

Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Velkoobjemové úložiště emailů

Bc. Patrik Lenárt

Vedúci práce: Ing. Jan Šedivý, CSc.

Študijný program: Otvorená informatika, Magisterský

Odbor: Softwarové inžinierstvo

13. mája 2011

Podakovanie

Rád by som poďakoval vedúcemu práce pánovi Ing. Janovi Šedivému, CSc. za konzultácie, cenné rady, pripomienky a návrhy, ktoré mi ochotne poskytol počas vypracovávania tejto práce. Taktiež sa chcem sa poďakovať svojim najbližším, bez ktorých podpory by táto práca nevznikla. Poďakovanie ďalej patrí Mgr. Jánovi Slaninkovi a Mgr. Ondrejovi Šterbákovi za ich cenné rady a korektúru textu.

Prehlásenie

Prehlasujem, že som svoju diplomovú prácu vypracoval samostatne a použil som iba podklady uvedené v priloženom zozname.

Nemám závažný dôvod proti užitiu tohto školského diela v zmysle §60 Zákona č. 121/2000 Sb., o autorskom práve, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon).

V Prahe dňa 10. 5. 2011

.....

Abstract

The aim of this thesis is to design an email archiving solution using a distributed database system, commonly known as NoSQL. We described the basic concepts that are used by distributed database systems. We made a strong analysis of requirements for the high capacity and fault tolerant storage of email messages, featuring attachment deduplication to provide an efficient space saving. We identified main criteria for high level comparison of modern NoSQL systems, then we chose two possible candidates that comply with requirements. Functions and architectures of Cassandra and HBase systems were described, subjects were thoroughly tested performance wise. We modeled and implemented the application prototype using Cassandra.

Abstrakt

Cieľom práce je návrh riešenia pre archiváciu elektronickej pošty s využitím distribuovaných databázových systémov, označovaných pod spoločným názvom NoSQL. V práci sú popísané základné koncepty, ktoré sa využívajú pri tvorbe distribuovaných databázových systémov. Analyzovali sme požiadavky na veľkoobjemový a vysokodostupný archív emailových správ, ktorý bude z dôvodu úspory diskového priestoru využívať deduplikáciu príloh. Stanovili sme základné kritéria na porovnanie NoSQL systémov, z ktorých sme následne vybrali dvoch kandidátov, spĺňajúcich požiadavky systému. Popísali sme funkcionality systémov Cassandra a HBase, ktoré sme podrobili výkonnostným testom. Navrhli sme model a implementovali prototyp aplikácie využívajúci systém Cassandra.

Obsah

1	Úvod	1
2	Databázové systémy	3
2.1	História	3
2.2	Distribúované databázové systémy	4
2.3	ACID	4
2.4	Škálovanie databázového systému	5
2.4.1	Replikácia	6
2.4.2	Rozdeľovanie dát	7
2.5	BASE	8
2.6	CAP	9
2.6.1	Konzistencia verzus dostupnosť	10
2.7	Eventuálna konzistencia	11
2.7.1	Konzistencia z pohľadu klienta	11
2.7.2	Systémová konzistencia	12
2.8	MapReduce	13
3	Definícia problému	17
3.1	Archivácia elektronickej pošty	17
3.2	Požiadavky na systém	18
3.2.1	Funkcionálne požiadavky	18
3.2.2	Nefunkcionálne požiadavky	20
4	NoSQL	23
4.1	Dátové modely	23
4.1.1	Relačný model	24
4.1.2	Kľúč-hodnota	24
4.1.3	Stĺpcovo orientovaný model	25
4.1.4	Dokumentový model	25
4.1.5	Grafový model	25
4.2	Porovnanie NoSQL systémov	26
4.2.1	Dátový a dotazovací model	26
4.2.2	Škálovateľnosť a schopnosť odolávať chybám	26
4.2.3	Elastickosť	28
4.2.4	Konzistencia dát	28

4.2.5	Prostredie behu	28
4.2.6	Bezpečnosť	29
4.3	Výber NoSQL systémov	29
5	Cassandra	31
5.1	Dátový model	31
5.2	Rozdeľovanie dát	32
5.3	Replikácia	33
5.4	Členstvo uzlov v systéme	34
5.5	Zápis dát	34
5.6	Čítanie dát	34
5.7	Zmazanie dát	35
5.8	Konzistencia	35
5.9	Perzistentné úložisko	35
5.10	Bezpečnosť	35
6	HBase	37
6.1	Dátový model	38
6.2	Rozdeľovanie dát	39
6.3	Architektúra systému	39
6.4	Replikácia	39
6.5	Zápis dát a čítanie dát	40
6.6	Zmazanie dát	40
6.7	Konzistencia	40
6.8	Perzistentné úložisko	40
6.9	Bezpečnosť	41
7	Testovanie výkonnosti	43
7.1	Testovacie prostredie	43
7.2	Popis testovacej metodológie	44
7.2.1	Testovací klient	44
7.2.2	Testovací prípad pre zápis dát	44
7.2.3	Testovací prípad pre čítanie dát	44
7.2.4	Záťažový test	45
7.3	HDFS	45
7.4	HBase	46
7.5	Cassandra	47
7.6	Voľba databázového systému	49
8	Návrh systému	51
8.1	Zdroj dát	51
8.2	Analýza dát	52
8.3	Databázová schéma	53
8.4	Fulltextové vyhľadávanie	55
8.5	Implementácia	56
8.5.1	Klient	57

8.5.2	Výpočet štatistík	58
8.5.3	Webové rozhranie	58
8.6	Overenie návrhu	58
8.7	Doporučenie najvhodnejšieho systému	59
9	Záver	61
A	Zoznam použitých skratiek	67
B	UML diagramy	69
C	Obsah priloženého DVD	71

Zoznam obrázkov

3.1	Približná distribúcia emailových správ. $\mu = 301$ kB, $\sigma = 1.3$ MB	19
4.1	Pozícia dátového modelu z pohľadu škálovania podľa veľkosti a komplexnosti. Zdroj: [17]	27
4.2	Rozdelenie databázových systémov podľa CAP, Zdroj: [31]	28
8.1	Obsah obálky z programu Qmail-scanner	52
8.2	Databázová schéma	54
8.3	JSON schéma pre fulltextové vyhľadávanie	55
8.4	Architektúra Celery, Zdroj: [44]	56
8.5	Spracovanie emailu	57
8.6	Programová ukážka v jazyku Pig	58
8.7	Brisk, Zdroj: [12]	60
B.1	Diagram nasadenia aplikácie	69
C.1	Výpis priloženého DVD	71

Zoznam tabuliek

2.1	Vstupné dáta pre funkciu <i>map</i>	15
4.1	Reprezentácia dát v relačnom modeli	24
4.2	Reprezentácia dát v stĺpcovom modeli	25
4.3	Stručný prehľad vlastností stĺpcovo orientovaných systémov NoSQL	30
7.1	Priepustnosť pri zápise dát na HDFS	45
7.2	Hbase: zápis riadkov o veľkosti 1000 B	46
7.3	HBase: čítanie riadkov o veľkosti 1000 B	47
7.4	Hbase: maximálna priepustnosť klastra v MB/s	47
7.5	Cassandra: Zápis riadkov o veľkosti 1000 B	48
7.6	Cassandra: Čítanie riadkov o veľkosti 1000 B	49
7.7	Cassandra: maximálna priepustnosť klastra v MB/s	49

Kapitola 1

Úvod

S neustálym rozvojom informačných technológií súčasne narastá objem informácií, ktoré je potrebné spracúvať. Tento fakt podnietil vznik databázových systémov, ktoré slúžia na organizáciu, uchovávanie a spracovanie dát. V dnešnej dobe existuje veľké množstvo databázových systémov, ktoré sa navzájom líšia napríklad architektúrou, dátovým modelom alebo výrobcom.

Od začiatku sedemdesiatych rokov 20. storočia sú v tejto oblasti dominantou relačné databázové systémy (Relational Database Management Systems). Z dôvodu rapídneho rastu dát v digitálnom univerze [23] začínajú byť tieto systémy nepostačujúce. Medzi hlavné faktory pre výber relačného databázového systému doposiaľ patrili výrobca, cena a pod. Vznikajúce moderné webové aplikácie (napríklad sociálne siete) požadujú od týchto systémov vlastnosti ako vysoká dostupnosť, horizontálna rozšíriteľnosť a schopnosť pracovať s obrovským objemom dát (PB, petabajt¹). Novo vznikajúce databázové systémy, spĺňajúce tieto požiadavky sa spoločne označujú pod názvom NoSQL (Not Only SQL). Pri ich výbere je dôležité porozumenie architektúry, dátového modelu a dát, s ktorými budú tieto systémy pracovať.

Táto práca si kladie za cieľ viacero úloh a je rozdelená do dvoch logických častí. V kapitole 2 popisujeme koncepty využívané pri tvorbe distribuovaných databázových systémov, ktoré zabezpečujú vysokú dostupnosť, spoľahlivosť a škálovateľnosť. Kapitola 3 definuje požiadavky na systém, schopný archivovať milióny emailových správ a tieto dáta ďalej spracúvať. Prehľad systémov NoSQL a kritéria pre ich porovnanie popisuje kapitola 4, ktorá v závere doporučuje výber dvoch vhodných kandidátov, systém Cassandra a HBase. Architektonické princípy a vlastnosti, z ktorých tieto systémy vychádzajú, sú popísané pre systém Cassandra v kapitole 5 a pre HBase v kapitole 6. V druhej časti práce sme v kapitole 7 opísali výkonnostné porovnanie týchto dvoch systémov zamerané na operáciu zápisu dát. Návrh modelu systému, výber vhodných nástrojov pre implementáciu prototypu aplikácie emailového úložiska a samotnú implementáciu popisuje kapitola 8. Záverečná časť tejto kapitoly popisuje dosiahnuté výsledky a doporučuje vhodný systém pre riešenie danej úlohy.

¹1PB = 10¹⁵ B

Kapitola 2

Databázové systémy

V tejto časti stručne popíšeme históriu vzniku databázových systémov. Identifikujeme problémy spojené so škálovaním relačných databázových systémov a uvedieme možné spôsoby ich riešenia. Ďalej popíšeme základné koncepty využívané pri tvorbe distribuovaných databázových systémov a techniku MapReduce, ktorá slúži na paralelné spracovanie veľkého objemu dát (PB).

2.1 História

V polovici šesťdesiatych rokov 20. storočia bol spoločnosťou IBM vytvorený informačný systém IMS (Information Management System), využívajúci hierarchický databázový model. IMS je po rokoch vývoja využívaný dodnes. Po krátkej dobe, v roku 1970, publikoval zamestnanec IBM, Dr. Edgar F. Codd článok pod názvom „A Relational Model of Data for Large Shared Data Banks“ [11], ktorým uviedol relačný databázový model. Prvým databázovým systémom implementujúcim tento model bol System R od IBM. Systém používal jazyk pod názvom SEQUEL, ktorý je predchodca dnešného SQL (Structured Query Language) slúžiaceho na manipuláciu a definíciu dát v relačných databázových systémoch. Tento koncept sa stal základom pre relačné databázové systémy, ktoré vďaka širokej škále vlastností (ako napríklad podpora transakcií a dotazovací jazyk SQL) patria v dnešnej dobe medzi najpoužívanéjšie riešenia na trhu.

V minulosti boli objem dát, s ktorým tieto systémy pracovali menší a výkon hardvéru mnohonásobne nižší. Dnes napriek tomu, že výkon procesorov a veľkosť pamäťových zariadení rapídne stúpa, je najväčšou slabinou počítačových systémov rýchlosť prenosu dát medzi pevným diskom a operačnou pamäťou. Tento fakt je kritický pre novovznikajúce webové aplikácie ako napríklad sociálne siete alebo *cloudové systémy*, ktoré majú neustále vyššie nároky na spracovávaný objem dát v reálnom čase a vyžadujú podporu škálovania, ktorá zabezpečuje vysokú dostupnosť a spoľahlivosť. Tieto požiadavky sa snažia efektívne riešiť novovznikajúce distribuované systémy pod spoločným názvom NoSQL, ktoré sú popísané v štvrtej kapitole tejto práce.

2.2 Distribuované databázové systémy

Distribuovaný databázový systém je tvorený pomocou viacerých samostatne operujúcich databázových systémov, ktoré nazývame uzly a ich komunikácia sa vykonáva prostredníctvom počítačovej siete. Užívateľovi alebo aplikácii sa javia ako jeden celok [31]. Podľa konfigurácie jednotlivých uzlov v systéme ich ďalej delíme na:

- distribuované databázové systémy typu „master-slave“
- decentralizované distribuované databázové systémy

Systémy master-slave

Táto konfigurácia obsahuje uzol *master*, ktorý plní jedinečnú úlohu a všetky uzly typu *slave* sú na ňom závislé. V prípade jeho havárie je ohrozená funkčnosť celého systému, nazývame ho kritický bod výpadku (SPOF, single point of failure). Túto konfiguráciu využívajú napríklad relačné databázové systémy.

Decentralizované systémy

Pod názvom decentralizácia sa myslí, že každý uzol v distribuovanom databázovom systéme vykonáva tú istú úlohu a je kedykoľvek nahraditeľný. Táto konfigurácia neobsahuje SPOF.

2.3 ACID

Relačné databázové systémy poskytujú veľkú množinu operácií, ktoré je možné vykonávať nad dátami v nich uloženými. Transakcie [28, 29] sú zodpovedné za korektné vykonanie operácií v prípade, že spĺňajú množinu vlastností ACID. Význam jednotlivých vlastností akronymu ACID je nasledovný:

- *Atomicita* (Atomicity) - zaisťuje, že sa vykonajú všetky operácie reprezentujúce transakciu, čo spôsobí korektný prechod systému do nového stavu. V prípade zlyhania transakcie nemá daná operácia žiaden vplyv na výsledný stav systému a prechod do nového stavu sa nevykoná.
- *Konzistencia* (Consistency) - každá transakcia po svojom úspešnom ukončení garantuje korektnosť svojho výsledku a zabezpečí, že systém prejde z jedného konzistentného stavu do druhého. Konzistentný stav zaručuje, že dáta v systéme odpovedajú požadovanej hodnote. Systém sa musí nachádzať v konzistentnom stave aj v prípade zlyhania transakcie.
- *Izolácia* (Isolation) - operácie, ktoré prebiehajú počas vykonávania jednej transakcie nie sú viditeľné ostatným. Operácie tvoriace transakciu musia mať konzistentný prístup k dátam a to aj v prípade, že u inej transakcie dôjde k jej zlyhaniu.
- *Trvácnosť* (Durability) - v prípade, že bola transakcia úspešne ukončená, systém musí garantovať trvácnosť jej výsledku aj v prípade svojho zlyhania.

Implementácia vlastností ACID, ktoré zaručujú konzistenciu, zvyčajne využíva u relačných databázových systémov metódu zamykania. Transakcia uzamkne dáta pred ich spracovaním a spôsobí ich nedostupnosť až do jej úspešného ukončenia, poprípade zlyhania. Transakcie sú vykonávané sekvenčne. Pre databázový systém, od ktorého požadujeme vysokú dostupnosť tento model nie je vyhovujúci. Zámky spôsobujú stavy, kedy ostatné transakcie musia čakať na ich uvoľnenie. Náhradou je mechanizmus s názvom „riadenie súbežného spracovania s viacerými verziami“ (MVCC, Multiversion concurrency control), ktorý umožňuje paralelné vykonávanie operácií nad dátami, jeho popis obsahuje práca od P. Bernsteina a N. Goodmana [6]. Tento mechanizmus je zároveň využívaný systémami NoSQL.

Transakcie splňujúce vlastnosti ACID využívajú v distribuovaných databázových systémoch dvojfázový potvrdzovací protokol (Two-phase commit protocol [7]). Systém využívajúci tento protokol, zaručuje konzistentnosť aj v prípade výskytu sieťových prerušení (network partitioning) alebo poruchám v systéme. Vlastnosti ACID nekladú žiadnu záruku na dostupnosť systému, naopak stav spôsobujúci nedostupnosť je žiadaný v prípade vykonania operácie, ktorá by mohla spôsobiť nekonzistenciu dát. Systémy garantujúce ACID sú vhodné pre aplikácie, v ktorých sa vykonávajú platobné operácie a pod. Medzi aplikácie, u ktorých sa uprednostňuje dostupnosť dát pre ich konzistenciou patria hlavne moderné webové aplikácie. Pri tvorbe distribuovaných databázových systémov je preto potrebné upustiť z niektorých ACID vlastností. Riešenie poskytuje model pod názvom BASE.

2.4 Škálovanie databázového systému

Obecná definícia pojmu *škálovateľnosť* [8] je náročná bez vymedzenia kontextu, ku ktorému sa vzťahuje. V tejto kapitole budeme pojem škálovateľnosť chápať v kontexte webových aplikácií, ktorých dynamický vývoj kladie na databázové systémy viacero požiadaviek. Medzi hlavné z týchto požiadaviek patrí vysoká dostupnosť, spoľahlivosť a odolnosť systému voči chybám. Definujme škálovateľnosť databázového systému ako vlastnosť, vďaka ktorej je systém schopný spracúvať požiadavky webovej aplikácie v definovanom časovom intervale. Typicky sa táto vlastnosť realizuje pridaním nových uzlov do aktuálneho systému s využitím replikácie. Aplikácia týchto mechanizmov má za následok využitie distribuovaného databázového systému. Škálovateľnosť delíme na vertikálnu, horizontálnu a systému dodáva ďalšie z nasledujúcich vlastností [31]:

- umožňuje zväčšiť veľkosť celkovej kapacity databázy a táto zmena by mala byť transparentná z pohľadu aplikácie na dáta
- zvyšuje celkové množstvo operácií, pre čítanie a zápis dát, ktoré je systém schopný vykonať v danú časovú jednotku
- v určitých prípadoch môže zaručiť, že systém neobsahuje kritický bod výpadku
- zvyšuje celkovú dostupnosť systému

Vertikálna škálovateľnosť je metóda, ktorá sa aplikuje pomocou zvyšovania výkonnosti hardvéru, do systému sa pridáva operačná pamäť, rýchlejšie viacjadrové procesory, zvyšuje sa kapacita diskov. Jednou z nevýhod tohoto riešenia je jeho vysoká cena a možná nedostupnosť

systému v prípade jeho zlyhania. Proces vertikálneho škálovania sa hlavne aplikuje v prípade použitia relačných databázových systémov a obsahuje nasledujúce kroky:

- zámena hardvéru za výkonnejší
- úprava súborového systému (napr. zrušenie žurnálu a uchovávanie informácie o poslednom prístupe k súborom)
- optimalizácia databázových dotazov, použitie indexov
- pridanie vrstvy pre kešovanie (memcached, EHCACHE, atď.)
- denormalizácia dát v databáze, čo má za následok porušenie databázovej normálizácie

V tomto prípade je možné naraziť na výkonnostné hranice bežne dostupného hardvéru a na rad nastupuje horizontálna škálovateľnosť, ktorá je omnoho komplexnejšia. Horizontálnu škálovateľnosť je možné realizovať pomocou využitia replikácie alebo metódou „rozdeľovania dát“ (sharding).

2.4.1 Replikácia

V distribuovaných systémoch má použitie replikácie za následok, že sa daná informácia nachádza na viacerých uzloch¹ tohto systému. Táto technika zvyšuje dostupnosť, spoľahlivosť a odolnosť systému voči chybám. Replikácia nie je určená pre zálohu dát.

V prípade distribuovaného databázového systému sa časť informácií uložených v databáze nachádza na viacerých uzloch. Toto usporiadanie môže napríklad zvýšiť výkonnosť operácií, ktoré pristupujú k dátam a to tak, že dochádza k čítaniu dát z databázy paralelne z viacerých uzlov. V systéme obsahujúcom repliku dát nedochádza k strate informácií v prípade poruchy uzla. Replikácia a propagácia zmien (pridanie alebo odstránenie uzla s replikou) v systéme sú z pohľadu aplikácie transparentné. Použitie replikácie nezvyšuje pridávaním nových uzlov celkovú kapacitu databázy. Problémom tejto techniky je konzistencia dát. Dáta sa zapisujú na viacero fyzicky oddelených uzlov a zmena sa nemusí prejaviť okamžite vo všetkých replikách. Z pohľadu klienta pristupujúceho k replikovaným dátam, môže byť ich obsah nekonzistentný. Medzi metódy pomocou, ktorých je možné zabezpečiť konzistenciu patria:

- *Read one - Write all* - u tejto metódy sa čítanie dát prevedie z ľubovlného uzla obsahujúceho repliku. Zápis dát sa vykoná na všetky uzly s replikou a až v prípade, že každý z nich potvrdí úspech tejto operácie je výsledok považovaný za korektný. Táto metóda nie je schopná pracovať v prípade ak dôjde k prerušeniu sieťového spojenia medzi uzlami alebo v prípade poruchy jedného z uzlov.
- *metóda kvóra* - funguje na princípe hlasovania, podrobnejší popis je v časti [2.7.2](#)

¹Pod pojmom uzol v tomto prípade myslíme samostatný počítačový systém, ktorý je súčasťou distribuovaného systému

V relačných databázových systémoch sa replikácia rieši pomocou architektúry master - slave. Uzol pod názvom master slúži ako jediný databázový stroj, na ktorom sa vykonáva zápis dát a replika týchto dát je následne distribuovaná na zvyšné uzly pod názvom slave. Táto metóda umožňuje mnohonásobne zvýšiť počet operácií, ktoré slúžia pre čítanie dát z databázového systému a v prípade zlyhania niektorého zo systémov máme neustále k dispozícii kópiu dát. Slabinou v tomto systéme je uzol v roli master, ktorý nezvyšuje výkonnosť v prípade operácií vykonávajúcich zápis a zároveň jeho porucha môže spôsobiť celkovú nedostupnosť systému.

Druhým možným riešením je technika „multi - master“, kde každý uzol obsahujúci repliku je schopný zápisu dát a následne tieto zmeny preposiela ostatným. Tento mechanizmus predpokladá distribuovanú správu zamykania a vyžaduje algoritmy pre riešenie konfliktov v prípade nekonzistentných dát.

2.4.2 Rozdeľovanie dát

Rozdeľovanie dát (sharding) je metóda založená na princípe, kde dáta obsiahnuté v databáze rozdeľujeme podľa stanovených pravidiel do menších celkov. Tieto celky môžeme následne umiestniť na navzájom rôzne uzly distribuovaného databázového systému. Táto metóda umožňuje zvýšiť výkonnosť operácií pre zápis a čítanie dát a zároveň pridávaním nových uzlov do systému zvyšuje celkovú kapacitu databáze. V prípade, že architektúra distribuovaného databázového systému je navrhnutá s využitím tejto metódy, zvýšenie výkonu operácií a objem uložených dát sa realizuje automaticky bez nutnosti zásahu do užívateľskej aplikácie.

Techniku rozdeľovania dát môžeme považovať za aplikáciu architektúry známej pod názvom „zdieľanie ničoho“ (shared nothing) [38]. Táto architektúra sa používa pre návrh systémov využívajúcich multiprocesory. V takomto prípade sa medzi procesormi nezdieľa operačná ani disková pamäť. Architektúra zabezpečuje takmer neobmedzenú škálovateľnosť systému a využíva ju mnoho NoSQL systémov ako napríklad Google Bigtable [10], Amazon Dynamo [16] alebo technológia MapReduce [14].

Pri návrhu distribuovaných databázových systémov s využitím tejto techniky patrí medzi kľúčový problém implementácia funkcie spojenia (JOIN) nad dátami, ktorá sa preto neimplementuje. V prípade, že sa, dáta nad ktorými by sme chceli túto operáciu vykonať, nachádzajú na dvoch rozdielnych uzloch prepojených sieťou, takéto spojenie by značne znížilo celkovú výkonnosť systému a viedlo by k zvýšeniu sieťového toku, záťaži systémových zdrojov a možným nekonzistentným výsledkom.

Keďže sa dáta nachádzajú na viacerých uzloch systému, hrozí zvýšená pravdepodobnosť hardvérového zlyhania, poprípade prerušenie sieťového spojenia a preto sa táto technika často kombinuje s pomocou využitia replikácie.

V prípade použitia tejto techniky v relačných databázach, je nutný zásah do logiky aplikácie. Dáta uložené v tabuľkách relačnej databáze zachytávajú vzájomné relácie a týmto spôsobom dochádza k celkovému narušeniu tohto konceptu. Príkladom môže byť tabuľka obsahujúca zoznam zamestnancov, ktorú rozdelíme na samostatné celky. Každá tabuľka bude reprezentovať mená zamestnancov, ktorých priezvisko začína rovnakým písmenom abecedy a zároveň sa bude nachádzať na samostatnom databázovom systéme. Táto technika so sebou

prináša problém, v ktorom je potrebné nájsť vhodný kľúč, podľa ktorého budeme dáta rozdeľovať a zabezpečíme tak rovnomerné zaťaženie uzlov v systéme. Existuje viacero metód, ktoré je možné použiť pre rozdeľovanie dát na úrovni aplikácie [31]:

- segmentácia dát podľa funkcionality - dáta, ktoré je možné popísať spoločnou vlastnosťou ukladáme do samostatných databáz a tieto umiestňujeme na rozdielne uzly systému. Príkladom môže byť samostatný uzol spravujúci databázu pre užívateľov a iný uzol s databázou pre produkty. Túto metódu spracoval Randy Shoup² [37], architekt spoločnosti eBay.
- rozdeľovanie dát podľa kľúča - v dátach identifikujeme kľúč, s ktorého využitím ich rovnomerne rozdelíme. Následne sa na tento kľúč aplikuje hašovacia funkcia a na základe jej výsledku sa tieto dáta umiestňujú na jednotlivé uzly.
- vyhľadávacia tabuľka - jeden uzol v systéme slúži ako katalóg, ktorý určuje, na ktorom uzle sa nachádzajú dané dáta. Tento uzol zároveň spôsobuje zníženie výkonu a v prípade jeho havárie spôsobuje nedostupnosť celého systému (SPOF).

Replikácia a rozdeľovanie dát patria medzi kľúčové vlastnosti využívané v NoSQL systémoch, ktoré popisuje kapitola 4.

2.5 BASE

Akronym BASE bol prvýkrát použitý v roku 1997 na sympóziu SOSP (ACM Symposium on Operating Systems Principles) [18]. Tento model poľavil na požiadavku zodpovednom za konzistenciu dát, ktorý je garantovaný vlastnosťami ACID. BASE tvoria nasledujúce slovné spojenia:

- „*bežne dostupný*“ (Basically Available) - systém je schopný zvládať čiastočné zlyhanie za cenu nižšej komplexity.
- „*zmiernený stav*“ (Soft State) - systém nezaručuje trvácnosť dát s cieľom zvýšenia výkonu.
- „*čiasťochne konzistentný*“ (Eventually Consistent) - je možné na určitú dobu tolerovať nekonzistentnosť dát, ktoré musia byť po uplynutí určitého časového intervalu znovu konzistentné.

Využitím tohto modelu v distribuovanom databázovom systéme sa dosahuje vyššia dostupnosť aj v prípade čiastočného zlyhania alebo sieťového prerušenia. Distribuovaný databázový systém môžeme klasifikovať ako systém spĺňajúci vlastnosti ACID, BASE alebo oboje.

BASE umožňuje horizontálne škálovanie relačných databázových systémov bez nutnosti použitia distribuovaných transakcií. Pre implementáciu tejto techniky môžeme použiť rozdeľovanie dát s metódou segmentácie dát podľa funkcionality [35].

²“If you can’t split, you cant scale it.” – Randy Shoup, Distinguished architect Ebay

Bankomatový systém je príkladom systému obsahujúceho čiastočnú konzistenciu dát. Po vybratí určitej čiastky z účtu, sa korektná informácia o aktuálnom zostatku môže zobraziť až za niekoľko dní, kdežto transakcia ktorá túto zmenu vykonala musí spĺňať vlastnosti ACID. Medzi webové aplikácie, u ktorých sa nepožadujú všetky vlastnosti ACID patria napríklad nákupný košík spoločnosti Amazon³, zobrazovanie časovej osi aplikácie Twitter, poprípade systémy spoločnosti Google⁴ indexujúce obsah webu. Ich nedostupnosť by znamenala obrovské finančné straty (napríklad zlyhanie vyhľadávania pomocou systému Google by znamenalo zobrazenie nižšieho počtu reklám, nedostupnosť nákupného košíka Amazon by spôsobila pokles predaja atp).

Aplikácia vyššie popísaných techník na relačné databázové systémy môže byť netriviálnou úlohou. Relačný model, je spôsob reprezentácie dát, ktorý umožňuje efektívne riešiť určité typy úloh, preto snaha prispôbiť tento model každému problému môže byť neefektívna. V tomto prípade, môžeme uvažovať alternatívne riešenia, medzi ktoré patria systémy NoSQL.

2.6 CAP

Moderné webové aplikácie kladú na systémy požiadavky, medzi ktoré patrí vysoká dostupnosť, zmiernená konzistencia dát a schopnosť odolávať chybám. Dr. Brewer v roku 2000 nastolil myšlienku, dnes známu pod názvom teória CAP [9]. U distribuovaných databázových systémov, ktoré používajú pre vzájomnú komunikáciu sieť musíme predpokladať s prítomnosťou sieťových prerušení. Táto teória tvrdí, že u takýchto systémov je možné súčasne dosiahnuť len dvojicu z vlastností CAP a to CP alebo AP. V roku 2002 platnosť tejto teórie pre asynchrónnu sieť matematicky dokázali Lynch a Gilbert [27]. Modelu asynchrónnej siete svojimi vlastnosťami zodpovedá Internet. Akronym CAP tvoria nasledujúce vlastnosti:

- *Konzistencia* (Consistency) - distribuovaný systém je v konzistentnom stave, ak každý jeho uzol v prípade požiadavku dát vracia tú istú odpoveď.
- *Tolerancia chýb* (Partition Tolerance) - uzly distribuovaného systému navzájom komunikujú pomocou siete, v ktorej hrozí strata správ. V prípade vzniku sieťového prerušenia dané uzly medzi sebou navzájom nedokážu komunikovať. Táto vlastnosť podľa definície (viď. Gilbert a Lynch) tvrdí, že v prípade vzniku zlyhania sieťovej komunikácie medzi niektorými uzlami, musí byť systém schopný naďalej pracovať korektne. V reálnych podmienkach neexistuje distribuovaný systém, ktorého uzly na vzájomnú komunikáciu využívajú sieť a nedochádza pri tom k strate správ, teda k poruchám sieťovej komunikácie.
- *Dostupnosť* (Availability) - distribuovaný systém je dostupný, ak každý jeho uzol, ktorý pracuje korektne, je schopný pri prijatí požiadavku zaslať odpoveď. V spojení s toleranciou chýb, táto vlastnosť hovorí, že v prípade ak nastane sieťový problém⁵, každá požiadavka bude vykonaná.

³<http://www.amazon.com>

⁴<http://www.google.com>

⁵týmto sa nemyslí porucha uzla

Pravdepodobnosť, že dôjde k zlyhaniu ľubovoľného uzla v distribuovanom systéme, exponenciálne narastá s počtom pribúdajúcich uzlov.

$$P(A) = 1 - P(B)^{\text{počet uzlov}}$$

$P(A)$ - pravdepodobnosť zlyhania ľubovoľného uzla

$P(B)$ - pravdepodobnosť, že individuálny uzol nezlyhá

2.6.1 Konzistencia verzus dostupnosť

V distribuovanom systéme nie je možné súčasne zaručiť vlasnosť konzistencie a dostupnosti. Ako príklad si predstavme distribuovaný systém obsahujúci tri uzly A, B, C, ktorý zaručuje obe vlastnosti aj v prípade vzniku sieťového prerušenia. Na všetkých uzloch sa nachádzajú identické (replikované) dáta. Ďalej uvažujme, že došlo k sieťovému prerušeniu, ktoré rozdelilo uzly na dva samostatné celky $\{A,B\}$ a $\{C\}$. V prípade, že uzol C obdrží požiadavku pre zmenu dát má na výber z dvoch možností:

1. vykonať zmenu dát čo spôsobí, že sa uzly A a B o tejto zmene dozvedia až vo chvíli ak bude sieťové spojenie obnovené
2. zamietnuť požiadavku na zmenu dát, z dôvodu že uzly A a B sa o tejto zmene nedozvedia

V prípade výberu možnosti číslo 1 zabezpečíme neustálu dostupnosť systému, naopak v prípade možnosti číslo 2 jeho konzistenciu. Nie je možný súčasný výber oboch možností.

CP

Ak od daného systému tolerujúceho sieťové prerušenia požadujeme konzistenciu na úkor dostupnosti jedná sa o alternatívu CP. Takýto systém zabezpečí konzistentnosť operácií pre zápis a čítanie dát a zároveň sa môže stať, že na určité požiadavky nebude schopný reagovať (možnosť číslo 2). Medzi takéto systémy môžeme zaradiť distribuovaný databázový systém Google Bigtable alebo systém využívajúci dvojfázový potvrdzovací protokol.

AP

V prípade, že poľavíme na požiadavku konzistencie tak takýto systém bude vždy dostupný aj napriek sieťovým prerušeniam. V tomto prípade sa jedná o alternatívu AP. Je možné, že v takomto systéme bude dochádzať ku konfliktným zápisom alebo operácie čítania budú po určitú dobu vracaať nekonzistentné výsledky. Tieto problémy s konzistenciou sa v distribuovaných databázových systémoch riešia napríklad pomocou metódy „vektorových hodín“ (Vector clock) [16] alebo na aplikačnej úrovni na strane klienta. Príkladom systému patriaceho do tejto kategórie je Amazon Dynamo.

CA

Ak systém nebude tolerovať sieťové prerušenia, tak bude splňovať požiadavok konzistencie a dostupnosti, varianta CA. Jedná sa o nedistribuované systémy pracujúce na jednom fyzickom hardvéri využívajúce databázové transakcie. Tieto vlastnosti splňuje napríklad relačný databázový systém MySQL.

Pri výbere distribuovaného databázového systému, môžeme vďaka vyššie popísaným vlastnostiam určiť vhodnosť jeho použitia, na základe požiadavkov aplikácie.

2.7 Eventuálna konzistencia

V distribuovaných systémoch sa pod pojmom konzistencie v ideálnych podmienkach rozumie vlastnosť, ktorá zaručí, že zmena dát (zápis alebo aktualizácia dát) sa prejaví súčasne s rovnakým výsledkom. Konzistencia je zároveň úzko spojená s replikáciou. Väčšina NoSQL systémov poskytuje čiastočnú konzistenciu, poprípade dáva možnosť výberu medzi vlastnosťami CP a AP (napríklad systém Cassandra⁶). V nasledujúcej časti popíšeme rôzne druhy konzistencie.

V predchádzajúcom texte sme už spomínali, že v dnešnej dobe existuje mnoho aplikácií, u ktorých je možné poľaviť na požiadavku konzistencie. Ak sa určitá zmena prejaví s miernym oneskorením funkčnosť systému nebude v tomto prípade ohrozená. Táto konzistencia nie je totožná s konzistenciou definovanou u vlastností ACID, kde ukončenie transakcie zaručuje, že sa systém nachádza v konzistentnom stave. Na konzistenciu sa môžeme pozeráť z dvoch pohľadov. Prvým, je klientský pohľad na strane zadávateľa problému resp. programátora, ktorý rozhodne aká je závažnosť zmien, ktoré sa budú vykonávať v systéme. Druhý pohľad je systémový, zabezpečuje technické riešenie a implementáciu techník zodpovedných za správu konzistencie v distribuovaných databázových systémoch.

2.7.1 Konzistencia z pohľadu klienta

Pre potrebu nasledujúcich definícií uvažujme distribuovaný databázový systém, ktorý tvorí úložisko dát a tri nezávislé procesy {A, B, C}, ktoré môžu v danom systéme zmeniť (vykonať zápis) a načítať hodnotu dátovej jednotky. Na základe toho ako jednotlivé nezávislé procesy pozorujú zmeny v systéme delíme konzistenciu na [41]:

Silná konzistencia (Strong consistency) - proces A vykoná zápis. Po jeho ukončení je nová hodnota dátovej jednotky dostupná všetkým procesom {A, B, C}, ktoré k nej následne pristúpia (vykonajú operáciu čítania). Túto konzistenciu zabezpečujú transakcie s vlastnosťami ACID.

Slabá konzistencia (Weak consistency) - proces A vykoná zápis novej hodnoty do dátovej jednotky. V takomto prípade systém negarantuje, že následne pristupujúce procesy {A, B, C} k tejto jednotke vrátia hodnotu zapísanú procesom A. Definujeme pojem „nekonzistentné

⁶<http://cassandra.apache.org>

okno“, ktoré zabezpečí, že po uplynutí stanovenej časovej doby sa táto nová hodnota dátovej jednotky prejaví vo všetkých procesoch, ktoré k nej pristúpia.

Eventuálna konzistencia (Eventual consistency) - je to špecifická forma slabej konzistencie. V tomto prípade systém garantuje, že ak sa nevykoná žiadna nová zmena hodnoty dátovej jednotky, po určitom čase budú všetky procesy prístupujúce k tejto jednotke schopné vrátiť jej korektnú hodnotu. Tento model má viacero variácií, niektoré z nich popíšeme v nasledujúcej časti textu.

Model čiastočnej konzistencie má viacero variácií:

Read-your-write consistency - v prípade, že proces A zapíše novú hodnotu do dátovej jednotky, žiadny z jeho nasledujúcich prístupov k tejto jednotke nevráti staršiu hodnotu ako naposledy zapísaná.

Session consistency - v tomto prípade prístupuje proces k systému v kontexte relácií. Po dobu trvania relácie platí predchádzajúci typ konzistencie. V prípade zlyhania relácie sa vytvorí nová, v ktorej môže systém vracaať hodnotu dátovej jednotky, zapísanú pred vznikom predchádzajúcej relácie.

Monotonic read consistency - v prípade, že proces načítal hodnotu dátovej jednotky, tak pri každom nasledujúcom prístupe nemôže vrátiť predchádzajúcu hodnotu dátovej jednotky.

Tieto typy konzistencie je možné navzájom kombinovať a ich hlavným cieľom je zvýšiť dostupnosť distribuovaného systému na úkor toho, že poľavíme na požiadavkách konzistencie. Príkladom systému s čiastočnou konzistenciou je asynchrónna replikácia v relačnom databázovom systéme využívajúca architektúru master-slave.

2.7.2 Systémová konzistencia

Techniky založené na protokoloch kvóra (Quorum-based protocols [26]) je možné použiť pre zvýšenie dostupnosti a výkonu v distribuovaných databázových systémoch s garanciou silnej alebo čiastočnej konzistencie. Definujme nasledujúcu terminológiu:

- N - počet uzlov, ktoré obsahujú repliku dát
- W - počet uzlov obsahujúcich repliku, na ktorých sa musí vykonať zápis, aby bola zmena úspešne potvrdená
- R - počet uzlov s replikou, ktoré musia vrátiť hodnotu dátového objektu v prípade operácie čítanie

V prípade, že platí $W + R > N$, operácie pre zápis a čítanie dát sa stále prekrývajú minimálne na jednom uzle, ktorý bude vždy obsahovať aktuálnu hodnotu danej operácie. Tento prípad, zabezpečuje silnú konzistenciu v systéme. V prípade $W + R \leq N$, môže nastať

situácia keď predchádzajúca podmienka neplatí a teda daná operácia je čiastočne konzistentná. Rôzna konfigurácia týchto parametrov zabezpečí rozdielnu dostupnosť a výkonnosť distribuovaného systému. Uvažujme nasledujúce príklady pre $N = 3$.

1. $R = 1$ a $W = N$, tento prípad zabezpečí, že systém bude optimalizovaný pre operácie čítania dát. Klient číta dáta z ľubovoľnej repliky. Operácie budú konzistentné, pretože uzol z ktorého dáta čítame sa prekrýva s uzlami na ktorých vykonávame zápis. Nevýhodou tohoto modelu je, že nedostupnosť jednej repliky znemožní vykonanie zápisu. V prípade systémov, u ktorých požadujeme aby obsluhovali veľký počet požiadavkov pre čítanie sa môže hodnota N pohybovať v stovkách až tisícoch, závisí to od počtu uzlov v systéme.
2. $W = 1$ a $R = N$, tento prípad je vhodný pre systémy u ktorých požadujeme rýchly zápis. Tento model môže spôsobiť stratu dát v prípade zlyhania uzla s replikou, na ktorú bol vykonaný zápis.
3. $W = 2$ a $R = 2$, zabezpečí optimálne nastavenie. Tento prípad je využívaný v aplikáciách spoločnosti Amazon využívajúcich systém Dynamo.

2.8 MapReduce

Nárast diskových kapacít a množstva dát, ktoré na nich ukladáme spôsobuje jeden z ďalších problémov, ktorým je analýza a spracovanie dát. Kapacita pevných diskov sa za posledné roky mnohonásobne zvýšila v porovnaní s dobou prístupu a prenosových rýchlostí pre čítanie a zápis dát na tieto zariadenia.

Pre jednoduchosť uvažujme nasledujúci príklad, v ktorom chceme spracovať pomocou jedného počítačového systému 1 TB dát uložených na lokálnom súborovom systéme, pri priemernej prenosovej rýchlosti diskových zariadení 100 MB/s. Za ideálnych podmienok by čas na prečítanie týchto dát presahoval dve a pol hodiny. V prípade, že by sme 1 TB dát rovnomerne rozdelili na sto počítačov a tieto úseky paralelne spracovali, celková doba by sa znížila za ideálnych podmienok na necelé dve minúty.

Spoločnosť Google v roku 2004 zverejnila programovací model pod názvom MapReduce [14], ktorý slúži na paralelne spracovanie obrovského objemu dát (PB). Model využíva vlastnosti paralelných a distribuovaných systémov, je optimalizovaný pre beh na klastri tvorenom vysokým počtom (tisícky) spotrebných počítačov. Pred programátorom sa snaží zastreť všetky problémy, ktoré prináša paralelizácia výpočtov, poruchovosť systémov, distribúcia dát vzhľadom na ich lokalitu a rovnomerné rozdeľovanie záťaže medzi systémami. MapReduce využíva informácie o lokalite dát, výpočet nad danými dátami sa vykonáva priamo na uzle kde sú umiestnené, čo je hlavnou výhodou v porovnaní s inými aplikáciami pre distribuovaný výpočet (napr. *Grid computing*).

MapReduce nie je vhodný na spracovanie dát v reálnom čase. Je optimalizovaný na dávkový beh. Jeho implementácia spoločnosťou Google, ktorá zároveň využíva distribuovaný súborový systém Google File System (GFS) [24], nie je k dispozícii. V rámci hnutia No-SQL vzniklo open source riešenie pod názvom Hadoop⁷, ktoré implementuje tento model

⁷<http://hadoop.apache.org>

na vlastnom distribuovanom súborovom systéme Hadoop Distributed File System (HDFS). K dispozícii sú aplikácie HIVE alebo PIG, ktoré sú nadstavbou modelu MapReduce a poskytujú vyššiu abstrakciu vo forme jazyka podobného SQL.

Pre použitie tohoto nástroja musí programátor zdefinovať dve funkcie pod názvom *map* a *reduce*. Funkcia *map* na jednotlivých uzloch systému, transformuje vstupné dáta na základe zadaného kľúča k a k nemu prináležiacim hodnotám H na medzivýsledok, ktorý obsahuje nové kľúče g_1, \dots, g_n a k nim odpovedajúce zotriedené hodnoty I . Tieto dáta sa odošlú na uzly vykonávajúce užívateľom definovanú funkciu *reduce*. Funkcia *reduce* vykoná nad hodnotami priradenými ku kľúčom g_1, \dots, g_n požadovanú operáciu, ktorej typickým výsledkom je jedna výsledná hodnota, poprípade viacero hodnôt I . Tieto operácie je možné popísať nasledovne:

```
map(k, H) -> list(g, I)
reduce(g, list(I)) -> list(I)
```

Príklad

Uvažujme vstupné dáta popísané v tabuľke 2.1. Nad dátami chceme spočítať celkovú veľkosť emailových správ, ktoré prináležia jednotlivým doménam. V tomto prípade bude vstupom do funkcie *map* ako kľúč k číslo riadku v tabuľke, H budú tvoriť jednotlivé hodnoty v stĺpcoch odpovedajúce riadku. Aplikáciou funkcie *map*, ktorá v tomto prípade bude zabezpečovať výber hodnôt zo stĺpcov *doména* a *veľkosť* dostaneme nasledujúci výstup:

```
(d1.com, [12, 3])
(d2.net, [65, 80, 102])
(d3.sk, [12])
```

Tieto záznamy následne spracuje funkcia *reduce*, kde kľúč g bude reprezentovaný názvom domény a *list(I)* budú tvoriť jednotlivé veľkosti emailových správ. Výstupom budú dvojice:

```
(d1.com, 15)
(d2.net, 247)
(d3.sk, 12)
```

Dáta obsiahnuté v tabuľke sa môžu nachádzať na ľubovoľnom počte počítačov so systémom Hadoop, ktorý zabezpečí aby na jednotlivých uzloch bola vykonaná funkcia *map* a jej výsledky sa pošlú na náhodné uzly vykonávajúce funkciu *reduce*. Týmto sa zabezpečí plne distribuovaný výpočet bez nutnosti preposielania zdrojových dát, čo môže efektívne znížiť zaťaženie siete a zvýšiť spoľahlivosť výpočtu.

doména	adresa	veľkosť	dátum
d1.com	jan@d1.com	12	02/01/2011
d3.sk	jan@d3.sk	12	02/01/2011
d2.net	jeny@d2.net	65	02/05/2011
d1.com	john@d1.com	3	03/02/2011
d2.net	jenny@d2.net	80	03/06/2011
d2.net	andre@d2.net	102	21/02/2011

Tabuľka 2.1: Vstupné dáta pre funkciu *map*

Kapitola 3

Definícia problému

Množstvo digitálnych informácií každým rokom prudko narastá. Podľa štatistík spoločnosti IDC [23] v roku 2006 dosahovala kapacita digitálneho univerza veľkosť 161 exabajtov¹(EB). Podiel elektronickej pošty (emailov) bez spamu, tvoril 3% z tohoto objemu. Predpoveď na rok 2011 [22] predpokladá celkovú kapacitu 1800 EB, čo je viac ako desaťnásobok nárastu pôvodnej kapacity v období piatich rokov. V rozmedzí rokov 1998 až 2006 sa mal počet schránok elektronickej pošty zvýšiť z 253 miliónov na 1.6 miliardy. Predpoveď IDC ďalej uvádzala, že po ukončení roku 2010 tento počet presiahne hodnotu dvoch miliard. Počas obdobia medzi rokmi 1998 až 2006 celkový počet odoslaných správ elektronickej pošty rástol trikrát rýchlejšie ako počet jej užívateľov, dôvodom tohoto prudkého nárastu bola nevyžiadaná elektronická pošta. Odhaduje sa, že až 85% dát z celkového predpokladaného objemu 1800 EB budú spracovávať, prenášať alebo zabezpečovať organizácie. Napriek tejto explózii digitálnych informácií je potrebné správne porozumieť hodnote týchto dát, nájsť vhodné metódy pre ich ukladanie do pamäti počítačových systémov, ich archiváciu a to tak, aby sme ich mohli ďalej spracúvať a efektívne využiť. Táto kapitola práce si kladie za cieľ analyzovať potreby pre archiváciu elektronickej pošty a definovať požiadavky pre vysokodostupný systém slúžiaci k archivácii emailov.

3.1 Archivácia elektronickej pošty

S narastajúcim objemom dát reprezentujúcim elektronickú poštu je potrebné porozumieť tejto štruktúre a následne tieto dáta vhodne spracovať. Cieľom je ukladať emailové správy tak aby sme dosiahli úsporu diskového priestoru, boli sme nad nimi schopný vykonávať operácie ako fulltextové vyhľadávanie, zber údajov pre tvorbu štatistík alebo umožnili ich opätovné sprístupnenie. Emaily obsahujú čoraz viac obchodných informácií a iný dôležitý obsah, z tohto dôvodu musia byť organizácie všetkých rozmerov schopné uchovávať tento obsah pomocou vhodných archivačných nástrojov. S problémom archivácie zároveň úzko súvisí problém bezpečnosti. Pod pojmom bezpečnosti v tejto oblasti máme na mysli hlavne ochranu proti nevyžiadanej pošte tj. spam, spyware, malware a phishingu. Na boj proti týmto hrozbám využívajú organizácie anti-spamové a anti-vírusové systémy. Možné dôvody prečo archivovať elektronickú poštu sú nasledovné [34]:

¹1EB = 10¹⁸B

- záloha dát a ich obnova v prípade havárie systému
- vysoká dostupnosť dát
- sprístupnenie dát koncovému užívateľovi
- spĺňanie regulačných noriem a zákonov
- ochrany súkromia a e-Discovery
- vyťažovanie dát (data mining)
- efektívne využitie úložného priestoru (deduplikácia príloh)

3.2 Požiadavky na systém

V nasledujúcej časti popíšeme požadované vlastnosti systému, ktorý bude slúžiť na archiváciu veľkého objemu emailových správ. Primárnou požiadavkou na systém je jeho neustála dostupnosť, rozširiteľnosť a nízkonákladová administrácia. Predpokladané množstvo uložených dát v tomto systéme bude dosahovať desiatky až stovky terabajtov² (TB). Takúto kapacitu dát nie je možné uchovať na jednom počítačovom systéme tvorenom z bežne dostupného hardvéru. Dáta uložené v systéme musia byť replikované, v prípade vzniku havárie niektorej z jeho častí. Nad uloženými emailovými správami je potrebné vykonávať výpočtovo náročné operácie ako generovanie štatistík a fulltextové vyhľadávanie v reálnom čase. Tieto požiadavky prirodzene implikujú využitie distribuovaného databázového systému. Medzi hlavných kandidátov, vďaka ktorým sme tieto požiadavky schopní vyriešiť patria NoSQL databázové systémy, ktoré popíšeme v nasledujúcej kapitole.

3.2.1 Funkcionálne požiadavky

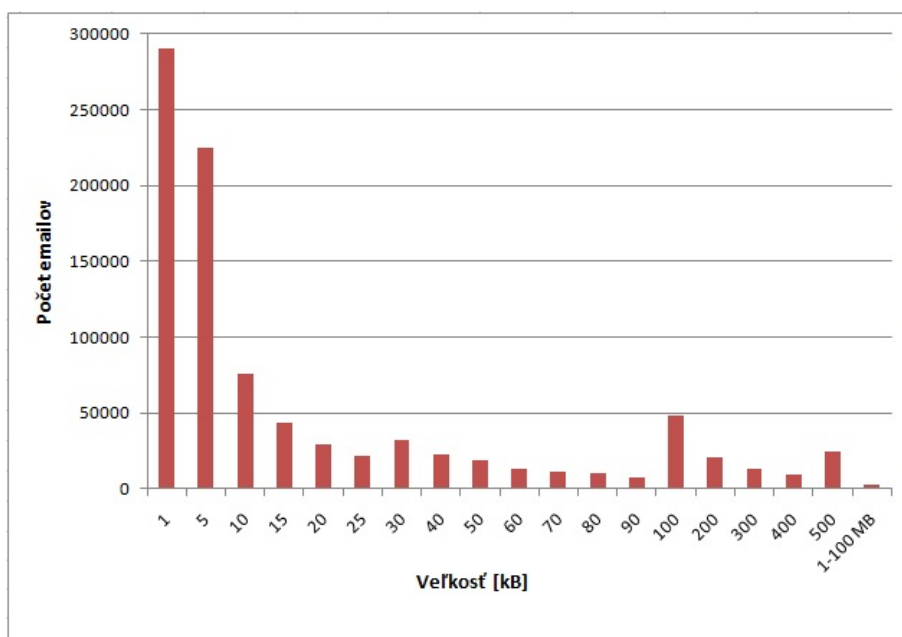
Ukladanie emailov

Základnou jednotkou, ktorú budeme do systému ukladať je emailová správa. Graf na obrázku 3.1 znázorňuje usporiadanie emailov podľa ich veľkosti nad vzorkom približne 1,000,000 emailových správ z reálneho prostredia³. Z daných dát vyplýva, že veľkosť cca 80% emailov je do 50 kB. Tieto údaje sú hrubou aproximáciou a závisia na konkrétnych používateľoch.

Systém musí umožňovať uloženie emailu bez porušenia jeho integrity. V prípade sprístupnenia emailu nemôže dôjsť k zmene alebo porušeniu dát. Kľúčovým požiadavkom je ukladanie emailových príloh, kde požadujeme aby každá príloha bola jednoznačne identifikovaná a v prípade jej duplicity nebola opakovane uložená v systéme. Cieľom je dosiahnutie efektívnej úspory diskového priestoru. Ďalším požiadavkom je automatické zmazanie emailov uložených v systéme, po predom špecifikovanej dobe, ktoré prináležia danej doméne.

²1TB = 10¹²B

³Vzorok emailov pre analýzu bol sprístupnený spoločnosťou Excello.



Obr. 3.1: Približná distribúcia emailových správ. $\mu = 301$ kB, $\sigma = 1.3$ MB

Export emailov

Systém musí umožňovať prístup k ľubovoľnému uloženému emailu v jeho pôvodnej podobe poprípade skupine všetkých emailov patriacej danému užívateľovi (inbox).

Vyhľadávanie emailov

Vyhľadávanie je potrebné realizovať nad všetkými emailovými správami uloženými v systéme, jednotlivo nad správami podľa názvu domény a nad správami, ktoré prináležia danému užívateľovi. Požadujeme fulltextové vyhľadávanie emailov podľa nasledujúcich údajov:

- príjemca emailovej správy
- odosielateľ emailovej správy
- predmet správy
- dátum obsiahnutý v hlavičke emailovej správy
- identifikátor emailu (*MessageID*)
- názvy príloh a ich veľkosti
- veľkosť emailu
- vyhľadávanie v tele emailu

Pre administrátorské účely požadujeme vyhľadávanie údajov podľa:

- originálny odosielateľ a príjemca
- IP adresa odosielateľa
- dátum a čas spracovania správy emailovým serverom

Štatistické údaje

Nad uloženými dátami požadujeme výpočet štatistík pomocou využitia MapReduce. Pre emaily patriace do danej domény je potrebné spracovať nasledujúce štatistické ukazovatele:

- počet emailov označených príznakom spam
- počet emailov bez príznaku spam
- celková veľkosť emailov pre danú doménu
- veľkosť najväčšieho emailu v doméne
- celková dĺžka filtrácie emailov v danej doméne

Nad celým úložiskom je ďalej potrebné spracovať tieto štatistiky:

- počet všetkých emailov
- počet unikátnych domén
- počet unikátnych príloh

3.2.2 Nefunkcionálne požiadavky

Dostupnosť

System musí byť neustále dostupný (99,9%), schopný odolávať sieťovým prerušeniam spôsobujúcim nedostupnosť uzlov, úplným zlyhaniam jednotlivých uzlov a umožňovať spracúvať tok pre zápis dát v rozmedzí 10 Mbit až 1 Gbit. Ďalším požiadavkom je aby sa dáta replikovali vo vnútri dátového centra (datacenter) na dva uzly a tretia replika bola umiestnená v dátovom centre, ktoré sa bude nachádzať na geograficky odlišnom mieste. Vyžadujeme aby systém neobsahoval kritický bod výpadku, požadujeme vlastnosť decentralizácie.

Rozšíriteľnosť

Predpokladáme použitie bežne dostupného spotrebného hardvéru⁴ (commodity hardware), namiesto superpočítačov. Z dôvodu neustalého nárastu objemu elektronickej pošty, musí systém podporovať horizontálne škálovanie, ktoré bude umožňovať zvýšenie celkovej kapacity dátového úložiska (desiatky petabajtov). Pridávanie nových uzlov do systému umožní zvýšiť celkový výpočtový výkon, ktorý sa využije na spracovanie dát pomocou metódy MapReduce. U distribuovaného databázového systému je nutná podpora replikácie, ktorá zvýši výkonnosť operácií pre čítanie, zápis a vďaka nej nebude potrebné riešiť zálohovanie pomocou externých systémov.

Nízkonákladová administrácia

Prevádzkovanie systému a jeho administrácia by mali byť čo najmenej závislé na zásahu ľudských zdrojov. Detekcia nefunkčných uzlov a automatické rozdeľovanie záťaže sa musí vykonávať automaticky. Pridanie poprípadе odobranie nového uzla nesmie ovplyvniť beh celkového systému.

Bezpečnosť

Osoby s oprávnením pre prístup k systému budú schopné manipulovať s jeho celým obsahom. Predpokladáme beh systému v bezpečnom prostredí a nekladíme žiadne požiadavky na užívateľské role v kontexte prístupu k dátam.

Implementačné požiadavky

Cieľom je implementácia systému s využitím dostupných open source technológií.

Z analýzy princípov, v predchádzajúcej kapitole, ktoré využívajú relačné databázové systémy vyplýva, že použitie týchto systémov nie je vhodné pre riešenie zadaného problému. Medzi základné problémy týchto systémov patrí náročné horizontálne škálovanie, čo negatívne ovplyvňuje primárny požiadavok vysokej dostupnosti. V nasledujúcej kapitole sa budeme zaoberať popisom NoSQL systémov a po ich analýze vyberieme vhodného kandidáta, ktorého použijeme k implementácii prototypu, z dôvodu vysokej komplexnosti riešeného problému.

⁴komponenty sú štandardizované, bežne dostupné a ich cenu neurčuje výrobca (IBM, DELL,...) ale trh. Príkladom môže byť konfigurácia s pevným diskom 2 x 250 GB SATA, 4-12 GB RAM, 8 jadrový CPU.

Kapitola 4

NoSQL

Názov NoSQL bol prvýkrát použitý v roku 1998 ako názov relačnej databáze¹, ktorej implementácia bola v interpretovaných programovacích jazykoch a neobsahovala dotazovací jazyk SQL. V druhej polovici roku 2009 sa názov NoSQL začal používať v spojení s databázovými systémami, ktoré nepoužívajú tradičný relačný model, dotazovací jazyk SQL, sú schopné horizontálneho škálovania, pracujú na spotrebných počítačových systémoch, vyznačujú sa vysokou dostupnosťou a používajú bezschémový dátový model.

Pôvodným cieľom hnutia NoSQL bolo vytvoriť koncept, pre tvorbu moderných databázových systémov, ktoré by boli schopné spracúvať požiadavky neustále sa rozvíjajúcich webových aplikácií. Filozofiou týchto systémov je nesnažiť sa za každú cenu prispôbovať dáta, ktoré do nich ukladáme relačnému databázovému modelu. Cieľom je vybrať systém, ktorý bude čo najvhodnejšie zodpovedať požiadavkám pre uloženie a spracovanie dát. NoSQL nepopisuje žiaden konkrétny databázový systém, namiesto toho je to obecný názov pre nerelačné (non-relational) databázové systémy, ktoré disponujú odlišnými vlastnosťami a umožňujú prácu s rôznymi dátovými modelmi. Medzi ich ďalšie znaky patrí napríklad eventúalna konzistencia, schopnosť spracúvať obrovské objemy dát (PB) a jednoduché programové rozhranie (API). Tieto systémy nepodporujú operáciu databázového spojenia a medzi dátami, ktoré do nich ukladáme je možné vytvárať vzájomné závislosti až na aplikačnej úrovni. Pre tieto databázové systémy ďalej platí, že sú distribuované, automaticky poskytujú replikáciu a rozdeľovanie dát. Jedná sa o relatívne mladé systémy, ktoré sú neustále vo vývoji. Ideou tohto konceptu je riešiť spomínané novovznikajúce problémy a zároveň koexistovať s relačnými databázovými systémami.

4.1 Dátové modely

NoSQL systémy delíme na dokumentové, grafové, stĺpcové a systémy s dátovým modelom typu „kľúč-hodnota“ (Key-value). V nasledujúcej časti práce ich stručne popíšeme.

¹http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql

4.1.1 Relačný model

Pre porovnanie stručne popíšeme štandardný relačný model. Databázový model reprezentuje dáta pomocou relácií, ktoré tvoria riadky uložené v tabuľkách. Štruktúra týchto záznamov je normalizovaná, aby sa predišlo ich duplikácii. Pre zabezpečenie referenčnej integrity jednotlivých entít sa využívajú cudzie kľúče. Tabuľky s popisom názvov stĺpcov a vzťahy medzi nimi nazývame databázovou schémou. Záznamy sú sekvenčne ukladané na pevný disk. Tento model je vhodný pre systémy, u ktorých sú dominantné operácie vykonávajúce zápis. Relačné databázové systémy sú teda optimalizované pre zápis. Pre efektívny prístup k dátam je možné použiť indexy. Tabuľka 4.1 zachytáva ukážku tabuľky v relačnom databázovom systéme.

Kľúč	Meno	Adresa	Telefón
1	Ján Mak	Zikova, Praha	773234512
2	John Smith	Broodway, NYC	14941234
3	Mary Novak	Vodickova, Praha	603192301

Tabuľka 4.1: Reprezentácia dát v relačnom modeli

4.1.2 Kľúč-hodnota

Tento model využíva pre ukladanie dát jednoduchý princíp. Blok dát s ľubovoľnou štruktúrou je v databáze uložený pod názvom kľúča, ktorý je často interne reprezentovaný ako pole bajtov a môže ním byť napríklad textový retazec. Databázové systémy využívajúce tento dátový model majú jednoduché programové rozhranie:

```
void Put(string kluc, byte[] data);
byte[] Get(string key);
void Remove(string key);
```

Výhodou tohto modelu je, že databázový systém je možné ľahko škálovať. V tomto prípade prácu so štruktúrou uložených dát zabezpečuje klient, čo umožňuje dosahovať vysokú výkonnosť na strane databázového systému.

Jednou z možných nevýhod v porovnaní s relačnými databázovými systémami je, že databázový systém nie je schopný medzi uloženými dátami zachytiť ich vzájomné vzťahy (relácie), čo môže byť jednou z požiadaviek pri tvorbe komplexných modelov. Úložisko typu kľúč-hodnota nevyužíva normalizáciu dát, dáta sú často duplikované, vzťahy a integrita medzi nimi sa riešia až na aplikačnej úrovni. Na obsah vkladáných dát a k nim asociovaným kľúčom sa nedefinujú žiadne obmedzenia.

Medzi databázové systémy využívajúce model kľúč-hodnota patria Amazon Dynamo, Tokyo Cabinet, Voldemort, Redis a iné.

4.1.3 Stĺpcovo orientovaný model

V dnešnej dobe existuje veľký počet aplikácií, u ktorých prevládajú operácie čítania nad zápisom. Patria sem dátové sklady, systémy pre vyťažovanie dát alebo analytické aplikácie pracujúce s obrovským objemom dát. Pre potreby týchto aplikácií a ich reprezentáciu dát je vhodné použiť stĺpcový model [3, 4], ktorý je optimalizovaný pre operácie čítania dát. Data reprezentujúce stĺpce sú uložené na pevnom disku v samostatných a súvislých blokoch. Načítanie dát do pamäti a následná práca s nimi je efektívnejšia ako u riadkových databáz, kde je potrebné načítať celý záznam obsahujúci aj hodnoty stĺpcov, ktoré sú v daný moment irelevantné. Možný príklad fyzickej reprezentácie tabuľky *Zamestnanec* zachytáva tabuľka 4.2, stĺpce *Kľúč* je možné vynechať, v tomto prípade v logickej reprezentácii budú riadky v rovnakom poradí zachytávať záznam.

Riadkový model obsahuje v jednom zázname dáta z rôznych domén, čo spôsobuje vyššiu entropiu v porovnaní so stĺpcovým modelom, kde sa predpokladá, že dáta v danom stĺpci pochádzajú z totožnej domény a sú si preto podobné. Táto vlastnosť umožňuje efektívnu komprimáciu dát, ktorá znižuje počet diskových operácií. Nevýhodou tohto modelu je zápis dát, ktorý spôsobuje vyššiu záťaž diskových operácií. Pre optimalizáciu operácií vykonávaných zápis sa používa dávkový zápis.

Tento model využívajú databázové systémy ako Google Bigtable, HBase, Hypertable alebo Cassandra.

Kľúč	Meno	Kľúč	Adresa	Kľúč	Telefón
1	Ján Mak	1	Zikova, Praha	1	773234512
2	John Smith	2	Broodway, NYC	2	14941234
3	Mary Novak	3	Vodickova, Praha	3	603192301

Tabuľka 4.2: Reprezentácia dát v stĺpcovom modeli

4.1.4 Dokumentový model

Dokumentové databázy sú založené na modeli typu kľúč-hodnota. Požiadavkou na ukladanie dát je, že musia byť v tvare, ktorý dokáže spracovať databázový systém. Štruktúra vkladaných dát môže byť určená napríklad pomocou XML, JSON, YAML. Schéma týchto systémov následne umožňuje okrem jednoduchého vyhľadávania pomocou modelu kľúč-hodnota vytvárať s využitím indexov zložitejšie dotazy nad dátami, ktoré sú vyhodnocované na strane databázového systému.

Medzi databáze reprezentujúce tento typ úložiska patrí napríklad CouchDB a MongoDB.

4.1.5 Grafový model

Tento typ databáz využíva pre prácu s dátami matematickú štruktúru graf. Dáta sú reprezentované pomocou uzlov, hrán a ich atribútov. Základným objektom je uzol. Pomocou hrán medzi uzlami modelujeme závislosti, ktoré sú popísané pomocou atribútov. Nad uzlami a hranami sa využíva model kľúč-hodnota. Medzi hlavné výhody patrí možnosť prechádzania týchto štruktúr s využitím známych grafových algoritmov.

Tento model sa napríklad využíva v aplikáciach sociálnych sietí alebo pre sémantický web. Patria sem databázové systémy Neo4j alebo FlockDB.

4.2 Porovnanie NoSQL systémov

V dnešnej dobe existuje veľké množstvo NoSQL databázových systémov, ktoré majú odlišné vlastnosti, komplexitu a vďaka tomu ich môžeme použiť pre rôzne účely. Snaha porovnať tieto systémy na globálnej úrovni je nerealizovateľná a často vedie k chybám. Cieľom tejto sekcie je definovať základné body, vďaka ktorým je možné tieto systémy kategorizovať a v rámci danej kategórie porovnávať. Tieto zistenia nám následne môžu pomôcť pri výbere vhodného systému, ktorý bude vhodný na riešenie našich požiadaviek. Následujúce body patria medzi hlavné kritéria pri porovnávaní týchto systémov:

- dátový model
- dotazovací model
- škálovateľnosť
- schopnosť odolávať chybám (failure handling)
- elasticita
- konzistencia dát
- prostredie behu
- bezpečnosť

4.2.1 Dátový a dotazovací model

Dátový model definuje štruktúru, ktorá slúži na ukladanie dát v databázovom systéme. Dotazovací model následne definuje obmedzenia a operácie, ktoré je možné nad uloženými dátami vykonávať. V predchádzajúcej sekcii sme popísali základnú kategorizáciu NoSQL systémov podľa dátového modelu. Dátový a dotazovací model do určitej miery popisuje výkonnosť, komplexnosť a vyjadrovaciu silu databázového systému. Dotazovací model popisuje programové rozhranie (API).

Pri výbere vhodného dátového modelu pre aplikáciu je dôležité porozumieť štruktúre dát a definovať operácie, ktoré nad týmito dátami budeme vykonávať. Na základe týchto operácií sa navrhne databázové schéma.

4.2.2 Škálovateľnosť a schopnosť odolávať chybám

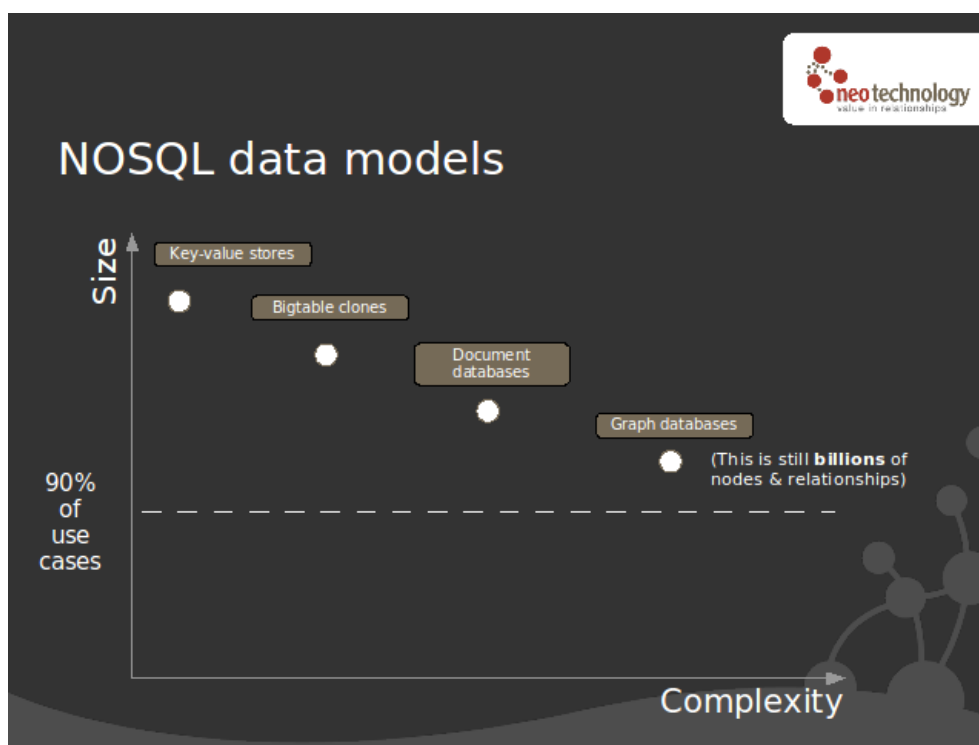
Tieto vlastnosti kladú na systém požiadavky ako podpora replikácie a rozdeľovanie dát. Distribuované databázové systémy implementujú tieto techniky na systémovej úrovni. V prípade ich podpory môžeme od systému požadovať:

- podporu replikácie medzi geograficky oddelenými dátovými centrami

- možnosť pridania nového uzlu do distribuovaného databázového systému, bez nutnosti zásahu do aplikácie

Počet uzlov, ktoré obsahujú repliku dát a konfigurácia databázového systému, ktorá podporuje geograficky oddelené dátové centrá určujú stupeň odolnosti systému voči chybám sieťového prerušenia.

Častou požiadavkou webových aplikácií na databázový systém, z dôvodu neustáleho nárastu dát, je podpora škálovania s cieľom zvýšenia veľkosti databáze. S neustálym vývojom nových aplikácií musíme zároveň uvažovať potrebu škálovania aj z pohľadu komplexnosti dát [36]. Výber dátového modelu môže byť ovplyvnený na základe komplexnosti dát. Obrázok 4.1 zachytáva pozíciu dátových modelov z pohľadu škálovania komplexnosti a veľkosti dát.



Obr. 4.1: Pozícia dátového modelu z pohľadu škálovania podľa veľkosti a komplexnosti. Zdroj: [17]

Dátový model typu kľúč-hodnota a stĺpcovo orientovaný model (Bigtable clones) majú jednoduchú štruktúru, ktorá kladie minimálnu náročnosť v prípade horizontálneho škálovania. Nevýhodou tohto prístupu je naopak to, že všetka práca s dátami a ich štruktúrou sa prenáša do vyšších vrstiev, o ktoré sa musí starať programátor. Naopak dokumentový a grafový model poskytuje bohatšiu štruktúru na prácu s dátami, ktorá spôsobuje komplikovanejšie škálovanie vzhľadom k veľkosti dát. Podľa odhadov spoločnosti Neotechnology² až 90% aplikácií, v prípade že sa nejedná o projekty spoločností Google, Amazon atď., spadá do rozmedzia, kde sa veľkosť záznamov pohybuje rádovo v miliardách. Za zmienku stojí fakt,

²<http://neotechnology.com/>

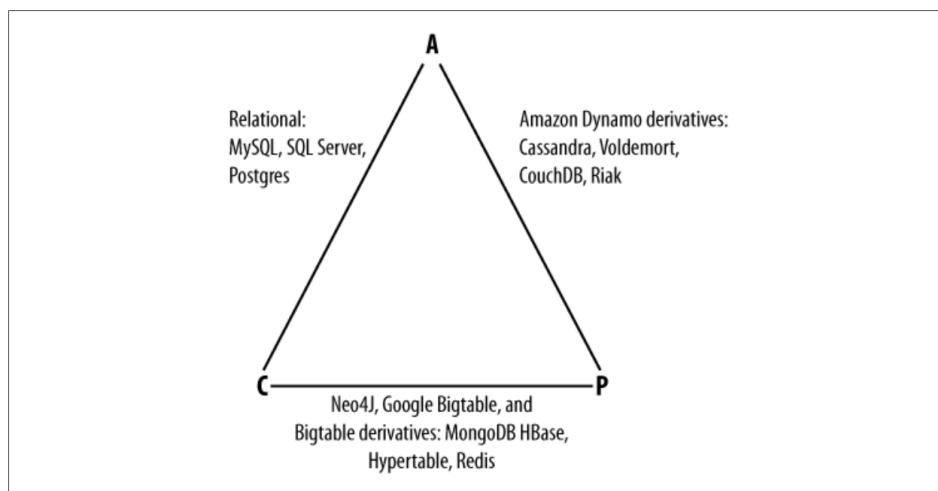
že aj napriek tomu, že tieto dátové modely sú si navzájom izomorfné, vhodnosť ich použitia závisí na konkrétnom príklade a požiadavkách na aplikáciu.

4.2.3 Elasticnosť

Elasticnosť škálovania popisuje ako sa daný systém dokáže vysporiadať s pridaním alebo odobraním uzla. Určuje do akej miery je v tomto prípade potrebný manuálny zásah pre rovnomerné rozloženie záťaže, prípadná potreba reštartovania systému alebo zmena v užívateľskej aplikácii. Ideálne by pridávanie nových uzlov do systému malo zabezpečiť lineárne zvyšovanie výkonu u operácií ako je čítanie alebo zapisovanie dát.

4.2.4 Konzistencia dát

Podľa teórie CAP (viď. 2.6) platí, že v prípade výskytu sieťových prerušení, ktorých prítomnosť v prostredí distribuovaných databázových systémov je prirodzená, nie je možné súčasne zaručiť vlasnosť konzistencie a dostupnosti. NoSQL systémy preto môžeme rozdeliť podľa tohto modelu do dvoch skupín, ktoré znázorňuje obrázok 4.2. Umiestnenie niektorých databázových systémov sa môže meniť na základe ich konfigurácie.



Obr. 4.2: Rozdelenie databázových systémov podľa CAP, Zdroj: [31]

4.2.5 Prostredie behu

NoSQL systémy ďalej delíme podľa prostredia, v ktorom pracujú. Väčšina týchto open source systémov umožňuje ich nasadenie do vlastného prostredia teda privátnych cloudov alebo do verejného cloudového riešenia EC2 od spoločnosti Amazon. Medzi tieto systémy patria napríklad Cassandra, HBase, Riak, Voldemort. Databázové systémy Amazon SimpleDB, Microsoft Azure SQL, Yahoo! YQL alebo Google App Engine sú poskytované ako komplexné cloudové riešenie. Rozhranie pre prácu s dátami a funkčnosť perzistentného úložiska zabezpečujú poskytovatelia týchto systémov.

4.2.6 Bezpečnosť

Distribučovaný databázový systém je možné nasadiť do cloudu. V prípade použitia verejných cloudov môže hroziť nebezpečenstvo zneužitia dát treťou stranou a preto je potrebné aplikovať bezpečnostné mechanizmy. Pri nasadení systému do privátneho cloudu zasa môžeme požadovať viacero úrovní ochrany pre prístup k dátam. Systémy NoSQL disponujú minimálnymi prvkami pre zaručenie autentizácie a autorizácie. Mnoho systémov tieto mechanizmy vôbec neimplementuje poprípade ich implementácia je v začiatkoch.

4.3 Výber NoSQL systémov

V predchádzajúcej sekcii sme tieto distribuované databázové systémy rozdelili do štyroch hlavných kategórií na základe ich dátového modelu, ktorý je jedným z kľúčových faktorov pri výbere vhodného kandidáta podľa požiadaviek cieľovej aplikácie. Detailný popis a výkonnostné porovnanie NoSQL systémov, ktoré reprezentujú jednotlivé kategórie by boli nad rámec tejto práce. Paralelne s touto prácou vzniká diplomová práca, ktorá rieši podobný problém s využitím dokumentových databázových systémov [5], preto túto kategóriu pri výbere vynecháme.

Podľa analýzy požiadaviek na našu aplikáciu, ktoré sme popísali v predchádzajúcej kapitole nie je vhodné použitie databázových systémov s grafovým modelom a modelom typu kľúč-hodnota. Systémy s grafovým modelom sú určené na odlišnú úlohu problémov. V prípade použitia systémov s dátovým modelom kľúč-hodnota by sme komplexnosť riešenej úlohy preniesli na úroveň klienta. Naším požiadavkám najlepšie vyhovuje stĺpcovo orientovaný model, ktorý využijeme v návrhu našej aplikácie. Hlavnou výhodou je, že tento model u systémov NoSQL nestanovuje žiadnu schému na dáta, ktoré budeme do systému vkladať, počet stĺpcov je ľubovoľný a závisí od vstupných dát. Pre potreby nášho riešenia preto použijeme stĺpcovo orientovaný NoSQL systém.

Stĺpcovo orientované NoSQL systémy

V tejto časti sa zameriame na vzájomné porovnanie systémov, ktoré poskytujú stĺpcovo orientovaný model. Medzi tieto open source systémy patria HBase, Cassandra a Hypertable. Napriek totožnému dátovému modelu sú tieto systémy založené na rôznych architektonických princípoch a disponujú čiastočne odlišnými vlastnosťami. Tabuľka 4.3 zobrazuje stručný prehľad vlastností, na ktoré sme sa zamerali pri výbere víťaznej dvojice.

Zo stručného prehľadu v tabuľke je vidieť, že systémy obsahujú množstvo spoločných vlastností. Pri výbere sme zohľadnili aj ich praktické využitie spoločnosťami globálne pôsobiacimi na IT trhu. Napríklad systém Cassandra je používaný spoločnosťou Facebook v aplikácii pre vyhľadávanie a uchovávanie súkromnej pošty. Ďalšími dôležitými požiadavkami pre výber boli dostupnosť klientských aplikácií, podpora komunity a kvalita dokumentácie. U systému Cassandra zohrala pri výbere hlavnú rolu decentralizácia a možnosť voľby medzi konzistenciou a dostupnosťou. Zo zvolených kandidátov sme vybrali systém HBase a Cassandra. Dôvodom prečo sme zavrhlí systém Hypertable je nepostačujúca dokumentácia, málo aktívna komunita a v čase výberu sa nachádzal vo verzii alfa. V nasledujúcich dvoch kapitolách detailne popíšeme oba systémy.

Počas písania tejto práce bol uvoľnený ďalší open source distribuovaný databázový systém implementujúci model Bigtable pod názvom Cloudata³, ktorý sme už do našej práce nezahrnuli.

Vlastnosť	Databázový systém		
	HBase	Cassandra	Hypertable
Distribuovaný systém	áno	áno	áno
Architektúra	Bigtable	Dynamo	Bigtable
Klient	Thrift, REST	Thrift, Avro	Thrift, C++
Perzistentné úložisko	HDFS, AS3 ⁴ , KFS	LSS ⁵ , AS3	HDFS, KFS, LSS
Kritický bod výpadku	áno	nie	voliteľné ⁶
Podpora viacerých dátových centier	áno	áno	áno
Automatické rozdeľovanie dát	áno	áno	áno
Replikácia	áno	áno	áno
Konzistencia	CP	voliteľná	CP
Kompresia dát	LZO, GZIP	nie	LZO, ZLIB
Programovací jazyk	Java	Java	C++
MapReduce	áno	áno	áno
Indexy	nie	áno	nie
Dokumentácia	+	++	—
Komunita	+	+	—

Tabuľka 4.3: Stručný prehľad vlastností stĺpcovo orientovaných systémov NoSQL

³<http://www.cloudata.org>

⁴Amazon S3, <http://aws.amazon.com/s3/>

⁵lokálny súborový systém

⁶záleží na voľbe perzistentného úložiska

Kapitola 5

Cassandra

Distribučný databázový systém Cassandra [43] bol vytvorený pre interné účely spoločnosti Facebook v roku 2007. Cassandra slúžila na vyhľadávanie v súkromnej pošte a poskytovala úložisko pre indexy vytvárané nad týmito dátami. Od systému sa predpokladalo, že bude obsluhovať miliardy zápisov denne, podporovať škálovanie podľa narastajúceho počtu používateľov, umožňovať beh na spotrebných počítačoch a podporovať replikáciu medzi geograficky oddelenými dátovými centrami. Ďalšou požiadavkou bola vysoká dostupnosť, aby chyba žiadneho uzlu nespôsobila celkovú nedostupnosť systému. Cassandra je decentralizovaný systém, kde každý uzol vykonáva tie isté operácie. V roku 2008 bola zverejnená ako open source projekt a je neustále vyvíjaná mnohými spoločnosťami a vývojármi. Tento systém využíva architektonické princípy distribučného databázového systému Dynamo [16] od spoločnosti Amazon a dátový model prevzal od distribučného databázového systému Bigtable [10] vytvoreného spoločnosťou Google. V nasledujúcom texte popíšeme hlavne princípy, na ktorých je tento systém založený.

5.1 Dátový model

Základnou jednotkou dátového modelu je stĺpec. Stĺpec je tvorený názvom, hodnotou a časovým odťahom, ktorý využíva Cassandra pri riešení konfliktov súbežného zápisu. Skupina stĺpcov je identifikovaná pomocou unikátneho kľúča a predstavuje riadok. Počet a názvy stĺpcov nie je potrebné vopred definovať. Zoradené riadky podľa hodnoty kľúčov a v nich zoradené stĺpce obaľuje štruktúra pod názvom „rodina stĺpcov“ (Column family). Kľúče interne reprezentované ako reťazec znakov sú zároveň zotriedené. Názvy stĺpcov môžu byť viacerých typov ako napríklad *ASCII*, *UTF-8*, bajtové pole a iné, podľa ktorých sú zotriedené. Je možné implementovať vlastnú metódu pre triedenie. Riadky obsiahnuté v jednej rodine stĺpcov sú na pevnom disku fyzicky umiestnené v jednom súbore typu *SSTable*. Je vhodné do rodiny stĺpcov ukladať relevantné záznamy, ku ktorým budeme pristupovať spoločne, čím sa vyhneme nadbytočným diskovým operáciám. V rámci jednej repliky sú operácie nad stĺpcami atomické. Operácie nad riadkom nevyužívajú zamykanie. Voliteľným príznakom, ktorý môžeme u stĺpca nastaviť je parameter TTL (Time to live), ktorý po uplynutí časového intervalu označí dáta príznakom pre zmazanie.

Cassandra k dátovému modelu systému Bigtable pridala štruktúru pod názvom „super stĺpec“ (Super column). Štruktúra super stĺpec je špeciálny typ stĺpca, ktorý je tvorený

obyčajnými stĺpcami. Stĺpec typu *super* má názov a jeho hodnota je tvorená zoznamom názvov obyčajných stĺpcov. Super stĺpce obaľuje štruktúra pod názvom „*super-rodina stĺpcov*“ (Super column family). *Keyspace* združuje rodiny a super-rodiny stĺpcov. Definuje faktor a metódu replikácie, ktorá môže byť závislá alebo nezávislá na sieťovej topológii. Na *keyspace* sa môžeme pozeráť ako na databázu a rodiny stĺpcov môžeme prirovnať k tabuľkám v relačných databázových systémoch.

Aktuálna verzia Cassandra definuje maximálnu veľkosť dát 2 GB, ktoré je možné uložiť do jedného stĺpca v riadku a stanovuje limit dve miliardy pre maximálny počet stĺpcov tvoriacich riadok.

5.2 Rozdeľovanie dát

Kľúčovou požiadavkou systému Cassandra je jeho schopnosť horizontálneho škálovania, čo vyžaduje pridávanie nových uzlov. Táto vlastnosť vyžaduje mechanizmus, ktorý zabezpečí automatické rozdeľovanie dát medzi uzlami systému. Uvažujme príklad, kde máme k dispozícii jeden server obsahujúci veľké množstvo objektov, ku ktorým pristupujú klienti. Medzi server a klientov vložíme vrstvu kešovacích systémov, kde každý z týchto systémov bude zodpovedný za prístup k časti objektov nachádzajúcich sa na serveri. Klient musí byť schopný určiť na základe hľadaného objektu, ku ktorému kešovaciemu systému musí pristúpiť. Predpokladajme, že klient realizuje výber jednotlivých kešovacím systémom pomocou lineárnej hašovacej funkcie $y = ax + b \pmod{p}$, kde p je počet kešovacích systémov. Pridanie nového kešovacieho systému alebo jeho zlyhanie bude mať katastrofálny dopad na funkčnosť systému. V prípade zmeny hodnoty p , bude hľadaná položka odpovedať novej a zároveň chybné lokácii. Tento problém rieši elegantne technika pod názvom „úplné hašovanie“ (Consistent hashing) [32], ktorá sa využíva v distribuovaných systémoch pre určenie pozície pre prístup k distribuovaným hašovacím tabuľkám a využíva ju systém Cassandra.

Výstup hašovacej funkcie MD5 je reprezentovaný pomocou „kruhu“, kde v smere hodinových ručičiek postupujeme od minimálnej hodnoty hašovacej funkcie 0 k maximálnej 2^{127} . Každý uzol v systéme má pridelenú hodnotu z tohto rozsahu, ktorá určí jeho jednoznačnú pozíciu. Identifikácia uzlu v systéme, na ktorý sa uložia dáta reprezentované hodnotou kľúča sa vykoná aplikáciou hašovacej funkcie na dáta reprezentujúce kľúč. Na základe tejto hodnoty je jednoznačne určená pozícia v kruhu a v smere hodinových ručičiek je vyhľadaný najbližší uzol, ktorému dáta prináležia. Výhodou tejto metódy je, že každý uzol je zodpovedný za hodnotu kľúčov, ktorých poloha sa nachádza medzi ním a jeho predchodcom. V prípade pridania nového uzlu alebo jeho odobratia, sa zmena mapovania kľúčov v kruhu prejaví len u jeho susedov. Táto technika prináša výhody, medzi ktoré patrí rovnomerná distribúcia dát a vyváženie záťaže. Dynamo tento problém rieši spôsobom, kde každý uzol je zodpovedný za viacero pozícií na kruhu, takzvané virtuálne pozície.

Cassandra využíva vlastné mechanizmy na monitorovanie záťaže uzlov a automaticky presúva ich pozície v kruhu. Zároveň je možné explicitne stanoviť polohu každého uzla v kruhu pomocou zadania jeho identifikátora. Tento spôsob je vhodný v prípade, ak vieme dopredu určiť, koľko uzlov bude systém obsahovať. V prípade zvyšovania počtu uzlov je možné tieto identifikátory zmeniť počas behu systému. Identifikátor polohy uzlov je možné určiť pomocou nasledujúceho programu v jazyku Python, kde K je počet uzlov v systéme.

```
def tokens(n):
    r = []
    for x in xrange(n):
        r.append(2**127 / n * x)
    return r

print tokens(K)
```

Štandardne Cassandra využíva *RandomPartitioner*, u ktorého výstup hašovacej funkcie MD5 aplikovaný na hodnotu kľúča určuje uzol, do ktorého sa zapíšu dáta. Týmto sa zabezpečí rovnomerné rozdelenie dát a záťaže na každom uzle v klastri. K dispozícii je taktiež *OrderPreservingPartitioner*, u ktorého je každý uzol zodpovedný za interval kľúčov. Napríklad v prípade kľúčov začínajúcich začiatočným písmenom abecedy, môže byť rozdelenie na troch uzloch následovné:

1. uzol: a - e
2. uzol: e - k
3. uzol: k - z

Táto metóda nezaručuje rovnomerne rozdelenie dát a záťaže.

5.3 Replikácia

S úplným hašovaním úzko súvisí replikácia. Každá dátová jednotka vložená do systému je replikovaná na N uzlov, kde počet N je voliteľne nastaviteľný pre daný keyspace. Každý uzol sa v prípade replikácie $N > 1$ stáva koordinátorom, ktorý je zodpovedný za replikáciu dát, ktorých kľúč spadá do jeho rozsahu. Počas operácie zápisu koordinátor replikuje dáta na ďalších $N - 1$ uzlov. Cassandra podporuje viacero spôsobov pre umiestňovanie replík.

Jednoduchá stratégia

Táto stratégia umiestňuje repliku dát bez ohľadu na umiestnenie uzlov (serverov) v dátovom centre. Primárnu repliku spravuje uzol, do ktorého rozsahu spadajú ukladané dáta a ostatné repliky sú uložené na $N - 1$ susedov v smere hodinových ručičiek. Z toho vyplýva, že každý uzol je zodpovedný za dáta, ktorých kľúče spadajú do jeho rozsahu a súčasne dáta, ktoré spravuje jeho N predchodcov.

Sieťová stratégia

Pri tejto metóde a úrovni replikácie s hodnotou aspoň tri, je systém schopný zabezpečiť umiestnenie dvoch replík v rozdielnych „rackoch“ v rámci jedného dátového centra. Tretia replika môže byť umiestnená do geograficky oddeleného dátového centra. Táto stratégia je výhodná v prípade ak chceme použiť časť serverov na výpočty pomocou MapReduce a zvyšné dve repliky budú slúžiť na obsluhu reálnej prevádzky.

5.4 Členstvo uzlov v systéme

Distribuívaný systém musí byť schopný odolávať chybám ako porucha uzlov alebo sieťové prerušenia. Decentralizácia a detekcia chýb využíva mechanizmy založené na *gossip protokoloch* [21]. Tieto protokoly slúžia pre vzájomnú komunikáciu uzlov vymieňajúcich si navzájom dôležité informácie o svojom stave. Periodicky v sekundových intervaloch každý uzol kontaktuje náhodne vybraný uzol, kde si overí či je tento uzol dostupný. Detekcia nedostupnosti uzla je realizovaná algoritmom s názvom Accrual Failure Detector [30].

Pridávanie nových uzlov a presun uzlov v rámci kruhu sa taktiež dejú pomocou Gossip protokolu, ktorý ďalej zabezpečuje, že každý uzol obsahuje informácie o tom, ktorý uzol je zodpovedný za daný rozsah kľúčov v kruhu. Ak sa vykonáva operácia čítania alebo zápisu dát na uzol, ktorý nie je zodpovedný za tento kľúč, dáta sú automaticky preposlané na správny uzol, ktorého výber je zabezpečený s časovou zložitou $O(1)$.

5.5 Zápis dát

Tento systém bol primárne navrhnutý tak, aby spracúval vysoký tok pre zápis dát. Ak uzol obdrží požiadavku pre zápis, dáta sú zapísané do štruktúry pod názvom *Commitlog*, ktorá je uložená na lokálnom súborovom systéme a zabezpečuje trvácnosť dát. Zápis do tejto štruktúry je vykonaný sekvenčne, čo umožňuje dosiahnuť vysokú priepustnosť bez nutnosti vystavovania diskových hlavičiek (disk seek operations). Dáta sú následne zoradené a nahrané do štruktúry pod názvom *Memtable*, ktorej obsah je uložený v operačnej pamäti. V prípade, že by tento zápis zlyhal alebo by došlo k neočakávanému pádu inštancie Cassandra, obnovenie obsahu týchto štruktúr je možné vďaka Commitlogu. Po dosiahnutí vopred stanovenej hodnoty, ktorá určuje maximálny počet dát uložených v Memtable, sú tieto štruktúry zapísané do súborov pod názvom *SSTable* (Sorted String Tables), ktorých obsah je zoradený podľa hodnoty kľúčov a nie je možné ich modifikovať. Súborové SSTables sa zlievajú (merge sort) v pravidelných intervaloch na pozadí a táto operácia je neblokujúca. Počas zlievania SSTables dochádza k odstraňovaniu dát určených na vymazanie a generovaniu nových indexov. Indexy slúžia na rýchly prístup k dátam uloženým v SSTable. Zároveň sa generujú štruktúry pod názvom *Bloom filters*¹, ktoré sa využívajú pri čítaní dát. Požiadavku pre zápis dát je možné zaslať na ľubovoľný uzol v klastri.

5.6 Čítanie dát

V prípade požiadavky na načítanie dát, sa požadované dáta najprv hľadajú v štruktúrach Memtable. Ak sa dané dáta nenachádzajú v operačnej pamäti, vyhľadávanie sa uskutočňuje podľa kľúča v súboroch SSTable. Keďže snahou systému, je čo najefektívnejšie vyhľadávanie, využívajú sa bloom filtre. Bloom filtre sú nedeterministické algoritmy, ktoré dokážu otestovať, či hľadaný prvok patrí do množiny a generujú len falošné pozitíva. Pomocou nich je možné priradiť kľúče uložené v súboroch SSTables do bitových polí, ktoré je možné uchovať v operačnej pamäti. Vďaka tomu sa výrazne redukuje prístup na disk. Požiadavka pre čítanie dát môžeme zaslať na ľubovoľný uzol.

¹http://en.wikipedia.org/wiki/Bloom_filter

5.7 Zmazanie dát

Počas vykonania operácie reprezentujúcej zmazanie dát sa tieto dáta nevymažú okamžite. Namiesto toho sa vykoná operácia, ktorá dané dáta označí príznakom pod názvom *tombstone*. Po uplynutí doby, ktorá je štandardne nastavená na desať dní, sa tieto dáta odstraňujú pri procese zlievajúcom súbory SSTables.

5.8 Konzistencia

Konzistencia systému je maximálne konfigurovateľná a využíva princípy techník založených na protokoloch kvóra. Klient určuje hodnotu R , ktorá stanovuje počet replík pre čítanie dát. Operácia je úspešná v prípade dostupnosti týchto replík. To samé platí pre zápis dát, kde počet replík je určený parametrom W . Ak je splnený vzťah $R + W > N$, kde N je počet replík, tak výsledok operácie spĺňa definíciu silnej konzistencie, naopak v prípade vzťahu $R + W < N$, sa jedná o eventuálnu konzistenciu čím zaručíme vysokú dostupnosť.

5.9 Perzistentné úložisko

Cassandra využíva ako perzistentné úložisko dát lokálny súborový systém.

5.10 Bezpečnosť

Implicitne Cassandra nevyužíva žiadne prvky zabezpečujúce bezpečnostné mechanizmy. K dispozícii je modul, ktorý umožňuje nastavenie autentizácie na úrovni Keyspace-u pomocou textových hesiel alebo ich MD5 odtlačkov. Obmedzovanie prístupu k dátam je preto potrebné zabezpečiť na aplikačnej úrovni.

Kapitola 6

HBase

V úvode tejto kapitoly stručne popíšeme distribuovaný súborový systém, ktorý je súčasťou projektu Hadoop a zároveň slúži ako perzistentné úložisko pre distribuovanú databázu HBase. Následne popíšeme základné princípy fungovania tohoto databázového systému.

HDFS

Hadoop¹ je open source projekt vytvorený v programovacom jazyku Java, ktorý tvorí distribuovaný súborový systém Hadoop Distributed Filesystem (HDFS) a framework MapReduce pre spracúvanie objemu dát v desiatkach PB [39]. Medzi hlavné vlastnosti tohoto systému patria vysoká dostupnosť, škálovateľnosť a distribuovaný výpočet. Architektúra HDFS vychádza z princípov distribuovaného súborového systému Google File System (GFS) [25] a pôvod frameworku MapReduce [15] pochádza taktiež od spoločnosti Google.

Súborový systém využíva architektúru master-slave. HDFS predpokladá prácu so súbormi rádovo v desiatkach gigabajtov, ktoré sú interne reprezentované dátovými blokmi o štandardnej veľkosti 65 MB. Informácie o priradení blokov k súborom a ich umiestnenie na uzloch typu slave sú reprezentované pomocou metadát. Tieto metadáta sú uložené v operačnej pamäti uzla typu master, ktorý sa nazývaná *Namenode*. Uzly slave pod názvom *Datanode* slúžia ako fyzické úložisko dátových blokov, ktoré sú zároveň na týchto uzloch replikované. Štandardne je nastavená úroveň replikácie na hodnotu tri, každý blok je v systéme uložený trikrát.

Predtým ako klient vykoná operáciu zápisu alebo čítania dát, tak kontaktuje uzol *Namenode*, ktorý mu poskytne informácie, na ktorých uzloch sa nachádzajú bloky reprezentujúce súbor a dátová komunikácia následne prebehne medzi klientom a uzlami typu *Datanode*. HDFS je optimalizovaný pre jednorázový zápis dát a ich následné mnohonásobné čítanie.

Hlavným nedostatkom tejto infraštruktúry je fakt, že uzol *Namenode* tvorí kritický bod výpadku, v prípade jeho nedostupnosti nie je možné pracovať s HDFS. Prípadná strata alebo poškodenie dát na tomto uzle spôsobí totálne zlyhanie súborového systému bez možnosti jeho obnovy. Súborový systém nie je vhodný pre ukladanie veľkého počtu malých súborov. Uzol *Namenode* alokuje pre objekt typu blok a objekt typu súbor 300 B metadát. V prípade

¹<http://hadoop.apache.org/>

uloženia súboru, ktorý nepresahuje veľkosť jedného bloku je potrebné alokovať 300 B dát. Ak uložíme 10,000,000 takýchto súborov veľkosť metadát, ktoré udržiava Namenode v operačnej pamäti zaberie 3 GB. Celkový počet uložených súborov je obmedzený veľkosťou operačnej pamäti RAM, ktorou disponuje uzol Namenode.

Z týchto pozorovaní vyplýva fakt, že distribuovaný súborový systém HDFS nemá praktické využitie ako úložisko dát slúžiace k archivácii emailových správ, pre ktoré sme definovali požiadavky v tretej kapitole.

HBase

HDFS a MapReduce slúžia na dávkové spracúvanie obrovského objemu dát. HBase je open source, distribuovaný, stĺpcovo orientovaný databázový systém, ktorý umožňuje prácu s veľkým objemom dát v reálnom čase. Ako perzistentné úložisko dát využíva distribuovaný súborový systém HDFS, je taktiež implementovaný v Jave a jeho architektonické koncepty vychádzajú zo systému Bigtable od spoločnosti Google. HBase bol vytvorený spoločnosťou Powerset na konci roka 2006, pre potreby spracúvania obrovského objemu dát a začiatkom roka 2008 sa stal oficiálnym podprojektom systému Hadoop [46].

6.1 Dátový model

Dátový model je totožný s konceptom Bigtable. Dáta, s ktorými pracuje HBase sa ukladajú do tabuliek. Každá tabuľka obsahuje riadky identifikované kľúčom, ktoré môžu byť tvorené ľubovoľným počtom stĺpcov. Kľúče sú reprezentované ako bajtové pole. Pre zápis ich hodnoty môžeme použiť ľubovoľný typ dát, napríklad reťazec alebo serializovanú dátovú štruktúru pomocou JSON, XML. Riadky sú radené podľa názvov kľúčov, ktoré sú radené podľa hodnoty jednotlivých bajtov. Stĺpce sú organizované do skupín, ktoré sa nazývajú „rodiny stĺpcov“ (Column families). Obsahu každej bunky, ktorej pozíciu určuje riadok a stĺpec prináleží verzia, ktorá je reprezentovaná časovou značkou a jej obsah je reprezentovaný ako pole bajtov. Časovú značku určuje klient pri zápise dát alebo je automaticky generovaná systémom HBase (reprezentuje ju systémový čas). Obsah buniek tabuľky je možné sprístupniť pomocou kľúča a názvu stĺpca alebo pomocou kľúča, názvu stĺpca a časovej značky. V prípade uloženia viacerých verzií v danej bunke, sú tieto dáta radené od najaktuálnejšej časovej značky po najstaršiu. Región tvorí interval riadkov, kde posledný riadok do daného intervalu nepatrí.

Dôležitým faktom je, že stĺpce, ktoré patria do rovnakej rodiny stĺpcov sú fyzicky uložené na tom istom mieste. Rodiny stĺpcov je potrebné zdefinovať počas vytvárania tabuliek. Ich názvy a počet je potrebné vhodne premyslieť už pri samotnom návrhu databázovej schémy. V prípade, že klaster HBase obsahuje viacero uzlov je na nich potrebné zabezpečiť synchronizáciu systémového času. V prípade veľkého časového rozdielu na jednotlivých uzloch klastru hrozí, že sa daná inštancia systému HBase nespustí.

6.2 Rozdeľovanie dát

Základnou jednotkou, ktorá zabezpečuje rozdeľovanie dát a teda umožňuje horizontálne škálovanie a rovnomerné rozloženie záťaže v klastri je región. Región má náhodne vygenerovaný identifikátor. Tabuľka je tvorená regiónmi, pri jej prvotnom vytvorení ju zvyčajne reprezentuje jeden región. V prípade, že jej veľkosť dosiahne vopred stanovenú hranicu (túto hranicu definuje konfiguračný súbor), dôjde k rozdeleniu riadkov do dvoch nových regiónov s podobnou veľkosťou. Tento mechanizmus zabezpečuje, že do systému je možné uložiť tabuľku o veľkosti, ktorú by nebolo možné uložiť alebo spracovať pomocou jedného fyzického počítača z dôvodu hardvérových limitov. Región je základný element, ktorý zabezpečuje vysokú dostupnosť a rovnomerné rozloženie záťaže.

6.3 Architektúra systému

HBase využíva architektúru master-slave. Uzol v role master sa nazýva *HBaseMaster*, uzly slave *RegionServers* (RS). Pre chod systému je ďalej potrebná služba ZooKeeper², ktorá zabezpečuje výber nového uzla master v prípade havárie. *ZooKeeper* je samostatná centralizovaná služba, ktorá sa môže použiť u distribuovaných databázových systémov, kde sa počíta so zlyhaniami. Služba slúži napríklad pre správu konfiguračných informácií, distribuovanú synchronizáciu a iné. V prípade systému HBase sa odporúča pre stabilný beh použiť minimálne tri uzly pre beh tohto systému, čo pridáva ďalšiu vrstvu k HDFS.

HBaseMaster

Tento uzol v systéme vykonáva priradovanie regiónov RegionServer-om, detekuje pridanie nového RS, jeho zlyhanie a rovnomerne rozdeľuje záťaž na RS-och v prípade rozdelenia regiónu.

RegionServer

RegionServer slúži pre obsluhu klientských požiadaviek, samotný zápis alebo čítanie dát sa vykonáva medzi klientom a RS. Každý RS spravuje niekoľko regiónov, automaticky rozdeľuje regióny a informuje o tom uzol HBaseMaster. Tieto uzly môžeme v systéme ľubovoľne pridávať alebo odoberať počas jeho prevádzky.

6.4 Replikácia

Primárnu replikáciu dát, ktorá spĺňa vlastnosti silnej konzistencie a zabraňuje strate dát zabezpečuje perzistentné úložisko, v tomto prípade HDFS.

HBase podporuje replikáciu v rámci viacerých geograficky oddelených dátových centier. Replikácia funguje na rovnakom princípe ako v databázovom systéme MySQL³, teda master-slave a je asynchrónna. Táto forma replikácie zanáša do distribuovaného systému

²<http://zookeeper.apache.org/>

³<http://dev.mysql.com/doc/refman/5.1/en/replication-formats.html>

eventuálnu konzistenciu, pretože negarantuje, že dáta budú zmenené na všetkých uzloch slave v rovnakom čase.

6.5 Zápis dát a čítanie dát

V prípade zápisu alebo čítania dát klient kontaktuje ZooKeeper, od ktorého obdrží informáciu o lokácii tabuľky *-ROOT-* a následne kontaktuje daný RS obsahujúci túto tabuľku. Z tabuľky *-ROOT-* sa určí RegionServer, ktorý obsahuje tabuľku *.META.*, tieto kroky sa lokálne kešujú na strane klienta. Následne klient kontaktuje daný RegionServer a v tabuľke *.META.* vyhľadá uzol obsahujúci cieľový región, do ktorého patria hľadané alebo cieľové dáta. V poslednom kroku prebieha celá dátová komunikácia medzi klientom a posledným vyhľadaným uzlom.

Dáta sú zapísané do štruktúry *HLog* (typu *Write-Ahead-Log*), ktorá je uložená v súborovom systéme HDFS. Po potvrdení o úspešnom zápise sú data nahrané do štruktúry *MemStore*, ktorá je uschovaná v operačnej pamäti. Štruktúry *MemStore* sú zapisované obdobne ako u systému Cassandra do súborov typu *HFile*, ktorých obsah je zoradený a štruktúra je podobná ako u súborov *SSTable*. V prípade, že RegionServer obdrží požiadavku na čítanie dát, dáta sú načítané buď zo štruktúry *MemStore* alebo *HFile*. Pre zvýšenie výkonnosti je možné použiť Bloom filtre.

Zápis alebo čítanie dát na úrovni riadku identifikovaného pomocou kľúča je atomická operácia.

6.6 Zmazanie dát

Pri operácii zmazania dát je možné určiť vymazávané data konkrétnou časovou značkou, poprípade je možné vymazanie všetkých verzií dát, ktoré sú staršie ako zadaná časová značka. Zmena sa nevykoná okamžite z dôvodu, že obsah štruktúr *HFile* nie je možné modifikovať (jedným z dôvodov je aby sa zabránilo vykonávaniu nadbytočných diskových operácií). Namiesto toho sa vykoná operácia, ktorá dané dáta označuje príznakom pod názvom *tombstone*. Tieto dáta sa odstránia obdobne ako v systéme Cassandra, pri procese zlievajúcom súbory *HFile*.

6.7 Konzistencia

Systém sa vyznačuje silnou konzistenciou. Z modelu CAP splňuje CP, teda uprednostňuje konzistenciu pred dostupnosťou.

6.8 Perzistentné úložisko

Tento distribuovaný databázový systém je schopný pracovať v lokálnom móde, kde vyššie spomínané komponenty bežia ako samostatné služby na jednom fyzickom uzle a ako úložisko

dát sa využíva lokálny súborový systém. V prípade distribuovaného módu rozlišujeme dva typy:

- pseudo-distribuovaný mód, kde všetky komponenty bežia na jednom uzle
- distribuovaný mód, jednotlivé komponenty bežia na samostatných uzloch

Obidve konfigurácie môžu využívať ako perzistentné úložisko distribuovaný súborový systém HDFS, Kosmos Distributed File System (KFS) alebo Amazon S3. Štandardne sa doporučuje použitie HDFS.

6.9 Bezpečnosť

Hadoop a HBase aktuálne poskytujú prvok autentizácie pomocou služby Kerberos. Možnosť pridania autorizácie na úrovni tabuliek a rodiny stĺpcov je neustále vo vývoji [45].

Kapitola 7

Testovanie výkonnosti

Výkonové porovnanie NoSQL systémov je zložitá úloha, neexistuje univerzálny nástroj, ktorým by bolo možné tieto systémy navzájom porovnať. Každý z týchto systémov sa vyznačuje špecifickými vlastnosťami, medzi ktoré patria typ konzistencie, optimalizácia pre zápis alebo čítanie a ich výber závisí na konkrétnom prípade použitia. Všeobecný nástroj pre ich porovnanie by preto nemal žiadne opodstatnenie. Aktuálne nie sú k dispozícii žiadne všeobecné nástroje, ktorými by bolo možné testovať napríklad konzistenciu, spoľahlivosť a iné vlastnosti. Výkon týchto systémov ovplyvňuje faktor replikácie, spôsob rozdeľovania dát a typ konzistencie. Veľmi častou a zároveň časovo náročnou metódou, ktorá slúži na porovnávanie týchto systémov je implementácia daného riešenia s využitím všetkých porovnávaných systémov. V tejto kapitole sa zameriame na popis testov, ktoré sme vykonali v reálnych podmienkach.

7.1 Testovacie prostredie

Pre výkonnostné testovanie sme mali k dispozícii 9 počítačov s rovnakou hardvérovou a softvérovou konfiguráciou, ktoré boli navzájom prepojené pomocou 10 Gbit smerovača a komunikovali po 1 Gbit linke. Konkrétnu softvérovú konfiguráciu testovaných aplikácií popíšeme jednotlivo v nasledujúcich podkapitolách.

Hardvérová konfigurácia

- 4 jadrový procesor Intel, 5506@2.13Ghz
- 4 GB RAM
- 5 pevných diskov (SATA, 7200RPM) o veľkosti 1TB zapojených v poli RAID0
- 1 Gbit sieťová karta

Softvérová konfigurácia

Každý uzol obsahoval inštaláciu operačného systému Debian GNU/Linux Lenny x64 a aplikáciu Sun Java JDK 1.6.0_ +88. Na každom uzle bol deaktivovaný odkladací priestor (swap). Za účelom monitorovania bol použitý softvér Zabbix a nástroje VisualVM, htop, iostat a dstat.

Sieťová konfigurácia

Hodnota maximálnej reálnej sieťovej priepustnosti medzi dvoma uzlami bola zmeraná pomocou aplikácie nuttcp¹ s výslednou hodnotou 940 Mbit/s.

7.2 Popis testovacej metodológie

Nad oboma distribuovanými databázovými systémami sme vykonali testy, na základe ktorých sme pozorovali ako tieto systémy ovplyvňuje faktor replikácie a typ konzistencie. Pozorovali sme ich schopnosť horizontálneho škálovania a chovanie sa pod záťažou. Testy boli zamerané na operáciu zápisu dát, ktorá je kritickým prvkom pre potreby našej aplikácie.

7.2.1 Testovací klient

Testovacím klientom bola aplikácia využívajúca vlákna, založená na princípe producent - konzument, kde konzumenti predstavovali jednotlivé vlákna vykonávajúce zápis alebo čítanie dát. Optimálny počet paralelne zapisujúcich vlákien bol stanovený na hodnotu 50, pri ich navýšení sa zvyšovala hodnota latencie a nedošlo k zvýšeniu dátového toku pre zápis. Pre čítanie dát bolo použitých 250 vlákien. Dôležitým bodom bolo zabezpečiť rovnaké zaťaženie každého uzla v klastri počas celého priebehu jednotlivých testov. Detailný popis splňujúci tento bod je obsiahnutý v časti popisujúcej test konkrétneho databázového systému.

7.2.2 Testovací prípad pre zápis dát

V tomto testovacom prípade sme postupne do klastra obsahujúceho jeden, tri a šesť uzlov zapisovali 8 miliónov riadkov. Každý riadok obsahoval jeden stĺpec o veľkosti 1000 B, ktorého obsah tvorili náhodne vygenerované dáta. Tento počet zapisovaných riadkov sme zvolili z dôvodu, aby dochádzalo k zápisu štruktúr Memtable a Memstore na pevný disk. Vďaka týmto operáciám sa chovanie klastra viac priblížilo reálnym podmienkam.

7.2.3 Testovací prípad pre čítanie dát

Dôvodom tohto testovacieho prípadu bolo určiť priepustnosť pri čítaní dát z databázového systému. Táto hodnota bude mať výrazný vplyv na celkovú dobu trvania výpočtov štatistík, pomocou MapReduce. V tomto testovacom prípade sme náhodne načítali 1 milión riadkov z databázového systému o veľkosti 1000 B. Počet uzlov tvoriacich klaster bol jeden, tri a šesť.

¹<http://www.wcisd.hpc.mil/nuttcp/>

7.2.4 Zátťažový test

Cieľom bolo zistiť stabilitu klastra v prípade, keď bude pod sústavným zápisom dát, budú v ňom prebiehať operácie pre zlievanie (compaction) štruktúr SSTable a HTable na pevnom disku. Tento test sme vykonávali po dobu troch hodín. Náhodne generované dáta o rôznej veľkosti boli zapisované do jedného stĺpca. Počas niektorých testovacích prípadov sme použili dvoch klientov z dôvodu, aby sme zaručili maximálnu saturáciu šírky prenosového pásma a vylúčili úzke hrdlo na strane klienta (1 Gbit linka umožňuje maximálny teoretický dátový tok 125 MB/s).

7.3 HDFS

Nad distribuovaným súborovým systémom HDFS sme vykonali testy určujúce maximálnu hodnotu priepustnosti pri zápise dát, z dôvodu aby sme vylúčili možné úzke hrdlo v jeho prepojení s databázovým systémom HBase. Pre účely testovania sme použili verziu Hadoop-0.20.2, veľkosť haldy pre JVM (Java Virtual Machine) bola nastavená na hodnotu 1 GB.

Meranie sme vykonali v troch konfiguráciách. Každá konfigurácia obsahovala jeden uzol v role master, na ktorom boli spustené služby Namenode a JobTracker. Na zvyšných uzloch typu slave bežali služby Datanode a TaskTracker. Konfigurácia klastra bola nasledovná:

- A - tri uzly slave s faktorom replikácie jedna
- B - tri uzly slave s faktorom replikácie tri
- C - šesť uzlov slave s faktorom replikácie tri

Testy boli vykonané nástrojom TestDFSIO, ktorý je súčasťou zdrojových kódov systému Hadoop. Počas jednotlivých testov sme na HDFS zapisovali tri rôzne veľkosti súborov a bola zachovaná štandardná veľkosť bloku 65 MB. Pomocou techniky MapReduce, sa na každom uzle lokálne zapisoval funkciou typu *map* súbor o danej veľkosti a v prípade faktora replikácie bol replikovaný na zvyšné uzly. Každý test bol vykonaný trikrát a výsledná hodnota bola určená ako aritmetický priemer. Výsledky testu, ktoré zobrazuje tabuľka 7.1, reprezentujú maximálny tok pre zápis dát v klastri.

Veľkosť súboru [MB]	Klaster		
	A	B	C
65	247 MB/s	134 MB/s	186 MB/s
512	389 MB/s	135 MB/s	188 MB/s
2048	452 MB/s	138 MB/s	191 MB/s

Tabuľka 7.1: Priepustnosť pri zápise dát na HDFS

Na základe týchto hodnôt pozorujeme, že zvýšenie hodnoty replikácie má zásadný negatívny vplyv na celkový výkon distribuovaného súborového systému. Dôležitý fakt, ktorý vyplynul z výsledkov testovania je, že v prípade, ak zvýšime dvojnásobne počet uzlov v klastri (prípád klastrov v konfigurácii B a C), narastá priepustnosť pri zápise, čo potvrdzuje

vysokú škálovateľnosť systému. HDFS je optimalizovaný pre zápis veľkých súborov rádovo v desiatkách GB a túto vlastnosť test potvrdil. Vo všetkých prípadoch bola hodnota zápisu väčšia ako 125 MB/s.

7.4 HBase

Pre test distribuovaného databázového systému sme nainštalovali systém Hadoop 0.20.2 a HBase 0.90.1. Na jednom fyzickom uzle bežali nasledujúce služby:

- HBase Master
- ZooKeeper
- Namenode

Tieto služby sú v oboch systémoch súčasťou uzla master. Na zvyšných uzloch boli spustené služby RegionServer a Datanode. Veľkosť haldy pre JVM sme v prípade systému HDFS nastavili na 1 GB operačnej pamäti a v prípade systému HBase na 2 GB.

Prázdna tabuľka je po vytvorení v systéme HBase reprezentovaná jedným regiónom. Tento región je uložený na jednom uzle. K rozdeleniu tohto regiónu dochádza v prípade, ak objem dát zapísaných v tabuľke prekročí štandardne nastavenú hranicu s hodnotou 256 MB. Prázdnu tabuľku sme vytvorili pomocou programového rozhrania (API) systému HBase a to tak, že sme ju predrozdelili na počet regiónov, ktorý odpovedal počtu uzlov typu slave v klastri. Názvy kľúčov sme generovali pomocou náhodného generátora. Vďaka tejto metóde sme dosiahli rovnomerné zaťaženie všetkých uzlov po celú dobu testovania. Tabuľka 7.2 zobrazuje výsledky meraní podľa testovacieho prípadu pre zápis dát.

Počet uzlov	Faktor replikácie	Čas [sek]	Riadok/sek	Priepustnosť [MB/s]
1	1	551	14519	14
3	1	202	39613	39
3	3	317	25110	25
6	3	211	37864	37

Tabuľka 7.2: Hbase: zápis riadkov o veľkosti 1000 B

Škálovateľnosť

Z daných meraní vidíme, že systém je maximálne škálovateľný a dvojnásobne zvýšenie počtu uzlov zvýši priepustnosť o cca 50%.

Replikácia

Zvýšenie faktora replikácie (riadky 2 a 3) spôsobilo zníženie prenosovej rýchlosti o cca 37%.

Konzistencia

HBase vyžaduje silnú konzistenciu pre operácie čítania a zápisu dát, na úkor dostupnosti. Tento fakt sme zaznamenali počas testovania, keď v určitých intervaloch došlo k zlyhaniu operácie zápisu, ktorá bola následne zopakovaná.

Čítanie dát

Pri vytváraní tabuľky v systéme HBase chýba automatická podpora Bloom filtrov, ktoré sú neustále vo vývoji. Aktiváciu týchto filtrov je potrebné vykonať z príkazového interpreta, ktorý slúži pre manipuláciu so systémom HBase. Bloom filtre boli počas testu aktivované. Výsledky testovacieho prípadu pre čítanie dát zaznamenáva tabuľka 7.3.

Počet uzlov	Faktor replikácie	Čas [sek]	Riadok/sek	Priepustnosť [MB/s]
1	1	246	4069	4
3	1	117	8561	8
3	3	127	7936	8
6	3	85	11904	12

Tabuľka 7.3: HBase: čítanie riadkov o veľkosti 1000 B

Záťažový test

V tabuľke 7.4 sú znázornené výsledky záťažového testu.

Počet uzlov				
Veľkosť riadku	3	4	5	6
1 KB	32	35	36	38
10 KB	31	37	41	49
100 KB	35	43	52	55
512 KB	25	40	51	63
1 MB	35	47	53	68

Tabuľka 7.4: Hbase: maximálna priepustnosť klastra v MB/s

7.5 Cassandra

Pri testovaní klastru bol použitý distribuovaný databázový systém Cassandra verzie 0.7.3. Adresár obsahujúci súbory typu commitlog bol na samostatnom fyzickom disku, dátový adresár bol na diskoch zapojených v poli RAID0. Veľkosť štruktúry Memtable bola nastavená na hodnotu 120 MB. Každý uzol zaberá na hašovacom kruhu rovnaký, vopred nastavený úsek.

Tabuľka 7.5 obsahuje výsledky z viacerých meraní, podľa testovacieho prípadu pre zápis dát. Ako hodnoty kľúčov boli pre jednotlivé zapisované riadky použité prirodzené čísla z rozsahu nula až celkový počet riadkov. Cassandra sme nastavili tak, aby boli jednotlivé kľúče a k nim prináležiace dáta náhodne zapisované na jednotlivé uzly systému.

Bol použitý *RandomPartitioner*. Toto nastavenie zabezpečilo rovnomernú záťaž každého uzla počas celkovej doby zápisu.

Počet uzlov	Faktor replikácie	Konzistencia	Čas [sek]	Riadok/sek	Priepustnosť [MB/s]
1	1	One	338	23669	23
3	1	One	207	38647	38
3	3	One	311	25723	25
3	3	Quorum	351	22792	22
6	3	One	202	39604	39
6	3	Quorum	263	30418	30

Tabuľka 7.5: Cassandra: Zápis riadkov o veľkosti 1000 B

Škálovateľnosť

Z výsledkov meraní je vidieť, že tento distribuovaný databázový systém je maximálne škálovateľný z pohľadu rýchlosti zápisu. V prípade, keď sme zdvojnásobili počet uzlov z troch na šesť (riadky 3 a 5) vzrástla priepustnosť zápisu o cca 54%.

Replikácia

V prípade zvýšenia faktora replikácie z 1 na 3 (riadky 2 a 3) sa automaticky znížila rýchlosť zápisu o jednu tretinu.

Konzistencia

Počas zápisu s konzistenciou kvóra, ktorá zabezpečuje silnú konzistenciu databázového systému, sa rýchlosť znížila podľa očakávaní. V tomto prípade, aby klient obdržal odpoveď o úspešnom zápise museli byť dáta zapísané na celkový počet replík $\lfloor N/2 \rfloor + 1$, kde N označuje počet uzlov v klastri. Počas zápisu v prípade konzistencie One, klient obdržal potvrdenie o úspešnosti po zápise na jeden uzol.

Čítanie dát

Výsledky testovacieho prípadu pre čítanie dát sú zaznamenané v tabuľke 7.6. Riadky boli čítané v náhodnom poradí a kešovanie kľúčov a riadkov, ktoré Cassandra podporuje bolo vypnuté. Počas tejto operácie boli automaticky aplikované Bloom filtre.

Záťažový test

Výsledky záťažového testu zobrazuje tabuľka 7.7, kde jednotlivé položky zobrazujú priemerný dátový tok počas doby testovania v MB/s. Počas testu sme klaster monitorovali a

Počet uzlov	Faktor replikácie	Konzistencia	Čas [sek]	Riadok/sek	Priepustnosť [MB/s]
1	1	One	383	2615	2.5
3	1	One	211	4745	4.6
3	3	One	51	19630	19
3	3	Quorum	103	9750	10
6	3	One	32	30903	30
6	3	Quorum	59	16924	17

Tabuľka 7.6: Cassandra: Čítanie riadkov o veľkosti 1000 B

zistil viacero závažných dôsledkov. Na všetkých uzloch prebiehali veľmi časté GC kolekcie (Garbage collections) z dôvodu častého zápisu štruktúr Memtable na pevný disk. Podľa hardvérovej špecifikácie pre systém Cassandra všetky uzly disponovali minimálnou veľkosťou operačnej pamäti 4 GB. Z dôvodu zabezpečenia stability systému bola horná hodnota pri ktorej sa zapisuje Memtable z operačnej pamäti na disk 120 MB. Následkom tohto nastavenia vznikalo veľké množstvo SSTable súborov na pevnom disku, ktoré Cassandra zlievala na pozadí (compactions), čo spôsobovalo záťaž vstupno výstupných operácií (I/O wait). V prípade, takto zataženého systému a veľkého množstva SSTable súborov by bola operácia čítania veľmi pomalá, pretože dáta patriace do jednej rodiny stĺpcov by boli uložené vo veľkom množstve samostatných súborov a ich načítanie by vyžadovalo zvýšené množstvo diskových operácií (seek). Celkový počet uzlov by preto bolo potrebné navýšiť.

Z tohoto testu ďalej vyplynulo pozorovanie, že v prípade zápisu malých súborov rádovo v kB, je hlavným úzkym hrdlom systému CPU, kdežto v prípade zápisu veľkých blokov dát sú to V/V diskové operácie.

Počet uzlov				
Veľkosť riadku	3	4	5	6
1 KB	18	28	30	33
10 KB	63	77	93	118
100 KB	71	92	111	134
512 KB	67	87	109	127
1 MB	62	92	100	126

Tabuľka 7.7: Cassandra: maximálna priepustnosť klastra v MB/s

7.6 Voľba databázového systému

Primárnou požiadavkou aplikácie je zvládať vysoký tok pre zápis dát a nízkonákladová administrácia. V oboch týchto prípadoch prevládajú vlastnosti systému Cassandra nad systémom

HBase. Systém HBase má okrem iného zložitú infraštruktúru, ktorá je závislá na externých službách ako HDFS a ZooKeeper. Naproti tomu decentralizácia systému Cassandra prináša vďaka svojej jednoduchosti mnoho výhod. Aj napriek tomu, že systém HBase neobsahuje kritický bod výpadku, čo zabezpečuje služba ZooKeeper, problematickým článkom je perzistentné úložisko HDFS, ktoré tento požiadavok nespĺňa. Výhodou Cassandra je podpora indexov, ktoré sa vytvárajú asynchrónne na pozadí a nie je potrebné na aplikačnej úrovni zabezpečovať ich správu pomocou samostatnej rodiny stĺpcov. Voľba úrovne konzistencie je ďalšou z výhod, ktorú môžeme využiť v prípade, ak k dátam budú pristupovať koncoví používatelia, pre získanie stručného prehľadu správ obsiahnutých v emailovej zložke (inbox).

Cassandra dosahovala vyššiu priepustnosť pre zápis dát, ktorú sme zmerali v záťažových testoch s obmedzeným počtom uzlov, ktoré sme mali k dispozícii. V prípade použitia šiestich uzlov sme prekročili hranicu 125 MB/s, teda sme boli schopní zápisu šírky pásma 1 Gbit, čo splňuje požiadavku na náš systém.

Domnievame sa, že jedným z možných obmedzení výkonnosti systému HBase môže byť fakt, že štruktúra HLog, ktorá zabezpečuje trvácnosť dát sa spolu so štruktúrami MemStore zapisuje do súborového systému HDFS, čo spôsobuje vyššiu záťaž na strane diskových operácií. Záťaž fyzického disku je dvojnásobná v porovnaní so systémom Cassandra. Cassandra tento problém rieši efektívne, pretože štruktúra commitlog, zaručujúca trvácnosť dát, môže byť uložená na fyzicky odlišnom pevnom disku, ako disk na ktorý sa následne zapisujú súbory SSTable. Systém HBase sa tento nedostatok snaží riešiť napríklad spôsobom, kde je možné pomocou programového rozhrania pri zápise dát nastaviť aby tieto dáta neboli zapisované do štruktúry HLog. V tomto prípade hrozí strata dát a systém nezaručuje vlastnosť trvácnosti (durability).

Kapitola 8

Návrh systému

V tejto kapitole popíšeme návrh systému, ktorý bude slúžiť na archiváciu emailov a splňať požiadavky, ktoré sme pre tento systém definovali. V prvej časti sa zameriame na popis štruktúry a zberu dát, ďalej navrhujeme databázovú schému. V druhej časti popíšeme výber vhodných open source nástrojov, ktoré využijeme pre implementáciu prototypu. V závere popíšeme dosiahnuté výsledky v testovacom prostredí, ktoré preukážu vhodnosť využitia NoSQL systému Cassandra pre riešenie tejto úlohy.

8.1 Zdroj dát

Základným prvkom, ktorý budeme v našom systéme archívovať je emailový objekt, ktorý definuje dokument RFC 2821 [2]. Tento objekt pozostáva z SMTP obálky a emailovej správy. Obálka obsahuje informácie, ktoré sú potrebné pre korektné doručenie správy pomocou emailového servera a patria tam napríklad odosielateľ emailového objektu a jeden alebo viacerí príjemcovia. Emailová správa predstavuje semištrukturovaný dokument [40] v textovej podobe, ktorého syntax popisuje štandard RFC 2822 [1] z roku 2001 pod názvom Formát Internetovej správy (Internet Message Format). Dokument RFC 2822 nahradzuje a upravuje pôvodné RFC 822 pod názvom Štandard pre formát Internetových textových správ ARPA z roku 1982 (Standard for the Format of ARPA Internet Text Messages). Obsah emailovej správy delíme na hlavičku a telo, ktoré sú od seba oddelené znakom reprezentujúcim prázdny riadok. Telo správy nie je povinné. Štruktúru tela správy a polia v hlavičke rozširujú štandardy, pod názvom MIME (Multipurpose Internet Mail Extensions), RFC 2045, RFC 2046 [19, 20], RFC 2047, RFC 2048 a RFC 2049. Tieto štandardy pridávajú možnosť použitia iných znakových sád (štandardná sada US-ASCII), ďalej umožňujú štruktúrovať telo správy (vnorené správy rfc822), definujú formát a typy pre zasielanie príloh atď.

Zber emailových objektov budeme realizovať na unixových serveroch, ktoré používajú emailový server Qmail¹. Tento server zároveň realizuje antispamovú kontrolu pomocou modulu Qmail-scanner², ktorý je naprogramovaný v jazyku Perl³. Po doručení emailového objektu na server je emailový objekt spracovaný programom Qmail, ktorý volá obslužný

¹<http://cr.yp.to/qmail.html>

²<http://qmail-scanner.sourceforge.net>

³<http://www.perl.org>

modul `qmail-scanner.pl` a následne dokončí doručenie správy. Tento modul sme vhodne modifikovali pre potreby našej aplikácie. Výstupom je súbor s príponou `.envelope`, ktorý obsahuje viacero štatistických údajov znázornených na obrázku 8.1. Detailný popis tejto štruktúry sa nachádza na webovej adrese <http://qmail-scanner.sourceforge.net/>. Výstupom aplikácie Qmail je, v prípade úspešného doručenia emailu, textový súbor reprezentujúci emailovú správu, ku ktorej je jednoznačne priradený súbor s príponou `.envelope`. Oba súbory sú spracúvané analyzátorom, ktorý popíšeme v nasledujúcej časti. Súčasťou prílohy tejto práce je modifikovaný súbor `qmail-scanner.pl`.

```
Tue, 15 Mar 2011 10:12:09 CET Clear:RC:1(88.208.65.55):SA:1 0.007811 9508
odosielatel@server prijemca@server2 predmet <1300180228103914546@aq>
1300180329.16836-0.forid1:5987 priloha1:134
```

Obr. 8.1: Obsah obálky z programu Qmail-scanner

Zo súboru s príponou `.envelope` sme použili nasledujúce údaje:

- dátum a čas prijatia emailového objektu serverom Qmail
- pole SA:1, kde hodnota 1 reprezentuje spam, inak nula
- čas spracovania antispamovým filtrom
- veľkosť emailu
- polia príjemca a odosielateľ

8.2 Analýza dát

Jednou z hlavných požiadaviek systému je deduplikácia príloh emailových správ z dôvodu úspory diskovej kapacity. Hlavička s názvom *Content-Type*, ktorú definuje RFC 2045 špecifikuje typ dát v tele MIME správy. Jej hodnota je tvorená z dvoch častí a to názov typu média (media type) a bližšie špecifikovaný podtyp, napríklad „*image/gif*“. Norma definuje päť základných typov médií a to text, image, audio, video a application. V prípade, našej aplikácie má zmysel využiť deduplikáciu na všetky tieto typy s výnimkou typu „*text/plain*“, kde predpokladáme, že sa jedná o bežnú textovú správu napísanú užívateľom. Obsah tejto správy bude slúžiť pre fulltextové vyhľadávanie.

Program pre analýzu a deduplikáciu emailovej správy bol napísaný v programovacom jazyku Python⁴. Tento program dodržiava špecifikáciu RFC 2822 a RFC2045. Medzi povinné polia hlavičky emailu patria pole *From* a *Date*. Aj napriek tomu, že tieto polia sú definované už od roku 1982 v RFC 1982 analýza nášho testovacieho vzorku preukázala prítomnosť správ, ktoré túto požiadavku nespĺňali, jednalo sa hlavne o správy typu spam. Z množiny, ktorá obsahovala približne 1,000,000 emailových správ bolo 0.022% emailov, ktoré nespĺňali štruktúru definovanú normou RFC 2045. Metódu deduplikácie sme riešili nasledujúcim postupom:

⁴<http://python.org>

- programom pre analýzu emailu sme určili časti, v ktorých sa nachádzajú prílohy
- H je výstup hašovacej funkcie SHA2 nad dátami reprezentujúcimi prílohu, ktorý sme použili ako unikátny identifikátor prílohy
- dáta reprezentujúce prílohu v emailu sme nahradili značkou v tvare MARK:H
- dáta reprezentujúce prílohu sme do databázy uložili pod kľúčom H

8.3 Databázová schéma

Databázovú schému sme navrhli s ohľadom na to, aké operácie nad danými dátami budeme vykonávať a pri návrhu sme využili poznatky získané štúdiom architektúry databázového systému Cassandra. Schéma je tvorená pomocou štyroch rodín stĺpcov:

- *messagesMetaData* - obsahuje meta informácie identifikované v obálke emailového objektu a emailovej správy, nad ktorými budeme vykonávať štatistické výpočty pomocou metódy MapReduce
- *messagesContent* - obsahuje obálku, hlavičku a telo správy
- *messagesAttachment* - slúži na ukladanie deduplikovaných príloh emailových správ
- *lastInbox* - v chronologickom časovom poradí, podľa hodnoty poľa *Date* v hlavičke emailu, zaznamenáva správy daného užívateľa

Tradičné techniky pre popis databázových schém nie je možné aplikovať na databázové systémy, ktoré vychádzajú z konceptov Bigtable alebo Dynamo. Jedným z dôvodom je, že na tieto schémy sa aplikuje denormalizácia, duplikácia dát a kľúče sú často komplexného charakteru. Dodnes neexistuje žiadny štandard, ktorý by definoval popis týchto schém. Článok pod názvom: „Techniky pre definíciu štruktúr pomocou diagramov v cloude a návrhové vzory“ (Cloud data structure diagramming techniques and design patterns [13]) definuje stereotypy pre diagramy v jazyku UML a obsahuje vzory pre popis štruktúry týchto dát. Obrázok 8.2 znázorňuje databázovú schému nášho modelu a využíva techniky popísané v spomínanom článku.

Ako jedinečný identifikátor emailovej správy v databáze využívame nasledujúcu schému:

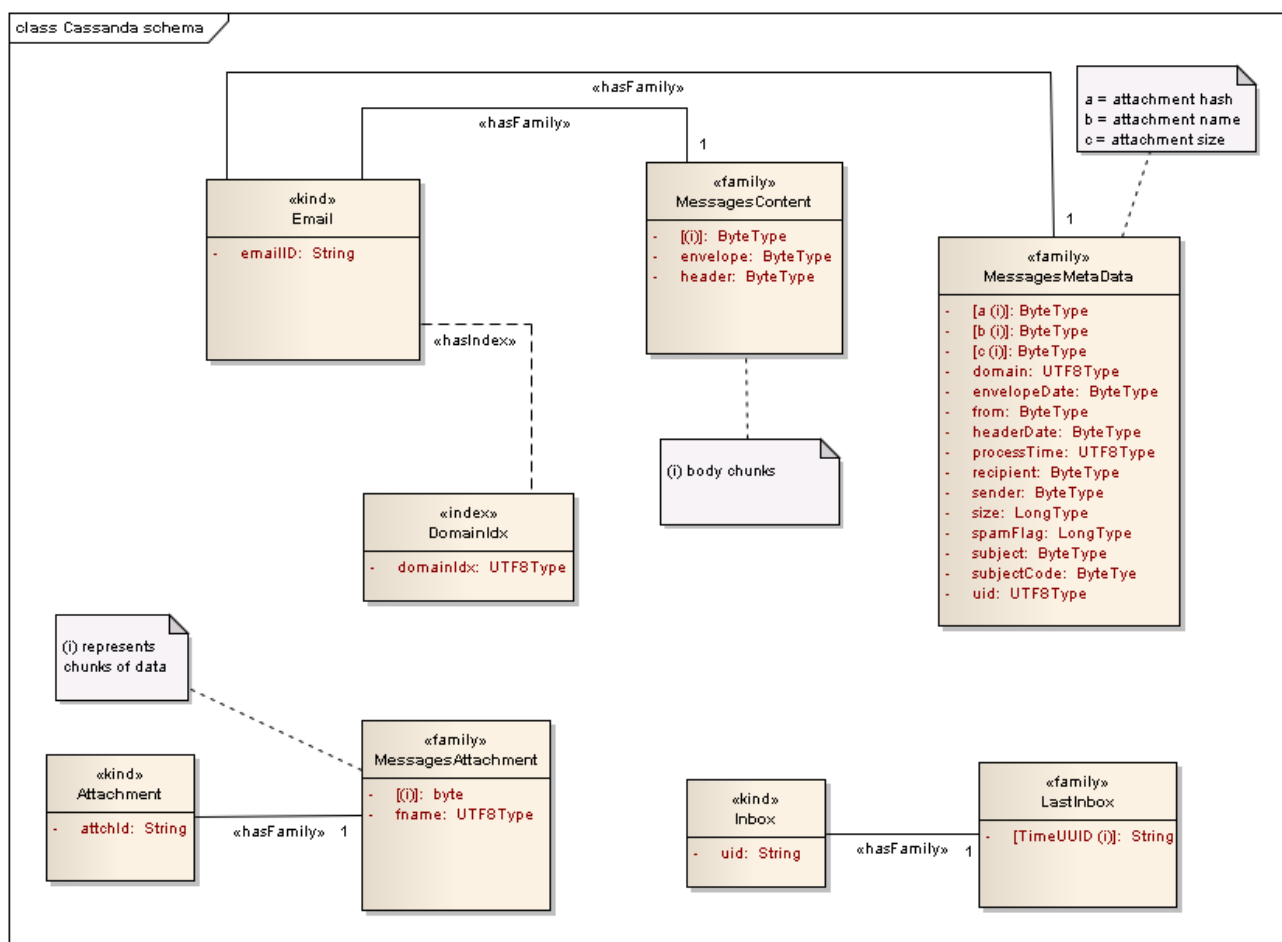
```
emailID = sha256(uid + MessageId + date)
```

V tejto schéme reprezentuje:

- *uid* emailovú adresu príjemcu, v tvare jan@mak.com
- *MessageID* je identifikátor správy z hlavičky daného emailu
- *date* je časová značka reprezentujúca čas, kedy bol email prijatý emailovým serverom (formát: rok, mesiac, deň, hodina, minúta, sekunda)

- *emailID* je výstup hašovacej funkcie SHA2 v hexadecimálnom tvare

V predchádzajúcej kapitole sme zistili, že Cassandra nie je optimalizovaná pre zápis blokov dát (blob), ktorých veľkosť presahuje 1 MB, avšak optimálne výsledky pre zápis dosahuje pri veľkosti blokov 512 kB. V prípade ak dáta reprezentujúce telo emailovej správy alebo objekt s prílohou presahujú veľkosť 1 MB, tak ich zapisujeme do samostatných stĺpcov o veľkosti 512 kB. Toto rozdeľovanie dát na menšie bloky vykonávame na aplikačnej úrovni na strane klienta zapisujúceho dáta do databázového systému. Názvy stĺpcov číslujeme vzostupne v intervale $0-N$, kde N je počet blokov. Spätnú rekonštrukciu dát vykonáva klient.



Obr. 8.2: Databázová schéma

Výsledný návrh databázovej schémy ma viacero výhod. Metadáta, nad ktorými budeme vykonávať výpočet štatistík (*messagesMetaData*) budú uložené v jednom súbore, čo zabezpečí ich efektívne spracovanie. V tejto rodine stĺpcov ďalej využívame index s názvom *domainIdx*, vďaka ktorému sme schopný získať všetky emaily uložené v systéme prináležiace určitej doméne, v tvare napríklad „cvut.cz“.

Všetky emailové správy, ktoré prináležia užívateľovi, ktorý je identifikovaný pomocou svojej emailovej adresy budú uložené v jednom riadku v rodine stĺpcov *lastInbox*. Názvy

stĺpcov budú reprezentované pomocou *TimeUUID*, čo zabezpečí ich zoradenie podľa času. *TimeUUID* je univerzálny identifikátor (UUID) využívajúci časovú značku. Túto rodinu stĺpcov môžeme použiť v aplikácii pre sprístupnenie obsahu emailového adresára koncovým užívateľom.

8.4 Fulltextové vyhľadávanie

Fulltextové vyhľadávanie realizujeme pomocou samostatného NoSQL systému ElasticSearch⁵. Hlavný index, ktorý obsahuje všetky indexované dáta ma názov *emailArchive*, a delíme ho na dva typy s názvom *email* a *envelope*. Schéma týchto typov obsahuje polia, podľa ktorých chceme v emailovom archíve vyhľadávať, a jej reprezentáciu zapísanú vo formáte JSON znázorňuje obrázok 8.3.

```
mappingsEmail = {
  "inbox": {"type": "string"},
  "from": {"type": "string"},
  "subject": {"type": "string"},
  "date" : {"type": "date"},
  "messageID" : {"type": "string", "index": "not_analyzed"},
  "attachments": {"type": "string"},
  "size": {"type": "long", "index": "not_analyzed"},
  "body": {"type": "string"}
}

mappingsEnvelope = {
  "sender": {"type": "string"},
  "recipient": {"type": "string"},
  "ip": {"type": "ip"},
  "date": {"type": "date"}
}
```

Obr. 8.3: JSON schéma pre fulltextové vyhľadávanie

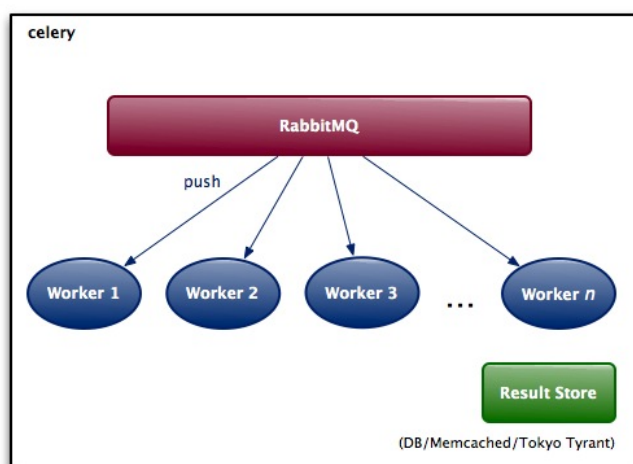
⁵<http://elasticsearch.org>

8.5 Implementácia

V programovacom jazyku Python sme implementovali klienta pre zápis dát do databáze Cassandra a Elasticsearch. Jednou z najdôležitejších vlastností týchto klientov je voľba úrovne konzistencie pri zápise. Našou prioritou je integrita dát a od databáze požadujeme silnú konzistenciu. Zvolili sme mód kvóra (quorum), ktorý zabezpečí zápis dát na $\lfloor N/2 \rfloor + 1$ replík a klient následne obdrží potvrdenie o úspešnosti zápisu, inak sa zápis zopakuje. Tieto vlastnosti nám zabezpečujú v prípade použitia faktoru replikácie tri (dátá sa v databázovom systéme nachádzajú trikrát) silnú úroveň konzistencie na strane klienta a databázového systému. Zápis do systému Cassandra využíval protokol Thrift a do systému Elasticsearch sa zapisovali dáta protokolom REST. Analýza emailovej správy a jej deduplikácia spotrebuje hlavne CPU zdroje. Moderné procesory obsahujú viacero jadier, čo môžeme využiť pre paralelné spracúvanie emailov, kde každé jadro CPU bude spracúvať jednu emailovú správu.

Celery

Paralelizáciu našej aplikácie sme zabezpečili pomocou využitia asynchrónnej fronty úloh pod názvom Celery⁶, ktorá využíva architektúru distribuovaného predávania správ (distributed message passing). Architektúru znázorňuje obrázok 8.4. „Pracovníci“ (workers) reprezentujú samostatné procesy (v našom prípade proces pre analýzu a deduplikáciu emailu), ktoré môžu bežať paralelne. Ako sprostredkovateľ (broker) je použitá aplikácia RabbitMQ⁷. Sprostredkovateľ obdrží od aplikácie qmail-scanner správu a uloží ju do fronty. Správa obsahuje identifikátor emailovej správy, v tomto prípade cestu na lokálnom súborovom systéme k súboru reprezentujúcemu email.



Obr. 8.4: Architektúra Celery, Zdroj: [44]

Táto správa je následne zaslaná ľubovoľnému pracovníkovi (v našom prípade klient vykonávajúci analýzu a deduplikáciu emailu), ktorý ju spracuje. Táto architektúra je plne dis-

⁶<http://celeryproject.org>

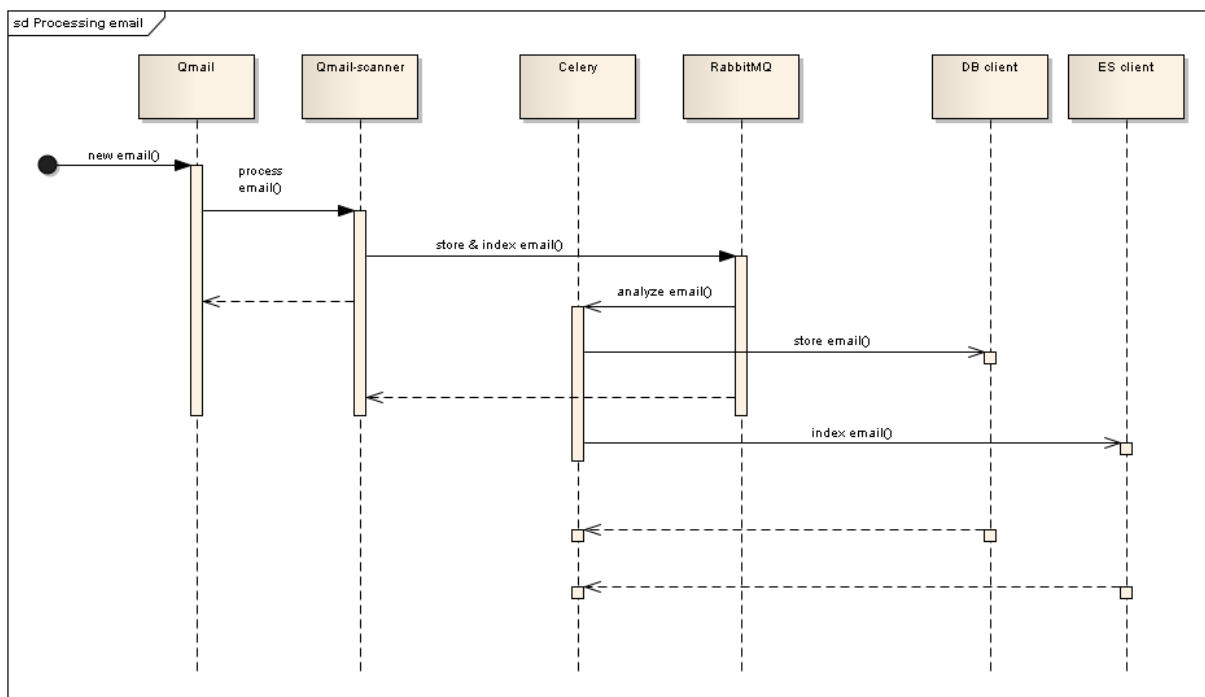
⁷<http://www.rabbitmq.com>

tribuovaná a dokáže odolávať chybám (napr. v prípade výpadku elektrickej energie správy naďalej pretrvávajú vo fronte).

Frontu RabbitMQ sme podrobili výkonnostnému testu, zvládala približne 10,000 operácií za sekundu pre zápis (publikovanie správ) a cca 5,000 operácií pre načítanie správ (konzumáciu správ).

8.5.1 Klient

Proces spracovania nového emailu je znázornený pomocou sekvenčného diagramu na obrázku 8.5. Pri príchode nového emailu, ktorý je spracovaný emailovým serverom Qmail, je vytvorená nová úloha pomocou aplikácie Celery. Táto úloha uloží do sprostredkovateľa identifikátor emailu. V prípade, že je v daný okamžik k dispozícii ľubovoľný pracovník, je emailová správa spracovaná pomocou analyzátora a následne zapísaná do databázy Cassandra a fulltextového systému Elasticsearch. Na testovacie účely sme nemali k dispozícii reálny dátový tok emailov. Vrstvu reprezentujúcu Qmail sme nahradili modulom, vytvárajúcim nové úlohy prechádzaním lokálneho súborového systému, ktorý obsahoval testovaciu množinu emailových správ.



Obr. 8.5: Spracovanie emailu

Pomocou klienta pycassa⁸ sme implementovali aplikáciu pre prístup k dátam uloženým v databázovom systéme. Klient umožňuje nasledovné operácie:

- načítanie pôvodnej štruktúry emailu, ktorý sme uložili do systému; prípadná deduplikácia je nahradená dátami reprezentujúcimi prílohu

⁸<http://pycassa.github.com/pycassa/>

- načítanie voliteľného počtu emailov pre daného užívateľa, ktoré sú zoradené chronologicky od najstaršieho
- náhodný výpis definovaného počtu emailov pre daného užívateľa
- výpis stručných informácií reprezentujúcich email, konkrétne polia: odosielateľ, predmet, dátum, veľkosť emailu, počet príloh

8.5.2 Výpočet štatistík

Cassandra spolupracuje so systémom Hadoop, čo nám dáva do rúk mocný nástroj na masívne paralelné spracovanie dát pomocou techniky MapReduce. V tomto prípade je potrebné na uzly, na ktorých je Cassandra nainštalovať HDFS a Hadoop. Tvorba aplikácií v tomto frameworku prebieha v Jave, je náročná a okrem toho programový model MapReduce obsahuje viacero problémov. Model napríklad neobsahuje primitíva na filtrovanie, agregáciu, spájanie dát a je potrebná ich vlastná implementácia. Tieto nedostatky rieši nástroj Pig [33], vďaka ktorému sme boli schopný efektívne spracúvať metadáta uložené v databáze. Obrázok 8.6 zobrazuje programovú ukážku, ktorá slúži na výpočet veľkosti najväčšieho emailu pre domény, ktorých emaily archivujeme. Pre jednoduchosť sme vynechali časti, ktoré slúžia na načítanie dát z databázy a obsahujú užívateľsky definovanú funkciu v programovacom jazyku Java, ktorá slúži na predprípravu vstupných dát. Podpora užívateľom definovaných funkcií je jednou z ďalších výhod nástroja Pig.

```
notSpam = FILTER grp BY group.spam == 1;
maxSize = foreach grp {
    size = rows.size;
    generate group, MAX(size);
};
STORE maxSize into 'biggestEmailPerDomainDomain' using PigStorage(',');
```

Obr. 8.6: Programová ukážka v jazyku Pig

8.5.3 Webové rozhranie

Vďaka tomu, že klient pre prístup k dátam, uloženým v systéme Cassandra, bol vytvorený v programovacom jazyku Python, sme pomocou webového frameworku Django⁹ vytvorili jednoduchú prezenčnú vrstvu, ktorou sme boli schopný pristupovať k uloženým emailovým správam pomocou webového rozhrania.

8.6 Overenie návrhu

Pomocou vyššie popísaného návrhu a implementovaných nástrojov sme overili funkčnosť nami navrhovaného modelu. Vhodná voľba daných verzií u aplikácií Celery a RabbitMQ

⁹www.djangoproject.com/

bola určená počas písania a ladenia aplikácie. Všetky tieto aplikácie sú neustále vo vývoji, to isté platí pre databázu Cassandra a systém Hadoop. Počas písania tejto práce prebehlo viacero rozhovorov s autormi týchto aplikácií. Konkrétne databáza Cassandra na začiatku práce neobsahovala takmer žiadnu ucelenú dokumentáciu, počas začiatkov experimentov sme začínali s verziou 0.7.0. Počas ukončovania tejto práce je k dispozícii aktuálna verzia 0.7.5 a medzitým vznikla kvalitná online dokumentácia od spoločnosti Datastax¹⁰.

Návrh možnej realizácie daného riešenia je zachytený pomocou diagramu nasadenia na obrázku B.1. V tomto prípade uvažujeme dve geograficky oddelené dátové centrá. Prvé centrum bude primárne slúžiť na obsluhu klientských požiadaviek, v druhom bude umiestnená tretia replika, ktorá bude slúžiť aj ako záloha a zároveň nad týmito dátami budú prebiehať MapReduce výpočty.

Konfigurácia

Hardvérová konfigurácia bola totožná s konfiguráciou z kapitoly 7. Na šiestich serveroch bola nainštalovaná databáza Cassandra 0.7.3, Hadoop 0.20.2, dvojica serverov obsahovala klientskú aplikáciu pre zápis dát, ktorú používala aplikácia Celery 2.6. V úlohe sprostredkovateľa bola použitá aplikácia RabbitMQ 2.1.1, ktorá bola nainštalovaná na samostatnom serveri.

Overenie integrity dát

Databázový klaster sme naplnili testovacími dátami obsahujúcimi emaily o objeme cca 300 GB. Tie isté dáta sme zároveň spracovali a zapísali do systému elasticsearch. Následne sme simulovali prípad obnovy dát z archívu, kde sme všetky emaily v náhodnom poradí z databázy načítali, zostavili ich do pôvodného tvaru klientskou aplikáciou a porovnali sme ich odtlačok pomocou hašovacej funkcie MD5 s odtlačkom pôvodných dát uložených na pevnom disku. Tento test prebehol bez akejkoľvek chyby. Klient, ktorý slúžil na čítanie dát z databázy využíval mód kvóra, z dôvodu požiadavku na integritu dát.

Dosiahnuté výsledky

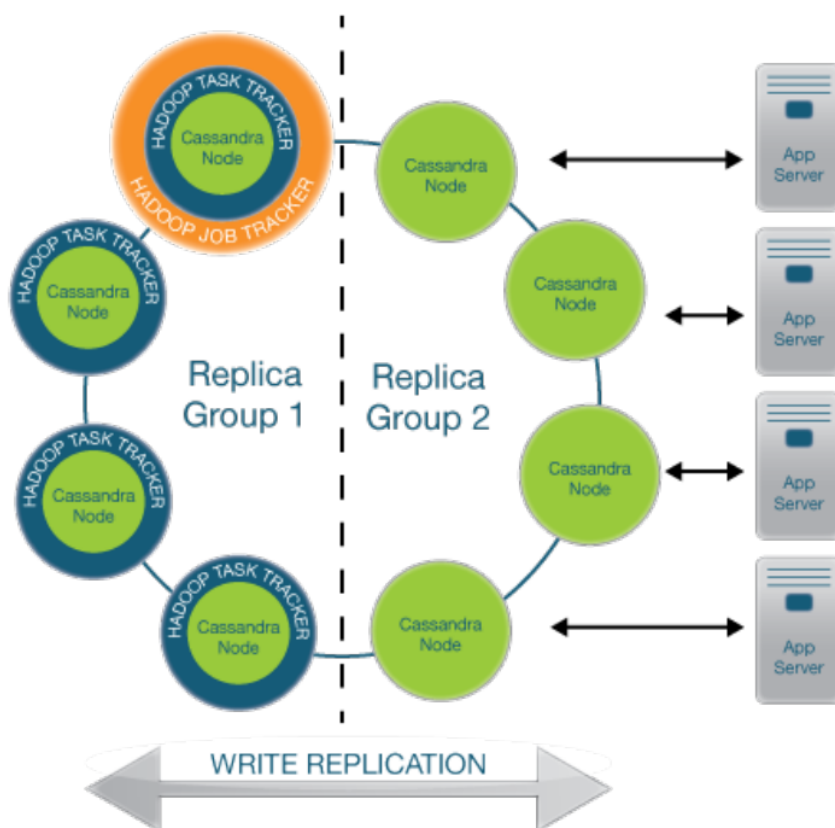
Dôležitým pozorovaním bol fakt, že z celkového objemu emailových správ cca 300 GB sa po deduplikácií príloh tento objem znížil na cca 99 GB, teda došlo k 67% úspore diskovej kapacity. Štruktúry, do ktorých sme ukladali dáta pre potrebu štatistík zaberali 0,4% z celkového objemu dát, čo je zanedbateľná položka.

8.7 Doporučenie najvhodnejšieho systému

V závere tejto práce bol spoločnosťou Datastax uvoľnený open source projekt Brisk [42]. Tento projekt kombinuje Hadoop, Hive a Cassandru. Jeho hlavnou výhodou je, že úplne eliminuje kritický bod výpadku u vrstvy HDFS. Ďalej maximálne uľahčuje správu tohto

¹⁰<https://datastax.com>

systému, pretože nie je potrebná inštalácia systému Hadoop, ZooKeeper a zvyšných aplikácií. Systém je maximálne decentralizovaný. Systém umožňuje súčasný beh aplikácií závislých na operáciách v reálnom čase a zároveň je možné vykonávať časovo náročné analytické operácie. Nie je potrebný žiadny prenos dát medzi viacerými systémami. Možnú architektúru použitia znázorňuje obrázok 8.7, kde vďaka využitiu replikácii všetky uzly obsahujú rovnaké dáta. Prvá skupina uzlov obsluhuje aplikácie v reálnom čase, kdežto na druhej skupine prebiehajú nad týmito dátami náročné analytické operácie pomocou MapReduce. Podobnú architektúru sme navrhovali v našom riešení.



Obr. 8.7: Brisk, Zdroj: [12]

Kapitola 9

Záver

Cieľom tejto práce bolo vybrať vhodný distribuovaný databázový systém, ktorý bude schopný spracúvať milióny emailových správ a odolávať sieťovým prerušeniam. Následne navrhnuť model systému a implementovať prototyp riešenia. Prototyp je implementovaný v jazyku Python, využíva systém Cassandra a bol naplnený približne miliónom emailových správ z reálneho prostredia, cieľ práce bol teda splnený. Práca pozostávala z viacerých častí, ktoré boli dôkladne spracované.

Kľúčovým prvkom pre preniknutie do danej problematiky distribuovaných databázových systémov bolo štúdium dostupných materiálov, medzi ktoré patrili hlavne technické články. Hlavným teoretickým prínosom tejto časti je ucelené zhrnutie základných konceptov a možných problémov, ktoré je treba zväžiť pri návrhu aplikácií využívajúcich distribuované databázové systémy. Taktiež sme popísali kľúčové časti databázových systémov Cassandra a HBase, ktorých znalosť je nutnou podmienkou v prípade ich použitia. Všetky použité zdroje zaoberajúce sa touto problematikou sú v angličtine, čím táto práca napísaná v slovenčine zaplňa diery v tejto oblasti. Zásadným problémom bola nepostačujúca dokumentácia týchto systémov, keďže sa jedná o relatívne mladé systémy, ktoré sú neustále vo vývoji. Množstvo informácií bolo preto diskutovaných v mailinglistoch alebo priamo konzultovaných s autormi daných systémov.

Analýzou problému sme stanovili detailné požiadavky na tento systém s tým, že pre spracovanie dát využijeme techniku MapReduce. Ako možných kandidátov pre realizáciu zadania sme zvolili distribuované databázové systémy Cassandra a HBase, ktoré sme nainštalovali na reálny klaster. V tomto testovacom prostredí sme nad danými systémami vykonali výkonostné testy. Táto časť práce bola do značnej miery náročná a vyžadovala množstvo nastavení daných systémov pre ich stabilný beh. Z výsledných systémov sme na základe viacerých vlastností popísaných v kapitolách 4 a 7 vybrali systém Cassandra.

Architektúra systému využíva viacero open source aplikácií. Pri analýze sme dbali na ich budúce využitie v reálnom nasadení. Zásadným bodom návrhu bolo využitie deduplikácie emailových správ, z dôvodu čo najefektívnejšieho využitia diskového priestoru. Systém bol implementovaný a vďaka metóde deduplikácie sme na vzorku z reálneho prostredia o veľkosti približne milión emailových správ dosiahli 65% úsporu. V budúcnosti by bolo vhodné skúsiť dané riešenie pomocou systému Brisk, ktorý bol oficiálne zverejnený pri dopísaní tejto diplomovej práce. Tento systém integruje framework MapReduce priamo so systémom Cassandra, čím by sa dosiahla maximálna decentralizácia celého riešenia.

Literatúra

- [1] Internet message format. 2001.
- [2] Simple mail transfer protocol. 2001.
- [3] D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.
- [4] D. Abadi, P. Boncz, and S. Harizopoulos. Column-oriented database systems, VLDB Tutorial. 2009.
www.cs.yale.edu/homes/dna/talks/Column_Store_Tutorial_VLDB09.pdf.
- [5] T. Barina. Email database for large volumes, máj 2011.
- [6] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.
- [7] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 5. Addison-wesley New York, 1987.
- [8] A. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.
- [9] E. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.
- [10] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
- [11] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.
- [12] DataStax. Evolving hadoop into a low-latency data infrastructure, 2011.
- [13] David Salmen, Tatiana Malyuta, Rhonda Fettes, Normert antunes. Cloud Data Structure Diagramming Techniques and Design Patterns, 2010.
- [14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

- [15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
- [16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.
- [17] E. Eifrem. Neo4j the benefits of graph databasesi [online]. Dostupné z WWW: <<http://wiki.neo4j.org/content/Presentations>>, [cit. 2011-01-05].
- [18] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 78–91. ACM, 1997.
- [19] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. 1996.
- [20] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part two: Media types. 1996.
- [21] A. Ganesh, A. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE transactions on computers*, pages 139–149, 2003.
- [22] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe. *IDC White Paper*, 2, 2008.
- [23] J. F. Gantz, J. Mcarthur, and S. Minton. The expanding digital universe. *Director*, 285(6), 2007.
- [24] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.
- [25] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP ’03, pages 29–43, New York, NY, USA, 2003. ACM.
- [26] D. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.
- [27] S. Gilbert and N. Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.
- [28] J. Gray et al. The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference*, pages 144–154. Citeseer, 1981.
- [29] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.
- [30] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *Symposium on Reliable Distributed Systems (SRDS’2004)*, pages 66–78. Citeseer, 2004.

- [31] E. Hewitt. *Cassandra: the definitive guide*. O'Reilly Media, Inc., 2010.
- [32] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.
- [33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.
- [34] A. O. R. W. Paper. Why Cloud-Based Security and Archiving Make Sense [online], March 2010.
Dostupné z WWW: <www.google.com/postini/pdf/why_cloud_based_wp.pdf>, [cit. 2011-11-15].
- [35] D. Pritchett. Base: An acid alternative. *Queue*, 6:48–55, May 2008.
- [36] T. Segaran and J. Hammerbacher. *Beautiful data: the stories behind elegant data solutions*. O'Reilly Media, 2009.
- [37] R. Shoup. The eBay Architecture, Striking a balance between site stability, feature velocity, performance, and cost [online], November 2006.
Dostupné z WWW: <addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf>, [cit. 2011-02-28].
- [38] M. Stonebraker. The case for shared nothing. *Database Engineering Bulletin*, 9(1):4–9, 1986.
- [39] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.
- [40] J. Udell. *Practical Internet GroupWare*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.
- [41] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.
- [42] Brisk [online].
Dostupné z WWW: <<http://www.datastax.com/products/brisk>>, [cit. 2010-09-05].
- [43] Cassandra [online].
Dostupné z WWW: <<http://cassandra.apache.org/>>, [cit. 2010-12-21].
- [44] Celery introduction [online].
Dostupné z WWW: <<http://ask.github.com/celery/getting-started/>>, [cit. 2011-04-07].

- [45] Discretionary access control [online].
Dostupné z WWW: <<https://issues.apache.org/jira/browse/HBASE-1697>>, [cit. 2011-04-03].
- [46] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

Dodatok A

Zoznam použitých skratiek

API - Application Programming Interface

EB - Exabyte

GC - Garbage Collector

GFS - Google File System

HDFS - Hadoop Distributed File System

JVM - Java Virtual Machine

KFS - Kosmos Distributed File System

NoSQL - Not Only SQL

PB - Petabyte

RFC - Request for Comments

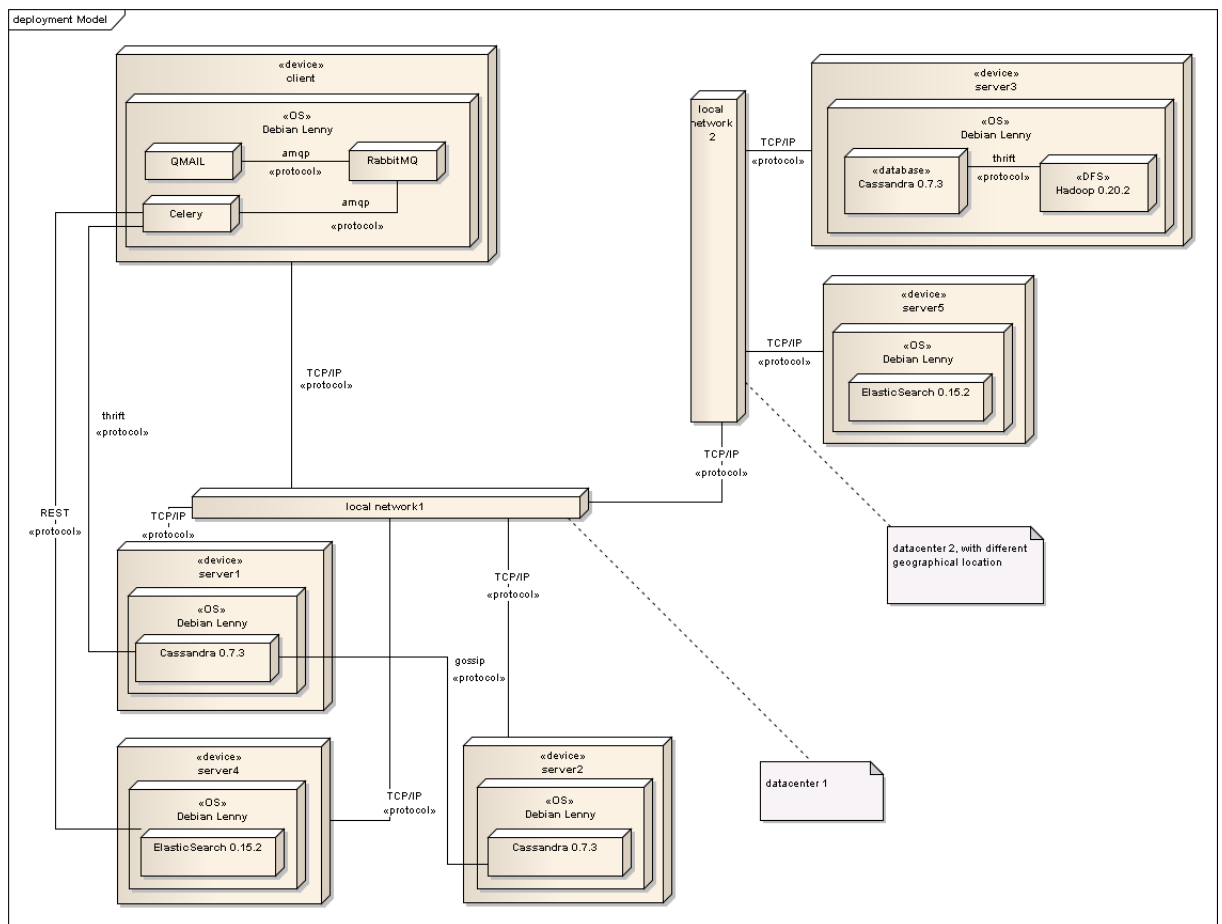
SPOF - Single Point Of Failure

TB - Terabyte

TTL - Time To Live

Dodatok B

UML diagramy

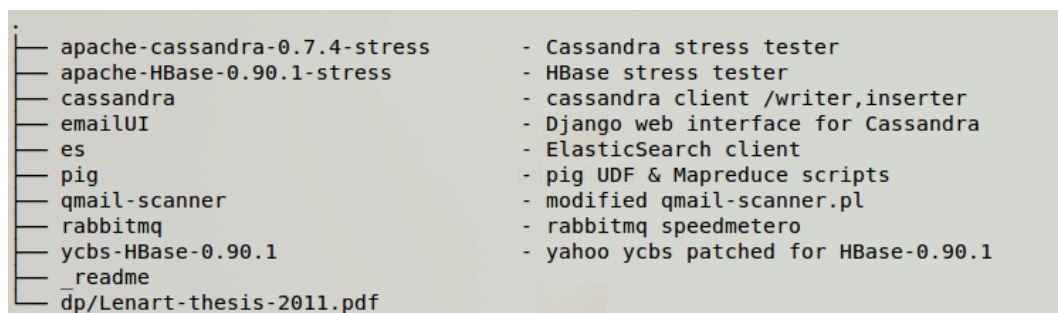


Obr. B.1: Diagram nasadenia aplikácie

Dodatok C

Obsah priloženého DVD

Následující obrázok [C.1](#) zobrazuje štruktúru priloženého DVD.



— apache-cassandra-0.7.4-stress	- Cassandra stress tester
— apache-HBase-0.90.1-stress	- HBase stress tester
— cassandra	- cassandra client /writer, inserter
— emailUI	- Django web interface for Cassandra
— es	- Elasticsearch client
— pig	- pig UDF & Mapreduce scripts
— qmail-scanner	- modified qmail-scanner.pl
— rabbitmq	- rabbitmq speedmetero
— ycbs-HBase-0.90.1	- yahoo ycbs patched for HBase-0.90.1
— _readme	
— dp/Lenart-thesis-2011.pdf	

Obr. C.1: Výpis priloženého DVD