Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Uchování a zpracování velkých dat typu JSON

Plzeň 2017 Ondřej Hovjacký

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: **2016/2017**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondřej HOVJACKÝ**
Osobní číslo:	**A15N0062P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Název tématu:	**Uchování a zpracování velkých dat typu JSON**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování:

1. Seznamte se s možnostmi uchování dat typu JSON v SQL a NoSQL databázích.
2. Analyzujte typy užití takových dat a navrhněte odpovídající datové kolekce.
3. Navrhněte systém, který bude zpracovávat velký objem dat řádu statisíců záznamů typu JSON a následně umožňovat rychlé dotazy nad daty a jejich statistické zpracování.
4. Implementujte navržený systém.
5. Porovnejte a vyhodnoťte výsledky při použití různých metod ukládání dat podle bodu 1. a 2.

Rozsah grafických prací: **dle potřeby**

Rozsah kvalifikační práce: **doporuč. 50 s. původního textu**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Doc. Ing. Josef Steinberger, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **1. září 2016**

Termín odevzdání diplomové práce: **18. května 2017**

Doc. RNDr. Miroslav Lávička, Ph.D.
děkan

L.S.

Doc. Ing. Přemysl Brada, MSc. PhD.
vedoucí katedry

V Plzni dne 12. září 2016

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 23. června 2017

Ondřej Hovjacký

Poděkování

Chtěl bych poděkovat vedoucím práce, panu Josefu Steinbergerovi z katedry informatiky a Martinu Horovi a jeho kolegům z firmy RTsoft za dobré návrhy a věcné připomínky. Hlavně bych však chtěl poděkovat své rodině – manželce Kristýně a dceři Noemi – za obrovskou trpělivost a podporu při tvorbě práce.

Abstract

This thesis inspects methods of storing big data (especially of type JSON), explores these options in greater detail and describes design and implementation of a system which provides a unified access to them. It also covers and evaluates tests of several NoSQL databases and MySQL.

Abstrakt

Tato práce se zabývá možnostmi ukládání velkých dat (především typu JSON), prozkoumává detailněji jednotlivé varianty a popisuje návrh a sestavení systému, který k nim umožňuje jednotný přístup. Dále popisuje a vyhodnocuje provedené testy několika NoSQL databází a MySQL.

Obsah

1 Úvod

V dnešní době je potřeba ukládat stále větší množství dat. Následně je ale také zapotřebí z nich něco vyčíst, získat informace. Úkolem této práce je zjistit, jaké jsou možnosti ukládání velkých dat (především typu JSON), prozkoumat detailněji jednotlivé varianty a sestavit systém, který k nim umožní jednotný přístup.

Zadání bylo sestaveno ve spolupráci s firmou RTsoft, která chtěla najít a vyhodnotit jednotlivé možnosti ukládání dat, aby mohla u reálných projektů efektivně vybírat, jakou databázi použít pro lepší výkon.

Jedním z hlavních záměrů této práce je také prozkoumat výkonnost ukládání JSON dokumentů do MySQL databáze, což by mohlo usnadnit vývoj projektu, který by používal jen jeden typ databáze místo například dvou. Spolu s několika nejznámějšími a nejpoužívanějšími NoSQL databázemi se tedy budeme zabývat těmito:

- MySQL,
- MongoDB,
- Apache Cassandra a
- Elasticsearch [1].

V poslední fázi se pak zaměříme na testování jednotlivých variant a jejich zhodnocení jak z hlediska rychlosti, tak po stránce pohodlnosti použití.

[1] Tento seznam byl sestaven na základě žebříčku serveru DB-Engines [5], který dlouhodobě monitoruje používání databází několika různými metodami.

2 Možnosti ukládání dat typu JSON

Na trhu je velké množství databázových systémů, které se zabývají ukládáním dat typu JSON, neboli *JavaScript Object Notation*. Tento formát se v poslední době stal velmi populárním, jelikož je lehce čitelný a srozumitelný a je možné jej použít napříč programovacími jazyky díky jeho textové podobě, nikoliv tedy jen v JavaScriptu, jak by se z názvu mohlo zdát.

Stejné vlastnosti má také formát XML, nicméně se v několika ohledech liší, XML:

- používá tagy – otevírací a uzavírací – což výrazně prodlužuje délku,
- je pro člověka hůře čitelný, zejména kvůli již zmíněným tagům,
- nemá podporu polí, což je velmi významný faktor ovlivňující výběr,
- je složitější jej parsovat [1].

2.1 Typy NoSQL databází

JSON data se asi nejčastěji a nejjednodušeji ukládají v NoSQL databázích. Někteří říkají, že název vznikl z „no RDBMS“ (*Relational Database Management System* – relační databáze), i když měl spíše znamenat *no relational* (ne relační). Dále byl také navržen název *NonRel*, ale neuchytil se. Někteří se pak snažili navrhnout, že originální název *NoSQL* znamená „not only SQL“ [24]. Ať už to bylo jakkoliv, tyto databáze se liší od klasických relačních databází tím, že většinou nepodporují tzv. ACID vlastnosti (*Atomicity, Consistency, Isolation, Durability*), nýbrž mají vlastnosti BASE (*Basically Available, Soft-state, Eventually consistent*) [20]. Nemají tedy většinou úplnou konzistenci, ale díky tomu mohou být mnohem rychlejší. Pěkná je definice Prof. Dr. Stefan Edlicha ze serveru *nosql-database.org*:

> *Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable [17].*

> Volným překladem: Databáze další generace z většiny splňující některé z bodů: jsou nerelační, distribuované, open-source a horizontálně škálovatelné.

Jak je vidět, není vůbec jednoznačná, a proto existuje mnoho různých skupin NoSQL databází.

Oproti relačním databázím se tedy většinou dají lépe horizontálně škálovat, mnoho jich poskytuje více či méně automatické sdružování uzlů do *clusteru*, možnost rozdělit data do několika uzlů při velkém objemu dat (tzv. *sharding*) a pro větší dostupnost (když nějaký uzel přestane odpovídat) a zvýšení výkonu (paralelní obsluha) je možné mít data *replikovaná*.

Horizontální škálování (zapojení většího počtu strojů, které nemusí být příliš výkonné) je v dnešní době upřednostňováno před vertikálním škálováním (pokud výkon nevyhovuje, nakoupí se lepší stroj, přidá se pameť apod.).

Pod názvem NoSQL se skrývá několik různých skupin databází, zde si projdeme ty nejpoužívanější.

2.1.1 Key/value databáze

Jedná se o velmi jednoduchý model databáze, kde se ukládá vždy pár *klíč – hodnota*. Podobá se hash tabulkám, klíč je zároveň indexem, takže vyhledání záznamu je velmi rychlé. Preferuje se hlavně rychlost a možnost obsluhy mnoha požadavků paralelně na úkor konzistence. Jednou ze slabin je nemožnost vytváření schéma, takže tyto databáze mají jen omezené využití [20].

Aplikace, která databázi používá, musí vědět, jaká data uložit a jaká data od databáze očekávat, protože databáze se o typ uložených dat většinou nestará. V této jednoduchosti však spočívá jejich síla – rychlost. Poskytují zpravidla jen tři základní operace – čtení, zápis a smazání, vše podle klíče. Většina systémů ještě umožňuje vytvářet *buckets* (skupiny nebo jmenné prostory) hodnot, aby se mohly hodnoty alespoň nějak třídit [19].

Asi nejznámější představitel tohoto typu je *Amazon DynamoDB* [16] a *Redis*.

Redis

Redis patří ke skupině hybridně persistentních databází, primárně funguje jako *in-memory* databáze, která má veškerá data v operační paměti, ale v pravidelných intervalech je ukládá

na disk. Snižuje tak výrazně čas čtení/zápisu [19]. K ukládání dat používá jednoduchou strukturu nazývající se *Simple Dynamic String* (SDS), která sestává ze tří částí:

- pole znaků se samotnými daty,
- délka pole znaků,
- počet dalších bytů volných k použití [24].

Redis může být replikován *master-slave* architekturou za použití i několika *slave* serverů. Ty umožňují čtení, které však může poskytovat stará data, pokud ještě nedošlo k synchronizaci s *master* serverem [22].

Redis nabízí rozšířené možnosti použití datových struktur – seznamy, množiny či asociativní pole – spolu s operacemi nad němi pro lepší práci s daty [19].

2.1.2 Sloupcové databáze

Tyto databáze se „naoko“ podobají relačním databázím nejvíce. Lze si je představit jako databáze, které ukládají data do sloupců, ovšem samozřejmě jen neprázdné údaje, takže prázdné buňky řádků nezabírají místo. Data však nejsou fakticky uložena v tabulce, ale ve struktuře zanořených dat *klíč – hodnota*. Sloupcově orientované databáze mají sloupcové rodiny (*column families*), které obsahují významově blízké sloupce. Tyto se musí předem nadefinovat, počet sloupců v nich je už většinou bez počátečního schéma a může se u jednotlivých řádků lišit. Dále zpravidla umožňují verzování podle času, takže se uchovává i historie jednotlivých buněk [24].

Níže vidíme ukázku uložení dat v podobě JSON dokumentu. Každá rodina sloupců má jeden dokument, první úroveň se skládá z identifikátorů řádek a druhá jsou jednotlivé sloupce s hodnotou uloženou v určitém čase [19].

Mezi tyto databáze patří *Apache Cassandra* a *Big Table* od společnosti Google, která je vybudována nad souborovým systémem *Google File System*. Používá jeden master server, který řídí umístění dat na jednotlivé servery, *garbage kolekci* apod., a tzv. *tablet* servery, kde jsou uložena samotná data [13]. Je designována pro škálování na tisíce strojů a používá jej např. Gmail a YouTube [20].

```
// "column_family_1"
{
  "row_key_1": {
    "column_1": {
      timestamp_1: "value_1"
    },
    "column_2": {
      timestamp_2: "value_2",
      timestamp_4: "value_3"
    }
  }
},
// "column_family_2"
{
  "row_key_1": {
    "column_3": {
      timestamp_3: "value"
    }
  }
}
```

Cassandra

Apache Cassandra patří do sloupcově orientovaných úložišť. Vychází z Google Bigtable [13] a Amazon Dynamo [16]. Jedná se o typ úložiště, které může pojmout velké množství dat (TB-PB), jež ukládá do řádků tabulky mající až tisíce sloupců [22]. Databáze je navíc decentralizovaná, takže žádný prvek nepředstavuje tzv. *single point of failure* – prvek, který má po selhání za následek výpadek. Všechny servery jsou si rovny a tudíž i zatížení se může rovnoměrně rozložit. Cassandra používá *Memtables* – struktury v hlavní paměti, do kterých se zapisuje, a z kterých jsou data zapsána na disk až když překročí stanovenou velikost nebo když *CommitLog* (který shromažďuje provedené změny) dosáhne své maximální velikosti [11].

Cassandra používá CQL jazyk, který je velmi podobný jazyku SQL. Od verze 2.1 se její *rodiny sloupců* (*column-families*) změnily na *tabulky* (*tables*), takže už není *schema-less*, tzn. že se tabulky musí definovat pře-

dem [19], např:

```
CREATE TABLE users (
  login text PRIMARY KEY,
  name text,
  emails set<text>,        /* sloupec typu mnozina */
  profile map<text, text> /* s. typu asociativni pole */
);
```

Dotaz pak někdy vypadá úplně shodně s SQL:

```
SELECT * FROM users WHERE name = 'John Smith';
```

2.1.3 Dokumentové databáze

Dokumentové databáze slouží k ukládání dat ve formě dokumentů. Díky tomu jsou nejuniverzálnějšími databázemi, protože jsou schopné uložit téměř cokoliv. Ukládají data ve standardních formátech jako např. JSON, BSON (binární reprezentace JSON obohacená o některé další datové typy) nebo XML [19]. Není definováno schéma, jednotlivé dokumenty si můžou být podobné, mohou se ale také úplně lišit. Odkazuje se na ně pomocí unikátního klíče, takže se podobají *key/value* databázím s tím rozdílem, že hodnotou je celý dokument [20].

Dokument ve formátu JSON může vypadat např. takto:

```
{
  "_id": 123456,
  "firstname": "John",
  "lastname": "Doe",
  "address": {
    "street": "Short",
    "city": "London",
    "zip": "xyx"
  }
}
```

Známými implementacemi jsou např. *MongoDB* nebo *CouchDB*, která používá JSON dokumenty a poskytuje *RESTful API* pro komunikaci se

serverem [20]. Jako dokumentovou databázi můžeme používat také Elasticsearch a nově MySQL.

MongoDB

MongoDB je dokumentová databáze, která ukládá data ve formátu *BSON*. Jednotlivé *dokumenty* tvoří *kolekce*, které patří do *databáze*.

Od verze 3.0 je možné jako interní metodu ukládání dat využívat *WiredTiger Storage Engine*, který se stal základní metodou ve verzi 3.2 [10]. Ten umožňuje efektivnější ukládání s rychlejším čtením dat a větším výkonem při velkém zatížení.

V databázi je možné hledat data dotazy jako (věk je vyšší nebo roven 18 a řazení podle příjmení sestupně):

```
db.users.find( { age: { $gte: 18 } } ).sort( { lastname: -1 } )
```

Databázi je možné distribuovat, MongoDB používá *master-slave* architekturu, tzn. že master server jako jediný provádí zápis a z ostatních uzlů je možné pouze číst. Pokud master server selže, zbylé servery si samy zvolí nového mastera. Toto se stane i při výpadku spojení master serveru a zbytku serverů, nový master server je zvolen, pokud je serverů více než polovina [19].

Elasticsearch

Elasticsearch slouží k ukládání a velmi rychlému vyhledávání a analýze dat. I když se primárně neřadí mezi dokumentové databáze, lze jej tak používat – základní jednotkou je totiž *dokument*. Dokumenty je možno vložit do některého z *typů*. *Typ* náleží *indexu*. Je to podobná struktura jako *záznam v tabulce*, *tabulka* a *databáze*, kterou známe z relačních databází.

Všechna data jsou uložena v jednotlivých *uzlech*, jež dohromady tvoří *cluster*, ke kterému se automaticky připojí podle názvu. *Cluster* může být tvořen i pouhým jedním *uzlem* [8].

Elasticsearch interně používá *Apache Lucene*, který je implementován v Javě [9]. Data jsou ukládána do invertovaného indexu, díky němuž je pak možné v nich rychle hledat.

Pro připojení nabízí *RESTful API*, v URL je oddělen lomítky název indexu, typu a operace, kterou chceme vykonat. Další data jsou v těle požadavku. Stejný dotaz uvedený u MongoDB vypadá následovně:

```
GET db/users/_search
{
  "query": {
    "range": {
      "age": {
        "gte": 18
      }
    }
  },
  "sort": [{
    "lastname": "desc"
  }]
}
```

MySQL

MySQL jako taková se samozřejmě neřadí mezi dokumentové databáze, ale i tento koncept se postupně snaží vytvořit (viz dále). Od verze 5.7.8 podporuje datový typ JSON. Dříve samozřejmě bylo možné uložit JSON jako text do textového sloupce, nyní ale MySQL poskytuje:

- validaci JSON dokumentů ukládaných do sloupce typu JSON,
- optimalizované ukládání tohoto sloupce.

Data nejsou uložena v textovém formátu, ale v interní binární podobě, která umožňuje rychlejší přístup. Při čtení se tak už nemusí jednotlivé JSON dokumenty parsovat [6].

Při práci je možné použít funkce jako `JSON_OBJECT`, `JSON_ARRAY` nebo `JSON_MERGE`, které z textové reprezentace vytvoří interní JSON a při tom jej validují. Pokud není dokument validní, dojde k chybě. Pro získání jednotlivých hodnot se použije `JSON_EXTRACT`, kde se ve druhém parametru specifikuje cesta k hodnotám, které chceme dostat.

Pro úpravu již existujících dokumentů lze použít funkce:

- `JSON_SET` – vloží novou hodnotu nebo upraví již existující,
- `JSON_INSERT` – vloží novou hodnotu, ale neupravuje existující,
- `JSON_REPLACE` – nahradí hodnotu,
- `JSON_REMOVE` – odstraní hodnotu [6].

MySQL a dokumentová databáze

MySQL nově podporuje také možnost dokumentové databáze, i když zatím ne v produkční verzi. Chce se tím přiblížit jiným NoSQL dokumentovým databázím. Od verze 5.7.12 je součástí MySQL *X Plugin*, který poskytuje komunikaci s databází pomocí *X Protocol*u. Různé konektory pro různé programovací jazyky nebo MySQL Shell pak poskytují připojení k databázi [6] a rozhraní *X DevAPI* [7].

Díky této funkci není třeba dopředu definovat tabulky, do databáze lze uložit v podstatě jakýkoliv JSON dokument.

Pomocí rozhraní *X DevAPI* můžeme pracovat (s drobnými úpravami podle programovacího jazyka) s funkcemi jako:

```
collection.add({ "key": "value" })
collection.find("age > 5").sort(["name desc"]).limit(2)
```

Zde vidíme velkou podobnost např. s MongoDB. Kolekce, se kterou takto pracujeme, je ve skutečnosti standardní SQL tabulka se dvěma sloupci:

```
CREATE TABLE `collection` (
  `doc` JSON NULL DEFAULT NULL,
  `_id` VARCHAR(32) AS
    (json_unquote(json_extract(`doc`,"$._id"))) STORED,
  PRIMARY KEY (`_id`)
) COLLATE="utf8_general_ci" ENGINE=InnoDB;
```

X Plugin tedy poskytuje nové možnosti komunikace s databází a práce s daty pomocí jednoduchých CRUD (*Create, Read, Update, Delete*) operací. Konektorů je však zatím málo, jmenovitě pro Javu, Node.js, .NET, Python a C++.

Důležitá je samozřejmě podpora vytváření indexů nad určitými poli, bez kterých by bylo vyhledávání velmi pomalé. MySQL toto umožňuje pomocí

sekundárních indexů nad generovanými virtuálními sloupci [6]. Pokud budeme chtít vytvořit index nad prvkem *name* následujícího JSON dokumentu (cesta je `$.name`, `$` označuje kořen dokumentu), provedeme to dvěma dotazy:

```
{
  "_id": 1,
  "name": "John",
  "surname": "Doe"
}
```

```
ALTER TABLE `collection` ADD COLUMN `name` VARCHAR(255)
GENERATED ALWAYS AS
  (JSON_UNQUOTE(JSON_EXTRACT(`doc`,"$.name"))) VIRTUAL;
ALTER TABLE `collection` ADD KEY (`name`);
```

Při použití následujícího dotazu je již použit index definovaný v předchozím příkladu kódu:

```
SELECT `doc` FROM `collection`
WHERE (JSON_EXTRACT(doc,"$.name") = "John");
```

2.1.4 Grafové databáze

Grafové databáze ukládají data ve formě grafů, které sestávají z hran a vrcholů. Vrcholy můžou mít různé vlastnosti. Grafy se dají rychle procházet a jsou vhodné pro semi-strukturovaná data. Používají se tam, kde jsou vztahy mezi entitami, zejména pro sociální sítě, správu účtů apod. [20]. Nevhodné jsou pro velká data, která chceme distribuovat. To se u grafových databází na rozdíl od těch předchozích dělá velmi obtížně [19].

Neo4j

Příkladem je databáze *Neo4j*, která je implementovaná v Javě a tudíž je přenositelná. Uzly mohou mít atributy tvořené klíčem (*String*) a hodnotou (jednoduché datové typy, *String* nebo příp. pole hodnot jednoduchých datových typů).

Na rozdíl od ostatních NoSQL databází zachovává vlastnosti *ACID* [19].

2.1.5 Objektově orientované databáze

Tyto databáze ukládají data jako objekty a poskytují velmi podobné vlastnosti jako objektově orientované programování, např. zapouzdření, polymorfismus nebo dědičnost. Vhodné jsou pro aplikace, kde jsou složité vztahy mezi objekty. Nevýhodou je vazba na programovací jazyk, ve kterém se databáze používá [20].

Objektově orientovanou databází je např. *db4o*.

2.2 Možnosti využití dat

Mnoho skupin znamená ještě více způsobů využití. Podle užití je také třeba vybírat databázi, protože se liší v různých kritériích. Jedno z nich může například být, jestli bude používána výhradně pro zápis nebo naopak pro čtení, či bude poměr operací podobný. Existují různé úrovně uchování dat – můžou být nějakou dobu uložena jen v paměti nebo se vždy ukládat na disk, v případě distribuované databáze se při zápisu může čekat na různý počet odpovědí jednotlivých uzlů apod. (podrobněji rozepsáno v kapitole 3.1).

V dnešní době stoupá obliba a využití IoT (*Internet of Things*), což vyúsťuje v potřebu ukládat a zpracovávat veliké množství dat, která zařízení sbírají, pro okamžité nebo pozdější vyhodnocování a analýzu.

Pro jednotlivé způsoby zatížení nebo jim podobné se provádějí testování pomocí benchmarků, které většinou zkoumají průchodnost a dobu odezvy jednotlivých požadavků. Nyní se na některé z nich podíváme a v další části práce provedeme naše vlastní testy.

3 Existující benchmark a testování

Pro testování databází existují různé benchmarky porovnávající různé vlastnosti. Mezi nejznámější asi patří benchmarky od *TPC* (*The Transaction Processing Performance Council*) [12], který existuje již od roku 1988 a definuje množství databázových a transakčních benchmarků, ovšem zaměřuje se hlavně na hardware a klasické relační databáze.

Pro NoSQL databáze existuje *YCSB* (*Yahoo! Cloud Serving Benchmark*) [14], který poskytuje sadu benchmarků pro různá zatížení a různé databáze. Jelikož se jedná o open-source, databází možnými s ním hodnotit stále přibývá. V době psaní této práce jich je již přes třicet.

3.1 YCSB

YCSB [14] je nástroj pro provádění testů výkonu databází. Dále se podíváme i na nějaké výsledky, které s ním dosáhli jak jeho tvůrci tak jiní. Poskytuje klienta (*YCSB Client*), který se stará o posílání požadavků na servery. Tyto požadavky vytváří podle definovaných způsobů zatížení (*Core workloads*) podobných těm výše popsaným. Umožňuje také jednoduše definovat vlastní rozložení zatížení.

Při výběru NoSQL databáze je vždy nutné dělat kompromisy, protože neexistuje žádné ideální řešení. Mezi tyto kompromisy patří:

- **Rychlost čtení vs. rychlost zápisu** jsou ovlivňovány způsobem uložení dat na disku. Pokud je požadována velká rychlost čtení, je potřeba, aby data jednoho záznamu byla uložena na jednom místě na disku a byl k nim tak rychlý přístup. Při zápisu, který data aktualizuje, pak musíme každý záznam vyhledat a upravit, což zápis výrazně zpomalí. Naproti tomu systémy, které data ukládají jako logy, nemusejí při zápisu nic hledat a nová data jednoduše připíšou na první volné místo. To však negativně ovlivní čtení, které musí projít tyto záznamy a z nich zkonstruovat aktuální podobu dat.

- **Doba odezvy vs. trvanlivost** závisí na tom, kdy se data fakticky uloží na disk. Pokud se tak stane před dokončením požadavku, klient může být ujištěn, že jsou data v pořádku uložena, což ale opět prodlouží dobu odezvy. Druhou možností je data uložit až po odpovědi, aby klient nemusel čekat, avšak např. při náhlém pádu databáze může dojít ke ztrátě těchto dat, aniž by to uživatel věděl.
- **Synchronní vs. asynchronní replikace** ovlivňují způsob ukládání dat na ostatní uzly v clusteru. Synchronní ukládání provede zápis do všech replik před odpovědí klientovi, čímž výrazně navyšuje dobu odezvy, zejména pokud je některý ze serverů nedostupný. Zaručuje však konzistenci, která není u asynchronní replikace zaručena.

Tvůrci *YCSB* hodnotili dvě kritéria – výkon a škálovatelnost databází.

Testování výkonu se zaměřuje na dobu odezvy požadavků, když je databáze pod velkým zatížením. Postupně můžeme přidávat požadavky do doby, než se průchodnost přestane zvyšovat a měříme při tom dobu odezvy. Hardwareová konfigurace je při tom konstantní.

Testování škálovatelnosti zjišťuje dopad přidávání dalších strojů na výkon. Jednak se může testovat doba odezvy na menším clusteru s menšími daty a na větším clusteru s proporcionálně většími daty, přičemž by měla zůstat konstantní. Druhou možností je přidávat nové servery za běhu a zjišťovat, jestli se výkon zlepší, což by u dobrých systémů mělo nastat.

YCSB obsahuje základní sestavu různých kombinací zátěže (*YCSB Core Package*), která se nesnaží napodobit konkrétní aplikace, ale vychází z několika různých případů užití databází (viz tabulku 3.1). Uživatelé si samozřejmě mohou vytvořit vlastní sestavy. Základní sestava používá tyto operace:

- **Vložení (*insert*)** – vloží nový záznam.
- **Úprava (*update*)** – upraví jednu hodnotu existujícího záznamu.
- **Čtení (*read*)** – přečte nějakou hodnotu nebo celý záznam.
- **Procházení (*range scan*)** – projde náhodný počet záznamů jdoucích po sobě od náhodně vygenerovaného klíče.

Pro náhodný výběr záznamu používá tři pravděpodobnostní rozdělení:

- **Rovnoměrné** – všechny záznamy mají stejnou pravděpodobnost výběru.

- **Zipfian** – některé záznamy jsou oblíbené, tudíž mají větší pravděpodobnost výběru, většina ji má však malou.
- **Poslední** – největší pravděpodobnost výběru mají naposledy vložené záznamy.

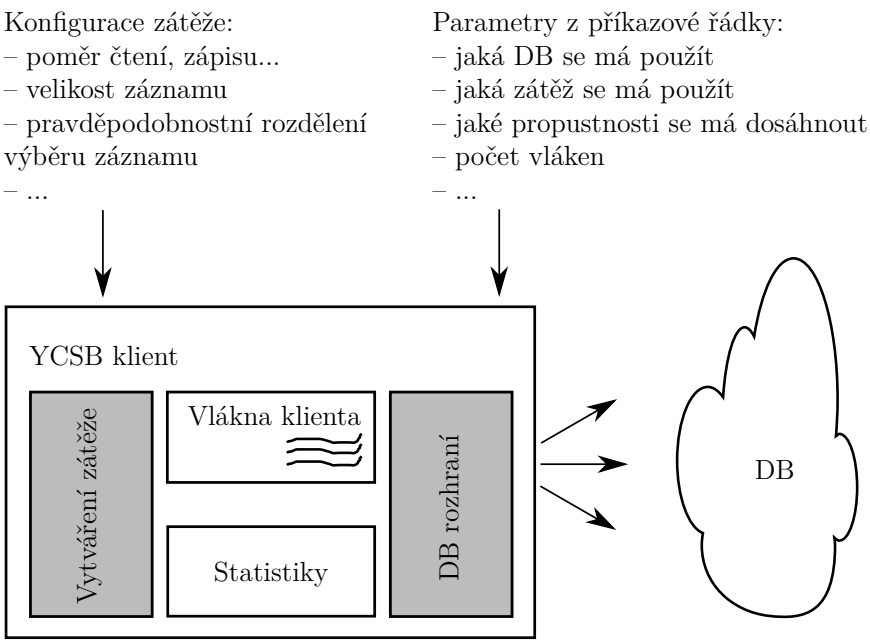
Zatížení	**Operace**	**Výběr záznamu**	**Příklad aplikace**
Převážně úpravy	Čtení: 50% Úprava: 50%	Zipfian	Úložiště session zaznamenávající operace prováděné uživatelem.
Převážně čtení	Čtení: 95% Úprava: 5%	Zipfian	Označování fotek tagy – přidání tagu je úprava, ale většina operací jsou čtení.
Pouze čtení	Čtení: 100%	Zipfian	Cache, jejíž obsah je sestaven jinde (např. Hadoop).
Čtení nejnovějších záznamů	Čtení: 95% Vložení: 5%	Poslední	Čtení statusů na sociálních sítích, lidé chtějí vidět ty nejnovější.
Krátké intervaly	Procházení: 50% Vložení: 50%	Zipfian / rovnoměrné	Konverzace ve vláknech, projitím chceme nalézt všechny příspěvky dané konverzace (seskupené podle ID vlákna).

Tabulka 3.1: Kombinace operací při různých způsobech zatížení

Pro každý výběr operace, která se použije, se používá multinomické rozdělení s parametry podle momentálně používané skupiny zátěže (tj. každá operace je vybrána s určitou pravděpodobností, součet pravděpodobností je roven jedné).

Na obrázku 3.1 je zobrazena architektura *YCSB* klienta, který je napsán v Javě. Základní operací je vytváření vláken komponentou Vytváření zátěže (*Workload Executor*). Vlákna generují dotazy na databázi, měří dobu odezvy a průchodnost a data předávají statistickému modulu. Ten na konci měření spočítá průměrnou dobu odezvy, 95. a 99. percentil.

Klient přijímá dva druhy konfigurace:



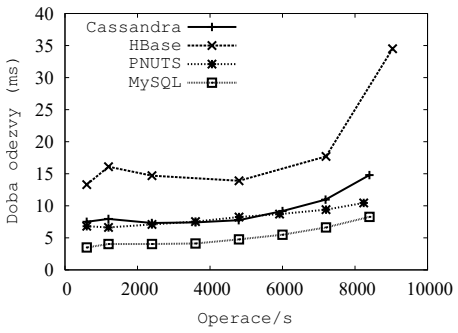
Obrázek 3.1: Architektura YCSB klienta

- **Konfigurace zátěže** určuje jaké bude rozložení operací, jaká pravděpodobnostní rozdělení se použijí apod. Je nezávislá na databázi, která se bude testovat.
- **Parametry z příkazové řádky** (můžou být předány v konfiguračním souboru) udávající jaká databáze se má použít a její parametry, počet klientských vláken apod.

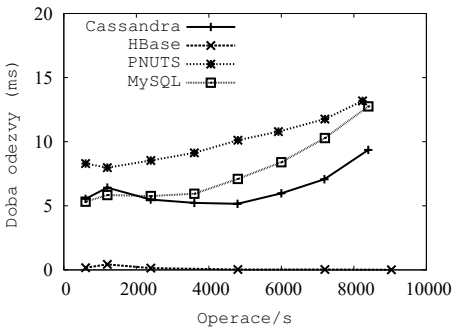
Při testování několika databází tedy konfigurace zátěže zůstává konstantní a mění se pouze parametry.

Po sestavení tohoto frameworku vývojáři v Yahoo! provedli testování čtyř databází v těchto verzích: Cassandra 0.5.0, HBase 0.20.3, MySQL 5.1.24 (pro PNUTS) a distribuované MySQL 5.1.24. Databáze obsahovala 120 milionů 1 KB záznamů, takže celkem 120 GB dat rozložených na 6 serverech, data se tedy nemohla vejít do operační paměti (8 GB). Výsledky jsou detailně popsány v [14], zde si uvedeme příklad testování zátěže převážně čtení (čtení 95% a úprava 5%). MySQL a PNUTS měly lepší doby odezvy při čtení, ale byly pomalejší pro zápis. Výsledné grafy jsou na obrázku 3.2.

Testování podle předpokladu ukázalo, že se databáze liší ve výkonu v různých vlastnostech a záleží na způsobu využití databází pro danou aplikaci.



(a) Čtení



(b) Úprava

Obrázek 3.2: Zátěž převážně čtení: (a) čtení, (b) úprava – závislost doby odezvy na průchodnosti, zdroj [14]

3.2 Testování v MongoDB

Další testování bylo provedeno společností United Software Associates [2] a testovaly se databáze MongoDB 3.0.1, Couchbase 3.0.2 a Cassandra 2.1.2 opět pomocí *YCSB*. Testování proběhlo s jedním serverem a jedním klientem, tedy v podstatně menším měřítku než v předchozím příkladu, a zkoumal se výkon při různých úrovních konzistence dat:

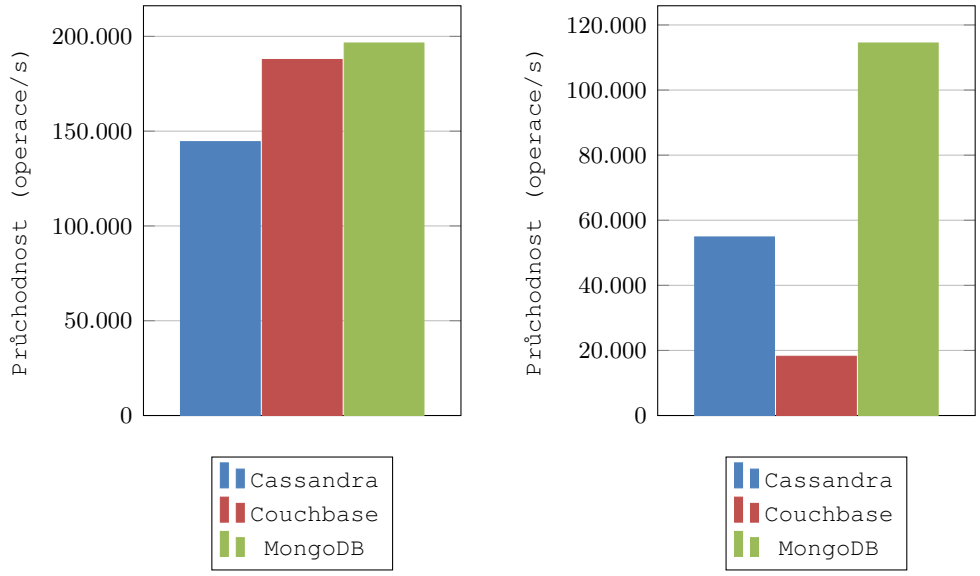
- **Optimalizace průchodnosti (*Throughput Optimized*)** – klient je informován o úspěchu, jakmile se záznam zapíše do RAM.
- **Optimalizace trvanlivosti dat (*Durability Optimized*)** – úspěch nastane až po zapsání na disk.
- **Kompromis (*Balanced*)** – úspěch při zápisu do RAM, ale data se na disk ukládají častěji (není možné u Couchbase).

	Čtení	Zápis
Cassandra	15ms	66ms
Couchbase	<1ms	236ms
MongoDB	1ms	10ms

Tabulka 3.2: Porovnání doby odezvy – optimalizace trvanlivosti (99. percentil), zdroj [2]

Pro zátěž převážně čtení (čtení 95% a úprava 5%) jsou výsledky celkem jednoznačně nejlepší pro MongoDB, které mělo pro optimalizovanou

průchodnost o 35% větší průchodnost než Cassandra (viz obr. 3.3a). Doby odezvy si byly velmi blízké, 99. percentil se pohyboval v jednotkách milisekund.



(a) Optimalizovaná průchodnost (b) Optimalizovaná trvanlivost

Obrázek 3.3: Zátěž převážně čtení, zdroj [2]

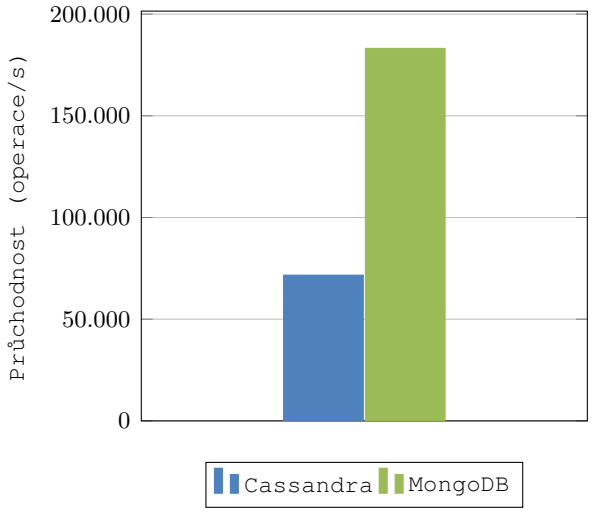
Pro optimalizovanou trvanlivost byly výsledky ještě markantnější, MongoDB bylo asi 2x rychlejší než Cassandra. Doby odezvy už se více lišily, 99. percentil je vidět v tabulce 3.2.

Poslední měření bylo pro kompromis, což je základní nastavení pro MongoDB i Cassandru. Obě databáze poskytovaly dobu odezvy v jednotkách milisekund, avšak průchodnost se celkem dramaticky lišila (viz obr. 3.4).

MongoDB bylo vždy nejlepší, což nabízí otázku, jestli měření nebyla nějak poupravena nebo nebyly pro MongoDB lépe nastaveny nějaké optimalizace (jelikož testy byly provedeny partnerskou společností MongoDB). V praktické části uvedeme výsledky vlastního měření a uvidíme, jestli se budou lišit.

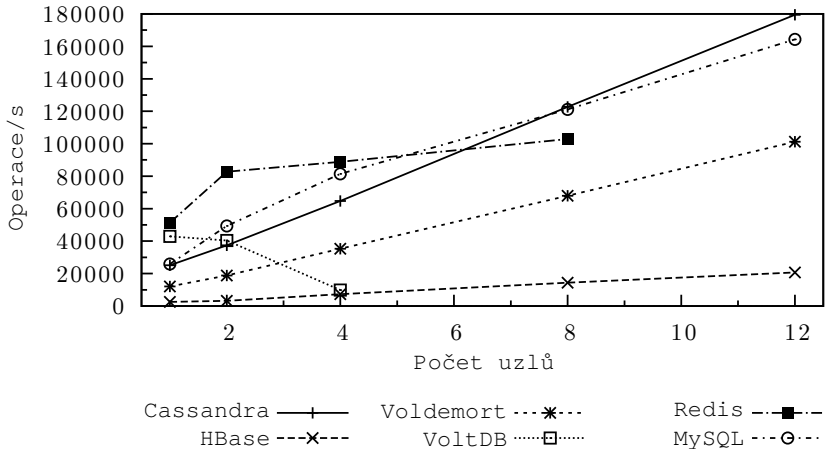
3.3 Testování součástí CA Technologies

Výzkumný tým z univerzity v Torontu spolu s kolegy ze CA Labs provedl testování šesti databází s cílem nalézt nejvhodnější databázi k použití pro *Application Performance Management* – systém umožňující monitorování vý-



Obrázek 3.4: Zátěž převážně čtení, kompromis, zdroj [2]

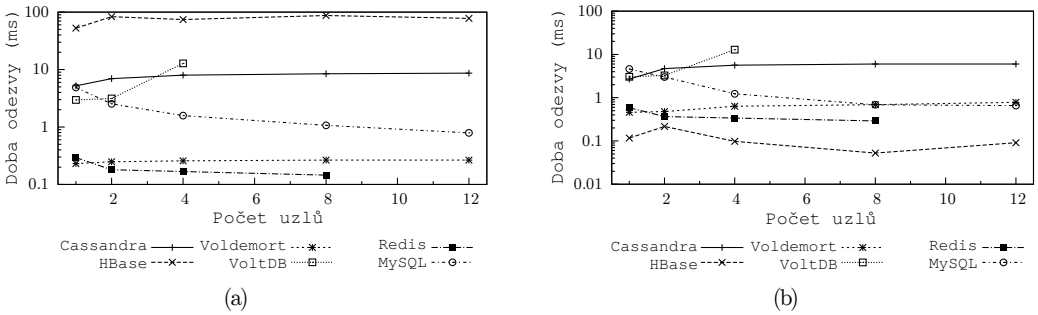
konu běžících aplikací v určitém prostředí a testovali databáze HBase 0.90.4, Cassandra 1.0.0-rc2, Voldemort 0.90.1, Redis 2.4.2, VoltDB 2.1.3 a MySQL 5.5.17.



Obrázek 3.5: Zátěž převážně čtení: průchodnost (operace/s) podle počtu uzlů, zdroj [22]

Testování proběhlo s několika různými zátěžemi na strojích se dvěma Intel Xeon quad core procesory a 16 GB RAM a pro měření dat, která se vešla do RAM, bylo na jednom uzlu asi 700 MB dat (10 milionů záznamů). Podobně jako výše si uvedeme výsledky zátěže převážně čtení. Průměrná doba odezvy a maximální průchodnost se měřila na různém počtu uzlů, výsledky jsou vidět na obrázcích 3.5 a 3.6.

Pro některé databáze vyšší počet uzlů již nebyl výhodný, VoltDB vykazovala příliš velkou dobu odezvy a pro Redis v době testování neexistovala cluster verze, proto byla použita knihovna *Jedis*, pomocí níž se vytvořila distribuovaná databáze – v grafech je možné pozorovat zlom u počtu dvou uzlů.



Obrázek 3.6: Zátěž převážně čtení: doba odezvy (v logaritmickém měřítku) (a) čtení, (b) zápisu podle počtu uzlů, zdroj [22]

Pěkné je pozorovat lineární růst průchodnosti u Cassandry, Voldemorta a HBase, které byly ke škálování přímo navrženy. Celkově Cassandra škálovala nejlépe, i když doba odezvy u ní byla dost vysoká. HBase prokázala nejnižší doby odezvy u zápisu, pro čtení byla ale pomalá, Voldemort se zařadil někde mezi Cassandru a HBase. MySQL měla průchodnost srovnatelnou s Cassandrou a doba odezvy se s větším počtem uzlů snižovala [22].

Měření byla prováděna na datech, která se vešla do RAM i na datech uložených na disku, detaily jsou v dokumentu *Solving Big Data Challenges for Enterprise Application Performance Management* [22].

3.4 Vyhodnocení stávajících testů

Výše popsané testování přineslo různé výsledky. Bylo prováděno na různých verzích jednotlivých databází a jelikož jde vývoj rychle kupředu, testování staré několik let už nemá téměř žádnou vypovídající hodnotu.

V prvním testování byla např. použita Cassandra ve verzi 0.5.0, ve třetím 1.0.0-rc2, přičemž stávající verze je 3.10, takže se dá předpokládat, že došlo k velkým změnám a optimalizaci výkonu. Podobně MySQL testovaná ve verzi 5.1 a 5.5 je již nyní ve verzi 5.7. Druhé testování je nejnovější, ale stále jsou verze databází starší – MongoDB 3.0.1 (nyní 3.4) a Cassandra 2.1.2.

Především se však nepodařilo najít výsledky nějakého testování, které by bylo provedeno na námi požadovaných databázích – Elasticsearch, MongoDB, Apache Cassandra a MySQL Document Store. Z tohoto důvodu provedeme vlastní testy těchto databází v jejich aktuálních verzích.

4 Testování pomocí YCSB benchmarku

Pro výše popsané důvody bylo rozhodnuto provést vlastní testování databází Elasticsearch, MongoDB, Apache Cassandra a MySQL Document Store (DS) pomocí YCSB benchmarku. Tento benchmark ovšem rozšíříme, abychom kromě operací vložení, čtení a úpravy mohli testovat také vyhledávání.

Schéma či tabulka, které se v YCSB používá, má následující strukturu (kód pro vytvoření tabulky v Cassandře):

```
CREATE TABLE usertable (
    y_id varchar PRIMARY KEY,
    field0 varchar,
    field1 varchar,
    field2 varchar,
    field3 varchar,
    field4 varchar,
    field5 varchar,
    field6 varchar,
    field7 varchar,
    field8 varchar,
    field9 varchar);
```

Obsahuje tedy primární klíč a deset textových polí. My toto schéma upravíme nahrazením jednoho z textových sloupců polem číselných hodnot, např. $[61, 510, 852]$, což si můžeme představit jako ID nějakých kategorií, do kterých záznam patří. V Cassandře bude sloupec definován jako

```
categories set<int>.
```

Budeme pak chtít nalézt záznamy, které patří do dané kategorie, tedy např. opět v Cassandře:

```
SELECT * FROM usertable
WHERE categories CONTAINS 123 LIMIT 5;
```

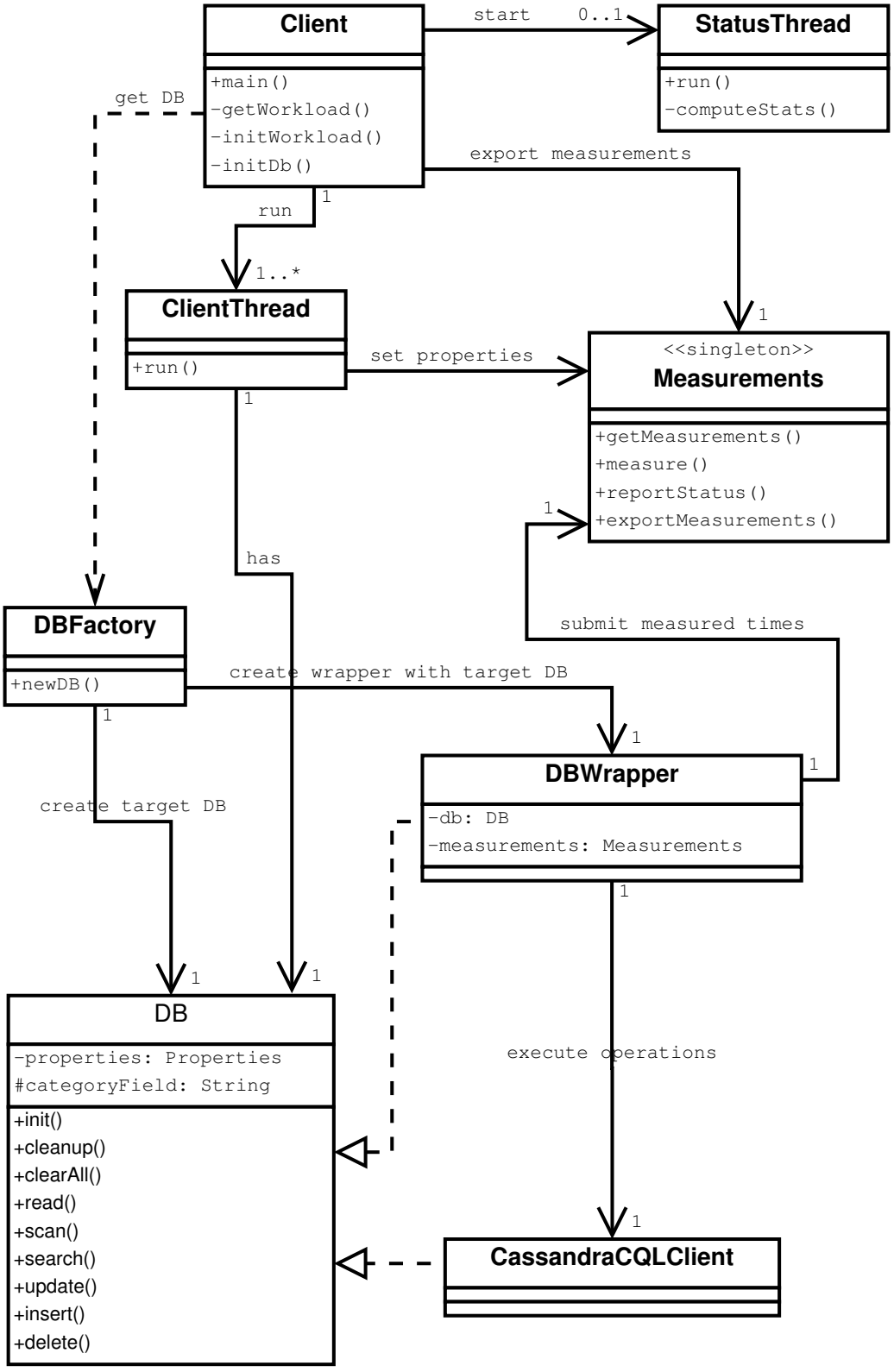


Diagram 4.1: Zjednodušený diagram tříd upraveného YCSB benchmarku

Diagram 4.1 zobrazuje zjednodušený diagram tříd upraveného YCSB benchmarku. Pro přehlednost jsou zde zachyceny jen některé důležité metody a minimum atributů. Hlavní třída s metodou *main* je třída *Client*. Ta nejprve načte parametry z příkazové řádky a souborů, získá instanci třídy dědící od *Workload*, která se stará o provádění transakcí definovaných v konfiguračním souboru zatížení a vytvoří požadovaný počet vláken *ClientThread*. Těm je v konstruktoru přiřazena instance třídy *DBWrapper* získaná pomocí *DBFactory*, která má v sobě atribut *db*, do kterého je uložena instance jedné ze tříd dědících od *DB*.

DBWrapper se stará o posílání naměřených časů jednotlivých operací třídě *Measurements* a může tedy být použita pro všechny klienty databází. Jednotlivé databázově specifické implementace operací jsou pak definovány v samotných klientech.

Vlákna *ClientThread* volají metody třídy dědící od *Workload*, podle toho jestli se jedná o fázi načítání dat (*load*, pak volají *doInsert*) nebo samotného testování (*run*, pak volají *doTransaction*). Třída *Workload* se stará o to, aby vygenerovala patřičné množství požadavků s požadovaným rozdělením operací podle konfiguračního souboru.

4.1 Vlastní úpravy YCSB

Pro usnadnění práce s testováním a kvůli potřebě otestovat vyhledávání v poli byl benchmark upraven následujícími způsoby.

4.1.1 Mazání dat

Při fázi *load* (načítání) se generují data, jejichž ID jsou stále stejná. To znamená, že pokud vygenerujeme tisíc záznamů a tuto fázi zopakujeme, nové záznamy budou mít stejná ID jako předchozí záznamy. Jelikož by takto docházelo ke konfliktům v databázi, před samotným načtením chceme nejprve data smazat. A poněvadž by se jednalo o další operaci navíc z hlediska uživatelského vstupu, bylo rozhodnuto toto zapracovat do fáze načítání.

Do třídy *DB* byla přidána metoda *deleteAllData*, kterou musí třídy od ní dědící implementovat. Pokud tak neučiní, data se samozřejmě nesmažou. Dále byla upravena metoda *initDb* třídy *Client*, která při tvorbě vláken zavolá metodu *deleteAllData* – jen pokud se jedná o fázi načítání a první vlákno.

4.1.2 Hromadné vkládání

Pokud chceme provádět měření na velkých datech, musíme počítat s dlouhou dobou běhu načítací fáze. Ta používá pro vkládání záznamů metodu *insert*, což znamená, že se záznamy vkládají po jednom a to výrazně zpomaluje běh programu.

Klient databáze *MongoDB* přítomný v YCSB již implementoval možnost tzv. *bulk insert*, tedy naskupení několika záznamů dohromady a jejich odeslání do databáze najednou. Po tomto vzoru bylo hromadné vložení implementováno i v ostatních klientech, což mnohonásobně zvýšilo propustnost. Nebyla ovšem implementována logika dodatečného importu, pokud se nenaplní stanovený počet záznamů pro hromadné vložení, která by byla zbytečně komplikovaná. Je tedy na uživateli, aby počet záznamů, které chce vložit, byl beze zbytku dělitelný počtem záznamů pro jedno hromadné vložení. Tzn. pokud je počet záznamů pro hromadné vložení (parametr *batchsize*) např. 1000, musí být počet záznamů, které chceme vygenerovat (parametr *recordcount* zadaný v souboru definice zátěže), tímto číslem dělitelný, např. 1.000.000. Pokud by byl počet např. 10.500, posledních 500 záznamů by nebylo vloženo.

4.1.3 Operace hledání

Jak už bylo naznačeno dříve, chtěli jsme benchmark upravit, aby testoval i méně triviální operaci a to hledání. Konkrétně vyhledání záznamů, které mají v poli čísel uloženém v jednom ze sloupců číslo hledané, což simuluje např. pole ID kategorií, do kterých záznam patří a potřebu získat záznamy patřící do hledané kategorie.

Do třídy *DB* byla tedy přidána metoda *search*, která rozšiřuje již stávající operace obdobným způsobem a byla vytvořena třída *SearchWorkload* dědící od *CoreWorkload*, která umožňuje tuto operaci volat. Dalším krokem bylo ovšem implementovat tuto metodu pro námi testované databáze, nyní se podíváme na jednotlivá řešení:

- **Apache Cassandra**

  Cassandra používá jazyk CQL, takže sestavíme dotaz podobný SQL dotazu:

  ```
  SELECT * FROM usertable
  WHERE categories CONTAINS 123 LIMIT 5;
  ```

Zbytek metody už probíhá obdobně jako u ostatních metod, s metodou scan je dokonce stejný, proto byl kód oddělen do samostatné metody *retrieveScanOrSearchResults*.

- **Elasticsearch**

  Pokud bychom chtěli dotaz sestavit pro *RESTful API*, vypadal by nějak takto:

```
GET ycsb/usertable/_search
{
  "query": {
    "match": {
      "categories": 123
    }
  },
  "size": 5
}
```

  Elasticsearch Java klient poskytuje pro sestavení dotazu třídu *QueryBuilders*, takže vyhledávací dotaz vytvoříme snadno:

```
client.prepareSearch("ycsb")
 .setTypes("usertable")
 .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
 .setQuery(
   QueryBuilders.termQuery("categories", 123)
 )
 .setSize(5)
 .get();
```

- **MongoDB**

  Použijeme-li konzoli *mongo*, která je standardně součástí distribuce MongoDB serveru, můžeme sestavit tento jednoduchý dotaz:

```
db.usertable.find( { "categories": 123 } ).limit(5)
```

  Při použití Java Driveru pak pomocí pomocných tříd pro správu dokumentů a datových typů obdobně uděláme:

```
collection.find(
  new Document("categories", new BsonInt32(123))
).limit(5);
```

- **MySQL Document Store**

  MySQL Java Connector (verze 6.0.6) zatím přímo neumožňuje vyhledávat podle hodnoty v poli pomocí jednoduchých příkazů jako u předchozích databází. Je možné v poli vyhledávat jen na konkrétní pozici, tzn. že můžeme např. vyhledat záznamy, které mají v poli kategorií na prvním místě určité číslo, ale to nám asi k ničemu nebude.

  Konektor umožňuje použití metody *sql* (podrobnosti v kapitole 4.1.4), do které můžeme napsat vlastní SQL kód, kterým už dokážeme cíleného chování docílit.

  ```
  SELECT * FROM usertable
  WHERE JSON_CONTAINS(doc, '123', '$.categories') = 1
  LIMIT 5
  ```

  Využijeme při tom funkci `JSON_CONTAINS`, která v zadaném sloupci (*doc*) hledá hodnotu (123) na cestě `$.categories`. Hodnotu hledáme ve sloupci *doc*, protože je tak tvořena kolekce (viz kapitolu 2.1.3 – MySQL a dokumentová databáze) a cesta zúžuje hledání na určitou část dokumentů. Znak `$` značí kořen a tečky označují zanoření [6].

4.1.4 Implementace klienta MySQL Document Store

Pro implementaci klienta byla použita knihovna *mysql-connector-java* ve verzi 6.0.6, která již implementuje rozhraní *X DevAPI*. Připojení k databázi se vytvoří jednoduše:

```
new XSessionFactory()
    .getSession(connectionUrl)
    .getSchema(schemaName);
```

Parametr metody *getSession* je URL specifikující připojení k databázi v běžném formátu:

```
mysqlx://host:port/schemaName
```

Port je základně nastaven na 33060 – všimněme si podobnosti s portem, na kterém standardně poslouchá MySQL. Parametr *schemaName* je název databáze, ke které se chceme připojit. K URL ještě můžeme připojit další parametry, např. specifikovat uživatele a heslo.

Volání metody *getSession* vrací objekt typu *XSession*, který umožňuje pracovat s kolekcemi i s tabulkami pomocí CRUD operací. Pod *XSession* může být schováno připojení i k několika MySQL serverům. Pokud však chceme použít dotaz psaný v SQL, potřebujeme tzv. *NodeSession*, která je fyzicky připojena jen k jednomu serveru. Pak můžeme volat:

```
new XSessionFactory()
    .getNodeSession(connectionUrl)
    .sql(query).execute();
```

Toto bylo použito při operaci hledání (viz kapitolu 4.1.3).

X DevAPI poskytuje základní CRUD operace, které byly využity následně:

```
com.mysql.cj.api.xdevapi.Collection collection =
    schema.getCollection(tableName);

collection.add(dbDoc).execute();
collection.find("_id = :id").bind("id", key).execute();
collection.modify("_id = :id").set("column", "value")
    .bind("id", key).execute();
collection.remove("_id = :id")
    .bind("id", key).execute();
```

Třída *Collection* představuje kolekci (tabulku), nad kterou pak můžeme provádět operace. Metody *find*, *modify* i *remove* mají jako parametr podmínku, podle které se záznamy vyhledají. U metody *modify* pak s nalezenými záznamy musíme provést úpravy, kromě metody *set* jsou k dispozici další metody jako např. *arrayInsert*, pro detailní dokumentaci viz [7].

4.2 Testování

Testování bylo provedeno na dvou počítačích, každý s touto konfigurací: procesor Intel Core i7-4770 3.40 GHz, 16 GB RAM, operační systém Windows

10 Pro, 64-bit. Jeden fungoval jako databázový server a druhý figuroval jako klient – generoval dotazy a počítal statistické údaje.

Databáze byly testovány ve verzích Elasticsearch 5.3.2, MongoDB 3.4.4, Apache Cassandra 3.10 a MySQL 5.7.18 (použita jako Document Store).

Měření bylo provedeno s daty, které se vešly do RAM – 1 milión záznamů o velikosti téměř 1 KB, tedy zhruba 1 GB dat. Data sestávala z těchto částí:

1. ID záznamu,
2. 9 sloupců náhodně vygenerovaných řetězců o velikosti přes 100 bytů,
3. pole jednoho až pěti náhodně vygenerovaných celých čísel.

První dvě již *YCSB* obsahoval, třetí část byla doimplementována (viz kapitolu 4.1.3).

Jak již bylo dříve rozebráno, existuje mnoho možností poměrů jednotlivých operací, které jsou testovány. Pro naše testování byly vybrány tyto tři typy zatížení (ID ke čtení a úpravě se vybírala s pravděp. rozdělením *Zipfian*, hledaná kategorie byla vygenerována s rovnoměrným rozdělením):

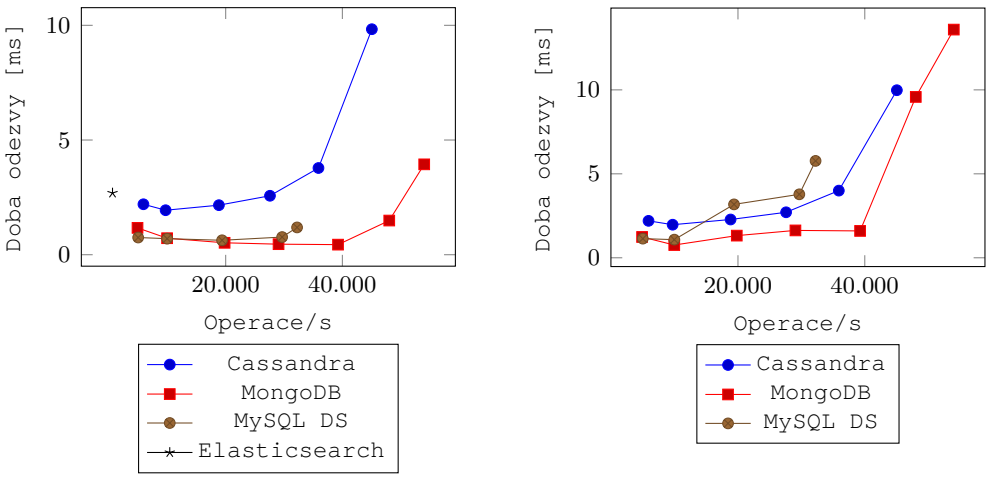
1. Převážně čtení – 95% čtení a 5% úprava.
2. Čtení/úprava – 50% čtení a 50% úprava.
3. Čtení/hledání – 50% a 50% hledání.

Měřila se doba odezvy s postupně zvyšující se propustností, které jsme chtěli dosáhnout. Ta se zadává spolu s počtem vláken jako parametr při spouštění testu. Pokud se nedosáhlo požadované propustnosti, počet vláken byl navýšen, ale začínalo se s menším počtem, aby nedocházelo ke zbytečně velké režii na straně klienta, která by měla opět za následek nižší propustnost. Nejméně však bylo spuštěno osm vláken.

4.2.1 Zátěž převážně čtení

Na obr. 4.1 vidíme graf pro zátěž převážně čtení s průměrnými dobami odezvy. Elasticsearch je však zanesen jen v prvním grafu, jelikož si v měření vedl dost špatně, průměrná doba odezvy pro operaci úprava byla 177 ms při průchodnosti jen 600 operací/s.

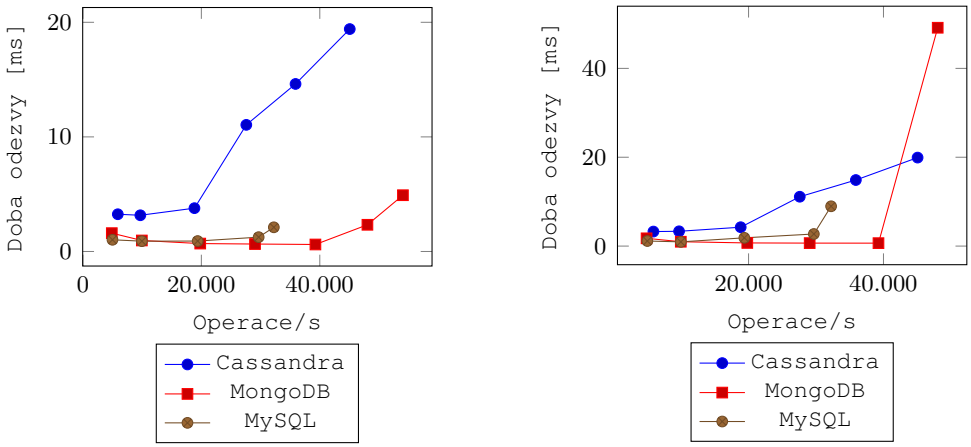
Když porovnáme zbývající tři databáze, tak si MongoDB a MySQL DS vede při čtení o něco lépe než Cassandra. Při úpravě však Cassandra předběhne MySQL DS a potvrdí tak svůj primární účel – rychlý zápis. Grafy mají víceméně exponenciální charakter.



(a) Operace čtení (b) Operace úprava

Obrázek 4.1: Zátěž převážně čtení, průměrná doba odezvy

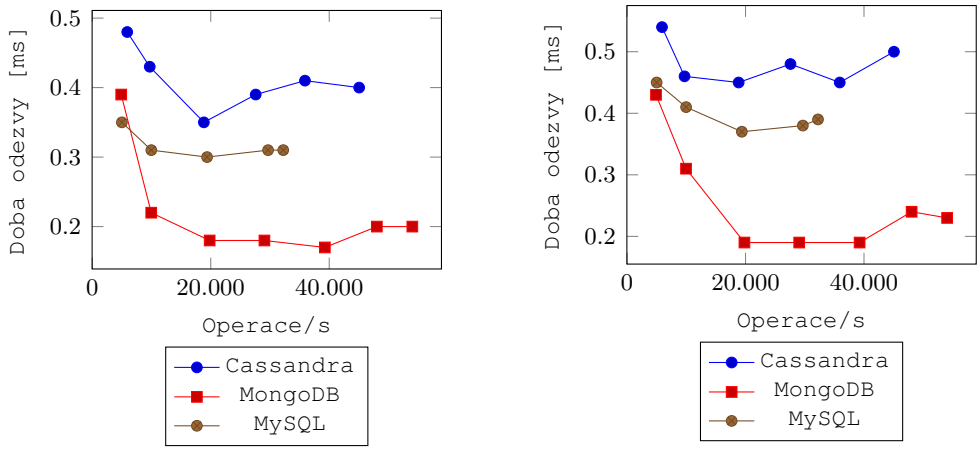
Na obrázku 4.2 je zobrazen 95. percentil doby odezvy, který má pro MongoDB téměř stejný průběh jako průměr, avšak Cassandra je na tom od průchodnosti 20.000 operací za sekundu v porovnání se svým průběhem průměru hůře. MySQL DS má zase pro úpravu průběh lepší.



(a) Operace čtení (b) Operace úprava

Obrázek 4.2: Zátěž převážně čtení, 95. percentil doby odezvy

Zajímavý je graf minimální doby odezvy (obr. 4.3), která se s větší propustností spíše zlepšovala, i když rozdíly jsou jen velmi malé – v desetinách milisekund.



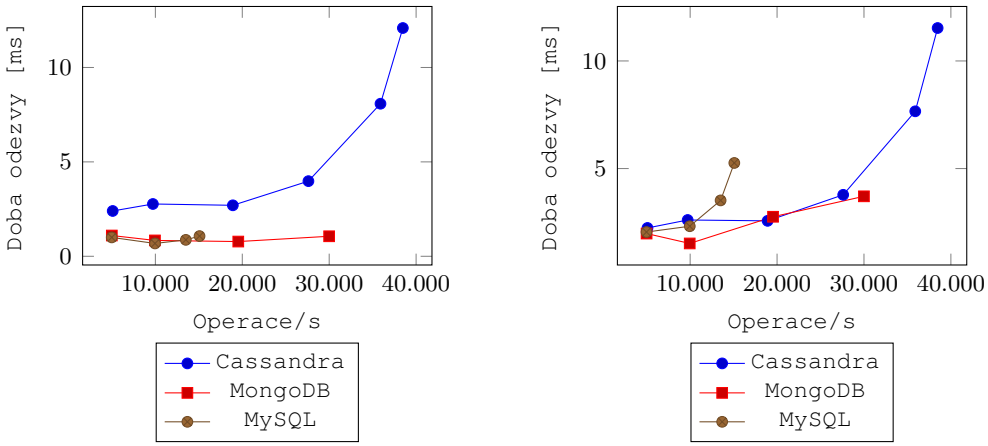
(a) Operace čtení

(b) Operace úprava

Obrázek 4.3: Zátěž převážně čtení, min. doba odezvy

4.2.2 Zátěž čtení/úprava

Další testovanou zátěží bylo čtení a úprava ve stejném poměru. Graf průměrné doby odezvy je vidět na obr. 4.4. Průběh je velmi podobný průměru doby odezvy u zátěže převážně čtení (obr. 4.1), jen se doba odezvy začíná dříve zvyšovat. U MySQL DS však už pozorujeme výraznější zhoršení.



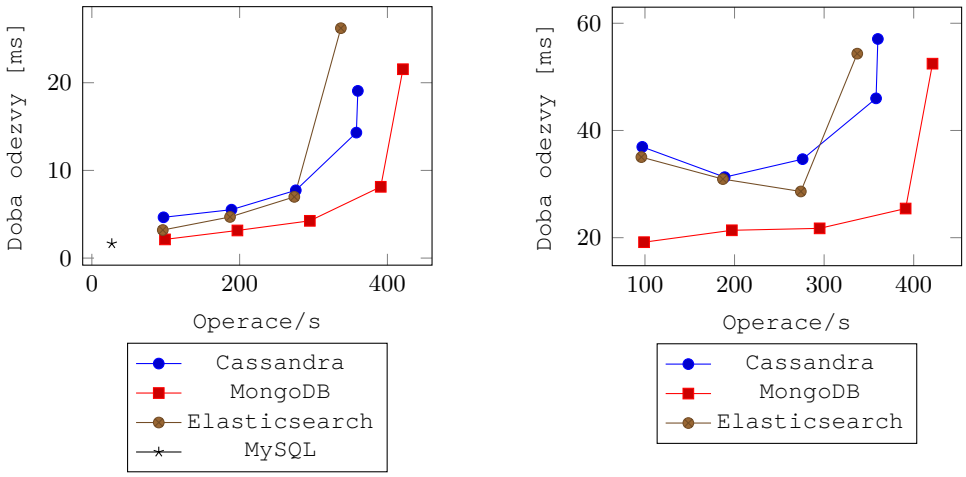
(a) Operace čtení

(b) Operace úprava

Obrázek 4.4: Zátěž čtení/úprava, průměrná doba odezvy

4.2.3 Zátěž čtení/hledání

Obr. 4.5 zobrazuje zátěž čtení a hledání, opět ve stejném poměru. Z grafů je patrné, že operace hledání je mnohem náročnější než operace čtení nebo úprava. Při tomto rozložení už si Elasticsearch vedl podobně jako Cassandra a MongoDB, průměrné doby odezvy operace čtení dosahovaly 20 ms pro průchodnost již kolem 350 operací/s. MySQL DS tentokrát dopadla velmi špatně, podařilo se dosáhnout průchodnosti pouze 27 operací/s.



(a) Operace čtení

(b) Operace hledání

Obrázek 4.5: Zátěž čtení/hledání, průměrná doba odezvy

Operace hledání byla ještě nákladnější, nejmenší průměrná doba odezvy byla dosažena u MongoDB při průchodnosti 100 operací/s a to 19 ms. Doba kolem 30-40 ms však zejména u Elasticsearch a Cassandry nebyla žádnou výjimkou.

Pro dosažení lepších výsledků byl vytvořen v Cassandře a MongoDB index nad sloupcem, ve kterém se vyhledávalo. Elasticsearch toto nepotřebuje, jelikož všechna data ukládá do invertovaného indexu. MySQL DS ale v tomto byla znevýhodněna, protože pro indexování hodnot ve sloupci typu JSON musíme vytvořit sekundární index nad virtuálním sloupcem, který z dokumentu extrahuje danou hodnotu, což je možné pouze pro hodnoty skalární – v našem případě se však jednalo o pole, proto také v druhém grafu MySQL není zanesena, jelikož pro průchodnost 27 operací/s dosáhla průměrné doby odezvy přes 1 s!

4.3 Shrnutí výsledků

Po tomto prvním testování můžeme konstatovat, že z něj MongoDB vyšla nejlépe ve všech ohledech.

Za ní následuje MySQL DS a Cassandra, které dosahovaly vždy jen o trochu horších výsledků, MySQL DS byla někdy na úrovni MongoDB, jindy zase na tom byla hůře než Cassandra.

Elasticsearch dokázal, že je vytvořený pro hledání, kde se s ostatními mohl utkat. Co se však týče pouhého čtení a úprav, nemohl ostatním konkurovat. Na vině je ale asi také fakt, že data byla tvořena náhodně vygenerovanými řetězci o délce přes sto znaků, např.:

```
&Gk\"?4#)&!2f9M%8@?/4\",Bg\"E/!D'(Te&9v8E9!Vy=(f2>x/[k>Su<X78A
#=\" !?**X;=D;/<n?0\" Q#%.:6-*$Fo7W/5P%-5,4
```

Jak je vidět, mezery se v řetězcích téměř nevyskytují (oproti standardnímu textu) a jelikož Elasticsearch dělá předzpracování a data ukládá do indexu, většina řetězců byla velmi dlouhých a unikátních, což mělo dopad na rychlost čtení a především úprav, ve kterých si Elasticsearch vedl dost špatně, i když mu byla přidělena polovina operační paměti (8 GB), do které se celý index vešel. Bylo by tedy lepší jej testovat s reálným textem, ale to by vyžadovalo další rozsáhlou úpravu YCSB benchmarku.

Nyní se podíváme na vytvoření celého nového systému, s jehož pomocí pak databáze otestujeme ještě z jiného pohledu.

5 Implementace společného rozhraní

Hlavním bodem zadání byla implementace systému (s požadavkem, aby byl napsán v programovacím jazyce PHP), který umožní dělat dotazy nad velkým množstvím dat a následné porovnání výkonu jednotlivých databází. Aby bylo možné v budoucnu tento systém použít univerzálněji, bylo také požadováno vytvoření jednotného rozhraní k databázím, aby se mohla použít jakákoliv z implementovaných databází, která by v případě potřeby byla jednoduše nahrazena jinou.

Všechny třídy implementací klientů tedy dědí od společné abstraktní třídy *DB*, která má základní CRUD metody, metody pro hromadné vkládání a mazání dat a hledání podle specifikovaných parametrů. Implementuje také metodu *checkParams*, která provádí kontrolu a transformaci parametrů vstupujících do metody hledání *findBy*. Schéma databázových tříd je vidět na diagramu 5.1.

Metoda *findBy* umožňuje hledat podle množství různých parametrů. Těmi můžou být:

- *fields* – specifikace sloupců, které chceme vrátit,
- *where* – podmínka,
- *aggregation* – agregace, funkce jako `SUM`, `AVG`, `MIN`, `MAX`,
- *groupBy* – název sloupce pro seskupení,
- *orderBy* – pole názvů sloupců, podle kterých se mají záznamy seřadit, může být s `desc`, tj. sestupně,
- *limit* – maximální počet záznamů, které chceme vrátit,
- *offset* – kolik záznamů se má přeskočit,
- *count* – chceme vrátit pouze počet záznamů.

V parametru *where* může samozřejmě být i booleovský výraz, jehož podoba se např. pro MySQL nemusí vůbec měnit. Pro Elasticsearch a MongoDB byla navíc vytvořena abstraktní třída *DBWithBooleanParsing*, která dědí od třídy *DB*, ale obohacuje ji o metody pro parsování booleovského dotazu (některé jsou abstraktní, některé společné implemetuje, viz diagram 5.1).

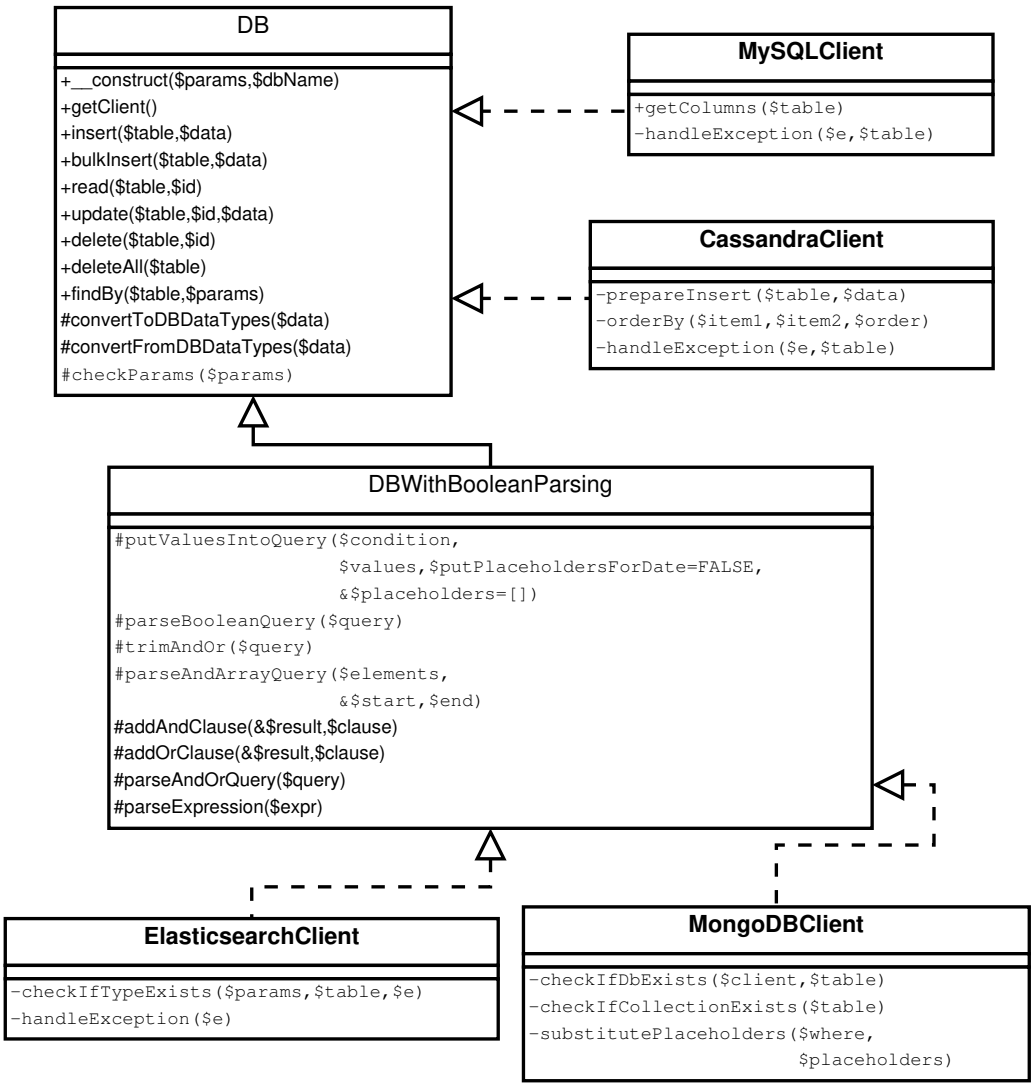
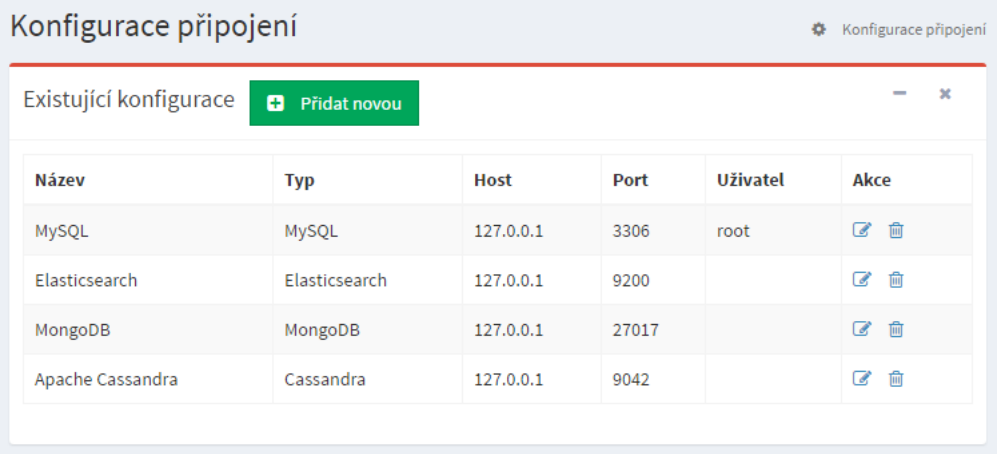


Diagram 5.1: Diagram databázových tříd. Metody napsané bezpatkovým písmem jsou abstraktní, monospace písmem nikoliv.

5.1 Konfigurace připojení

Pro obsluhu databází byl vytvořen CRUD pro editaci konfigurací k jednotlivým databázím. Konfigurace se ukládají do MySQL tabulky a obsahují název připojení, databázi, ke které se připojení váže, adresu, port a případně uživatele a heslo (viz obr. 5.1).



Obrázek 5.1: Konfigurace připojení k databázím

5.2 Specifika jednotlivých databází

Každý klient má konstruktor, který přijímá parametry pro připojení k databázi (z konfigurace připojení) a název databáze (či ekvivalentu, např. indexu u Elasticsearch). Nyní se podíváme na specifické body implementace jednotlivých klientů.

5.2.1 MySQL

MySQL zatím neposkytuje konektor pro dokumentovou databázi v PHP takže byla tentokrát použita klasická MySQL s ukládáním dat do sloupců. Nepočítá se tedy s ukládáním zanořených struktur, se kterými by byl problém.

Jelikož je projekt psaný v *Nette* [18], připojení k MySQL je vytvořeno pomocí *Nette\Database\Connection* (které interně používá *PDO*) a *Context*. Ten poskytuje metody jako *query* a *queryArgs* pro vytváření dotazů a *table*,

která vrací *Table\Selection*, nad kterou se dají volat operace jako *get*, *insert* a *delete*, ale i *select*, *where*, *order* apod. – těch bylo využito v metodě *findBy*.

Při dotazech jsou odchyceny výjimky a předány metodě *handleException*, která zjišťuje známé chyby jako absenci tabulky nebo databáze pro pěkný výpis chybové hlášky. Pokud je výjimka neznámá, je její zpráva vypsána uživateli v originálním znění.

5.2.2 Cassandra

Pro připojení z PHP i jiných programovacích jazyků ke Cassandře je možné použít PHP Driver od společnosti DataStax [21] – např. stáhnout si binární *dll* soubor a použít jej jako rozšíření (*extension*) v `php.ini`. Spojení jednoduše vytvoříme následovně:

```
$cluster = Cassandra::cluster()
    ->withContactPoints($params["host"])
    ->withPort($params["port"])
    ->build();
$client = $cluster->connect($dbName);
```

To nám pak umožní volat metody *prepare*, *execute* nebo *executeAsync*, do kterých můžeme psát dotazy v jazyce CQL. Jelikož je velmi podobný jazyku SQL, je možné použít booleovské podmínky nezměněné, podobně jako v MySQL.

Problém však nastává u použití `OR`, Cassandra jej, alespoň v době psaní této práce, nepodporuje, takže pokud je obsaženo v podmínce, vyhodí klient Cassandry výjimku.

Při vkládání je třeba ošetřit některé datové typy:

- *DateTime* se musí převest na *Cassandra\Timestamp*,
- desetinné číslo se musí převest na *Cassandra\Float*.

Při získávání výsledků se provede opačný proces. K těmto konverzím slouží metody *convertToDBDataTypes* a *convertFromDBDataTypes*.

Sestavování dotazu pro metodu *findBy* probíhá obdobně, jako kdyby to byl SQL dotaz. Výjimku tvoří přidání `ALLOW FILTERING` na konec dotazu, čímž databázi dáváme souhlas s provedením dotazu, který hledá ve sloupcích, jež nejsou *cluster columns*, s vědomím, že daná operace může trvat delší dobu [15].

Dalším problémem je použití `ORDER BY`, které je podporováno jen nad sloupcem, jenž je primárním klíčem. Nepředává se tedy do dotazu, ale výsledky jsou seřazeny až v PHP podle pořadí, které bylo definováno v parametrech.

Použití `GROUP BY` je možné pouze nad primárním klíčem, toto je však ponecháno na uživateli a v případě problému vyhozena původní chyba.

Cassandra nepodporuje `OFFSET`, proto pokud je v parametrech zadán *limit* a *offset*, do dotazu je dosazen:

```
$limit = $params["limit"] + $params["offset"];
```

a výsledky jsou následně v PHP o daný *offset* zkráceny.

Pokud chceme zjistit počet všech záznamů v tabulce, Cassandra je musí na rozdíl od ostatních databází natvrdo spočítat, což při velkém počtu záznamů zabere opravdu hodně času. Klient tedy počítá s tím, že tabulka s názvem *name* má k sobě tabulku s názvem *name_count*, ve které je pouze jediný řádek, který uchovává počet záznamů v první tabulce.

Pro hromadné vložení byl použit *BatchStatement*. I když se to z názvu může zdát, není primárně určený pro hromadné vkládání z hlediska rychlosti, ale pro atomické vkládání více záznamů. Proto je základním nastavením `LOGGED`, což znamená, že se nejprve operace zapíší do logu a následně provedou a to zajistí, že pokud nějaké vložení selže, selže také celá dávka. Druhou možností je vkládat záznamy asynchronně, nicméně *BatchStatement* se osvědčil jako rychlejší (při použití `UNLOGGED`), přičemž chyba při testování nenastala [15].

Podobně jako v MySQL klientovi je zde implementována metoda *handleException*.

5.2.3 Elasticsearch

Elasticsearch clienta je možné nainstalovat např. pomocí *Composeru*. Připojení je podobné jako u Cassandry:

```
$connectionParams[] =
    $params["host"] . ":" . $params["port"];
$client = ClientBuilder::create()
    ->setHosts($connectionParams)
    ->build();
```

Rozdíl je však v tom, že se nepřipojujeme přímo k určitému indexu, ten je i s typem zadán v každém dotazu, viz např. vložení:

```
// data, ktera chceme vlozit
$data = [
  "id" => 1,
  "name" => "Jane",
  "surname" => "Doe"
];

// parametry - obsahujici nazev indexu a typu + data
$params = [
  "index" => "indexName",
  "type" => "typeName",
  "body" => $data
];

// pokud je v datech ID, vlozi se do parametru
if (!empty($data["id"]))
{
  $params["id"] = $data["id"];
}

$response = $this->client->index($params);
```

Pokud chceme, aby měl záznam nějaké ID (ne automaticky vygenerované), musí se specifikovat také v rámci paramterů, ale ne v *body*, nýbrž jako samostatná položka vedle *body*. Proto je zde podmínka, při které testujeme existenci `$data["id"]`, a vkládáme jej do parametrů. Poslední příkaz provede samotnou indexaci.

Na rozdíl od ostatních databází vytvoří Elasticsearch při vložení záznamu index i typ automaticky, pokud neexistují.

Jak už bylo řečeno dříve, jelikož se v Elasticsearch a MongoDB provádějí booleovské dotazy jinou syntaxí než v MySQL, bylo třeba vytvořit jednoduchý parser. Pro představu si uvedeme stejný dotaz v MySQL a v Elasticsearch (v JSON notaci):

```
SELECT * FROM person
WHERE name = "Jan" AND (height > 190 OR weight <= 60)
LIMIT 10;
```

```
GET db/person/_search
{ "size": 10,
  "query": {
    "bool": {
      "filter": [
        {
          "match": {
            "name": "Jan"
          }
        },
        {
          "bool": {
            "should": [
              {
                "range": {
                  "height": {
                    "gt": 190
                  }
                }
              },
              {
                "range": {
                  "weight": {
                    "lte": 60
                  }
                }
              }
            ]
          }
        }
      ]
    }
  }
}
```

Booleovské dotazy se tedy dají vytvářet pomocí `bool`, operátor `AND` je nahrazen slovem *filter* a `OR` slovem *should*. Podmínky se do sebe dají zanořovat, v našem příkladu je *should* uvnitř *filter*, stejně jako v MySQL dotazu.

Tento dotaz by v implementovaném klientovi bylo možné zadat následovně:

```
$client->findBy("person", [
  "where" => [
    "name = ?" => "Jan",
    "height > ? OR weight <= ?" => [190, 60]
  ],
  "limit" => 10
]);
```

Jednotlivé položky *where* parametru jsou spojovány logickým `AND`, operátor `OR` musí být explicitně uveden. Pro každou položku se provedou tyto kroky:

1. Za otazníky se dosadí hodnoty. O to se stará metoda *putValuesIntoQuery* ve třídě *DBWithBooleanParsing*.
2. Výsledek se vloží jako parametr metody *parseBooleanQuery* stejné třídy.
3. Zkontroluje se, že počet otevíracích a uzavíracích závorek se shoduje.
4. Pokud jsou v dotazu závorky, postupně se rekurzivně rozděluje a vyhodnocuje (pro podrobnosti viz metodu *parseBooleanQuery*).
5. Pokud v dotazu závorky nejsou, předá se přímo metodě *parseAndOrQuery*.
6. Metoda *parseAndOrQuery* je implementovaná v každém potomkovi třídy *DBWithBooleanParsing*. Nejprve rozloží dotaz podle operátoru `OR` a pak každou ze vzniklých skupin podle `AND`.
7. Jednotlivé výrazy předkládá metodě *parseExpression* (rovněž implementovaná v každém potomkovi třídy *DBWithBooleanParsing*) a výsledky skládá dohromady.
8. V metodě *parseExpression* se podle operátoru ($\geq, >, ...$) vrátí pole v požadovaném formátu (za použití *range*, *match* nebo *wildcard*, viz vzorový příklad).

Výsledky pro jednotlivé položky se pospojují pomocí *filter* a tento tvar použijeme pro položení dotazu Elasticsearch klientovi.

Po získání výsledků je musíme správně zpracovat, uspořádání výsledků se liší, pokud šlo o:

- `GROUP BY`,
- pouze agregace (bez `GROUP BY`),
- ostatní případy.

Podobně jako u Cassandry musíme provést konverzi, zde převádíme pouze *DateTime* na formát ISO 8601 a zpět při získání výsledků.

Při provádění operací nad klientem může být vyhozena výjimka, např. když budeme hledat neexistující dokument podle ID. Z ní můžeme zjistit, že dokument nebyl nalezen, ale není zřejmá příčina, proto zavoláme metodu *checkIfTypeExists*, která ověří, zda vůbec existuje typ, ve kterém dokument hledáme. Pokud existuje, vrátíme pouze `FALSE`, pokud však neexistuje, vyhodíme výjimku o absenci typu.

5.2.4 MongoDB

Pro připojení k MongoDB potřebujeme stejně jako u Cassandry externí ovladač, který je možné stáhnout jako PECL knihovnu [23] a použít jej v `php.ini`. I když by tento ovladač stačil, není snadné jej používat, a proto je lepší si např. *Composer*em nainstalovat *MongoDB PHP Library* [3].

K databázi se připojíme jednoduchým způsobem:

```
$uri = 'mongodb://' .
       $params['host'] . ':' . $params['port'];
$mongoClient = new Client($uri);
$client = $mongoClient->dbName;
```

K práci s daty slouží metody jako *insertOne*, *insertMany*, *findOne*, *updateOne*, *deleteOne* nebo *find*, jejichž použití je celkem intuitivní a poměrně dobře zdokumentované [3].

Stejně jako u Elasticsearch se konverze při vkládání dat dělá pouze pro *DateTime* – převádí se na *MongoDB\BSON\UTCDateTime*. Pokud však chceme datum použít v podmínce *where*, musíme jej dosadit jako objekt do sestavené podmínky. Sestavování však funguje tak, že se nejprve nahradí otazníky skutečnými hodnotami (viz kroky sestavení dotazu u Elasticsearch

– kapitola 5.2.3). Abychom tomuto předešli, umožňuje metoda *putValuesIntoQuery* specifikovat parametrem, že chceme datum nahradit pouze nějakým znakem (#0#, #1#, ...) a do pole, které jí předáme, uloží jednotlivá data k jednotlivým znakům. Po sestavení dotazu pak za jednotlivé znaky v metodě *substitutePlaceholders* dosadíme skutečná data ve formátu *MongoDB\BSON\UTCDateTime* podle převodního pole, které naplnila metoda *putValuesIntoQuery*.

Při čtení z databáze navíc knihovna nevrací výsledky v poli, ale jako objekt typu *MongoDB\Model\BSONDocument* – ten stačí jednoduše přetypovat na pole.

Jak už bylo řečeno, MongoDB klient také dědí od třídy *DBWithBooleanParsing* a postupuje velmi podobně jako Elasticsearch. Lépe je však na první pohled rozumět operátorům – `$and` a `$or`.

Podíváme-li se na dotaz popisovaný výše, budou předané parametry vypadat takto:

```
$filter = [
  "$and" => [
    0 => [
      name => "Jan"
    ],
    1 => [
      "$or" => [
        0 => [
          height => [
            "$gt" => 190
          ]
        ],
        1 => [
          weight => [
            "$lte" => 60
          ]
        ]
      ]
    ]
  ]
]
```

Trošku zvláštností je předání limitu až v druhém poli parametrů metody *find*:

```
$options = [
  "limit" => 10
]
$client->person->find($filter, $options);
```

MongoDB nevyhazuje mnoho výjimek – nad navráceným objektem u jednotlivých operací je třeba kontrolovat úspěšnost voláním různých metod, např. *getMatchedCount*, *getDeletedCount* nebo *isAcknowledged* a případně zkontrolovat, jestli vůbec existuje daná kolekce. Existence databáze se testuje už v konstruktoru.

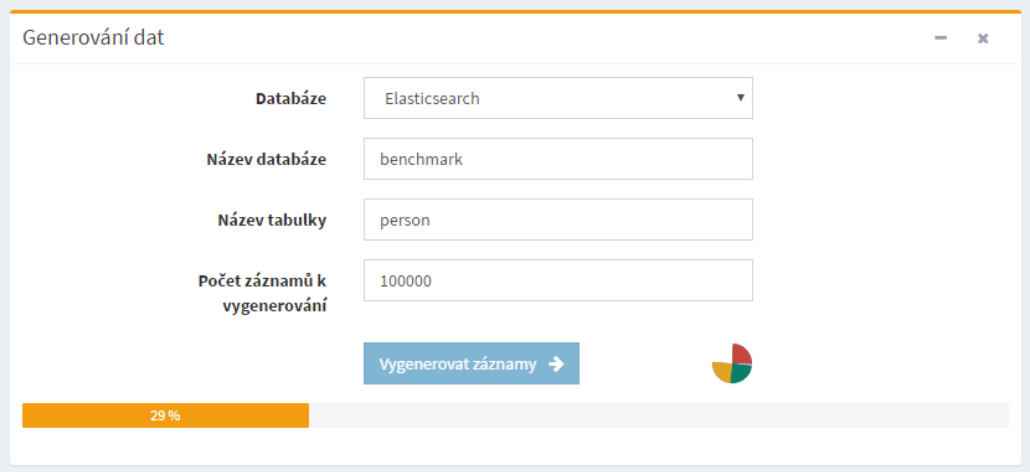
5.3 Uživatelské rozhraní pro generování dat a testování

Pro usnadnění testování bylo vytvořeno uživatelské rozhraní se třemi komponentami. Každá obsahuje formulář, ve kterém musí uživatel zvolit databázi (z nakonfigurovaných připojení, viz kapitolu 5.1) a vyplnit název databáze a tabulky. Těmito komponentami jsou:

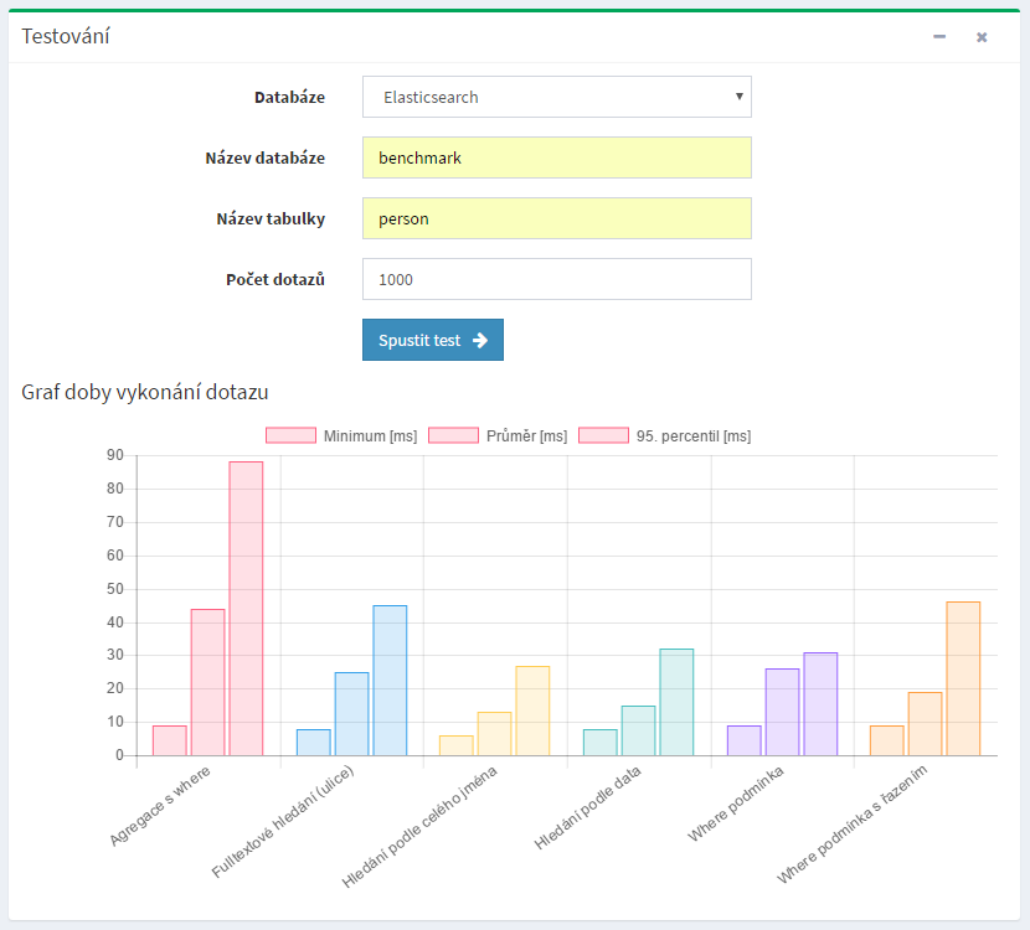
- Smazání dat – obsahuje také možnost zjištění počtu záznamů v tabulce.
- Generování dat – potřebuje ještě zadat počet záznamů, které se mají vygenerovat.
- Testování – navíc s parametrem počet dotazů, které se mají vykonat.

Všechny komponenty posílají *AJAX*ové požadavky. Generování a testování většinou trvá dlouho, takže bylo implementováno jednoduché zobrazení stavu, viz obr. 5.2. Proces na serveru ukládá procento zpracovaných příkazů do *session* a klient se dalšími asynchronními požadavky ptá na stav.

Po skončení testování jsou na serveru spočítány statistické hodnoty, které jsou uloženy do databáze a poslány klientovi, jenž je vykreslí do grafu pro přehlednější zobrazení (obr. 5.3).



Obrázek 5.2: Generování dat



Obrázek 5.3: Zobrazení výsledků testování

5.4 Testování

Druhé testování bylo provedeno na stejných počítačích jako první – každý měl procesor Intel Core i7-4770 3.40 GHz, 16 GB RAM a operační systém Windows 10 Pro, 64-bit. Jeden fungoval jako databázový server a druhý jako klient s Apache serverem ve verzi 2.4.25. Databáze byly testovány opět ve verzích Elasticsearch 5.3.2, MongoDB 3.4.4, Apache Cassandra 3.10 a MySQL 5.7.18.

PHP bylo použito ve verzi 7.0.9, protože ovladač pro Cassandru vyžadoval maximální verzi PHP 7.0.99.

Do databází bylo vygenerováno 100 milionů záznamů, takže se data již nevešla do operační paměti. Testování bylo tentokrát provedeno nad pseudoreálnými daty. Nejednalo se o data reálná, nicméně jejich charakterem se tomu blížila a poskytovala různé typy polí pro testování různých dotazů – text, čísla i datum. Asi nejsnazší a nejsrozumitelnější bude uvézt SQL kód pro vytvoření tabulky (s informacemi o osobách):

```
CREATE TABLE `person` (
    `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    `surname` VARCHAR(255) NOT NULL,
    `email` VARCHAR(255) NOT NULL,
    `street` VARCHAR(255) NOT NULL,
    `house_number` SMALLINT(6) NOT NULL,
    `city` VARCHAR(255) NOT NULL,
    `zip` VARCHAR(6) NOT NULL,
    `height` DOUBLE NOT NULL,
    `weight` DOUBLE NOT NULL,
    `date_of_birth` DATETIME NOT NULL,
    PRIMARY KEY (`id`)
);
```

Data byla generována náhodně následujícím postupem (seznamy jmen, příjmení, ulic i obcí byly staženy ze serveru *http://www.jmenaprijmeni.cz*):

- Křestní jméno (*name*) bylo vybíráno náhodně s exponenciálním rozdělením ze seznamu asi čtyř set nejpoužívanějších jmen.
- Příjmení (*surname*) bylo vybíráno stejným způsobem ze seznamu asi tisíce nejčastějších příjmení. Pro jednoduchost generování se nerozlišo-

valo pohlaví, takže se generovala dámská jména s pánským příjmením a naopak, což však na testování nemá vliv.

- Email (*email*) byl generován ve formátu prijmeni.jmeno@server.pripona, kde server byl náhodně vybrán z tohoto seznamu: gmail.com, seznam.cz, email.cz, volny.cz, centrum.cz. Toto sice vedlo k vytvoření duplicit, ale pole email nakonec nebylo při testování použito a sloužilo pouze k navýšení objemu dat.
- Ulice (*street*) byla vybrána náhodně s rovnoměrným rozdělením ze seznamu asi 1600 náhodně vybraných ulic.
- Číslo domu (*house_number*) bylo vygenerováno náhodně s rovnoměrným rozdělením od 1 do 3000.
- Město (*city*) bylo vybráno náhodně s exponenciálním rozdělením ze seznamu asi 270 obcí seřazených podle počtu obyvatel.
- PSČ (*zip*) bylo vygenerováno jako náhodné číslo od 10000 do 99999.
- Výška (*height*) byla vygenerována náhodně od 140 do 200 cm jako desetinné číslo s přesností na desetiny.
- Váha (*weight*) byla vygenerována náhodně od 40 do 140 kg jako desetinné číslo s přesností na setiny. Realitě bližší by bylo generování váhy i výšky s Gaussovo rozdělením, ale pro jednodušší a rychlejší generování se použilo rozdělení rovnoměrné, což by testování nemělo ovlivnit.
- Datum a čas narození (*date_of_birth*) byl vygenerován náhodně tak, že se generoval zvlášť rok (1970-2016), měsíc (1-12), atd. Generovaly se i vteřiny, které se sice u data narození nezaznamenávají, ale pro testování jsme chtěli, aby toto pole reprezentovalo jakýkoliv datum a čas.

Testovalo se několik různých dotazů, všechny však z databáze pouze četly (zápis byl testován již v prvním testování). Dotazy také byly komplikovanější, konkrétně:

1. Agregace s podmínkou – Datum narození bylo v určitém rozpětí (`date_of_birth >= ? AND date_of_birth <= ?`), max. však šest let a výška měla horní hranici (`height <= ?`), zkoumala se průměrná výška.
2. Podmínka – Váha byla v rozpětí `weight > ? AND weight < ?` a výška měla opět horní hranici (`height <= ?`).

3. Podmínka s řazením – Stejná podmínka jako v předchozím dotazu, jen se výsledky seřadily podle výšky sestupně a podle váhy vzestupně.
4. Hledání podle celého jména – Křestní jméno bylo vybráno náhodně s exponenciálním rozdělením, příjmení náhodně s rovnoměrným rozdělením a hledalo se podle podmínky `surname = ? AND name = ?`.
5. Hledání podle data – Dvě náhodně vygenerovaná data byla dosazena do podmínky `date_of_birth >= ? AND date_of_birth <= ?` (dřívější datum samozřejmě nahradilo první otazník, pozdější druhý).
6. Fulltextové hledání – Hledalo se ve sloupci ulice (*street*). Ze seznamu ulic byla jedna náhodně vybrána a z ní byl náhodně vybrán řetězec, mohlo se jednat o jeden znak, ale i celý název. Ohraničen z obou stran znakem `%` (tzn. jakýkoliv znak, jako v SQL) pak byl dosazen do podmínky `street LIKE ?`.

Průběh testování byl odlišný od prvního, dotazy se na serveru generovaly v jednom vlákně, takže databáze vždy zpracovávala jen jeden dotaz. Dotazů bylo vygenerováno tisíc, přičemž typ dotazu se vždy zvolil náhodně.

V databázích byly (kromě Elasticsearch) vytvořeny indexy pro rychlejší zpracování:

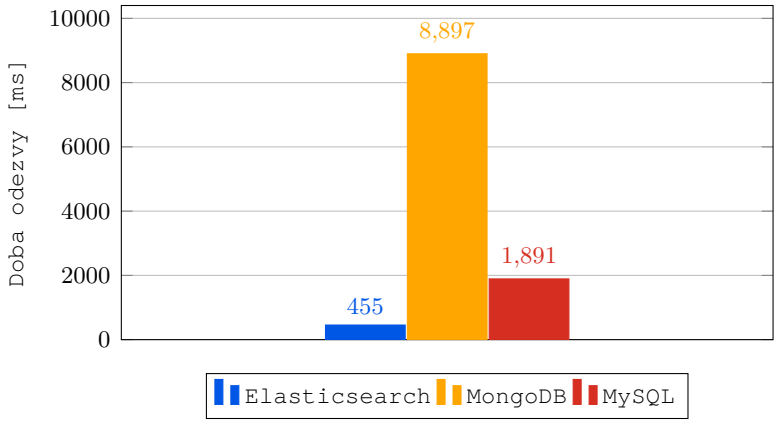
- MySQL měla index vytvořený nad sloupci *name*, *surname*, *date_of_birth*, *(height, weight)*, *(date_of_birth, height)*, kde poslední dva páry v závorkách jsou složené indexy nad dvěma sloupci. První pár by sice měl být ve tvaru *height DESC, weight*, ale MySQL zatím sestupný index nepodporuje, a proto byl také třetí dotaz v testování pro MySQL upraven pro vzestupné řazení.
- MongoDB mělo podobné indexy jako MySQL: *(name, surname)*, *date_of_birth*, *(height -1, weight)*, *(date_of_birth, height)*. U dvojice *height -1, weight* číslo za prvním sloupcem udává, že je index sestupný.
- Cassandra nepoužívá index při porovnávání větší/menší než, takže byl index vytvořen jen nad sloupci *name*, *surname* a *date_of_birth*. Poslední index pro fulltextové vyhledávání byl vytvořen nad sloupcem *street* následujícími dvěma příkazy:

```
CREATE CUSTOM INDEX street_idx ON person (street)
 USING 'org.apache.cassandra.index.sasi.SASIIndex';
```

```
CREATE CUSTOM INDEX street_idx2 ON person (street)
 USING 'org.apache.cassandra.index.sasi.SASIIndex'
 WITH OPTIONS = {
  'mode': 'CONTAINS',
  'analyzer_class': 'org.apache.cassandra.index.
                     sasi.analyzer.StandardAnalyzer',
  'case_sensitive': 'false'
 };
```

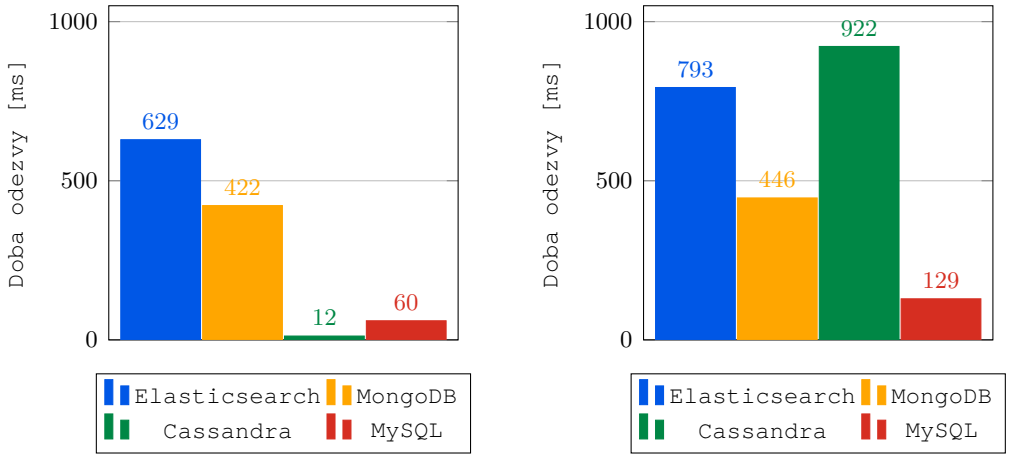
5.4.1 Výsledky jednotlivých dotazů

Prvním dotazem byla agregace s podmínkou, která nebyla testována u Cassandry, jelikož dotaz trval velmi dlouho (tj. více než půl minuty). Výsledky vidíme na obr. 5.4. Nejlépe si vedl Elasticsearch, který byl čtyřikrát rychlejší než MySQL a téměř dvacetkrát rychlejší než MongoDB.



Obrázek 5.4: Agregace s podmínkou, průměrná doba odezvy

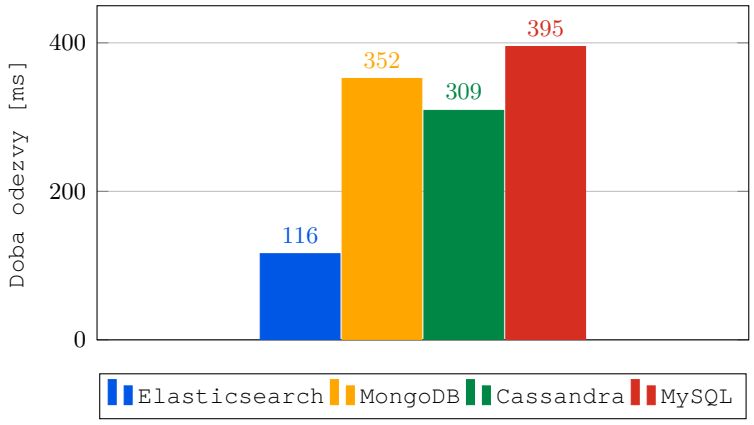
Další dva dotazy spolu souvisí, protože se jednalo o stejnou podmínku, jen podruhé se seřazením výsledků (obr. 5.5). Nejlépe zdánlivě dopadla MySQL, která však řadila vzestupně podle výšky i hmotnosti. Pokud by řadila podle výšky sestupně jako ostatní databáze, délka dotazu by přesáhla půl minuty. U Cassandry jsou výsledky řazeny v PHP, proto zde vidíme takový nárůst. MongoDB je s řazením v podstatě stejně rychlé jako bez, jelikož díky vytvořenému indexu vrací výsledky seřazené automaticky, i když to není požadováno. Elasticsearch tentokrát dopadl asi nejhůře.



(a) Podmínka (b) Podmínka s řazením

Obrázek 5.5: Podmínka s a bez řazení, průměrná doba odezvy

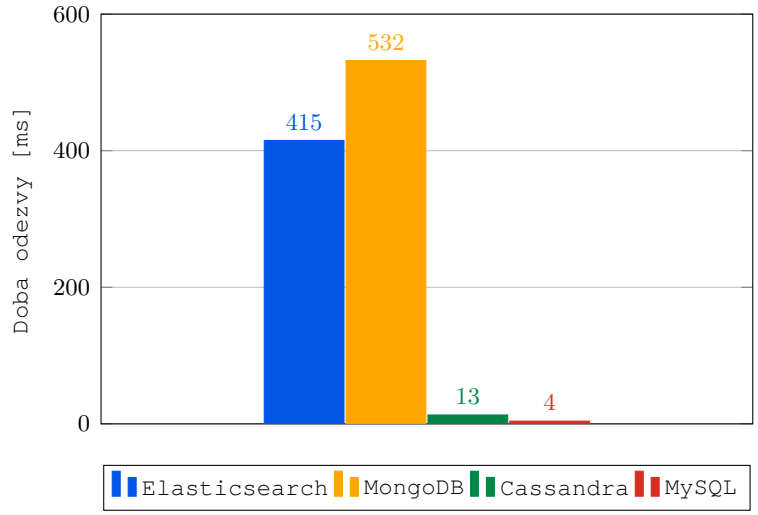
Čtvrtým dotazem bylo hledání podle celého jména, jehož výsledek vidíme na obr. 5.6, ve kterém byl Elasticsearch nejlepší. Ostatní databáze byly poměrně vyrovnané.



Obrázek 5.6: Hledání podle jména, průměrná doba odezvy

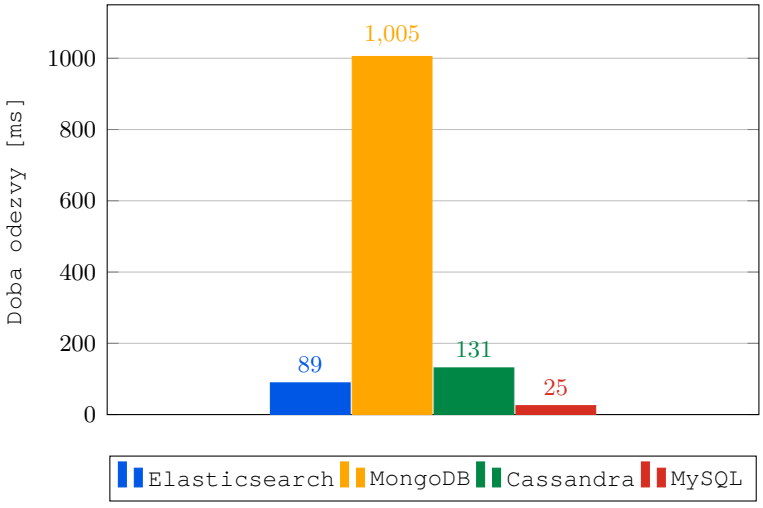
Při hledání podle data MySQL s Cassandrou pravděpodobně pomocí indexů výrazně předběhly ostatní. Elasticsearch a MongoDB se pak od sebe příliš nelišily. Výsledky jsou na obr. 5.7.

Posledním dotazem se testovalo fulltextové vyhledávání, které nejlépe dopadlo pro MySQL (viz obr. 5.8). Elasticsearch a Cassandra ale pořád výrazně předběhly MongoDB, kterému hledání průměrně trvalo 1 s. Sloupec, ve kterém se hledalo, však obsahoval krátký text, takže by se v nějakém



Obrázek 5.7: Hledání podle data, průměrná doba odezvy

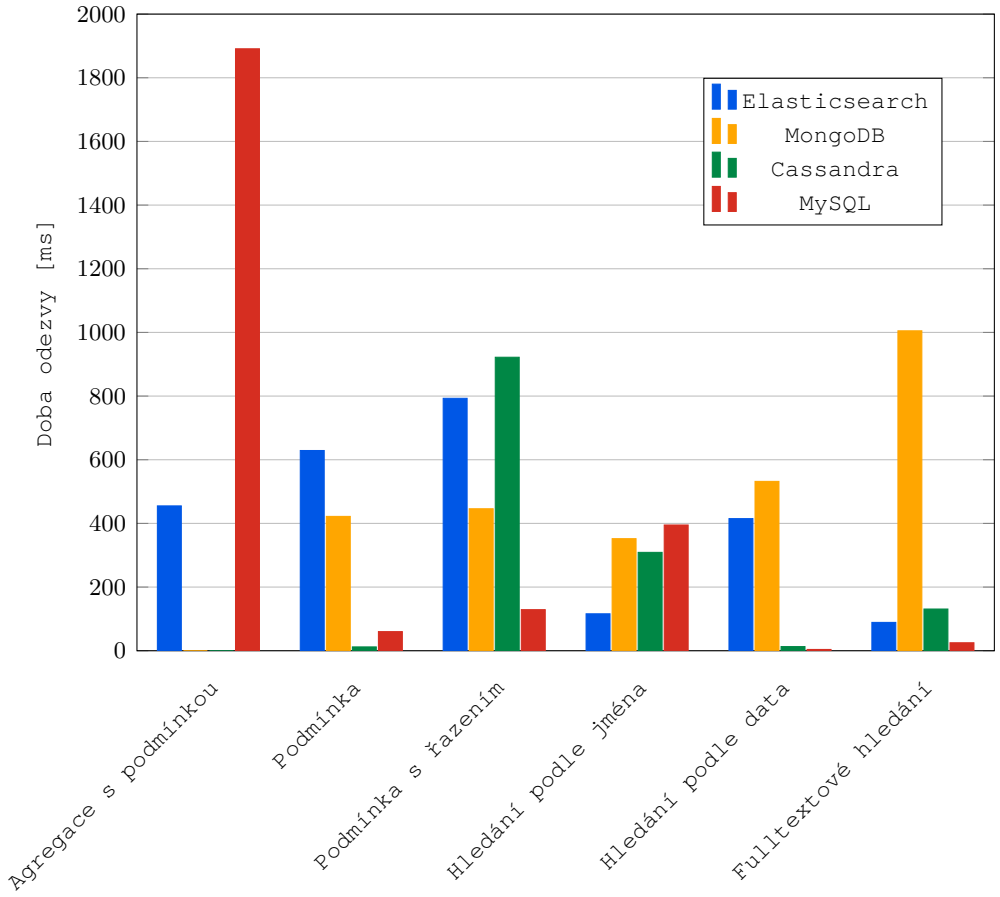
budoucím testování mohl zahrnout do dat také sloupec s delším textem, např. o rozsahu několika vět.



Obrázek 5.8: Fulltextové hledání (ulice), průměrná doba odezvy

5.4.2 Zhodnocení výsledků

Pro lepší představu se ještě můžeme podívat na graf na obrázku 5.9 obsahující všechny průměrné doby odezvy dotazů kromě *Agregace s podmínkou* pro MongoDB (průměrná doba 8,9 s by učinila graf nečitelným).



Obrázek 5.9: Všechny operace, průměrná doba odezvy

Jak je vidět, každá databáze je v některých dotazech lepší, v jiných horší. Poměrně dobrých výsledků dosáhla MySQL, ale jen u dotazů, u kterých používala index. Podobně je na tom Cassandra, která má ale různé limity při používání. Elasticsearch dokázal, že je stvořený ke hledání, jelikož také dosáhl poměrně pěkných výsledků, na rozdíl od jiných však bez vytváření indexů navíc. U ostatních databází vytvoření jednoho indexu trvalo rámcově 0,5-1 h. Nejhůře dopadlo MongoDB.

6 Zhodnocení práce s databázemi

V této kapitole zhodnotíme obtížnost práce a přívětivost jednotlivých databází od dokumentace k samotnému používání.

6.1 MySQL

Tato databáze je z námi hodnocených nejstarší a tudíž má velkou uživatelskou komunitu a také poměrně dobrou a rozsáhlou dokumentaci [6]. Za dlouhou dobu její existence se podařilo odstranit mnoho chyb a vylepšit různé algoritmy, takže podává poměrně slušné výsledky, jak jsme mohli vidět při testování. Aby mohla nabídnout co největší možnosti, začalo se také s implementací dokumentové databáze, která má ještě co dohánět, ale není úplně k zahození, viz první testování v kapitole 4.2. Velkou výhodou (ale i nevýhodou) je možnost mít jednu databázi, kterou budeme používat jako relační i dokumentovou.

Rychlost nahrávání dat byla víceméně podobná s MongoDB, 100 milionů záznamů bylo náhráno zhruba během osmi hodin.

Co se však týče škálovatelnosti, je na tom v porovnání s ostatními o dost hůře. Je možné nastavit nějaké repliky, ale potřebujeme k tomu několik různých procesů a celý postup je dosti komplikovaný. Zde je znát, že pro to nebyla stavěna, a že tyto možnosti byly až časem napasovány na něco, co už dlouho existovalo.

6.2 Cassandra

Dokumentace ke Cassandře [11] je ze všech nejhorší. Mnoho věcí v ní chybí, a nebo jsou dost stručné. Druhou možností kde hledat informace jsou stránky společnosti DataStax [15], která Cassandru používá a je i tvůrcem některých ovladačů (např. pro PHP).

Cassandra je stvořená pro rychlý zápis a to také potvrdila při nahrávání dat – 100 milionů záznamů za 5,5 h.

Příjemná je přítomnost jazyka CQL, který je velmi podobný SQL, a tudíž uživateli většinou blízký. Pozor si však musí dát na některé funkce, které v Cassandře nefungují, jako například logický operátor `OR`. Cassandra není stavěná pro sběr dat, o jejichž použití teprve zauvažujeme. Je rychlá, ale je třeba provést dobrou analýzu a návrh uložení dat, aby se pak dala dobře a rychle používat. O to víc toto platí při distribuované databázi, na kterou je Cassandra dobře stavěná. Sestává se z uzlů, které jsou si všechny rovny, není zde žádný bod selhání (*single point of failure*).

6.3 Elasticsearch

Vývoj Elasticsearch jde rychle kupředu, jen během psaní této práce bylo vydáno několik menších verzí. Dokumentaci k němu [8] ale přesto docela dobře udržují.

Velkou výhodou od ostatních databází je odlišnost ukládání dat. Elasticsearch používá Apache Lucene [9], který implementuje invertovaný index, díky němuž je pak vyhledávání velmi rychlé. Nevýhodou je pomalý zápis, protože se data musí předzpracovat a uložit do indexu. Ukládání 100 milionů záznamů trvalo přes 20 h!

Podobně jako Cassandra umí Elasticsearch dobře škálovat a uzly automaticky vytváří *cluster*. Díky komunikaci přes *REST* rozhraní je možné Elasticsearch použít téměř v jakémkoliv programovacím jazyce, nicméně při vývoji trochu chyběl jednoduchý *CLI* klient. Je sice možné psát dotazy pomocí programu *curl*, ale není to tak snadné. Druhou možností je použití *Kibany* [4], jejíž spuštění je přece jen náročnější než jen spustit příkazovou řádku a připojit se k databázi.

6.4 MongoDB

MongoDB je na rozdíl od Cassandry a Elasticsearch, kteří jsou napsáni v Javě, napsáno v C++ a C. Toto se např. projevuje při startu a vypnutí databáze, které je mnohem rychlejší, což člověk, pokud je provádí při vývoji často, ocení. Taktéž dokumentace [10] je pěkně zpracovaná a přehledná.

Rychlost nahrávání dat je také celkem dobrá, 100 milionů záznamů bylo nahráno zhruba za 7,5 h. Práce se *CLI* klientem je snadná a intuitivní. Nastavit distribuovanou databázi je možné bez obtíží, jeden server však figuruje

jako primární a ostatní jsou sekundární. Při výpadku si ale zbylé servery automaticky zvolí nový primární server. Pokud chceme data rozprostřít na více serverů (vytvořit tzv. *shards*), budeme potřebovat jeden další proces navíc, což je trošku komplikovanější než např. u Cassandry a Elasticsearch.

Celkově se s MongoDB pracovalo asi nejlépe ze všech databází, ale toto hodnocení je samozřejmě subjektivní.

7 Závěr

V této práci bylo úkolem prozkoumat možnosti ukládání velkých dat a vyhodnotit jednotlivé metody. Jak se asi dalo očekávat, není zde jasný vítěz, jelikož má vše svá pro a proti.

Podrobně jsme se zabývali databázemi MySQL (relační i dokumentová), Elasticsearch, MongoDB a Apache Cassandra. Z bádání vyplynulo, že je důležité vědět, co chceme s daty dělat a co od databáze očekáváme. Hodnotit je můžeme podle mnoha kritérií, proto zde uvedeme souhrnnou tabulku 7.1, ze které by některé vlastnosti mohly jít ihned vyčíst.

	Rychlost čtení podle ID	Rychlost úpravy	Rychlost hledání	Distribuovaná DB	Potřebuje schéma předem
MySQL (dokumentová)	+	0	-	-	ne
MySQL (relační)	N/A	N/A	+ (za použití indexů)	-	ano
Elasticsearch	-	-	+	+	ne
MongoDB	++	++	0	+	ne
Apache Cassandra	0	+	+ (za použití indexů)	+	ano

Hodnocení od nejlepšího po nejhorší: ++, +, 0, -.

Tabulka 7.1: Porovnání databází

MySQL jako relační databáze není v prvních dvou sloupcích hodnocena, jelikož nebyla zahrnuta v prvním testování.

Hodnocení v tabulce není úplně objektivní a mělo by být bráno s nadhledem. Při rozhodování je zajisté třeba se s databázemi seznámit více. Některé vlastnosti se však z tabulky vyčíst dají a můžeme ji použít jako počáteční bod, od kterého se lze odrazit.

Seznam zkratek

ACID Atomicity, Consistency, Isolation, Durability

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

BASE Basically Available, Soft-state, Eventually consistent

CLI Command-Line Interface

CQL Cassandra Query Language

CRUD Create, Read, Update, Delete

JSON JavaScript Object Notation

MySQL DS MySQL Document Store

PDO PHP Data Objects

PECL PHP Extension Community Library

RAM Random-Access Memory

REST Representational State Transfer

SQL Structured Query Language

TPC The Transaction Processing Performance Council

URL Uniform Resource Locator

XML Extensible Markup Language

YCSB Yahoo! Cloud Serving Benchmark

Literatura

[1] *JSON vs XML* [online]. W3Schools, 2017. [cit. 2017/03/13]. Dostupné z: https://www.w3schools.com/js/js_json_xml.asp.

[2] *HIGH PERFORMANCE BENCHMARKING: MongoDB and NoSQL Systems* [online]. United Software Associates, 2015. [cit. 2017/03/25]. Dostupné z: http://info-mongodb-com.s3.amazonaws.com/High%2BPerformance%2BBenchmark%2BWhite%2BPaper_final.pdf.

[3] *MongoDB PHP Library* [online]. MongoDB, 2017. [cit. 2017/06/05]. Dostupné z: https://docs.mongodb.com/php-library/master.

[4] *Kibana* [online]. Elastic, 2017. [cit. 2017/06/02]. Dostupné z: https://www.elastic.co/products/kibana.

[5] *DB-Engines Ranking* [online]. DB-Engines, 2017. [cit. 2017/03/15]. Dostupné z: http://db-engines.com/en/ranking.

[6] *MySQL Documentation* [online]. MySQL, 2017. [cit. 2017/03/13]. Dostupné z: https://dev.mysql.com/doc.

[7] *X DevAPI User Guide* [online]. MySQL, 2017. [cit. 2017/04/30]. Dostupné z: https://dev.mysql.com/doc/x-devapi-userguide/en.

[8] *Elasticsearch Reference* [online]. Elastic, 2017. [cit. 2017/03/13]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current.

[9] *Apache Lucene* [online]. The Apache Software Foundation, 2016. [cit. 2017/03/13]. Dostupné z: https://lucene.apache.org/.

[10] *The MongoDB 3.4 Manual* [online]. MongoDB, 2017. [cit. 2017/03/15]. Dostupné z: https://docs.mongodb.com/manual/.

[11] *Apache Cassandra* [online]. The Apache Software Foundation, 2016. [cit. 2017/03/23]. Dostupné z: http://cassandra.apache.org/.

[12] *The Transaction Processing Performance Council* [online]. TPC, 2017. [cit. 2017/03/24]. Dostupné z: http://www.tpc.org/.

[13] Chang, F. et al. Bigtable: A Distributed Storage System for Structured Data. *OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, s. 205–218.

[14] Cooper, B. F. et al. Benchmarking Cloud Serving Systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing*. June 2010, s. 143–154.

[15] DataStax. *CQL for Apache Cassandra 2.2 & later* [online]. DataStax, 2017. [cit. 2017/05/22]. Dostupné z: `https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlReferenceTOC.html`.

[16] DeCandia, G. et al. Dynamo: Amazon's highly available key-value store. *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, s. 205–220.

[17] Edlich, P. D. S. *NOSQL Databases* [online]. Prof. Dr. Stefan Edlich, 2009. [cit. 2017/03/22]. Dostupné z: `http://nosql-database.org/`.

[18] Foundation, N. *Nette Framework* [online]. Nette Foundation, 2017. [cit. 2017/05/15]. Dostupné z: `https://nette.org/en/`.

[19] I., H. et al. *Big Data a NoSQL databáze*. Grada Publishing, a.s., 2015. ISBN 978-80-247-5466-6.

[20] Nayak, A. – Poriya, A. – Poojary, D. Type of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*. March 2013, vol. 5, no. 4, s. 16–19.

[21] Penick, M. *DataStax PHP Driver for Apache Cassandra* [online]. GitHub, 2017. [cit. 2017/05/15]. Dostupné z: `https://github.com/datastax/php-driver`.

[22] Rabl, T. et al. Solving Big Data Challenges for Enterprise Application Performance Management. *Proceedings of the VLDB Endowment*. August 2012, vol. 5, no. 12, s. 1724–1735.

[23] Rethans, D. – Mikola, J. – Magnusson, H. *MongoDB driver for PHP* [online]. PECL, 2017. [cit. 2017/06/05]. Dostupné z: `https://pecl.php.net/package/mongodb`.

[24] Tiwari, S. *Professional NoSQL*. John Wiley & Sons, Inc., 2011. ISBN 978-0-470-94224-6.