Masarykova univerzita
Fakulta informatiky


# Porovnávacie testy databáz založených na rôznych paradigmách

Bakalárska práca

**Soňa Mastráková**

Brno, podzim 2013

## Prehlásenie

Prehlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracovala samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používala alebo z nich čerpala, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Soňa Mastráková

**Vedúci práce:** RNDr. Pavel Minařík

# Poďakovanie

Touto cestou by som sa chcela poďakovať vedúcemu bakalárskej práce, RNDr. Pavlovi Minaříkovi, za umožnenie vypracovania tohto zadania, ako i za jeho čas a odborné vedenie práce, rovnako ako Mgr. Martinovi Juřeňovi za jeho trpezlivosť, názory a cenné rady pri vypracovávaní. Ďalej ďakujem Bc. Michalovi Kimlemu za jeho neustálu ochotu a pomoc počas štúdia a Mgr. Borisovi Parákovi za pomoc pri zabezpečovaní testovacieho prostredia. V neposlednom rade by som sa chcela poďakovať svojim rodičom za skvelú podporu nielen počas štúdia na fakulte a tiež priateľovi za jeho podporu a pochopenie.

# Zhrnutie

Cieľom tejto bakalárskej práce je porovnanie rôznych databázových systémov. Na začiatku je poskytnutý historický pohľad na evolúciu databázových systémov. Následne sú predstavené a vysvetlené hlavné rozdiely medzi databázovými modelmi. Praktická časť je rozdelená na 2 časti - v prvej časti sa porovnávajú vybrané databáze odlišného paradigmatu podľa dostupnosti dokumentácie, konfigurácie, možnosti správy atď. V druhej časti sú vytvorené a spustené Java aplikácie (pre každý databázový systém iná aplikácia), z ktorých sa získajú výsledky pre každý testovaný dotaz. V závere sa tieto výsledky navzájom porovnávajú.

## Kľúčové slová

SQL, NoSQL, testovací nástroj, výkon, DBMS, Java, záverečná práca

# Obsah

# 1 Úvod

Slovo databáza [13] je definované ako množina štruktúrovaných dát alebo informácií, uložených v počítačovom systéme takým spôsobom, že počítačový program alebo človek môže jednoducho pristúpiť k ich obsahu, prípadne ho meniť a spravovať.

Spája sa jednoznačne s rozširujúcim sa využívaním informačných technológií v bežnom živote. Takmer všade dochádza k nepriamemu styku s databázami. Málokto z bežných užívateľov si však uvedomuje dôležitosť predovšetkým výberu typu datábaze a jej databázového systému (ďalej DBMS[1]), ktorý nám umožňuje s ňou manipulovať, vzhľadom na jej následovné použitie v praxi.

Databáze sú rozlišované podľa tzv. modelov, ktoré určujú typ databáze (ako je tomu napríklad u distribuovaných databáz). [3] Každý z týchto modelov má špecifické spôsoby, či už uskladnenia dát, komunikácie medzi nimi, prípadne viacerými databázami. Výber vhodného modelu záleží na rôznych aspektoch ako je prostredie, v ktorom bude daná databáza používaná, či účely, na aké má slúžiť. Napríklad v poslednej dobe masívne sa rozširujúce sociálne siete preferujú databáze vhodné na veľký objem dát (najlepšie škálovateľná databáza) a zároveň rýchle vykonanie požiadavku. Naopak nejaký menší internetový obchod by si vystačil i s databázou, ktorá by bola menej škálovateľná a pre menší objem dát, nakoľko by nedochádzalo k až takému masívnemu prísunu nových dát, s ktorými by si musela vedieť poradiť.

Na type databáze môžu byť závislé aj rôzne výskumy či projekty. Vhodným príkladom je aktuálne využívanie dokumentových databáz pre prácu s korpusmi (súbormi obsahujúce texty prirodzeného jazyka), ktorých vývoj prebieha priamo na Fakulte Informatiky Masarykovej Univerzity.

Práca zoznamuje čitateľa najprv so základnými typmi databázových modelov a postupne prechádza k predstavovaniu už konkrétnych zástupcov vybraných modelov a k ich stručnému porovnaniu na základe prvotného použitia. Následne poskytuje informácie o samotnom testovaní, na čo sa testovanie zameriava i akým spôsobom to bude realizované. Práca je realizovaná v prostredí operačného systému Linux, distribúcia Fedora. Súčasťou zadania sú i testovacie dáta, detailnejšie rozobrané v prílohe C.

Praktická časť bakalárskej práce zahŕňa určenie si testovaných dotazov vzhľadom na vybrané oblasti a vytvorenie testovacích nástrojov pre konkrétne databázové systémy v jazyku Java. Následne budú systémy otestované za rovnakých prevádzkových podmienok. Posledným bodom praktickej časti je zozbieranie výsledkov, ich spracovanie a vyvodenie záveru vzhľadom na výkon databáz pri danom type dát. V závere je zhodnotená možnosť ďalšieho využitia a prínosnosti výsledkov.

1. DBMS = Database Management System, (v preklade: systém riadenia báze dát)

# 2 Stručná história vývoja databázových systémov

Historické hľadisko v tejto práci je dôležité pre základný prehľad o spôsobe a čase vývoja jednotlivých modelov, ktorých bližšie špecifické vlastnosti sú popísané v kapitole 3.

Skôr než sa databázové systémy [20] začali vyvíjať, fungovali flat-file systémy (prosté databázové systémy). Išlo prevažne o súbory typu `.txt`, v ktorých boli záznamy (vety) a ich položky, kde každý záznam pripadol na nový riadok.

Medzi najstaršie modely v komerčnej sfére patrí hierarchický model databáz. Information Management System (IMS) [18], vyvinutý v IBM a North American Northwell Company, bol jednou z prvých hierarchických databáz. V roku 1970 sa stal svetovou vedúcou hierarchickou databázou. Viac o špecifických vlastnostiach tohto modelu nájdete v kapitole 3.1.

V polovici 60.rokov sa začal vyvíjať sieťový model. Išlo o nadstavbu hierarchického, kde bolo možné zobrazovať komplexnejšie prepojenie medzi dátami pomocou vzťahov. V tej dobe vyvinula firma General Electric systém s názvom Integrated Data Store (IDS), založený na báze sieťového modelu.

Príchodom systémov došlo i k vzniku troch jazykov – *Data Definition Language* (dáta definujúci jazyk, ďalej len DDL) umožňujúci administrátorom definovať schéma danej databáze, *subschéma DDL* umožňujúca programom definovať časti databáze a *Data Manipulation Language* (jazyk manipulujúci s dátami, ďalej len DML) umožňujúci manipuláciu s dátami.

Hierarchickému i sieťovému modelu chýbala väčšia nezávislosť medzi dátami, mali minimálny teoretický základ a tak v roku 1969 vznikol úplne nový, na predchádzajúcich modeloch nezávislý, relačný model. Na základe neho sa spustil projekt s názvom System R, pod vedením IBM San Jose Research Laboratory v Kalifornii, ktorý mal dokázať implementovateľnosť tohto modelu. V projekte vznikol známy dotazovací jazyk Structured Query Language (jazyk štruktúrovaných požiadavkov, ďalej len SQL) a následne v roku 1980 i prvé DBMS založené na relačnom modeli – konkrétne DB2, SQL/DS a známy Oracle od Oracle Corporation.

Z relačného modelu sa postupom času vytvoril ďalší samostatný celok – objektový model, ktorý sa môže ešte ďalej rozdeľovať na Object-Oriented a Object Relational (objektovo relačný). V prípade objektového modelu vznikol taktiež samostatný dotazovací jazyk Object Query Language (jazyk požiadavkov nad objektami, ďalej len OQL), ďalej popísaný v časti 3.1.

Medzi najnovšie databáze v oblasti databázových systémov patria NoSQL databáze, ktorých názov sa vzťahuje na niekoľko nových, rôznych databáz – dokumentovo orientované, kľúč-hodnota, grafové a XML. Začali sa vyvíjať v 21.storočí a ide o databáze, ktoré nie sú založené na báze relačného modelu.

Stručná schéma vzniku jednotlivých databáz je zobrazená v prílohe B.

# 3 Prehľad a vlastnosti jednotlivých paradigmát

## 3.1 Rozdelenie databáz podľa dátových modelov

Dátový model (často tiež nazývaný databázový model) [10] definuje základnú štruktúru databáze. Účelom modelu je poskytnúť detailné zobrazenie spôsobu riadenia systému, či pre zhodnotenie vhodnosti využitia v rámci projektu, alebo na jeho lepšie pochopenie.

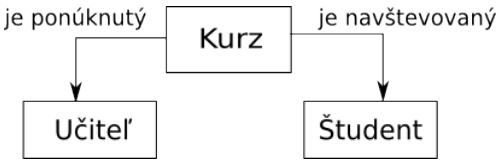
Zahŕňa v sebe [18]:

- statické vlastnosti (objekty, atribúty a vzťahy)
- dynamické vlastnosti (operácie a pravidlá definujúce nový stav databáze vzhľadom na zmeny v nej)
- integritné pravidlá nad objektami a operáciami

### 3.1.1 Hierarchický model

Hierarchický model bol široko využívaný v prvých databázových systémoch. Dáta rovnako ako sieťový model uchováva vo forme záznamov. Hlavnými rozdielmi oproti sieťovému modelu sú:

- stromová štruktúra uchovávania dát – na vrchole sa nachádza 1 koreňový záznam, z ktorého vychádzajú ďalšie záznamy
- každý záznam má maximálne jednoho rodiča – uchováva iba one-to-one a one-to-many asociácie pri prechádzaní zhora nadol (viď 3.1), pričom asociácie nie sú možné horizontálne alebo diagonálne, ale iba vertikálne (čiže nie je žiadna priama spojitosť medzi záznamami na rovnakej úrovni v strome)




**Obr. 3.1:** Zobrazenie príkladu hierarchickej štruktúry modelu (prevzaté z [10])

Tieto rozdiely zároveň spôsobujú, že sieťový model je flexibilnejší a komplexnejší než hierarchický.

Známymi reprezentantmi hierarchického modelu sú IMS od firmy IBM[1] a Windows Registry. V rámci testovania bude využívaný GT.M založený na báze systému MUMPS,

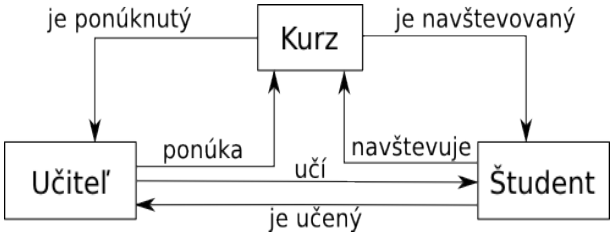
1. viac nájdete na http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/ibmims/

ktorý má aj svoj vlastný programovací jazyk, taktiež s názvom MUMPS. Viac je o nich popísané v kapitole 4.1.

### 3.1.2 Sieťový model

Patrí medzi prvé vyvinuté a používané modely. Dáta uchováva vo forme záznamov, ktoré sú organizované ako súbor ľubovoľných sietí.

V sieťovom modeli môže mať každý detský záznam (resp. člen) ľubovoľný počet rodičov (resp. vlastníkov), takže je možná kardinalita many-to-many(viď 3.2). Vzťahy sa označujú ako väzby (linky).




**Obr. 3.2:** Názorné zobrazenie štruktúry sieťového modelu, prevzaté z [10]

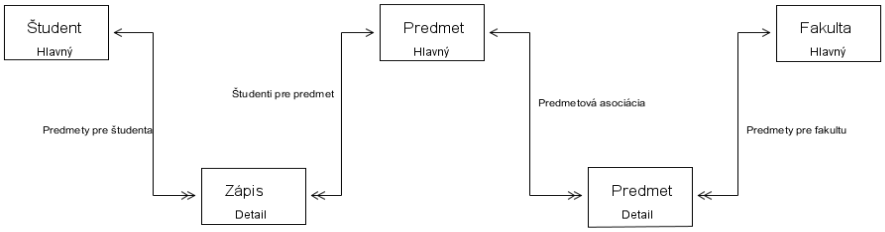
Rozlišujeme 3 základné druhy sieťových modelov [14]:

- jednoduchý – môže byť vzťah iba one-to-one alebo one-to-many.
- komplexný – máme aspoň 1 kardinalitu typu many-to-many. V praxi tento druh predstavuje náročnú implementáciu. Dalo by sa povedať, že v podstate neexistuje, pretože sa rieši rozdelením každého many-to-many vzťahu na dva rozdielne one-to-many vzťahy.
- limitovaný – každý súbor je rozdelený na hlavný a transakčný súbor, pričom vzťah medzi nimi vedie z hlavného ku transakčnému, viď 3.3.

Sieťový model má tiež svoj dotazovací jazyk. Nakoľko patrí medzi CODASYL databáze (tj. tie, ktoré sú vytvorené CODASYL konzorciom), prislúcha mu CQLF[2].

Známymi reprezentantmi tohto modelu sú IDMS[3], TurboIMAGE[4]. V rámci testovania však nebude použitý žiadny sieťovy DBMS, keďže sa táto práca zaoberá výhradne open source databázami.

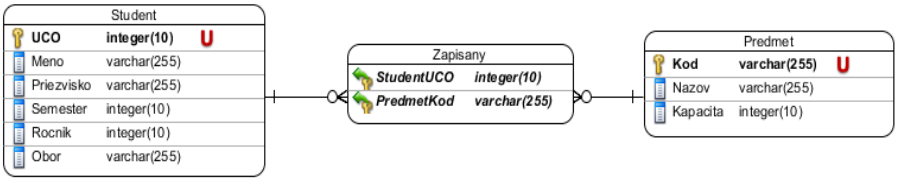
2. CQLF = CODASYL Query Language, Flat
3. IDMS = Integrated Database Management System, viac nájdete na http://en.wikipedia.org/wiki/IDMS
4. viac nájdete na http://www.hp.com/products1/evolution/e3000/download/turboimage_db.pdf




**Obr. 3.3:** Zobrazenie limitovaného sieťového modelu (prevzaté z [14])

### 3.1.3 Relačný model

Relačný model sa začal vyvíjať koncom 60.rokov a dodnes sa zdokonaľuje. Hlavnou charakteristickou vlastnosťou tohto modelu je uchovávanie dát v tabuľkách, kde stĺpce obsahujú dáta rovnakého typu a riadky predstavujú závislosť dát v rámci množiny. Čiže relačný model je množina tabuliek, pričom každá z nich má unikátny identifikátor. Pomocou tabuliek sa dajú zobraziť aj vzťahy medzi nimi. Jednoduchým príkladom relačného modelu je obrázok 3.4.




**Obr. 3.4:** Relačný model databázy uchovávajúcej informácie o študentoch a predmetoch a vzťahoch medzi nimi

Na to, aby bola každá transakcia v rámci modelu spoľahlivá, boli vytvorené tzv. ACID vlastnosti[5]. Tieto vlastnosti nám zabezpečujú validitu prenosov v relačných modeloch, kde prenos predstavuje hocijakú logickú operáciu nad dátami. Podrobnejší význam je:

- Atomicity – každá transakcia buď skončí celá úspešne, alebo ak sa čo i len časť nevykoná, tak sa transakcia skončí bez zmien v databáze
- Consistency – každá transakcia, ktorá sa vykoná, nesmie narušiť konzistenciu databáze, ak by ju narušiť chcela, tak sa databáza obnoví do pôvodného stavu (tj. do stavu pred vykonaním transakcie)

5. ACID = Atomicity, Consistency, Isolation, Durability

- Isolation – transakcie sa nesmú medzi sebou rušiť (napríklad ak by jedna transakcia pristupovala k dátam, s ktorými práve manipuluje druhá transakcia)

- Durability – žiadna transakcia, ktorá bola vykonaná nad databázou, sa nesmie stratiť, jej výsledky musia byť okamžite uložené

Takisto sa pri relačnom modeli objavujú tzv. normálové formy, ktoré napomáhajú minimalizovať redundanciu a závislosti. Konkrétne vznikli 3 normálové formy[19], ktorých autorom je E.F.Codd[6]:

- 1.NF – dátový záznam R je v 1.NF, pokiaľ sa v ňom nevyskytujú opakujúce sa skupiny položiek

- 2.NF – dátový záznam R je v 2.NF, pokiaľ je v 1.NF a každá nekľúčová položka v R je plne funkčne závislá na každom kandidátnom kľúči z R

- 3.NF[7] – dátový záznam R je v 3.NF, pokiaľ je v 2.NF a každá nekľúčová položka v R je netranzitívne závislá na každom kandidátnom kľúči z R

Normálových foriem je samozrejme viac, medzi najznámejšie patrí aj BCNF[8], taktiež zameraná na funkčnú závislosť a 4.NF, ktorá je odvodená od BCNF, ale na rozdiel od nej je zameraná na tzv. viacznačnú závislosť.

V rámci vývoja relačného modelu vznikol i jeho vlastný dopytovací jazyk SQL3.5. Slúži na definíciu dát a manipuláciu s nimi a ich kontrolu. Je teda kombináciou DDL, DML a DCL[9]. Obsahuje základné konštrukcie:

- DDL – `SELECT, INSERT, DELETE, UPDATE`

- DML – `CREATE, ALTER, DROP`

- DCL – `GRANT, REVOKE`

Zobrazenie využitia konštrukcií v praxi viď príklad 3.5.

Pre jazyk SQL existujú aj rôzne rozšírenia, ktoré ho doplňujú o funkcie procedurálneho jazyka, napríklad MySQL, SQL/JRT(na podporu Java kódu v AQL[10] databázach), PL/SQL (ProceduralLanguage/SQL), PL/pgSQL (PL/PostgreSQL) a ďalšie.

---

6. E.F.Codd je vynálezcom relačného databázového modelu pre databázové systémy
7. 3. NF má 2 definície, v tejto práci popisujem verziu od E.F.Codda z roku 1971
8. BCNF = Boyce-Codd Normal Form, v preklade: Boyce-Coddova normálová forma
9. DCL = Data Control Language, (v preklade: jazyk kontrolujúci dáta)
10. AQL = Annotation Query Language

```
SELECT meno
FROM student
WHERE rocnik=3
ORDER BY priezvisko;
```

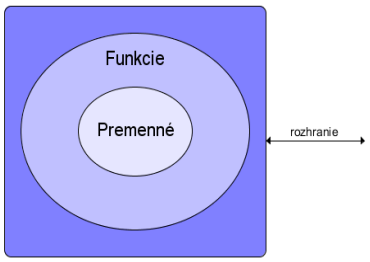
**Obr. 3.5:** Štruktúra požiadavku: výpis mien všetkých študentov, ktorí sú v 3.ročníku, zoradených vzostupne abecedne podľa priezviska

Známymi reprezentantmi tohto modelu sú MS SQL Server[11], Oracle[12], MySQL[13], MS Access[14]. V rámci testovania bude využívaný však PostgreSQL, ktorý je zároveň aj zástupcom OODBMS.

### 3.1.4 Objektový model

Ako bolo spomenuté vyššie v kapitole 2, objektový model sa vyvinul z relačného. Platia preň teda podobné pravidlá ako pre relačný model, s výnimkou objektového paradigmatu a to tak, že informácie sú reprezentované formou objektov, podobne ako v objektových programovacích jazykoch. Jednotlivé objekty v sebe zahŕňajú dáta (premenné) a metódy (funkcie) na manipuláciu s dátami.

Tie, ktoré obsahujú rovnaký typ premenných aj funkcií, sú zoskupované do tried, ktoré predstavujú definíciu typu objektu. V prípade, že nejaký objekt A potrebuje dáta od objektu B, môže ich získať vyvolaním metódy objektu B cez jeho rozhranie, viď 3.6.




**Obr. 3.6:** Zobrazenie štruktúry objektu v OODBMS

Štruktúra OO[15] databáz je vysoko voliteľná, pretože nie je presne určená ako pri ostatných modeloch.

11. viac nájdete na https://www.microsoft.com/sqlserver/cs/cz/
12. viac nájdete na http://www.oracle.com/us/products/database/overview/index.html
13. viac nájdete na http://www.mysql.com/
14. viac nájdete na http://office.microsoft.com/cs-cz/access/
15. OO = objektovo orientovaný

OO databáze majú svoj dotazovací jazyk. Je podobný ako SQL, takisto pracuje s konštrukciami `SELECT`, `FROM`, `WHERE` a inými, ale s tým rozdielom, že OQL pracuje s objektami. Podporuje polymorfizmus a môže byť súčasťou iných objektových programovacích jazykov. Zobrazenie využitia konštrukcií v praxi viď 3.7.

```
SELECT p.name
FROM Persons p
WHERE p.vek=25
```

**Obr. 3.7:** Štruktúra požiadavku: výpis mien všetkých osôb, ktoré majú 25 rokov, kde Persons môže predstavovať trieda Student aj trieda Teacher

Vzhľadom na vzrastajúcu komplexnosť databáz rozlišujeme [18]:

- objektovo-orientovaný databázový model – ide o objektový model, dáta sú reprezentované vo forme objektov
- objektovo-relačný databázový model (nazývaný aj rozšírený relačný model) – ide o relačný model, ktorý je rozšírený o možnosť využívania objektov, snaží sa o premostenie relačného modelu a objektovo-orientovaného modelu, čiže integruje databázu s objektovo orientovanými programovacími jazykmi.

Známymi reprezentantmi objektového modelu sú Objectivity/DB[16], InterSystems Caché [17], db4o[18], VelocityDB[19]. V rámci testovania bude využívaný však PostgreSQL, ktorý je zároveň aj zástupcom RDBMS.

## 3.2 NoSQL databáze

Význam [21] slova NoSQL DBMS niektorí prezentujú ako nerelačné databázové systémy, iní zase tvrdia, že je to „Not Only SQL“[20]. Tak či onak je to dnes označenie pre databáze, ktoré nenásledujú známe relačné princípy, a ktoré sa pasujú so škálovateľnosťou (nárazovým spracovaním väčšieho objemu dát). Nie je to však názov pre konkrétny systém, ide o viacero produktov zastrešených pod týmto názvom.

U NoSQL databáz sa vyskytuje pojem MapReduce. Ide o model od spoločnosti Google, ktorý umožňuje distribuované, paralelné spracovanie dát na počítačových clusteroch. Popis chovania modelu sa dá rozdeliť podľa funkcií `map` a `reduce` z pohľadu

16. viac nájdete na http://www.objectivity.com/products/objectivitydb/
17. tvorí výnimku, pretože nemá OQL, má implementované real-time mapovanie objektu na relácie a pre dotazovanie sa používa SQL, viac nájdete na http://www.intersystems.com/cache/
18. viac nájdete na http://www.db4o.com/about/
19. viac nájdete na http://velocitydb.com/About.aspx
20. v preklade: nie len SQL

funkcionálneho programovania. V prvej časti hlavný uzol príjme požiadavok od užívateľa a rozpošle ostatným uzlom funkciu `map`, ktorí ju následne spracujú a odošlú výsledok naspäť do hlavného uzlu. Ten v druhej časti, keď má dostatok prijatých výsledkov na spracovanie, môže (ale nie je to povinné) aplikovať agregačné funkcie `reduce` na spracovanú množinu dát. Dôležitým faktom je, že všetky funkcie v prvej fázi môžu bežať nezávisle a výsledky posúvať priamo do druhej fáze. Štruktúru modelu demonštruje obrázok 3.8.

Zároveň sa spolu s vlastnosťou škálovateľnosti vyvinul aj tzv. CAP teorém[21], popisujúci 3 základné požiadavky kladené na distribuované systémy v prostredí nestabilnej siete, z ktorých sú vždy splnené len dva, viď obrázok 3.9.

Význam jednotlivých častí CAP teorému:

- Konzistencia – všetci klienti pracujú v rovnakom čase s tou istou verziou dát
- Dostupnosť – systém funguje za každých podmienok a užívateľ od neho obdrží vždy nejakú odpoveď
- Tolerancia k rozdeleniu – systém je schopný naďalej pracovať aj po prerušení komunikácie s niektorými uzlami

### 3.2.1 Databáze typu kľúč-hodnota

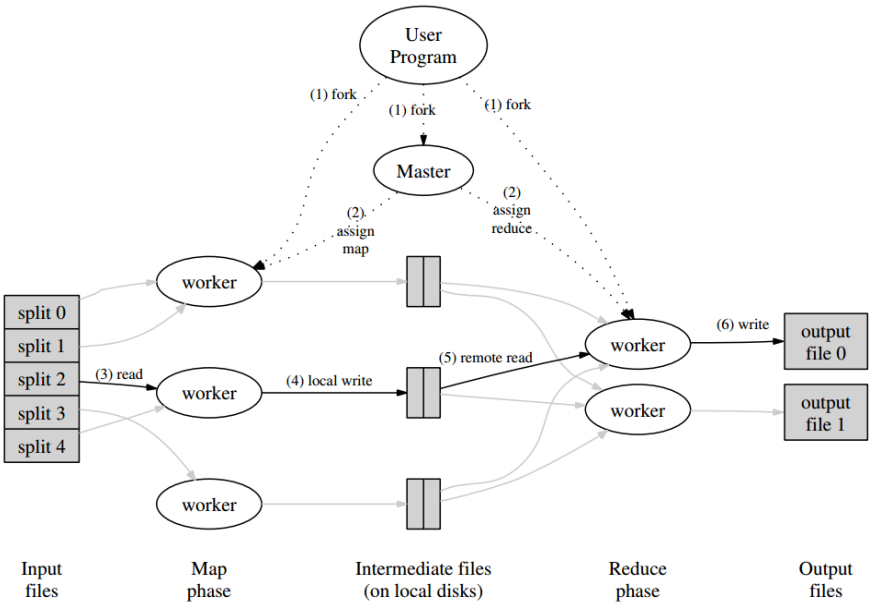
NoSQL model s názvom kľúč-hodnota sa podobá či už na známy `HashMap`[22], alebo na asociované pole. V tomto modeli ide o množinu dvojíc, pričom každá z nich má svoj unikátny kľúč, podľa ktorého je dvojica identifikovateľná v rámci množiny a ľubovoľne štruktúrovanú hodnotu. Dátové štruktúry tohto typu bývajú obľúbené, pretože sú efektívne pre spracovanie veľkého množstva nekomplexných dát.

Páry kľúč-hodnota bývajú spracovávané rôzne – niektoré sú ukladané do pamäte, iné podporujú možnosť uloženia dát na disk. Medzi známy spôsob patrí i cache, ktorého účelom je znížiť počet prístupov na disk pomocou „zachytenia“ často používaných dát v pamäti.

Známymi reprezentantmi tohto druhu sú Voldemort[23], Redis[24]. V rámci testovania nebude použitý žiaden DBMS tohto typu, pretože štruktúra zvolených dát a následných dotazov sa zamieriava prevažne na DBMS so značne odlišným účelom.

---

21. CAP = Consistency (v preklade: konzistencia), Availability (v preklade: dostupnosť), Partition Tolerance (v preklade: tolerancia k rozdeleniu)
22. štruktúra využívaná v objektovom programovacom jazyku Java
23. viac nájdete na http://www.project-voldemort.com/voldemort/
24. viac nájdete na http://redis.io/




**Obr. 3.8:** Model MapReduce od spoločnosti Google (prevzaté z [7])

### 3.2.2 Dokumentové databáze

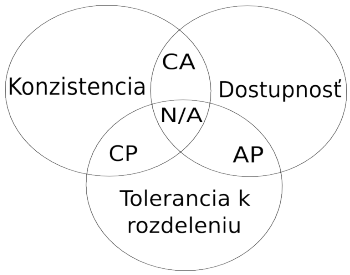
Dokumentové databáze sa dajú jednoducho popísať práve porovnaním s relačnými databázami.

V relačnom modeli máme vždy na začiatku vytvorenú nejakú tabuľku, obsahujúcu stĺpce s vopred definovanými typmi, ktoré sa do nich budú vkladať. Bez tohto kroku by sme tabuľku v relačnej databáze nemohli zostrojiť. Aj v prípade, že by v danom riadku niektorého stĺpca nemala byť žiadna hodnota, doplní sa hodnota `null`.

V dokumentových databázach postupujeme úplne odlišne. Tento typ databáze sa skladá z kolekcií, ktoré sú tvorené vlastnými dokumentami. Dokumenty následne uchovávajú už konkrétne záznamy.

Princíp fungovania tohto systému sa dá vysvetliť následovným príkladom: v dokumentových databázach môžeme v jednom kroku do dokumentu pridať záznam s atribútom MENO a hodnotou typu `String`, v ďalšom kroku môžeme do toho istého dokumentu pridať záznam s atribútom VEK a hodnotou typu `Int`, pričom oboje patrí do toho istého dokumentu a sú to valídne dáta. Ak by sme niečo podobné chceli vykonať nad relačnou tabuľkou, išlo by to jedine v prípade, že by obsahovala oba atribúty (aj VEK aj MENO) a v prvom prípade by bol VEK typu `null` a v druhom MENO typu `null`.

Podstatou toho celého je, že pri tomto type databáz nemáme presne určenú štruktúru dokumentu ani presný typ hodnôt, aké je potrebné vkladať. Predstavujú akýsi prechod




**Obr. 3.9:** CAP teorém zobrazený pomocou Vennovho diagramu (prevzaté z [1])

z key-value databáz na komplexnejšiu štruktúru, v ktorej sa nenachádza žiadny striktný postup pri vytváraní databáze a jej hodnôt.

Známymi reprezentantmi tohto druhu sú Couchbase[25], CouchDB[26], Cassandra[27]. V rámci testovania bude využívaný MongoDB.

### 3.2.3 Grafové databáze

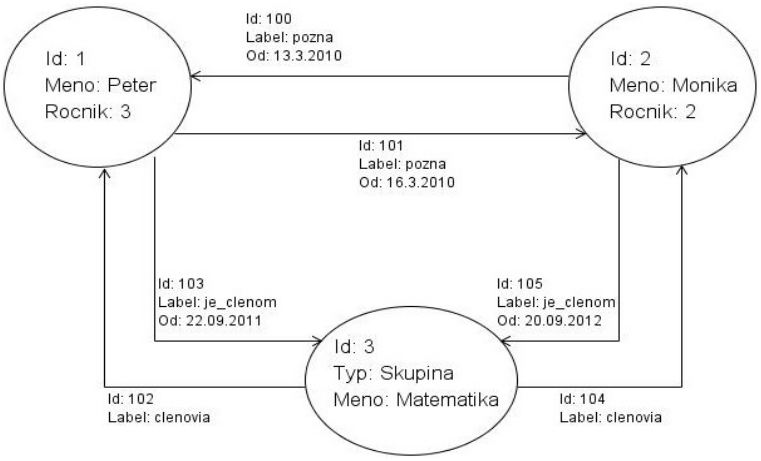
Grafové databáze [12] sú dalšou alternatívou k relačným databázam. Celá podstata je založená na teórii grafov. V tomto prípade vymieňame tabuľky a jej stĺpce za uzly, hrany a ich vlastnosti, pričom platí:

- každý uzol je samostatná, plne nezávislá jednotka, predstavujúca nejakú entitu
- každá hrana je závislá od existencie 2 uzlov
- vlastnosti sú atribútmi uzlov

Samotnú štruktúru názorne demonštruje obrázok 3.10. Tento druh databází je vhodný na zobrazenie komplexných informácií a vzťahov medzi nimi tak, ako je to i v skutočnosti (napr. zobrazenie zmýšľania pri riešení problému a prepojenie medzi viacerými problematikami), na rozdiel od relačných databáz, kde všetko jednoducho ukladáme do tabuliek.

Grafové databáze sa využívajú najmä v prostredí [2] sociálnych sietí (*Facebook, Google* i *Twitter* ich v sebe majú zabudované), sieťového a cloudového manažmentu (telefónne spoločnosti), bioinformatiky a mnohých iných odvetví.

25. viac nájdete na http://www.couchbase.com/couchbase-vs-couchdb
26. viac nájdete na http://couchdb.apache.org/
27. viac nájdete na http://cassandra.apache.org/




**Obr. 3.10:** Zobrazenie grafových DB

Známymi reprezentantmi tohto druhu sú GraphBase[28], OrientDB[29], AllegroGraph[30]. V rámci testovania bude využívaný Neo4j.

### 3.2.4 XML databáze

Tento druh NoSQL databází sa začal rozvíjať s príchodom štandardu XML. Databázové systémy podporovali ukladanie aj vyhľadávanie dát v tomto formáte, avšak bolo to veľmi neefektívne, keďže išlo o vyhľadávanie textu, bez indexácie.

Rozlišujú sa 2 druhy – XML-enabled (XML-podporujúce) a Native XML (natívne XML) databáze. Hlavný rozdiel medzi nimi je, že XML-enabled databáze na vstupe dostanú dáta v XML formáte, zapíšu ich v typickej forme (napríklad ako tabuľku) a na výstupe sú opäť dáta v XML formáte. Native XML databáze podľa modelu delíme na:

- textovo orientované – XML dokumenty sú ukladané ako text
- modelovo orientované – XML dokumenty sa prevedú na objektový model, s ktorým následne v databáze pracujeme

Mnohé XML databáze majú podporu dotazovacieho jazyka XQuery. Ide o rozšírenie jazyka XPath o tzv. FLWOR[31], ktorý podporuje iteráciu a viazanie premenných na prechodné výsledky. Zobrazenie využitia jazyka v praxi viď 3.12.

28. viac nájdete na http://graphbase.net/What.html
29. viac nájdete na http://www.orientdb.org/
30. viac nájdete na http://www.franz.com/agraph/allegrograph/
31. FLWOR = FOR, LET, WHERE, ORDER BY, RETURN

```
<?xml version="1.0" encoding="UTF-8" ?>
<students>
 <student category="bachelor degree">
  <meno> James </meno>
  <priezvisko> Bond </priezvisko>
  <uco> 007 </uco>
  <semester> 0 </semester>
  <rocnik> 0 </rocnik>
  <obor> Professional </obor>
 </student>
</students>
```

**Obr. 3.11:** Príklad formátu XML dát

```
doc("school.xml")/students/student[rocnik=3]
```

**Obr. 3.12:** Štruktúra požiadavku

Známymi reprezentantmi tohto druhu sú MarkLogic Server[32], BaseX[33], Sedna[34]. V rámci testovania bude využívaný eXist-db.

## 3.3 Rozdelenie databáz podľa spôsobu uskladnenia dát

Databáze môžeme rozdelovať podľa viacerých kritérií, či už podľa modelov na základe ktorých sú vytvárané, spôsobu ukladania dát, typu systému a mnohých ďalších. V tejto kapitole sa zameriavam práve na druhú variantu, čiže rozdelenie databáz na On Disk a In-Memory.

Z historického hľadiska nie je ani jeden z uvedených druhov ničím novým. Väčšina známych databáz sú druhu On Disk, ku ktorým je možnosť rozšírenia práve na In-Memory DB. Samotné In-Memory databáze sa začali prvýkrát používať už v 80.rokoch minulého storočia v oblasti telekomunikácií. Prvým oficiálnym komerčným In-Memory DBMS sa stal TimesTen, dnes vlastnený spoločnosťou *Oracle*.

### 3.3.1 In-Memory databáze

In-Memory[22] databáze sú tie, ktoré primárne dáta uskladňujú v operačnej pamäti. Ich architektúra je odlišná od klasických relačných databáz v tom, že všetky dáta sa

32. viac nájdete na http://www.marklogic.com/what-is-marklogic/enterprise-nosql/
33. viac nájdete na http://basex.org/products/
34. viac nájdete na http://www.sedna.org/

nachádzajú v pamäti Random Access Memory (ďalej len RAM), tj. algoritmy berú do úvahy túto zmenu a sú odľahčené o hash tabuľky alebo stromovú štruktúru. Väčšina In-Memory databáz je založená na relačnom modeli (podpora ACID vlastností), konkrétne z radu SQL databáz (tj. dotazovanie pomocou jazyka SQL). K dátam[15] pristupujeme pomocou špecifických indexov. Pevný disk sa používa len na zaznamenávanie aktualizácií v databáze.

In-Memory databáze sa využívajú najmä v oblastiach, kde je predovšetkým dôležitý čas odozvy (odpovede) na požiadavok vyvolaný užívateľom (oblasť telekomunikácií, sietí). Vhodným príkladom využívania In-Memory databáze je aj svetoznámy mimoburzový trh NASDAQ[35] vlastnený spoločnosťou *The Nasdaq Stock Market, Inc.*

Známymi reprezentantmi tohto druhu databáz sú tiež SAP HANA[36], solidDB[37], VoltDB[38]. V rámci testovania budú využívané MemSQL a CSQL.

## 3.4 Zhrnutie

Následujúca tabuľka 3.1 stručne a prehľadne porovnáva markantné rozdiely medzi jednotlivými modelmi databáz popísaných v častiach 3.1 a 3.2. Nezahŕňa porovnávanie In-Memory a On Disk databáz.

| | **hierarchický** | **sieťový** | **relačný** | **objektový** | **kľúč-hodnota** | **dokumentové** | **grafové** | **XML** |
|---|---|---|---|---|---|---|---|---|
| **štruktúra** | stromová | grafová | tabuľková | tabuľková | tabuľková | dokumentová | grafová | stromová |
| **základná jednotka databáze** | záznam | záznam | tabuľka | tabuľka | záznam | záznam | uzol | záznam |
| **dotazovací jazyk** | M | CQLF | SQL | OQL | - | - | CYPHER, GREMLIN | xQuery |
| **stále vo vývoji** | nie | nie | áno | áno | áno | áno | áno | áno |
| **známi reprezentanti** | MUMPS, GT.M | IDMS, TurboIMAGE | PostgreSQL, MySQL | db4o, VelocityDB | Voldemort, Redis | Cassandra, MongoDB | OrientDB, Neo4j | BaseX, eXist-db |

**Tabuľka 3.1**

35. NASDAQ = National Association of Securities Dealers Automated Quotations
36. viac nájdete na http://www.saphana.com/welcome
37. viac nájdete na http://www-01.ibm.com/software/data/soliddb/
38. viac nájdete na http://voltdb.com/

# 4 Testovacie prostredie

V tejto kapitole popisujem presné parametre hardvéru, na ktorom sa testovanie uskutočnilo. Ďalej jednotlivé databázové systémy, ich vlastnosti a prvotné porovnanie na základe používania, ako dostupnosť dokumentácií, náročnosť administrácie a iné. Na konci kapitoly je opísaný spôsob priameho testovania dotazov, konkrétne testovacieho nástroja, ktorý bol pre tento účel vytvorený, a takisto samotné dáta.

## 4.1 Databázové systémy

V následujúcej časti sú popísané jednotlivé DBMS, popis ich vlastností, a priebeh inštalácie spolu s nastavením konfigurácie, vhodnej vzhľadom na účel dátabáze. Pretože sa jedná o rôzne typy systémov, pri každom z nich je popísaný spôsob daného nastavenia konfigurácie.

### 4.1.1 GT.M

MUMPS je názov jednak programovacieho jazyka a jednak samotného databázového systému. Postupom času sa začal ekvivalentne používať pojem M technológie. Najnovšia implementácia databáze MUMPS s názvom M21 však nie je open source, a z toho dôvodu nebude použitá v tejto práci. Pre testovanie bol zvolený systém GT.M, V60001. Po stiahnutí súboru ho stačí rozbaliť a spustiť inštalačný skript `./configure`. Presný popis inštalácie a nastavenia premenných je popísaný v priloženom návode v prílohe A. Spustenie databáze sa vykoná príkazom `gtm`.

Štruktúra databáze je rovnaká ako v prípade hierarchického modelu. Informácie [4] sú skladované v dátových štruktúrach (konkrétne poliach), ktoré môžu byť označené ako lokálne alebo globálne. V prípade programovacieho jazyka MUMPS pojem globálna premenná znamená, že jej hodnota sa uloží permanentne (nie na RAM), aby bola po ukončení aplikácie prístupná aj iným aplikáciám. Takéto premenné sa označujú znakom '^', napríklad `^Student("Rocnik")`. Všetky ostatné vytvorené premenné sú vytvárané dynamicky bez deklarovania a sú lokálne. Ďalším dôležitým faktom je, že jazyk MUMPS má jednotný univerzálny typ dát, ktorý sa odvíja od toho, čo práve potrebujeme.

Systém GT.M je v súčasnosti napojiteľný cez Javu pomocou JavaM API, Python a jazyk C, no všetky sú neustále vo vývoji. Na testovanie sa však využíva JavaM API len sčasti, dôvod a popis testovania je vysvetlený nižšie v kapitole 4.3.

Pre administráciu databáze je k dispozícii samotný systém pomocou príkazu `gtm`.

### 4.1.2 PostgreSQL

Na oficiálnych stránkach tohto systému (www.postgresql.com) sú dostupné jednak PostgreSQL Live CD so zameraním na Fedoru, a jednak binárne balíky pre jednotlivé platformy (Windows, Linux, Mac OS X i BSD). Pre Linuxovú platformu je systém plne dostupný (Fedora, Debian, Ubuntu, SuSE a iné). Postup pre inštaláciu sa nachádza taktiež na stránkach, takže inštalácia základných balíkov prebehla bez akýchkoľvek problémov. V dobe písania práce je najnovšia dostupná stabilná verzia 9.2.

Štruktúra databáze je rovnaká ako v prípadne relačného modelu, tj. dáta sú nahrávané do tabuliek, s tou výnimkou, že je nad nimi možné uplatniť objektovo orientované programovacie jazyky.

V prípade PostgreSQL je možnosť využívania PL/pgSQL, čo je procedurálna forma jazyka SQL. Jeho výhodou oproti samotnému SQL jazyku je, že môžeme zoskupovať viacero dotazov a priamo aj výpočtov do jednotlivých skupín v databázovom serveri, čo sa významne odráža na zvýšení výkonnosti pri ich vykonávaní. PostgreSQL má širokú škálu aplikačných rozhraní. V súčasnosti sú v základnej distribúcii dostupné pre jazyky pgSQL, Perl, Python a je možné si ju rozšíriť aj o Javu, PHP, Ruby a ďalšie zo známych jazykov. V tejto práci som sa rozhodla pre programovací jazyk Java, hlavne preto, že bol jediným dostupným jazykom u všetkých testovaných systémoch.

Na administráciu databáze existuje mnoho platforiem a nástrojov, z toho 2 obľúbené:

- *phpPgAdmin* – napísaný v jazyku PHP, ktorý slúži na administráciu serveru
- *pgAdmin* – open source administratívna platforma pre PostgreSQL, cez ktorú je možné písať jednoduché SQL dotazy až po samotný vývoj databáz.

Čo sa týka nastavenia konfigurácie, nachádza sa v súbore `/var/lib/postgres/data/postgresql.conf` a pre účel tejto práce budú zmenené hodnoty len v prípade `shared_buffers`, ktorá má podľa dokumentácie obsahovať približne 25% z dostupnej vyhradenej operačnej pamäte. Okrem tejto premennej nenastanú žiadne zmeny, pretože testovanie bude prebiehať na lokálnom serveri bez prístupu zvonka, čo je v PostgreSQL predvolene nastavené v hodnote `listen_addresses = 'localhost'` (v opačnom prípade by hodnota mohla byť nastavená na * alebo na konkrétnu adresu IP).

### 4.1.3 MongoDB

Filozofiou MongoDB bola najmä:

- škálovateľnosť a s ňou spojené nečakané hromadné prístupy k databáze
- rýchlosť vývoja – prispôsobiť databázu tak, aby bol vývoj elegantný a jednoduchý

- komplexnosť dát – polymorfizmus, štruktúrované a neštruktúrované dvojice

Na oficiálnych stránkach (www.mongodb.org) je prehľadný postup na stiahnutie a inštaláciu potrebného balíka. V dobe písania tejto práce je k dispozícii najnovšia verzia 2.2.3. Všeobecne sa odporúča verzia pre 64-bitovú architektúru, pretože 32-bitová je limitovaná len do 2GiB dát. Inštalácia prebiehala bez problémov, samotný server sa po inštalácii spustí príkazom `mongod` a pripojenie príkazom `mongo`.

Štruktúra databáze je dokumentová. Jednotlivé záznamy sú uchovávané vo forme dokumentov, ktoré sú skladované v kolekciách. Tie sú zoskupované vo svojich databázach. V prípade potreby je možné využiť tzv. sharded cluster, ktorý slúži na zminimalizovanie čakacej doby na výsledok dotazu pri veľkom množstve dát. Sharded cluster sa skladá z 3 častí - tzv. *shard* (ide mongod inštanciu alebo ich zoskupenie, pričom obsahuje nejakú podmnožinu kolekcii), tzv. *config serverov* (ide o mongod inštancie, ktoré obsahujú metadata o cluster-i) a *inštancií mongos* (slúžia na komunikáciu s určitými shardami). Pre presnejší popis je priložený obrázok 4.1. Toto škálovanie je však momentálne použiteľné len na rozdeľovanie kolekcií v databáze, nie na rozdeľovanie dát z jednej kolekcie.

Dotazovacím jazykom je v tomto prípade JSON[1], ktorého cieľom je oddeliť dáta od kódu.
Napr.

```
db.parts.find(_id:"Q33")
```

vyhľadá a vypíše všetky dáta, ktorých `_id = Q33`. Dotazy sa skladajú z :

**`nazov_databaze.nazov_kolekcie.funkcia(dáta)`**

Ďalej sa v tomto systéme vyskytuje i BSON[2], pomocou ktorého sa v MongoDB uchovávajú dáta/objekty.

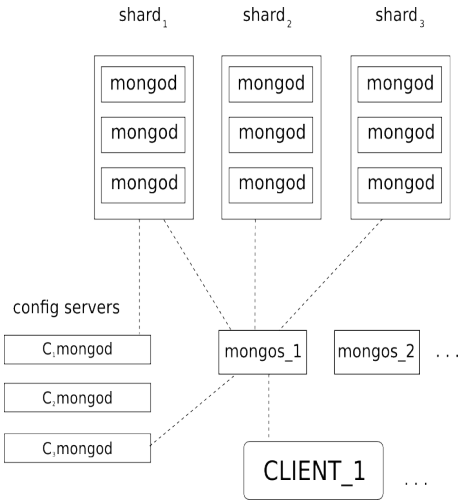
MongoDB má silnú podporu pripájania z vonkajšieho prostredia. Existujú knižnice pre jazyk C, C++, C#, Java, Perl, PHP, Python a mnohé iné. Na oficiálnych stránkach sa pre každý z nich nachádza dostatočná dokumentácia pre prvotné pripojenia. V tejto práci sa prikláňam k využitiu programovacieho jazyka Java.

Správa databáze môže prebiehať priamo z príkazového riadka, zavolaním príkazu `mongo`, prípadne cez admin klienta s grafickým rozhraním ako je Edda, Fang of Mongo, či UMongo, poskytovaní tretími stranami. Dostupní sú i mnohí iní klienti, bližšie popísaní priamo na stránkach spoločnosti MongoDB.

V prípade tohto databázového systému je konfigurácia jednoduchá. Jediné, čo bolo treba nastaviť, je `bind_ip` na hodnotu `127.0.0.1`. Vzhľadom na charakter poskytnutých testovacích dát nie je potrebné vytvárať sharded cluster, pretože sú zoskupené do jedinej spoločnej kolekcie, zobrazenej v kapitole 5.1.

---

1. JSON = JavaScript Object Notation, (v preklade: JavaScriptová objektová notácia)
2. BSON = Binary JSON, (v preklade: binárne zakódovaná JavaScriptová objektová notácia)




**Obr. 4.1:** Grafická prezentácia sharded cluster-a, prevzaté z [6]

### 4.1.4 Neo4j

Tento databázový systém má nenáročnú inštaláciu i spustenie. Oficiálna stránka (www.neo4j.org) poskytuje dostatočné informácie pre prvotné použitie. Po stiahnutí a rozbalení súboru je potrebné len nastaviť konfiguráciu, a spustiť server príkazom `./neo4j start` z priečinka `bin/`. Dokumentácia k Neo4j je dobre dostupná a prehľadná, navyše sú na internete mnohé diskusné fóra zamerané na túto databázu.

Štruktúra databáze je založená na grafovom modeli, bližšie popísanom v kapitole 3.2. Fungovanie tejto databáze spočíva vo vytváraní tzv. vzorov. Vzor predstavuje prepojenie uzlu `a` s uzlom `b` pomocou vzťahu `r`, čiže `a-[:r]->b`.

Dotazovacích jazykov pre tento systém je dostupných viacero. V rámci tejto práce bol spočiatku uprednostnený jazyk CYPHER pred jazykom GREMLIN, jednak pre lepšiu názornosť a jednoduchšie pochopenie pre nového užívateľa, a jednak pre lepšiu výkonnosť nad nižšie uvedenými testovanými dotazmi, vzhľadom na ich jednoduchú štruktúru v prípade prostredia bez indexácie. Jazyk CYPHER má veľmi podobnú štruktúru ako jazyk SQL a využíva existenciu vzorov na vykonávanie dotazov nad uzlami a väzbami medzi nimi. Nakoniec sa však vykonávanie dotazov realizuje čisto prostredníctvom Java Core API, ktoré síce nemá implementované všetky funkcie databáze (na rozdiel od jazyka CYPHER), no výkonnosť tohto API je niekoľkonásobne vyššia v porovnaní s výkonnosťou jazyka CYPHER.

Možnosť pripojenia k databáze z externého programu pomocou konektorov je vysoko podporovaná. V súčasnej dobe je k dispozícií viac než 20 driverov priamo na oficiálnej stránke aj so stručným návodom na pripojenie. Medzi najznámejšie podporované jazyky

patria REST API, Ruby, PHP, Java, Python, Haskell. Ako už bolo vyššie spomenuté, v tejto práci používam jazyk Java, pre ktorý je v tomto prípade i priama podpora pre vývojové prostredia ako sú Eclipse či NetBeans.

Pre administráciu databáze sú dve možnosti - cez príkazový riadok spustením príkazu `./neo4j-shell` alebo využitím web admina cez internetový prehliadač na adrese `localhost:7474/webadmin/`.

Nastavenie konfigurácie prebieha v priečinku `conf/neo4j-server.properties`, kde je defaultne nastavený localhosting. Ďalším dôležitým nastavením je pridelenie pamäte. Podľa odporúčaní dokumentácie má byť pridelené následujúce množstvo pamäte:

- pre uzly: `predpokladany_pocet_uzlov * 9 bytov`
- pre väzby: `predpokladany_pocet_vztahov * 33 bytov`

### 4.1.5 eXist

Táto databáza patrí medzi známe XML databáze, častokrát označované i ako dokumentové. Samotná inštalácia nebola náročná, v prípade problémov bola k dispozícií dokumentácia na oficiálnej stránke (www.exist-db.org). Spustenie inštalácie prebieha zadaním príkazu `java -jar existdb.jar -console` pre mód v príkazovom riadku. Spustenie servera sa vykoná spustením skriptu `./bin/startup.sh`.

Ako už bolo vyššie spomenuté, eXist patrí medzi XML databáze, to znamená, že dáta uchováva vo formáte XML. Samotná databáza sa skladá z kolekcií a dokumentov. V porovnaní s relačnými systémami sa dá povedať, že kolekcie predstavujú databázu a dokumenty predstavujú tabuľky.

Použitým dotazovacím jazykom je xQuery, bližšie popísaným v kapitole 3.2.

Samotná databáza vychádza z Javy, jej podpora na pripojenie z Javy je teda zrejmá. Okrem toho ponúka možnosť pripojenia pomocou REST API, XML:DB, či Fluent-API.

Administrácia systému je možná jednak cez príkazový riadok príkazom `./bin/client.sh` alebo cez internetový prehliadač pomocou Java Admin Client-a.

V rámci nastavenia konfigurácie dochádza k zmenám v hodnote `-Xmx`, ktorá definuje maximálne množstvo využitej pamäte a `<db-connection>`, ktorá predstavuje cca. $\frac{1}{3}$ z hodnoty `-Xmx`. Bližšie parametre nastavenia databáze sú presne popísané v prílohe A.

### 4.1.6 MemSQL

Systém patrí medzi In-Memory databáze, opísané vyššie v kapitole 3.3. Na stiahnutie inštalačného programu je však nutná registrácia, napriek tomu, že systém je open source. Pred spustením inštalácie je k dispozícii overovací skript, pomocou ktorého sa dá zistiť, či je dostupné množstvo pamäte vyhovujúce, aké knižnice treba dosťahovať a pod. Samotná inštalácia prebehla ľahko. Dokumentácia na oficiálnych stránkach

(http://www.memsql.com/) je dostatočná a prehľadná. Taktiež bolo potrebné pre úspešné spustenie nainštalovat MySQL databázu a server, cez ktoré sa napája na MemSQL.

Ako už z názvu vyplýva, MemSQL patrí do radu SQL databází, tj. je založená na relačnom modeli. Dáta teda uchováva už tradične vo forme tabuliek.

Takisto dotazovací jazyk je SQL. Oproti PostgreSQL sa líši len v drobných syntaktických obmenách.

Pripojenie k tejto databáze prebieha pomocou JDBC, pričom spojenie sa realizuje cez MySQL knižnice, keďže MemSQL je protokolovo kompatibilný s MySQL.

Administrácia MemSQL môže prebiehať jednak priamo cez príkazový riadok alebo MemSQL-admina.

Konfigurácia systému je defaultne nastavená na localhosting. Maximálne množstvo pamäte pre tabuľky je nastavené zhruba na 90% z celkového množstva pamäte, v prípade testovania v rámci tejto práce bude prenastavená na 80%. Pre zvýšenie výkonu počas testovania bolo potrebné vypnúť tzv. durability (tj. pravidelné commitovanie akýchkoľvek zmien na disk) priamo v Javovej aplikácii. V praxi sa toto nastavenie neodporúča, pretože dochádza k riziku stratenia dát pri zlyhaní systému, bez možnosti ich obnovy.

### 4.1.7 CSQL

Posledným testovaným systémom je CSQL z radu SQL. Tento systém v porovnaní s ostatnými je najnáročnejší na inštaláciu vzhľadom na prostredie, v ktorom testovanie prebehlo. Pred samotnou inštaláciou je nutné doinštalovať množstvo knižníc, zväčša v 32 bitových verziách. Detailnejší postup a zoznam knižníc je dostupný v prílohe A, v ktorom sú spísané všetky potrebné kroky. Dokumentácia k tomuto systému je slabšia i v rámci jeho používania. Je rozdelená na 3 časti - užívateľskú, programátorskú a caché dokumentáciu.

Štruktúra databáze je založená na relačnom modeli, čiže prostredie predstavujú tabuľky obsahujúce záznamy v riadkoch.

Dotazovacím jazykom je SQL, popísaným vyššie v kapitole 3.5. V tomto prípade sa synakticky takmer nelíši od jazyka, ktorý bol použitý pri MemSQL.

Na pripojenie k databáze sa už tradične využíva JDBC konektor. Okrem toho je možné pripojenie cez ODBC, PHP driver či SQLAPI. V rámci tejto práce je využívaná možnosť pripojenia cez JDBC konektor.

Správa databáze sa vykonáva cez nástroj s názvom csql, ktorý sa po vykonaní všetkých krokov obsiahnutých v návode v prílohe A spustí príkazom `csql`. Po spustení sa zmení prompt príkazového riadka na „`CSQL>`".

Nastavenie konfigurácie je pomerne jednoduché. Rovnako ako pri MemSQL je potrebné vypnúť automatické commitovanie po 1 dotaze, príkazom
`setAutocommit(false)` priamo v Javovej aplikácii. Okrem toho sa ešte nastavuje parameter `MAX_SYS_DB_SIZE`, ktorý určuje veľkosť systému obsahujúceho metadata,

pričom musí byť násobkom parametra `PAGE_SIZE`, predvolene nastaveným na 8 MiB a `MAX_DB_SIZE`, ktorý určuje veľkosť systému, kde sa nachádzajú užívateľské dáta, pričom tiež musí byť násobkom parametra `PAGE_SIZE`. Oba tieto parametre (`MAX_SYS_DB_SIZE` i `MAX_DB_SIZE`) sú defaultne nastavené na 10 MiB.

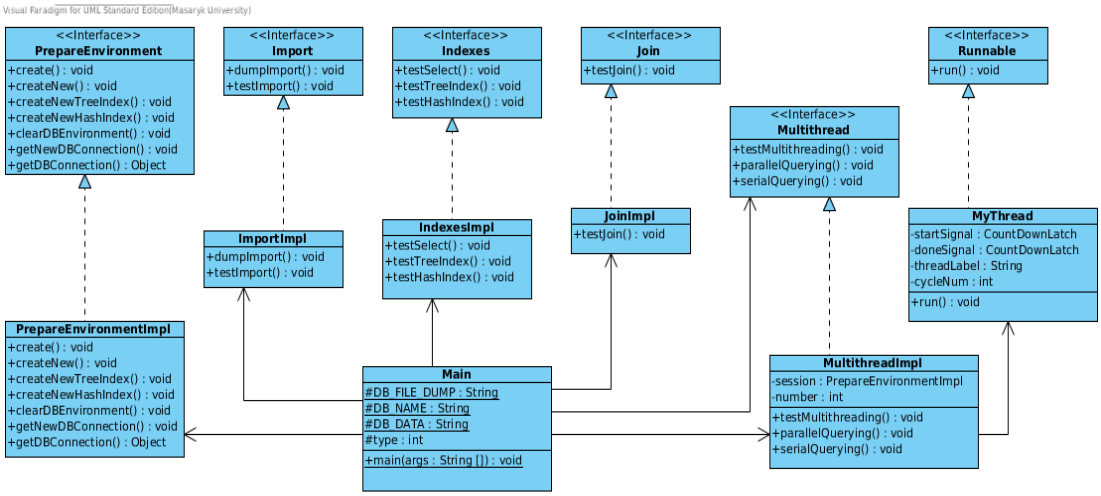
## 4.2 Hardvér

Samotné testovanie prebiehalo na vzdialenom virtuálnom stroji, pripojenie bolo riešené pomocou SSH protokolu bez grafického rozhrania. Na stroj bolo vymedzených dokopy 110 GiB pevného disku, z toho určité množstvo zabral operačný systém Linux, distribúcia Fedora 18 (64bit). Ďalej bolo vymedzených 80 GiB RAM pamäte (vzhľadom na náročnosť niektorých dotazov) a počas testovania boli využívané 2 CPU.

## 4.3 Popis prostredia, prostriedkov na testovanie

Poskytnuté testovacie dáta majú 597 MiB a sú vo formáte `.csv`. Táto veľkosť súboru obsahuje cca 8 miliónov záznamov, k testovaniu bol z nich využitých len 1 milión, kvôli časovej náročnosti niektorých testovaných dotazov.

Testovací nástroj bol vytváraný s dôrazom na podobnú štruktúru programu. Po prvotnej analýze požadovaných testovaných dotazov a spôsobov pripájania jednotlivých databáz, bola zvolená následujúca štruktúra, názorne zobrazená pomocou diagramu tried 4.2. Veľkou výhodou pri tvorbe takmer identických aplikácií bola možnosť využitia jednotného programovacieho jazyka.




**Obr. 4.2:** Znázornenie základnej funkcionality jednotlivých tried v programe

Na spustenie nástroja je potrebné mať nainštalovanú najnovšiu verziu JDK[3] a JRE[4] (v čase písania tejto práce ide o JDK 7 a JRE 7). Pri spúšťaní nástroja na databázovom systéme CSQL bolo potrebné nainštalovať JDK verziu pre 32-bitové operačné systémy. Všetky potrebné knižnice na úspešné spustenie sú dostupné v priečinku `lib/`, v prípade potreby je v súbore `build.xml` zakomentovaná časť, ktorá tieto knižnice pribalí k súboru `.jar`.

Všetky nástroje sa spúšťajú pomocou svojho lokálneho skriptu `run.sh` umiestneného v priečinku `run/`, ktorý spustí svoj `.jar` súbor s `main()` metódou. V prípade odkomentovania časti v `build.xml`, ktorá pribaľuje potrebné knižnice k `.jar` súboru, je potrebné odkomentovať i v `run.sh` skripte riadok súšťajúci novo vytvorený `.jar` súbor (typicky začínajúci názvom `Combined-nazov_suboru.jar`).

Po spustení skriptu sa zjaví ponuka s možnými príkazmi nad danou databázou. Všetky ďalšie operácie sú na výber prostredníctvom tejto ponuky, ktorá užívateľovi dáva na výber, čo môže spustiť. Typicky je to:

- príkaz `create` slúži na vytvorenie prostredia na testovanie, po zvolení tohto príkazu dochádza k výberu typu prostredia, do ktorého budeme importovať dáta (bezindexová tabuľka/tabuľka so stromovým indexom/tabuľka s hashovým indexom)
- príkaz `import` slúži na testovanie rýchlosti sekvenčného importu dát, pri väčšine systémoch po zvolení tohto príkazu nedochádza k žiadnej ďalšej voľbe
- príkaz `dump` slúži na testovanie rýchlosti hromadného importu dát, po zvolení tohto príkazu nedochádza k žiadnej ďalšej voľbe
- príkaz `indexes` slúži na testovanie vplyvu indexácie na rýchlosť vyhľadávania požadovaných hodnôt v databáze, po zvolení tohto príkazu dochádza k výberu typu testovaného dotazu (zameranie na tree/hash index)
- príkaz `join` slúži na testovanie rýchlosti vyhľadávania pri spájaní napr. 2 tabuliek, po zvolení tohto príkazu nedochádza k žiadnej ďalšej voľbe
- príkaz `multi` slúži na testovanie rýchlosti vykonania dotazu pri sériovom vykonávaní a pri paralelnom dotazovaní, po zvolení tohto príkazu dochádza k výberu typu vykonania dotazu (parallelné spustenie 10 vlákien/sériové vykonanie dotazu 10 krát)

Takisto v prípade vypisovacích príkazov (tj. `indexes`, `join` a `multi`) je možné zvoliť, či chceme zobraziť výstup príkazu, voľbou `yes/no`.

3. JDK = Java Development Kit
4. JRE = Java Runtime Environment

Výnimku tvorí nástroj na testovanie systému GT.M. Vzhľadom na dostupnosť Javových API a konektorov bol uprednostnený priamy prístup k databáze cez Javu, využitím systémových volaní ktoré sa pripoja na databázový systém a spustia vopred nahratú rutinu v adresári, obsahujúcu potrebné funkcie na otestovanie. Táto cesta testovania bola zvolená hlavne preto, že v dostupnom JavaM API nie sú implementované všetky potrebné funkcie a zároveň pre snahu o zachovanie jednotnosti programovacieho jazyka. Spustenie programu prebieha takisto pomocou skriptu `run.sh`, ktorý bol súčasťou adresára obsahujúceho `JavaM.jar`. Po spustení skriptu je dostupná odlišná ponuka (chýba `create` príkaz, ktorý je nahradený príkazom `import`). Je to z toho dôvodu, že v prípade GT.M nebolo potrebné nastavovať pripojenie k databáze pomocou `create`, pretože v rámci súboru `javam-master/` priloženého v prílohe A, máme k dispozícii pripojenie k GT.M databáze v podobe jej inštancie, s ktorou automaticky môžeme pracovať. Typ prostredia bol určený pri vkladaní dát do databáze. Presný popis je dostupný v DVD prílohe A.

Rovnako i aplikácie pristupujúce k systémom MongoDB a CSQL, využívajú priamy prístup pomocou systémových volaní pri príkazoch `indexes`, `join` a `multi`, pretože výsledky merania pri využívaní dostupných API neboli valídne.

Dohromady bolo vytvorených 7 nástrojov (pre každý systém jeden). Pri každom z nich je priložený krátky návod na použitie, pretože nie každý systém podporoval všetky testované druhy dotazov, a tým pádom pri niektorých nie sú implementované všetky vyššie spomenuté funkcie.

## 4.4 Zhrnutie

Stručné porovnanie v tabuľke 4.1:

| | GT.M | PostgreSQL | MongoDB | Neo4j | eXist-db | MemSQL | CSQL |
|---|---|---|---|---|---|---|---|
| **najnovšia stabilná verzia** | V60001 | 9.2 | 2.4 | 1.8 | 2.0 | 1.9 | 3.3 |
| **databázový model** | hierarchický | relačný | dokumentový systém | grafový systém | XML systém | relačný | relačný |
| **úroveň dokumentácie** | stredná | výborná | výborná | výborná | stredná | výborná | slabá |
| **dostupnosť informácií** | slabá | výborná | výborná | výborná | stredná | stredná | slabá |
| **náročnosť inštalácie** | náročná | ľahká | ľahká | ľahká | ľahká | ľahká | náročná |
| **API** | JavaM, Python, C | Perl, Python, Java, Ruby a iné | C, C++, Perl, Java, PHP a iné | Ruby, PHP, Java, Haskell a iné | XML:DB, Fluent-API, REST API a iné | JDBC | PHP, SQLAPI, JDBC |

**Tabuľka 4.1**

# 5 Dotazy, vlastnosti, kritériá, metodika

## 5.1 Štruktúra dát

Už z názvu tejto práce vyplýva, že vygenerované dáta budú mať odlišnú štruktúru. V prípade troch SQL databáz sa štruktúra medzi nimi neodlišuje. Zobrazenie daných štruktúr sa nachádza v prílohe C.

Pre každú štruktúru boli vytvorené parserovacie nástroje, ktoré dáta z pôvodného `.csv` súboru upravili do vyššie uvedeného tvaru. Tieto nástroje sú súčasťou priloženého DVD v prílohe A a sú napísané v jazyku Ruby.

Testovanie prebiehalo pre 5.000, 10.000, 100.000, 500.000 a 1.000.000 záznamov. Nižšie uvedené grafy sú zostavené z 2 vytvorených skupín databáz podľa ich výsledkov (tj. skupiny s lepšími a skupiny s horšími výsledkami). Je to z toho dôvodu, že testovaných databáz je príliš veľa nato, aby ich zobrazenie v jednom grafe bolo prehľadné, čitateľné a zároveň prínosné. Dôležitou informáciou je i to, že hodnoty sú zobrazené pomocou logaritmickej mierky, kvôli dobre viditeľnému rozdielu medzi výsledkami. Vzhľadom na stĺpcový charakter výsledných grafov sú výsledky priložené k práci i formou tabuliek v prílohe G.

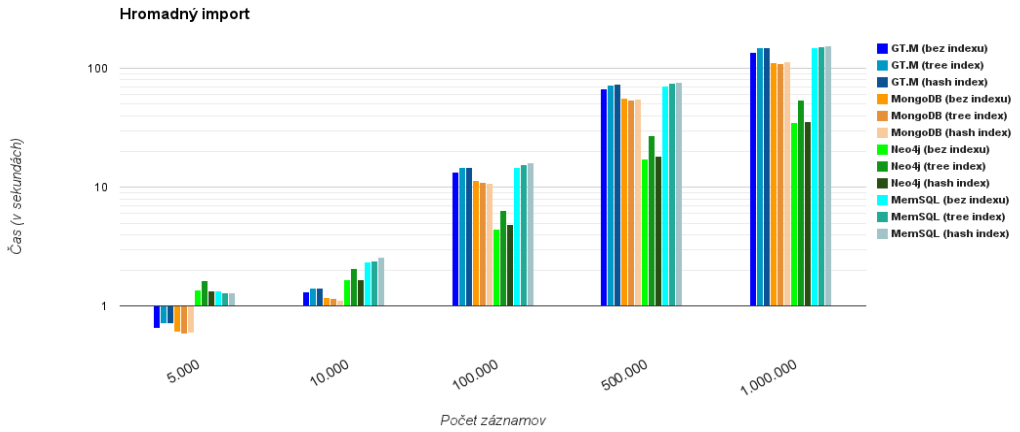
## 5.2 Importy

U jednotlivých systémov boli testované 2 druhy importov:

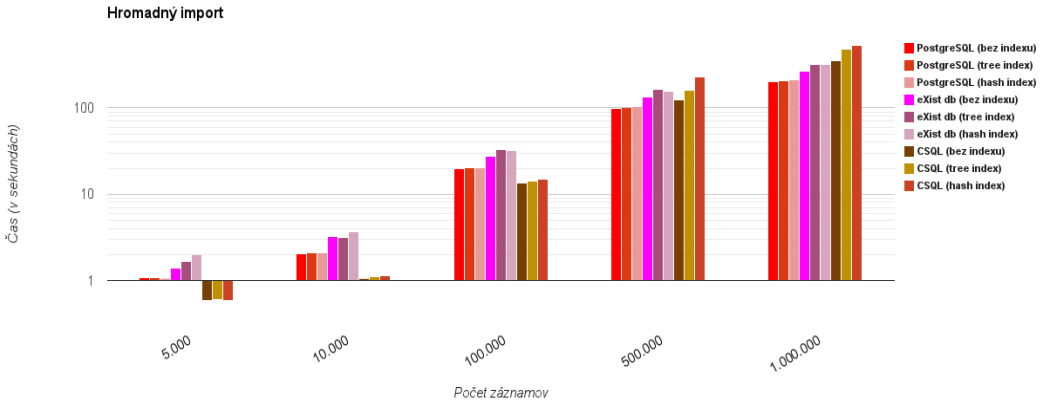
- hromadný import = import vopred naparserovaných dát do požadovanej štruktúry, pričom pri testovaní sa do času potrebného na vykonanie dotazu započítava aj čas potrebný na parserovanie dát
- sekvenčný import = import jednotlivých záznamov do databáze zvlášť pomocou príkazov (napr. `INSERT` pre SQL databáze)

Takisto v prípade, že daný systém podporoval využívanie indexácie, tak sa testoval čas potrebný pre import do prostredia bez indexácie a následne s indexáciou. Používané typy indexácie sú bližšie popísané v podkapitole 5.3. Samotné príkazy v jednotlivých dotazovacích jazykoch sú uvedené v prílohe D.

### 5.2.1 Výsledky testovania

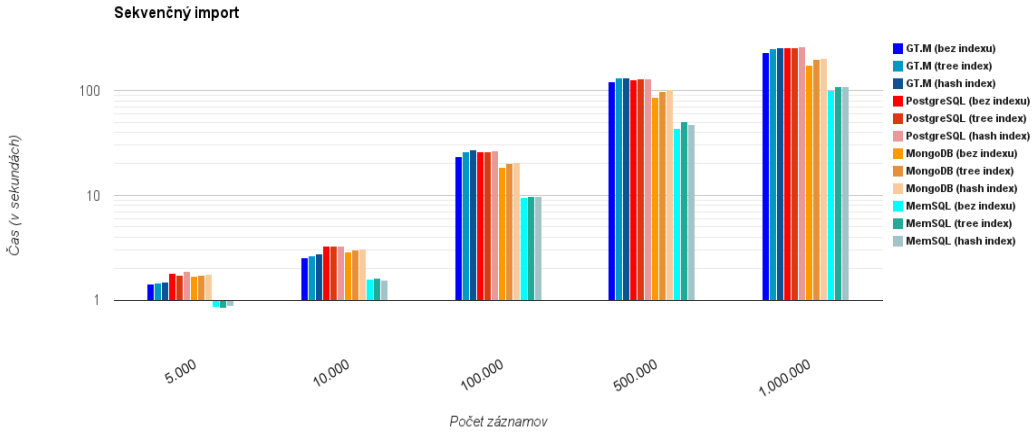



**Obr. 5.1:** Lepšie výsledky hromadného importu

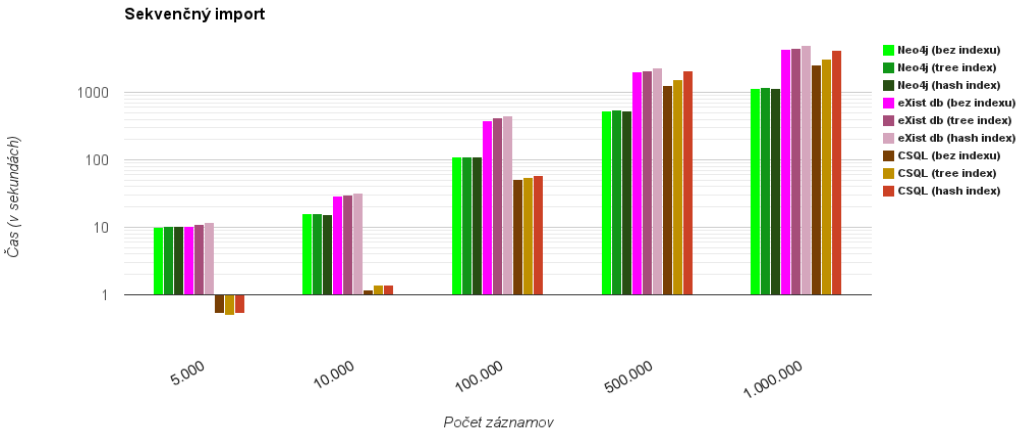



**Obr. 5.2:** Horšie výsledky hromadného importu

Pri testovaní hromadného importu bol predpokladaný najlepší čas pre dokumentovú databázu, čo sa potvrdilo pri najmenšom množstve dát. Naopak vzhľadom na dostupné API a štruktúru dát bol prekvapivý výsledok pre grafovú databázu, ktorá v konečnom dôsledku predbehla ostatné systémy, na čom má svoj podiel i fakt, že pre túto databázu nebolo nutné vytvárať samostatný parser. Predpokladaný rozdiel pri importovaní do bezindexového a indexového prostredia sa potvrdil, takmer u všetkých systémov bolo časovo najnáročnejšie importovať dáta do prostredia s hash indexom.




**Obr. 5.3:** Lepší výsledok sekvenčného importu




**Obr. 5.4:** Horší výsledok sekvenčného importu

Výsledky testovania výkonnosti sekvenčného importu sú v porovnaní s hromadným importom rôznorodé. U niektorých systémov nie je zmena až tak viditeľná, no u iných sa časy importu od seba až niekoľkonásobne líšia. Drvivá väčšina systémov si pri sekvenčnom importe najhoršie poradila s importom do prostredia s hash indexom, ako to bolo i pri hromadnom importovaní dát. Najlepšie zo všetkých systémov tento import zvládal systém MemSQL, či už s indexami alebo bez nich. Naopak najhoršie výsledky dosiahol systém eXist db.

## 5.3 Indexácia

Indexácia v databázových systémoch je významnou vlastnosťou, ktorá značne skracuje čas vykonávania dotazov či vyhľadávanie požadovaných dát. Vybraté boli práve 2 typy na testovanie – Hash index [1] a Tree index[2], pretože tieto typy boli v najväčšej miere zastúpené medzi testovanými systémami.

Dotazy, pomocou ktorých sa zisťovala miera vplývania indexu na čas ich vykonania, boli (napr. pre SQL databáze) následovné:

- pre tree index:

```
SELECT *
FROM transaction
WHERE transferred>500 AND transferred<1000;
```

  Tento druh indexu sa využíva najmä pri vyhľadávaní hodnôt v určitom rozsahu, tj. pri využívaní znakov '<', '>', '≤', '≥', pretože usporiadava indexy do stromovej štruktúry, ktorá je v tomto prípade časovo veľmi výhodná.

- pre hash index:

```
SELECT *
FROM transaction
WHERE destination_ip_address=12469945614
ORDER BY net_flow_start;
```

  Tento druh indexu sa využíva pri vyhľadávaní presných hodnôt, tj. pri využívaní znaku '=', pretože na vyhľadávanie využíva hashovaciu tabuľku, ktorá je v tomto prípade vhodnou variantou vzhľadom na čas.

V tejto podkapitole sú popísané spôsoby vytvárania testovaného indexu, čo je prínosné najmä v prípade NoSQL databáz. Na jej konci je prehľadná tabuľka 5.1 indexov, ktoré sú podporované jednotlivými systémami. Ich bližší popis sa nachádza v prílohe E.

### 5.3.1 GT.M

Tento systém nepodporuje indexáciu. V dokumentácii sa nachádza zmienka o indexácii v zmysle, že ak má programátor záujem o indexáciu, musí si ju sám zabezpečiť vytvorením novej globálnej premennej, ktorá bude obsahovať hodnoty, ktoré by chcel naindexovať. Následne v prípade, že budú tieto hodnoty využívané v nejakom dotaze, sa vyhľadajú najprv v novej globálnej premennej.

V prípade tohto databázového systému bolo potrebné využiť navyše index
`^StartIndx(net_flow_start, destination_ip_address)=transaction`

1. v preklade: hashovací index
2. v preklade: stromový index

pri testovaní druhého dotazu kvôli zoradeniu výslednej množiny podľa hodnoty `net_flow_start`, pretože samotný systém nato nemá žiadnu funkciu. Tento spôsob vyhľadávania sa aplikoval už pri testovaní bezindexového prostredia, a vzhľadom na jeho štruktúru to bol najvýhodnejší spôsob vyhľadávania i pre indexové prostredie, a tak je štruktúra druhého dotazu v oboch prostrediach identická.

### 5.3.2 PostgreSQL

Indexácia sa definuje po vytvorení novej tabuľky pomocou konštrukcie:

- pre B-tree index (je ekvivalentom pre tree index)

```
CREATE INDEX nazov_indexu
ON nazov_tabulky (indexovana_hodnota_v_tabulke);
```

- pre Hash index

```
CREATE INDEX nazov_indexu
ON nazov_tabulky
USING hash (indexovana_hodnota_v_tabulke);
```

PostgreSQL má k dispozícii ďalšie 3 druhy indexov (GiST[3], SP-GiST[4] a GIN[5] index), popísané v prílohe E.

### 5.3.3 MongoDB

V prípade MongoDB sa indexácia definuje po vytvorení/napojení sa na kolekciu, príkazom:

- pre Tree index

```
db.collection.ensureIndex({"meno_indexu"});
```

- pre Hash index

```
db.collection.ensureIndex({"meno_indexu", "hashed"});
```

Je nutné poznamenať, že samotný systém automaticky indexuje nové záznamy pomocou ním vytváranej premennej `_id`. Okrem toho má však k dispozícii široké spektrum indexácie (Unique, Primary, Compound, Multikey, Sparse, Geospatial a Text index) na nami vybrané hodnoty, popísané v prílohe E.

---

3. GiST = Generalized Search Tree (v preklade: generalizovaný vyhľadávací strom)
4. SP-GiST = Space-partitioned GiST (v preklade: priestorovo-rozdelený GiST)
5. GIN = Generalized Inverted Index (v preklade: generalizovaný invertovaný index)

### 5.3.4 Neo4j

Indexáciu vytvárame v závislosti na spôsobe importovania dát.

V prípade prvotného importu (tj. pomocou BatchInserter-a, popísaného v prílohe D) vytvárame indexáciu pomocou BatchInserterIndexProvider-a, ktorý predstavuje Lucene index, uspôsobený na využitie počas hromadného importu. Pre čo najlepší výkon je vhodné dotazy vykonávať v takom poradí, v akom je striedanie zapisovacích (vytváranie uzlov/vzťahov) a čítacích (vyhľadávanie uzlov/vzťahov) úkonov čo najmenšie.

V prípade sekvenčného importu využívame Apache Lucene indexáciu, ktorá každú hodnotu berie ako reťazec znakov a tak boli v Jave zadané v tvare

```
new ValueContext("hodnota").indexNumeric()
```

pre úspešné vyhľadávanie rozsahu hodnôt.

### 5.3.5 eXist db

Tento systém, takisto ako MongoDB, ponúka široký výber indexácie (Structural, Legacy Fulltext, Fulltext, NGram, Spatial index), viac o nich v prílohe E. V tomto prípade je testovaným indexom Range index pre oba typy dotazov. Funguje takým spôsobom, že podľa určených hraničných hodnôt vyhľadáva najprv tie, ktoré sú do maximálnej hodnoty a potom tie, ktoré sú nad minimálnou, prípadne naopak. V prípade exaktnej hodnoty `destination_ip_address`, na ktorej vyhľadávanie slúžil Hash index v iných systémoch, môžeme aplikovať i Range index v eXist db, pričom bude aplikovaný rovnaký postup vyhľadávania s tým rozdielom, že hraničné hodnoty budú zhodné (tj. bude nimi vyhľadávaná hodnota `destination_ip_address`).

Nevšednosťou oproti ostatným systémom je, že konkrétny typ indexácie neurčujeme pri vytváraní nového dokumentu (tj. nie v kóde testovacieho nástroja), ale explicitne priamo v konfiguračnom súbore databázového systému (`conf.xml`) alebo v konfiguračnom súbore konkrétnej kolekcie (`collection.xconf`), kde v prípade dedičnosti medzi kolekciami dochádza taktiež k dedeniu ich konfiguračného súboru. V ňom sú indexy vyjadrené pomocou `modulov`, kde následne špecifikujeme, na ktoré uzly sa indexácia bude viazať. Súbor použitý v rámci tejto práce sa nachádza v prílohe A.

### 5.3.6 MemSQL

Rovnako ako u väčšiny SQL databáz i u MemSQL dochádza k definovaniu indexácie po vytvorení tabuľky príkazom:

- pre Skip List index (je ekvivalentom pre Tree index)

```
CREATE INDEX nazov_indexu
USING BTREE ON nazov_tabulky (hodnota_v_tabulke);
```

- pre Hash index

```
CREATE INDEX nazov_indexu
USING HASH ON nazov_tabulky (hodnoty_v_tabulke);
```

V prípade tohto systému bolo nutné nastaviť Hash index na n-ticu, pretože MemSQL vyžaduje v Hash indexe unikátne hodnoty. Vzhľadom na poskytnuté dáta bolo nutné tomuto indexu priradiť až 5 hodnôt (konkrétne ide o `destination_ip_address`, `destination_port`, `net_flow_start`, `source_ip_address` a `source_port`), pomocou ktorých bola splnená podmienka unikátnosti.

### 5.3.7 CSQL

Podobne ako u MemSQL dochádza k definovaní indexov okamžite po vytvorení novej tabuľky:

- pre Tree index

```
CREATE INDEX nazov_indexu
ON nazov_tabulky (indexovana_hodnota_v_tabulke) TREE;
```

- pre Hash index

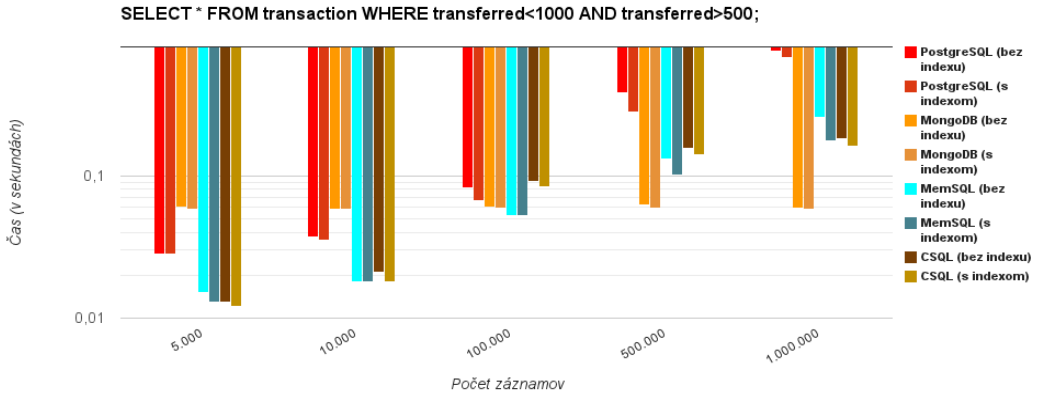
```
CREATE INDEX nazov_indexu
ON nazov_tabulky (indexovana_hodnota_v_tabulke);
```

### 5.3.8 Zhrnutie

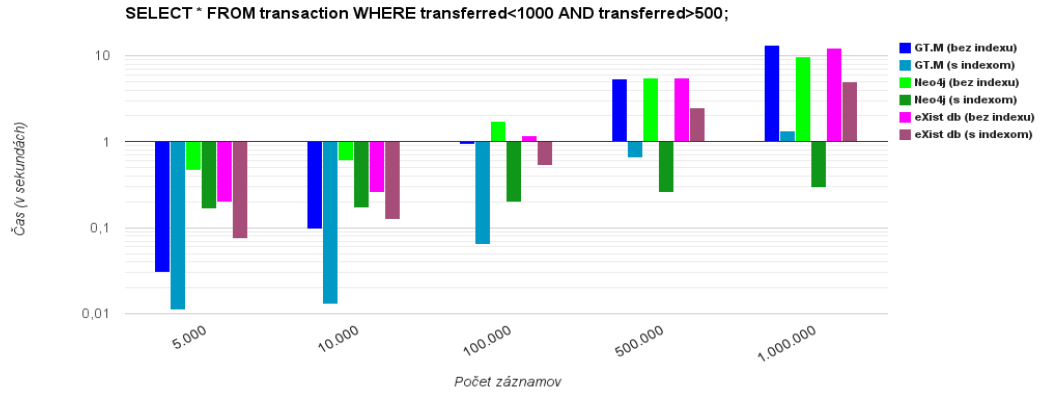
| GT.M | PostgreSQL | MongoDB | Neo4j | eXist | MemSQL | CSQL |
|---|---|---|---|---|---|---|
| žiadny index | B-Tree | Hash Index | BatchInserter Index | Range Index | Skip List Index | Tree Index |
| | Hash Index | Compound Index | Index engine (Apache Lucene) | Structural Index | Hash Index | Hash Index |
| | GiST | Unique Index | | Fulltext Index | | |
| | SP-GiST | Primary Index | | Legacy Fulltext Index | | |
| | GIN | Sparse Index | | NGram Index | | |
| | | Geospatial Index | | Spatial Index | | |
| | | Text Index | | | | |
| | | Multikey Index | | | | |

**Tabuľka 5.1:** Tabuľka testovaných systémov a ich indexácie

### 5.3.9 Výsledky testovania

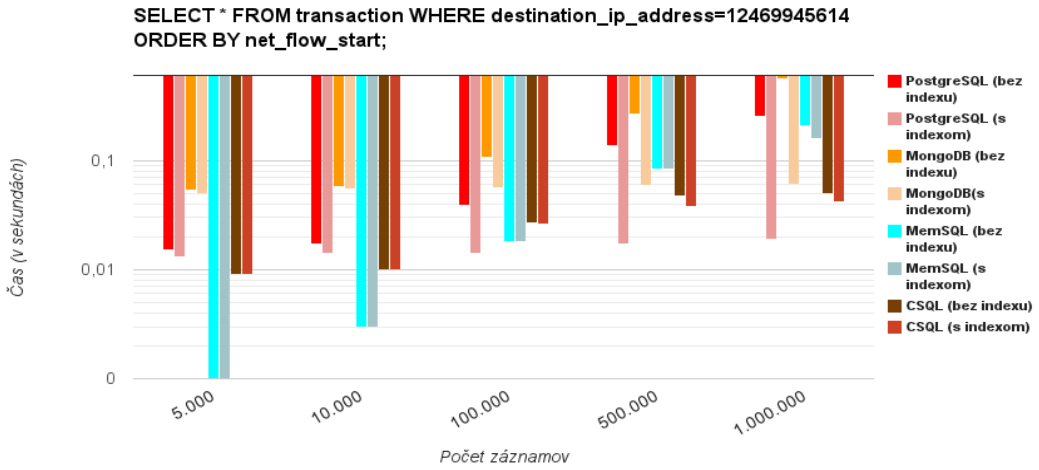



**Obr. 5.5:** Lepšie výsledky 1. selektu

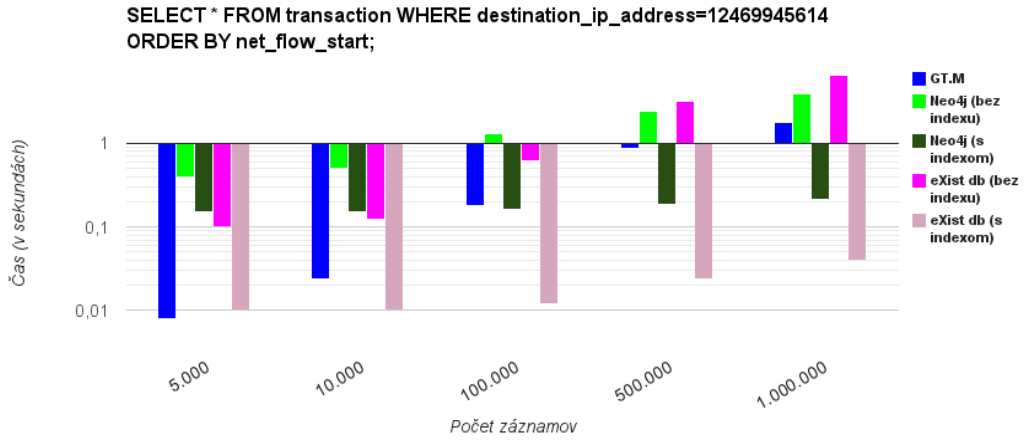



**Obr. 5.6:** Horšie výsledky 1. selektu

Všeobecne pri porovnávaní času potrebného na výkon dotazu s indexom a bez pomoci indexu je základným prepokladom, že dotaz v bezindexovom prostredí sa bude vykonávať dlhšie ako v prostredí s vhodným indexom. Vyššie uvedené výsledky dokazujú, že v niektorých prípadoch pri malom množstve dát to tak byť nemusí (viď výsledky PostgreSQL či MemSQL), naopak u niektorých systémov môže byť rozdiel badateľný už vtedy (viď Neo4j a GT.M). V skupine s lepšími časmi sa umiestnili oba In-Memory DBMS, čo bolo vzhľadom na ich charakter viacmenej predpokladané a následne i potvrdené. Na uvedených grafov je takisto dobre badateľné, že niektoré systémy sú výkonnejčie pri menšom množstve dát.




**Obr. 5.7:** Lepšie výsledky 2. selektu




**Obr. 5.8:** Horšie výsledky 2. selektu

Ako už bol vyslovený predpoklad pri testovaní tree indexu, aj v prípade hash indexu ho niektoré výsledky v konečnom dôsledku vyvracajú. Konkrétne sa jedná o výsledky MemSQL databáze, ktorej výsledky sú odlišné až pri milióne dát. Môže to byť ovplyvnené tým, že MemSQL požaduje unikátne hodnoty v rámci hash indexu a vzhľadom na typ testovacích dát to bolo uskutočnené až pri 5 zaindexovaných hodnotách, čo môže i negatívne vplývať na výkon systému. Takisto je nutné poznamenať, že sa v prípade GT.M systému nerozlišuje indexové a bezindexové prostredie zdôvodu popísaného vyššie v kapitole 5.3. Zároveň pri porovnaní výhodnosti využitia indexácie na vyššie uvedené dva dotazy bol efekt viditeľnejší pri využití hash indexu.

## 5.4 Kartézsky súčin, join

V prípade zlučovania tabuliek pomocou kartézskeho súčinu alebo joinu dochádza k viacerým situáciám. V prvom rade je nutné poznamenať, že tieto konštrukcie (kartézsky súčin i join) sa využívajú predovšetkým v relačných databázach, kde pracujú s tabuľkami. Ostatné systémy (NoSQL) nevyužívajú podobné konštrukcie a preto je potrebné prispôsobiť konkrétny dotaz možným funkciám daného systému. Druhým dôležitým faktorom sú zdrojové dáta. Vzhľadom na ich charakter a informácie, ktoré obsahujú, bola napríklad v prípade relačnej databáze zvolená len 1 tabuľka, ktorej 1 riadok obsahuje všetky hodnoty 1 záznamu a preto je možnosť zakomponovania konštrukcie kartézskeho súčinu nevyužitá. V prílohe F sú popísané typy joinov, ktoré sú systémami podporované a takisto, či systémy podporujú kartézsky súčin. V rámci testovania bol zvolený tzv. self-join v tvare:

```
SELECT t1.destination_ip_address
FROM transaction as t1, transaction as t2
WHERE t1.destination_ip_address=t2.source_ip_address;
```

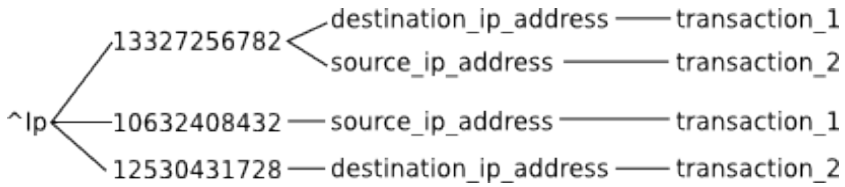
V následujúcom texte sú popísané metódy riešenia situácie len pri systémoch bez podpory joinu a taktiež prehľadná tabuľka 5.2, kde je zobrazené, ktorý systém podporuje aký typ joinu.

### 5.4.1 GT.M

Štruktúra systému GT.M neumožňuje využitie joinu a tým pádom ani kartézskeho súčinu.

V rámci testovania bol vytvorený dotaz, ktorý využíva indexáciu (dochádza k vytvoreniu novej globálnej premennej `^Ip`, pomocou ktorej sa v rámci 1 `FOR` cyklu hľadá taká ip adresa, ktorej oba detské záznamy `destination_ip_address` a `source_ip_address` majú aspoň 1 potomka, ktorým je číslo transakcie).

Napríklad pre transakciu č. 1, ktorej `destination_ip_address=13327256782` a `source_ip_address=10632408432` a transakciu č. 2, ktorej `destination_ip_address=12530431728` a `source_ip_address=13327256782`, bude vyzerať premenná `^Ip` následovne:




**Obr. 5.9:** Znázornená štruktúra novo vytvoreného indexu ^Ip

### 5.4.2 MongoDB

V prípade systému MongoDB sa tiež nedá hovoriť o joine. Dotaz bol prispôsobený tomu, čo hľadáme. Najprv si do novej kolekcie `dList` v Jave uložíme všetky hodnoty `destination_ip_address` bez duplicít. Následne si vytvoríme druhú novú kolekciu `sList`, do ktorej priamo ukladáme hodnoty `source_ip_address` taktiež bez duplicít, ale len vtedy, ak sa nachádzajú aj v kolekcii `dList`. Priamo nad databázou je toto možné pomocou premenných `var`.

### 5.4.3 Neo4j

Systém Neo4j nemá takisto podporu joinov. Na oficiálnej stránke je k nim prirovnávaná `MATCH` konštrukcia v rámci jazyka CYPHER, v ktorej dochádza k prechodnému označovaniu uzlov počas prechádzania určitého vzoru. V rámci tejto práce boli hľadané hodnoty nájdené postupným prechádzaním každého uzlu predstavujúceho ip adresu (viď štruktúra databáze C). V prípade, že do tohto uzlu smeroval aspoň jeden vzťah typu `DESTINATION_IP_ADDRESS` a zároveň doň smeroval aspoň jeden vzťah typu `SOURCE_IP_ADDRESS`, tak sa vypísala hodnota jeho atribútu `ip_number`.

### 5.4.4 eXist db

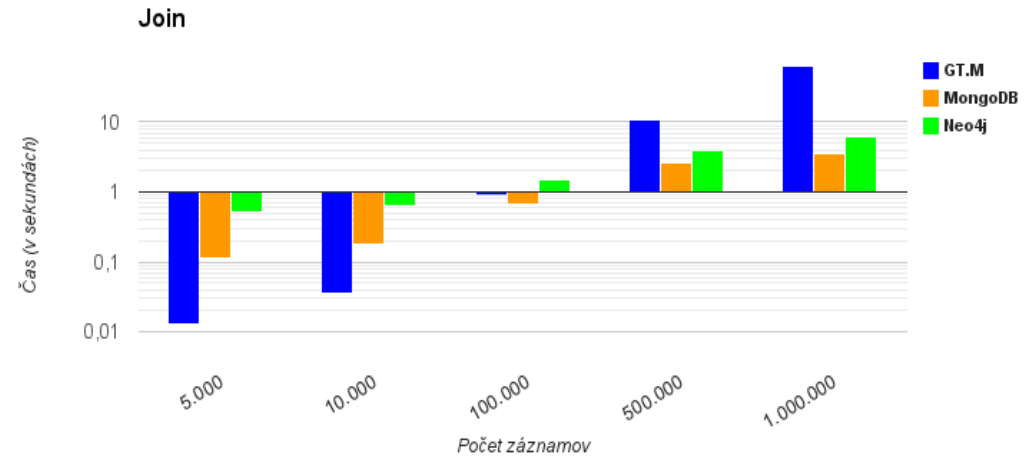
Databázový systém eXist db patrí taktiež medzi NoSQL systémy bez podpory akéhokoľvek druhu joinu, no vďaka jazyku XQuery sme schopní tento nedostatok nahradiť vlastnými príkazmi bez spájania akýchkoľvek tabuliek. Výsledný dotaz sa nachádza v prílohe D.

### 5.4.5 Zhrnutie

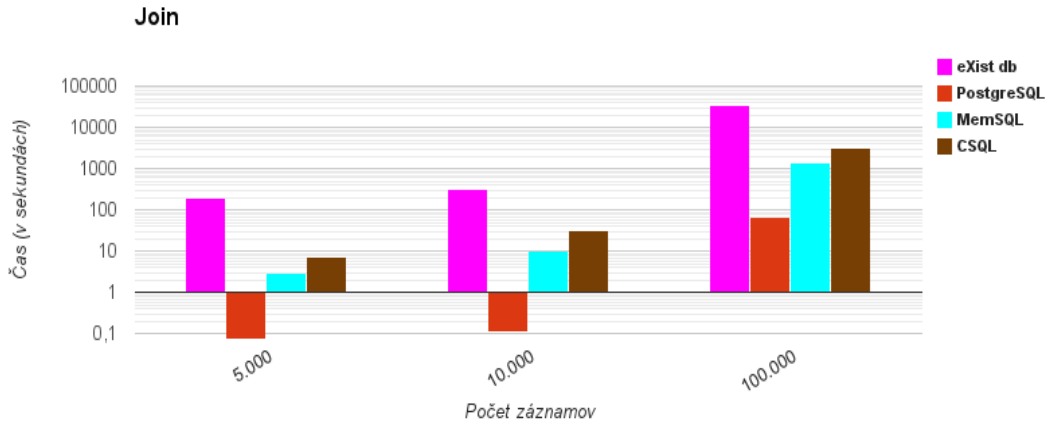
| | GT.M | PostgreSQL | MongoDB | Neo4j | eXist | MemSQL | CSQL |
|---|---|---|---|---|---|---|---|
| Inner Join | nemá | má | nemá | nemá | nemá | má | má |
| Outer Join | nemá | má | nemá | nemá | nemá | má | má |
| Left Join | nemá | má | nemá | nemá | nemá | má | má |
| Right Join | nemá | má | nemá | nemá | nemá | má | nemá |
| Cross Join | nemá | má | nemá | nemá | nemá | má | má |
| Self Join | nemá | má | nemá | nemá | nemá | má | má |

**Tabuľka 5.2:** Tabuľka testovaných systémov a ich joinov

### 5.4.6 Výsledky testovania




**Obr. 5.10:** Lepšie výsledky joinu




**Obr. 5.11:** Horšie výsledky joinu

Pri testovaní dotazu uvedeného na začiatku kapitoly 5.4 sa vyskytli viaceré nepriaznivé okolnosti. Jednou z nich je časová náročnosť pri vykonávaní dotazu v prostredí systému eXist db (ako i v prostredí relačných databáz), kde dochádzalo nielen k časovej ale i pamäťovej náročnosti dotazu. Z toho dôvodu sa testovanie na týchto systémoch obmedzilo len po maximálne 100.000 záznamov. Zvyšné systémy (takmer všetky NoSQL) prispôsobené typy dotazov zvládali bez problémov a tak sa testovanie vykonalo až po milión záznamov. Vzhľadom na štruktúru relačných databáz bol tento priebeh očakávaný.

## 5.5 Paralelizácia dotazov nad tabuľkami

Vykonávanie dotazov paralelne má svoje klady i zápory. Sčasti sa vystavujeme nebezpečenstvu, že narušíme konzistenciu dát, ich korektnosť, validitu. Väčšina systémov v prípade, že môže dôjsť k podobnému narušeniu, vykonáva zmeny v transakciách, počas ktorých si záznam uzamkne, aby k nemu nemalo iné vlákno prístup a po ukončení dotazu zmenu commitne[6] do databáze a odomkne záznam pre iné vlákna. Keďže ale testujeme dotaz, ktorý dáta len číta z databáze a počas neho sa nevykonáva žiadny iný dotaz, ktorý by mohol narušit valídnosť výsledku, tak to nebolo potrebné.

Testovaným dotazom pre porovnanie času potrebného na paralelné vykonávanie dotazu 10 vláknami a sériové vykonanie dotazu 10 krát za sebou bol:

```
SELECT *
FROM transaction
WHERE protocol = 17;
```

### 5.5.1 PostgreSQL, MemSQL

PostgreSQL je odlišný od ostatných systémov tým, že podporuje len určitú formu paralelizácie. Na to, aby došlo k zdieľaniu medzi 10 vláknami je potrebné mať 10 zvlášť pripojení k databáze, pričom každé pripojenie je jednovláknové. Po ukončení vykonávania dotazu je potrebné týchto 10 pripojení zatvoriť. Takisto systém MemSQL využíval 10 nových pripojení k databáze, ktoré sa po ukončení dotazovania zatvorili.

### 5.5.2 Neo4j

Na rozdiel od vyššie spomenutých SQL systémov je odporúčané vykonávať paralelné dotazy v rámci 1 pripojenia k databáze. Takisto vytvorenie viacerých execution engine[7] sa neodporúča, pretože to spôsobuje horší výkon. V rámci tejto práce vďaka Java Core API nebolo nutné využívať execution engine.

### 5.5.3 eXist db

Pre povolenie paralelného vykonávania dotazov je potrebné server spúšťať príkazom:

```
./startup.sh -t 10
```

pre povolenie prístupu maximálne 10 vláknam zároveň.

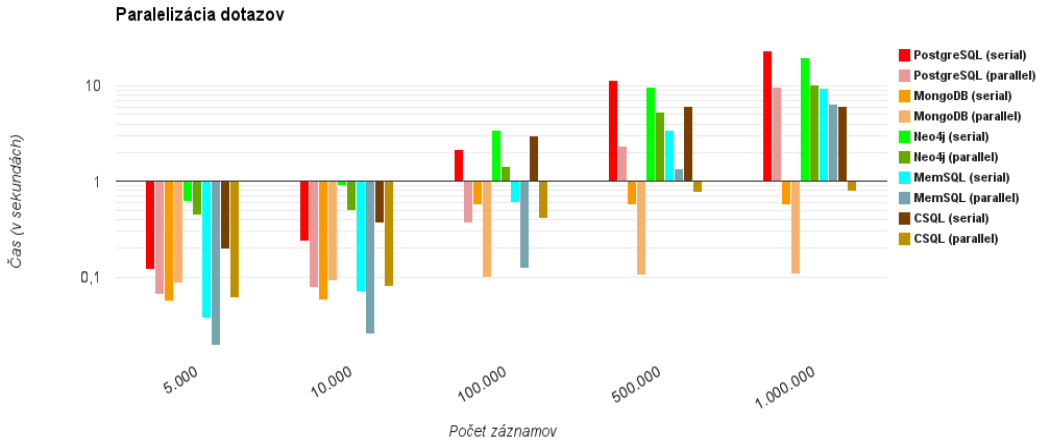
---

6. commit = nahrá nový stav položky
7. v preklade: spúšťací engine, slúžiaci na spustenie dotazov nad databázou, napr. dotazy v jazyku CYPHER

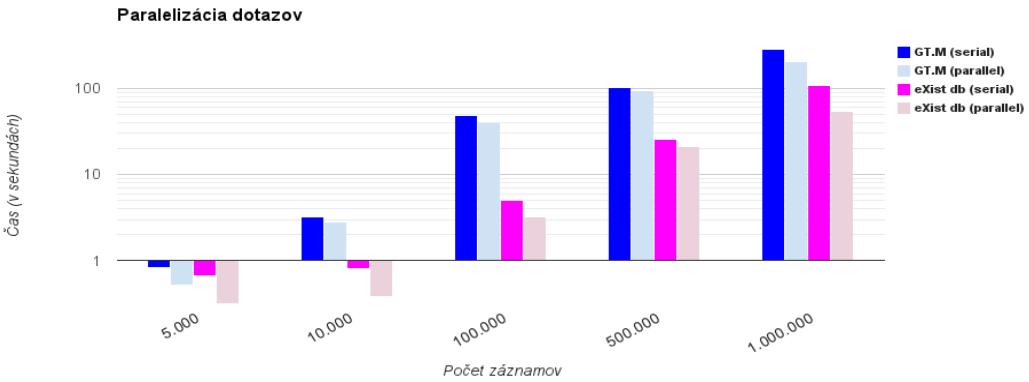
### 5.5.4 CSQL, MongoDB

CSQL a MongoDB sa pripájali k databáze zvonku pomocou vytvorených skriptov. V prípade testovania paralelizácie boli vytvorené vlákna, ktoré paralelne spúšťali nad databázami skripty priložené k testovacím nástrojom.

### 5.5.5 Výsledky testovania




**Obr. 5.12:** Lepšie výsledky paralelných dotazov




**Obr. 5.13:** Horšie výsledky paralelných dotazov

Prepokladom pri testovaní paralelného prístupu k databázovým systémom bola niekoľkonásobná výhodnosť oproti jednému prístupu. Tento predpoklad bol následne potvrdený i v praxi pomocou vyššie zobrazených výsledkov.

# 6 Zhrnutie

V následujúcom texte sa nachádza krátke zhrnutie silných a slabých stránok jednotlivých testovaných DBMS vzhľadom na testované okruhy.

GT.M sa ukázal veľmi výkonný pri hromadnom i sekvenčnom importe menšieho množstva dát. So stúpajúcim množstvom sa jeho výkonnosť oproti ostatným systémom (vo svojej skupine výsledkov) trochu zhoršila. Tvorba indexácie tento systém však nijak výraznejšie neovplyvnila vo výslednom čase, napriek tomu, že samotný systém indexáciu ako takú nepodporuje a musela byť dotváraná ručne. Efektívnosť využívania tejto indexácie je však najmä pri veľkom množstve dát očividná, pri milióne dát je výsledok až 13-násobne lepší.

PostgreSQL import dát do rôznych prostredí zvláda obdobne ako GT.M. Výkonnosť jednotlivých importov sa dá považovať vzhľadom na ostatné systémy skôr priemerná. Za silnú stránku tohto systému sa dá považovať hash index, ktorého vhodnosť využitia sa prejavila najmä pri veľkom množstve dát. Zo svojej kategórie výsledkov v rámci testovania joinu bol PostgreSQL najvýkonnejším, no napriek tomu v porovnaní s lepšou skupinou výsledkov bol jeho výkon niekoľkonásobne horší. Ďalšou silnou stránkou tohto systému je paralelný prístup a dotazovanie nad databázou.

MongoDB ako jedným z predstaviteľov NoSQL databáz jednoznačne nesklamal pri testovaní join príkazu a vysporiadaní sa s viacerými pripojeniami k databáze, kde bol jeho výkon najlepší. Takisto importovanie dát tento systém zvládal pomerne dobre oproti ostatným. Za jeho slabšiu stránku sa dá považovat tree index, ktorého efektivita sa prejavila v menšej miere ako to bolo pri hash indexe. Vhodnosť využitia efektivita hash indexu potvrdzujú výsledky, no oproti ostatným systémom je jeho výkon priemerný, pri menšom množstve dokonca porovnateľný (miestami horší) s výkonom PostgreSQL bez využitia hash indexu.

Neo4j sa prejavil najvýkonnejší pri hromadnom importovaní dát, pri sekvenčnom to bolo skôr opačne. Použitá idnexácia výrazne zlepšuje výkonnosť systému (aj tree aj hash index), no stále je Neo4j pomalší ako niektoré iné systémy. Z môjho pohľadu môže byť dôvodom poskytnutý typ dát, ktorá je skôr vhodný pre iné paradigma. Napriek tomu štruktúra databáze umožňuje veľmi rýchle vykonanie join príkazu a taktiež paralelné pripojenia tento systém zvláda veľmi dobre.

EXist db využíval na rozdiel od ostatných systémov len 1 typ indexácie pre oba dotazy. V porovnaní s ostatným systémami sú jeho výsledky skôr horšie, no zlepšenie pri používaní indexácie je evidentné. Použitý range index však, v závislosti na predpokladanom množstve dát, lepšie zvládal menšie množstvo, čo vysvetľuje rozdielne zlepšenie pri 1. selekte (2-násobné) a 2. selekte (6-násobné). Za slabú stránku tohto systému jednoznačne môžeme považovať import dát i testovanie príkazu join.

MemSQL, ako jeden z predstaviteľov In-Memory DBMS, sa ukázal veľmi výkonným pri

sekvenčnom importe i hromadnom importe, ako i pri selektoch. Využívanie hash indexu však nie je až také výhodné ako pri ostatných systémoch, nakoľko je nutná unikátna hodnota v hash indexe, čo pri danom type dát výrazne zhoršilo jeho výkonnosť. Takisto ho môžeme považovať za najvýkonneší pri paralelnom dotazovaní nad databázou, no so zvyšujúcimi sa množstvom dát sa jeho výkonnosť zhoršuje oproti ostatným systémom. Pri testovaní príkazu join si na malom množstve viedol dobre, no vzhľadom na fakt, že ide o In-Memory databázu, sa jej pamäťové nároky pri väčšom množstve dát príliš zväčšili.

CSQL, ako druhý z predstaviteľov In-Memory DBMS, tiež zvládal paralelné dotazovanie nad databázou veľmi dobre oproti ostatným systémom, aj na malom aj na väčšom množstve dát. Indexy výkonnosť dotazov do určitej miery zlepšujú, no nie je to až v takej miere ako pri iných systémoch. Tento systém jednoznačne zle zvláda import dát, rovnako ako to je i u eXist db.

# 7 Záver

Cieľom tejto práce bolo sprostredkovať prehľad a porovnanie rôznych druhov databázových systémov vzhľadom na ich paradigmá.

V úvodnej časti tejto práce je čitateľovi poskytnuté priblíženie témy a predpokladaný priebeh vývoja práce ako i jej ciele. Teoretická časť bola pokrytá troma kapitolami. V prvej z nich sa čitateľ zoznámil so stručnou históriou vývoja databázových systémov. V druhej boli postupne popisované jednotlivé databázové modely, ich špecifické vlastnosti a pravidlá pri ich používaní. Takisto boli zakaždým spomenutí niekoľkí významní zástupcovia daného databázového modelu a predstavený zástupca, ktorý bol následne súčasťou testovacej časti. V poslednej kapitole teoretickej časti bol opísaný každý vybratý zástupca jednotlivých modelov, najskôr z pohľadu náročnosti inštalácie a dostupnosti dokumentácie, potom na základe množstva existujúcich konektorov a API na prácu s konkrétnym systémom, ako i spôsob administrácie a nevyhnutná konfigurácia pre optimalizáciu výsledkov.

Druhá časť práce bola čisto praktická. Boli vybraté 4 okruhy, v rámci ktorých sa testovali jednotlivé modely so zameraním na ich výkonnosť. Prvým testovaným okruhom bol import dát do databáz - aj hromadný aj postupný. Takisto sa porovnávali výsledky importov do prostredia s indexami a bez nich. Následne sa testovala užitočnosť indexácie vzhľadom na typ testovacích dát a dostupných typov indexov v rámci databáz. Tretím okruhom bolo testovanie výkonnosti joinu v rámci SQL databáz ako i prispôsobovanie dotazov (vzhľadom na požiadavky) pri NoSQL databázach, ktoré nepodporujú join. Posledným okruhom bolo testovanie výkonnosti a výhodnosti paralelného dotazovania sa nad 1 kolekciou/tabuľkou/dokumentom.

Výsledkom tejto práce je 7 testovacích nástrojov, parserovacie nástroje, návody na inštaláciu niektorých nie úplne najznámejších databázových systémov a grafy zobrazujúce výhody i slabiny jednotlivých systémov pri danom type dát.

Za najzdĺhavejšiu a pravdepodobne i najnáročnejšiu časť tejto práce považujem úplny začiatok, kedy bolo potrebné všetky databázové systémy úspešne rozbehnúť na tom istom operačnom systéme za relatívne podobných podmienok, pričom každý zo systémov mal iné požiadavky na prostredie, inú kvalitu a spôsob dokumentácie.

Prínosnosť tejto práce spočíva v precíznom porovnaní širokého záberu databázových modelov, rovnako i samotné množstvo rôznych modelov môže byť pre niekoho obohacujúce. Ďalšie pokračovanie tejto práce si viem predstaviť rôznymi spôsobmi, buď pokračovaním v testovaní a zameranie sa na porovnanie In-Memory a On-Disk databáz, zakomponovanie databáze typu kľúč-hodnota, prípadne najprv v zjednotení testovacích nástrojov do jedného a prirobením GUI pre jednoduchšiu manipuláciu a následného rozširovania tohto nástroja a zameranie sa na iné, v tejto práci netestované, oblasti databáz.

# Literatúra


[1] *CAP Theorem Diagram for distribution* [online]. Január 2013. <http://blog.nosqltips.com/2011/04/cap-diagram-for-distribution.html>.

[2] *Graph Databases: The New Way to Access Super Fast Social Data* [online]. Január 2013. <http://www.postgresql.org/docs/8.3/static/plpgsql-overview.html#PLPGSQL-ADVANTAGES>.

[3] *Databázové modely* [online]. Január 2013. <http://www.databaze.chytrak.cz/modely.htm>.

[4] *What is M Technology? How does it relate to KBSQL?* [online]. Január 2013. <http://kbs.custhelp.com/app/answers/detail/a_id/869/~/what-is-m-technology-%28mumps%29%3F-how-does-it-relate-to-kb_sql%3F>.

[5] CHODOROW, K. *50 Tips and Tricks for MongoDB Developers*. California, Sebastopol : O'Reilly Media, Inc, Apríl 2011. ISBN 978-1-449-30461-4.

[6] CHODOROW, K. *Sharding with the Fishes* [online]. Január 2013. <http://www.kchodorow.com/blog/2010/03/30/sharding-with-the-fishes/>.

[7] DEAN, J. – GHEMAWAT, S. *MapReduce: Simplified Data Processing on Large Clusters* [online]. Január 2013. <http://research.google.com/archive/mapreduce.html>.

[8] DRAKE, J. D. – WORSLEY, J. C. *Practical PostgreSQL*. O'Reilly Media, Inc, Január 2002. ISBN 978-1-565-92846-6.

[9] FAWCETT, J. – QUIN, L. R. – AYERS, D. *Beginning XML, 5th Edition*. John Wiley Sons, Jún 2012. ISBN 978-1-118-16213-2.

[10] GILL, P. *Database Management Systems*. I.K.International Publishing House Pvt Ltd, 2008. ISBN 978-81-89866-83-9.

[11] HARDY, D. – MALLUS, G. – MREUR, J.-N. *Networks: Internet, Telephony, Multimedia*. Springer, 2002. ISBN 2-7445-0144-1.

[12] HEWITT, E. *Cassandra: The Definitive Guide*. California, Sebastopol : O'Reilly Media, Inc, 1. vydanie, November 2010. ISBN 978-1-449-39041-9.

[13] NARANG, R. *Database Management Systems*. PHI Learning Pvt Ltd, 2. vydanie, Apríl 2011. ISBN 978-81-203-4313-9.

[14] PANNERSELVAM, R. *Database Management Systems*. PHI Learning Pvt Ltd, August 2006. ISBN 81-203-2028-X.

[15] PLATTNER, H. – ZEIER, A. *In-Memory Data Management: Technology and Applications*. Berlin : Springer, 2. vydanie, Máj 2012. ISBN 978-3-642-29575-1.

[16] PROCHÁZKA, D. *Oracle - průvodce správou, využitím a programováním nad databázovým systémem*. Praha : Grada Publishing a.s., 2009. ISBN 978-80-247-2762-2.

[17] REDMOND, E. – WILSON, J. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf, Máj 2012. ISBN 978-19-343-5692-0.

[18] SINGH, S. *Database Systems: Concepts, Design and Applications*. Pearson Education India, 2009. ISBN 978-81-317-6092-5.

[19] SOCHOR, J. – RÁČEK, J. *Modelování dat* [online]. Január 2013. <https://is.muni.cz/auth/el/1433/podzim2012/PB007/um/35424437/35424448/an_04_erd.pdf?studium=598629>.

[20] SUMATHI, S. – ESAKKIRAJAN, S. *Fundamentals of Relational Database Management Systems*. Springer, 2007. ISBN 978-3-540-48397-7.

[21] TIWARI, S. *Professional NoSQL*. John Wiley Sons, 1. vydanie, September 2011. ISBN 978-0-470-94224-6.

[22] VENKATESH, P. – VAMSI, B. *The Importance of In-memory Databases* [online]. Január 2013. <http://www.linuxforu.com/2012/01/importance-of-in-memory-databases/>.

[23] WARDEN, P. *Big Data Glossary*. California, Sebastopol : O'Reilly Media, Inc, September 2011. ISBN 978-1-449-31459-0.

# A Príloha - DVD

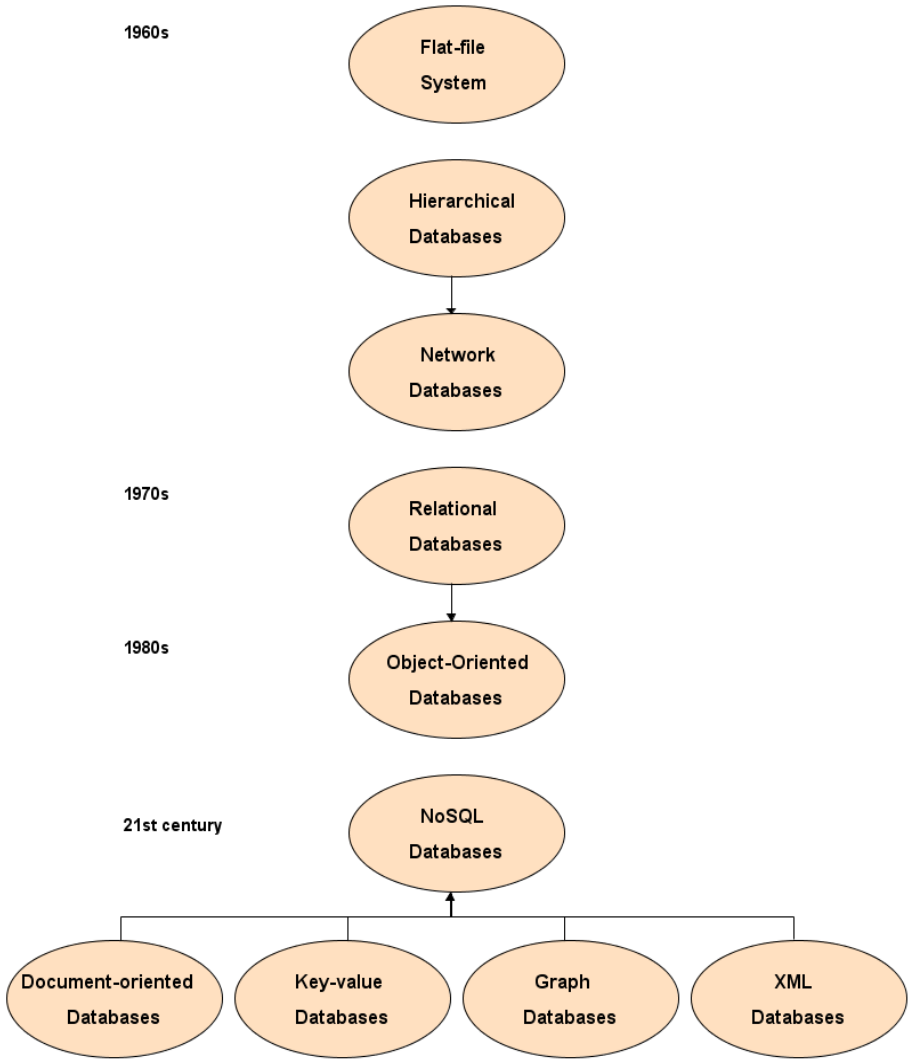
K práci je priložený DVD nosič obsahujúci adresáre:

```
|-- data/               # adresár obsahujúci zdrojové dáta
|-- data_parsers/       # adresár s parsermi
  |-- README              súbor popisujúci spustenie parserov
|-- tools/              # adresár s testovacími aplikáciami
  |-- gtm/
  |-- postgresql/
  |-- mongodb/
  |-- neo4j/
  |-- existdb/
  |-- memsql/
  |-- csql/
|-- bc_dbms.pdf           pdf súbor bakalárskej práce
|-- README                súbor na orientáciu v adresári
```

Každá aplikácia v adresári `tools/` má následujúcu štruktúru:

```
|-- app/                # adresár konkrétnej aplikácie
  |-- dist/             # adresár s .jar súbormi
  |-- doc/              # adresár s dokumentáciou
  |-- lib/              # adresár s dodatočnými knižnicami
  |-- res/              # adresár s dodatočnými súbormi
    |-- queries/        # adresár s vykonávanými dotazmi
  |-- run/              # adresár so spúšťacím skriptom
    |-- run.sh/
  |-- src/              # adresár so zdrojovými kódmi
    |-- app/
  |-- build.xml
  |-- README              súbor popisujúci presnú inštaláciu
                          databáze a spustenie aplikácie
```

# B Príloha - Schéma vzniku jednotlivých databáz




**Obr. B.1:** Schéma druhov databáz a ich doba vzniku

# C Príloha - Štruktúra dát

## C.1 Zdrojové data

```
133272567782,64531,.A....,SNGL,0,2013-01-11 14:55:12.957,
1,6,131028811119,443,40,0
```

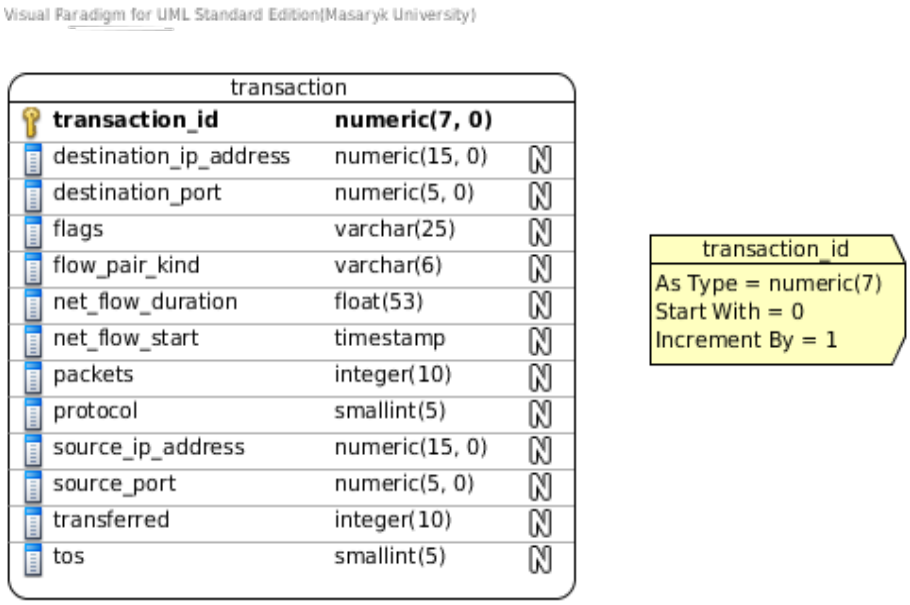
**Obr. C.1:** Pôvodné data (zľava destination_ip_address, destination_port, flags, flow_pair_-kind, net_flow_duration, net_flow_start, packets, protocol, source_ip_address, source_port, transferred, tos

## C.2 M formát dát

```
^Network,transaction_1,1
^Network,transaction_1,destination_ip_address,133272567782
^Network,transaction_1,destination_port,64531
^Network,transaction_1,flags,1,ack
^Network,transaction_1,flow_pair_kind,SNGL
^Network,transaction_1,net_flow_duration,0
^Network,transaction_1,net_flow_start,2013-01-11 14:55:12
^Network,transaction_1,packets,1
^Network,transaction_1,protocol,6
^Network,transaction_1,source_ip_address,131028811119
^Network,transaction_1,source_port,443
^Network,transaction_1,transferred,40
^Network,transaction_1,tos,0
```

**Obr. C.2:** data.m

## C.3 SQL formát dát




**Obr. C.3:** data.sql

Transaction_id predstavuje sekvenciu typu numeric, ktorá sa automaticky zvyšuje o 1.

## C.4 JSON formát dát

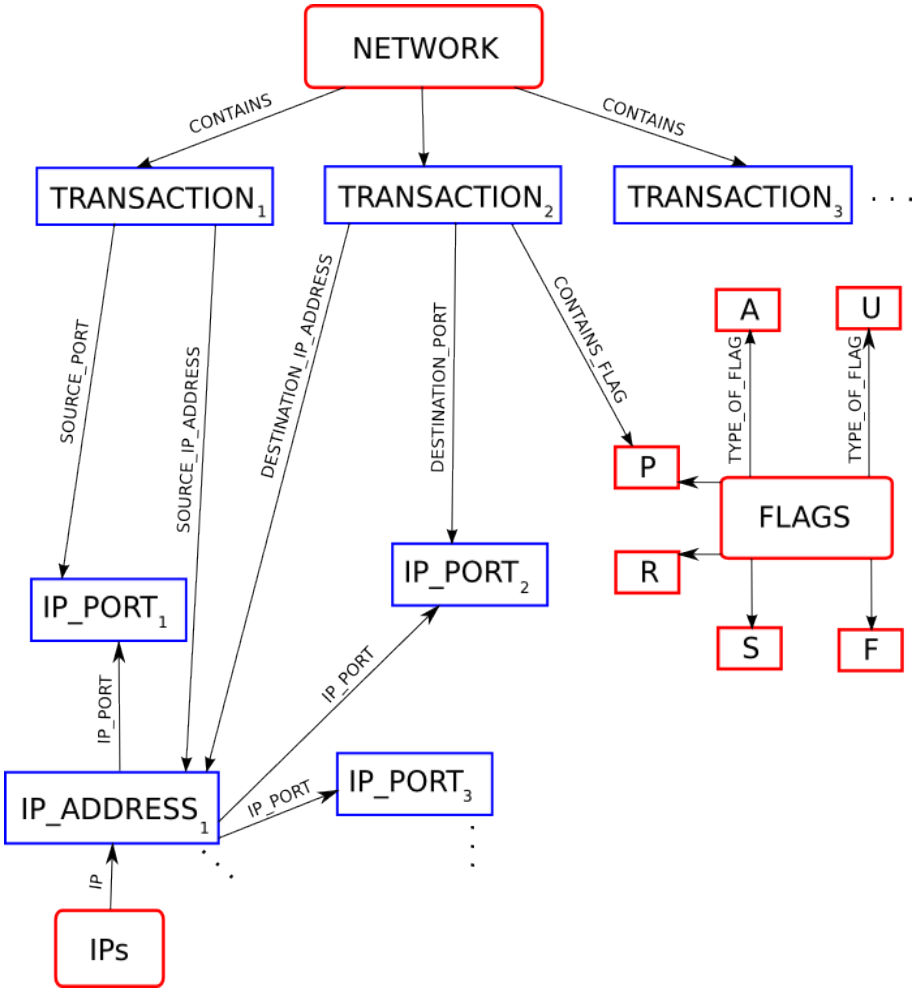
```
{"destination_ip_address":133272567782,
"destination_port":64531,"flags":["ack"],
"flow_pair_kind":"SNGL","net_flow_duration":0,
"net_flow_start":"2013-01-11,14:55:12.957","packets":1,
"protocol":6,"source_ip_address":131028811119,
"source_port":443,"transferred":40,"tos":0}
```

**Obr. C.4:** data.json

## C.5 CYPHER formát dát

Uzly sú farebne rozdelené na 2 druhy:

- červené – uzly, ktoré sú vytvárané ako prvé pri tvorbe prostredia (trieda `PrepareEnvironment`)
- modré – uzly, ktoré sú dotvárané počas testovania (trieda `Import`)




**Obr. C.5:** data.cypher

## C.6 XML formát dát

```
<network>
 <transaction id="1">
  <destination_ip_address>133272567782</destination_ip_address>
  <destination_port>64531</destination_port>
  <flags>
   <ack/>
  </flags>
  <flow_pair_kind>SNGL</flow_pair_kind>
  <net_flow_duration>0</net_flow_duration>
  <net_flow_start>2013-01-11 14:55:12.957</net_flow_start>
  <packets>1</packets>
  <protocol>6</protocol>
  <source_ip_address>131028811119</source_ip_address>
  <source_port>443</source_port>
  <transferred>40</transferred>
  <tos>0</tos>
 </transaction>
</network>
```

**Obr. C.6:** data.xml

# D Príloha - Dotazy

## D.1 GT.M

- Hromadný import
  Vykonávame ho pomocou programu MUPIP, ktorý je súčasťou adresára s nainštalovaným GT.M systémom. V Javovej aplikácii následne prebieha hromadý import pomocou tohto programu príkazom

  ```
  /opt/gtm/mupip load nazov_suboru
  ```

- Sekvenčný import

  ```
  set(^..)
  ```

- Tree index (bez indexácie)

  ```
  set transaction=""
  for set transaction=$ORDER(^Network(transaction))
     quit:transaction="" do
  . if ^Network(transaction,"transferred")<1000,
     ^Network(transaction,"transferred")>500 do
  . . zwrite ^Network(transaction,*)
  . quit
  quit
  ```

- Tree index (s indexáciou)

```
set transf=""
set i=""
for set transf=$ORDER(^TransfIndx(transf)) quit:transf=""
   do
. if transf>500,transf<1000 do
. . for set i=$ORDER(^TransfIndx(transf,i)) quit:i="" do
. . . set transId=$PIECE(^TransfIndx(transf,i),"_",2)
. . . set transac=""
. . . for set transac=$ORDER(^Network(transac)) quit:
   transac="" do
. . . . if ^Network(transac)=transId zwrite ^Network(
   transac,*)
. . . quit
. . quit
. quit
quit
```

- Hash index

```
set destIp=""
set indexData=""
for set indexData=$ORDER(^StartIndx(indexData))
   quit:indexData="" do
. for set destIp=$ORDER(^StartIndx(indexData,destIp))
   quit:destIp="" do
. . if destIp=12469945614 do
. . . set transactionId=$PIECE(^StartIndx(indexData,destIp
   ),"_",2)
. . . set transaction=""
. . . for set transaction=$ORDER(^Network(transaction))
   quit:transaction="" do
. . . . if ^Network(transaction)=transactionId do
. . . . . zwrite ^Network(transaction,*)
. . . . quit
. . . quit
. . quit
. quit
quit
```

- Join

```
set transact=""
for set transact=$ORDER(^Ip(transact))
   quit:transact="" do
. if $DATA(^Ip(transact,"destination_ip_address"))=10,
   $DATA(^Ip(transact,"source_ip_address"))=10 do
. zwrite ^Ip(transact,*)
. quit
quit
```

- Paralelizácia

```
set transact=""
for set transact=$ORDER(^Network(transact))
   quit:transact="" do
. if ^Network(transact,"protocol")=17 do
. . zwrite ^Network(transact,*)
. quit
quit
```

## D.2 PostgreSQL

- Hromadný import

```
COPY transaction
FROM '/data/data.csv'
DELIMITER ',' CSV
```

  V našom prípade je hromadný import vykonávaný pomocou Java API, v ktorom namiesto postupného spúšťania dotazov pomocou premennej `statement.execute(dotaz)`, odchádza k jej naplneniu dotazmi, pomocou metódy `statement=addBatch(dotaz)` a následnému hromadnému spusteniu pomocou `statement.executeBatch()`.

- Sekvenčný import

```
INSERT INTO transaction VALUES(..);
```

- Tree index

```
SELECT *
FROM transaction
WHERE transferred>500 AND transferred<1000;
```

- Hash index

```
SELECT *
FROM transaction
WHERE destination_ip_address = 12469945614
ORDER BY net_flow_start;
```

- Join

```
SELECT t1.destination_ip_address
FROM transaction as t1, transaction as t2
WHERE t1.destination_ip_address = t2.source_ip_address;
```

- Paralelizácia

```
SELECT *
FROM transaction
WHERE protocol=17;
```

## D.3 MongoDB

- Hromadný import

```
mongoimport --db dbName --collection colName < data
```

V rámci MongoDB sa vykoná hromadný import, ktorý má 2 rôzne módy – import dát do kolekcie alebo update existujúcich dát a import nových. Prvý sa majoritne využíva pri prvotnom vkladaní záznamov, druhý ak manipulujeme s kolekciou, ktorá už obsahuje nejaké dáta a chceme ich pozmeniť. V našom prípade prebieha hromadný import v rámci Java API pomocou kolekcie `List<DBObject> basicDBObjects` naplnenou objektami `(DBObject) JSON.parse(line)`.

- Sekvenčný import

```
db.collection.insert({..});
```

- Tree index

```
db.collection.find({transferred:{$gt:500,$lt:1000}});
```

- Hash index

```
db.collection.find(
{destination_ip_address:12469945614}).sort(
{net_flow_start});
```

- Join

```
var destinationIpAddressList = db.collection.
distinct("destination_ip_address");
var sourceIpAddressList = db.collection.
distinct("source_ip_address",{"source_ip_address":
{$in:destinationIpAddressList}});
```

- Paralelizácia

```
db.collection.find({protocol:17});
```

## D.4 Neo4j

- Hromadný import (v Java Core API)

```
BatchInserter batchDB =
   BatchInserters.inserter(db_path);
Map<String, Object> newNodeProperty =
   new HashMap<String, Object>();
newNodeProperty.put("name", "Node001");
long newNode = batchDB.createNode(newNodeProperty);
```

  Predstavuje východzí import. Využíva sa výhradne pri prvotnom vkladaní dát do systému a nesmie sa vykonávať vo viacerych vláknach pre zachovanie konzistencie dát v databáze.

- Sekvenčný import (v Java Core API)

```
GraphDatabaseService graphDb =
   session.getDBConnection();
Node newNode = graphDb.createNode();
newNode.setProperty("name", Node001);
```

  V prípade importovania dát v Neo4j bolo nutné pri hodnotách `ip` a `port` kontrolovať unikátnosť uzla (tj. či daný uzol v databáze už existuje). V tom spočívala nevýhoda API oproti dotazovaciemu jazyku CYPHER, ktorý má vlastnú kontrolu pomocou klauzuly `CREATE UNIQUE`. Princíp kontroly je však rovnaký – pomocou vytvoreného indexu v databáze nad požadovanou unikátnou hodnotou. V našom prípade sa teda nikdy nebude jednať o čisto bezindexové prostredie, pretože je potrebné na utvorenie unikátnych uzlov.

- Tree index (bez indexácie, ukážka v jazyku CYPHER)

```
START n=node(0)
MATCH n-[]->t-[]->d_ip-[]->d_p<-[]-
   t-[]->s_ip-[]->s_p<-[]-t-[]->f
WHERE NOT(t.transferred<=500 OR t.transferred>=1000)
RETURN t, d_ip, d_p, s_ip, s_p, f;
```

  V aktuálnej verzii Java Core API je chyba pri vyhodnocovaní logickej spojky `AND`, preto je príkaz v zložitom tvare

- Tree index (s indexáciou v Java API)

```
Index<Node> trans = index.forNodes("transferred");
trans.query(QueryContext.numericRange(
   "transferred", from, to));
```

- Hash index (bez indexácie, ukážka v jazyku CYPHER)

```
START n=node(0)
MATCH n-[]->t-[]->d_ip-[]->d_p<-[]-
   t-[]->s_ip-[]->s_p<-[]-t-[]->f
WHERE d_ip.ip_number=12469945614
RETURN t, d_ip, d_p, s_ip, s_p, f
ORDER BY t.net_flow_start;
```

- Hash index (s indexáciou v Java API)

```
Index<Node> ip = index.forNodes("ip_number_hash");
ip.get("ip_number", "12469945614");
```

- Join (ukážka v jazyku CYPHER)

```
START n=node(0)
MATCH n-[]->t1-[]->ip<-[]-t2
WHERE t1.id=t2.id
RETURN ip.ip_number;
```

- Paralelizácia (ukážka v jazyku CYPHER)

```
START n=node(0)
MATCH n-[]->t-[]->d_ip
WHERE t.protocol=17
RETURN t, d_ip;
```

## D.5 eXist

- Hromadný import

```
put(data.xml)
```

- Sekvenčný import

```
update insert<node></node> into /network
```

- Tree index

```
for $x in /network/transaction[
   transferred<1000 and transferred>500]
return $x
```

- Hash index

```
for $x in /network/transaction[
   destination_ip_address = 12469945614]
order by $x/net_flow_start
return $x
```

- Join

```
for $x in doc('data.xml')/network/transaction
where $x/destination_ip_address =
   $x/source_ip_address
return $x/destination_ip_address
```

- Paralelizácia

```
for $x in /network/transaction[protocol = 17]
return $x
```

## D.6 MemSQL

- Hromadný import
  Import prebieha tým istým spôsobom ako v prípade PostgreSQL, popísaným v D.2.

- Sekvenčný import

```
INSERT INTO transaction VALUES();
```

- Tree index

```
SELECT *
FROM transaction
WHERE transferred>500 AND transferred<1000;
```

- Hash index

```
SELECT *
FROM transaction
WHERE destination_ip_address = 12469945614
ORDER BY net_flow_start;
```

- Join

```
SELECT t1.destination_ip_address
FROM transaction as t1, transaction as t2
WHERE t1.destination_ip_address = t2.source_ip_address;
```

- Paralelizácia

```
SELECT *
FROM transaction
WHERE protocol=17;
```

## D.7 CSQL

- Hromadný import

```
csql -s data.sql
```

Tento druh importu sa musel testovať pomocou systémových volaní z Javy, pretože príslušné API pre CSQL neobsahuje možnosť napĺňania premennej tak, ako to bolo v prípade PostgreSQL či MemSQL.

- Sekvenčný import

```
INSERT INTO transaction VALUES();
```

- Tree index

```
SELECT *
FROM transaction
WHERE transferred>500 AND transferred<1000;
```

- Hash index

```
SELECT *
FROM transaction
WHERE destination_ip_address = 12469945614
ORDER BY net_flow_start;
```

- Join

```
SELECT t1.destination_ip_address
FROM transaction as t1, transaction as t2
WHERE t1.destination_ip_address = t2.source_ip_address;
```

- Paralelizácia

```
SELECT *
FROM transaction
WHERE protocol=17;
```

# E Príloha - Druhy indexácie

## E.1 GiST/SP-GiST/GIN index

GiST index (PostgreSQL)
– ide o index, ktorý slúži ako vzor na vytváranie nových prístupových metód stromovej štruktúry

SP-GIST index (PostgreSQL)
– ide o rozšírenie GiST indexu o podporu rozdeľovacích vyhľadávacích stromov[1]

GIN index (PostgreSQL)
– slúži na prehľadávanie zložených položiek (dokument, pole, atď), kedy hľadáme konkrétnu hodnotu/reťazec znakov v tejto položke

## E.2 Geospatial/Text index

Geospatial index (MongoDB)
– delí sa na 2 typy: *2d index* pre výpočty rovinného povrchu a *2dsphere index* pre výpočty guľovitého povrchu

Text index (MongoDB)
– slúži na vyhľadávanie slov (stringov) v dokumente/kolekcii,
– existuje podpora len pre niektoré jazyky[2]

## E.3 Multikey index

Multikey index (MongoDB)
– vzniká, keď do indexu vložíme záznam, ktorý predstavuje pole hodnôt

## E.4 Unique/Primary (_id) index

Unique index (MongoDB)
– slúži na odmietanie dokumentov s duplicitnými hodnotami indexovaného záznamu

---

1. napríklad k-d strom či suffixový strom
2. viac na http://docs.mongodb.org/manual/core/text-search/

Primary index = (_id) index (MongoDB)
– ide o automatickú indexáciu zo strany MongoDB, kde `_id` záznam je unikátna hodnota

## E.5 Sparse index

Sparse index (MongoDB)
– používa sa na automatickú indexáciu záznamov v dokumentoch, ktoré majú indexovaný nejaký záznam/pole

## E.6 NGram, Spatial index

NGram index (eXist db)
– slúži na presné vyhľadávanie substringov v rámci dlhej sekvencie stringov
– berie do úvahy aj interpunkciu a prázdne znaky, na rozdiel od fulltext indexu

Spatial index (eXist db)
– uchováva niektoré hodnoty z GML[3]

## E.7 Structural/Fulltext/Legacy Fulltext index

Structural index (eXist db)
– automatická indexácia v eXist db
– uchováva si jednotlivé cesty v stromovej štruktúre každej kolekcie

Fulltext index (eXist db)
– je založený na Apache Lucene
– slúži na vyhľadávanie celých slov/reťazcov

Legacy Fulltext index (eXist db)
– slúži na vyhľadávanie dlhých sekvencií slov/viet, ktoré sú následne rozparserované na jednotlivé indexované slová

3. GML = Geography Markup Language, (v preklade: zemepisný značkovací jazyk)

# F Príloha - Druhy joinov

## F.1 Inner/Outer join

Inner join
– ide o základný join pomocou krížového spojovania, kde všetky výstupné riadky vyhovujú podmienke (tj. neobsahuje žiadne `null` hodnoty)
– zväčša je predikát v tvare: cudzí_kľúč = primárny_kľúč

Outer join
– ide o join, ktorého vyhodnocovanie je podobné Inner joinu s tým rozdielom, že podľa typu Outer joinu sa dopĺňajú nevyhovujúce riadky (sprava/zľava/z oboch strán) hodnotou `null`

## F.2 Left/Right join

Left join
– doplnok k Outer joinu, ktorý špecifikuje dopĺňanie hodnôt `null` zľava

Right join
– doplnok k Outer joinu, ktorý špecifikuje dopĺňanie hodnôt `null` sprava

## F.3 Cross join

Cross join
– ide o krížové spájanie tabuliek, ktorého výsledkom je ich kartézsky súčin

## F.4 Self join

Self join
– typ joinu, ktorý berie 1 tabuľku ako 2 rôzne a ktorých hodnoty následne porovnáva

## F.5 Equi/Non-equi join

Equi join
– ide o join, ktorého predikát obsahuje iba znak `'='`

Non-equi join
– ide o join, ktorého predikát obsahuje znaky `'>', '<', '<>',` atď.

# G Príloha - Výsledky

## G.1 Hromadný import

| | GT.M (bez indexu) | GT.M (tree index) | GT.M (hash index) | MongoDB (bez indexu) | MongoDB (tree index) | MongoDB (hash index) | Neo4j (bez indexu) | Neo4j (tree index) | Neo4j (hash index) | MemSQL (bez indexu) | MemSQL (tree index) | MemSQL (hash index) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **5.000** | 0.655 | 0.712 | 0.715 | 0.612 | 0.59 | 0.598 | 1.362 | 1.649 | 1.354 | 1.342 | 1.306 | 1.301 |
| **10.000** | 1.32 | 1.417 | 1.43 | 1.191 | 1.161 | 1.128 | 1.687 | 2.073 | 1.678 | 2.362 | 2.386 | 2.573 |
| **100.000** | 13.3 | 14.596 | 14.801 | 11.492 | 10.926 | 10.815 | 4.418 | 6.363 | 4.83 | 14.668 | 15.499 | 16.16 |
| **500.000** | 66.637 | 72.72 | 73.718 | 56.433 | 53.84 | 55.298 | 17.286 | 27.214 | 18.143 | 70.346 | 74.616 | 76.794 |
| **1.000.000** | 135.034 | 148.656 | 149.706 | 111.292 | 109.21 | 113.241 | 35.093 | 54.484 | 35.817 | 147.746 | 152.315 | 153.111 |

**Tabuľka G.1:** Lepšie výsledky hromadného importu (časy v sekundách)

| | PostgreSQL (bez indexu) | PostgreSQL (tree index) | PostgreSQL (hash index) | eXist db (bez indexu) | eXist db (tree index) | eXist db (hash index) | CSQL (bez indexu) | CSQL (tree index) | CSQL (hash index) |
|---|---|---|---|---|---|---|---|---|---|
| **5.000** | 1.067 | 1.074 | 1.057 | 1.401 | 1.644 | 1.983 | 0.59 | 0.604 | 0.583 |
| **10.000** | 2.033 | 2.089 | 2.078 | 3.227 | 3.142 | 3.662 | 1.051 | 1.102 | 1.125 |
| **100.000** | 19.55 | 20.293 | 20.37 | 27.309 | 33.146 | 31.753 | 13.606 | 14.31 | 14.989 |
| **500.000** | 97.785 | 101.619 | 102.823 | 133.651 | 164.171 | 156.357 | 122.74 | 160.516 | 230.908 |
| **1.000.000** | 200.563 | 207.053 | 210.285 | 269.12 | 317.417 | 321.122 | 352.825 | 482.747 | 525.71 |

**Tabuľka G.2:** Horšie výsledky hromadného importu (časy v sekundách)

## G.2 Sekvenčný import

| | GT.M (bez indexu) | GT.M (tree index) | GT.M (hash index) | PostgreSQL (bez indexu) | PostgreSQL (tree index) | PostgreSQL (hash index) | MongoDB (bez indexu) | MongoDB (tree index) | MongoDB (hash index) | MemSQL (bez indexu) | MemSQL (tree index) | MemSQL (hash index) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **5.000** | 1.426 | 1.468 | 1.496 | 1.81 | 1.74 | 1.876 | 1.703 | 1.741 | 1.774 | 0.844 | 0.836 | 0.875 |
| **10.000** | 2.521 | 2.655 | 2.785 | 3.265 | 3.265 | 3.276 | 2.884 | 3.034 | 3.048 | 1.57 | 1.613 | 1.56 |
| **100.000** | 23.662 | 25.813 | 26.98 | 25.888 | 26.25 | 26.559 | 18.63 | 20.063 | 20.505 | 9.629 | 9.782 | 9.787 |
| **500.000** | 120.689 | 133.383 | 131.616 | 126.802 | 129.837 | 130.121 | 87.29 | 98.331 | 102.23 | 43.179 | 50.351 | 47.334 |
| **1.000.000** | 231.662 | 251.813 | 256.145 | 254.872 | 258.836 | 263.411 | 173.868 | 197.211 | 202.253 | 99.708 | 109.924 | 109.924 |

**Tabuľka G.3:** Lepšie výsledky sekvenčného importu (časy v sekundách)

| | Neo4j (bez indexu) | Neo4j (tree index) | Neo4j (hash index) | eXist db (bez indexu) | eXist db (tree index) | eXist db (hash index) | CSQL (bez indexu) | CSQL (tree index) | CSQL (hash index) |
|---|---|---|---|---|---|---|---|---|---|
| **5.000** | 9.935 | 10.242 | 10.331 | 10.362 | 11.223 | 11.842 | 0.529 | 0.497 | 0.52 |
| **10.000** | 15.687 | 16.214 | 15.589 | 28.651 | 30.201 | 32.305 | 1.176 | 1.382 | 1.406 |
| **100.000** | 109.646 | 109.885 | 109.483 | 382.785 | 416.713 | 444.421 | 51.24 | 54.402 | 59.571 |
| **500.000** | 537.097 | 549.004 | 541.091 | 1995.121 | 2122.811 | 2322.984 | 1284.813 | 1555.701 | 2100.214 |
| **1.000.000** | 1136.761 | 1177.061 | 1150.137 | 4337.666 | 4598.112 | 4975.411 | 2595.581 | 3142.83 | 4242.856 |

**Tabuľka G.4:** Horšie výsledky sekvenčného importu (časy v sekundách)

## G.3 Tree index

| | PostgreSQL (bez indexu) | PostgreSQL (s indexom) | MongoDB (bez indexu) | MongoDB (s indexom) | MemSQL (bez indexu) | MemSQL (s indexom) | CSQL (bez indexu) | CSQL (s indexom) |
|---|---|---|---|---|---|---|---|---|
| **5.000** | 0.028 | 0.028 | 0.06 | 0.058 | 0.015 | 0.013 | 0.013 | 0.012 |
| **10.000** | 0.037 | 0.035 | 0.058 | 0.058 | 0.018 | 0.018 | 0.021 | 0.018 |
| **100.000** | 0.082 | 0.067 | 0.06 | 0.059 | 0.052 | 0.052 | 0.091 | 0.083 |
| **500.000** | 0.38 | 0.28 | 0.062 | 0.059 | 0.131 | 0.101 | 0.156 | 0.14 |
| **1.000.000** | 0.75 | 0.668 | 0.059 | 0.058 | 0.256 | 0.175 | 0.182 | 0.16 |

**Tabuľka G.5:** Lepšie výsledky 1.selektu (časy v sekundách)

| | GT.M (bez indexu) | GT.M (s indexom) | Neo4j (bez indexu) | Neo4j (s indexom) | eXist db (bez indexu) | eXist db (s indexom) |
|---|---|---|---|---|---|---|
| **5.000** | 0.03 | 0.011 | 0.467 | 0.168 | 0.2 | 0.074 |
| **10.000** | 0.096 | 0.013 | 0.597 | 0.17 | 0.26 | 0.125 |
| **100.000** | 0.926 | 0.064 | 1.717 | 0.2 | 1.18 | 0.53 |
| **500.000** | 5.371 | 0.65 | 5.573 | 0.255 | 5.531 | 2.46 |
| **1.000.000** | 13.23 | 1.351 | 9.817 | 0.296 | 12.177 | 5.049 |

**Tabuľka G.6:** Horšie výsledky 1.selektu (časy v sekundách)

## G.4 Hash index

| | PostgreSQL (bez indexu) | PostgreSQL (s indexom) | MongoDB (bez indexu) | MongoDB(s indexom) | MemSQL (bez indexu) | MemSQL (s indexom) | CSQL (bez indexu) | CSQL (s indexom) |
|---|---|---|---|---|---|---|---|---|
| **5.000** | 0.015 | 0.013 | 0.053 | 0.05 | 0.001 | 0.001 | 0.009 | 0.009 |
| **10.000** | 0.017 | 0.014 | 0.058 | 0.054 | 0.003 | 0.003 | 0.01 | 0.01 |
| **100.000** | 0.039 | 0.014 | 0.107 | 0.056 | 0.018 | 0.018 | 0.027 | 0.026 |
| **500.000** | 0.136 | 0.017 | 0.263 | 0.059 | 0.084 | 0.084 | 0.047 | 0.038 |
| **1.000.000** | 0.255 | 0.019 | 0.555 | 0.061 | 0.205 | 0.16 | 0.05 | 0.042 |

**Tabuľka G.7:** Lepšie výsledky 2.selektu (časy v sekundách)

| | GT.M | Neo4j (bez indexu) | Neo4j (s indexom) | eXist db (bez indexu) | eXist db (s indexom) |
|---|---|---|---|---|---|
| **5.000** | 0.008 | 0.391 | 0.149 | 0.1 | 0.01 |
| **10.000** | 0.024 | 0.505 | 0.15 | 0.122 | 0.01 |
| **100.000** | 0.177 | 1.306 | 0.163 | 0.61 | 0.012 |
| **500.000** | 0.882 | 2.4 | 0.184 | 3.206 | 0.024 |
| **1.000.000** | 1.753 | 3.849 | 0.215 | 6.461 | 0.04 |

**Tabuľka G.8:** Horšie výsledky 2.selektu (časy v sekundách)

## G.5 Join

| | GT.M | MongoDB | Neo4j |
|---|---|---|---|
| **5.000** | 0.013 | 0.113 | 0.527 |
| **10.000** | 0.036 | 0.18 | 0.637 |
| **100.000** | 0.924 | 0.68 | 1.471 |
| **500.000** | 10.827 | 2.599 | 3.863 |
| **1.000.000** | 62.5 | 3.613 | 6.324 |

**Tabuľka G.9:** Lepšie výsledky joinu (časy v sekundách)

| | eXist db | PostgreSQL | MemSQL | CSQL |
|---|---|---|---|---|
| **5.000** | 195.89 | 0.075 | 2.883 | 7.586 |
| **10.000** | 310.777 | 0.113 | 10.025 | 31.143 |
| **100.000** | 33870.055 | 70.491 | 1468.944 | 3216.489 |

**Tabuľka G.10:** Horšie výsledky joinu (časy v sekundách)

## G.6 Paralelizácia dotazov

| | PostgreSQL (seri6.5al) | PostgreSQL (parallel) | MongoDB (serial) | MongoDB (parallel) | Neo4j (serial) | Neo4j (parallel) | MemSQL (serial) | MemSQL (parallel) | CSQL (serial) | CSQL (parallel) |
|---|---|---|---|---|---|---|---|---|---|---|
| **5.000** | 0.119 | 0.065 | 0.0564 | 0.087 | 0.613 | 0.44 | 0.037 | 0.019 | 0.194 | 0.06 |
| **10.000** | 0.237 | 0.078 | 0.0566 | 0.091 | 0.896 | 0.494 | 0.069 | 0.025 | 0.363 | 0.079 |
| **100.000** | 2.188 | 0.368 | 0.57 | 0.099 | 3.439 | 1.43 | 0.602 | 0.122 | 3.021 | 0.41 |
| **500.000** | 11.518 | 2.376 | 0.576 | 0.105 | 9.748 | 5.312 | 3.479 | 1.345 | 6.074 | 0.761 |
| **1.000.000** | 23.127 | 9.687 | 0.576 | 0.107 | 19.997 | 10.415 | 9.438 | 6.546 | 6.121 | 0.787 |

**Tabuľka G.11:** Lepšie výsledky paralelizácie dotazov (časy v sekundách)

| | GT.M (serial) | GT.M (parallel) | eXist db (serial) | eXist db (parallel) |
|---|---|---|---|---|
| **5.000** | 0.825 | 0.524 | 0.674 | 0.318 |
| **10.000** | 3.215 | 2.838 | 0.812 | 0.381 |
| **100.000** | 48.694 | 40.314 | 4.99 | 3.191 |
| **500.000** | 102.182 | 94.764 | 25.429 | 21.071 |
| **1.000.000** | 282.1 | 201.88 | 108.163 | 53.926 |

**Tabuľka G.12:** Horšie výsledky paralelizácie dotazov (časy v sekundách)