pro JAVA při paměti 512MB dokáže nad našími tabulkami(ve formátu viz: 2.1.3) pracovat s maximálně 150 000 tabulkami. Na konec jsme zvolili RAM 1024, 2048 a 4096. Tyto RAM paměti bez problému pojmou i naši největší zvolenou tabulku (250 000).

2.1.3 Vygenerování databáze

Pro generaci potřebných dat jsme si vybrali csv generátor [17] jednotlivé řádky jsme se rozhodli selectovat pomocí indexu("seq"), bylo ovšem zapotřebí, aby se tabulka podobala tabulkám se kterými se pracuje v reálném životě, proto jsme si zvolili, že každý řádek bude obsahovat hodnoty pro tyto sloupce:

$$"seq, first, last, age, street, city, state, zip,$$

$$dollar, pick, date, email, digid, latitude, longitude,$$

$$pick2, string, domain, float, ccnumber, bool, yn"$$

Například:

$$1, Jesse, Watts, 51, HusfoTerrace, Livemil, HI, 75091, \$2932.33,$$

$$YELLOW, 7/13/1993, sajuvubug@uj.net,$$

$$147889110758, 12.63144, -166.12131, UP, W4P9nY0yubdKsQu)sxI,$$

$$boto.co.uk, -275370638258.9952, 6304284025402256, true, N$$

Velikost jednotlivých řádků tabulek by neměla ovlivnit poměr časů vyhledávání, ale slouží k tomu, aby jsme vygenerovali vstupní data k simulaci ve kterých uvidíme větší hodnoty s menší pravděpodobností vzniklých nepřesností.

Tyto data jsme umístili na VM1SERVER, tak že jsme na něm spustili databázi a připojili jsme se na ní pomocí PG ADMIN 3 [18] a nahráli zde tabulky ze kterých jsme plánovali získávat data.

Tabulky Původně jsme plánovali pracovat s tabulkami velkými až 10 000 000, to jsme ovšem velice rychle zavrhli z důvodu velké časové náročnosti výpočtů. Nakonec jsme pracovali s tabulkami do velikosti 250 000.

2.1.4 Automatizace pomocí BASH

Z bash scriptu byl volán JAVA program [link] postupně po jednom tak aby byla provedena kombinace všech vstupních parametrů z předchozí sekce(RAM, TABLES,LIKEE). Naměřené hodnoty jsme ukládali do formátu csv. Při

2.1.5 Naměření hodnot JAVA

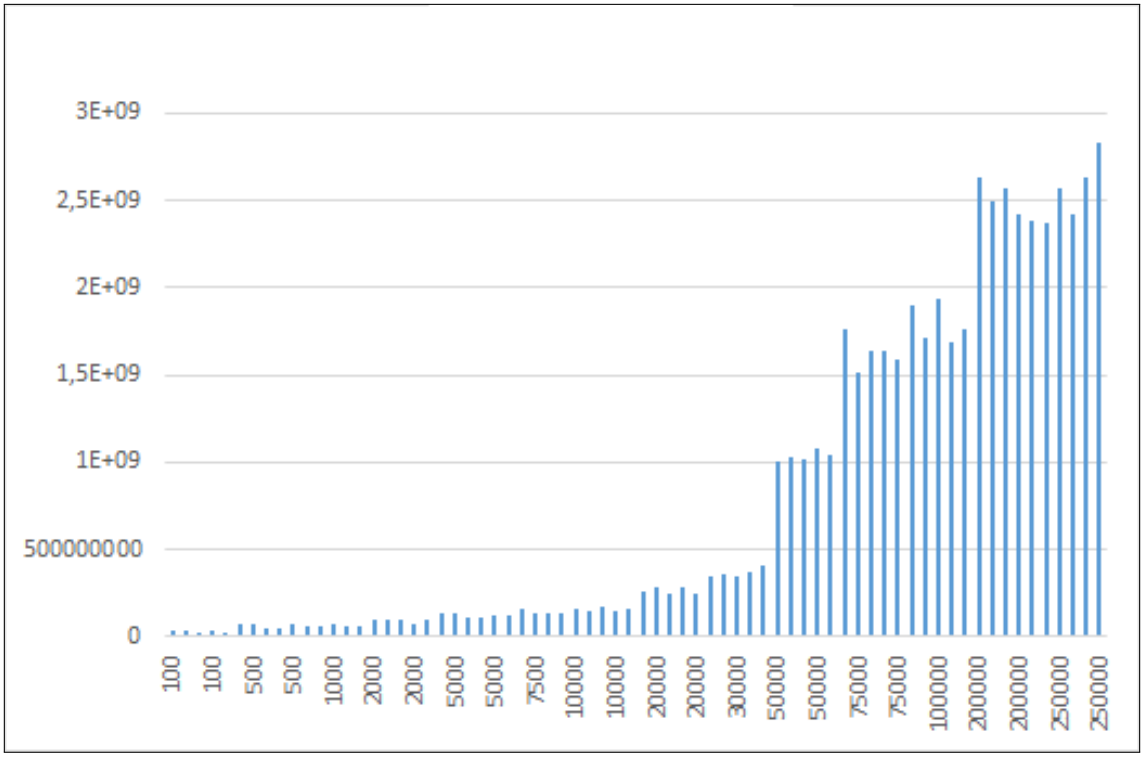
Pro získávání dat jsme vytvořili program v jazyce JAVA (viz./*dist/testApp.jar*). V tomto programu spuštěném na VM2CLIENT se připojujeme na databázový server Postgresql běžícím na VM1SERVER. Naměřili jsme si tyto hodnoty:

- Doba selectu z db bez cache
- Doba selectu z db s cache

- Doba selectu z db bez cache a s filtrací
- Doba selectu z db s cache a s filtraci
- Doba vytvoření objektů v JAVA
- Doba filtrace v JAVA

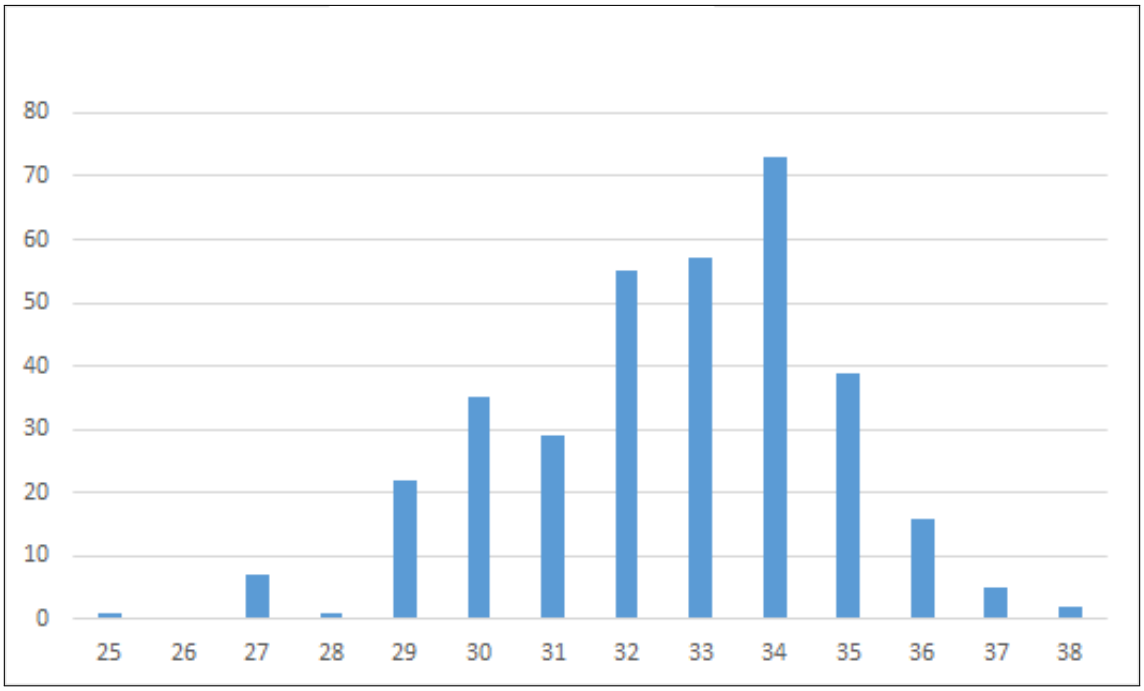
2.2 Zpracování naměřených hodnot

Výstupní *csv* soubory jsme si prevedli do tabulkového formatu *xlsx* pomocí aplikace Microsoft Excel [23], kde jsme nad jednotlivými časy vytvářeli grafy. Na obrázku číslo 1 můžete vidět dobu vytváření objektů všech prvků z tabulky na základě její velikosti. Z grafu je jasně vidět, jak se jednotlivé časy mění při změně velikosti tabulky.

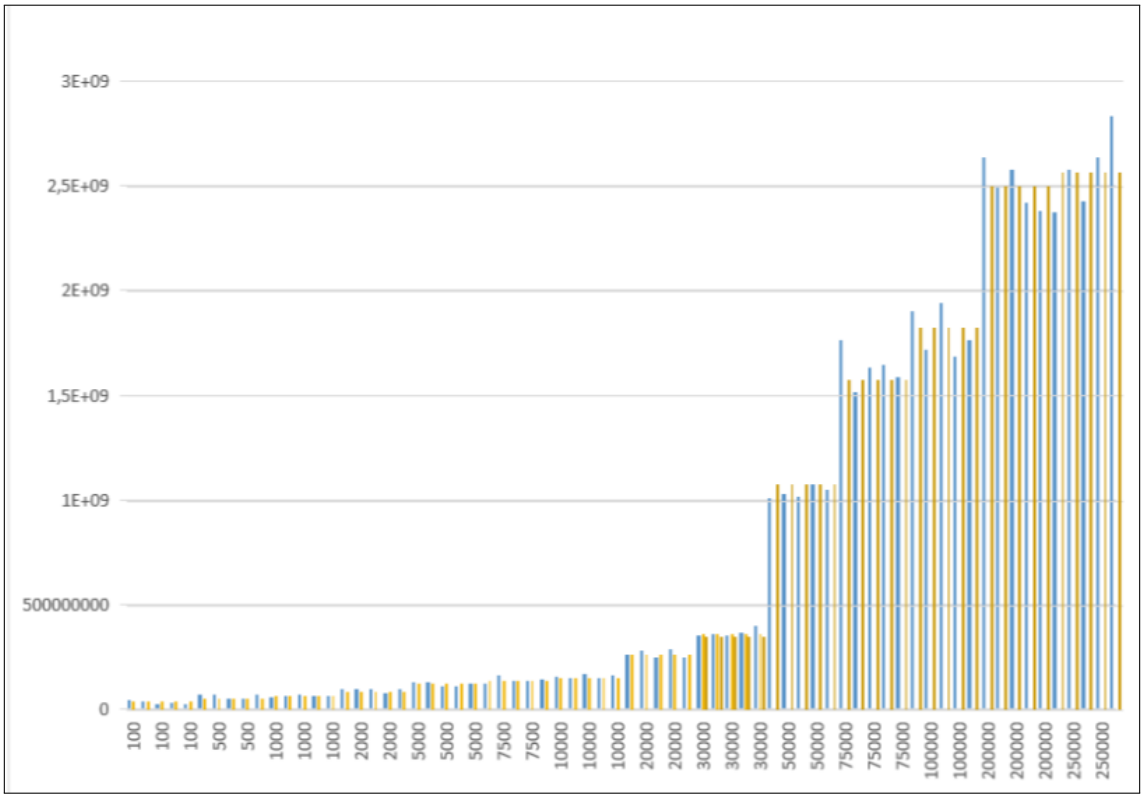


Obrázek 1: LIKEE-1-CREATE-ALL - 5x měřená hodnota pro každou velikost tabulky - Vertikální osa čas 10^{-9} sekund a horozontální značí velikosti tabulek.

Normální rozložení Potřebovali jsme na základě těchto hodnot vytvořit funkce podle kterých, by jsme mohli určit přibližnou hodnotu různě velkých tabulek, pro které nemáme hodnoty naměřené. Na základě měření jsme rozhodli, že veškeré měřené hodnoty mají normálního rozložení (například tabulka o velikosti 100, nad kterou byl volán Select všech rádků (viz. obrázek **číslo 2**). U každého jednotlivého grafu znázorňující naměřené hodnoty (viz. obrázek1) bylo zapotřebí zvážit zda není vhodné ho rozdělit na více intervalů, kde by vytvořená funkce počítala přesněji. Z hodnot ze zvoleného intervalu jsme přes online nástroj pro vytváření polynomu viz. [21] vždy získali dány polynom většinou čtvrtého řádu, do kterého jsme potom dosadili původní velikosti tabulek. Původní graf hodnot a hodnoty z nových funkcí jsme si dali dohromady do grafu na porovnání. Na obrázku **číslo 3** můžeme vidět jak žlutá barva znázorňuje body spočítané z nově vytvořených funkcí.



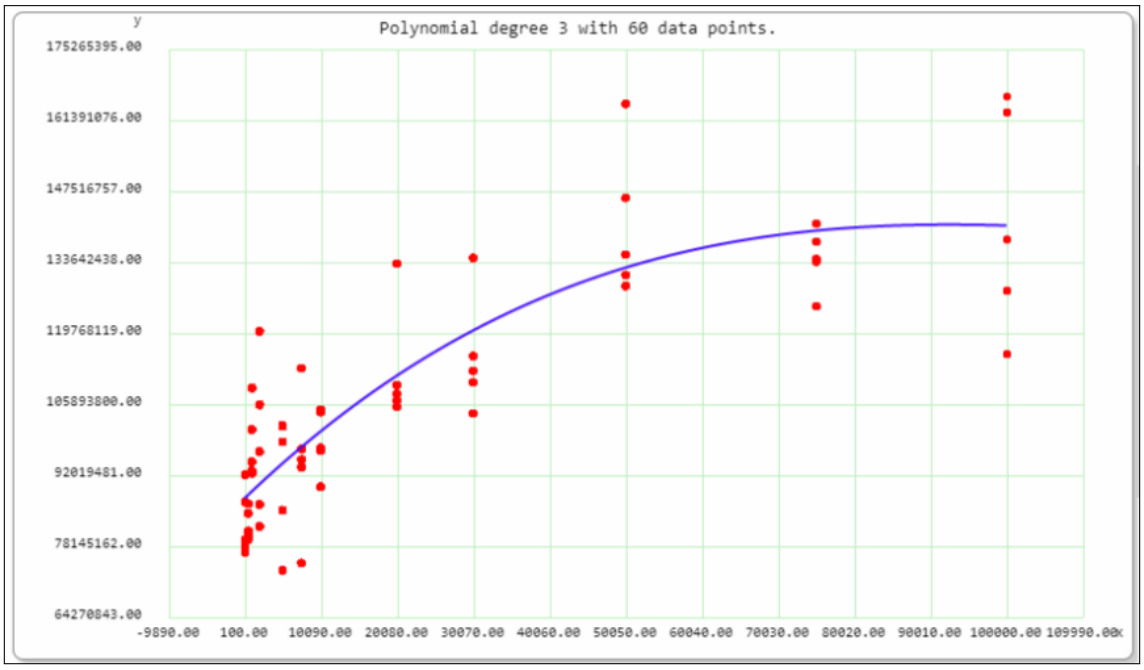
Obrázek 2: 100-TABLE-SELECT-ALL - rozložení při 342 měřeních - vertikální osa značí počet výskytů a horozontální značí 10^{-2} sekund zaokrouhleno na celá čísla



Obrázek 3: LIKEE-1-CREATE-ALL-FUNCTIONS - 5x měřená hodnota pro každou velikost tabulky - Vertikální osa čas 10^{-3} sekund a horozontální značí velikosti tabulek.

DX - odchylka Pro jednotlivé intervaly jsme si u každého bodu co jsme měli naměřenou hodnotu spočítali hodnotu funkce, porovnali s naměřenou hodnotou, a tím zjistili chybu v jednotlivých bodech. Tyto chyby stačilo zprůměrovat a získali jsme hodnotu odchylky.

EX - Střední hodnota Střední hodnotu jsme zjistili z funkcí aproximovaných z naměřených hodnot (obrázek **číslo 4**). Tato funkce nám potom vypočítávala pro danou velikost tabulky její střední hodnotu.

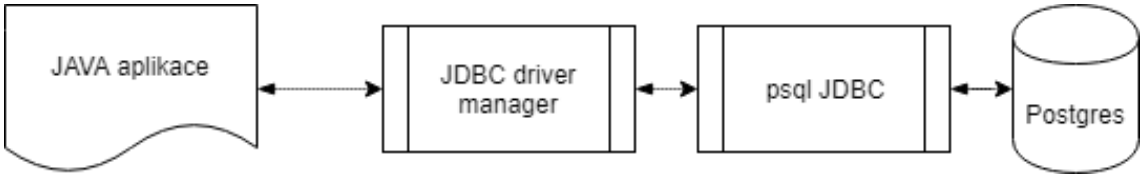


Obrázek 4: Příklad proložení naměřených hodnot krivkou zapoužití online nástroje *arachnoid* [22] při vytváření polynomů

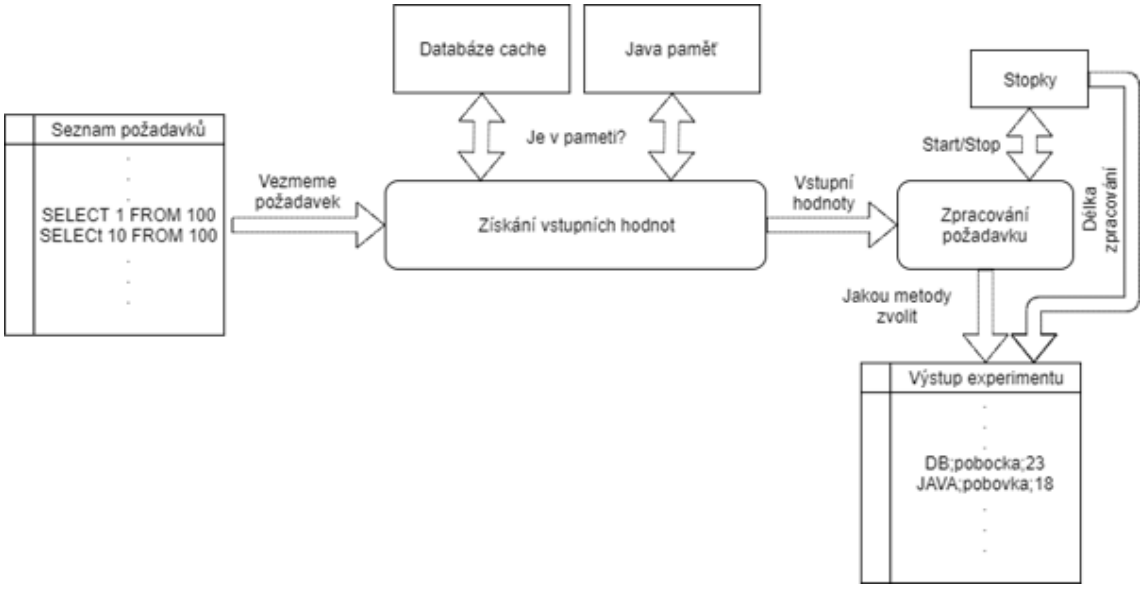
3 Koncepce modelu

3.1 Komunikace JAVA a databáze

Na obrázcích **č. 5** a **č. 6** je znázorněna komunikace mezi JAVou a databází za pomoci JDBC - API pro programovací jazyk Java, který definuje jednotné rozhraní pro přístup k relačním databázím [24]. Pro přístup ke konkrétnímu databázovému serveru je potřeba JDBC driver (ovladač), který poskytuje tvůrce databázového serveru. Pro náši PostgreSQL databázi dostupný na oficiálních stránkách [25].



Obrázek 5: Komunikace JAVA-DB



Obrázek 6: Schéma JAVA-DB

3.2 Simulace experimentu

Jednotlvé experimenty (viz. kapitola **Experimenty 5**) nám představují seznam požadavků (viz. kapitola **Experimenty vysvětlení požadavků 5**). Bereme postupně požadavky a získáváme informace o velikosti tabulky a přesnosti filtrování viz. **LIKEE 2.1.1**. Na základě těchto informací se zeptáme jestli je tato tabulka v cache databáze a jestli je v paměti JAVY. Tyto čtyři hodnoty pak pošleme do procesu na zpracování jednoho požadavku. Stopujeme si čas jak dlouho trvá pro JAVU a Databázi zpracování požadavku. Jednotlivé doby trvání zpracování požadavku dáváme do výstupního souboru pro daný experimentu.

3.3 Architektůra sim modelu

Simulační model byl napsán v jazyce C++[26] za pomocí simulační knihovny Simlib[4, 6]. Dále popsáno v kapitole **Simlib 4**.

3.4 Konceptuálního modelu na simulační

Simulační model je vytvořen na základě modelu přistupování do databázového serveru běžně používaném v reálném světě. V našem projektu jsme pracovali pouze s daty z virtuální verze, ve které jsme se snažili co nejpřesněji reálný systém imitovat.

3.4.1 Mapování konceptuálního modelu na simulační

Konceptuální model	**Procedůra**
Databáze	**DB_* procedůry (...)**
Vytváření objektů	Proces_create_objects
Select radku z DB First	Proces_select_for_first_time
Select radku z DB N	Proces_select_N
FIltrace v Javě	Proces_filter_objects

3.5 Zpracování požadavku

Zpracování jednoho požadavku máme znázorněné na Petriho síťi (viz. IMS - Modelovani a simulace - strana 123. [5]) (**Petriho sítě Obr. č. 7**). Toto zpracování nám pouze určuje jakou metodu filtrace dat(výběr mezi JAVA a Databází), bychom měli zvolit pro vstupní hodnoty.
Význam vstupních hodnot:

- Přesnost filtrace: jaký LIKEE je daný požadavek
- Velikost tabulky: počet řádků vstupní tabulky
- Tabulka je v ram. JAVA: říká nám jestli je tabulka nacacheovaná v paměti JAVy
- Tabulka je v ram. Databáze: určuje jestli je tabulka nacacheovaná v paměti databáze

3.6 Petriho sít

Naše petriho sít (Obr . č. 7) je pouhým zjednodušeným modelem simulace. Při tvorbě skutečné simulace jsme sice vycházeli z toho to návrhu, ale trošku se nám liší (viz. kapitola **Simlib 4**).

V obrázku Petriho sítě(Obr . č. 7) máme zakresleno, že nám vstupní hodnoty Přesnost filtrace a Velikost tabulky vstupují do přechodu s názvem Výběr z DB / Vytváření objektů / Filtrace v JAVě. Znamená to, že tyto vstupní hodnoty použijeme při zjišťování doby konání těchto přechodů.

Pokud máme hodnoty Tabulka je v pam. JAVě a Tabulka je v pam. databáze zadané, tak se nám vygenerují dva procesy.

Procesy Start/Stop stopky pro Javu a Databáni nám určují místo kdy se začne měřit čas potřebný pro zpracování požadavku, pro danou metodu.

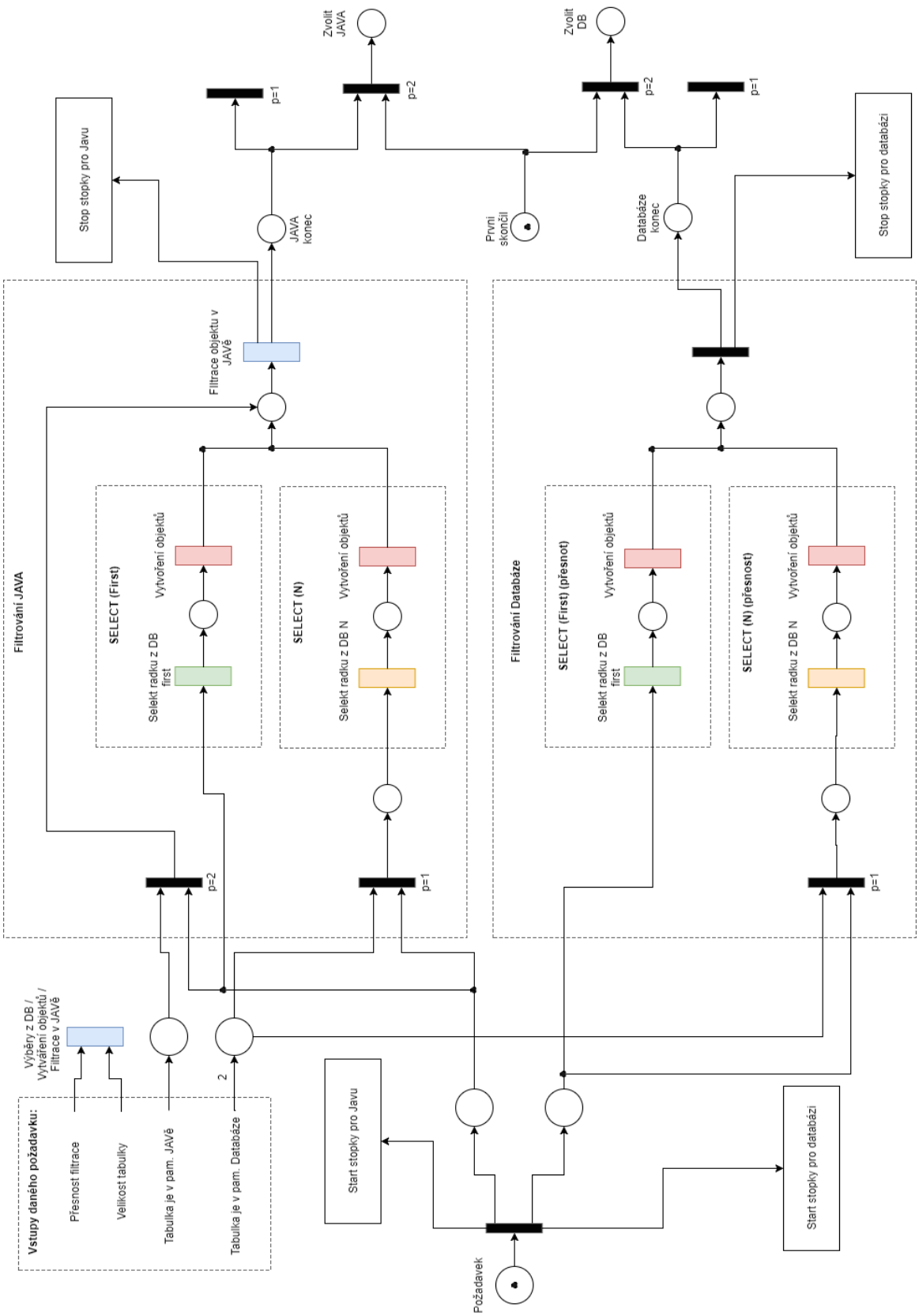
Na začátku petriho sítě (Obr. **č. 7**) nám vznikne jeden proces v **Požadavek**. Ten se rozdělí na dva procesy, jeden bude představovat zpracování Javou a druhý zpracování databází. Na základě vstupních požadavků jsou vykonány pro JAVU a databázi rozdílné přechody. Na příklad pokud máme na vstupu dáno, že je tabulka v paměti databáze, tak se vykoná proces SELECT (N). Jakmile se proces pro JAVU a databázi dokončí, zastaví stopky a pokud skončil jako první, tak přejde do stavu Zvolit. Stavy Zvolit máme dva, jeden pro každou metodu. Jde zvolit vždy jen jeden z těchto dvou stavů. Dozvíme se tak, kterou z těchto dvou metod máme zvolit. Jednotlivé přechody a jejich významy jsou vysvětleny v kapitole **Simlib 4**.

Filtrování Filtrování **JAVA** se skládá ze tří částí:

- SELECT (first) - případ, kdy databáze ještě nemá tabulku v paměti cache (nevyužívá přesnosti)
- SELECT (N) - databáze již má tabulku v paměti (nevyužívá přesnoti)
- SELECT (N) - databáze již má tabulku v paměti (využívá přesnoti)

Filtrování **Databáze** (SQL dotazy) je složena ze dvou částí:

- SELECT (first) (přesnost) - je stajná jako u filtrace v Javě, je s rozdílem, že využívá přesnosti
- SELECT (N) (přesnost) - také ještě navíc využívá přesnosti



Obrázek 7: Petriho sít

4 Simlib

Program v simlibu[4, 6] je rozdělen na dvě hlavní větve znázorňující model pro filtraci dat přes JAVu a přes Databázový server. Jednotlivé doby vyhledání jsou zakresleny a jejich průběhy popsány v kapitole **Petriho síte 3.6**. Na začátku programu se na základě předem určených dat, nad kterými se budou provádět simulace (zvolených experimentů) vytvoří fronty pro větve Javy a databáze. Na základě hodnot spočítaných z funkcí, které jsme získali při analýze vstupních dat (viz. kapitola **Zpracování naměřených hodnot 2.2**) propočítáváme doby trvání.

5 Experimenty

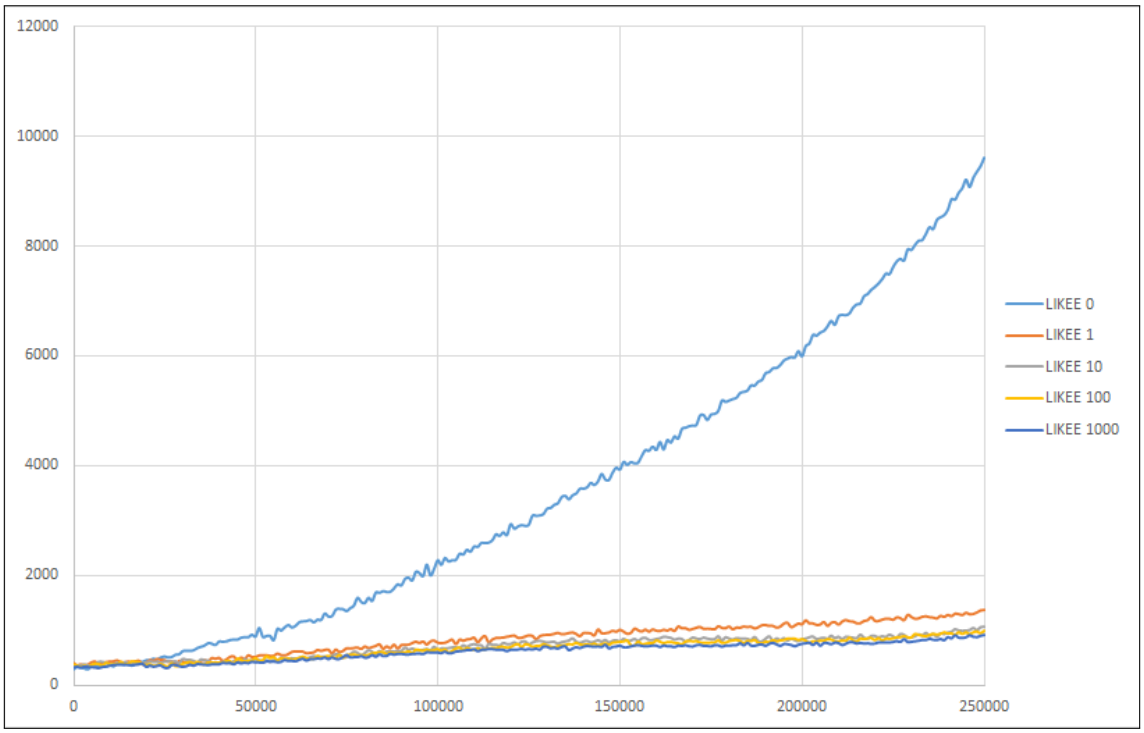
5.1 Podstata simulačních experimentů a jejich průběh

Vytvořili jsme několik experimentů. Chceme na nich ukázat, při jakých parametrech se vyplatí filtrovat data v databázi a kdy se nám vyplatí si načíst celou tabulku a data si pak filtrovat v javě.

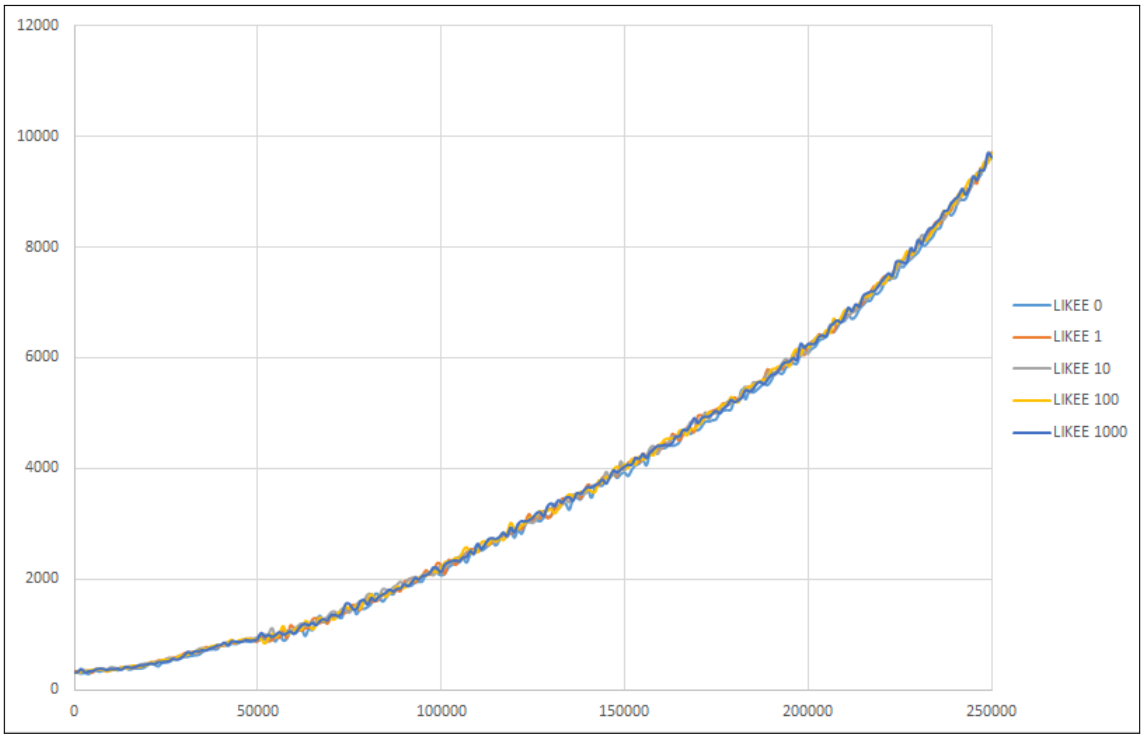
5.2 Příklady

5.2.1 SELECT FIRST MEMORY

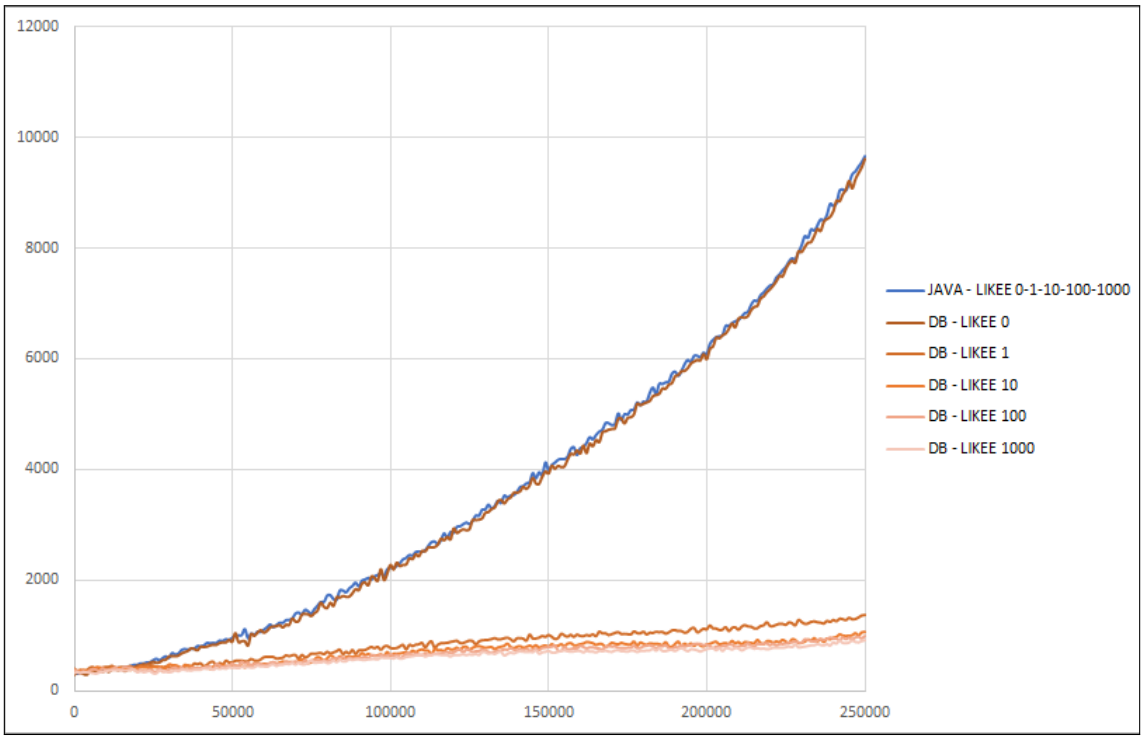
Na obrázcích 8, 9, 10 jsou znázorněny průběhy dob vyhledávání v JAVA oproti Databázi. Jedná se o experiment s nenacachovanými hodnotami (FIRST). Na obrázcích můžeme videt jak se jednotlivé selecty chovají. JAVA jelikož není nacachovaná bude pro filtraci s jakoukoliv přesností mít velice podobný průběh. Databáze ovšem jasně znázorňuje, že čas za který, při přesnějším vyhledávání, proběhne je mnohem menší.



Obrázek 8: FIRST - DB - LIKEE 0-1-10-100-1000 - Vertikální osa čas 10^{-3} sekund a horozontální značí velikosti tabulek.



Obrázek 9: FIRST - JAVA - LIKEE 0-1-10-100-1000 - Vertikální osa čas 10^{-3} sekund a horozontální značí velikosti tabulek.



Obrázek 10: FIRST - DB AND JAVA - LIKEE 0-1-10-100-1000 TABLE 0-40000 - Vertikální osa čas 10^{-3} sekund a horozontální značí velikosti tabulek.

5.2.2 Příklad ze života

Tento experiment jsme vytvářeli jako příklad z nějakého většího systému. Předpokládáme, že máme nějakou firmu, která má spousty zaměstnanců a my potřebujeme vyhledat informace o jednom zaměstnanci. Víme, kde pracuje a jakou má pozici.

Na začátku si zobrazíme všechny pobočky. Potom si pomoci filtrace najdeme určitou pobočku, zobrazíme si informace o pobočce. Vybereme si jednoho zaměstnance. Zobrazíme si opět všechny pobočky. V kódu experimentu to pak bude vypadat následovně:

Vypsani všech poboček:

- 39000, 0, //LIKEE(0) = SELECT vsech řádků tabulky

Vybrani jedne pobočky:

- 39000, 1000,

Zobrazení informací o pobočce

- 100, 100, // dohledani nazvu prodejny
- 500, 10, // dohledani zamestnancu na prodejne
- 451, 10, // dohledani dalsich informaci
- 452, 10, // dohledani dalsich informaci
- 453, 10, // dohledani dalsich informaci

Vypsání opět všech poboček:

- 39000, 0, //LIKEE(0) = SELECT vsech řádků tabulky

5.2.3 Další:

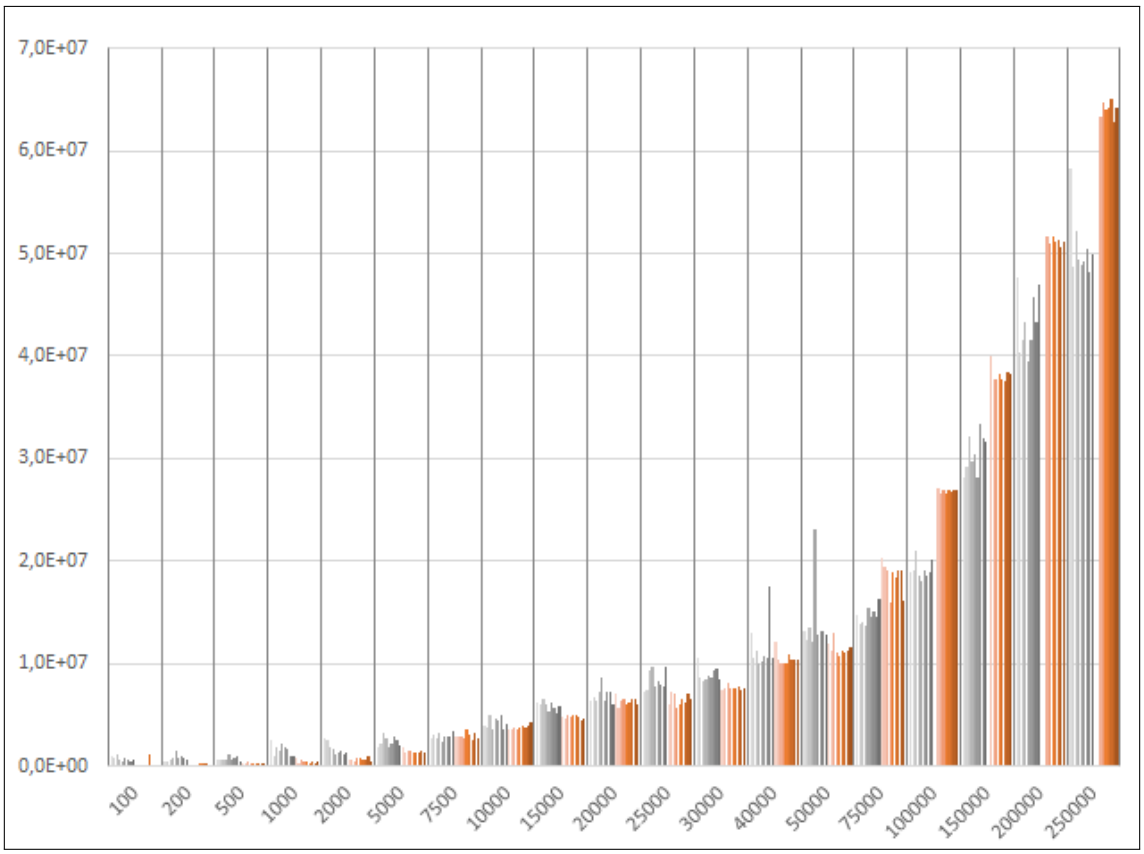
- Obecný - Máme velkou tabulku a z ní si pak filtrujeme
- Pouze jedna opravdu velkou
- filtrace velké tabulky - ale z nich jen malá data
- Smíšený
- Zátěžový - spoustu velký tabulek
- Spoustu nejpřesnějších informací

5.3 Výsledky/Grafy

Vzhledem k tomu, že jsme zjistili chybu ve sběru dat, musíme s lítostí přeskočit prezentaci výsledků a grafů, jelikož by vyvozovali závěry neodpovídající reálnému systému.

6 Závěr

Na základě výsledků z předběžných měřeních jsme očekávali, že filtrování pomocí databáze se má vyplatit u tabulek větších než 70 000 řádků. Tuto hodnotu jsme odhadovali na základě **Obr. č. 11**, kterou jsme naměřili při prvotním měření dat v systému s jinými parametry. Považujeme ji za správnou, jelikož logicky Databázové SQL SELECTY musí filtrovat být při větším počtu řádků v tabulce mnohem efektivněji, protože nemusí se přenášet veškerá data jako JAVA.



Obrázek 11: JAVA - oranžová, DB - šedá — Vertikální osa čas 10^{-6} sekund a horozontální značí velikosti tabulek.

Po provedení našich simulací, se nám toto tvrzení nepodařilo prokázat. Následně jsme zjistili, že je to zapříčiněno chybou v měření. Chybu jsme zpětně nalezli ve špatném postupu měření dob. Chyba byla způsobena rychlostí disku. Virtuální počítače, na kterých probíhal sběr dat, se navzájem přetahovali o přístup do disku.

Předpokládáme, to že kdybychom našemu simulačnímu programu dali přesnější vstupní data, tak by jsme dosáhli výsledků odpovídajícím hodnotám reálného systému.

Literatura

[1] IMS - Modelování a simulace: [online]. [vid. 2017-12-06].
URL <http://www.fit.vutbr.cz/study/course-l.php.cs?id=12167>

[2] Fakulta informačních technologií Vysokého učení technického v Brně: [online]. [vid. 2017-12-06].
URL <http://www.fit.vutbr.cz/>

[3] Zadání č.6: [online]. [vid. 2017-12-06].
URL <http://perchta.fit.vutbr.cz:8000/vyuka-ims/42>

[4] Simlib: [online]. [vid. 2017-12-06].
URL <http://www.fit.vutbr.cz/~peringer/SIMLIB/>

[5] Modelovani a simulace - Petr Peringer - peringer AT fit.vutbr.cz, Martin Hruby - hrubym AT fit.vutbr.cz: [online]. [vid. 2017-12-06].
URL <http://www.fit.vutbr.cz/study/courses/IMS/public/prednasky/IMS.pdf>

[6] Simlib-3.04-20171004.tar.gz: [online]. [vid. 2017-12-06].
URL <http://www.fit.vutbr.cz/~peringer/SIMLIB/source/>

[7] Postgresql: [online]. [vid. 2017-12-06].
URL <https://www.postgresql.org/?&>

[8] Java: [online]. [vid. 2017-12-06].
URL <https://java.com/en/download/>

[9] JDK-1.8.0_151: [online]. [vid. 2017-12-06].
URL <http://www.oracle.com/technetwork/java/javase/8u151-relnotes-3850493.html>

[10] Ubuntu 16.04.3 LTS: [online]. [vid. 2017-12-06].
URL <http://fridge.ubuntu.com/2017/08/05/ubuntu-16-04-3-lts-released/>

[11] GNU Bash: [online]. [vid. 2017-12-06].
URL <https://www.gnu.org/software/bash/>

[12] OpenSSH: [online]. [vid. 2018-01-05].
URL <https://man.openbsd.org/ssh.1>

[13] Ubuntu system requirements: [online]. [vid. 2018-01-05].
URL <https://help.ubuntu.com/community/Installation/SystemRequirements>

[14] Postgresql system requirements: [online]. [vid. 2018-01-05].
URL <https://www.commandprompt.com/blog/postgresql_mininum_requirements/>

[15] The PostgreSQL Licence: [online]. [vid. 2018-01-05].
URL <https://opensource.org/licenses/postgresql>

[16] Oracle Database Server: [online]. [vid. 2018-01-05].
URL <https://www.oracle.com/database/index.html>

[17] Csv generator: [online]. [vid. 2018-01-05].
URL <http://www.convertcsv.com/generate-test-data.htm>

[18] PG Admin 3 v. 1.22.2: [online]. [vid. 2018-01-05].
URL <https://www.postgresql.org/ftp/pgadmin/pgadmin3/v1.22.2/win32/>

[19] JAVA funkce Startswith(): [online]. [vid. 2018-01-05].
URL <https://www.tutorialspoint.com/java/java_string_startswith.htm>

[20] SQL dotaz LIKE: [online]. [vid. 2018-01-05].
URL <https://www.w3schools.com/sql/sql_like.asp>

[21] Online Polynomial Regression - Xuru: [online]. [vid. 2018-01-05].
URL <http://www.xuru.org/rt/PR.asp#CopyPaste>

[22] Online Polynomial Regression - Arachnoid: [online]. [vid. 2018-01-05].
URL <https://arachnoid.com/polysolve/>

[23] Microsoft Excel: [online]. [vid. 2018-01-05].
URL <https://support.office.com/cs-cz/excel>

[24] Java Database Connectivity: [online]. [vid. 2018-01-05].
URL <https://cs.wikipedia.org/wiki/Java_Database_Connectivity>

[25] PostgreSQL JDBC: [online]. [vid. 2018-01-05].
URL <https://jdbc.postgresql.org/>

[26] C++: [online]. [vid. 2018-01-05].
URL <https://cs.wikipedia.org/wiki/C%2B%2B>