VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DISTRIBUOVANÉ ZPRACOVÁNÍ ROZSÁHLÝCH DAT NA PLATFORMĚ JAVA

DISTRIBUTED BIG DATA PROCESSING ON THE JAVA PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JAKUB TUTKO**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/20647/2017/xtutko00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Tutko Jakub, Bc.**
Obor: Informační systémy
Téma: **Distribuované zpracování rozsáhlých dat na platformě Java**
Distributed Big Data Processing on the Java Platform
Kategorie: Informační systémy

Pokyny:

1. Seznamte se s problematikou distribuovaného zpracování rozsáhlých dat na platformě Java. Prostudujte existující řešení, zejména Hadoop, Apache Spark a další.
2. Prostudujte existující implementace grafových databází na platformě Java. Po dohodě s vedoucím vyberte vhodnou množinu těchto řešení.
3. Navrhněte architekturu aplikace pro testování možností jednotlivých řešení v distribuovaném prostředí. Zaměřte se na možnosti vkládání dat, dotazování a na výkonnost těchto operací.
4. Implementujte navrženou aplikaci pomocí vhodných technologií.
5. Proveďte experimentální testování zvolených databázových řešení na reálných datech pomocí vytvořené aplikace.
6. Sumarizujte dosažené výsledky.

Literatura:

- Lasila, I., Swick, R. R.: Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation 22 February 1999, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222
- Gupta, S.: Neo4j Essentials. Olton Birmingham: Packt Publishing, 2015.
- White, T.: Hadoop: the definitive guide. 3rd ed. Sebastopol: O'Reilly, 2012

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Burget Radek, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto práca sa zameriava na možnosti distribuovaného spracovania rozsiahlych dát na platforme Java s využitím grafových databáz. Analyzuje niekoľko distribúcii grafových databáz a spôsob ich prepojenia so systémom pre distribuované spracovanie dát, Apache Hadoop. Pre testovanie efektivity jednotlivých databázových riešení je výsledkom práce aplikácia, ktorá sťahuje dáta zo sociálnych sietí Twitter a Facebook. Tieto dáta je potom schopná zapísať a analyzovať pomocou dvoch rôznych databázových frameworkov. Jedná sa o frameworky Halyard a HGraphDB.

Abstract

This thesis is focused on the distributed Big Data processing on the Java platform, together with graph databases. It analyses several graph database distributions and the possibilities to connect them to the Apache Hadoop system for distributed data processing. For the purpose of testing database solutions effectiveness, the thesis outcome is an application, which is downloading data from social networks Twitter and Facebook. It is able to write and analyse data with two different database frameworks which are Halyard and HGraphDB.

Kľúčové slová

Hadoop, HBase, Spark, grafová databáza, RDF, Neo4j, RDF4J, JanusGraph, HGraphDB, Halyard, distribuovaná databáza, property graf

Keywords

Hadoop, HBase, Spark, graph database, RDF, Neo4j, RDF4J, JanusGraph, HGraphDB, Halyard, distributed database, property graph

Citácia

TUTKO, Jakub. *Distribuované zpracování rozsáhlých dat na platformě Java.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

Distribuované zpracování rozsáhlých dat na platformě Java

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Radka Burgeta, Ph.D.. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Jakub Tutko
14. mája 2018

Poďakovanie

Chcel by som sa poďakovať pánovi Ing. Radkovi Burgetovi, Ph.D. za odbornú pomoc pri tvorbe tejto práce.

Obsah

Kapitola 1

Úvod

V súčasnej dobe existuje stále viac dátových setov, ktoré nie sú vhodné pre relačný databázový model. Sú to dátové sety, ktoré sa zameriavajú na vzťahy medzi objektami. Ich najväčším tvorcom sú dnes sociálne siete. Relačný model prestáva byť pri vysokom počte vzťahov efektívny. Vhodnou alternatívou je v takomto prípade grafová databáza. Tá je optimalizovaná pre prácu s miliónmi objektov, prepojených rovnakým počtom vzťahov.

S narastajúcou veľkosťou dát postupne prestane úložný priestor jedného serveru postačovať. V takomto prípade je potrebné dáta distribuovať. Existuje viacero možností, ktoré umožňujú distribuované spracovávanie dát. Aktuálne je najpoužívanejšie open–source riešenie Apache Hadoop. Tomuto trendu sa postupne prispôsobil aj vývoj grafových databáz a dnes je možné si vybrať z viacero distribúcií, ktoré umožňujú svoje dáta distribuovať medzi jednotlivé stroje Hadoop systému.

Prvá kapitola je zameraná na analyzovanie možností spracovania rozsiahlych dát v Jave. Podrobnejšie rozoberá frameworky Apache Hadoop, Apache HBase a Apache Spark. Zameriava sa na ich architektúru a spôsobom akým s dátami narábajú.

Druhá kapitola sa zaoberá grafovými databázami. Na začiatku popisuje rôzne grafové modely. Neskôr sú v kapitole analyzované niektoré distribúcie grafových databáz. Kapitola obsahuje podrobnejšie informácie o databázach Neo4j, RDF4J a distribuovaných databázach JanusGraph, Halyard a HGraphDB.

Zameraním ďalšej kapitoly je návrh aplikácie pre testovanie efektivity vybraných databázových riešení. Orientuje sa na frameworky Halyard a HGraphDB. Výsledná aplikácia sťahuje dáta zo sociálnych sietí Twitter a Facebook, a ukladá ich pomocou zmieňovaných frameworkov. Vďaka tejto aplikácii je možné analyzovať efektivitu jednotlivých riešení pri vkladaní a dopytovaní sa nad dátami. Aplikácia umožňuje dáta pomocou oboch frameworkov vkladať a neskôr nad nimi vykonávať databázové dopyty. Výsledná aplikácia obsahuje niekoľko rôznych databázových dopytov, ktoré sú implementované oboma zmieňovanými frameworkami.

Nadväzujúca kapitola popisuje spôsob implementácie danej aplikácie. Obsahuje implementačné detaily časti aplikácie, ktorá sa zameriava na sťahovanie dát. Porovnáva rozdiely medzi získavaním podobných dát zo sociálnej siete Twitter a Facebok. V tejto časti sa čitateľ dozvie o limitoch používania ich API a o frameworkoch, ktoré celú komunikáciu s týmito sociálnymi sieťami zjednodušujú. Druhá časť kapitoly sa zmeriava už na databázové frameworky Halyard a HGraphDB. Popisuje spôsob, akým je možné pomocou nich dáta zapisovať a neskôr analyzovať databázovými dopytmi. Posledná časť tejto kapitoly popisuje implementáciu distribuovania výpočtu pomocou Spark frameworku.

Záverečná kapitola práce zhrňuje dosiahnuté výsledky. Porovnáva efektivitu zápisu totožných dát pomocou Halyard a HGraphDB. Čitateľ sa tak dozvie, ktorý framework dokáže rovnaké množstvo dát rýchlejšie zapísať a akú veľkosť budú tieto dáta vo výsledku zaberať. Ďalej kapitola obsahuje výsledné časy vykonávania implementovaných databázových dopytov. V texte je možné vidieť rozdielnu efektivitu dopytovania sa pomocou frameworkov Halyard a HGraphDB.

Kapitola 2

Distribuované spracovanie rozsiahlych dát

Žijeme v dobe, kedy sme neustále obklopený dátami. Ľudia stále fotia nové fotografie, nahrávajú videá, píšu správy svojim blízkym, nechávajú statusy a komentáre na Facebooku. K tomu sa ešte pridávajú stroje, ktoré generujú a ukladajú čoraz viac dát potrebných k rôznym analýzam.

Tento nárast dát ako prvých zasiahol veľkých technologických gigantov ako Google, Yahoo, Amazon a Microsoft. Potrebovali prechádzať terabyty až petabyty dát, aby zistili, ktoré webové stránky sú populárne, ktoré produkty su momentálne najviac žiadané, prípadne, ktoré reklamy sa zobrazili, ktorým užívateľom. Existujúce nástroje už pomaly prestali takéto nároky zvládať. Google prišiel ako prvý so systémom MapReduce, ktorý im umožňoval distribuované spracovávanie rozsiahlych dát. Tento systém vzbudil veľký záujem aj u mnohých iných firiem, no pre väčšinu z nich nebolo možné takýto systém vyvinúť. Doug Cutting využil túto príležitosť a pustil sa do vývoja open-source riešenia Googlovského MapReduce, nazývaného Hadoop. Dnes je Hadoop základným kameňom infraštruktúr viacerých veľkých webových spoločností, ako napríklad Yahoo, Facebook, LinkedIn a Twitter.[16]

Hadoop, respektíve distribuované spracovávanie rozsiahlych dát sa postupne stáva dôležitou znalosťou každého programátora. Časom sa zaradí k dôležitým znalostiam ako sú dnes relačné databázy, sieťové technológie a bezpečnosť.

2.1 Hadoop

Hlavným problémom spracovávania rozsiahlych dát je najmä to, že kým veľkosť diskov rapídne rastie, rýchlosť ich zápisu/čítania už tak nestúpa. Bežný HDD v 90tych rokoch mával kapacitu približne 40MB a prenosové rýchlosti okolo 1MB/s [1], to znamená, že sa dal celý disk prečítať zhruba do minúty. Dnešné pevné disky majú kapacitu okolo jedného terabajtu a prenosové rýchlosti okolo 100MB/s. Ich celé prečítanie teda môže trvať aj hodiny.[16]

Čas sa dá výrazne zrýchliť, ak použijeme väčšie množstvo diskov a dáta budeme čítať paralelne. Samozrejme pre optimálne paralelné čítanie musia byť dáta vhodne rozdelené. S použitím viacerých diskov nám navyše rapídne vrastie objem dát, ktoré sme schopní uložiť.

Veľkou nevýhodou distribuovaných dát je to, že je ľahké prísť o značnú časť dát. So zvyšujúcim počtom diskov totiž rastie aj pravdepodobnosť, že sa jeden z nich pokazí. Riešením

[1] Parametre disku Maxtor 7040A

tohto problému je redundancia dát. Dáta su replikované naprieč diskami, takže pri zlyhaní niektorého z diskov máme kópiu znehodnotených dát stále k dispozícii. Toto riešenie je už implementované napríklad v technológii RAID alebo v Hadoop Distributed Filesystem (HDFS), ktorý bude podrobnejšie rozoberaný neskôr.

Ďalším problémom je to, že pre analýzu uložených dát potrebujeme spájať dáta, ktoré môžu byť rozmiestnené medzi enormné množstvo diskov. Väčšina distribuovaných súborových systémov umožňuje kombinovať dáta z viacerých zdrojov, no implementácia tohto riešenia býva veľmi náročná. MapReduce poskytuje programovací model, ktorý vytvára nad čítaním a zápisom z viacerých diskov abstrakciu a transformuje to do výpočtu nad jedinou kolekciou kľúčov a hodnôt. MapReduce bude rovnako rozoberaný podrobnejšie v ďalších kapitolách.[20]

Hadoop poskytuje spoľahlivé zdieľané úložisko a systém pre analýzu dát. Úložisko je implementované pomocou HDFS a analýza pomocou MapReduce. Hadoop obsahuje aj ďalšie súčasti, no práve tieto dve tvoria jeho základ. Vďaka tomu Hadoop umožňuje písanie a spúšťanie distribuovaných aplikácií pre spracovávanie veľkého množstva dát. Hadoop pracuje na zhlukoch počítačov. Zhluk môže mať veľkosť od zopár uzlov až po desaťtisíce spolupracujúcich strojov. Podstatnou myšlienkou pri návrhu Hadoopu bolo, aby jednotlivé uzly nemuseli byť vysoko výkonnými servermi, ale spoľahlivo fungujúcimi aj na bežnom komoditnom hardvéri. Bežný hardware má však vysokú chybovosť. Hadoop je teda navrhnutý tak, aby sa jednoducho dokázal s výpadkom uzlov vysporiadať. Hlavnou výhodou Hadoopu je to, že je dostatočne jednoduchý a robustný zároveň. Znamená to, že bežný študent si vie jednoducho vytvoriť svoj vlastný Hadoop cluster[2] a takmer okamžite na ňom začať spúšťať svoje prvé úlohy, no je tiež vhodný pre tie najzložitejšie úlohy veľkých spoločností. Preto sa stal obľúbený ako na akademickej pôde, tak aj v korporátnych spoločnostiach.[16]

2.1.1 Apache Hadoop

Projekt Apache Hadoop je open-source framework, ktorý umožňuje distribuované spracovávanie rozsiahlych dátových setov nad zhlukom počítačov s pomocou jednoduchého programovacieho modelu. Je navrhnutý tak, aby ho bolo možné jednoducho rozšíriť z niekoľkých serverov na tisíce strojov s využitím všetkého výpočtového výkonu a úložiska, ktoré poskytujú. Namiesto toho, aby sa knižnica spoliehal na dostupnosť jednotlivých hardwarov, knižnica je sama o sebe navrhnutá tak, aby detekovala chyby už na aplikačnej vrstve. Vďaka tomu poskytuje vysoko spoľahlivú službu nad zhlukom počítačov aj napriek tomu, že na jednotlivých strojoch môže dochádzať ku chybám.[3]

Projekt je zložený z nasledovných modulov:

- **Hadoop Common**
  Všeobecné nástroje, ktoré využívajú ďalšie moduly.
- **Hadoop Distributed File System (HDFS)**
  Distribuovaný súborový systém, ktorý poskytuje vysoko priepustný prístup k aplikačným dátam.
- **Hadoop YARN**
  Framework pre rozvrhovanie úloh a manažment zdrojov.

[2] Výraz Hadoop cluster definuje množinu počítačov, ktoré jeden Hadoop systém využíva pre distribuované úložisko dát a úlohy s nimi spojené.

- **Hadoop MapReduce**
  Systém založený na YARN pre paralelné spracovávanie rozsiahlych dát.

Ďalšími projektami, ktoré spolupracujú s Hadoop sú:

- **Ambari**
  Webový nástroj pre správu a monitorovanie Apache Hadoop zhlukov.
- **Avro**
  Systém pre serializáciu dát.
- **Cassandra**
  Škálovateľná, distribuovaná databáza, odolná voči poruchám.
- **Chukwa**
  Systém pre správu veľkých distribuovaných systémov.
- **HBase**
  Škálovateľná, distribuovaná databáza, ktorá podporuje štrukturované dáta a rozsiahle tabuľky.
- **Hive**
  Systém pre jednoduché dopytovanie nad distribuovanými dátami.
- **Mahout**
  Knižnica pre strojové učenie a data mining.
- **Pig**
  Vysoko úrovňový programovací jazyk pre paralelné výpočty spolu s jeho prostredím pre vykonávanie.
- **Spark**
  Výpočtový nástroj pre Hadoop dáta.
- **Tez**
  Systém postavený nad Hadoop YARN, ktorý nahradzuje klasický Hadoop MapReduce. Používaný je napríklad v systémoch Pig a Hive.
- **ZooKeeper**
  Koordinačná služba pre distribuované aplikácie.

2.1.2 Hadoop Distributed File System HDFS

Ak veľkosť dát presiahne objem jedného lokálneho úložiska, jedným z riešení je dáta rozdeliť medzi viacero počítačov. Súborový systém, ktorý sa rozkladá na niekoľko sieťovo prepojených strojoch sa nazýva distribuovaný súborový systém. Keďže je potrebné dáta prenášať po sieti, je distribuovaný systém omnoho komplexnejší ako systém lokálny. Jednou z implementácií takéhoto systému je HDFS.

HDFS je súborový systém pre rozsiahle distribuované dáta, fungujúci na komoditných hardvéroch. Je prevádzkovaný ako abstrakcia nad natívnym súborovým systémom. Jeho výhodou oproti bežným distribuovaným súborovým systémom je jeho vysoká tolerancia voči chybám a to, že dokáže fungovať aj na bežných nízko-nákladových zariadeniach.

So súbormi o veľkosti niekoľkých terabajtoch je v bežnom súborovom systéme obtiažne narábať. HDFS je však optimalizovaný pre prácu so súbormi o veľkostiach aj v stovkách terabajtov. Súbor je schopný rozdeliť medzi viacero uzlov a rýchlo v ňom čítať, prípadne zapisovať do neho ďalšie dáta.

Chybovosť hardvérov je ich bežná súčasť a nevyskytuje sa ojedinele. Pri používaní tisícov hardvérov v jednom zhluku je takmer isté, že aspoň jeden hardvér bude v daný okamžik mimo prevádzku. Jednou zo základných požiadaviek pri vytváraní HDFS bola vysoká tolerancia voči chybám. HDFS dokáže takmer ihneď chybu detektovať a automaticky chvíľkový výpadok dát vyriešiť. To všetko sa deje bez toho, aby si užívateľ niečo všimol.

HDFS je založený na princípe pre prístup k súboru *write-one-read-many*[3]. Súbor, ktorý je vytvorený, zapísaný a uložený nie je možné ďalej meniť. Jedinou výnimkou je zápis na koniec súboru a orezanie jeho časti. Vďaka tomuto prístupu HDFS poskytuje vysoko rýchlostné čítanie dát. Aplikácie pracujúce nad HDFS tak musia rátať s vyššou dobou odozvy. Rýchlosť náhodného prístupu je na úkor rýchleho čítania pomalšia ako v bežných súborových systémoch.

Výpočet prevádzaný aplikáciou je optimálnejší, ak sú požadované dáta v jeho blízkosti a nie je potrebné ich presúvať po sieti. Optimálnejší je najmä, ak sa jedná o rozsiahle dátové sety. Sieť je z tohoto dôvodu menej preťažená, čo výrazne zrýchľuje celý beh aplikácií. HDFS poskytuje rozhranie pre presun výpočtu k dátam, ktoré potrebuje.[3]

Dátové bloky HDFS

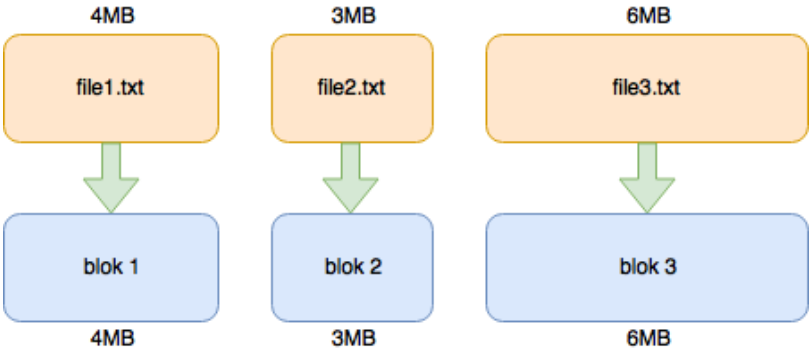
Alokačný blok disku je minimálna veľkosť dát, ktoré môžu byť z disku čítané alebo zapisované. Bežný súborový systém ukladá všetky svoje dáta do takýchto blokov. Každý blok má vždy rovnakú veľkosť nezávisle na tom, koľko dát je v ňom reálne zapísaných. Bežná veľkosť bloku zvykne byť štyri kilobajty. Pri zápise väčšieho súboru je súbor rozdelený medzi viacero takýchto blokov.

HDFS je tiež založený na koncepte blokov. Bloky majú omnoho väčšiu veľkosť a to obecne 128MB, no túto hodnotu je možné meniť.[3] Oproti alokačným blokom však majú jednu významnú vlastnosť. Pri zápise súboru menšieho ako veľkosť bloku sa na disku nezaplní miesto bloku, ale len veľkosť daného súboru (obr. 2.1).[20] To znamená, že veľkosť bloku pre súbor o veľkosti 4MB bude len 4MB. V opačnom prípade pri zápise veľkého súboru, bude mať posledný blok tiež len obmedzenú veľkosť (obr. 2.2). Vďaka tejto vlastnosti sa dostupný úložný priestor nezaplní rýchlejšie ani pri používaní menších súborov.

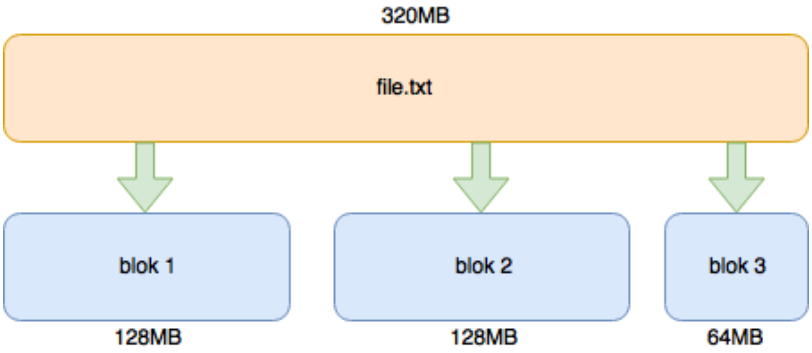
Výrazne väčšie bloky oproti blokom na bežných súborových systémoch majú v Hadoop systéme viaceré výhody. HDFS bol navrhnutý pre ukladanie rozsiahlych súborov, ktoré môžu dosahovať aj stovky terabajtov. Ak by sme použili klasické veľkosti blokov, udržanie všetkých ich metadát by bolo neúnosné. Navyše, zredukovaním počtu blokov potrebných pre prečítanie jedného súboru klesá aj celkový čas ich nájdenia. Používanie väčších blokov má aj svoje nevýhody. Výrazne tým narastá čas potrebný na prečítanie posledných dát daného bloku.[20]

Po rozdelení súboru na menšie bloky, je potom snahou HDFS bloky rovnomerne rozmiestniť medzi jednotlivé uzly. Ideálnym stavom je, aby bol každý blok na samostatnom stroji.[3] V takomto prípade je možné každý blok čítať paralelne, vďaka čomu sa rýchlosť prečítania súboru niekoľko násobne zrýchli.

[3] WROM - *raz zapíš, mnohokrát čítaj* je princíp hovoriaci o tom, že dáta, ktoré sú už zapísané, sú prístupne len pre čítanie a nie je ich možné viac upravovať.

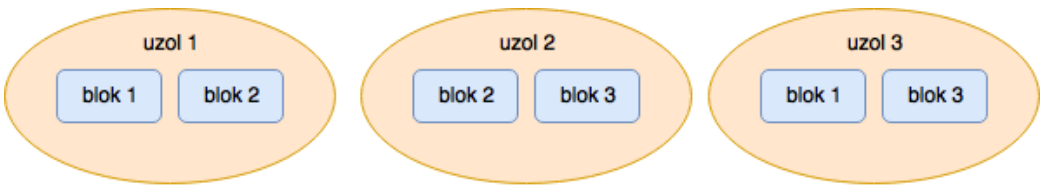


Obr. 2.1: Súbory menšie ako veľkosť obecného HDFS bloku sa uložia do menších blokov.



Obr. 2.2: Posledný HDFS blok pre uloženie veľkého súboru bude mať len potrebnú veľkosť.

Využívanie konceptu blokov umožňuje aj jednoduchú repliku dát. Každý zapísaný blok sa automaticky niekoľko krát duplikuje a každá kópia je zapísaná na iný fyzický uzol (obr. 2.3). V štandardnom nastavení má každý blok presne tri kópie. Replikačný faktor je možné zmeniť v nastaveniach HDFS. Pri narušení nejakého bloku, prípadne celého stroja sú dáta načítané okamžite z duplikátu, bez toho, aby užívateľ niečo postrehol. Vypadnutý blok je znovu nahradený, aby bola opäť dodržaná trojstupňová replikácia.[3]



Obr. 2.3: HDFS bloky sú rovnomerne replikované medzi jednotlivé uzly.

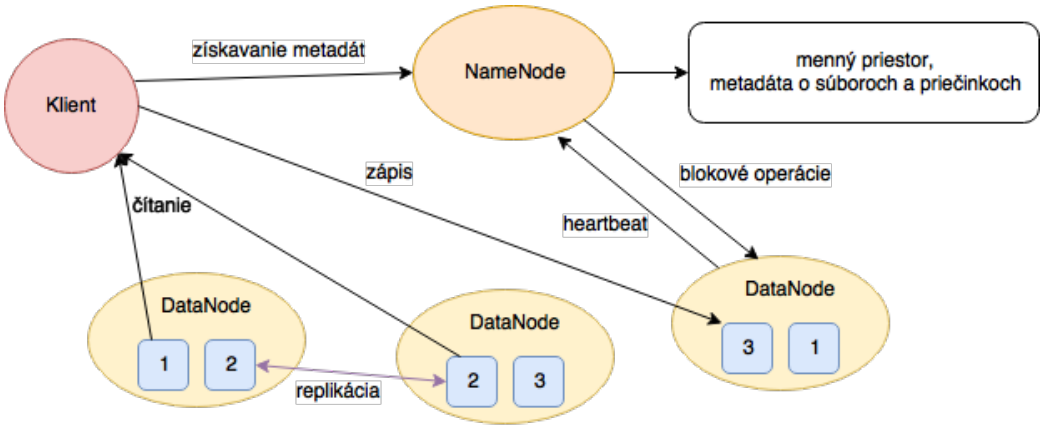
Architektúra HDFS

HDFS má *master–slave* architektúru.[20] Celý HDFS cluster je zložený z jedného master uzlu, nazývaného NameNode a viacerých slave uzlov, nazývaných DataNode. NameNode je zodpovedný za správu menného priestoru a metadát všetkých súborov a priečinkov súborového systému. Tieto informácie sú uložené na lokálnom disku NameNode v dvoch súboroch. Menný priestor je uložený v súbore `FsImage` a metadáta v súbore `EditLog`.[3] `EditLog` ob-

sahuje okrem samotných metadát aj všetky zmeny, ktoré sa v súborovom systéme odohrali. NameNode taktiež reguluje užívateľom prístup k súborom. Pri zápise súboru, si NameNode ukladá informácie o tom, ktorý blok sa zapísal do ktorého DataNode. Tieto informácie neukladá trvalo na svoj disk, ale preráta ich pri každom štarte systému.

Úloha DataNode je veľmi jednoduchá. Príma požiadavky od HDFS klientov pre čítanie a zápis dát, a periodicky oznamuje NameNode, že je stále aktívny a ktoré bloky obsahuje. Tieto správy sa nazývajú `Heartbeat`. Ak NameNode detekuje, že sa DataNode dlhšiu dobu nehlási, uloží si ho ako neaktívny a prestane ho využívať.[3]

NameNode je pre HDFS veľmi dôležitý. Pri jeho výpadku klienti nie sú schopní súbory čítať ani zapisovať. Metadáta uložené v NameNode sa v štandardnom nastavení HDFS nezálohujú. HDFS však umožňuje, aby NameNode svoje metadáta ukladal okrem lokálneho disku aj na disk vzdialený. Toto zabezpečenie umožňuje NameNode po jeho výpadku obnoviť.



Obr. 2.4: HDFS architektúra.

Obrázok 2.4 ukazuje architektúru HDFS. NameNode udržuje informácie o mennom priestore a metadáta o súboroch a priečinkoch potrebné pre beh systému. NameNode navyše riadi všetky blokové operácie na jednotlivých DataNode. Rozhoduje napríklad o tom, ktoré bloky sa replikujú a kde sa uloží ich záloha. Tieto bloky presúvajú DataNode samy medzi sebou.[3]

HDFS klient pri čítaní najskôr kontaktuje NameNode a získa informácie, kde sa nachádzajú všetky bloky hľadaného súboru. S touto informáciou klient komunikuje už len s DataNode. Zápis prebieha podobne ako čítanie. Klient kontaktuje NameNode a ten poskytne informácie o DataNode, kam sa majú dáta zapísať. Klient potom dáta zapisuje priamo do DataNode. Čítanie aj zápis môže prebiehať paralelne, ak sa bloky nachádzajú na rôznych DataNode.[20]

NameNode a DataNode sú programy napísané v Jave, ktoré sú navrhnuté tak, aby fungovali na bežnom komoditnom hardvéri. Znamená to, že môžu byť spustené na ľubovoľnom stroji, ktorý podporuje Javu. Typické prostredie obsahuje jeden fyzický stroj, na ktorom je spustený NameNode a ďalšie stroje hosťujúce DataNode. V HDFS je možné, aby bežalo viacero DataNode na jednom fyzickom stroji, no v reálnych prípadoch sa to nepoužíva.[3]

2.1.3 Hadoop MapReduce

MapReduce je programovací model pre spracovávanie dát. Hadoop MapReduce je založený na tomto modeli a bol navrhnutý pre jednoduché písanie programov, ktoré spracovávajú obrovské množstvo dát paralelne na veľkom Hadoop clusteri komoditných počítačov. Hadoop umožňuje, aby boli MapReduce programy napísané v rôznych programovacích jazykoch. Medzi najpoužívanejšie patria Java, Ruby, Python a C++.[20]

Výpočet Hadoop MapReduce programu je rozdelený do dvoch fáz. Na začiatku sú dáta spracované Map funkciou a jej výstup je spracovaný pomocou Reduce funkcie. Obe tieto funkcie pracujú paralelne na všetkých dostupných uzloch Hadoop clusteru. Pre jednoduchý spustiteľný MapReduce program stačí definovať lokácie vstupných a výstupných dát, a implementovať funkcie Map a Reduce. Obe funkcie pracujú výlučne s dvojicami (kľúč, hodnota), anglicky (key, value). Tieto dvojice očakávajú na vstupe a vracajú ich na výstupe.[3]

Keďže Hadoop MapReduce beží na spoločnom clusteri ako HDFS, výpočet sa vykonáva na rovnakých strojoch ako sú uložené dáta. Hadoop MapReduce tohto faktu využíva a snaží sa vykonávať jednotlivé programové úlohy na počítačoch, kde sú uložené dáta potrebné pre ich beh. Je omnoho jednoduchšie presunúť pomerne krátky kód výpočtu, ako presúvať obrovské dátové sety.

Hlavnou výhodou MapReduce je jeho abstrakcia nad Hadoop clusterom. Programátor sa nemusí zameriavať nad tým, ako program efektívne pararelizovať a ako rozdeliť dáta medzi jednotlivé uzly distribuovaného systému. Vďaka tomu môže väčšinu svojho času zamerať na samotný algoritmus výpočtu. Škáľovateľnosť Hadoop systému navyše zaručuje, že program navrhnutý pre cluster o veľkosti dvoch počítačov bude pracovať rovnako na clusteri s tisíckami strojov.

Map

Na začiatku MapReduce výpočtu sa vstupné dáta rozdelia na dvojice (kľúč, hodnota). Pri čítaní bežného textového súboru to môžu byť napríklad číslo a text daného riadku. Po rozdelení sú tieto dvojice pridelené medzi jednotlivé uzly Hadoop clusteru. Dvojice sa prideľujú na základe ich fyzického uloženia. Snahou totiž je, aby sa dáta spracovali na uzle, kde sú uložené a nemuseli sa posielať po sieti. Uzol, ktorý vykonáva Map funkciu sa nazýva Mapper.

Potom ako Mapper príme vstupnú kolekciu dvojíc, začne ich transformovať na výstupné dvojice. Transformácia sa deje na základe implementácie Map funkcie. Výsledok transformácie je potom uložený na lokálny disk daného uzlu. Veľkosť vstupnej a výstupnej kolekcie dvojíc nemusí byť rovnako veľký.[13]

Príklad fungovania Map funkcie si môžeme ukázať na MapReduce programe, ktorý ráta počet výskytov slov v textovom súbore. Mapper dostane na vstupe dvojice zložené z čísla a textu riadku. Na výstupe Mapper vygeneruje kolekciu dvojíc, v ktorých kľúč je slovo a hodnota značí počet výskytov daného slova, teda číslo jeden. Pre riadok `"Na strome je jablko"` by bol výstup kolekcia dvojíc `"(Na,1),(strome,1),(je,1),(jablko,1)"`.

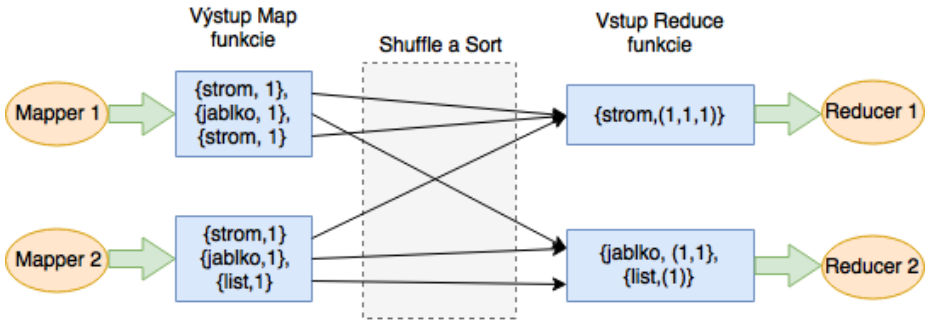
Combine

Medzi fázami Map a Reduce prebieha ešte jedna, nemenej podstatná fáza, nazývaná Combine. Táto fáza je rozdelená na dve operácie Shuffle a Sort. Ich úlohou je vziať výstup operácie Map a vhodne transformovaný ho priradiť uzlu Reduce. Táto fáza je už imple-

mentovaná v rámci MapReduce a programátor ju má možnosť nastavovať len pomocou MapReduce konfigurácie.[3]

Shuffle operácia vyberie vhodný Reducer pre spracovanie daného kľúča a presunie k nemu potrebné dáta. Operácia Sort spojí relevantné dáta s rovnakým kľúčom do kolekcie hodnôt.[13]

Príklad fungovania Shuffle a Sort operácií je možné vidieť na obrázku 2.5.



Obr. 2.5: Príklad fungovania MapReduce operácií Shuffle a Sort.

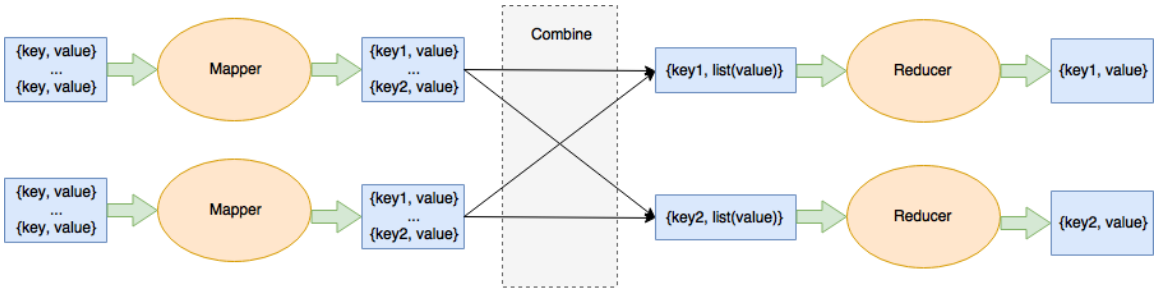
Reduce

Posledná fáza Mapreduce programu je Reduce operácia. Reducer prijíma kolekciu hodnôt s rovnakým kľúčom a redukuje ju na menší počet hodnôt. Vo väčšine prípadov vracia Reducer práve jednu hodnotu pre daný kľúč. Reduce operácie fungujú pre každý kľúč paralelne. Výstup operácie je uložený na lokálny disk uzlu.

V MapReduce programe nie je vykonanie fázy Reduce povinné. V takomto prípade sa nevykoná ani fáza Combine a výsledkom programu je výstup Map fázy.[3]

Priebeh Reduce operácie programu, ktorý ráta počet výskytov slov v texte je triviálny. Tento program bol spomínaný pri ukážke fungovania Map funkcie. Reducer dostane na vstupe dvojicu zloženú z kľúča a kolekcie hodnôt. Kľúč v tomto prípade predstavuje konkrétne slovo, ktorého výskyt rátame a kolekcia hodnôt obsahuje jednotlivé výskyty tohto slova. Reducer potom jednoducho zráta sumu hodnôt a tým dostane počet výskytov daného slova. Pre vstup `"(strom,(1,1,1,1))"` by bol výstup Reduce funkcie dvojica `"(strom,4)"`, ktorá značí štyri výskyty tohto slova.

Spoločné fungovanie všetkých troch fáz MapReduce je možné vidieť na obrázku 2.6.



Obr. 2.6: Hadoop MapReduce výpočet.

2.2 HBase

HBase je open-source distribuovaná databáza, ktorá funguje na Hadoop clusteri. Je navrhnutý tak, aby bol jednoducho škálovateľný a umožňoval spracovávanie obrovských dátových setov. Aplikáciam poskytuje rýchly náhodný prístup k dátam.

Dáta sú v HBase ukladané do tabuliek. Tabuľka pozostáva zo stĺpcov, ktoré spoločne vytvárajú jeden riadok tabuľky. Každý riadok má svoj jedinečný identifikátor, nazývaný *row key*. Riadky sú vždy usporiadané lexikálne práve podľa tohto identifikátoru. *Row key* má v HBase tabuľke rovnakú úlohu ako primárny index v klasickej relačnej databáze.

Stĺpce tabuľky sú spojené do takzvaných *column families*. Tieto skupiny vytvárajú sémantické hranice medzi dátami riadkov. Pri vytváraní HBase tabuľky sa definujú práve tieto *column families*. Samotné stĺpce sa vytvárajú až pri vkladaní dát. HBase neodporúča meniť už vytvorené *column families* a mať ich v tabuľke príliš veľké množstvo. V súčasnosti HBase odporúča maximálne tri *column families*. Vyšší počet zapríčiní výrazný pokles výkonnosti databázy. Naopak počet stĺpcov v jednej *column family* nie je nijako obmedzený. *Column family* môže obsahovať aj milióny stĺpcov. Štruktúra tabuľky je znázornená na obrázku 2.7.[1]

Samotné dáta sa v HBase ukladajú do buniek tabuľky. Štrukturovaná informácia jednej bunky sa nazýva *KeyValue*. Obsahuje *row key*, názov *column family*, názov stĺpca, časovú známku a hodnotu bunky. Každá bunka tabuľky, ktorá obsahuje dáta nesie so sebou aj všetky tieto informácie.[17]

RowKey	Column Family 1		Column Family 2	
	Column 1	Column 2	Column 3	Column 4
row1	KeyValue1	KeyValue2	KeyValue3	KeyValue4
row2	KeyValue5	KeyValue6	KeyValue7	KeyValue8

Obr. 2.7: Štruktúra HBase tabuľky.

Zaujímavou vlastnosťou HBase databázy je jej schopnosť uchovávať históriu zmien. Pri zmene dát sa nové dáta vytvoria s novou časovou známkou a pôvodné dáta v databáze ostanú. Tieto dáta je možné rozlíšiť na základe ich časovej známky. Pri dopytovaní sa na bunky tabuľky je potrebné uviesť aj verziu uložených dát. Ak sa verzia nešpecifikuje, štandardne sa vrátia najnovšie dáta. Koľko verzií dát sa uloží je možné definovať pri vytváraní *column family*. Explicitne HBase ukladá len jednu verziu dát.[1]

Tabuľky sú rozdelené na základe ich *row key* do regiónov. Keďže sú riadky usporiadané lexikálne, každý región obsahuje určitý rozsah identifikátorov. Tieto regióny sú potom distribuované medzi uzly Hadoop clusteru. HBase bol navrhnutý pre jednoduché pridávanie ďalších ulov clusteru. Preto celá distribúcia dát prebieha na pozadí bez vedomia užívateľa.

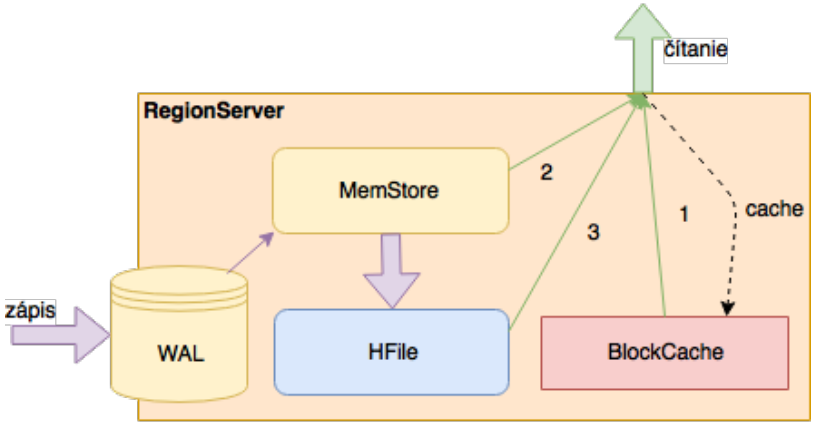
Databáza HBase je orientovaná na miesto riadkov do stĺpcov. Takéto databázy sa nazývajú *column-oriented*. Rozdiel stĺpcových databáz oproti bežným riadkovým databázam je v spôsobe uloženia ich dát. Riadkové databázy ukladajú spoločne dáta týkajúce sa jedného riadku. Narozdiel od toho, stĺpcové databázy spájajú dáta, ktoré majú spoločný stĺpec. Tento spôsob uloženia je v HBase výhodný. HBase je navrhnutý pre ukladanie rozsiahlych dátových setov, ktorých tabuľky môžu obsahovať milióny riadkov a státisíce stĺpcov. Prehľadávať takéto tabuľky riadok po riadku by bolo veľmi neefektívne, keďže väčšina dát je pre náš dopyt neadekvátna. Ďalšou výhodou tejto orientácie je jej škálovateľnosť. V HBase je omnoho jednoduchšie rozšíriť tabuľky o nové stĺpce.[11]

2.2.1 HBase architektúra

Architektúra HBase systému je založená na princípe *master-slave*. Systém sa skladá z troch druhov fyzických serverov: *RegionServer*, *HMaster* a *Zookeeper*. Vyššie bolo spomenuté, že tabuľky sa rozdeľujú do regiónov po skupinách riadkov. Tieto regióny su distribuované medzi uzly Hadoop clusteru. Uzol, ktorý ukladá tieto regióny a poskytuje ich pre úpravy a čítanie sa nazýva *RegionServer*. Jeden server môže spravovať viacero regiónov. *HMaster* rozhoduje o tom, ktoré regióny sa uložia do ktorého *RegionServeru*. Jeho ďalšou úlohou je vytvárať, upravovať a mazať HBase tabuľky. Pre správne fungovanie distribuovaného systému je nutnosťou udržovať stav jednotlivých uzlov clusteru. Zisťovanie informácií o stave uzlov je úlohou serveru nazývaného *Zookeeper*.

Dáta sú fyzicky uložené v *RegionServeroch*. HBase systém obsahuje viacero takýchto serverov, kde je každý zodpovedný za svoje regióny. *RegionServer* umožňuje klientom čítať dáta, ktoré obsahuje a zapisovať doň nové dáta. Keďže klient komunikuje priamo so servermi obsahujúcimi fyzické dáta, čítanie aj zápis je možné vykonávať paralelne. *RegionServer* je zložený zo štyroch súčastí: *WAL*, *MemStore*, *HFile* a *BlockCache* (obrázok 2.8).

WAL[4] ukladá všetky novo prijaté dáta. Udržuje ich pokiaľ sa trvalo nezapíšu na disk serveru. Keďže uloženie dát na disk sa nedeje okamžite, *WAL* slúži aj pre obnovu stratených dát. Po zápise do *WAL* sa dáta presunú do pamäťového úložiska *MemStore*. Jeho úlohou je akumulovať prijaté dáta a po dosiahnutí určitej veľkosti ich uložiť vždy do nového *HFile* súboru. *HFile* je klasický HDFS súbor uložený na disku, ktorý obsahuje zoradené *KeyValues*. Jeden server môže obsahovať aj niekoľko takýchto súborov. *BlockCache* slúži ako vyrovnávacia pamäť pre najčastejšie pristupované dáta. Pri získavaní dát prehľadáva *RegionServer* najskôr *BlockCache*, potom *MemStore* a ako posledný *HFile*.[20]



Obr. 2.8: Architektúra RegionServeru.

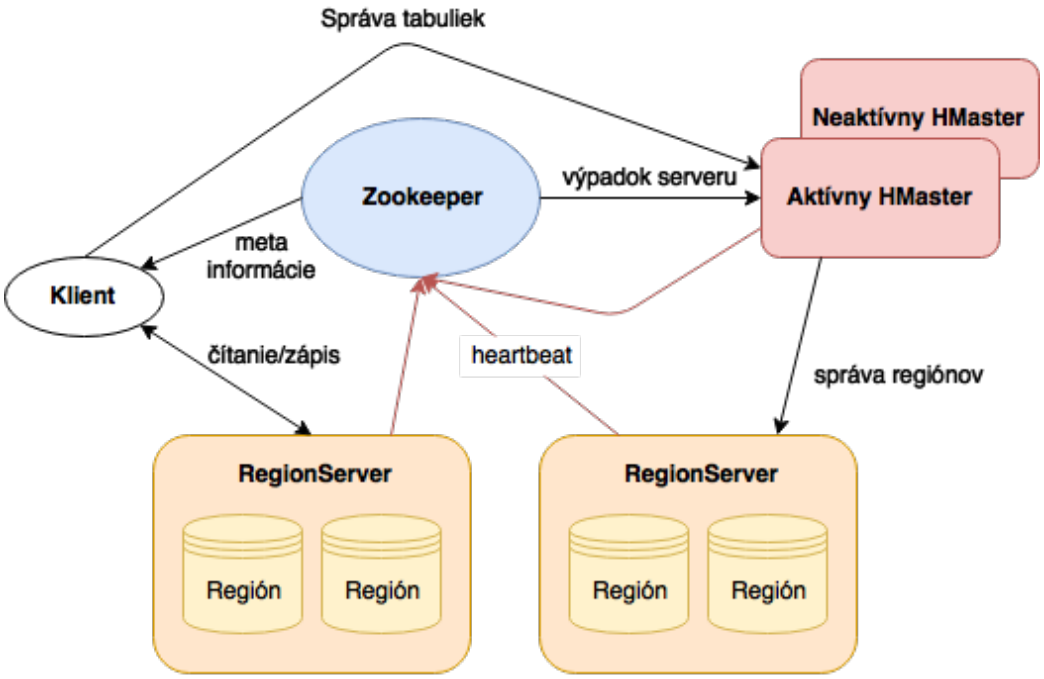
Úlohou už spomínaného master serveru nazývaného *HMaster* je spravovať rozloženie regiónov v HBase systéme. Na začiatku pridelí regióny jednotlivým *RegionServerom*. Počas behu systému pravidelne kontroluje vyťaženosť serverov a ak zistí, že je niektorý z nich preťažený, presunie jeho regióny na iný server. *HMaster* je navyše zodpovedný za vytváranie, mazanie a úpravu tabuliek. HBase systém môže obsahovať viacero master serverov. Podmienkou je, že v jednom okamihu môže byť len jeden z nich aktívny. Neaktívne serveri slúžia ako záloha v prípade výpadku aktívneho *HMaster*.

[4]Write Ahead Log

Zookeeper je samostatný *open-source* projekt. Jedná sa o spoľahlivú, distribuovanú, koordinačnú službu Hadoop clusteru. *Zookeeper* zisťuje, ktorý server systému je aktívny a poskytuje informácie o ich výpadkoch. Každý *RegionServer* si vytvorí svoj vlastný uzol v službe *Zookeeper*, ktorý sa nazýva *ephemeral*. Uzly používa master server pre objavenie všetkých dostupných *RegionServerov* a pre odhalenie výpadku niektorého zo serverov. *HMaster* takisto vytvára svoj *ephemeral* uzol. Ten však okrem držania informácie o aktivite tohto serveru zaručuje, že v HBase systéme bude len jeden aktívny master server. *Zookeeper* rozozná prvý *HMaster* a všetky ostatné označí ako neaktívne.

Master aj region servery musia pravidelne obnovovať svoj *ephemeral* uzol za pomoci *heartbeat* správ. Ak *Zookeeper* zistí, že niektorý uzol nie je dlhšiu dobu obnovený, zmaže tento uzol a oznámi to náležitému serveru. Výpadky bežného *RegionServeru* sú oznamované aktívnemu *HMaster* a ten dáta jednoducho replikuje na iný aktívny regionálny server.

Pred zápisom/čítaním dát, klient získa od *Zookeeper* serveru informácie o *RegionServeri*, ktorý obsahuje meta tabuľku. Jedná sa o klasickú HBase tabuľku, ktorej obsahom je umiestnenie všetkých regiónov systému. Klient potom požiada server obsahujúci tieto dáta o region server, ktorý obsahuje hľadaný *row key*. S touto informáciou klient kontaktuje konkrétny *RegionServer* a požiada, prípadne zapíše do neho dáta. Klient si informácie o danej lokácií uloží a opätovne sa už *Zookeper* serveru nedopytuje.[11]



Obr. 2.9: HBase architektúra.

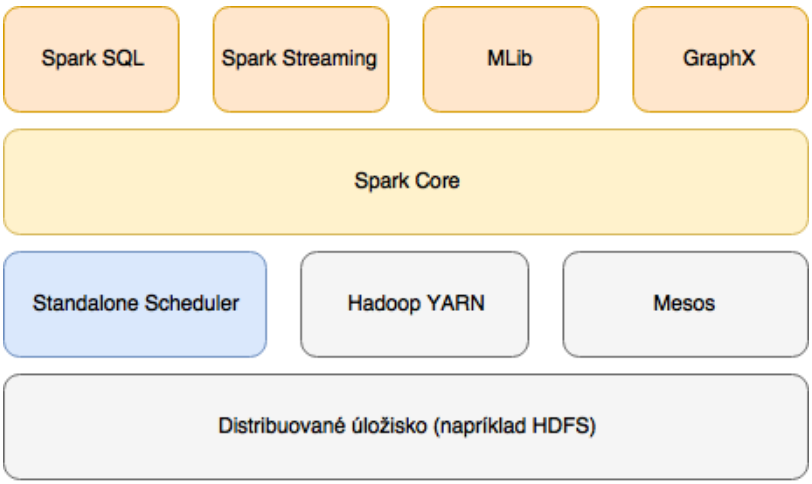
2.3 Spark

Apache Spark je *open-source* projekt pre distribuované spracovávanie rozsiahlych dát. Rozširuje MapReduce model o viaceré typy možných výpočtov, napríklad interaktívne dopyty a prúdové spracovanie dát. Snahou Sparku je čo najviac zrýchliť spracovávanie. Jeho najvýznamnejšou vlastnosťou pre rýchlosť spracovania dát je schopnosť uskutočňovať výpočty v pamäti počítačov. Okrem toho je Spark efektívnejší aj v spracovaní na disku. Spark

na svojich stránkach tvrdí, že jeho programy fungujú v pamäti 100-krát a na disku 10-krát rýchlejšie ako Hadoop MapReduce.[2]

Spark bol vyvinutý v jazyku *Scala*, no funguje na klasickom virtuálnom stroji Javy. Umožňuje mu to beh na všetkých platformách, ktoré podporujú Javu. Vývoj Sparkových aplikácií je možný v programovacích jazykoch Java, Scala, Python a R.

Projekt Spark sa skladá z niekoľkých úzko spätých komponent. Ich hierarchia je zobrazená na obrázku 2.10. Úzka závislosť jednotlivých súčastí Sparku má niekoľko výhod. Komponenty na vyšších vrstvách prosperujú zo všetkých zmien nižšej vrstvy. Optimalizácia jadra Sparku teda okamžite ovplyvní rýchlosť všetkých ďalších súčasti. Ďalšou výhodou je to, že v jednej aplikácii je možné využiť celý obsah Spark balíčku.



Obr. 2.10: Architektúra Spark systému.

Spark Core obsahuje všetko potrebné pre správnu funkčnosť systému. Zahrňuje rozvrhovanie úloh, správu pamäti, komunikáciu s úložiskom, obnovy po chybách a mnoho ďalšieho. V *Core* je tiež definovaná *resilient distributed dataset (RDD)* štruktúra, ktorá je hlavným stavebným blokom Spark výpočtu.

Spark SQL umožňuje spracovávať štrukturované dáta pomocou SQL a HQL[5] jazyka. Jeho súčasťou je *API*[6] *DataFrame*, ktoré programátori využívajú pre spúšťanie ich vytvorených dopytov nad buď externými dátami alebo Sparkovými integrovanými kolekciami.

Veľa aplikácií potrebuje k svojmu behu spracovávať a analyzovať prúdy nových dát v reálnom čase. Pre tieto potreby obsahuje Spark *Spark Streaming*.

Strojové učenie v Sparku prebieha pomocou *MLib* komponenty. *MLib* obsahuje mnoho algoritmov využívaných v strojovom učení ako napríklad klasifikácia, regresia, zhlukovanie a ďalšie potrebné funkcionality. Všetky súčasti *MLib* sú prispôsobené distribuovanému prostrediu.

Posledným vysoko úrovňový komponent Spark systému je *GraphX*. Ten umožňuje vytvárať a spravovať grafy a vykonávať nad nimi rôzne grafové operácie.

Spark je navrhnutý tak, aby ho bolo možné jednoducho rozšíriť z jedného počítača na cluster o veľkosti niekoľko tisíc počítačov. Aby toho mohol dosiahnuť s maximálnou

[5] *Hive Query Language* je jazyk Apache Hive projektu pre jednoduché dopytovanie a analyzovanie rozsiahlych dát.

[6] *Application Programming Interface* je zbierka funkcií, metód, tried či protokolov softvérovej knižnice, ktoré môže programátor vo svojich programoch používať.

jednoduchosťou, Spark môže fungovať nad viacerými typmi clusterov. Najjednoduchší cluster, ktorý umožňuje beh Sparku sa nazýva *Standalone Scheduler*. Je súčasťou štandardného Spark frameworku a umožňuje ho rýchlo rozšíriť na kolekciu strojov, na ktorých ešte nebeží žiaden iný systém. Ak je skupina počítačov už prepojených nejakým systémom, Spark na nich môže fungovať, ak sa jedná o Hadoop YARN alebo Mesos cluster. Iný typ clusteru Spark zatial nepodoporuje.

Úložisko pre Spark dáta môže tvoriť ľubovoľný distribuovaný súborový systém, ktorý podporuje Hadoop API. Takéto úložisko je napríklad HDFS, Amazon S3, HBase a ďalšie.

Z predchádzajúceho textu vyplýva, že Spark k svojmu behu nepotrebuje Hadoop. Najčastejšie však Spark funguje nad HDFS úložiskom a Hadoop YARN clusterom.[12]

2.3.1 Resilient distributed Dataset RDD

Resilient distributed DataSet (RDD) je základná dátová jednotka Sparku. Je to distribuovaná kolekcia elementov, ktorá po vytvorení už nemôže byť menená. Všetká funkcionalita, ktorú Spark ponúka je buď vytváranie nových RDD alebo tranformácia existujúcich RDD, alebo vykonávanie operácií nad RDD. Každá kolekcia RDD je rozdelená na niekoľko časti, ktoré môžu byť spracovávané na rozdielnych uzloch clusteru.

RDD kolekcia môže byť vytvorená dvoma spôsobmi. Prvý spôsob je načítanie dát súboru z distribuovaného súborového systému a druhým spôsobom je premena natívnej lokálnej kolekcie (mapy alebo setu) na distribuovanú RDD kolekciu. Nevýhodou premeny lokálnej kolekcie na kolekciu distribuovanú je to, že nemôže presiahnuť veľkosť lokálnej pamäte. Pri čítaní distribuovaného súboru využíva Spark pamäť všetkých ulov clusteru. Znamená to, že Spark je schopný načítať súbor presahujúci veľkosť pamäti jedného počítača. Ak ani súčet všetkých pamätí systému nie je postačujúci na uloženie celej kolekcie, Spark využíva distribuované úložisko pre dočasné odloženie časti dát.

Spark podporuje dva typy operácií nad RDD dátami: transformácie a akcie. Transformácie sú operácie nad RDD kolekciami, ktorých výsledkom je vždy nová, upravená kolekcia. Sú to operácie ako `map()` a `filter()`. Akcie získavajú z RDD kolekcií konkrétne hodnoty, prípadne kolekcie zapisujú do úložiska. Akciou je napríklad funkcia `count()`, ktorá spočíta veľkosť RDD alebo funkcia `first()`, ktorá vráti prvý element kolekcie. Spark umožňuje užívateľom definovať aj vlastné operácie. Po implementovaní vlastnej transformácie je možné ju vykonať nad RDD pomocou `map()` funkcie. Funkcia `reduce()` slúži na spúšťanie užívateľom implementovaných akcií.[12]

Kapitola 3

Grafové databázy

V súčasnej dobe sú relačné databázy stále najpoužívanejšia databázová technológia. Sú vhodné pre väčšinu moderných aplikácií, ktoré potrebujú úložisko pre ich dáta. Relačný databázový systém je vo všeobecnosti veľmi efektívny, pokiaľ nezačne obsahovať príliš veľa vzťahov medzi tabuľkami. Ak sa v SQL dopytoch začne objavovať väčšie množstvo spojení nad rozsiahlymi tabuľkami, výkonnosť dopytov sa výrazne zníži. Pre programátorov to navyše znamená čoraz obtiažnejšiu tvorbu databázových dopytov. Momentálne začína narastať dopyt pre databázové úložiska, ktoré nie sú založené na klasickom relačnom modeli. Takéto úložiska sa dnes nazývajú NoSQL databázy. Jedným z vhodných typov NoSQL úložiska pre náhradu relačnej databázy s veľkým množstvom vzťahov je grafová databáza.[18]

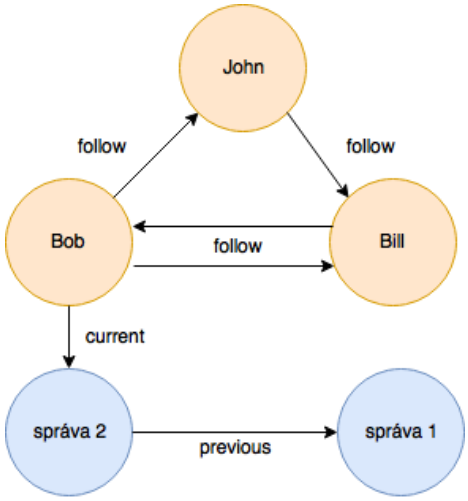
3.1 Grafový databázový model

Graf by bolo možné popísať ako kolekciu ulov a vzťahov medzi nimi. Graf reprezentuje databázové entity ako uzly a spôsob akým sú tieto entity prepojené medzi sebou ako vzťahy. Spôsob, akým sú dáta v grafe uložené, závisí na grafovom modeli. Grafové databázy využívajú pod povrchom viacero možných grafových modelov. Najčastejšie používané sú však modely Property graf a RDF.

3.1.1 Property graf

Autori v knihe [14] uviedli zaujímavý príklad, ako možno využiť graf pre reprezentáciu dát z reálneho sveta. Na ukážku použili vzťahy dát v sociálnej sieti Twitter (obr. 3.1). Entity Twitteru ako užívateľov a správy predstavujú uzly grafu. To, že užívateľ začal nasledovať iného užívateľa je možné zaznačiť do grafu pomocou vzťahu s názvom `follow`. Nové správy, ktoré Bob zverejnil sa v grafe dajú jednoducho zoradiť podľa času zobrazenia. K najnovšej správe sa dá dostať vždy priamo z uzla užívateľa, cez väzbu `current`. Pre históriu správ sa využije vzťah `previous`. Tento príklad poukazuje nato, že je omnoho jednoduchšie a čitateľnejšie zobrazovať dáta a ich vzťahy pomocou grafov. Predstavme si, že by sme mali zobraziť rovnaký príklad pomocou tabuliek relačného modelu. Informáciu o tom, ktorý užívateľ nasleduje ktorého užívateľa by bolo potrebné vyhľadať cez referencie. Pre nájdenie poslednej správy by sme museli porovnávať časy zobrazenia jednotlivých správ.

Predchádzajúca ukážka využívala pre zobrazenia dát takzvaný property graf. Je to typ grafového modelu, ktorý najčastejšie využívajú práve grafové databázy. Jeho charakteristikou je to, že obsahuje uzly a vzťahy, ktoré môžu obsahovať viaceré vlastnosti vo formáte (kľúč, hodnota). Vzťahy v takomto grafe musia byť navyše orientované a vždy mať na oboch



Obr. 3.1: Ukážka zobrazenia Twitter dát pomocou grafu.

koncoch nejaký uzol. Vzťahy môžu byť pomenované. Property grafy využíva aj veľmi známa Neo4j grafová databáza.

3.1.2 RDF

Ďalším často využívaným typom grafového databázového modelu je *Resource Description Framework* RDF. RDF je W3C[1] štandard pre prácu s dátami Sémantického Webu[2]. Na reprezentáciu dát využíva grafy. Základnou štruktúrou tohto modelu je trojica subjekt – predikát – objekt.

Subjekt je uzol grafu, z ktorého vychádza hrana a o ktorom je celé tvrdenie. Predikát reprezentuje hranu (vzťah) a objekt je konečný zdroj tvrdenia. Môže to byť uzol alebo literál. Ako ukážku môžme použiť tvrdenie `'Autorom skladby Bohemian Rhapsody je Freddie Mercury'`. V tomto tvrdení je subjektom skladba `'Bohemian Rhapsody'`, predikátom je `'autorom je'` a objekt reprezentuje `'Freddie Mercury'`.

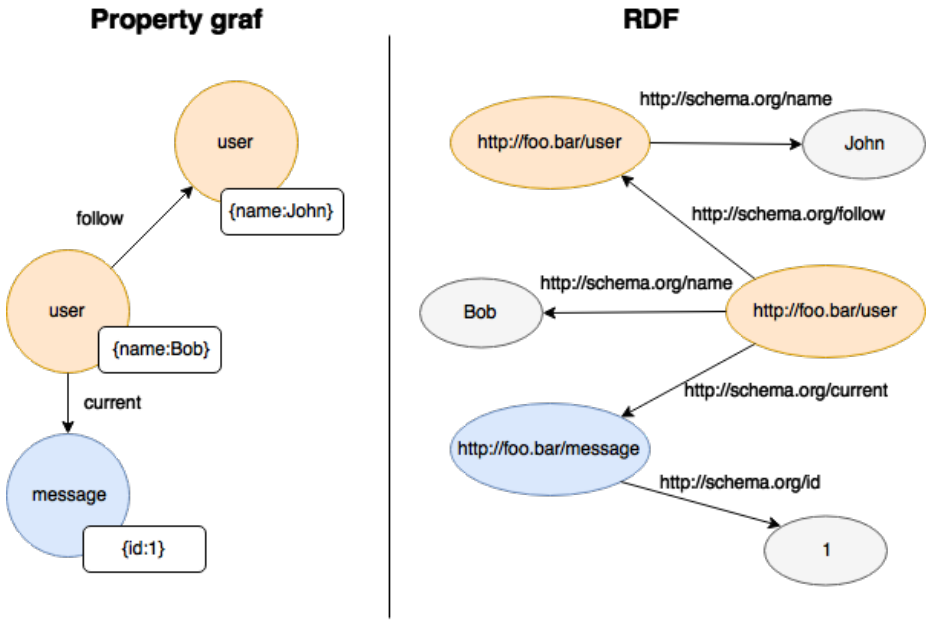
V RDF je každý uzol, hrana alebo literát identifikovaný pomocou URI identifikátoru. Prefix URI má formu webovej adresy a slúži pre spájanie objektov do kontextov. Ak dáta zdieľajú spoločný prefix, znamená to, že sa týkajú jedného spoločného zdroja. Napríklad všetky dáta s prefixom `http://dbpedia.org/` sú dáta z dátového setu DBPedia.

Hlavným rozdielom RDF modelu oproti Property grafu je to, že v RDF uzly ani hrany nemôžu obsahovať žiadne parametre. Každá vlastnosť uzlu musí byť teda definovaná pomocou ďalšej trojice s literálom, ktorý predstavuje samotnú hodnotu. V praxi to znamená, že totožné dáta budú v RDF modeli vyobrazené pomocou omnoho väčšieho grafu. Porovnanie veľkostí je možné vidieť na obrázku 3.2.

RDF dáta môžu byť serializované do rôznych formátov. Jednotlivé formáty sa líšia v spôsobe zápisu RDF trojíc. Medzi najznámejšie patria formáty RDF/XML, Turtle, N-Triples a ďalšie. Porovnanie formátu RDF/XML a Turtle je možné vidieť na obrázku 3.3.[9]

[1] W3C je medzinárodné konzorcium, ktorého členovia spolu s verejnou komunitou vyvíjajú webové štandardy pre World Wide Web.

[2] Semantický web je rozšírením klasického webu, ktorý sa zameriava na definovanie dát a ich vzťahov, tak aby boli pre stroje jednoducho zrozumiteľné.



Obr. 3.2: Porovnanie grafového databázového modelu Property graf a RDF.

3.2 Grafové databázy

Grafová databáza je real-time systém pre správu dát, ktorá umožňuje pracovať s grafovým dátovým modelom pomocou CRUD operácií (Create, Read, Update a Delete). Vo všeobecnosti je navrhnutá pre prácu s transakčnými systémami OLTP.

Jednotlivé implementácie grafových databáz je možné rozdeliť podľa typu ich úložiska a výpočetného nástroju.

Niektoré grafové databázy používajú natívne grafové úložisko, ktoré je optimalizované pre ukladanie a manipulovanie s grafovými dátami. Sú však grafové databázy, ktoré grafové dáta serializujú a ukladajú do iného typu úložiska, napríklad do relačnej databázy.

Výpočetné nástroje grafových databáz by sa dali rozdeliť podľa ich schopnosti vytvárať nesusedské indexy. Natívne grafové databázy obsahujú prirodzene indexy medzi uzlami. Vďaka tomu je možné prechádzať celý grafový priestor. Ak však databáza umožňuje vytvárať aj iné indexy, jej prehľadávanie sa razom niekoľko-násobne zefektívni.[14]

3.2.1 Neo4j

Jednou z implementácií grafových databáz je Neo4j. Podľa jej webovej stránky [8] je to dnes najpoužívanejšia grafová databáza. Disponuje všetkými vlastnosťami potrebnými k tomu, aby bola využiteľná v reálnych produkčných aplikáciach. Podporuje ACID transakcie, tvorbu clusterov, obnovovanie z chýb, vysokú dostupnosť a služby pre monitorovanie. Pokročilejšie služby sú dostupné len v platenej enterprise edícii. Neo4j funguje vo virtuálnom stroji Javy. Vďaka tomu je spustiteľný na ľubovoľnom stroji, ktorý podporuje spúšťanie Java aplikácií.

RDF/XML

```
<?xml version="1.0"?>

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:ex="http://www.example.com/">

<rdf:Description rdf:about="http://www.data.com/songs#Bohemian-Rhapsody">
  <ex:author>Freddie Mercury</ex:author>
</rdf:Description>

</rdf:RDF>
```

Turtle

```
@prefix ex: <http://www.example.com/> .

<http://www.data.com/songs#Bohemian-Rhapsody>
   ex:author "Freddie Mercury".
```

Obr. 3.3: Porovnanie RDF formátov RDF/XML a Turtle.

Dátový model

Neo4j využíva pre ukladanie dát *property graf* model. Znamená to, že v Neo4j môže každý uzol alebo vzťah obsahovať viacero atribútov. Tieto atribúty sú uložené priamo pri objekte, ktorého sa týkajú a nevytvárajú ďalšie uzly grafu.

Uzly v Neo4j neobsahujú žiaden typ. Aby bolo možné uzly medzi sebou rozlišovať, môže každý uzol obsahovať jeden, prípadne viacero štítkov (napríklad pridaním štítku `song` je možné definovať množinu uzlov, ktoré reprezentujú pesničky). Užívateľ vie vďaka štítkom špecifikovať databázový dopyt len na určitý typ uzlov.[19]

Cypher

Pre prácu s dátami, implementoval Neo4j svoj vlastný dopytovací jazyk, nazývaný Cypher. Snahou bolo vytvoriť jazyk, ktorý sa jednoducho používa a je ľahké ho čítať. Cypher je declarativný jazyk navrhnutý na základe SQL, ktorý umožňuje grafové dáta z Neo4j získavať, zapisovať, aktualizovať alebo odstraňovať. Keďže Neo4j pracuje s Property graf modelom so štítkami, Cypher obsahuje značky pre všetky jeho prvky:

`()` označuje uzly
`{}` označuje atribúty
`[]` označuje vzťahy
`:` označuje štítky

Dopyt, ktorý získa z databázy mená všetkých autorov pesničky Bohemian Rhapsody by vyzeral nasledovne:

```
MATCH (author:PERSON)-[:AUTHOR]->(:SONG {name: "Bohemian Rhapsody"})
RETURN author.name
```

V tomto dopyte `(author:PERSON)` značí všetky uzly grafu so štítkom `PERSON`. Slovo `author` je alias, popisujúci všetky tieto uzly, ktorý platí len v rámci tohto dopytu. Ďalšia časť dopytu `-[:AUTHOR]->` označuje vzťah medzi dvoma uzlami, ktorý má štítok `AUTHOR`. Posledná časť `(:SONG {name: "Bohemian Rhapsody"})` definuje všetky uzly so štítkom `SONG`, ktorých názov je `Bohemian Rhapsody`.[15]

Index

Moderné databázy sa dnes bez možnosti tvoriť indexy nezaobýdu. Vďaka indexu je možné zložitejšie dopyty niekoľkonásobne zefektívniť. Neo4j si je toho vedomí a disponuje plnou podporou pre vytváranie vlastných indexov. Index je však možné vytvoriť len na atribút uzla so špecifickým štítkom. Napríklad `CREATE INDEX ON :Person(name)` vytvorí index uzlu so štítkom `PERSON` na jeho atribút `name`.[15]

Práca s Neo4j

Samotná databáza Neo4j funguje ako webová aplikácia a je prístupná cez REST Api. Vďaka tomuto prístupu je možné sa k nej jednoducho pripojiť z rôznych klientov, prípadne si implementovať svoju vlastnú aplikáciu. Neo4j distribúcia obsahuje pre prácu s databázou Java Api a webovú aplikáciu. Java Api slúži programátorom pre vytváranie vlastných aplikácií alebo integráciu dát z Neo4j do svojich už existujúcich systémov. Webová aplikácia Neo4j poskytuje jednoduché, no robustné grafické užívateľské rozhranie pre prístup k databázovému serveru. Zaujímavou funkcionalitou je interaktívne, grafické zobrazovanie dopytovaných dát. Java Api aj webová aplikácia umožňujú vytvárať databázové dopyty pomocou Cypher jazyka.[8]

Podpora RDF dát

Práca s grafovými dátami uložených v súboroch rôznych RDF formátov nie je v Neo4j podporovaná. Jediný formát súboru, ktorý Neo4j umožňuje do databázy priamo vložiť je formát CSV. Pri exportovaní Neo4j podporuje len formáty JSON a CSV.[15]

3.2.2 Eclipse RDF4J

Eclipse RDF4J je framework pre prácu s RDF dátami. Umožňuje RDF dáta čítať, zapisovať, filtrovať, permanentne ukladať a vytvárať nad nimi rôzne databázové dopyty.

RDF úložiska

Výhodou RDF4J je to, že je nezávislý nad grafovým úložiskom. Podporuje rôzne druhy úložísk, ktoré implementujú RDF4J Api. Samotný RDF4J obsahuje dva typy lokálnych úložísk, no umožňuje sa pripojiť aj na vzdialené databázy a využívať ich úložisko. Pri vytváraní nového úložiska RDF4J poskytuje následovné možnosti:

in-memory store je vstavané, natívne grafové úložisko, ktoré ukladá dáta v hlavnej pamäti. Jeho nevýhodou je obmedzená veľkosť dát, keďže disponuje len menšou operačnou pamäťou. Je jednoduché ho nakonfigurovať, takže je vhodný pre prácu s jednoduchými dátovými setmi alebo na testovacie aplikácie. Pri vytváraní tohto úložiska je možné nastaviť jeho ukladanie na disk, ktoré sa deje pred vypnutím systému. *In-memory store* navyše disponuje zálohovacím mechanizmom, ktorý priebežne ukladá dáta z operačnej pamäte na disk, vďaka čomu ich je možné po páde systému obnoviť.

native store je taktiež vstavané, natívne grafové úložisko. Na rozdiel od *in-memory store* ukladá a zapisuje všetky dáta priamo na disk. Jeho výhodou je to, že má k dispozícii väčší úložný priestor a nie je limitovaný operačnou pamäťou. Samozrejme je kvôli práci s diskom pomalší ako *in-memory store*, no je vhodný pre väčšie dátové sety. *Native store* podporuje vytváranie užívateľských indexov pre efektívnejšie dopyty po dátach.

remote RDF store je alternatívou k predchádzajúcim dvom úložiskám. Umožňuje pripojiť sa k inej grafovej databáze a využívať jej dátové úložisko. Podmienkou pre vzdialenú databázu je to, aby implementovala RDF4J, prípadne Sesame[3] Api. Momentálne je *Ontotext GraphDB*^{TM} jediná grafová databáza, ktorá implementuje RDF4J Api, no Sezame Api implementuje niekoľko rôznych databáz (napríklad Stardog, Blazegraph a ďalšie).

Po vytvorení je úložisko obalené takzvaným repozitárom, ktorý tvorí jeho identifikáciu. Repozitáre je možné prehľadávať a opätovne sa k nim pripájať. Tvoria akýsi prístupový bod k už vytvoreným úložiskám.[5]

Práca s RDF4J

RDF4J framework poskytuje užívateľovi viacero možností, ako s ním pracovať. Užívateľ môže k RDF dátam pristupovať cez webové rozhranie, konzolovú aplikáciu alebo pomocou Java Api. Každý z týchto prístupov pracuje nad samostatným *in-memory* alebo *native* úložiskom. Znamená to, že dáta uložené cez Java Api užívateľ neuvidí ani vo webovej, ani v konzolovej aplikácii. Aby všetky tri prístupy mohli pracovať s rovnakým, zdielaným úložiskom, je v nich potrebné využiť vyššie spomínaný *remote RDF store*.

Webové rozhranie RDF4J frameworku sa skladá z dvoch webových aplikácií. *Server* je aplikácia pre správu databázy. Poskytuje HTTP prístup k svojim RDF úložiskám a umožňuje nad nimi vykonávať SPARQL[4] dopyty. RDF4J *Server* bol navrhnutý tak, že k nemu budú pristupovať len špecializované aplikácie. Znamená to, že okrem výpisu logov neobsahuje žiadne užívateľské rozhranie. Druhou samostatnou webovou aplikáciou je RDF4J Workbench. Jeho úlohou je vytvárať grafické užívateľské rozhranie, ktoré užívateľovi umožní jednoducho pristupovať k dátam uloženým na *Serveri*. Uživateľ vie pomocou Workbenchu spravovať RDF úložiska, zobrazovať ich dáta a vykonávať nad nimi SPARQL dopyty.

Ďalším prístupom k RDF4J je prístup cez konzolovú aplikáciu *Console*. Tá obsahuje rovnakú funkcionalitu pre prácu s RDF úložiskami a dátami ako aplikácia *Server*, no poskytuje priamy užívateľský prístup. *Console* môže pracovať nad vlastným *in-memory* alebo *native* úložiskom a navyše sa vie pripojiť na bežiaci RDF4J *Server*. Vtedy je možné pracovať s rovnakými dátovými setmi pomocou konzolovej aj webovej aplikácie.

Posledným spôsobom práce s RDF4J frameworkom je pomocou Java Api. Knižnica poskytuje všetky potrebné nástroje pre prácu s RDF dátami. Spravovať dáta môže užívateľ pomocou klasických Java metód alebo môže využívať SPARQL dopyty. Rovnako ako konzolová aplikácia aj RDF4J Java Api disponuje samostatným *in-memeory* alebo *native* úložiskom a tiež možnosťou napojiť sa na vzdialený server.[5]

Formáty dát

RDF4J poskytuje nástroje pre zapisovanie a čítanie RDF dát v rôznych formátoch. Podporuje formáty RDF/XML, Turtle, N-Triples a ďalšie.[5]

3.3 Distribuované grafové databázy

Pri ukladaní stále väčšieho množstva grafových dát postupom času úložný priestor jedného počítača prestane byť dostatočný. Existujú dve riešenia tejto situácie. Jedným z riešení je

[3]Sesame je open–source framework pre dopytovanie a analyzovanie RDF dát.
[4]SPARQL je dopytovací jazyk nad dátami uloženými vo formáte RDF.

zväčšiť kapacitu disku. Toto riešenie je dostatočné, no po určitom čase dôjdeme k stavu, keď dosiahneme maximálnu možnú kapacitu. Druhým, omnoho efektívnejším riešením je dáta distribuovať medzi viacero počítačov. Týmto spôsobom dosiahneme takmer neobmedzeného limitu úložného priestoru, toleranciu voči výpadku strojov a ďalšie výhody. V ďalších podkapitolách si predstavíme niekoľko možných riešení, ktoré umožňujú grafové dáta distribuovať.

3.3.1 JanusGraph

Janus Graph je *open–source* škálovateľná grafová databáza, ktorá je prispôsobená pre prácu nad obrovskými dátovými setmi obsahujúcimi miliardy uzlov a hrán. Keďže uložiť takéto rozsiahle dátové sety na jednom počítači a vykonávať nad nimi operácie nie je efektívne, JanusGraph umožňuje grafové dáta distribuovať na viac uzlový cluster. Rovnako ako bežné grafové databázy podporuje konkurenčné transakcie a tvorbu vlastných indexov.

Pôvodom JanusGraph databázy bol projekt Titan[5]. Vývoj Titanu však v roku 2015 skončil, jeho zdrojový kód sa presunul do projektu JanusGraph a dnes sa všetky nové funkcionality a opravy chýb implementujú pod JanusGraph projektom. Na projekte sa stále pracuje, no zatiaľ je len v alfa verzii.

JanusGraph umožňuje svoje dáta ukladať do dvoch rôznych distribuovaných databáz. Disponuje podporou pre Apache Cassandra, Apache HBase a Oracle Berkeley DB. Databáza Oracle Berkeley ako jediná nie je distribuovaná a slúži len na testovacie účely. Vďaka týmto úložiskám vie jednoducho zaručiť distribuovanie, toleranciu voči chybám a konzistenciu dát.

Pre prácu s dátami využíva JanusGraph jazyk Gremlin. Je to dopytovací jazyk nad grafovými dátami, ktorý vznikol pod Apache TinkerPop[6] projektom.

Užívateľ môže s JanusGraph komunikovať buď priamo, volaním jeho vnútorných API, alebo s použitím JanusGraph serveru, ktorý podporuje Gremlin dopyty.[7]

3.3.2 Halyard

Halyard je open–source úložisko pre RDF dáta. Je navrhnutý pre prácu s rozsiahlymi grafmi, ktoré môžu presahovať veľkosť úložiska jedného počítača. Pre dopytovanie sa po dátach využíva dopytovací jazyk SPARQL. Implementovaný bol v jazyku Java pomocou Eclipse RDF4J frameworku a Apache Spark databázy.

Úložisko RDF4J

Ako bolo už spomenuté, RDF4J disponuje viacerými typmi RDF úložísk. Konkrétne to boli typy: *in-memory store*, *native store* a *remote RDF store*. Halyard rozširuje RDF4J funkcionalitu a pridáva do neho ďalší typ vstavaného úložiska nazývaného *Halyard HBase Store*. Toto úložisko sa navonok chová ako ostatné typy, no fyzicky ukladá RDF dáta do HBase tabuliek. Pri použití distribúcie RDF4J rozšírenej o Halyard je možné nový typ úložiska vytvoriť priamo z konzoly alebo webovej aplikácia Workbench. Podpora tohto úložiska pre Java Api zatiaľ nie je implementovaná.[6]

[5]Stránky frameworku Titan: http://titan.thinkaurelius.com
[6]Stránky frameworku TinkerPop: http://tinkerpop.apache.org

row-key	<http://ex.com/songs#Bohemian-Rhapsody> <http://ex.com#author> "Freddie Mercury" .
0<hash(subjekt)><hash(predikát)><hash(objekt)>	empty
1<hash(predikát)><hash(objekt)><hash(subjekt)>	empty
2<hash(objekt)><hash(subjekt)><hash(predikát)>	empty

Obr. 3.4: Spôsob uloženia RDF trojíc v HBase tabuľke.

Formát HBase dát

Vzhľadom nato, že HBase tabuľky podporujú len dáta tvaru (kľúč, hodnota), Halyard musel prísť s vlastným riešením, ako mapovať RDF trojice na HBase dvojice. Halyrad používa HBase tabuľky netradičným spôsobom. Využíva ich schopnosť obsahovať milióny stĺpcov. Trojice RDF neukladá ako hodnoty buniek tabuľky, ale ako názvy jej stĺpcov. Každá trojica ďalej obsahuje tri riadky, kde *row-key* týchto riadkov tvoria kombinácie hašov subjektu, predikátu a objektu. Samotná bunka tabuľky obsahuje prázdnu hodnotu.

Príklad tohto formátu je možné vidieť na obrázku 3.4. Každý *row-key* začína číslom, ktoré definuje kombináciu hašu. Pri vyhľadávaní trojíc podľa subjektu a predikátu sa vrátia všetky názvy stĺpcov pre *row-key* začínajúce číslom 0, následujúcim hašom subjektu a predikátu. Podobne je možné vyhľadávať podľa iných kritérií.

Dôvodom takéhoto formátu bola obmedzená dĺžka *row-key* HBase tabuľky. RDF trojice môžu mať ľubovoľnú dĺžku a nemuseli by sa vždy zmestiť do tohto limitu. Vhodným riešením tohto problému bolo využiť haš trojice ako kľúč tabuľky, ktorý má dĺžku vždy rovnakú. Rôzne trojice sa s malou pravdepodobnosťou môžu zobraziť do rovnakej haši. Zapisovanie trojice do bunky tabuľky by mohlo prepísať inú uloženú hodnotu. Riešenie s ukladaním trojíc do názvu stĺpca priradí danej haši jednoducho dve rôzne hodnoty.[6]

RDF4J repozitár a HBase tabuľka

RDF4J repozitár predstavuje len akýsi prístupový bod k RDF úložisku. Jednotlivé repozitáre existujú vždy len v rámci jedného kontextu. Znamená to, že repozitár vytvorený v RDF4J konzole nie je možné vidieť cez webovú aplikáciu. Halyard ukladá RDF dáta do HBase tabuliek. Pri vytváraní repozitára s *Halyard HBase Store* je možné pre dáta vytvoriť novú HBase tabuľku alebo využiť už existujúcu. Možnosť pripojiť repozitár s existujúcou tabuľkou, umožňuje vytvoriť dva rozdielne repozitáre, ktoré však fungujú nad spoločnými RDF dátami.[6]

Práca s dátami

Halyard nástroj *Bulk Load* umožňuje nahrať rozsiahly HDFS súbor s RDF dátami priamo do HBase tabuľky. Pre nahrávanie veľmi rozsiahlych súborov obsahuje Halyard ďalší nástroj *Pre Split*, ktorého úlohou je zefektívniť vkladanie dát cez *Bulk Load*. Prácou *Pre Split* nástroja je odhadnúť rozdelenie vstupných dát na regióny a na základe tohto výpočtu vytvoriť prázdnu HBase tabuľku. Oba tieto nástroje využívajú Hadoop MapReduce pre čo najefektívnejší výpočet.

Práca s RDF dátami je v Halyard totožná ako v aplikácii RDF4J.[6]

3.3.3 HGraphDB

HGraphDB je *open–source* framework, ktorý umožňuje pracovať s HBase tabuľkou ako s grafovou databázou. Je implementáciou rozhrania Apache TinkerPop.

Apache TinkerPop

Apache TinkerPop je *open–source* framework pre prácu s grafovými dátami. Dáta ukladá pomocou jednoduchých dátových typov, ktoré nazýva vrcholy (vertices) a hrany (edges). Vrcholy reprezentujú uzly grafu a ukladajú samotné objekty. Hrany určujú vzťahy medzi týmito uzlami.

Pre prácu s dátami TinkerPop využíva svoj vlastný dopytovací jazyk Gremlin. Dáta môžu byť dopytované z rôznych programovacích jazykov (napríklad Java alebo Python), prípadne pomocou konzolovej aplikácie Gremlin Console.

Framework TinkerPop pozostáva z viacerých nezávislých komponent. Samotné grafové úložisko nie je súčasťou tohto frameworku. Výhodou takejto architektúry je, že nie je závislá na konkrétnej implementácii úložiska. Môže teda pracovať nad ľubovoľným dátovým úložiskom, ktoré implementuje TinkerPop Api. HGraphDB je jednou takouto implementáciou.[10]

Dátový model

Grafový mode, ktorý HGraphDB využíva, je *property graf*. K uzlom a vzťahom umožňuje pridávať navyše štítky, ktoré reprezentujú typ objektu. Oproti štítkom v Neo4j, HGraphDB vie ku každému uzlu alebo vzťahu priradiť len jeden konkrétny štítok. Naproti tomu, počet parametrov nie je nijako obmedzený.

Každý uzol aj vzťah grafu musí byť v HGraphDB jedinečne identifikovaný pomocou identifikátoru. Zaujímavou vlastnosťou je, že identifikátor môže byť rôznych dátových typov.

Nasledujúca časť kódu ukazuje jednoduchý príklad vytvorenia dvoch uzlov, prepojených jedným vzťahom:

```
Vertex v1 = graph.addVertex(T.id, 1L, T.label, "person", "name", "John");
Vertex v2 = graph.addVertex(T.id, 2L, T.label, "person", "name", "Bob");
v1.addEdge("knows", v2, T.id, "edge1", "since", LocalDate.now());
```

Tento kód vytvorí uzol `v1` s identifikátorom `1` typu `long`. Uzol `v1` obsahuje štítok `person` a jeden parameter `name` s hodnotou `John`. Obdobne sa vytvorí uzol `v2`. Vzťah medzi týmito uzlami sa definuje pomocou metódy `addEdge()`. Jeho štítok je `knows`, identifikátor má hodnotu `edge1` a obsahuje jeden parameter `since` obsahujúci dnešný dátum.[4]

Práca s dátami

Ako bolo vyššie spomenuté, HGraphDB tvorí len grafové úložisko pre TinkerPop framework. Znamená to, že je k nemu možné pristupovať pomocou viacerých programovacích jazykov (napríklad Java, Python, Scala, JavaScript a ďalšie). K práci s dátami je možné využiť taktiež konzolovú aplikáciu Gremlin Console. Dopytovací jazyk používaný v HGraphDB je jazyk Gremlin.[4]

Row Key	Column: label	Column: createdAt	Column: [property1 key]	Column: [property2 key]	...
[vertex ID]	[label value]	[createdAt value]	[property1 value]	[property2 value]	...

Obr. 3.5: Štruktúra HGraphDB tabuľky pre ukladanie uzlov. Prevzaté z [4].

Row Key	Column: label	Column: fromVertex	Column: toVertex	Column: createdAt	Column: [property1 key]	Column: [property2 key]	...
[edge ID]	[label value]	[fromVertex ID]	[toVertex ID]	[createdAt value]	[property1 value]	[property2 value]	...

Obr. 3.6: Štruktúra HGraphDB tabuľky pre ukladanie hrán. Prevzaté z [4].

Štruktúra HBase tabuľky

HGraphDB ukladá dáta v HBase tabuľke do výšky. Znamená to, že každý uzol a vzťah grafu sú uložené do nového riadku. Parametre objektov sa ukladajú ako nové stĺpce tabuľky. Pre uloženia dát jedného grafu využíva HGraphDB päť HBase tabuliek.

Prvá tabuľka slúži pre uloženie všetkých uzlov grafu. Jej *row-key* je identifikátor uzlu grafu. Ďalej obsahuje stĺpec *label*, ktorého hodnota je názov štítku uzlu a stĺpec *createdAt*, ktorý ukladá čas vytvorenia uzlu. Všetky ďalšie stĺpce reprezentujú parametre uzlu. Jej štruktúru je možné vidieť na obrázku 3.5.

Druhá tabuľka slúži pre ukladanie vzťahov medzi uzlami. Obsahuje statické stĺpce *row-key* (identifikátor hrany), *label* (názov štítku hrany), *fromVertex* (identifikátor uzlu, z ktorého hrana vychádza), *toVertex* (identifikátor uzlu, kam hrana vstupuje), *createdAt* (časová známka vytvorenia hrany). Ďalšie stĺpce slúžia pre uloženie parametrov hrany. Štruktúru tejto tabuľky je možné vidieť na obrázku 3.6.

Ďalšie dve tabuľky slúžia pre uloženie indexov hrán a uzlov. Posledná tabuľka obsahuje metadáta o indexoch a definuje, či je daný index aktívny alebo nie.

Ak sa v HGraphDB zapne manažment schématu, vytvoria sa ešte dve tabuľky. Jedna ukladá metadáta o štítkoch a druhá obsahuje vzťahy medzi jednotlivými štítkami.

Kapitola 4

Návrh riešenia

Cieľom tejto práce je analyzovanie možností práce s grafovými databázami v distribuovanom prostredí. Výsledná aplikácia by sa mala zamerať najmä na možnosti vkladania dát, dopytovanie sa nad nimi a na výkonnosť týchto operácií. Zo spomínaných grafových databáz som sa rozhodol zamerať sa na frameworky Halyard a HGraphDB.

4.1 Implementačný jazyk

Programovací jazyk pre prácu s frameworkami bude Java. Zvolil som si ju, pretože každý spomínaný framework disponuje jej rozhraním. Navyše je Java platformovo nezávislá a vysoko rozšírená. Vývoj vďaka tomu môže prebiehať na inom stroji ako bude výsledná aplikácia testovaná.

Dopytovacie jazyky nad dátami budú závisieť na použitom frameworku. Framework Halyard umožňuje získavať dáta pomocou SPARQL jazyka. Na rozdiel od toho, HGraphDB disponuje jazykom Gremlin, ktorý má dokonca svoje vlastné Java API.

4.2 Distribuované prostredie

Základnou časťou návrhu distribuovanej aplikácie je prostredie, v ktorom bude spúšťaná. V dnešnej dobe je vo väčšine prípadov zbytočné prichádzať s vlastným riešením distribuovania programu. Existuje niekoľko open-source riešení, ktoré riešia tento problém a odbremeňujú tak programátora.

4.2.1 Apache Hadoop

Ako základný stavebný kameň distribuovaného prostredia som sa rozhodol zvoliť framework Apache Hadoop. Jeho výhodou je, že je jednoducho konfigurovateľný a má veľkú komunitu, vďaka čomu existuje veľké množstvo návodov k jeho používaniu. Najväčšou výhodou Hadoopu je jeho jednoduchá škálovateľnosť. Hadoop aplikáciu je možné testovať na clusteri o veľkosti jedného počítača v domácom prostredí a potom jednoducho spustiť na niekoľko strojovom clusteri. Výstup aplikácie bude vždy rovnaký, zmení sa len rýchlosť výpočtu.

Hadoop sa skladá z niekoľko súčastí. Pre túto prácu bude však relevantný len jeho súborový systém HDFS. Výpočtová súčasť MapReduce bude nahradená frameworkom Apache Spark. Vďaka HDFS bude možné navyše do prostredia pridať jednoducho databázu Apache HBase a framework Apache Spark.

4.2.2 Apache HBase

Cieľom tejto práce je analyzovať možnosti využitia grafových databázových frameworkov Halyard a HGraphDB v distribuovanom prostredí. Ako bolo spomínané, oba frameworky nedisponujú vlastným databázovým nástrojom. Ich úlohou je vytvárať abstrakciu nad distribuovanou databázou HBase a umožňujú s ňou pracovať, ako keby sa jednalo o grafovú databázu. V prípade Halyardu je možné pracovať s HBase ako s RDF úložiskom a HGrapDB umožňuje do HBase zapisovať grafové dáta vo formáte property graf.

Fakt, že oba frameworky pracujú s rovnakou databázou HBase prináša viaceré výhody. Jednou výhodou je, že pre beh aplikácie nepotrebujeme konfigurovať a spravovať dve rôzne prostredia. Ďalšou oveľa dôležitejšou výhodou je presnejšie meranie výkonnosti jednotlivých riešení. Frameworky Halyard aj HGraphDB budú totiž bežať v totožne nakonfigurovanom distribuovanom prostredí. Oba frameworky budú dokonca pracovať s rovnakou bežiacou inštanciou HBase databázy. Výsledná analýza nebude vďaka tomu závislá od prostredia, ale len od implementácie frameworkov Halyard a HGraphDB.

Vo väčšine distribuovaných aplikácií je potrebné vyhnúť sa efektu úzkeho hrdla. Tento efekt nastáva, ak je výkonnosť celého systému limitovaná len jedným komponentom. Pri perzistentných distribuovaných aplikáciach býva úzkym hrdlom práve databáza. Pri použití centralizovanej databázy bude celý systém limitovaný jej výkonnosťou a nie počtom strojov vykonávajúcich výpočet. Vďaka použitiu Apache HBase tento efekt kvôli databáze nenastane. HBase je rozložený na všetkých strojoch clusteru a klient získava/zapisuje dáta priamo na stroj, ktorý tieto dáta obsahuje. Pridaním ďalšieho počítača do clusteru získame ďalší prístupový bod do databázy.

4.2.3 Apache Spark

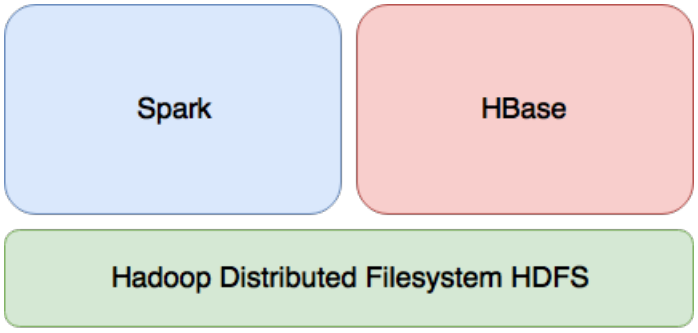
Poslednou časťou distribuovaného prostredia je nástroj, ktorý umožňuje tvoriť paralelné programy. S využitím Apache Hadoop ako základ prostredia, som mal na výber medzi frameworkami Hadoop MapReduce a Apache Spark. Keďže je Spark modernejší a jednoduchší na konfiguráciu, uprednostnil som ho pred vstavaným MapReduce. Výpočet výslednej aplikácie by mal prebiehať v pamäti. Spark na svojich stránkach tvrdí, že jeho programy bežiace v pamäti sú sto-krát efektívnejšie ako totožné programy implementované pomocou frameworku MapReduce. To bol ďalší dôvod, prečo bolo vhodnejšie využiť framework Spark.

Spark podporuje niekoľko rôznych typov clusterov. Spark aplikácia môže byť spúšťaná nad vstavaným Spark Standalone Scheduler, Apache YARN alebo Mesos clusterom. Výsledné distribuované prostredie bude obsahovať vstavaný Spark Standalone Scheduler. Súborový systém, ktorý bude Spark k svojmu chodu využívať bude HDFS.

Výsledné distribuované prostredie je možné vidieť na obrázku 4.1.

4.3 Návrh aplikácie

Výstupom tejto práce je analýza možností použitia grafových databáz v distribuovanom prostredí so zameraním sa na výkonnosť ich operácií. Ako bolo v predošlých kapitolách spomenuté, zameral som sa na dve riešenia distribuovaných grafových databáz. Je to framework Halyard a HGraphDB. Oba tieto frameowrky nie sú databázy v pravom slova zmysle, ale tvoria akési rozhranie nad distribuovanou databázou HBase a umožňujú s ňou



Obr. 4.1: Distribuované prostredie aplikácie.

pracovať ako s grafovou databázou. Halyard podporuje grafové dáta vo formáte RDF trojíc a framework HGraphDB umožňuje do HBase vkladať dáta vo formáte property graf.

4.3.1 Zdroje dát

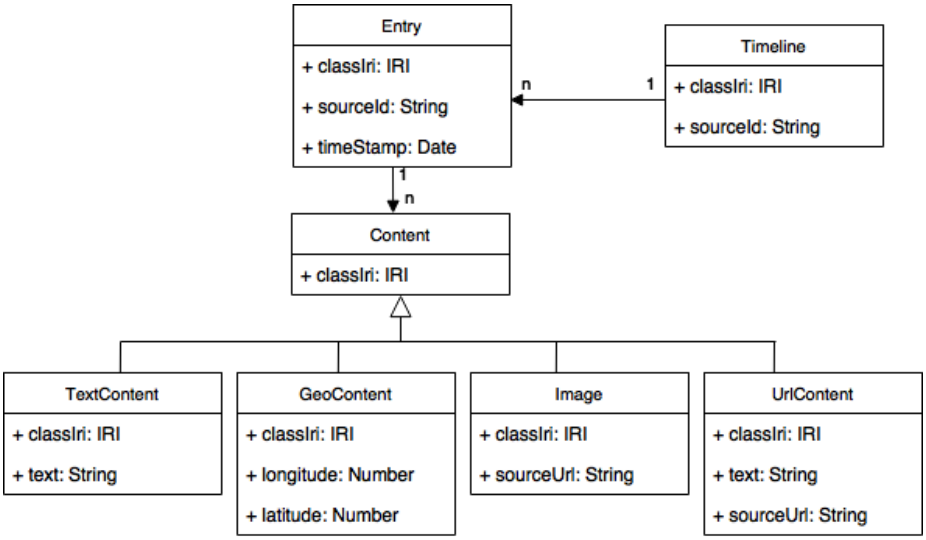
Pri návrhu výslednej aplikácie som sa zameral na využitie reálnych dát. V dnešnej dobe obsahujú najväčšie dátové sety databázy sociálnych sietí. Milióny užívateľov nahráva do týchto sietí stále nové a nové dáta či už sa jedná o rôzne statusy, obrázky alebo videá. Analýza týchto dát býva náročná, častokrát presahujúca možnosti jedného počítača. Výsledná aplikácia sa bude zameriavať práve na tieto dáta. Konkrétne bude získavať a analyzovať dáta z dvoch dnes najpoužívanejších sociálnych sietí Facebook a Twitter.

Twitter

Sociálna sieť Twitter obsahuje najväčšie množstvo dát vo formáte takzvaných Tweetov. Tweet je príspevok od užívateľa. Užívateľ môže takýto Tweet uverejniť na svojom účte, ktorý je jedinečne identifikovaný pomocou unikátneho textového reťazca. Množina všetkých Tweetov patriacich jednému Twitter účtu sa nazýva Timeline. Zo získaných Tweetov sa zameriam na ich textový obsah, obrázky, polohu a url odkazy. Url odkazy Tweetu je množina všetkých url odkazov použitých v jeho textovej správe. Väčšina Timelinov je na sociálnej sieti Twitter verejne dostupná, vďaka čomu sa aplikácie nebude musieť zameriavať len na nejakú malú podmnožinu všetkých existujúcich účtov.

Facebook

Na sociálnej sieti Facebook sú dáta podobné dátam na Twitteri. Dáta sú štruktúrované do formátu príspevkov. Príspevok predstavuje podobnú informáciu ako Tweet na sociálnej sieti Twitter. Každý príspevok patrí buď jednému Facebookovému účtu, alebo jednej Facebookovej stránke. Facebookový účet alebo stránka sú jedinečne identifikované identifikátorom vo formáte textového reťazca. Všetky príspevky patriace jednému účtu alebo stránke tvoria jeden Feed. Problémom na Facebooku je to, že väčšina Feedov nie je verejne dostupná. Vo výslednej aplikácii som sa preto zameral len na Facebookove stránky, ktoré majú svoje Feedy zverejnené. Rovnako ako u Twitteru, z príspevkov získam ich url odkazy, text, obrázky a polohu.



Obr. 4.2: Dátový model aplikácie.

4.3.2 Model dát

Dáta získané zo sociálnych sieti nemajú totožný formát. Vo výslednej aplikácii bude preto vhodné tieto dáta transformovať do normovaného formátu. Cieľom aplikácie je pracovať s grafovými dátami. Z tohto dôvodu budú získané dáta zo sociálnych sieti ukladané vo formáte grafových dát. Vzhľadom k už existujúcim riešeniam som zvolil formát ontológie RDF. Keďže sa jedná o jednoduchý formát, bude ho možné jednoducho previesť aj do property grafu, takže s ním bude môcť pracovať rovnako framework Halyard aj HGraphDB.

Model je možné vidieť na obrázku 4.2. Trieda *Timeline* obsahuje identifikáciu zdroja a všetky jeho stiahnuté príspevky. Zdroj môže byť Twitter účet alebo Facebookova stránka. Každý stiahnutý príspevok reprezentuje trieda *Entry*. Tá obsahuje identifikátor príspevku a zoznam obsahov, ktoré môžu nadobúdať rôzne typy (na obrázku trieda *Content*). Obsah môže byť textová správa príspevku (trieda *TextContent*), poloha pridania príspevku s jeho zemepisnou lokáciou (trieda *GeoContent*), obrázok príspevku definovaný jeho zdrojovou adresou (trieda *Image*) a nakoniec url odkaz príspevku obsahujúci url odkaz a jeho čitateľnú verziu (trieda *UrlContent*). Každá trieda obsahuje navyše svoju IRI identifikáciu, ktorá umožní v grafe rozoznať typ uzlu.

Tento model je prevzatý z už existujúceho frameworku Timeline Analyzer[1]. Jeho autorom je Burget Radek, Ing., Ph.D..

4.3.3 Klient pre sťahovanie dát

Podstatnú časť aplikácie bude tvoriť klient pre sťahovanie sociálnych dát. Jeho úlohou je pripojenie sa na sociálnu sieť, stiahnutie požadovaných príspevkov a ich transformácia do modelu aplikácie. Vstupom klienta je identifikátor a typ zdroja. Identifikátor je názov účtu alebo stránky a typ zdroja je Twitter alebo Facebook. Klient sa potom pripojí na požadované API a stiahne všetky dostupné príspevky.

[1]Stránky frameworku Timeline Analyzer: https://github.com/nesfit/timeline-analyzer

Twitter

Tweety jedného Twitter účtu sú dostupné na API ceste */statuses/user_timeline*. Typ dopytu pre získanie týchto dát je *GET*. Počet príspevkov, ktoré môžme pomocou jedného dopytu získať je obmedzený. Pre získanie všetkých príspevkov bude preto potrebné tento endpoint zavolať niekoľkokrát. Maximálne však môžme stiahnuť tritisíc dvesto najnovších príspevkov.[2]

Zvolenie účtu, ktorého príspevky chceme stiahnuť je možné pomocou url parametru *screen_name*. Parameter príma textový reťazec, ktorý jedinečne identifikuje názov účtu. Vstupom Twitter klienta bude práve tento identifikátor. Daný dopyt umožňuje získať dáta aj pomocou číselného identifikátoru účtu, no používanie textového identifikátoru bude pre užívateľa aplikácie pohodlnejšie.

Dáta, ktoré získame daným dopytom obsahujú všetky potrebné informácie na vyplnenie aplikačného modelu. Pre ich transformovanie na vnútorný dátový model bude preto potrebné len správne nastaviť jednotlivé hodnoty. V prípade obrázkov sa budú do databázy vkladať len ich zdrojové adresy. Celý obsah obrázku sa sťahovať nebude.

Limity Twitter API

Twitter nám umožňuje stiahnuť maximálne dvesto Tweetov v jednom dopyte. Počet dopytov, ktoré môžme poslať na jeden endpoint je taktiež limitovaný. Limit endpointu pre získanie Tweetov účtu je tisíc päťsto dopytov za pätnásť minút. Pre získanie tritisíc dvesto najnovších príspevkov jedného účtu je tento limit dostatočný. Výsledná aplikácie bude sťahovať dáta z niekoľkých účtov paralelne. V takomto prípade môžme stiahnuť maximálne tristotisíc Tweetov za pätnásť minút. Túto hodnotu som vypočítal následovne:

```
(200 Tweetov/dopyt) * (1 500 dopytov/15 min) = (300 000 Tweetov/15 min)
```

Keďže je tento limit fixný a nie je ho možné jednoducho dvihnúť, bude s ním musieť užívateľ pri používaní aplikácie počítať. Výsledná aplikácia musí taktiež tento limit očakávať a adekvátne sa mu prispôsobiť. Ideálnym riešením je odchytiť chybovú hlášku evokujúcu presiahnutie limitu a na nejaký čas prestať posielať ďalšie dopyty.

Facebook

Feed jednej Facebookovej stránky je dostupný na API ceste */page-id/feed*. Parameter *page-id* reprezentuje textový identifikátor Facebookovej stránky. Tento endpoint je rovnako ako v prípade Twitter API limitovaný. Maximálny počet príspevkov, ktoré môžme pomocou jedného dopytu stiahnuť je len sto príspevkov. Znamená to, že pomocou jedného dopytu získame len polovicu dát oproti využitiu Twitter API, čoho výsledkom bude väčšie množstvo poslaných dopytov pre získanie rovnakého objemu dát. Najviac času pri získavaní príspevkov zo sociálnych sietí zaberá práve naviazanie spojenia so vzdialeným serverom. Stiahnutie dát a ich spracovanie je už pomerne časovo nenáročná úloha. Vzhľadom k spomínanému limitu, bude získavanie dát zo sociálnej siete Facebook výrazne pomalšie.

Ďalším obmedzením API je množstvo prístupných príspevkov jednej stránky. Facebook umožňuje stiahnuť len šesťsto najnovších príspevkov v každom roku.[3] Ak nejaká stránka

[2] Limity Twitter API: https://developer.twitter.com/en/docs/basics/rate-limits

[3] Limity endpointu pre získanie Facebookových príspevkov: https://developers.facebook.com/docs/graph-api/reference/v2.12/page/feed

existuje len dva roky znamená to, že z nej aplikácia môže stiahnuť len tisíc dvesto príspevkov. Napriek tomu, že by sa pri tomto obmedzení zdalo, že takto je možné z jednej stránky stiahnuť menšie množstvo príspevkov ako v prípade Twitteru, nie je to tak. Mnoho známych Facebookových stránok existuje už viacero rokov a tak veľkosť jedného Feedu môže presiahnuť aj desať tisíc príspevkov. Navyše v niektorých prípadoch Facebook vráti aj väčšie množstvo príspevkov v jednom roku. Vzhľadom k tomuto faktu, bude vhodným riešením umožniť užívateľovi definovať dátum najstaršieho príspevku, ktorý bude stiahnutý. Užívateľ môže vďaka tomu vylúčiť príliš staré príspevky, ktoré by pre neho neboli už relevantné.

Dáta, ktoré chceme z príspevkov získať je potrebné pri volaní endpointu presne zadať. Ak nezadáme žiaden požadovaný parameter, z daného endpointu získame len číselný identifikátor príspevku. Aby bolo možné stiahnutý príspevok transformovať do modelu aplikácie, z Facebook API budem požadovať tieto dáta:

`created_time` čas vytvorenia príspevku
`message` textový obsah príspevku
`link` url odkazy v príspevku
`place` poloha pridania príspevku
`attachments` url odkazdy na obrázky obsiahnuté v príspevku

Obrázky budú ukládané rovnako ako v prípade Twitter klienta len pomocou ich odkazu.

Limity Facebook API

API poskytnuté Facebookom nemá presne definované limity pre získavanie príspevkov zo stránok. Odchytávať túto chybu je však rovnako potrebné ako pri implementácii Twitter klienta. Posielanie dopytov je po presiahnutí limitu potrebné opäť na nejaký čas obmedziť.

4.3.4 Databázový klient

Úlohou databázového klienta je perzistencia stiahnutých dát. Ako úložisko využije HBase databázu. Cieľom výslednej aplikácie je porovnávať grafové databázové frameworky, preto je potrebné stiahnuté dáta ukladať rôznymi spôsobmi. Keďže frameworky Halyard a HGraphDB nepracujú s HBase pomocou rovnakého dátového formátu, aplikácia bude musieť pracovať s dvomi rôznymi grafovými modelmi. Výhodou týchto frameworkov je to, že dokážu pracovať s rovnakou HBase databázou bez toho, aby si navzájom poškodzovali dáta. Každý framework totiž pracuje v samostatnej HBase tabuľke, čo mu umožňuje jeho dáta izolovať. HGraphDB navyše využíva iný formát pomenovania vzniknutých tabuliek, takže užívateľ nemusí voliť rozdielne názvy pre rôznych databázových klientov.

Halyard

Framework Halyard nemá vlastné Java API. Ako bolo v teórii spomenuté, tvorí len akési úložisko framewoku RDF4J. Zápis, úprava a čítanie dát teda prebieha cez centralizovaný RDF4J server. Ak by sme využili toto riešenie vo výslednej aplikácii, server by tvoril jej výrazné úzke hrdlo. Hoci by zápis dát prebiehal paralelne z viacerých počítačov, výkonnosť serveru by sa rozširovaním clusteru nijako nemenila. Postupným pridávaním nových uzlov by sme časom dosiahli hranicu, kedy by pridanie nového uzlu neprinieslo aplikácii žiaden výkonnostný zisk. Pri klientovi využívajúceho framework Halyard bude teda potrebné zvoliť iný prístup. Keďže je formát HBase tabuľky vygenerovanej frameworkom Halyard známy,

klient môže zapisovať dáta priamo do HBase databázy. Tento prístup je plne pararelný a s každým pridaným uzlom vznikne aj nový prístupový bod do databázy.

Získavanie dát z HBase pomocou frameworku Halyard bude prebiehať centralizovane. Využitie priameho prístupu do HBase databázy by mohlo priniesť isté výkonnostné vylepšenia, no takáto aplikácia by musela byť špecializovaná na jeden typ databázových dopytov. V prípade výslednej aplikácie bude snahou umožňovať využitie čo najviac možných databázových dopytov. Framework Halyard navyše disponuje vlastnou optimalizáciou využívajúcou distribuované prostredie. Práca s dátami bude preto prebiehať pomocou RDF4J Java API a dopytovacieho jazyka SPARQL.

HGaphDB

Práca s frameworkom HGraphDB bude v tomto prípade jednoduchšia. HGraphDB rozširuje známy framework TinkerPop, ktorý disponuje vlastným Java API. S využitím tohto frameworku prebieha prístup do HBase databázy priamo z jedného stroja. Vďaka tomu je možné vytvoriť súčasne niekoľko spojení a dáta zapisovať z viacerých strojov paralelne. V dokumentácii frameworku HGraphDB sa nespomína nič o maximálnych možných pripojeniach v rovnakom čase. Ak by bola aplikácia obmedzená takýmto limitom, pripojenie do databázy by bolo potrebné vykonať na každom fyzickom stroji iba raz a využívať ho pre všetky jeho vlákna. Potrebu tohto riešenia ukáže až testovanie.

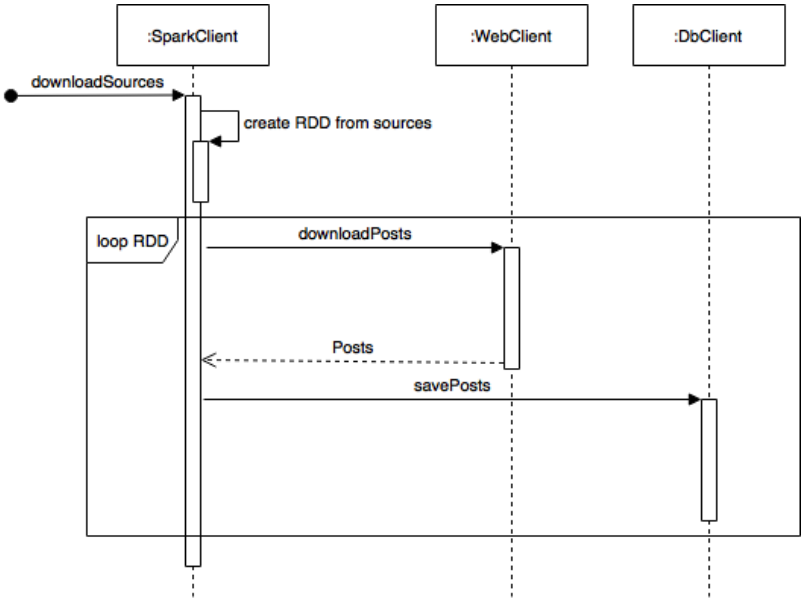
Zaujímavou vlastnosťou tohto frameworku je to, že každému vrcholu a hrane grafu je potrebné nastaviť unikátny identifikátor. Súčasťou klienta bude preto aj generovanie týchto identifikátorov. Keďže ukladanie dát prebieha paralelne, jednoduchá inkrementálna sekvencia nebude postačujúca. Pretože úlohou jedného behu klienta je uloženie príspevkov z jedného zdroja, ktorý je jedinečne identifikovaný textovým reťazcom, identifikátor grafových hrán a vrcholov obohatíme práve o túto hodnotu. Pri viacerých behoch aplikácie nad rovnakými zdrojmi by tento prístup spôsobil konflikt unikátnych kľúčov. K identifikátoru preto pridám ešte časovú známku, čo odlíši dva rovnaké behy aplikácie. Poslednou časťou identifikátoru bude ešte typ sociálnej siete. Táto časť nám zaručí to, že totožne nazvaná Facebook stránka nebude kolidovať s rovnako pomenovaným Twitter účtom. Identifikátor bude mať následovný tvar: <f/t>_<zdroj>_<časové razitko>_<poradie>. Napriek tomu sa týmto spôsobom budú totožné dáta v tabuľke duplikovať, kontrolovať výskyt každej stiahnutej informácie by zbytočne predlžovalo beh aplikácie. Výsledná aplikácia navyše k takémuto použitiu nie je určená.

Získavanie dát pomocou frameworku HGraphDB bude prebiehať centralizovane z totožného dôvodu ako v prípade frameworku Halyard. K dopytovaniu sa nad dátami využijem framework TinkerPop a jeho jazyk Gremlin. Tento jazyk umožňuje vytvárať databázové dotazy priamo v Java aplikácii.

4.3.5 Spark klient

Poslednou z hlavných súčastí aplikácie je Spark klient, ktorý spája funkcionalitu klienta pre sťahovanie dát a databázového klienta do jedného celku. Jeho hlavnou úlohou bude distribuovať tento výpočet pomocou frameworku Spark. Základnú jednotku distribuovaného výpočtu bude tvoriť stiahnutie všetkých príspevkov z jedného Twitter účtu alebo Facebookovej stránky a ich uloženie do HBase databázy pomocou grafového frameworku.

Vstupom aplikácie bude zoznam identifikátorov zdrojov na sociálnych sieťach. Zdroj je v tomto prípade Facebooková stránka alebo Twitter účet. Z tohto zoznamu vytvorí distribuovanú kolekciu RDD, kde jeden identifikátor zdroja bude jeden prvok tejto kolekcie.



Obr. 4.3: Výpočet Spark klienta.

Vďaka tomu Spark túto kolekciu rovnomerne rozdistribuuje do všetkých dostupných uzlov clusteru. Vstupom jedného uzlu bude teda jeden identifikátor zdroja. Uzol na základe typu tohto zdroja kontaktuje sociálnu sieť Facebook alebo Twitter, stiahne všetky dostupné príspevky a uloží ich do databázy.

K uloženiu dát sa využije databázový klient. O type tohto klienta, či to bude framework Halyard alebo HGraphDB rozhodne užívateľ aplikácie pri jej spustení. Po nastavení budú všetky uzly využívať práve tento framework a nebude ho možné počas behu aplikácie meniť.

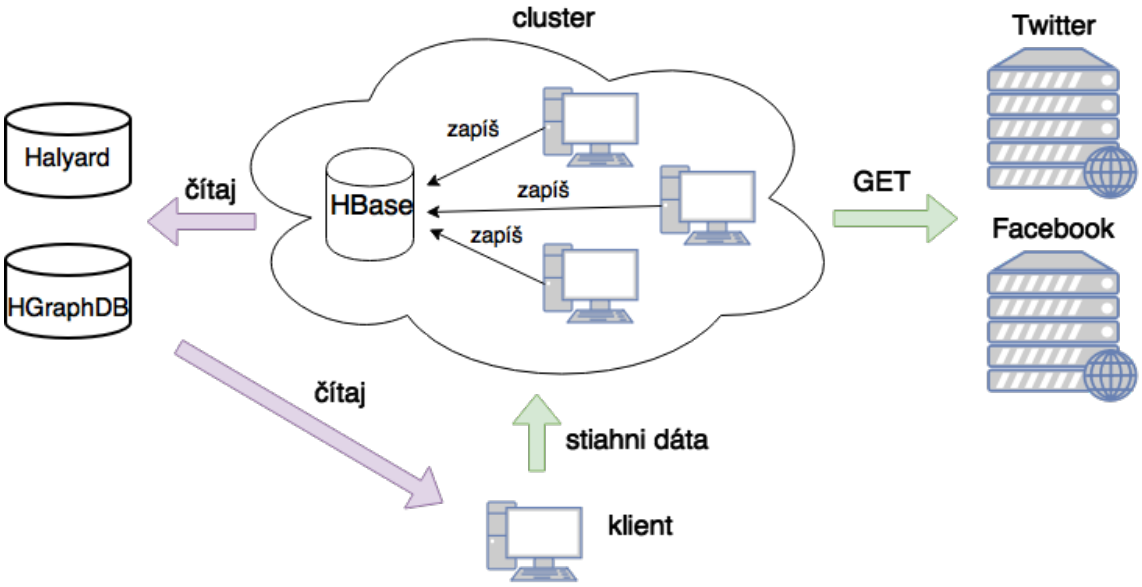
Návrh fungovania Spark klienta je možné vidieť na obrázku 4.3. Objekt *SparkClient* dostane na vstupe zoznam identifikátorov zdrojov na sociálnej sieti Twitter alebo Facebook. Klient premení tieto dáta na distribuovanú kolekciu RDD. Ďalej nad touto kolekciou vykoná distribuovaný výpočet. V jednom kroku výpočtu Spark klient stiahne dáta zo sociálnej siete Twitter alebo Facebook pomocou objektu *WebClient* a tieto dáta uloží s využitím databázového klienta, ktorý predstavuje objekt *DbClient*.

4.4 Návrh testovania aplikácie

Cieľom testovania aplikácie je zamerať sa na jednotlivé databázové frameworky. Výsledkom tejto práce by mala byť analýza ich použiteľnosti a výkonnosť operácii vkladania dát a dopytovanie sa nad nimi.

4.4.1 Hardware prostredia

Výsledná aplikácie bude k svojmu behu potrebovať distribuované prostredie. K dispozícii budem mať už nakonfigurovaný cluster o veľkosti štyroch počítačov. Každý počítač disponuje šestnástimi virtuálnymi jadrami a šestnástimi Gigabatami pamäte. Celý cluster poskytuje teda až šesťdesiat štyri pracovných uzlov.



Obr. 4.4: Architektúra výslednej aplikácie.

4.4.2 Testovanie zápisu

Prvou časťou testovania databázových frameworkov bude ich efektivita zapisovania dát. Testovacie dáta budú tvoriť príspevky stiahnuté zo sociálnych sietí. Výsledkom testu bude čas, ktorý bol potrebný pre zápis všetkých stiahnutých dát. Frameworky Halyard aj HGraphDB budú zapisovať totožné dáta.

Keďže výsledná aplikácia dáta zapisuje ihneď po ich stiahnutí, pre potreby testovania bude potrebné toto chovanie obmeniť. Dáta budú musieť byť po stiahnutí držané v pamäti a až po získaní všetkých dát ich aplikácia zapíše do databázy. Týmto spôsobom získame čistý čas zápisu. V klasickom behu aplikácie by sa totiž určitá časť dát zapisovala do databázy, kým ďalšia časť by bola len sťahovaná. Výsledný čas by preto záležal prevažne na čase sťahovania dát a nie na ich zápise. Obmenený beh aplikácie teda všetky dáta stiahne a potom ich naraz zapíše najprv pomocou frameworku Halyard a potom pomocou HGraphDB.

4.4.3 Testovanie čítania

Testovanie efektívnosti čítania bude prebiehať pomocou databázových dopytov. V aplikácii bude istá množina vopred definovaných dopytov, ktoré budú implementované pomocou frameworku Halyard a HGraphDB. Oba frameworky budú mať za úlohu z databázy získať rovnakú množinu dát. Výsledkom testu bude čas potrebný pre získanie týchto dát.

Pri výbere dopytov som sa zameral na to, aby obsiahli základné funkcionality databáz. Snahou bolo otestovať schopnosť frameworkov dáta radiť podľa hodnoty, radiť na základe funkcie, zoskupovať dáta, rátať počet výskytov, filtrovať na základe hodnoty a filtrovať podľa funkcie. Navyše som chcel v dopytoch otestovať schopnosť frameworkov pracovať s dátumami.

Databázové dotazy pre získavanie dát budú následovné:

`najdlhší text príspevku` dopyt pre získanie príspevku s najdlhším textom. Úlohou tohto dopytu bude otestovať možnosť frameworku radiť dáta podľa určitej funkcie.

príspevky novšie ako rok 2018 jednoduchý databázový dopyt pre získanie príspevkov novších ako rok 2018. Dopyt bude testovať efektívnosť filtrovania dát podľa určitej podmienky. Jeho ďalším výstupom bude navyše možnosť frameworkov pracovať s dátumami.

zoradené príspevky novšie ako rok 2018 totožný dopyt ako predošlý, no výsledné dáta budú zoradené od najnovšieho príspevku po najstarší. Tento dotaz otestuje radenie podľa obyčajnej hodnoty.

počet príspevkov výsledkom tohto dopytu bude zoznam Faebookových stránok a Twitter účtov s počtom ich príspevkov. Úlohou testu je otestovať schopnosť frameworku zoskupovať dáta podľa určitej hodnoty.

zdieľané odkazy dopyt z databázy získa počet výskytov každého url odkazu a výpíše len odkazy, ktoré sú v databáze uložené aspoň dvakrát a zoradí ich podľa počtu výskytu. Dopyt otestuje schopnosť frameworku filtrovať dáta na základe funkcie.

Kapitola 5

Implementácia

Cieľom tejto kapitoly je zdokumentovať priebeh a výsledky implementácie výslednej aplikácie. Implementácia vychádzala z návrhu predošlej kapitoly. Architektúru bolo možné vidieť na obrázku 4.4. Úlohou aplikácie je stiahnutie dát zo sociálnych sieti Twitter a Facebook, a ich následná analýza. K ukladaniu dát je využívaná distribuovaná databáza HBase.

Výsledná aplikácia sa skladá z troch hlavných častí. Prvá časť má za úlohu stiahnutie dát zo sociálnej siete a ich transformáciu do grafového modelu aplikácie. Druhou časťou je databázový klient, ktorý umožňuje dáta zapisovať do databázy a následne k nim pristupovať. Posledná časť spája stiahnutie dát a ich uloženie, a tento výpočet distribuuje medzi všetky dostupné uzly clusteru.

Aplikáciu som sa rozhodol implementovať ako open-source riešenie dostupné všetkým záujemcom. K jej zverejneniu využijem verzovaciu stránku Github, ktorá umožňuje zdrojové kódy aplikácie zverejniť a neskôr ich aj upravovať.

5.1 Klient pre sťahovanie dát

K testovaniu rôznych databázových riešení bolo potrebné získať dostatočné množstvo dát na testovanie. V dnešnej dobe je najväčšia časť verejných dát kumulovaná na sociálnych sieťach. K stiahnutiu týchto dát do modelu aplikácie slúži klient pre sťahovanie dát. Ako už bolo spomenuté, jeho úlohou je stiahnuť dáta a transformovať ich do grafového modelu aplikácie.

Vstupom tohto klienta je názov zdroja. Zdrojom je podľa typu sociálnej siete buď Facebookova stránka alebo Twitter účet. Klient následne stiahne všetky dostupné príspevky a vráti ich ako objekt typu *Timeline*. To, či klient pracuje so sociálnou sieťou Twitter alebo Facebook závisí na jeho implementácii. Obaja klienti implementujú totožné rozhranie, vďaka čomu odlučujú typ sociálnej siete od ich používateľa. Objekt, ktorý tohto klienta používa nemusí rozlišovať, či komunikuje so sociálnou sieťou Twitter alebo Facebook, a dáta stiahne totožným spôsobom. Ktorá implementácia sa použije, rozhoduje továreň na klientov. Tá na základe typu sociálnej siete vytvorí buď klienta pre Twitter alebo Facebook.

5.1.1 Twitter

Jednou z impelemntácii je klient pre sťahovanie dát zo sociálnej siete Twitter. K implementácii som využil knižnicu Twitter4J[1]. Je to open-source knižnica, ktorá poskytuje jedno-

[1]Stránky frameworku Twitter4J: http://twitter4j.org/en/index.html

duchú komunikáciu s Twitter API. Nejedná sa o oficiálnu knižnicu Twitteru, ale je to len výsledok snahy jedného vývojára a komunity zjednodušiť komunikáciu s Twitterom. Užívateľ sa nemusí zaoberať implementáciou sieťovej komunikácie a pracuje len s dostupnými triedami. Knižnica Twitter4J je celá implementovaná v programovacom jazyku Java.

Základnou podmienkou využívania Twiter API je to, aby bol používateľ úspešne autentizovaný. Ako bolo spomenuté Twitter poskytuje dve možnosti prihlásenia sa. Prihlásiť sa je možné buď pomocou užívateľského alebo aplikačného tokenu. Keďže môj požadovaný API endpoint pre získavanie užívateľských príspevkov má miernejší limit pri použití aplikačného tokenu, zvolil som tento spôsob autentizácie. Knižnica Twitter4J túto autentizáciu podporuje a k úspešnému prihláseniu vyžaduje aplikačný kľúč a tajomstvo.

K získaniu aplikačného kľúča a tajomstva je potrebné vytvoriť si Twitter aplikáciu. Tá sa vytvára na stránkach Twitteru[2] určených pre vývojárov. Po vytvorení má vývojár priamo k dispozícii kľúč aj tajomstvo.

Po získaní prihlasovacích údajov je možné získať z Twitteru požadované dáta. Mojou snahou je stiahnuť všetky zverejnené príspevky jedného Twitter účtu. K tomuto účelu slúži trieda Twitter4J knižnice *Twitter*, presnejšie jej metóda `getUserTimeline(String, Paging)`. Táto metóda príma ako parameter názov požadovaného účtu a objekt typu *Paging*. Jej návratovou hodnotou je kolekcia príspevkov. Metóda však nevracia všetky príspevky, ale len jednu stránku, ktorá môže obsahovať maximálne dvesto príspevkov. Pre získanie ďalších stránok slúži zmienený parameter *Paging*. Ten je možné postupne inkrementovať, vďaka čomu vieme získať ďalšie stránky. Tento parameter je možné inkrementovať až pokiaľ nedostaneme prázdnu stránku, ktorá neobsahuje žiaden príspevok. Tento stav značí to, že sme stiahli všetky dostupné príspevky.

Sťahovanie veľkého množstva stránok môže naraziť na limit. Vtedy Twitter zakáže aplikácii na určitý čas sťahovať ďalšie dáta. Mojou snahou pri implementácii bolo, aby sa s tým aplikácia vedela vysporiadať. Mojím riešením je detekovanie presiahnutia limitu a následné prerušenie vykonávania programu na najbližších pätnásť sekúnd. Po skončení limitu bude aplikácia pokračovať v štandardnom režime. Presiahnutie limitu je možné detekovať na základe vyhodenej výnimky. Tá vznikne pri pokuse stiahnuť novú stránku s príspevkami. Z výnimky je možné zistiť či vznikla kvôli presiahnutiu limitu alebo z nejakého iného dôvodu. Vývojári Twitter4J knižnice detekovanie značne zjednodušili. Ich výnimka obsahuje metódu `exceededRateLimitation()`, vďaka ktorej nie je potrebné vykonávať žiadne iné kontroly.

Stiahnutím dát funkcionalita tohto klienta nekončí. Stiahnutý príspevok je reprezentovaný Twitter4J objektom *Status*. Poslednou úlohou klienta je preto transformovanie tohto objektu do modelu aplikácie. Ako bolo spomínané, príspevok reprezentuje v aplikácii trieda *Entry*. Viacero príspevkov potom spája trieda *Timeline*. K transformácii stiahnutých objektov na výsledný model aplikácie som využil framework Timeline Analyzer. Keďže ide o open-source riešenie, framework som do výslednej aplikácie nezahrnul celý, ale využil som len jeho zdrojové kódy. V tomto prípade som použil metódu pre transformáciu objektov z knižnice Twitter4J na objekty aplikácie. Táto metóda príma kolekciu objektov *Status* a vracia už požadovaný objekt *Timeline*.

5.1.2 Facebook

Druhou implementáciou je klient pre sťahovanie dát zo sociálnej siete Facebook. Oficiálna knižnica pre Javu rovnako ako pri Twitteri neexistuje. K implementácii som teda využil

[2]Twitter stránky pre vývojárov: https://apps.twitter.com

open-source framework RestFB[3]. Vďaka nemu je možné komunikovať s Facebook API pomocou objektov bez potreby zaoberať sa nejakou sieťovou komunikáciou.

Facebook API nie je možné používať bez úspešnej autentizácie. Prihlásenie prebieha na základe aplikačného kľúča a tajomstva Facebookovej aplikácie. Po úspešnom prihlásení získame prístupový token, vďaka ktorému je možné dopytovať sa na ľubovoľné endpointy. Knižnica RestFB túto funkcionalitu zjednodušuje a po získaní tokenu je ho potrebné nastaviť len raz pri inicializácii RestFB klienta. Tento klient potom pri komunikácii s Facebookom token pripája ku každému sieťovému dopytu.

Podobne ako pri sociálnej sieti Twitter, je k získaniu aplikačného kľúča a tajomstva potrebné vytvoriť si Facebook aplikáciu. K tomuto účelu slúžia vývojárske stránky Facebooku[4]. Po vytvorení aplikácie Facebook okamžite vygeneruje jej kľúč aj tajomstvo, vďaka ktorým sa je možné úspešne autentizovať.

Po úspešnom nastavení tokenu prebieha celá komunikácia s Facebookom len pomocou tried knižnice RestFB. K volaniu Facebooku sa využíva trieda *FacebookClient* (v nasledujúcom texte však bude pojem Facebookový klient reprezentovať naďalej klienta mojej výslednej aplikácie pre sťahovanie dát z Facebooku a nie zmieňovanú triedu RestFB knižnice).

Úlohou Facebookového klienta je stiahnutie verejných príspevkov z Facebookových stránok. K tomuto účelu slúži metóda `FacebookClient::fetchConnection(String, Class, List<Parameter>)`. Oproti špecifickej metóde v prípade Twitter4J knižnice je zmieňovaná metóda generickejšia. Znamená to, že sa pomocou nej dajú sťahovať dáta z rôznych endpointov Facebook API. Z ktorého endpointu sa dáta stiahnu, závisí na prvom parametri. Ten reprezentuje cestu k požadovanému endpointu.

Endpoint pre sťahovanie príspevkov má cestu */<názov stránky>/feed*. Rovnako ako v prípade knižnice Twitter4J aj tento endpoint vracia jedným zavolaním len obmedzené množstvo príspevkov. Podľa dokumentácie Facebook API je to maximálne sto príspevkov. Metóda `fetchConnection()` s týmto obmedzením počíta a umožňuje výsledky postupne prechádzať. Jej návratová hodnota totiž nie je priamo kolekcia príspevkov ale iterátor nad príspevkami. Každým iterovaním sa teda zavolá nový Facebook API dopyt a dostaneme novú kolekciu príspevkov. Výsledná aplikácia toho využíva a iterátor iteruje pokiaľ nenarazí na jeho koniec. Medzivýsledky si postupne ukladá do vlastnej kolekcie a po poslednej iterácii obsahuje všetky požadované príspevky.

Aj Facebook API má limity na množstvo stiahnutých príspevkov. Oproti Twitteru však tieto limity nie sú presne stanovené. Pri implementácii som sa preto snažil presiahnutie limitu detektovať a rozumne ho vyriešiť. Trieda pre sťahovanie dát *FacebookClient* neobsahuje nejaký jednoduchý spôsob ako zaznamenať presiahnutie limitu. Výnimky vzniknuté pri sťahovaní dát je preto potrebne kontrolovať vlastným spôsobom. Podľa dokumentácie Facebook API, prekročenie limitu indikujú dva chybové kódy. Sú to hodnoty 4 a 17. Ak teda pri sťahovaní dát vznikne výnimka, porovnávam jej chybový kód s týmito dvomi číslami. Ak je aspoň jedno porovnanie pravdivé, program zavolá metódu pre obsluhu prekročenia limitu. Táto metóda rovnako ako v prípade Twitter klienta preruší vykonávanie programu na dobu pätnásť sekúnd. Po prebratí sa klient pokúsi opätovne stiahnuť dáta.

Počet príspevkov, ktoré je možné z jednej stránky stiahnuť nie je limitovaný. Hoci Facebook API tvrdí, že dovoľuje stiahnuť maximálne šesťsto najnovších príspevkov v jednom roku, po testovaní som zistil, že to neplatí vždy a niekedy nám API dovolí stiahnuť aj väčšie množstvo dát. Oproti Twitter klientovi, kde môžeme stiahnuť maximálne

[3]Stránky RestFB frameworku: http://restfb.com

[4]Facebook stránky pre vývojárov: https://developers.facebook.com/apps/

tritisíc dvesto najnovších príspevkov je v tomto prípade vhodné zaviesť určitý limit. Stiahnutie desaťročných príspevkov totiž nemusí byť vždy relevantné. Facebook klient preto obsahuje vo výslednej aplikácii možnosť pre stanovenie limitu najstaršieho príspevku. Metóda `fetchConnection(String, Class, List<Parameter>)` umožňuje tento limit nastaviť pridaním hodnoty do kolekcie parametrov. Nový parameter vytvorím v programe následovne: `Parameter.with('until', untilDate)`. Premenná `untilDate` obsahuje časové razítko vo formáte textového reťazca. Po zadaní tejto podmienky metóda `fectConnection()` vracia len príspevky novšie ako zvolený dátum.

Príspevok je v knižnici RestFB uložený do objektu *Post*. Pre jeho transformáciu do modelu aplikácie som opäť využil zdrojový kód frameworku Timeline Analyzer. Metóda tohto frameworku príma ako parameter kolekciu objektov *Post* a na výstupe vracia požadovaný objekt *Timeline*.

5.2 Databázový klient

Najpodstatnejšou časťou výslednej aplikácie je databázový klient. Vďaka nemu je možné otestovať efektivitu zvolených databázových riešení. V aplikácii som sa zameral na frameworky Halyard a HGraphDB. Ich hlavným rozdielom je dátový model, ktorý využívajú pre ukladanie dát. Halyard dáta ukladá pomocou RDF trojíc a v prípade HGraphDB je to property graf model. Jednou z úloh pri implementácii bolo preto nájsť spôsob ako model aplikácie uložiť čo najjednoduchšie do požadovaných formátov.

Úlohou databázového klienta je uložiť objekt typu *Timeline* do HBase databázy v grafovom formáte. Snahou pri impelementácii bolo, aby bol tento krok plne paralelný, bez žiadneho úzkeho hrdla. Nad uloženými dátami vie potom klient vykonávať rôzne databázové dopyty. Výsledná aplikácia nie je generická a užívateľovi neumožňuje vytvárať vlastné dopyty. Narozdiel od toho obsahuje niekoľko vopred implementovaných dopytov, medzi ktorými sa môže užívateľ pri používaní aplikácie rozhodovať. Získavanie informácii z databázy už neprebieha na rozdiel od vkladania paralelne.

Pred samotným uložením dát do databázy je potrebné ešte vytvoriť požadované tabuľky. V spôsobe tvorenia je medzi frameworkami Halyard a HGraphDB takisto podstatný rozdiel. Líšia sa v počte aj štruktúre tabuliek. Jednotlivé spôsoby tvorenia tabuliek budú podrobnejšie rozpísané vo frameworkových podkapitolách.

5.2.1 Halyard

Prvou implementáciou je klient, ktorý k ukladaniu dát využíva framework Halyard. Nevýhodou tohto frameworku je to, že k jeho používaniu potrebujeme centralizovaný RDF4J server. Táto podmienka je najmä pri paralelnom zápise veľmi nevhodná, keďže nám tento centralizovaný server bude tvoriť úzke hrdlo celej aplikácie. Pri implementácii zapisovania dát do HBase som sa preto zameral na obídenie použitia serveru RDF4J. Narozdiel od toho nebude čítanie dát prebiehať paralelne, preto som pri čítaní využil štandardný prístup.

Keďže som nechcel pre ukladanie dát využiť RDF4J server, musel som prísť s iným riešením. Využil som, že Halyard framework je open-source a jeho zdrojové kódy sú verejne dostupné na stránke Github[5]. Tento balík sa však nenachádza na verejnom Maven sklade, preto som ho nemohol do svojej aplikácie pridať ako klasickú závislosť. Zdrojové kódy som si stiahol do svojho počítača, kde som si ich preložil a vytvoril z nich vlastný

[5]Stránky frameworku Halyard: https://github.com/Merck/Halyard

balík. Ten som potom nainštaloval do svojho lokálneho skladu. Po nainštalovaní som už Halyard framework mohol do svojej aplikácie pridať ako klasickú závislosť a používať ho ako Java knižnicu. Nevýhodou tohto riešenia je to, že aplikáciu je možné samostatne spúšťať len na stroji, kde je balík Halyard nainštalovaný. Aby aplikácia fungovala aj v iných prostrediach, musia byť pri nej pribalené aj všetky závislosti. Celková veľkosť aplikácie je tak výrazne väčšia.

Konfigurácia

Pre správne fungovanie Halyard knižnice s HBase databázou je potrebné nastaviť v aplikácii HBase konfiguráciu. Knižnica Halyard očakáva v zdrojoch aplikácie súbor s názvom *hbase-site.xml*, ktorý obsahuje konfiguráciu HBase. Tento súbor sa nachádza v priečinku HBase inštalácie daného stroja. Pred spustením aplikácie je preto potrebné skopírovať tento súbor do jej zdrojov.

Tvorba tabuliek

Prvou úlohou tohto klienta je vytvorenie tabuľky pre ukladanie dát. Tabuľku je potrebné vytvoriť ešte pred samotným sťahovaním dát. Sťahovanie totiž prebieha paralelne čo by mohlo spôsobiť pri vytváraní konflikt. K tvoreniu tabuliek som využil triedu *HalyardTableUtils*, ktorá je súčasťou Halyard knižnice. Táto trieda obsahuje statickú metódu `getTable()`. Spomínaná metóda slúži pre získavanie referencie na HBase tabuľku, vďaka ktorej je do nej možné ukladať dáta. Jej podstatnou vlastnosťou je však to, že ak tabuľka s daným menom v HBase databáze ešte neexistuje, vytvorí ju. Toto správanie zapína jej boolean parameter `create`. Výsledné chovanie je v mojom prípade veľmi výhodné a využil som ho pre tvorenie tabuliek. Samotnú referenciu na tabuľku, ktorú táto metóda vracia v programe pri vytváraní tabuliek zahadzujem.

Vkladanie dát

Po vytvorení tabuľky je do nej možné vkladať dáta. Pri použití Halyard frameworku je možné tieto dáta vkladať priamo do HBase tabuľky. K tomuto účelu som využil opäť triedu *HalyardTableUtils* a jej statickú metódu `getTable()`. Návratovú hodnotu tejto metódy už nezahadzujem. Keďže ide o referenciu na HBase tabuľku, môžem pomocou nej do databázy vkladať dáta.

Referencia tabuľky je v programe objekt typu *HTable*. Je to súčasť frameworku Halyard a umožňuje okrem iného vkladať dvojice (kľúč, hodnota) priamo do HBase tabuľky. Tieto dáta sú uložené v správnom formáte, takže ich vie Halyard neskôr bez komplikácii čítať. Vďaka tejto funkcionalite som sa v aplikácii nemusel zameriavať na dodržiavanie správneho formátu tabuliek.

K uloženiu dát potrebuje objekt *HTable* dvojicu (kľúč, hodnota). Tá je reprezentovaná Halyard objektom *KeyValue*. Hlavnou úlohou vkladania dát bolo preto transformovať objekt *Timeline* na kolekciu týchto objektov. K tomuto účelu som musel využiť framework RDF4J.

Model aplikácie je prevzatý z frameworku Timeline Analyzer. Jeho hlavnou vlastnosťou je to, že rozširuje základnú entitu RDF4J frameworku. Tá v sebe obsahuje metódu pre pridanie samej seba do RDF modelu. Model je vo frameworku RDF4J len alias pre kolekciu RDF tvrdení. Tieto vlastnosti sú v mojej aplikácii veľmi žiadané. Halyard trieda *HalyardTableUtils* obsahuje okrem metódy pre získavanie referencia na HBase tabuľku a mnoho ďalších metód aj metódu pre transformáciu RDF tvrdení na kolekciu objektov *KeyValue*.

Vo výsledku Halyard klient prijme na vstupe objekt *Timeline*. Ten následne vloží do RDF modelu, vďaka čomu získa kolekciu RDF tvrdení. Tieto tvrdenia transformuje pomocou triedy *HalyardTableUtils* na kolekciu objektov *KeyValue*. Tieto objekty klient v poslednom kroku vloží pomocou referencie na HBase tabuľku do databázy.

Analýza dát

Analyzovanie dát prebieha v tomto klientovi pomocou dopytovacieho jazyka SPARQL a RDF4J serveru. Klient sa pripojí na vzdialený RDF4J server a získava z neho požadované dáta.

Výhodou tohto riešenia je najmä jednoduchá tvorba dopytov. Všetok výpočet prebieha na strane serveru. Klientovi sa stačí pripojiť na tento server a zadať názov úložiska s ktorým bude pracovať. Toto riešenie tak pred klientom tieni reálnu implementáciu úložiska a klient bude pracovať rovnako s natívnym RDF4J úložiskom ako aj s distribuovaným úložiskom Halyard. Po pripojení posiela klient na server SPARQL dopyty a ten odpovedá požadovanými dátami. Ďalšou výhodou tohto riešenia je to, že aplikácia nemusí byť spúšťaná na distribuovanom clusteri.

Dopyt pre získanie najdlšieho textového obsahu

Halyard klient implementuje všetky navrhnuté databázové dopyty. Prvý databázový dopyt má za úlohu získať z databázy zdrojový identifikátor príspevku s najdlhším textovým obsahom.

V jazyku SPARQL vieme získať dĺžku reťazca pomocou funkcie `strlen()`. Tejto hodnote je navyše možné nastaviť alias, pomocou ktorého je možné dáta radiť a filtrovať. K nastaveniu aliasu slúži databázová klauzula `BIND AS`. V implementovanom dopyte príspevky radím podľa dĺžky ich textového obsahu a limitujem na prvý výsledok. K radeniu slúži klauzula `ORDER BY` a k limitovaniu klauzula `LIMIT`.

Výsledok dopytu je identifikátor príspevku s najdlhším textovým obsahom a dĺžka tohto textu. Dopyt v SPARQL jazyku vyzerá následovne:

```
SELECT ?entry ?sourceId ?textlen
WHERE {
   ?entry <http://nesfit.github.io/ontology/ta.owl#sourceId> ?sourceId .
   ?entry <http://nesfit...#contains> ?content .
   ?content <http://nesfit...#type> <http://nesfit...#TextContent> .
   ?content <http://nesfit...#text> ?text .
   BIND (strlen(?text) AS ?textlen)
}
ORDER BY DESC (?textlen)
LIMIT 1
```

Dopyty pre získanie času pridania príspevkov

Úlohou ďalšieho databázového dopytu bolo získať všetky príspevky novšie ako rok 2018. Podobnú úlohu mal aj ďalší dopyt, ktorý tieto príspevky navyše radí podľa ich dátumu pridania.

Jazyk SPARQL disponuje podporou práce s časmi a umožňuje podľa týchto hodnôt filtrovať alebo radiť. Filtrovanie dát na základe určitej hodnoty prebieha v SPARQL jazyku

pomocou klauzule `FILTER`. Tej môžme priradiť ľubovoľnú podmienku, čoho následkom bude vypustenie dát, ktoré túto podmienku nespĺňajú. Dopyt pre získanie dát a ich následné radenie vyzerá následovne:

```
SELECT ?label ?sourceId ?timestamp
WHERE {
   ?timeline <http://www.w3.org/2000/01/rdf-schema#label> ?label .
   ?entry <http://nesfit...#sourceTimeline> ?timeline .
   ?entry <http://nesfit...#timestamp> ?timestamp .
   ?entry <http://nesfit...#sourceId> ?sourceId
   FILTER ( ?timestamp >= xsd:dateTime('2018-01-01T00:00:00.00Z') )
}
ORDER BY DESC (?timestamp)
```

Dopyt pre získanie počtu príspevkov

Ďalší dopyt má za úlohu získať počet príspevkov každej Facebookovej stránky a Twitter účtu (v databáze uložených pod rovnakou entitou *Timeline*). Dopyt bol zameraný na schopnosť zhlukovať dáta podľa určitej hodnoty a rátať veľkosť týchto zhlukov. Zhlukovať dáta je v SPARQL jazyku možné pomocou klauzuly `GROUP BY`. K rátaniu veľkosti zhluku slúži funkcia `count()`. Dopyt pre získanie týchto dát vyzerá následovne:

```
SELECT ?label (count(?entry) AS ?numberOfEntries)
WHERE {
   ?timeline <http://www.w3.org/2000/01/rdf-schema#label> ?label .
   ?entry <http://nesfit...#sourceTimeline> ?timeline .
}
GROUP BY ?label
```

Dopyt pre získanie zdieľaných odkazov

Posledný dopyt má za úlohu získať odkazy, ktoré sa v rôznych príspevkoch vyskytujú aspoň dvakrát a zoradiť ich od najväčšieho výskytu po najmenší. Prvou úlohou dopytu bolo teda odfiltrovať odkazy, ktoré sa v databáze nachádzajú iba raz. K tomuto účelu nie je možné využiť už spomínanú klauzulu `FILTER`, ale podobne ako v bežnom SQL jazyku k tomu slúži klauzula `HAVING`. Odfiltrované dáta sú ďalej v dopyte zoradené podľa výskytu. Výsledný dopyt vyzerá následovne:

```
SELECT ?sourceUrl (count(?content) AS ?numberOfContents)
WHERE {
   ?content <http://www.w3...#type> <http://nesfit...#URLContent> .
   ?content <http://nesfit...#sourceUrl> ?sourceUrl
}
GROUP BY ?sourceUrl
HAVING (?numberOfContents > 1)
ORDER BY DESC(?numberOfContents)
```

5.2.2 HGraphDB

Druhým spôsobom práce s HBase databázou je klient, ktorý k práci využíva HGraphDB framework[6]. Oproti Halyard frameworku je v tomto prípade jeho použitie jednoduchšie. Balík HGraphDB je verejne dostupný v Maven sklade balíkov, preto už len jeho pridanie do závislostí aplikácie je omnoho jednoduchšie.

Práca s HBase prebieha pomocou frameworku priamo bez žiadneho potrebného serveru. Vďaka tomu som nepotreboval študovať zdrojové kódy HGraphDB frameworku, napriek tomu, že sú taktiež verejne dostupné. Mohol som ho v aplikácii využívať tak, ako bol pri implementácii určený.

Konfigurácia

Rovnako ako v prípade Halyard aj HGraphDB potrebuje k správnej komunikácii s HBase databázou jej konfiguráciu. HGraphDB si však túto konfiguráciu nevie načítať samostatne a je potrebné všetky parametere nastaviť priamo v Jave. Toto riešenie pre mňa nebolo vhodné. Ak by som totiž všetky HBase parametre musel natvrdo nastavovať v zdrojových kódoch, bolo by to veľmi neefektívne. Pri zmene prostredia by som navyše musel zdrojové kódy opätovne prepisovať.

Vhodným riešením by pre mňa preto bolo, ak by som mohol túto konfiguráciu nastaviť priamo z *hbase-site.xml* súboru. Takýmto spôsobom by som nemusel udržiavať dve rôzne konfigurácie pre dva rôzne frameworky. V aplikácii som si preto implementoval vlastný parser HBase konfigurácie. Jeho úlohou je prečítať ľubovoľný súbor vo formáte XML konfigurácie a vrátiť jeho dáta ako kolekciu kľúčov a ich hodnôt.

Tento parser som potom využil v HGraphDB klientovi. Ako parameter mu poskytnem cestu k *hbase-site.xml* súboru, vďaka čomu získam všetky parametre HBase konfigurácie. Tieto informácie môžem potom dynamicky nastaviť v HGraphDB konfigurácii. Takýmto spôsobom potrebujem v celej aplikácii len jeden súbor s HBase konfiguráciou, ktorý využijem v oboch databázových klientoch.

Tvorba tabuliek

V prípade frameworku HGraphDB je tvorba tabuliek opäť odlišná. Halyard potrebuje k behu len jednu tabuľku, zatiaľ čo HGraphDB dáta ukladá do niekoľkých rôznych tabuliek. Oproti Halyard však tabuľky využíva štandardným spôsobom, kde každý záznam je vo vlastnom riadku a stĺpce tvoria vlastnosti uloženej entity. Všetky vzniknuté tabuľky sa vyznačujú totožným prefixom, ktorý môže tvoriť ľubovoľný textový reťazec zadaný užívateľom. Prefix má samozrejme určité obmedzenia týkajúce sa špeciálnych znakov. Nemôže napríklad obsahovať spojovník, lomítko a ďalšie znaky.

Tvorba tabuliek je už zahrnutá vo frameworku, vďaka čomu sa o tvorbu a formátovanie tabuliek postará samotný framework. K tomuto účelu som však musel využiť až dva rôzne frameworky a to samotný HGraphDB a framework TinkerPop. Ako bolo v predošlých kapitolách spomenuté, HGraphDB je s týmto frameworkom úzko spätý a samostatne ho nie je možné používať.

Tabuľky sa v HGraphDB frameworku tvoria veľmi podobne ako to bolo v prípade Halyard. Metóda, ktorá nám vytvára spojenie do databázy obsahuje doplnkovú funkcionalitu. Ak tabuľky so zvoleným prefixom ešte nie sú vytvorené, metóda ich pred vytvorením databázového spojenia vytvorí. V programe mi preto stačí vytvoriť spojenie do databázy

[6]Stránky frameworku HGraphDB: https://github.com/rayokota/hgraphdb

a následne ho zatvoriť. Metóda je súčasťou TinkerPop triedy *GraphFactory* a ako parameter príma ľubovoľnú konfiguráciu. V našom prípade je to HBase konfigurácia z balíčka HGraphDB. Volanie metódy pre vytvorenie tabuliek vyzerá v zdrojovom kóde následovne: `(GraphFactory.open(HBaseConfiguration)).close()`. Prefix tabuliek je zahrnutý v konfigurácii HBase.

Vkladanie dát

Vkladanie dát prebieha opäť pomocou dvoch frameworkov. Framework TinkerPop vytvára spojenie do HBase databázy a o vkladanie dát sa stará framework HGraphDB.

Spojenie do databázy vytvorím rovnako metódou ako pri tvorbe tabuliek. V tomto prípade ho však ihneď nezatváram. Vytvorenie spojenia vyzerá v zdrojovom kóde následovne:

```
HBaseGraph graph = (HBaseGraph) GraphFactory.open(HBaseGraphConfiguration);
```

Konfigurácia, ktorú metóda pre vytvorenie spojenia príma ako parameter obsahuje HBase konfiguráciu zo súboru *hbase-site.xml* a prefix tabuliek. Táto metóda vracia objekt, ktorý implementuje TinkerPop rozhranie *Graph*. Implementácia tohto rozhrania závisí na použitej databáze. V tomto prípade budem k ukladaniu dát využívať databázu HBase. Implementácia, ktorá nám umožňuje pracovať s HBase je obsiahnutá v knižnici HGraphDB pod názvom *HBaseGraph*. Vďaka tejto implementácii môžem pracovať s HBase databázou pomocou TinkerPop frameworku ako keby to bola štandardná centralizovaná grafová databáza.

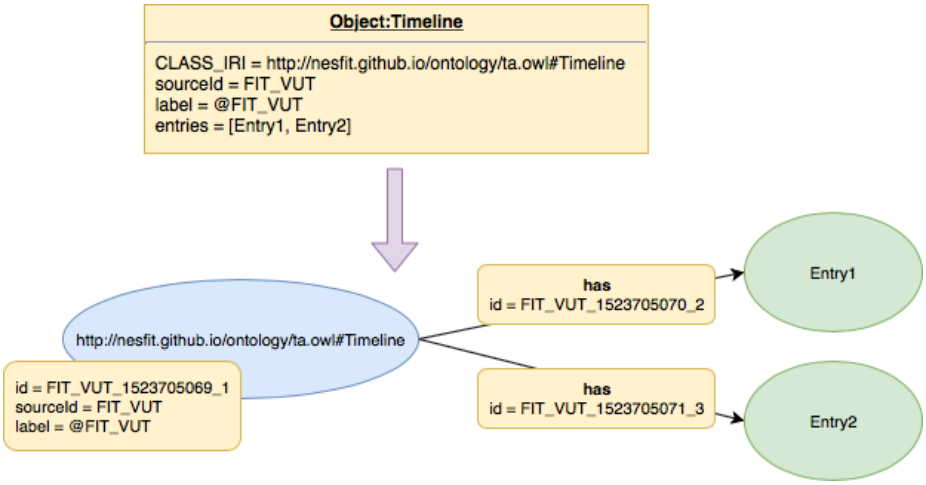
Objekt *HBaseGraph* umožňuje to databázy vkladať vrcholy a hrany grafu. Keďže HGraphDB pracuje s grafovým model property graf so štítkami, uzly aj hrany grafu môžu obsahovať rôzne vlastnosti. Každá hrana aj uzol má navyše svoj jedinečný identifikátor a štítok, ktorý reprezentuje typ grafovej entity.

Keďže je model aplikácie prispôsobený formátu RDF, bolo ho potrebné rozšíriť o podporu grafového formátu property graf. Pre túto potrebu som každú triedu aplikačného modelu rozšíril o metódu, v ktorej inštancia tejto triedy pridá samú seba do databázy. Databázová entita bude obsahovať jedinečný identifikátor, štítok (ten bude reprezentovaný pomocou IRI identifikátoru) a všetky inštančné vlastnosti. K tejto entite navyše vytvorím hrany vedúce do objektov, ktoré obsahuje. Všetky štítky hrán majú v aplikácii hodnotu *"has"*, keďže ich nie je potrebné rozlišovať.

Príklad vytvorenia databázovej entity je možné vidieť na obrázku 5.1. Objekt *Timeline* obsahuje tri vlastnosti:

`CLASS_IRI` značí IRI identifikátor triedy objektu.
`sourceId` značí textový identifikátor Twitter účtu alebo Facebook stránky.
`label` obsahuje ľubovoľný popis entity.

Vzniknutá grafová entita je typu *Timeline*, kde je jej typ (štítok) reprezentovaný celým IRI identifikátorom. Všetky vlastnosti objektu *Timeline* sú uložené ako vlastnosti entity s rovnakým pomenovaním. Je potrebné poznamenať, že vlastnosť *label* nereprezentuje štítok grafovej entity, hoci sú v anglickom jazyku pomenované rovnako. Entita obsahuje navyše ešte jedinečný identifikátor, ktorý je zložený z jej zdrojového identifikátoru, časového razítka a indexu. Jednotlivé príspevky danej *Timeline* sú uložené v samostatnej entite. Medzi entitami *Timeline* a príspevkom je vytvorená grafová hrana so štítkom *"has"* a opäť s jedinečným identifikátorom. Takýmto spôsobom sú uložené aj ďalšie objekty.



Obr. 5.1: Tranformácia objektu Timeline na property graf.

Vo výslednej aplikácii vytváram z objektov jednotlivé uzly a hrany, ktoré potom ukladám pomocou referencie na HBase tabuľku. Po uložení celého objektu *Timeline* je spojenie s databázou ukončené a dáta je možné čítať.

Analýza dát

Analyzovanie dát vo frameworku HGraphDB prebieha odlišným spôsobom ako tomu bolo v Halyard. Klient sa pripája priamo na HBase databázu a dopytuje sa na ňu. Oproti Halyard klientovi musí byť preto aplikácia spúšťaná na serveri, kde beží databáza HBase. Medzi klientom a databázou teda nie je žiadna ďalšia vrstva. Keďže sa však databázové dopyty tvoria pomocou frameworku TinkerPop a jazyka Gremlin, všetky implementované dopyty sú kompatibilné s ďalšími Tinkerpop úložiskami. Implementovaný dopyt som mohol testovať na malom, centralizovanom úložisku a neskôr sa choval rovnako s HGraphDB distribuovaným úložiskom.

Jazyk Gremlin umožňuje písať databázové dopyty priamo v jazyku Java. Ako prvé je potrebné pripojiť sa na HBase tabuľku. S pomocou pripojenia a metódy `traversal()` vytvorím objekt typu *GraphTraversalSource*. Tento objekt umožňuje prehľadávať grafový priestor pomocou Gremlin dopytov a získavať tak z neho požadované dáta.

Dopyt pre získanie najdlhšieho textového obsahu

Prvým databázovým dopytom je dopyt pre získanie príspevku s najdlhším textovým obsahom. Gremlin jazyk neobsahuje vstavanú funckciu pre rátanie dĺžky textového reťazca ako tomu bolo v prípade Halyard, no podporuje radenie podľa lambda výrazu. V tomto výraze je teda možné získať dĺžku reťazca a radiť podľa nej. V programe získam pre každý uzol typu *TextContent* jeho hodnotu parametru *text* a z nej už viem vyrátať dĺžku reťazca. Funkcia `order().by()` tieto hodnoty vezme a zoradí podľa nej výsledky.

Výstup dopytu má byť zdrojový identifikátor príspevku. Táto hodnota sa nachádza v uzle typu *Entry*. Keďže dopyt pracuje nad uzlami typu *TextContent* vo výsledku bolo ešte potrebné získať *Entry* uzol, ktorý ukazuje do uzlu *TextContent* s najdlhším reťazcom. K tomuto účelu slúži funkcia `in()`. Jej výstupom je uzol, ktorý ukazuje do aktuálneho uzlu. Opačnú funkcionalitu má funkcia `out()`.

Aby som mohol z výsledku vytiahnuť požadované hodnoty, v dopyte bolo potrebné jednotlivým výstupom priradiť určitý alias. K tomu účelu slúži funkcia `project().by()`, ktorá umožňuje každej hodnote priradiť textový názov. Z výsledku dopytu vďaka tomu môžem získať hodnoty podľa názvu. Výsledný Gremlin dopyt vyzerá následovne:

```
g.V()
.hasLabel("http://nesfit.github.io/ontology/ta.owl#TextContent")
.order().by(v -> v.property("text").value().length(), Order.decr)
.limit(1)
.project("text", "sourceId")
   .by(values("text"))
   .by(in("has").values("sourceId"))
```

Výstupom tohto dopytu je dvojica (textový obsah príspevku, identifikátor zdroja príspevku). V jazyku Gremlin sa mi nepodarilo z databázy získať priamo dĺžku textového obsahu. Túto hodnotu si preto vyrátam až pri vypisovaní dát.

Dopyty pre získanie času pridania príspevkov

Ďalší dopyt mal za úlohu získať z databázy príspevky novšie ako rok 2018. Podobný dopyt ich mal ešte navyše zoradiť od najnovšieho po najstarší. Dopyty mali za úlohu otestovať schopnosť frameworku pracovať s dátumami. Rovnako ako Halyard, aj HGraphDB disponuje plnou podporou dátumov a umožňuje na ich základe dáta radiť a filtrovať. Výsledný dopyt, ktorý dáta navyše radí vyzerá následovne:

```
g.V()
.hasLabel("http://nesfit.github.io/ontology/ta.owl#Entry")
.has("timestamp", P.gt(year2018))
.order().by(values("timestamp"), Order.decr)
.project("sourceId", "timestamp", "label")
   .by(values("sourceId"))
   .by(values("timestamp"))
   .by(in("has").values("label"))
```

Dopyt pre získanie počtu príspevkov

Úlohou ďalšieho dopytu bolo otestovať schopnosť frameworku zhlukovať dáta a rátať počet záznamov v jednom zhluku. Gremlin však nie je optimalizovaný na zoskupovanie podľa otcovských uzlov. K tomuto účelu preto nebolo vhodné využiť zoskupovaciu funkciu, ktorá navyše ráta veľkosť zhlukov `groupCount()`. Po testovaní na väčších dátových setoch totiž dopyt postupne prestal získavať výsledky v reálnom čase.

Vhodnejším riešením bolo využiť projekciu. V dopyte získam všetky uzly typu *Timeline* a pre každý následne vyrátam počet ich uzlov *Entry*. Len pre porovnanie, v Halyard prebiehal tento výpočet presne naopak. Najskôr som z databázy vytiahol uzly typu *Entry* a tie som následne zoskupil podľa uzlu do nich smerujúceho (v tomto prípade uzol typu *Timeline*). Výsledný dopyt vyzerá následovne:

```
g.V()
.hasLabel("http://nesfit.github.io/ontology/ta.owl#Timeline")
.project("label", "count")
   .by(values("label"))
```

```
.by(out().count());
```

Dopyt pre získanie zdieľaných odkazov

Posledný databázový dopyt mal za úlohu získať odkazy, ktoré boli zmieňované v rôznych príspevkoch aspoň dvakrát a následne ich zoradiť od najväčšieho výskytu po najmenší. Zo všetkých predošlých dopytov bol tento na implementáciu najnáročnejší.

V prvom kroku som zoskupil všetky uzly typu *URLContent* podľa url odkazu. Následne som výsledky rozložil pomocou `unfold()`. Táto funkcia rozloží všetky vnorené výsledky na jednu úroveň. Jej výsledok znázorním v následujúcom príklade:

```
[[{url1: 2}, {url2: 1}]] -> [{url1: 2}, {url2: 1}]
```

V tomto príklade som získal z funkcie `groupCount()` jednotlivé url a počet ich výskytov. Tieto dáta však boli vnorené v ďalšej kolekcii, takže by bolo náročné s nimi ďalej pracovať. Všetky ďalšie Gremlin výrazy by sa totiž týkali celkovej kolekcie a nie každého záznamu zvlášť. Vďaka funkcii `unfold()` sa dáta rozložia na najnižšiu úroveň a je možné nad nimi zadávať ďalšie funkcie.

Ďalším krokom dopytu je vyradenie odkazov, ktoré sa v databáze nachádzajú iba raz. K tomuto účelu som využil funkciu `filter()`, ktorá prijíma ako parameter ľubovoľný lambda výraz.

Posledným krokom dopytu je zoradenie dát podľa počtu výskytov. Opäť som využil funkciu `order().by()` a jej schopnosť radiť na základe lambda výrazu. Výsledný dopyt vyzerá následovne:

```
g.V()
.hasLabel("http://nesfit.github.io/ontology/ta.owl#URLContent")
.groupCount().by(values("sourceUrl"))
.unfold()
.filter(entry -> entry.get().getValue() > 1)
.order().by(entry -> entry.getValue(), Order.decr)
```

5.3 Spark klient

Poslednou súčasťou aplikácie je Spark klient. Jeho úlohou je spojiť funkcionalitu klienta pre sťahovanie dát a databázového klienta. Ich výpočty navyše distribuuje na Spark cluster. Jeho snahou je to, aby mal každý uzol clusteru na starosti stiahnutie všetkých dostupných príspevkov z jedného zdroju a ich následné uloženie do HBase databázy pomocou grafového formátu. Samozrejme pri používaní aplikácie nebude vždy možné, aby mal jeden uzol na starosti len jeden zdroj.

Konfigurácia

Pre správne fungovanie Spark frameworku je mu potrebné dodať konfiguráciu Sparku. Táto konfigurácia je závislá na prostredí, kde je Spark cluster nainštalovaný. Konfiguráciu poskytuje aplikácii užívateľ. Ten má možnosť túto konfiguráciu nastaviť staticky v konfigurácii aplikácie alebo ju zadať pri spúšťaní aplikácie. Statická konfigurácia je vždy uprednostňovaná pred konfiguráciou zadanou dynamicky. Pre zmenu statickej konfigurácie je potrebné

celú aplikáciu opäť preložiť a vytvoriť z nej nový balíček. Všetky možné parametre, ktoré je možné meniť sú k dispozícii na stránkach Sparku[7].

Distribúcia zdrojov

Vstupom Spark klienta je kolekcia zdrojov na sociálnuch sieťach. V programe je tento zdroj reprezentovaný triedou *Source*. Tá obsahuje identifikátor zdroju (názov Twitter účtu alebo Facebookovej stránky) a typ tohto zdroja (Twitter alebo Facebook). Túto kolekciu klient rozdistribuuje nad celý Spark cluster.

Aby bolo možné vstupnú kolekciu zdrojov distribuovať na celý Spark cluster, je z nej potrebné vytvoriť distribuovanú kolekciu RDD. Metóda, ktorá umožňuje transformovať ľubovoľnú Java kolekciu na kolekciu RDD je zahrnutá v triede *JavaSparkContext*. Tento kontext je možné vytvoriť na základe Spark konfigurácie.

Výpočet jedného uzlu

Každý uzol clusteru dostane na vstup práve jeden zdroj. Uzol si na základe typu zdroja vytvorí inštanciu klienta pre sťahovanie dát. Pomocou tohto klienta získa zo sociálnej siete objekt *Timeline*. Tento objekt predá ďalej inštancii databázového klienta. Typ klienta bude závisieť na argumente, ktorý zadá užívateľ aplikácii pri jej spustení.

Kód jedného uzlu Spark clusteru vyzerá následovne:

```
SourceClient sourceClient = sourceFactory.createClient(source.getType());
Timeline timeline = sourceClient.getTimeline(source.getName());

HBaseClient hBaseClient = hBaseClientFactory.createClient();
hBaseClient.saveTimeline(timeline);
```

Objekt *sourceClient* reprezentuje klienta pre sťahovanie dát a objekt *hBaseClient* značí klienta databázového. Ako je možné z kódu vidieť, Spark klient nie je závislí na implementácii týchto dvoch klientov. Vďaka tomuto je možné do aplikácie jednoducho pridať podporu ďalšej socíalnej siete alebo iného databázového frameworku.

[7]Stránky frameworku Spark: https://spark.apache.org/docs/latest/configuration.html

Kapitola 6

Dosiahnuté výsledky

Výstupom tejto práce je analýza dvoch riešení pre prácu s grafovou databázou v distribuovanom prostredí. Zameral som sa na dve rôzne riešenia. Ide o framework Halyard a HGraphDB. Mojou snahou bolo analyzovať ich schopnosť vkladať a čítať dáta, a efektívnosť týchto operácií.

Oba tieto frameworky nie sú distribuovanou grafovou databázou v pravom slova zmysle, ale ide o akési rozhranie, ktoré umožňuje pracovať s distribuovanou databázou HBase ako keby to bola databáza grafová. Rovnako oba frameworky neposkytujú užívateľské rozhranie ani vlastné Java API. Implementujú však rozhranie, s ktorým vedia pracovať frameworky pre prácu s grafovými databázami.

Halyard implementuje úložisko frameworku RDF4J. Framework HGraphDB je úložiskom frameworku TinkerPop. Vďaka tomu je celá práca s databázou pred užívateľom skrytá. Znamená to, že užívateľ pri ich používaní pracuje totožne s databázou distribuovanou ako aj s centralizovanou.

Rozdiel medzi frameworkami Halyard a HGraphDB je najmä v ich dátovom modeli, ktorý využívajú k ukladaniu dát. V prípade frameworku Halyard je to model RDF. HBase databáza tvorí v tomto prípade takzvaný sklad trojíc. Dáta sú v tomto sklade uložené ako trojice subjekt, predikát a objekt. Subjekt a objekt tvoria uzly grafu a predikát je väzba medzi nimi. Uzly a hrany grafu môžu obsahovať v tomto modeli len jednu jedinú hodnotu. Táto hodnota je buď typ grafovej entity, alebo literál.

Framewok HGraphDB využíva dátový model property graf. Tento model ukladá do databázy zvlášť uzly a zvlášť hrany medzi nimi. Každý uzol a hrana môže navyše obsahovať niekoľko rôznych parametrov.

6.1 Použiteľnosť

Ako bolo spomínané, práca s Halyard a HgraphDB prebieha pomocou frameworku tretej strany. V prípade Halyard je to framework RDF4J a pre prácu s HGraphDB slúži framework TinkerPop.

Výhodou RDF4J je to, že dokáže fungovať ako samostatný server, prístupný cez REST API. Komunikovať s ním môžu preto takmer všetky druhy zariadení. V prípade Javy, existuje pre komunikáciu s týmto serverom knižnica, ktorá sa stará o všetku sieťovú komunikáciu. Vďaka tomu je možné jednoducho na tento server posielať dopyty a získavať z neho dáta. K pripojeniu na server je potrebná len adresa serveru a názov úložiska, s ktorým sa pracuje. Dopyty sú tvorené pomocou jazyka SPARQL, ako textový reťazec. Server má

navyše k dispozícii vlastné webové rozhranie, kde je možné pracovať s dátami bez vlastnej aplikácie.

Vkladanie dát je možné v RDF4J takisto pomocou serveru. Klient sa pripojí na vzdialený server a vloží do neho kolekciu RDF trojíc. V distribuovanom prostredí však vkladanie dát cez centralizovaný server nie je vhodným riešením. Pre optimálnejšie vkladanie dát je však možné využiť priamo zdrojové kódy frameworku Halyard. V tomto prípade je možné obísť RDF4J server a dáta vložiť priamo do HBase databázy. V podstate je možné využiť rovnaký kód, ktorý by pre vloženie dát použil RDF4J server.

Práca s frameworkom TinkerPop je odlišná. Pomocou tohto frameworku sa k databáze pripája priamo v zdrojovom kóde. Aplikácia preto musí byť spustená na clusteri, kde beží HBase databáza. Výhodou tohto prístupu je absencia úzkeho hrdla programu. Dáta je možné analyzovať pomocou dopytovacieho jazyka Gremlin. V tomto jazyku sa dopyty skladajú Java jazykom.

Vkladanie dát prebieha takisto priamo do databázy. Vkladať je možné uzly a hrany medzi nimi. Pre optimalizáciu je framework HGraphDB vybavený vlastnou triedou pre vkladanie dát. Oproti klasickej TinkerPop triede je táto optimalizovaná pre veľké množstvo vkladaných dát.

6.2 Vkladanie dát

V prvom kroku testovania aplikácie som sa zameral na efektívnosť jednotlivých frameworkov pri vkladaní veľkého množstva dát. Výsledkom tohto testu je teda čas, ktorý bol potrebný pre vloženie všetkých dát. Pre účely testovania som využil dva rôzne testy. Prvý test mal za úlohu otestovať efektívnosť frameworkov v reálnej aplikácii a druhý test mal za úlohu izolovať časť aplikácie, ktorá dáta zapisovala. Výstupom prvého testu je teda čas potrebný pre stiahnutie a zápis dát, zatiaľ čo výstup druhého testu je len čas zápisu do databázy.

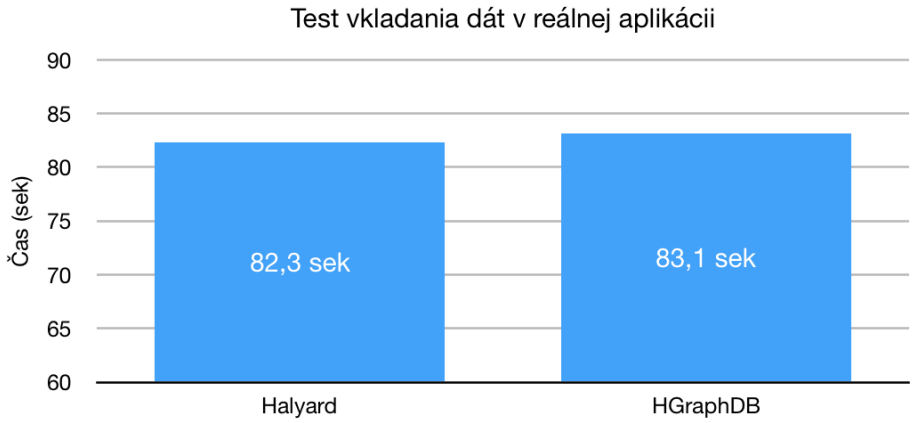
6.2.1 Reálne použitie

Prvý test mal za úlohu simulovať použitie frameworkov v reálnej aplikácii. Dáta sa po stiahnutí uložia okamžite do databázy. Vo výsledku to znamená, že kým sa niektoré dáta ešte len sťahujú, iné sú už ukladané. Tento prístup zaisťuje najefektívnejšie využitie všetkých dostupných zdrojov.

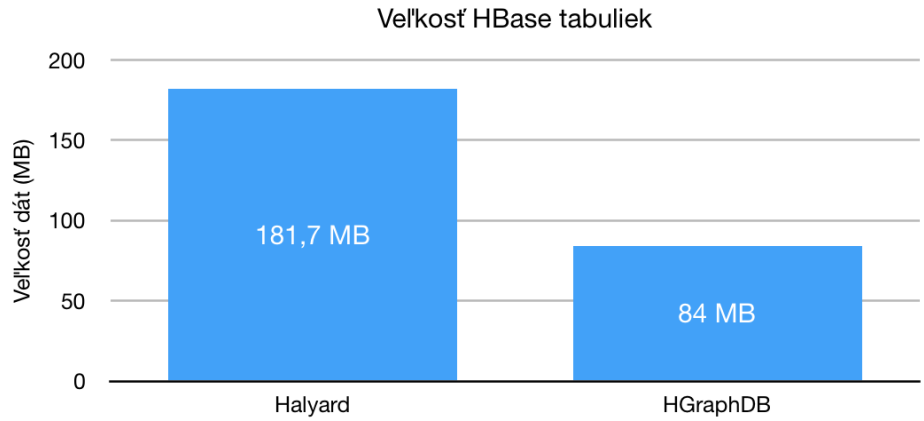
Vstupom toho testu bol obsah 64 časových ôs zo sociálnych sietí Facebook a Twitter (124 888 Facebookových príspevkov a 69 792 Tweetov). Vo frameworku Halyard to znamenalo viac než dva milióny RDF trojíc a HGraphDB tieto dáta uložil pomocou pol milióna uzlov a rovnakého množstva hrán.

Výsledok tohto testu je možné vidieť v grafe 6.1. Výsledná hodnota je priemerný čas z troch behov aplikácie pre každý framework. Celkový čas behu aplikácie bol s oboma frameworkami takmer rovnaký. Spôsobené to bolo tým, že najväčší čas zabralo sťahovanie dát. Čas ich uloženia teda vo výsledku zanikol. Výsledok tohto testu znamená, že sťahovanie a ukladanie dát v implementovanej aplikácii nezávisí na implementácii databázového frameworku.

Zaujímavým výstupom tohto testu bolo tiež veľkosť uložených dát. Výsledné veľkosti je možné vidieť na grafe 6.2. Dáta uložené pomocou RDF modelu frameworku Halyard zaberajú na disku takmer dvojnásobné množstvo miesta. RDF model totiž dáta v značnej časti duplikuje. Pre príklad si môžme uviesť Facebook stránku, ktorá obsahuje tri príspevky. V modeli RDF budú tieto dáta uložené ako tri trojice. Samotná stránka sa bude teda v dá-



Obr. 6.1: Časy stiahnutia a uloženia časových ôs zo sociálnych sietí Facebook a Twitter.



Obr. 6.2: Veľkosť uložených dát pomocou rôznych frameworkov.

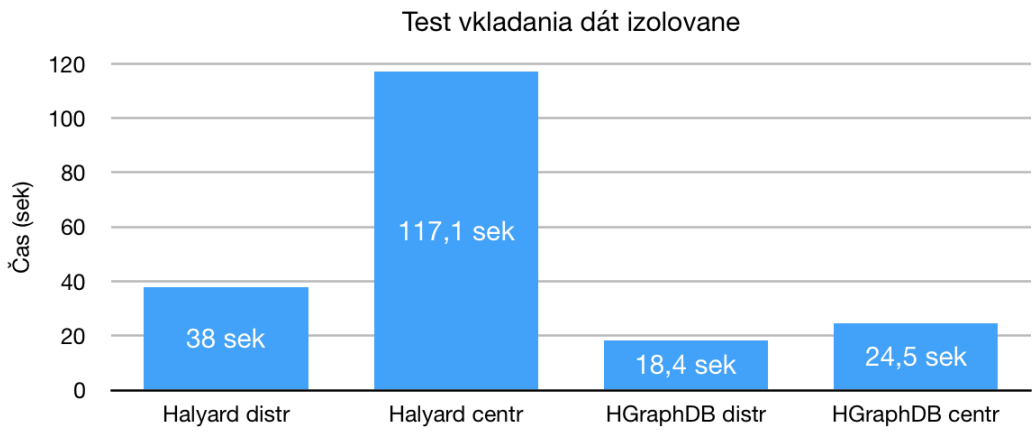
tach nachádzať trikrát. V modeli property graf, ktorý využíva framework HGraphDB budú tieto dáta uložené samostatne. Databáza bude obsahovať štyri uzly a tri hrany. Facebooková stránka sa bude v tomto prípade nachádzať v databáze len raz.

6.2.2 Čistý zápis

Druhý test už testoval schopnosť frameworkov zvládnuť zapisovanie veľkého množstva dát naraz. Stiahnuté dáta sa najprv stiahli a zhromaždili na jednom stroji clusteru. Ak by dáta neboli zhromaždené, framework Spark by sa snažil spustiť ich ukladanie ihneď po stiahnutí. Zhromaždené dáta boli potom opäť distribuované na cluster a ukladané do databázy. Keďže dáta boli k dispozícii v pamäti jedného stroja, pre zaujímavosť som skúsil tieto dáta vložiť do databázy centralizovane. Výhodou centralizovaného vkladania bolo najmä to, že pripojenie do databázy prebiehalo iba raz.

Vzhľadom k tomu, že všetky stiahnuté dáta museli byť držané v pamäti jedného stroja clusteru, veľkosť dát som musel oproti predchádzajúcemu testu obmedziť. Dáta som obmedzil len na 32 časových ôs zo sociálnej siete Facebook. Vo výsledku to predstavovalo 124 888 príspevkov.

Výsledok tohto testu je možné vidieť v grafe 6.3. Framework Halyard je pri distribuovanom vkladaní o niečo pomalší ako HGraphDB. Pomalší čas spôsobila najmä potreba



Obr. 6.3: Časy čistého vkladania dát centralizovane a distribuovane.

transformácie dát na RDF trojice a následne na *KeyValues*. Tento náročnejší výpočet sa na čase prejavil najmä pri vkladaní dát centralizovane pomocou Halyard.

Hoci malo centralizované vkladanie výhodu v tom, že nemuselo pred každým vkladaním časovej osi vytvárať nové databázové spojenie, vo výsledku mu to nepomohlo. Z výsledkov tohto testu vyplýva, že vkladanie dát distribuovane je efektívnejšie.

6.3 Analýza dát

Druhou úlohou aplikácie bolo preveriť efektívnosť databázových riešení pri dopytovaní sa nad dátami. Každý framework mal za úlohu z databázy získať rovnaké informácie a vo výsledku som porovnával čas, za ktorý to stihli vykonať. Pre účely testovania som implementoval niekoľko rôznych databázových dopytov, kde každý mal preveriť iný typ funkcionality. Každý test som potom spustil niekoľkokrát a priemerný výsledný čas som zaznamenal ako výsledok testu.

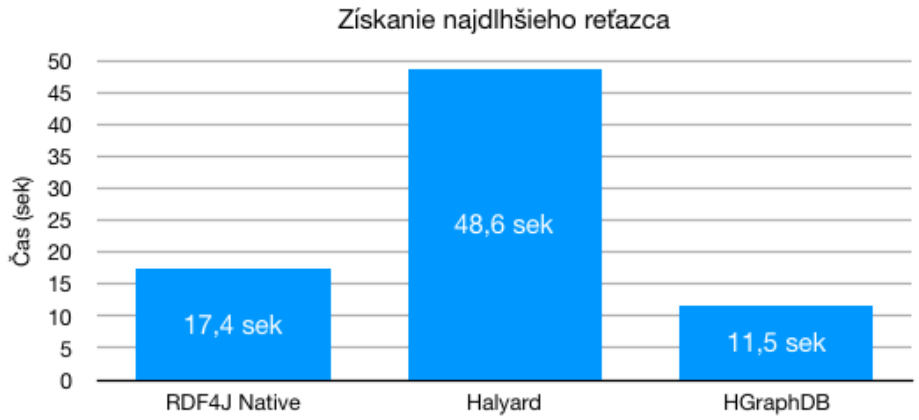
Frameworky pracovali s totožnou množinou dát zo sociálnych sietí. V prípade Halyard, bola veľkosť uložených dát 676 MB, čo predstavovalo 4 369 579 RDF trojíc. Framework HGraphDB uložil tieto dáta pomocou 1 277 882 uzlov a 1 277 772 hrán. Vo výsledku bola veľkosť týchto hrán a uzlov 171 MB. Výhodou HGraphDB bolo už na začiatku to, že pracoval s menšou veľkosťou dát.

Pre porovnanie som každý dopyt vykonal nad centralizovaným natívnym RDF4J úložiskom. Toto úložisko obsahovalo totožné dáta ako framework Halyard a dáta boli uložené na hlavnom uzle clusteru.

6.3.1 Dopyt pre získanie najdlhšieho textového obsahu

Prvý dopyt mal za úlohu získať príspevok s najdlhším textovým obsahom. Výsledok testu je možné vidieť na grafe 6.4.

Najrýchlejší framework bol v tomto prípade HGraphDB. Podobnú rýchlosť mal aj framework RDF4J s natívnym centralizovaným úložiskom. Efektivita frameworku Halyard bola výrazne slabšia. Hoci tento framework obsahuje optimalizácie pre distribuované analyzovanie dát, radenie výsledkov mu spôsobuje problémy. Naproti tomu framework HGraphDB bol



Obr. 6.4: Výsledné časy dopytov pre získanie príspevku s najdlhším reťazcom.

	RDF4J Native	Halyard	HGraphDB
Prvý výsledok bez radenia	0,55 sek	0,77 sek	0,26 sek
Posledný výsledok bez radenia	14,34 sek	54,28 sek	321,65 sek
Prvý výsledok s radením	21,36 sek	66,03 sek	15,01 sek
Posledný výsledok s radením	23,41 sek	68,89 sek	237,42 sek

Obr. 6.5: Výsledné časy dopytov pre získanie časov pridania príspevkov s/bez radenia.

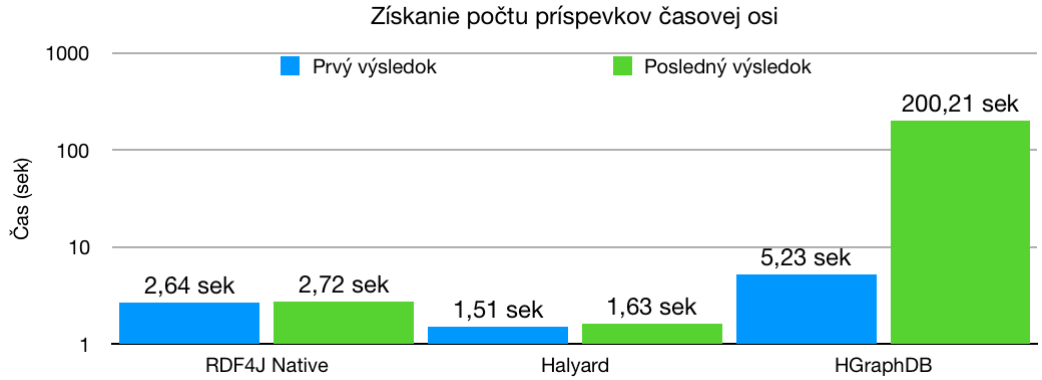
efektívnejší aj napriek faktu, že musel pracovať oproti centralizovanému natívnemu RDF4J úložisku s dátami distribuovanými medzi viacerými samostatnými strojmi.

6.3.2 Dopyty pre získanie času pridania príspevkov

Úlohou ďalšieho dopytu bolo získať z databázy všetky príspevky novšie ako rok 2018 a vypísať ich raz s radením a raz bez radenia. Pri tomto teste bolo vhodné rozdeliť výsledné časy do dvoch skupín. Prvý čas predstavuje hodnotu, ako dlho trvalo získanie prvého záznamu z databázy. Ten druhý zase značí čas potrebný pre získanie všetkých ďalších výsledkov. Oba frameworky totiž všetky dáta nevrátia naraz, ale umožňujú nimi postupne iterovať. Pri získavaní veľkého množstva dát sa tieto dve hodnoty veľmi líšia.

Výsledky testu je možné vidieť v tabuľke 6.5. Získanie prvého výsledku bez radenia bol v podstate len čas potrebný na odfiltrovanie všetkých príspevkov starších ako rok 2018. Z tabuľky je možné vidieť, že tento úkon trval databázam veľmi krátku dobu. Najrýchlejším riešením bol opäť framework HGraphDB, ktorému stačilo vďaka svojmu dátovému modelu filtrovať menšie množstvo dát. Halyard bol oproti svojmu centralizovanému riešeniu znovu pomalší.

Získanie obrovského množstva výsledkov bola už náročnejšia úloha. Pri tomto úkone sa prejavilo náročnejšie čítanie dát, ak sú distribuované nad viacerými strojmi. Centralizované riešenie bolo v tomto prípade niekoľkonásobne efektívnejšie. Hoci bol framework HGraphDB v určitých prípadoch omnoho efektívnejší, čítanie veľkého množstva dát trvalo v jeho prípade rapídne dlhšie.



Obr. 6.6: Výsledné časy dopytu pre získanie počtu príspevkov časových ôs.

Dopyt, ktorý obsahoval aj radenie výsledkov mal podobné výsledky. Získanie prvého záznamu z databázy bolo opäť najrýchlejšie pri frameworku HGraphDB. Výpis zbytku záznamov mu však opäť trvalo niekoľkonásobne dlhšie. Zaujímavým paradoxom pri HGraphDB je to, že daný dopyt bol vo výsledku rýchlejší s radením.

6.3.3 Dopyt pre získanie počtu príspevkov

Ďalší dopyt mal za úlohu získať počet príspevkov každej Facebook stránky, prípadne Twitter účtu. Výstupom tohto dopytu bol teda zoznam časových ôs s počtom ich príspevkov. Dopyt testoval schopnosť databázových riešení zoskupovať dáta podľa otcovského uzlu.

Ako bolo spomínané framework HGraphDB mal s týmto typom dopytu značné problémy. Väčšina implementácii tohto dopytu nepriniesla výsledky v reálnom čase. Nakoniec som musel dopyt prerobiť opačným spôsobom. Najskôr je z databázy vytiahnutý každý uzol typu *Timeline* a vo výsledku sa vyráta počet jeho príspevkov. Za následok to malo opäť menej efektívny výpis všetkých záznamov, čo sa prejavilo na výsledných časoch. Ani získanie prvého výsledku nebolo týmto spôsobom efektívnejšie.

Výsledné časy je možné vidieť v grafe 6.6. Tento dopyt bol vhodný pre framework Halyard. Výpočet počtu príspevkov mohol prebiehať distribuovane, vďaka čomu bolo možné získať výsledky o takmer polovicu rýchlejšie oproti natívnemu RDF4J úložisku.

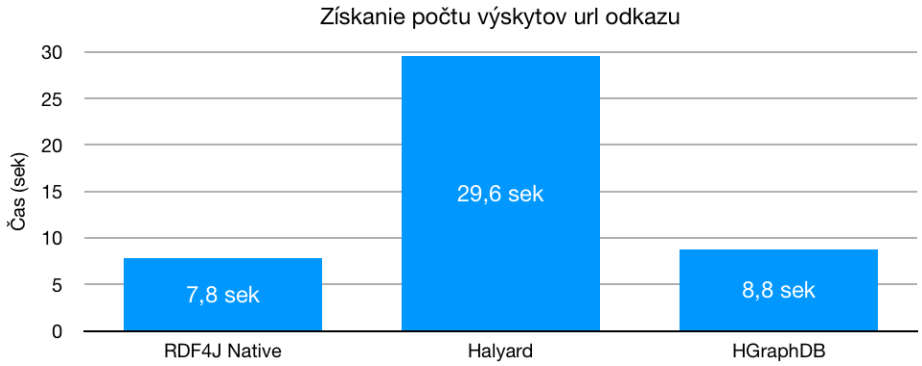
6.3.4 Dopyt pre získanie zdieľaných odkazov

Posledný dopyt získaval z databázy počet výskytov url odkazov. Vypisoval len odkazy, ktoré sa v databáze nachádzajú aspoň dvakrát a vo výsledku ich ešte zoradil podľa výskytu. Oproti predchádzajúcemu dopytu sa tento zameriaval na zoskupovanie výsledkov podľa hodnoty.

Výsledné časy je možné vidieť v grafe 6.7. Framework HGraphDB mal v predošlom prípade značné problémy. Zoskupovanie dát podľa hodnôt mu už nerobí problémy a vďaka rapídne rýchlemu radeniu dosahoval podobné výsledky ako centralizovaná RDF4J databáza. Framework Halyard bol opäť niekoľko násobne pomalší.

6.4 Zhrnutie

Výsledky testov nepriniesli jednoznačný výsledok. Oba frameworky sú niečím špecifické a efektívnejšie v určitých prípadoch. Výber vhodného frameworku by preto mal závisieť na implementovanej aplikácii. Ako bolo ukázané, preniesť dáta na iný typ databázy nie je



Obr. 6.7: Výsledné časy dopytu pre získanie počtu výskotov url odkazov.

vôbec náročný úkon. Dokonca je možné pomerne jednoducho transformovať dáta z modelu RDF do property graf. K dispozícii existuje navyše mnoho frameworkov, ktoré nezávisia nad implementáciou databázy. V tejto práci to boli frameworky RDF4J a TinkerPop.

Pre túto aplikáciu bolo najvhodnejším riešením framework HGraphDB. Exceloval najmä veľkosťou svojich tabuliek, kde k uloženiu dát potreboval približne o tretinu menej miesta na disku. Vo väčšine prípadoch bol navyše omnoho efektívnejší. Dokázal dáta rýchlejšie zapisovať a neskôr sa nad nimi dopytovať. Problém začal mať HGraphDB pri výpise väčšieho množstva dát a pri počítaní uzlov smerujúcich do/z istého uzlu. Oproti frameworku Halyard bol v týchto prípadoch až desať-násobne pomalší.

Framework Halyard je však takisto excelentný nástroj pre prácu s grafovými dátami v distribuovanom prostredí. Jeho hlavnou výhodou bolo najmä jednoduchšie tvorenie databázových dopytov. Tvorenie dopytov je v prípade jazyka SPARQL pomerne jednocestné. Užívateľ nemá veľa možností akými by požadovaný dopyt napísal. Narozdiel od toho, v HGraphDB je možné napísať dopyt viacerými možnými spôsobmi, kde každá obmena výrazne vplýva na jeho celkovú výkonnosť. Pri zápise dát dosahoval Halyard podobné výsledky ako framework HGraphDB.

Vhodným riešením by bolo vo výslednej aplikácii aj centralizované RDF4J úložisko. Hlavnou výhodou tohto riešenia je to, že cluster nemusí obsahovať nakonfigurovanú HBase databázu. Ak by však aplikácia postupne pracovala so stále väčším množstvom dát, v určitej chvíli by nám centralizované riešenie mohlo prestať postačovať. Pre testovaný dátový set bolo však toto úložisko pri dopytovaní sa najefektívnejšie.

Kapitola 7

Záver

Cieľom tejto práce bolo analyzovať možnosti prepojenia grafových databáz s distribuovaným prostredím pre spracovanie rozsiahlych dát na platforme Java. Podrobne som preberal systémy Hadoop, HBase a Spark, ktoré slúžia pre prácu s dátami v distribuovanom prostredí. Zameral som sa na ich vnútorné fungovanie a ich prepojenie s ďalšími systémami. Pri grafových databázach som popísal dvoch reprezentantov databáz fungujúcich na jednom stroji (Neo4j a RDF4J) a tri implementácie distribuovanej grafovej databázy (JanusGraph, Halyard a HGraphDB). Pri distribuovaných databázach som popísal aj ich spôsob prepojenia s Hadoop systémom.

Ďalej som v práci navrhol možné riešenie pre testovanie rôznych databázových distribúcii. Rozhodol som sa, že vhodnými kandidátmi na testovanie budú frameworky Halyard a HGraphDB. Rozdiel medzi nimi je najmä v dátovom modeli, ktorý používajú k ukladaniu dát. Halyard využíva RDF trojice a framework HGraphDB ukladá dáta pomocou property graf modelu. Vo výsledku bolo teda zaujímavé porovnať aj tieto dva dátové modeli. Distribuovanú databázu JanusGraph som do návrhu nezahrnul pretože je stále len v alfa verzii.

Práca ďalej obsahuje návrh a popis implementácie aplikácie pre testovanie týchto dvoch frameworkov. Úlohou výslednej aplikácie je sťahovanie dát zo sociálnych sietí Facebook a Twitter, a ich následné ukladanie pomocou jednotlivých databázových riešení. Tieto dáta predstavujú príspevky Facebookových stránok a Tweety pridané na rôznych Twitter účtoch. Aplikácia dokáže sťahovať len verejne prístupné príspevky a Tweety. Sťahovanie a zápis dát prebieha distribuovane na Spark clusteri. Aplikácia umožňuje na základe vstupného argumentu využiť k zápisu dát framework Halyard alebo HGraphDB. Vďaka tomu je ju možné spustiť dvakrát za sebou s rôznym databázovým frameworkom a porovnať časy behu aplikácie. Čas potrebný pre stiahnutie príspevku bol výrazne väčší oproti času potrebného pre jeho zápis. Keďže je výsledná práca zameraná na efektivitu databázových riešení, aplikácia umožňuje dáta zapísať tiež pomocou oboch frameworkov a to až po stiahnutí posledného príspevku. V tomto prípade získame z aplikácie čistý čas zápisu dát, bez času potrebného pre ich stiahnutie.

Výsledná aplikácia ďalej obsahuje niekoľko rôznych databázových dopytov, ktorých úloha je analyzovať efektivitu frameworkov dopytovania sa nad dátami. Tieto dopyty testovali ich schopnosť dáta radiť podľa hodnoty, radiť podľa funkcie, zoskupovať dáta, rátať počet výskytov, filtrovať podľa hodnoty, filtrovať podľa funkcie a získavať z databázy väčšie množstvo dát. Pre tieto účely obsahuje aplikácia implementáciu piatich rôznych databázových dopytov. Tieto dopyty su implementované zvlášť pomocou frameworku Halyard a zvlášť pomocou HGraphDB.

Posledná kapitola práce obsahuje dosiahnuté výsledky. Zápis dát je efektívnejší pomocou frameworku HGraphDB. Tento framework vďaka modelu property graf ukladá dáta trikrát efektívnejšie. Vzniknuté tabuľky majú teda oproti Halyard omnoho menšiu veľkosť. Efektivita dopytovania sa už takáto jednoznačná nebola. Framework HGraphDB exceluje pri dopytoch, ktorých výstupom je menšie množstvo dát. Pri výpise obrovského množstva dát bol niekedy aj desať-násobne pomalší ako framework Halyard. Ďalším problémom frameworku HGraphDB boli dopyty, ktoré vyžadovali dáta zoskupovať podľa otcovského/synovského uzlu. V tomto prípade bolo náročné dopyt navrhnúť tak, aby som získal výsledok v reálnom čase. Výsledný čas dopytu bol aj tak oproti Halyard výrazne pomalší.

Ďalším zameraním práce by mohlo byť analyzovanie novo vzniknutého frameworku JanusGraph, ktorý je stále len v alfa verzii. Práca s týmto frameworkom je možná pomocou TinkerPop frameworku, takže by bolo možné využiť totožné implementácie dopytov ako v prípade HGraphDB. Zaujímavá by bola tiež analýza databázy Neo4J. Je to centralizovaná databáza, ktorá využíva k ukladaniu dát rovnaký dátový model ako HGraphDB. Vo výsledku by teda bolo možné porovnať, či je framework HGraphDB efektívnejší oproti svojmu podobnému centralizovanému riešeniu.

Literatúra

[1] Apache HBase™ Reference Guide. 2017, [Online; navštívené 07.01.2018].
URL https://hbase.apache.org/book.html

[2] Apache Spark™ - Lightning-Fast Cluster Computing. 2017, [Online; navštívené 09.01.2017].
URL https://spark.apache.org

[3] Hadoop - Apache Hadoop 2.9.0. 2017, [Online; navštívené 30.12.2017].
URL http://hadoop.apache.org/docs/r2.9.0/

[4] HGraphDB - HBase as a TinkerPop Graph Database. 2017, [Online; navštívené 14.01.2018].
URL https://github.com/rayokota/hgraphdb

[5] Eclipse RDF4J Documentation. 2018, [Online; navštívené 12.01.2018].
URL http://docs.rdf4j.org

[6] Halyard Documentation. 2018, [Online; navštívené 13.01.2018].
URL https://merck.github.io/Halyard/getting-started.html

[7] JanusGraph Documentation. 2018, [Online; navštívené 13.01.2018].
URL http://docs.janusgraph.org/

[8] The Noe4j Graph Platform. 2018, [Online; navštívené 11.01.2018].
URL https://neo4j.com

[9] RDF Primer. 2018, [Online; navštívené 12.01.2018].
URL https://www.w3.org/TR/rdf-primer/

[10] TinkerPop3 Documentation. 2018, [Online; navštívené 14.01.2018].
URL http://tinkerpop.apache.org/docs/3.3.1/reference/

[11] George, L.: *HBase: The Definitive Guide*. O'Reilly Media, 2011, ISBN 978-1-449-39610-7.

[12] Holden Karau, P. W. M. Z., Andy Konwinski: *Learning Spark*. O'Reilly Media, 2015, ISBN 978-1-449-35862-4.

[13] Holmes, A.: *Hadoop in Practice*. Manning, 2012, ISBN 978-1-617-29023-7.

[14] Ian Robinson, E. E., Jim Webber: *Graph Databases*. O'Reilly Media, 2013, ISBN 978-1-449-35626-2.

[15] Kemper, C.: *Beginning Neo4j*. Aperss, 2015, ISBN 978-1-4842-1227-1.

[16] Lam, C.: *Hadoop in Action*. Manning, 2010, ISBN 978-1-935-18219-1.

[17] Nick Dimiduk, A. K.: *HBase in Action*. Manning, 2013, ISBN 978-1-617-29052-7.

[18] Vicknair, C.; Macias, M.; Zhao, Z.; aj.: A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM, 2010, ISBN 978-1-4503-0064-3, s. 1–6.

[19] Vukotic, A.; Watt, N.; Abedrabbo, T.; aj.: *Neo4J in Action*. Manning Publications Co., 2014, ISBN 978-1-617-29076-3.

[20] White, T.: *Hadoop: The Definitive Guide*. O'Reilly Media, 2009, ISBN 978-0-596-52197-4.

Príloha A

Obsah priloženého pamäťového média

`/xtutko00-dp.pdf` elektronická verzia textovej správy
`/sources/text` priečinok obsahujúci zdrojový kód textovej správy
`/sources/socializer` priečinok obsahujúci zdrojové súbory výslednej aplikácie
`/sources/socializer/README.md` návod na použitie aplikácie
`/data/sources.txt` súbor obsahujúci zoznam Facebookových stránok a Twitter účtov použitých pri testovaní aplikácie (môže byť použitý ako vstup aplikácie)
`/data/socializer-1.0-jar-with-dependencies.jar` balíček obsahujúci potrebnú konfiguráciu a závislosti pre fungovanie v testovacom prostredí NES