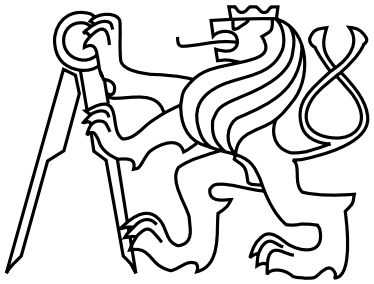
České vysoké učení technické v Praze
Fakulta elektrotechnická
Otvorená informatika



Diplomová práca

Veľkoobjemové úložisko emailov

Bc. Patrik Lenárt

Vedúci práce: Ing. Ján Šedivý, CSc.

Študijný program: Otvorená informatika, Magisterský

Odbor: Softwarové inžinierstvo

9. mája 2011

Poďakovanie

Rád by som poďakoval vedúcemu práce pánovi Ing. Jánovi Šedivému, CSc. za konzultácie, cenné rady, pripomienky a návrhy, ktoré mi ochotne poskytol počas vypracovávania tejto práce. Tak isto sa chcem poďakovať svojim najbližším, bez ktorých podpory by táto práca nevznikla.

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 niktorých zákonov (autorský zákon).

V Prahe dňa 1. 3. 2011 .....

Abstract

Abstrakt

Obsah

Zoznam obrázkov

Zoznam tabuliek

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 prácu s veľkým objemom dát. V dnešnej dobe existuje množstvo databázových systémov, ktoré sa navzájom líšia svojou architektúrou, dátovým modelom, výrobcom atď.

Od začiatku sedemdesiatych rokov 20. storočia sú v tejto oblasti dominantou relačné databázové systémy (Relational Database Management Systems). Vďaka neustálemu prudkému rozvoju internetových technológií a rapídnemu rastu dát v digitálnom univerze [13] začínajú byť tieto systémy nepostačujúce. Medzi hlavné faktory pre výber relačného databázového systému doposiaľ patrili výrobca, cena a pod. V dnešnej dobe so vznikom moderných aplikácií (napríklad sociálne siete, dátové sklady, analytické aplikácie a iné), požadujeme od týchto systémov vlastnosti ako vysoká dostupnosť, horizotnálna rozšíriteľnosť a schopnosť pracovať s obrovským objemom dát (petabyte). Novo vznikajúce databázové systémy, spĺňajúce tieto požiadavky sa spoločne označujú pod názvom NoSQL (Not Only SQL). Pri ich výbere je v tomto prípade dôležité porozumenie architektúry, dátového modelu a dát, s ktorými budú tieto systémy pracovať.

Táto práca si kladie za cieľ viacero úloh, ktorými sú pochopenie a popis základných konceptov, ktoré tieto systémy využívajú, určenie kritérií vďaka ktorým môžeme tieto systémy navzájom porovnávať. Ďalej je úlohou analýzovať a popísať požiadavky pre systém veľkoobjemového úložiska elektronickej pošty, ktorý bude schopný spracovávať milióny emailov. Poslednou úlohou je na základe našich požiadavkov vybrať, čo najlepšie odpovedajúci NoSQL systém a s jeho použitím implementovať prototyp aplikácie.

Osnova

...

Kapitola 2

Databázové systémy

V tejto časti stručne popíšeme históriu vzniku databázových systémov, základné problémy pri tvorbe distribuovaných relačných databázových systémov a uvedieme možné spôsoby ich riešenia. Ďalej popíšeme základné koncepty, ktoré sa využívajú pri tvorbe distribuovaných databázových systémov a techniku MapReduce, ktorá slúži na prácu s veľkým objemom dát uloženým v systémoch NoSQL.

2.1 História

V polovici šesťdesiatych rokov 20. storočia bol spoločnosťou IBM vytvorený informačný systém IMS (Information Management System), využívajúci hierarchichký databázový model. IMS je po rokoch vývoja využívaný dodnes. Po krátkej dobe, v roku 1970, publikoval zamestnanec IBM, Dr. Edger F. Codd článok pod názvom „A Relational Model of Data for Large Shared Data Banks“ [8], ktorým uviedol relačný databázový model. Prvým databázovým systémom, ktorý tento model implementoval bol System R od IBM. Tento systém používal jazyk pod názvom SEQUEL, ktorý je predchodca dnešného SQL (Structured Query Language) slúžiacého na manipuláciu a definícu dát v relačných databázových systémoch. Tento koncept sa stal základom pre relačné databázové systémy, ktoré vďaka širokej škále vlastností ako napríklad podpora tranzakcií, dotazovací jazyk SQL, patria v dnešnej dobe medzi najpoužívanejšie riešenia na trhu.

V minulosti boli objem dát, s ktorým tieto systémy pracovali a výkon hardvéru mnohonásobne nižšie. Dnes napriek tomu, že výkon procesorov a veľkosť pamäťových zariadení rapídne stúpa, je najväčšou slabinou počítačových systémov rýchlosť prenosu dát medzi pevným diskom a hlavnou pamäťou. Ako príklad si vezmime bežnú konfiguráciu počítačového systému, ktorá obsahuje pevný disk o veľkosti 2TB a operačnú pamäť veľkosti 64Gb. Napriek týmto vysokým kapacitám tento systém bohužial dokáže v daný moment spracúvať maximálne 64Gb dát, čo je zlomok veľkosti v porovnaní s kapacitou pevného disku. Vznik nových webových aplikácií napr. sociálne siete, zavádzanie cloud computingu vyžadujú od systémov podporu škálovania, ktorá zabezpečuje vysokú dostupnosť, spoľahlivosť a ich nároky na spracovávaný objem dát sa neustále zvyšujú. Tieto nové požiadavky efektívne riešia distribuované systémy pod spoločným názvom NoSQL, ktoré popisuje následujúca kapitola.

2.2 ACID

Relačné databázové systémy poskytujú veľkú množinu operácií, ktoré sa vykonávajú nad ich dátami. Tranzakcie [7][8] sú zodpovedné za korektné vykonanie operácií v prípade, že spĺňajú množinu vlastností ACID. Význam jednotlivých vlastností akronýmu ACID je následovný:

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

Implementácia vlastností ACID, ktoré zaručujú konzistenciu, zvyčajne využíva u relačných databázových systémoch metódu zamykania. Tranzakcia uzamkne dáta pred ich spracovaním a spôsobí ich nedostupnosť až do jej úspešného ukončenia, poprípade zlyhania. Pre databázový systém, od ktorého požadujeme vysokú dostupnosť alebo prácu pod zvýšenou záťažou, tento model nie je vyhovujúci. Zámky spôsobujú stavy, kedy ostatné operácie musia čakať na ich uvoľnenie. Jeho náhradou je Multiversion concurrency control, ktorý využívajú aj niektoré NoSQL databázové systémy.

Tranzakcie splňujúce vlastnosti ACID využívajú v distribuovaných databázových systémoch [1] dvojfázový potvrdzovací protokol (two-phase commit protocol). Distribuovaný databázový systém využívajúci tento protokol, ktorého tranzakcie splňujú vlasnosť ACID zaručuje konzistentnosť a je schopný odolávať čiastočným poruchám na sieti alebo v prípade čiastočnej poruchy systému. Vlastnosti ACID nekladú žiadnu záruku na dostupnosť systému. Takéto systémy su vhodné pre Internetové tranzakcie, aplikácie využívajúce platby apod. Existuje množstvo aplikácií, u ktorých má dostupnosť prednosť pred konzistenciou. Pri tvorbe distribuovaných databázových systémov je preto potrebné upustiť z niektorých ACID vlastností, čo spôsobilo vznik nového modelu pod názvom BASE.

2.3 Škálovanie databázového systému

Obecná definícia pojmu škálovateľnosť [4] je náročná bez vymedzenia kontextu, ku ktorému sa vzťahuje. V tejto práci budeme škálovateľnosť chápať v kontexte webových aplikácií,

[1] Distribuované databázové systémy sú tvorené pomocou viacerých samostatne operujúcich databázových systémov, ktoré nazývame uzly a môžu komunikovať pomocou sieti a užívateľovi alebo aplikácii sa javia ako jeden celok [ref].

ktorých dynamických vývoj kladie na databázové systémy viacero požiadavkov. Medzi hlavné patrí neustála potreba zvyšovania diskového priestoru a teda zvyšovanie veľkosti databáze alebo schopnosť obslúžiť čoraz vyšší počet užívateľov aplikácie (zvýšenie počtu operácií pre čítanie a zápis do databázového systému). V tomto prípade pod pojmom škálovatelnosť databázového systému rozumieme vlasnosť, vďaka ktorej je systém schopný spracúvať narastajúce požiadavky webovej aplikácie v definovanom čase intervale. Typicky pridaním nových systémov, čo spôsobuje vznik distribuovaného databázového systému.

Škálovateľnosť delíme na vertikálnu a horizontálnu. Táto metóda dodáva systému nasledujúce vlastnosti [5]:

- umožňuje zväčšiť veľkosť celkovej kapacity databáze 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 niektorých prípadoch môže zaručiť, že systém neobsahuje jednotku, ktorá by v prípade zlyhania spôsobila nedostupnosť celého systému (single point of failure).

Vertikálna škálovateľnosť je metóda, ktorá sa aplikuje pomocou zvýšovania výkonnosti hardvéru, tj. do systému sa pridáva operačná pamäť, rychlejšie viacjádrové procesory, zvyšuje sa kapacita diskov. Jednou z nevýhod tohoto riešenia je jeho vysoká cena a možná chvíľková nedostupnosť systému. Proces verktikálneho škálovania nad relačnou databázou obsahuje následujúce kroky:

- zámena hardvéru za výkonnejší
- úprava súborového systému (napr. zrušenie žurnálu)
- optimalizácia databázových dotazov, indexovanie
- pridanie vrsty pre kešovanie (memcached, EHCache, atď.)
- denormalizácia dát v databáze, porušenie normalizácie

V tomto prípade je možné naraziť na hranice Moorovho zákona [6] a na rad nastupuje horizontálna škálovateľnosť, ktorá je omnoho komplexnejšia. Horizontálnu škálovateľnosť je možné realizovať pomocou replikácie alebo metódou rozsekávania dát (sharding).

2.3.1 Replikácia

V distribuovaných systémoch sa pod pojmom replikácia rozumie vlastnosť, ktorá má za následok že sa daná informácia nachádza v konzistentnom stave na viacerých uzloch [2] tohoto systému. Táto vlastnosť zvyšuje dostupnosť, spoľahlivosť a odolnosť systému voči chybám.

V prípade distribuované databázového systému sa časť informácií uložených v databáze nachádza na viacerých uzloch. Táto vlasnosť môže napríklad zvýšiť výkonnosť operácií, ktoré

[2] Pod pojmom uzol v tomto prípade myslíme samostatný počitačový systém, ktorý je súčasťou distribuovaného systému

pristupujú k dátam a to tak, že dochádza k čítavaniu dát z databázy paralelne z viacerých uzlov. V systéme obsahujúcom repliku dát nedochádza k strate informácií v prípade poruchy uzlu. Replikácia a propagácia zmien v systéme sú z pohľadu aplikácie transparentné. Metóda replikácie nezvyšuje pridávaním nových uzlov celkovú kapacita databáze. Problémom tejto techniky je zápis dát, pri ktorom sa zmena musí prejaviť vo všetkých replikách. Existuje viacero metód pomocou, ktorých je možné zabezpečiť túto funkcionalitu:

- Read one - write all, u tejto metódy sa čítanie dát prevedie z ľubovolné uzlu obsahujúceho repliku. Zápis dát sa vykoná na všetky uzly obsahujúce repliku a v prípade, že každý z nich potvrdí úspech tejto operácie, zmena sa považuje úspešnú. Táto metóda nie je schopná pracovať v prípade, že dôjde k prerušeniu sieťového toku medzi uzlami (network partitioning) alebo v prípade poruchy uzlu.
- Quorum consensus - zápis na jeden uzol a následná asynchrónna propagácia repliky na ostatné uzly. Táto metóda je schopná zvládať stav pri ktorom dojde k prerušeniu sieťového toku alebo poruche uzlu. Implementácie využíva algoritmy pod názvom kôrum konsenzus (quorum consensus). ???

Výber metódy replikácie čiastočne popisuje dvojicu vlastností distribuovaného databázového systému a to dostupnosť a konzistenciu. Poďla teórie s názvom CAP (viď. nasledujúca kapitola) nie je možné aby systém disponoval súčasne vlastnosťami ako dostupnosť, konzistencia dát a schopnosť odolávať poruchám v prípade chyby v sieťovej komunikácii.

V relačných databázových systémoch sa replikácia rieši pomocou techniky 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. Touto metódou sme schopný mnohonásobne zvýšiť počet operácií, ktoré slúžia pre čítanie dát z databazového systému a v prípade zlyhania niektorého zo systémov máme neustále k dispozícii kópiu dát. Slabinou tejto techniky je uzol v roli master, ktorý znižuje výkonnosť v prípade operácií vykonavajúcich zápis a zároveň môže jeho porucha spôsobiť celkovú nedostupnosť systému.

Druhým riešením je technika Multi-master, kde každý uzol obsahujúci repliku je schopný zápisu dát a následne tieto preposiela zmeny ostatným. Tento mechanizmus predpokláda distribuovanú správu zamykania a vyžaduje algoritmy pre riešenie konfliktov spôsobujúcich nekonzistenciu dát.

2.3.2 Rozsekávanie dát (sharding)

Rozsekávanie dát 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 sme schopný zvyšovať celkovú kapacitu databáze. V prípade, že architektúra distribuovaného databázového systému využíva túto techniku, zvýšenie výkonu jeho operácií a objemu uložených dát sa realizuje automaticky bez nutnosti zásahu do aplikácie.

Techniku rozsekávania môžeme považovať za architektúru známu pod názvom zdieľanie ničoho [16] (shared nothing). Táto architektúra sa používa pre návrh systémov využivajúcich multiprocesory. V takomto prípade sa medzi procesormi nezdieľa operačná ani disková

pamäť. Táto architektúra zabezpečuje takmer neobmedzenú škálovateľnosť systému a využíva ju mnoho NoSQL systémov ako napríklad Google Bigtable, Amazon Dynamo alebo technológia MapRreduce.

Pri návrhu distribuovaných databázových systémov, s využitím tejto techniky, patrí medzi kľúčový problém implementácia funkcie spojenia (join) nad dátami, ktorá sa radšej 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 a záťaži systémových zdrojov.

Keďže sa dáta nachádzajú na viacerých uzloch systému, hrozí zvýšená pravdepodobnosť hardverového 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. Týmto spôsobom dochádza k celkovému narušeniu tohoto 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 rozsekávať a zabezpečíme tak rovnomerné zaťaženie uzlov daného systému. Existuje viacero metód [9] a to:

- segmentácia dát poďla funkcionality - dáta, ktoré sme schopný popísať spoločnou vlastnosťou ukladáme do samostatných databáz a tieto umiestňujeme na rozdielné uzly systému. Príkladom môže byť samostatný uzol spravujúci databázu pre užívateľov a iný uzol s databázov pre produkty. Túto metódu spracoval Randy Shoup [3] [19], architekt spoločnosti eBay.

- rozsekávanie podľa kľúča - v datách hľadáme kľúč, podľa ktorého sme schopný ich rovnomernej distribúcie. Následne na tento kľúč aplikujeme hašovaciu funkciu a na základe je výsledku tieto dáta umiestňujeme na jednotlivé uzly.

- vyhľadávacia tabuľka - jeden uzlol 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.

Replikácia a rozsekávanie dát patria medzi kľúčové vlastnosti využívané v NoSQL systémoch.

2.4 BASE

Akroným BASE [3] bol prykrát použitý v roku 1997 na sympóziu SOSP (ACM Symposium on Operating Systems Principles). BASE tvoria následujúce slovné spojenia:

[3] "If you can't split, you cant scale it." – Randy Shoup, Distinguished architect Ebay

- bežne dostupný (Basically Available) - systém je schopný zvládať jeho čiastočné zlyhanie za cenu nižšej komplexity.
- zmiernený stav (Soft State) - systém nezaručuje trvácnosť dát za cieľom zvýšenia výkonu.
- čiastočná konzistencia (Eventually Consistent) - je možné na určitú dobu tolerovať nekorektnosť dát, ktoré musia byť po určitom časovom intervale konzistentné.

Tento model poľavil na požiadavku zodpovednom za konzistenciu dát, s tým že dosiahol vyššiu dostupnosť v distribuovanom databázovom systéme aj v prípade čiastočného zlyhania. Prakticky môžeme každý systém klasifikovať ako systém spĺňajúci vlasnosti ACID alebo BASE.

BASE umožnuje horizontálne škálovanie relačných databázových systémov bez nutnosti použitia distribuovaných tranzakcií. Použitie tejto metódy je možné vďaka rozsekávaniu dát s využitím metódy segmentácie dát podľa funkcionality, ďalej sa využívajú perzistentné fronty a princípy událosťami riadenej architektúry (event driven architekture) [18]. Poľavením na požiadavku konzistencie dát sa v tomto prípade myslí stav, že dáta budú konzistentné po uplynutí určitého časového intervalu.

Systém obsahujúci čiastočnú konzistenciu dát je bankomatový systém. Po vybraní určitej čiastky z účtu, sa korektná informácia o aktuálnom zastatku účtu môže zobraziť až za niekoľko dní, kdežto tranzakcia ktorá túto zmenu vykonala musí spĺňať vlasnosti ACID. Medzi podobné webové aplikácie, u ktorých sa nepožadujú všetky vlasnosti ACID patria nákupný košík spoločnosti Amazon, zobrazovanie časovej osi aplikácií Twiter poprípade indexy Googlu. Ich nedostupnosť by znamenala obrovské finančné straty, napríkad v prípade, zlyhanie vyhľadávania pomocou Googlu by znamenalo zobrazenie nižšieho počtu reklám, nedstupnosť nákupného košíka v Amazone by spôsobila pokles predaja atď.

Aplikácia vyššie popísaných techník na relačné databázové systémy môže byť netrivialnou úlohou. Relačný model, je spôsob reprezentácie dát, ktorý umožnuje efektívne riešit určité typy problémov, preto snaha prispôsobiť tento model každému problému je nezmyselná. V tomto prípade, musíme uvažovať alternatívne riešenia, medzi ktoré patria systémy NoSQL.

2.5 CAP

Moderné webové aplikácie kladú na systémy požiadavky, medzi ktoré patrí vysoká dostupnosť, konzistencia dát a schopnosť odolávať chybám [4]. Dr. Brewerer v roku 2000 nastolil myšlienku dnes známu pod názvom teória CAP [5], ktorá tvrdí že je možné súčasne dosiahnúť len dvojicu z týchto vlastností. V roku 2002 platnosť tejto teórie pre asynchrónnu sieť matematicky dokázali Lynch a Gilbert [**?**]. Modelu asynchrónnej sieťe odpovedá svojimi vlasnosťami sieť Internet. Akroným CAP tvoria následujúce vlasnosti:

- Konzistencia (Consistency) - distribuovaný systém je v konzistentnom stave ak sa zmena aplikuje na všetky relevantné uzly systému v rovnakom čase.

[4] Chybou v tomto kontexte myslíme, prerušenie sieťovej komunikácie medzi uzlami daného systému, poprípade hardverová porucha uzlu???

- 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, tato vlastnosť hovorí, že prípade ak nastane sieťový problém [5] každý požiadavok bude vykonaný.

- Tolerancia chýb (Partition Tolerance) - uzly distribuovaného systému navzájom komunikujú pomocou siete, v ktorej hrozí stráta správ. V prípade vzniku sieťovej poruchy dané uzly medzi sebou navzájom nedokážu komunikovať. Táto vlasnosť podľa definície viď. Gilbert a Lynch tvrdí, že v prípade vzniku zlyhania sieťovej komunikácie medzi niektorými uzlami musí byť systém schopný naďalej pracovať korektne. Neexistuje reálny distribuovaný systém, ktorého uzly na vzájomnú komunikáciu využívajú sieť a nedochádza pri tom k stráte správ, teda k poruchám sieťovej komunikácie.

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

$$P(ubovonhozlyhania) = 1 - P(individulnyuzolnezlyh)^{poetuzlov}$$

2.5.1 Konzistencia verzus dostupnosť

V distribuovanom systéme nie je možné zaručiť súčasne vlasnosť konzistencie a dostupnosti. Ako príklad si predstavme distribuovaný systém, ktorý zaručuje obe vlasnosti aj v prípade sieťového prerušenia. Tento systém obsahuje tri uzly A,B,C, na ktorých 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 obdrži požiadavok pre zmenu dát má na výber dve možnosti:

1. vykonať zmenu dát, v tomto prípade sa uzly A a B o tejto zmene dozvedia až v prípade, že bude sieťové prerušenie odstranené

2. zamietnuť požiadavok na zmenu dát, z dôvodu že uzly A a B sa o tejto zmene nedozvedia až do jej odstránenia

V prípade výberu možnosti čislo 1 zabezpečíme dostunosť systému naopak v prípade možnosti číslo 2 jeho konzistenciu, avšak nie je možný súčasný výber oboch riešení.

Ak od daného systému tolerujúceho sieťove prerušenia požadujem konzistenciu na úkor dostupnosti jedná sa o alternatívu CP. Takýto systém zabezpečí atomickosť operácií ako zápis a čítanie a zároveň sa môže stať, že na určité požiadavky nebude schopný odpovedať. Medzi takéto systémy môžeme zaradiť distribuovaný databázový systém využívajúci dvojfázový potvrdzovací protokol (2PC).

V prípade, že poľavíme na požiadavku konzistencie tak takýto systém bude vždy dostupný aj napriek sieťovým prerušeniam. V tomto prípade sa jedná o model AP. Je možné, že v takomto systéme bude dochádzať ku konfliktným zápisom alebo operácie čítania budú po určitú dobu vraciať nekorektné výsledky. Tieto problémy s konzistenciou sa v distribuovaných databázových systémoch riešia napríklad pomocou metódy vektorový časovač (Vector

[5] týmto sa nemyslí porucha uzlu

clock) alebo na aplikačnej úrovni. Príkladom systému patriaceho do tejto kategórie je služba DNS.

V prípade, že systém nebude schopný zvládať sieťové prerušenia tak takýto systém bude spĺňovať požiadavok konzistencie a dostupnosti, varianta CA. Jedná sa o nedistribuované systémy pracujúce na jednom fyzickom hardvéri využivajúce tranzakcie.

Vyššie popísané vlasnosti nám umožnia vhodný výber distribuovaného databázového systému podľa požiadavkov našej aplikácie.

2.6 Eventuálna konzistencia

V ideálnom svete je predstava konzistencie v distribuovaných systémoch následovná: v prípade, že sa v systéme vykoná zmena (zápis dát), na všetkych uzloch sa táto zmena prejaví súčasne s rovnakým výsledkom. Konzistencia v distribuovanom databázovom systéme je úzko spojená s replikáciou. Keďže podľa CAP teórie distribuovaný systém nemôže súčastne spĺňať požiadavok dostupnosti, konzistencie v prostredí s možným sieťovým prerušením, je na našom zvážení ktorú z týchto vlasností uprednostníme pri návrhu a tvorbe aplikácií. Väčšina NoSQL systémov poskytuje eventuálnu konzistenciu. V následujúcej časti preto popíšeme rôzne typy konzistencie.

V predchádzajúcom texte sme už spomínali, že v dnešne dobe existuje mnoho aplikácií, u ktorých je možné poľaviť na požiadavku konzistencie a funkčnosť systému nebude v tomto prípade ohrozená, ak sa určitá zmena prejaví s miernym oneskorením. Takáto konzistencia je odlišná od definície vlastností ACID, kde ukončenie tranzakcie spôsobí, že systém sa nachádza v konzistentom stave. Na konzistenciu sa môžeme pozerať z dôch 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 serverový, ktorý zabezpečuje technické riešenie a implementáciu techník zodpovedných za konzistenciu v distribuovaných databázových systémoch.

2.6.1 Konzistencia z pohľadu klienta

Pre potrebu následujúcich definíc uvažujme distribuovaný databázový systém, ktorý tvorí úložisko dát a tri nezávislé procesy A,B,C, ktoré možu v danom systéme zmeniť hodnotu dátovej jednotky, tj. vykonať zápis. Tieto procesy môžu zároveň zo systému hodnotu dátovej jednotku prečítať. Na základe toho ako dané procesy pozorujú nezávisle zmeny systému delíme konzistenciu [**?**] na:

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 tranzakcie s vlasnosťami 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 jej novú hodnotu. Definujeme pojem „nekonzistentné okno“ zabezpečujúci, že po uplynutí určitej 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 žiadná nová zmena hodnoty dátovej jednotky, po určitom čase budu všetky procesy pristupujúce k tejto jednotke schopné vrátiť jej korektnú hodnotu. Tento model ma viacero variacií, niektoré z nich popíšeme v nasledujúcej časti textu.

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

Session consistency - v tomto prípade pristupuje 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 vraciať hodnotu dátovej jednotky, zápisanú pred vznikom predchádzajúcej relácie.

Monotonic read consistency - v prípade, že proces vrátil určitú hodnotu dátovej jednotky, tak v každom ďalšom prístupe, nemôže nastať situácia, kde by vrátil 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 požiadavkoch konzistencie. Príkladom môže byť asynchrónna replikácia v modernom relačnom databázovom systéme, ktorá spôsobí, že systém bude eventuálne konzistentný.

2.6.2 Konzistencia na strane servera

Kôrum je minimálny počet hlasov, ktorý musí obdržať distribuovaná tranzakcia aby mohla následne vykonať operáciu v distribuovanom systéme. Technika založená na protokoloch kvôra (quorum-based protocols) je pužívaná na vykonávanie konzistentných operácií v distribuovaných databázových systémoch.

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 replikov, ktoré musia vrátiť hodnotu dátového objektu v prípade operácie čitanie

Rôzna konfigurácia týchto parametrov zabezpečí rozdielnu výkonnosť a dostupnosť distribuovaného systému. Uvažujme následujúce príklady, kde N = 3.

1. R = 1 a W = N, v takomto prípade zabezpečíme že systém bude optimalizovaný pre operácie čítania dát. Operácie budú konzistentné, pretože uzol z ktorého dáta čítame sa prekrýva s uzlami na ktorých vykonávame zápis. Nevýhodou tohoto modelu je, že v prípade nedostupnosti všetkých replík nebude možné do systému zapisovať. V prípade systémov, kde vyžadujeme rýchle čítanie a na systém je obrovský počet požiadavok čítania 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, že systém s replikou na ktorú sa vykoná zápis zlyhá.

3. W + R <= N, tento model spôsobí, že uzly, na ktoré sa vykonáva zápis a čítanie sa neprekrývajú, z čoho vyplýva u distribuovaného databázového systému vlasnosť eventuálnej konzistencie.

Nekonzistentnosť dát, môže byť tolerovaná v distribuovaných systémoch , ktoré sú vysoko škálovateľné za cieľom dosiahnutia lepšieho výkonu operácií, ktoré slúžia pre zápis a čítanie dát, celkovej výkonnosti a dostupnosti systému. Hranica do akej miery je možné dovoliť nekonzistenciu je určená požiadavkom klientskej aplikácie a vyššie spomínané modely sa ju snažia riešiť. Väčšinu z týchto modelov implementujú NoSQL systémy.

2.7 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 rýchlosti pre čítanie a zápis dát na tieto zariadenia.

Pre jednoduchosť uvažujme nasledujúci príklad, v ktorom chceme spracovať pomocou jedného počítačového systému 1TB dát uložených na lokálnom súborovom systéme, pri priemernej prenosovej rýchlosti diskových zariadení 100Mb/s. Za ideálnych podmienok by čas na prečítanie týchto dát presahoval dve a pol hodiny. Tento čas je z praktických dôvodov neprípustný. V prípade, že by sme tento 1TB dát vhodne rozdelili na sto počítačov a na každom z nich tento úsek spracovali, celková doba spracovania by sa znížila za ideálnych podmienok na necelé tri minúty. Spoločnosť Google v roku 2004 zverejnila programovací model pod názvom MapReduce [**?**], ktorý rieši tento problém pomocou paralelizácie výpočtu.

MapReduce je programovací model, ktorý slúži na paralelne spracovanie dát (PB). Model využíva vlastnosti paralelných a distribuovaných systémov, je optimalizovaný pre beh na klastri, tvorenom vysokým počtom (tisícky) spotrebných počítačov. Jeho cieľom je pre programátora zastrieť všetky problémy, ktoré spôsobuje paralelizácia, poruchovosť systémov, distribúcia dát vzhľadom na ich lokalitu a rovnomerne rozvhovanie záťaže medzi systémami. Poskytuje rozhranie pre automatickú paralelizáciu a rozsiahly distribuovaný výpočet.

Pre použitie tohoto nástroja musí programátor zadefinovať dve funkcie pod názvom map a reduce. Funkcia map na jednotlivých uzloch systému, transformuje vstupné data na základe zadefinovaného kľúča (k1) na medzivýsledok, ktorý obsahuje nové kľúče (g1,...,gn) a k ním odpovedajúce hodnoty. Tieto hodnoty sa zoradia podľa ich prislušnoti ku kľúčom (g1,...,gn), následne sa odošlú na uzly s funkciou reduce. Užívateľom definovaná funkcia reduce nad hodnotami priradenými pre kľúč (g1,...,gn) prevedie operáciu, ktorej typickým výsledkom je jedna výsledná hodnota.

map(kľuč k1, hodnota) -> list(kľuč(z g1,...,gn), hodnota2)

reduce(kľúč(z g1,...,gn), list(hodnota2)) -> list(hodnota3)

2.7.1 Príklad

? histogram slov

2.7.2 Architektúra

Obrázok

2.7.3 Použitie

MapReduce nie je vhodný na spracovanie dát v reálnom čase, online spracovanie dát, ktoré sú citlivé na latenciu a to z dôvodu jeho optimalizácie pre dávkový beh.

Výhodou tohoto modelu je, že dokáže pracovať s neštrukturovanými dátami. Jeho implementácia spoločnosťou Google, ktorá zároveň využíva distribuovaný súborový systém GFS nie je verejna. V rámci hnutia NoSQL vzniklo open-source riešenie pod názvom Hadoop, ktoré implementuje tento model na vlastnom distribuovanom súborovom systéme HDFS. Zároveň vznikli frameworky ako HIVE alebo PIG, ktoré sú nadstavbou modelu MapReduce, majú syntax podobnú jazyku SQL a využívajú ich NoSql databázové systémy pre spracovanie dát.

2.8 Zhrnutie kapitoly

Cieľom tejto časti bolo pochopiť základné princípy, ktoré platia v distribuovaných databázových systémoch a sú súčasťou systémov NoSQL. Taktiež sme identifikovali problémy, ktoré môžu nastať v prípade tvorby distribuovaných relačných databázových systémov a stručne popísali možnosti ich riešení.

Kapitola 3

Definícia problému

Množstvo digitálnych informácii, každým rokom prudko narastá. Podľa štatistík spoločnosti IDC[13] v roku 2006 dosahovala kapacita digitálneho univerza veľkosť 161 exabytov[1](EB). Podiel elektronickej pošty bez spamu, tvoril 3% z tohoto objemu. Podľa odhadov na rok 2011[12] má kapacita digitálneho univerza dosiahnuť veľkosť 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, nazývaná spam. Predpokladá sa, že až 85% dát z celkového odhadovaného objemu 1800 EB budú spracovávať, prenášať alebo zabezpečovať organizácie. Napriek tejto explózii digitálnych informácii 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 analýzovať potreby pre archiváciu elektronickej pošty (tj. emaily) a definovať požiadavky pre systém slúžiaci k archivácii emailov.

3.1 Archivácia elektronickej pošty

S neustálym nárastom informácii obsiahnutých v digitálnom univerze, sa zároveň zvyšuje objem dát, ktorý reprezentuje elektronickú poštu. Je preto potrebné porozumieť štruktúre emailových správ a následne ich vhodne spracovať. Tieto dáta je potrebné uložiť tak aby sme dosiahli úsporu diskového priestoru, boli sme nad nimi schopný vykonávať operácie ako fultextové vyhľadávanie, zber údajov pre tvorbu štatistík alebo 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. spamu, spyware, malware a phishingu. Na boj proti týmto hrozbám využívajú orga-

[1] $1EB = 10^{18}B$

nizácie anti-spamové a anti-vírusové systémy. Možné dôvody prečo archivovať elektronickú poštu sú následovné[17]:

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

3.2 Požiadavky na systém

V následujúcej časti popíšeme požadované vlastnosti systému, ktorý bude slúžiť na archiváciu veľkého objemu emailových správ. Primárnou požiadavkou na systém je jeho neustála dostupnosť, rozšíriteľnosť a nízkonakladová administrácia. Predpokladané množstvo uložených dát v tomto systéme bude dosahovať desiatky až stovky terabajtov[2] (TB). Takúto kapacitu dát nie je možne uchovať na bežne dostupnom hardvéri. Dáta uložené v systéme musia byť replikované v prípade vzniku havárie niektorej z jeho časti. Nad uloženými dátami, je potrebné vykonávať výpočtovo náročné operácie ako generovanie štatistík a fultextové 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, ktorých porovnanie popíšeme v následujúcej kapitole.

3.2.1 Funkčné požiadavky

Ukladanie emailov

Základnou jednoutkou, ktorú budeme do systému ukladať je emailová správa. Graf XY znázorňuje uporiadanie emailov podľa ich veľkosti nad vzorkom približne 1,000,000 emailových správ z reálneho prostredia[3]. Z grafu je vidieť, že veľkosť takmer XY% emailov spadá do intervalu XY kB. Tieto údaje sú len aproximáciou a sú závislé na konkrétnych používateľoch.

Systém musí umožnovať uloženie emailu bez porušenia jeho integrity. Klúčovým požiadavkom je ukladanie príloh emailov, kde požadujeme aby každá príloha bola jednoznačne identifikovaná a v prípade jej duplicity nebola opakovane uložená v systéme. Dôvodom je vysoká úspora diskového priestoru. Ďalším požiadavkom je automatické zmazanie emailov patriacich do danej domény po uplynutí predom špecifikovanej doby.

[2] $1TB = 10^{12}B$

[3] Vzorok emailov pre analýzu bol sprístupnený spoločnosťou Excello.

Export emailov

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

Vyhľadávanie emailov

Vyžadujeme fultextové vyhľadávanie emailov podľa následujúcich údajov:

- príjemca emailovej správy
- odosielateľ emailovej správy
- predmet správy
- dátum obsiahnutý v emailovej správe
- identifikátor emailu (message ID)
- názvy príloh a ich veľkosti
- veľkosť emailu
- vyhľadávanie v tele emailu

Pre administrátorské účely požadujeme vyhľadávanie podľa údajov:

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

Vyhľadávanie je potrebné realizovať nad všetkými emailovými správami uloženými v systéme a jednotlivo nad správami podľa danej domény a nad správami, ktoré prináležia danému uživateľovi.

Štatistické údaje

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

- 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
- počet deduplikácií

3.2.2 Nefunkčné požiadavky

Dostupnosť

Systém musí byť neustále dostupný a schopný odolávať poruchám v sieťovej komunikácií (angl. network partitions), krátkodobej nedostupnosti uzlov, úplným zlyhaniam jednotlivých uzlov a umožnovať spracúvať tok pre zápis dát v rozmedzí 10 Mbit až 1 Gbit. Ďalším požiadavkom je aby sa dáta replikovali vo vnútri datacentra na dva uzly a tretia replika bola umiestnená v datacentre, ktoré sa bude nachádzať na geograficky odlišnom mieste. Vyžadujeme aby systém neobsahoval bod, ktorého porucha by spôsobila celkovú nedostupnosť systému (ang. single point of failure). Tento problém rieši vlasnosť decentralizácie, ktorá zabezpečuje, že každá jednotka systému vykonáva rovnakú funkciu a je kedykoľvek plne nahraditeľná.

Rozšíriteľnosť

Predpokladáme použitie bežne dostupného spotrebného hardvéru (angl. commodity hardware), namiesto superpočítačov. Z dôvodu neustalého nárastu objemu elektronickej pošty, musí systém podporovať horizontálne škálovanie, ktoré bude schopné umožnovať zvýšenie celkovej kapacity dátového úložiska (desiatky petabajtov[4]). Pridávanie nových uzlov do systému umožní zvyšiť celkový vypočetný výkon, ktorý sa využije na spracovanie dát pomocou techniky MapReduce. U distribuovaného databazového systému je nutná podpora replikácie, ktorá zvýši výkonnosť operácií pre čítanie a zápis do systému a vďaka nej nebude potrebné riešiť zalohovanie 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 balancovanie záťaže sa musí vykonávať automaticky. Pridanie poprípade odobranie nového uzla nesmie ovplyvniť beh celkového systému.

Bezpečnosť

Osoby s oprávnením pre prístup k systému budú schopné oprerovať s jeho celým obsahom. Nekladieme žiadne požiadavky na uživateľské role z dôvodu, že predpokladáme beh systému v bezpečnom prostredí.

[4] $1PB = 10^{15}B$

Implementačné požiadavky

Cieľom je implementácia systému s využitím dostupných open source technologií.

Z analýzy obecných vlastností relačných databázových syśtémov v prechádzajúcej kapitole vidíme, že použitie týchto systémov nie je vhodné pre riešenie definovaného problému. Medzi základné problémy patrí náročné horizontálne škalovanie a problémy s vysokou dostupnosťou. V nasledujúcej kapitole sa budeme zaoberať popisom NoSQL systémov a po ich analýze vyberieme vhodného kandidáta, ktorého použijeme k implementácii prototypu, z dôvodu vyskokej komplexnosti riešeného problému.

Kapitola 4

NoSQL

Názov NoSQL bol prvykrát použitý v roku 1998 ako názov relačnej databázy, ktorej implementácia bola prevažne v interpretovaných programovacích jazykoch a neobsahovala jazyk SQL. V druhej polovici roku 2009 [?] sa názov NoSQL začal používať v spojení s databázovými systémami, ktoré nepoužívajú SQL dotazovací jazyk a tradičný relačný model, sú schopné horizontálneho škálovania pracujú na bežných spotrebných počítačoch, vyznačujú sa vysokou dostupnosťou, odolávajú chybám (hw sw siet) a používajú jednoduchý alebo bezschémový dátový model.

Novo vznikajúce webové aplikácie ako napríklad sociálne siete spracúvajú čoraz väčší objem dát, musia byť schopné v daný moment obslúžiť čoraz väčší počet užívateľov a neustále dostupné. Pôvodným cieľom hnutia NoSql, bolo vytvoriť koncept, pre tvorbu moderných databáz, ktoré by boli schopné riešiť tieto nové požiadavky. Idea týchto systémov je založená na filozofii, ktorá tvrdí, že nemá zmysel sa za každú cenu snažiť prispôsobiť dáta modelu relačnej databáze. Cieľom je vybrať systém, ktorý bude čo najvhodnejšie opovedať požiadavkom na uloženie a spracovanie našich dát. NoSQL obecne nepopisuje, žiaden konkrétny databázový systém, namiesto toho je to obecný názov pre nerelačné (non-relational) databázové systémy, ktoré majú odlišné vlastnosti a umožnujú prácu s rôznými dátovými modelmi. Medzi ich ďalšie znaky patrí slabá konzistencia, možnosť spracúvať obrovské objemy dát (PB), jednoduché API a možnosť asynchrónného zápisu dát. Tieto systémy nepodporujú operáciu databázového spojenia z dôvodu, že znižuje výkonnosť a zvýšuje zaťaženie siete (v prípade, že by sa táto operácia mala vykonať nad dátami, ktoré sa nachádzajú na rôznych uzloch). Pre tieto databázove systémy ďalej platí, že sú distribuované, podporujú automatickú replikáciu a rozsekávanie dát. Kedže sa jedná o relatívne mladé systémy, jedným z ich nedostatkov je malá podpora frameworkov, neustále sa meniace API a taktiež u mnohých chýbajúce rôzne grafické utility pre ich správu a monitoring. Medzi dátami, ktoré do nich ukladáme, je možné vytvárať vzájomné závislosti až na aplikačnej vrstve. Cieľom tohoto konceptu je riešiť spomínané novo vznikajúce problémy a zároveň koexistovať s relačnými databázovými systémami.

4.1 Typy NoSQL databázových systémov

Medzi nerelačné databázové systémy patria objektové, dokumentové, grafové, stĺpcové a databázové systémy s dátovým modelom typu kľúč-hodnota. V následujúcej časti stručne

popíšeme štvoricu najpopulárnejších.

4.1.1 Kľúč-hodnota (Key-value)

Tento model využíva pre ukládanie dát jednoduchý princíp. Blok dát, ktorý môže mať ľubovoľnú štruktúru je v databáze uložený pod názvom kľúča, ktorý zvykne byť reprezentovaný ako textový reťazec. Databázové systémy, využívajúce tento model majú jednoduché API rozhranie:

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

Výhodou tohoto modelu je, že databázový systém je možné ľahko škálovať. Bohužial v tomto prípade sa o štruktúru uložených dát musí starať klient, čo umožnuje dosahovať vysokú výkonnosť na strane databázového systému. Tento model existuje v mnohých modifikáciach.

Relačný databázový model reprezentuje dáta pomocou tabuliek, pre ktoré definujeme ich štruktúru a ktoré sú normalizované aby sme predišli duplikácii dát. Pre zabezpečenie integrity jednotlivých entít a referenčnej integrity využívame primárne a cudzie kľúče. Tabuľky s popisom názvov ich stĺpcov a vzťahy medzi nimi nazývame databázovou schémou.

Najväčšou nevýhodou je, že databázový systém nie je schopný medzi uloženými dátami zachytiť ich vzájomné relácie, čo patrí medzi základné požiadavky pri modelovaní dát. Úložiško typu kľúč-hodnota nevyužíva normalizáciu dát, dáta sú často duplikované, vzťahy a integrita medzi nimi sa riešia až na aplikačnej úrovni. Pre vkladané dáta a k nim asociované kľúče sa nedefinujú žiadne obmedzenia.

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

4.1.2 Stĺpcovo orientovaný model (Column [Family] Oriented)

Množstvo databázových systémov využíva pre reprezentáciu dát tabuľky, ktoré sú tvorené stĺpcami a popísané schémou (tj. názvy stĺpcov, tabuliek). Každý riadok tabuľky reprezentuje dáta, ktoré sa nazývajú záznamy a tieto sú následne sekvenčne ukladané na pevný disk. Tento model, nazývaný riadkový, je vhodný pre systémy, u ktorých sú dominantou operácie vykonávajúce zápis. Relačné databázové systémy, využívajúce tento model sú teda optimalizované pre zápis. Pre efektívny prístup k dátam môže tento model využívať indexy.

V dnešnej dobe existuje veľký počet aplikácií, u ktorých prevládajú operácie čítania nad zápisom. Patria sem dátové sklady, customer relationship management (CRM) systémy, 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 [13][14], ktorý je zároveň optimalizovaný pre operácie čítanie dát. Data reprezentujúce stĺpce sú uložené na pevnom disku v samostatných a súvislých blokoch. Načítanie dát do pamäti a následná práca s nimi je efektivnejšia ako u riadkových databáz, kde je potrebné načítať celý záznam obsahujúci hodnoty stĺpcov, ktoré sú pre nás v daný moment irelevantné.

Riadkový model obsahuje v jednom zázname dáta z rôznych domén, čo spôsobuje vyššiu entropiu v porovnaní so stĺpcovým modelom, kde sa predpokláda, že dáta v danom stĺpci pochádzajú z totožnej domény a môžu si byť podobné. Táto vlasnosť umožnuje efektívnu komprimáciu dát, ktorá znižuje počet diskových operácií. Nevýdou tohoto modelu je zápis dát, ktorý spôsobuje vyššiu záťaž diskových operácií. Optimalizáciou môže byť dávkový zápis dát.

4.1.2.1 Stĺpcovo orientovaný model v NoSQL

Predchodcom tohto nového prístupu k stĺpcovému modelu v NoSQL systémoch je databázový systém od Google - Bigtable. V tomto prípade sa využíva kombinácia modelu kľúč-hodnota so stĺpcovo orientovaným modelom. Na takýto model sa môžeme pozerať ako na viacdimenzionálne úložisko typu kľúč-hodnota. Detailnejšie tento model popíšeme v následujúcej kapitole.

Tento model sa používa v databázových systémoch ako Google BigTable, Hbase, Hypertable alebo Cassandra.

4.1.3 Dokumentový model

Dokumentové databáze sú založené na predchádzajúcom modeli typu kľúč-hodnota. Požiadavkom na ukládané data je, že musia byť v tvare ktorému rozumie databázový systém. Štruktúra vkládaných dát môže byť popísaná napríklad pomocou XML, JSON, YAML. Táto štruktúra nám následne umožnuje okrem jednoduchého vyhľadávania pomocou kľúč-hodnota vytvárať s využitím indexov zložitejšie dotazy nad dátami na strane databázového systému.

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

4.1.4 Grafový model

Tento typ databáz využíva pre prácu s dátami matematickú štruktúru - graf. Dáta sú reprezentované pomocou uzlov, hran a ich atribútov. Uzol je základný samostatný a nezávislý objekt. Pomocou hran medzi uzlami modelujeme závislosti, ktoré popisujeme 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 sieti alebo pre sémantický web. Patria sem databáze ako Neo4j alebo FlockDB.

4.2 Porovnanie NoSQL systémov

V dnešnej dobe existuje veľké množstvo NoSQL databázových systémov, ktoré majú odlišné vlasnosti, komplexitu a vďaka tomu ich môžeme použiť pre rôzne účely. Snaha porovnať tieto systémy na globálnej úrovni je nerealizovateľná a často vedie k omylu. Cieľom tejto sekcie je definovať základné body vďaka, ktorým je možné tieto systémy kategorizovať a vrámci danej kategórizácie porovánať. Tieto zistenia nám následne môžu pomôcť pri výbere vhodného systému odpovedajúceho našim požiadavkom.

Následujúce body patria medzi hlavné kritéria pri porovnávaní týchto systémov:

1. dátový model
2. dotazovací model
3. škálovateľnosť
4. schopnosť odolávať chýbam (failure handling)
5. elastickosť
6. konzistencia dát
7. typ perzistentného úložiska
8.

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 sme schopný nad uloženými dátami vykonávať. V predchádzajúcej sekcii sme popísali základnú kategorizáciu NoSQL systémov poďla dátového modelu. Dátový a dotazovací model do určitej miery popisuje výkonnosť, komplexnosť a vyjadrovaciu silu databázového systému. Dotazovací model popisuje API.

Pre výber vhodného dátového modelu pre našu aplikáciu je dôležité porozumieť štruktúre dát a definovať operácie, ktoré nad týmito dátami budeme vykonávať.

4.2.2 Škálovateľnosť a schopnosť odolávať chybám

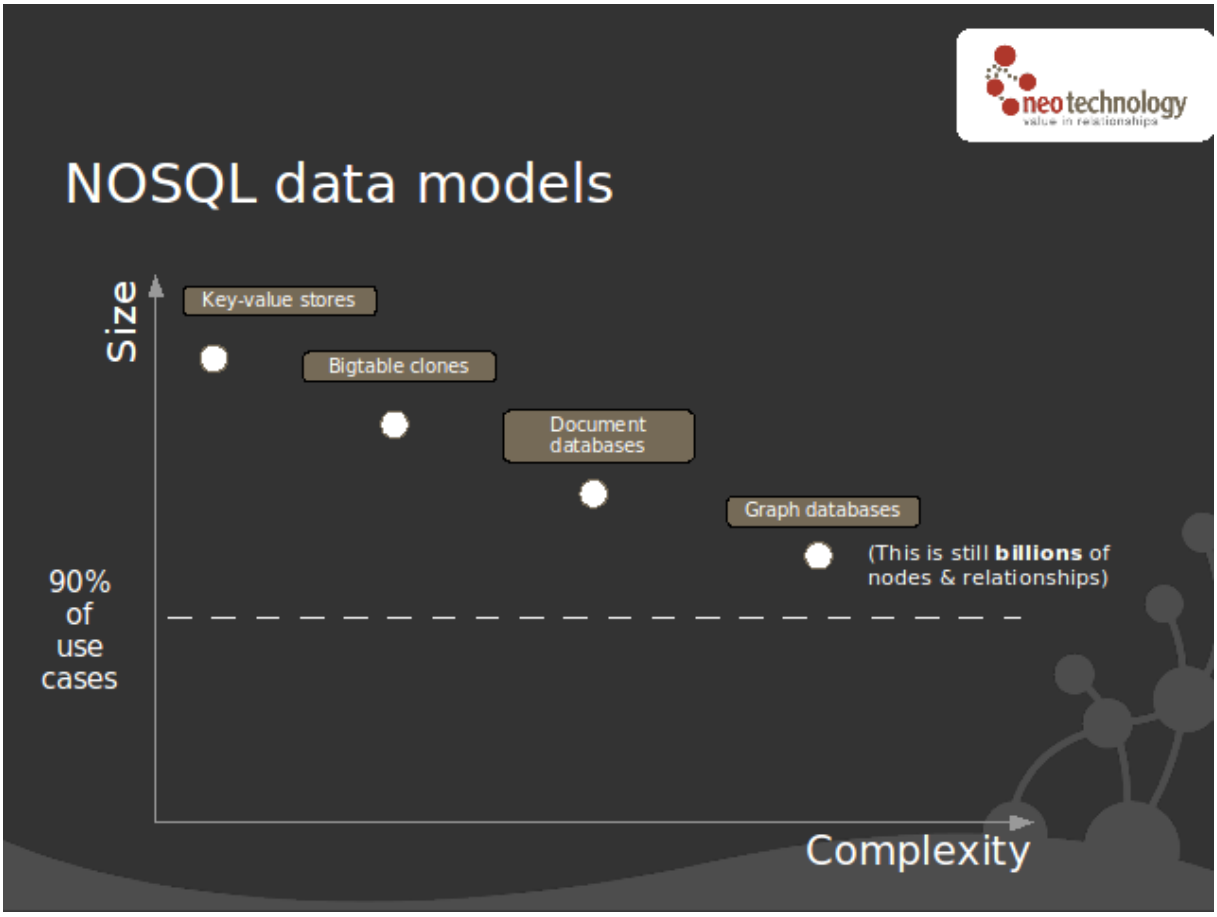
Táto vlasnosť kladie na systém požiadavky ako podpora replikácie a rozsekávania dát. Distribuované databázové systémy implementujú tieto techniky na systémovej úrovni. V prípade ich podpory, môžeme od systému požadovať:

- podporu replikcácie medzi geograficky oddelenými dátovými centrami
- možnosť pridania nového uzlu do distribuovaného databázového systému, bez nutnosti zásahu do aplikácie

Počet uzlov, na ktorých sa vykonáva replika dát a konfigurácia databázového systému, ktorá podporuje geograficky oddelené dátové centrá zároveň určujú akej miery je systém schopný odolávať chybám, medzi ktoré môžeme zaradiť poruchu uzlu alebo sieťové prerušenia.

Častým požiadavkom webových aplikácií, z dôvodu neustáleho nárastu dát, na databázový systém je podpora škálovania s cieľom zvýšenia veľkosti databáze. S neustálym vývojom nových aplikácií musíme uvažovať potrebu škálovania z pohľadu komplexnosti. V tomto prípade predpokladáme, že štruktúra dát ktoré do databázového systému ukladáme sa môže s postupom času meniť. Pojem škálovanie z pohľadu komplexnosti je popísaný v knihe... Schopnosť škálovania z pohľadu komplexnosti ovplyvňuje výber dátového modelu.

Obrázok XY zachytáva pozíciu NoSQL dátových modelov z pohľadu škálovania komplexnosti a veľkosti dát.



Obr. 4.1: Pozícia dátového modelu z pohľadu jeho škálovania podľa veľkosti a komplexnosti. Zdroj: Neo4J a NOSQL overview and the benefits of graph databases, Emil Eifrem, prezentacia.

Dátový model typu kľuč-hodnota a stĺpcovo orientovaný model (Bigtable clones) majú jednoduchú štruktúru, ktorá sa dá horizontálne škálovať. Nevýhodou tohoto prístupu je naopak to, že všetká 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 vhľadom na veľkosť dát. Podľa odhadov spoločnosti Neotechnology až 90% aplikáci, v prípade že sa nejedná o projekty spoločností Google, Amazon atď., spadá do rozmedia kde sa veľkosť záznamov pohybuje rádovo v miliardách. Za zmienku stojí fakt, že aj napriek tomu, že tieto dátové modely sú si navzájom izomorfné, vhodnosť ich použitia závisí na konkrétnom príklade a požiadavkoch na aplikáciu.

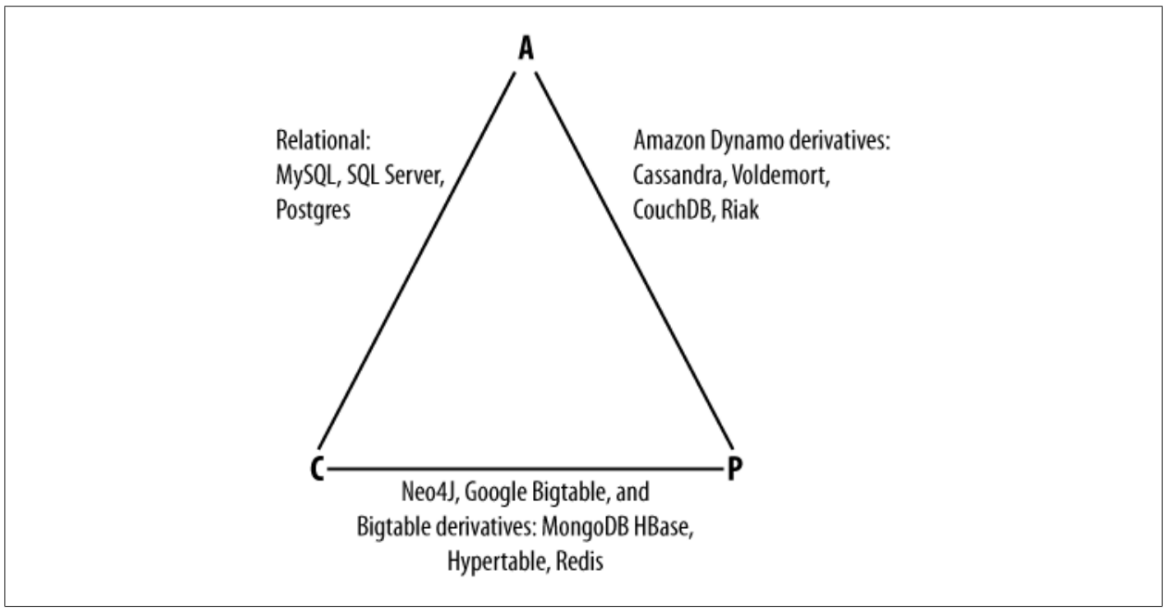
4.2.3 Elastickosť

Vďaka horizontálnemu škálovaniu sa snažime o zýšenie kapacity celkového dátového úložiska. Elastickosť škálovania popisuje ako sa daný systém dokáže vysporiadať s pridaním alebo

odobraním uzlu. U tejto vlastnosti sledujeme či je potrebné manuálne rebalancovanie dát, reštartovanie celého systému alebo zmena v uživateľskej aplikácii. Táto vlastnosť by ideálne mala zabezpečiť lineárne zvyšovanie výkonu u operácií ako je čítanie alebo zapisovanie dát.

4.2.4 Konzistencia dát

Poďla teórie CAP platí, že v prípade výskytu sieťových prerušení, ktoré sú súčasťou distribuovaného databázového systému nie je možné súčasne zaručiť vlasnosť konzistencie a dostupnosti. NoSQL systémy preto môžeme rozdeliť podľa tohoto modelu.



Obr. 4.2:

Umiestnenie niektorých NoSQL datázových systémvo sa môže meniť podľa ich konfigurácie.

Podpora replikácie, rozsekávania dát a zaradenie systému podľa modelu CAP určuje jeho dostupnosť.

4.2.5 Perzistentné úložisko

Typ perzistentného úložiska popisuje interný spôsob ukládania dát v databázovom systéme. NoSQL systémy môžu používať na ukládanie dát následujúce štruktúry:

- B-stromy
- distribuované hešové tabuľky
- Memtable / SSTable [1]

[1] Tieto štruktúry popíšeme v následujúcej sekcii

- spojové zoznamy
- operačná pämeť, ktorej obsah je pravidelných intervaloch ukládaný na pevný disk

Podľa požiadavkov na našu aplikáciu sme čiastočne schopný odhadnúť pomocou, ktorej štruktúry by sme mohli dosiahnúť čo najefektívnejšiu výkonnosť.

4.3 Výber NoSQL systémov

V predchádzajúcej sekcii sme tieto systémy rozdelili do štyroch hlavných kategórií podľa ich dátového modelu, ktorý je kľúčový pri výbere vhodného databázového systému podľa požiadavkov aplikácie. 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 **??**, preto túto kategóriu vynecháme.

NoSQL systémy môžeme rozdeliť podľa toho v akom prostredí pracujú. Väčšina týchto systémov vyžaduje ich inštaláciu na počítačové systémy, patria sem napríklad Voldemort, Cassandra, Riak, Hbase. Tieto systémy sú open-source. Okrem nich existujú distribuované databázové systémy, ktoré sú poskytované ako cloudové riešenie a to Amazon SimpleDB, Microsoft Azure SQL, Yahoo! YQL a prostredie od spoločnosti Google AppEngine. Tieto systémy poskytujú priamo rozhranie na prácu s dátami a funkčnosť perzistentného úložiska zabezpečujú poskytovatelia týchto služieb. Ich hlavným cieľom je zefektívnenie vývoja a nasadzovania aplikácií.

Podľa analýzy požiadavkov na našu aplikáciu a štruktúry dát, ktoré budeme do databázového systému ukládať nie je vhodné použitie databázových systémov s grafovým modelom a modelom kľúč-hodnota. Systémy s grafovým modelom sú určené na diametrálne odlišnú úlohu problémov naopak v prípade, použita systémov kľúč-hodnota by bola práca na strane aplikácie zbytočne náročna. Našim požiadavkom najlepšie vyhovuje stĺpcovo orientovaný model, ktorý sme sa rozhodli použiť pre návrh našej aplikácie.

V tejto časti práce sa zameriame na stručný prehľad a vzájomné porovnanie systémov, ktoré poskytujú stĺpcovo orientovaný model. Medzi tieto voľne dostupné (open-source) systémy patria HBase, Cassandra a Hypertable. Aj napriek totožnému dátovemu modelu, majú tieto systémy odlišné vlasnosti. Následujúca tabuľka zobrazuje popis vlasností, na ktoré sme sa zamerali pri výbere víťaznej dvojice.

Z porovania je vidieť, že systémy obsahuju množstvo spoločných vlastností. Pri výbere systémov sme zohľadnili aj ich praktické využitie spoločnostiami pôsobiacimi na trhu v produkčných podmienkách. Systém Cassandra je používaný spoločnosťou Facebook v aplikácii na súkromnú poštu. Medzi ďalšie požiadavky patrili podpora komunity, dokumentácia a vývojový cyklus týchto systémov. Z týchto systémov sme následne vybrali dva a to HBase a Cassandru. Dôvodom prečo sme zavrhli systém Hypertable je nepostačujúca dokumentácia, málo aktívna komunita a pomalý vývojový cyklus. V následujúcej sekcii popíšeme ich detaily.

Vlastnosti/Databázový systém	Hbase	Cassandra	Hypertable
Distribuovaný systém	áno	áno	áno
Dátový model	Bigtable [2]	Bigtable	Bigtable
Dotazovací model			
Klient	Thrift, REST	Thrift, Avro	Thrift, C++
Perzistentné úložisko	HDFS	LSS[3]	HDFS, KFS, LSS
SPOF [4]	áno	nie	áno
Pridanie uzlu do živého systému	áno	áno	
Podpora viacerých datacentier	áno	áno	áno
Rozsekávanie dát	áno	áno	áno
Replikácia	pomocou HDFS	áno	áno
Elastickosť	áno	áno	
Konzistencia	CP	AP	
Programovací jazyk	Java	Java	C++
MapReduce	áno	áno	áno
Komunita	+	+	-

Kapitola 5

Cassandra

Distribuovaný databázový systém Cassandra bol vytvorený pre interné účely spoločnosti Facebook v roku 2007. Cassandra slúžila na vyhľadávanie v súkromnej pošte, poskytovala úložisko pre indexy. Hlavnými požiadavkami na tento systém bolo zvládať miliardu zápisov denne, schopnosť škálovania podľa narástajúceho počtu používateľov, beh na spotrebných počítačoch a podpora replikácie medzi geograficky oddelenými dátovými centrami. Ďalším požiadavkom bola vysoká dostupnosť, teda aby chyba žiadného uzlu nespôsobila celkovú nedostupnosť systému. Cassandra je teda 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 distribuovaného databázového systému Dynamo od spoločnosti Amazon a zároveň ich kombinuje s dátovým modelom distribuovaného databázového systému Bigtable vytvoreného spoločnosťou Google. V následujúcom texte popíšeme hlavne princípy, na ktorých je tento systém založený.

5.1 Dátový model

Cassandra k dátovému modelu systému Bigtable pridala štruktúru pod názvom „super stĺpec“ (angl. super column). Základnou jednotkou dátového modelu je stĺpec. Stĺpec je tvorený názvom, hodnotou a časovým odtlačkom, ktorý využíva Cassandra pri riešení konfliktov. Skupina stĺpcov je identifikovná pomocou unikátneho kľúča a predstavuje riadok, avšak počet a názvy stĺpcov nie je potrebné vopred definovať. Zoradené riadky podľa hodnoty kľučov a v nich zoradené stĺpce obaľuje štruktúra pod názvom „rodina stĺpcov“ (ang. column family). Kľúče sú interne reprezentované ako reťazec znakov a zároveň zotriedené. Názvy stĺpcov možu byť viacerých typov ako napríklad Ascii, Utf-8, Byte 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 ukládať relevantné záznamy, ku ktorým budeme pristupovať spoločne, čím sa vyhneme zbytočným diskovým operáciam. Operácie nad stĺpcami, ktoré identifikuje daný kľúč sú atomické v rámci repliky. Operácie nad daným riadkom nevyužívajú zamykanie. Voliteľným príznakom, ktorý môžeme u stĺpcu nastaviť je parameter TTL (angl. time to live), ktorý po uplynutí časového intervalu označí dáta za zmazané.

Štruktúra super stĺpec je špecialny typ stĺpca, ktorý je tvorenými obyčajnými stĺpcami. Stĺpec typu super má názov a jeho hodnota je tvorená zoznamom názvov obyčajných stĺpcov. Tento prístup pridáva ďalšiu úroveň v štruktúre. Super stĺpce obaľuje podobná štruktúra pod názvom super-rodina stĺpcov (angl. super column family).

Keyspace definuje faktor replikácie a jej metódu, ktorá môže byť závyslá poprípade nezávislá na sieťovej topológii. Na keyspace sa môžeme pozerať ako na databázu v relačných databázových systémov obsahujúcu rodiny stĺpcov, ktoré môžeme prirovnať k tabuľkám v relačných databázach.

Aktuálna verzia Cassandri definuje maximálnu veľkosť dát 2GB, ktoré je možné uložit do jedného stĺpca a stanovuje limit dve miliardy pre maximálny počet stĺpcov v jednom riadku.

5.2 Rozdeľovanie dát

Kľúčovým požiadavkom systému Cassandra je jeho schopnosť škálovania do šírky, čo vyžaduje pridávanie nových uzlov. Tento požiadavok vyžaduje mechanizmus, ktorý zabezpečí dynamické 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 pre rýchly prístup k danej časti objektov nachádzajúcich sa na serveroch. Klient teda musí byť schopný určiť, ku ktorém kešovaciemu systému musí pristúpiť v prípade, že chce daný objekt. Predpokladajme, že klientom zabezpečíme výber jednotlivých kešovacím systémom pomocou hešovania s využitím lineárnej hešovacej funkciek (x -> ax + b (mod p), kde p je počet kešovacích systémov). Pridanie nového kešovacieho systému alebo jeho zlyhanie bude mať katastrofálny dopad na funkčnosť systému. V prípade, že sa zmení parameter p, teda počet kešovacích systémov každá položka bude odpovedať novej a zároveň chybnej lokácii. Tento problém rieši elegantne technika pod názvom úplné hašovanie (angl. consistent hashing) [15], ktorá sa využíva v distribuovaných systémoch pre prácu s distribuovanými hašovacími tabuľkami. Túto techniku taktiež využíva systém Cassandra.

Výstup hašovacej funkcie MD5 reprezentujeme pomocou „kruhu“, kde v smere hodinových ručičiek postupujeme od minimálnej hodnoty hešovacej funkcie (tj. 0) k maximálnej. Každý uzlol v systéme má pridelenú náhodnu hodnotu z tohoto rozsahu, ktorá určí jeho jednoznačnú pozíciu. Identifikácia uzlu v systéme, na ktorý sa uložia dáta reprezentované hodnotou kľúča sa vykoná aplikáciou hešovacej funkcie na dáta reprezentujúce kľúč. Na základe tejto hodnoty je jednoznačne určená pozícia v kruhu a v smere hodinových ručiek je vyhľadaný najbližší uzol. 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 zároveň prináša nevýhody, 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 mechanizmi na monitorovanie záťaže a automaticky presúva pozície uzlov. Taktiež je možné explicitne u každého uzlu stanoviť jeho polohu v kruhu pomocou zadania jeho identifikátora. Tento spôsob je vhodný v prípade, ak vieme predom určiť koľko uzlov bude obsahovať náš systém. V prípade zvyšovania počtu uzlov je možné tieto identifikátory a teda ich polohu v kruhu zmeniť za chodu systému,

čím sme schopný opäť dosiahnúť jeho rovnomerné vyváženie. Identifikátor polohy uzlov vieme určiť pomocou následujúceho programu, kde K je počet uzlov v systéme.

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

print tokens(K)
```

5.3 Replikácia

S úplným hašovaním úzko súvisý replikácia, ktorá zabezpečuje vysokú dostupnosť a odolnosť dát proti ich stráte (angl. durability). 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 na kruhu. V prípade zápisu koordinátor replikuje dáta na ďalších N - 1 uzlov. Cassandra podporuje viacero spôsobov pre umiestňovanie replík.

Jednoduchá stratégia

Táto stratégia umiestňuje replikú dát bez ohľadu na umiestnenie serverov v datacentre. Replika dát uzlu je uložená na jeho N-1 susedov v smere hodinových ručičiek. Z toho vyplýva, že každý uzol je zodpovendný za dáta, ktorých kľúče spadajú do jeho rozsahu a taktiež dáta, ktorých kľúče spravuje jeho N predchodcov.

Sieťová stratégia

Pri tejto metóde a úrovni replikácie s hodnotou aspoň tri, sme schopný zabezpečiť umiestnenie dvoch replík v rozdielnych rackoch, tretia replika bude umiestnená do iného datacentra. Táto stratégia je výhodna 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

Distribuovaný systém musí byť schopný odolávať chybám ako porucha uzlov alebo sieťové prerušenia. Podpora decentralizácie a detekcia chýb využíva mechanizmi založené na gosship protokoloch. Tieto protokoly slúžia pre vzájomnú komunikáciu uzlov vymnieňajúcich si navzájom doležité informácie o svojom stave. Periodicky v sekundových intervaloch každý uzol kontaktuje náhodne vybraný uzol, kde si overí či je tento uzol dostupný. Detekcia možného zlyhanai uzlu je realizovaná algoritmom s názvom Accrual Failure Detector [23].

Pridávanie novýchu uzlov, presun uzlov v rámci kruhu a iné operácie sa taktiež dejú pomocou Gosship protokolu. Tento protokol 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 s časovou zložitosťou O(1).

5.5 Perzistentné úložisko

Tento systém bol primárne navrhnutý tak aby spracúval vysoký tok dát pre zápis, s tým že čo najmenej ovlplyvní efektívnosť operácií na čítanie. Cassandra využíva ako perzistentné úložisko dát lokálny súborový systém.

5.6 Konzistencia

Konzistencia systému je maximálne konfigurovateľná a využíva princípy techník založených na protokoch kôra. Klient si môže nastaviť hodnotu R určujúcu koľko replík musí potvrdiť úspešnosť operácie čítania dát. Hodnota W určuje na koľko replík je potrebné vykonať zápis a následne vrátiť potvrdenie o jeho úspešnosti klientovi. V prípade, že platí vzťah $R + W > N$, kde N je počet replík tak sa jedná o silne konzistentný systém, naopak v prípade voľby klienta, kde $R + W < N$, sa jedná o slabú konzistenciu čím zaručíme vysokú dostupnosť.

5.7 Zápis dát

Ak uzol obrží požiadaok pre zápis, dáta sú zapísané do štruktúry pod názvom commit log, ktorá je uložená na lokálnom súborovom systéme a zabezpečí trvácnosť dát. Zápis do tejto štruktúry je vykonávaný sekvenčne čo umožnuje dosiahnúť vysokú priepusnosť. Dáta sú následne nahrané do štruktúry pod názvom memtable, ktorá sa nachádza v operačnej pamati a v prípade, že by tento zápis zlyhal alebo by došlo k neočakávanému zlyhaniu inštancie Cassandri je možné ich obnovenie z commit logu. Po dosiahnutí určitéh prahu, tj. počtu dát uložených v memtable sú tieto štruktúry asynchrónne zapísané do štruktúr pod názvom SSTable (Sorted String Tables), ktoré už nie je možné modifikovať pomocou aplikácie. Štruktúry SSTables sa následne zlievajú v pravidelných intervaloch na pozadí počas behu, táto operácia je neblokujúca. Počas zlievania SSTables dochádza k zotriedenému zlievaniu kľúčov, k nim prinaležiacím dát, odtraňovaniu dát určených na vymazanie a generovaniu nových indexov. Taktiež dochádza ku generovaniu štruktúr pod názvom Bloom filters[1] pre každú SSTable.

Zápis dát nevykonáva žiadne diskové operácie, ktoré by potrebovali čítať dáta, je atomický pre danú ColumnFamily a v prípade, že systém Cassandra beží je stále dostupný a rýchly.

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

5.8 Čítanie dát

V prípade požiadavku na načitanie dát, sa požadované dáta najprv hľadajú v štruktúrach memtable, ktoré sú uložené v operačnej pamati. Ak sa dané data nenachádzajú v operačnej pamati, vyhľadávanie sa uskutočnuje podľa kľúča v diskových štruktúrach SSTable. Kedže snahou systému je čo najefektívnejšie vyhľadávanie, využívajú sa bloom filtre. Bloom filtre sú nedeterministické algoritmy, ktoré dokážu otestovať či element patrí do množiny. Napriek ich nedeterminizmu generujú len falošné pozitíva. Pomocou nich je možné namapovať kľúče zo štruktúr SSTables do bitových polí, ktoré je možné uchovať v operačnej pamati. Vďaka tomu sa redukuje prístup na disk, keď hľadáme súbor, ktorý obsahuje dáta odpovedajúce hľadanému kľúču. Požiadavok pre čítanie dát možeme zaslať na ľubovoľný uzol.

5.9 Zmazanie dát

Keď vykonáme operáciu reprezentujúcu zmazanie dát, tieto dáta sa nevymažú okamžite. Namiesto toho sa vykoná operácia, ktorá dané dáta označkuje príznakom pod názvom tombstone. Po uplynutí doby, ktorá je štandardne nastavená na desať dní, sa tieto dáta odstránia pri procese zlievajúcom štruktúry SSTables.

5.10 Bezpečnosť

Implicitne Cassandra nevyužíva žiadne prvky, ktoré by poskytovali možnosť znemožnenia prístupu k dátam v nej uložených. K dispozícii je modul, ktorý umožnuje nastavenie auntentizácie na úrovni Keyspace-u pomocou textových hesiel alebo ich MD5 odtlačkov. Obmedzovanie prístupu k datám je preto potrebné zabezpečiť na aplikačnej úrovni.

Kapitola 6

HBase

V tejto kapitole 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.

Hadoop

Hadoop[1] je open source projekt v programovacom jazyku Java, ktorý tvorý distribuovaný súborový systém HDFS (Hadoop Distributed Filesystem) a framework MapReduce pre spracúvanie objemu dát v desiatkách PB [20]. Medzi hlavné požiadavky tohoto systému patria vysoká dostupnosť, škálovateľnosť a distribuovaný výpočet. Architektúra HDFS vychádza z princípov distribuovaného súborového systému Google File System[14] (GFS) a pôvod frameworku MapReduce pochádza taktiež od spoločnosti Google[7].

Súborový systém využíva architektúru master-slave. Uzol master, pod názvom Namenode, udržiava v operačnej pamäti metadata, ktoré popisujú štruktúru súborov, adresárov, reprezentujú mapovanie súborov na bloky a určujú ich umiestnenie na uzloch Datanode. HDFS predpokláda prácu so súbormi rádovo v desiatkách gigabajtov, ktoré sú interne reprezentované dátovými blokmi o štandardnej veľkosti 65 MB. Tieto bloky sú uložené v uzloch typu slave, ktoré sa nazývajú Datanode. Klient v prípade načítania súboru kontaktuje Namenode, ktorý mu poskytne informácie, na ktorých uzloch typu Datanode sa nachádzajú bloky reprezentujúce súbor a dátová komunikácia následne prebehne medzi klientom a uzlom Datanode. HDFS je optimalizovaný pre jednorázový zápis dát a ich následné mnohonásobné čítanie. Bloky sa replikujú na uzly Datanode. Štandardne je nastavená úroveň replikácie na hodnotu tri, teda každý blok je uložený trikrát.

Hlavným nedostatkom tejto infraštruktúry je fakt, že uzlol Namenode tvorí kritický bod systému, v prípade jeho nedostupnosti nie je možné pracovať s HDFS a prípadná strata 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 uloženia súboru, ktorý nepresahuje veľkosť jedného bloku je potrebné alokovať 300B dát. Ak uložíme 10,000,000

[1] http://hadoop.apache.org/

takýchto súborov veľkosť metadát, ktoré udržiava Namenode v operačnej pamati zaberie 3 GB. Celkový počet uložených súborov je obmedzený veľkosťou operačnej pamati 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 zadefinovali požiadavky v tretej kapitole.

HBase

Aplikácie ako HDFS alebo MapReduce slúžia na dávkové spracúvanie obrovského objemu dát. HBase je open source, distribuovaný, stĺpcovo orientovaný databázový systém, ktorý umožnuje prístup k veľkému objemu dát a ich zápis v reálnom čase. Ako perzistentné úložisko dát využíva distribuovaný súborový systém HDFS, je taktiež implementovaný v Jave a jeho architektornické koncepty vychádzajú z článku Bigtable od spoločnosti Google. HBase bol vytvorený spoločnosťou Powerset na konci roka 2006, pre potreby spracúvania obrovského objemu dát[22] a začiatkom roka 2008 sa stal oficiálnym podprojektom systému Hadoop.

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. Klúče sú reprezentované ako pole bajtov (Java byte[]), preto je možné použiť ľubovoľný typ dát napríklad reťazec alebo serializovanú dátovú štruktúru (JSON). Riadky sú radené podľa názvov kľúčov. Stĺpce sú organizované do skupín, ktoré sa nazývajú „rodiny stĺpcov“ (angl. column families). Obsahu každej bunky, ktorej pozíciu určuje riadok a stĺpec prináleži 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 (zýskava ju z operačného systému??) 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.

Dôležitým faktom je, že stĺpce, ktoré patria do rovnakej rodiny stĺpcov sú fyzicky uložené na tom istom mieste. Rodiny stĺpcov je potrebné zadefinovať 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 sychronyzáciu systémového času. V prípade veľkého časového rozdielu hrozí, že daná inštancia systému HBase sa nespustí.

6.2 Architektúra systému

HBase využíva architektúru master-slave. Uzol v role master sa nazýva HBase Master, uzly slave RegionServers. Pre chod systému je ďalej potrebná služba Zookeeper ref + pre synch?.

HBase master

Tento uzol v systéme vykonáva priradzovanie regiónov RegionServer-om, detekujem pridanie nového RS, jeho zlyhanie a balancuje záťaž na RS v prípade rozdelenia regiónu.

RegionServer

Region server sĺúži pre obsluhu klientských požiadavkov, samotný zápis alebo čítanie dát sa vykonáva medzi klientom a RS. Každý RS spravuje niekoľko regiónov, automaticky rozdeľuje regióny a informuje o tom uzol HBase master. Tieto uzly môžeme v systéme ľubovoľne pridávať alebo odoberať počas jeho prevádzky.

6.3 Rozdeľovanie dát

Základnou jednotkou, ktorá zabezpečuje rozdeľovanie dát a teda umožnuje horizontálne škálovanie a rovnomerné rozloženie záťaže v klastri je region. Region má náhodne vygenerovaný identifikátor a tvorí ho interval riadkov, kde posledný riadok do daného intervalu nepatrí. Tabuľka je tvorená regiónmi, pri jej prvotnom vytvorení ju zvyčajne reprezentuje jeden región. V prípade, že jej veľkosť dosiahne predom stanovenú hranicu (závyslé na konfigurácii), dojde k rozdeleniu riadkov do dvoch nových regiónov s podobnou veľkosťou. Tieto regióny sú v klastri distribuované na uzly typu RegionServer. Tento mechanizmus zabezpečuje, že do systému je možné uložiť tabuľku, ktorej veľkosť by nebolo možné spracovať pomocou jedného fyzického počítača.

Región je základný element, ktorý zabezpečuje dostpnosť a rovnomerné rozloženie záťaže.

6.4 Replikácia

HBase podporuje replikáciu v rámci viacerých geograficky oddelených datacentier. Replikácia funguje na rovnakom princípe ako v databázovom systméme MySQL [2], teda master-slave a je asynchrónna. Táto forma replikácie zanáša do daného distribuovaného systému eventual consistency na strane uzlov typu slave.

Primárnu replikáciu dát, ktorá zabezpečuje silnú konzistenciu a zabraňuje stráte dát je možné zabezpečiť pomocou perzistentného úložiska, v tomto prípade na úrovni HDFS.

6.5 Perzistentné úložisko

Tento distribuovaný databázový systém je schopný pracovať v lokálnom móde, kde vyššie spomínané komponenty bežia ako samostatné služby na jednom fyzickom uzly a ako úložisko sa využíva lokálny súborový systém.

V prípade distribuovaného módu rozlišujeme dva typy:

- pseudo distribuovaný mód, kde všetky komponenty bežia na jednom uzly

[2] http://dev.mysql.com/doc/refman/5.1/en/replication-formats.html

- distribuovaný mód, jednotlivé komponenty bežia na samostatných uzloch

Obidve konfigurácie môžu využívať ako perzistentné úložisko distribuovaný súborový systém HDFS, KFS alebo Amazon S3. Štandardne sa doporučuje použitie HDFS.

6.6 Konzistencia

Systém sa vyznačuje silnou konistenciou. Z modelu CAP splňuje CP, teda uprednosňuje konzistenciu pred dostupnosťou.

6.7 Zápis dát a čítanie dát

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

S týmto RS následne prebieha dátová komunikácia. Dáta sú zapísané do štruktúry HLog (typu WAL), ktorá je uložená na HDFS. Po potvrdení o úspešnom zápise sú data nahrané do štruktúry MemStore, ktorá je uschovaná v operačnej pamäti. V prípade, že RegionServer obrdží požiadavok na čítanie dát, dáta sú načítane buď zo štruktúry MemStore alebo HFile.

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

6.8 Zmazanie dát

U operácii zmazania dát je potrebné špecifikovať či chceme zmazať dáta staršie určitá časová značka poprípade dáta odpovedajúcej konkrétnej časovej značke. Keď vykonáme operáciu reprezentujúcu zmazanie dát, tieto dáta sa nevymažú okamžite z dôvodu, že štruktúry HFile sú po zápise nemenné a aby sa nevykonávali zbytočné diskové operácie. Namiesto toho sa vykoná operácia, ktorá dané dáta označkuje príznakom pod názvom „tombstone“. Tieto dáta odstránia pri procese zlievajúcom štruktúry HFile.

6.9 Bezpečnosť

Distribuovaný databázový systém je možné nasadiť do claudu. V prípade, použitia verejných audov môže hroziť nebezpečenstvo zneužitia našich dát treťou stranou a preto je potrebné aplikovať bezpečnostné mechanizmi. Pri nasadení systému do privátneho klaudu zasa môžeme požadovať viacero úrovní ochrany pre prístup k dátam. Hadoop a HBase aktuálne poskytujú prvok autentizácie pomocou Kerberosu. Možnosť pridania autorizácie na úrovni tabuliek a rodiny stĺpcov je neustále vo vývoji.

Secure HBase, Hadoop Group Trend Micro: Andrew Purtell, Gary Halmling, Joshu Ho, Eugene Koontz, Mingjie Lail http://www.slideshare.net/ghelmling/secure-hbase-hw2010

https://issues.apache.org/jira/browse/HBASE-1697 https://issues.apache.org/jira/browse/HBASE-3025

http://hbaseblog.com/2010/10/11/secure-hbase-access-controls/ http://hbaseblog.com/2010/07/21/u and-running-with-secure-hadoop/

Kapitola 7

Testovanie výkonnosti

Výkonové porovnanie NoSQL systémov je zložitá úloha, neexistuje univerzálny nástroj, ktorým by bolo možné tieto systémy navzájom porovnať. Tieto systémy sa vyznačujú rôznymi vlastnosťami ako typ konzistencie, dostupnosť, optimalizácia pre zápis alebo čítanie a ich výber závisí na prípade použitia. Všeobecný nástroj pre ich porovnanie by preto nemal žiadne opodstatnenie. Taktiež nie sú k dispozíci žiadne všeobecné techniky, ktorými by bolo možné testovať napríklad konzistenciu týchto distribuovaných systémov, spoľahlivosť a iné. Výkon týchto systémov môže ovplyvňovať faktor replikácie, spôsob rozdeľovania dát alebo úroveň 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 zameriam na popis výkonnostných testov, ktoré som vykonal v reálnych podmienkach a pri ktorých som pozoroval ako systémy HBase a Cassandra zvládajú vysoký zápis v rôzných konfiguráciách faktoru replikácie, počtu uzlov v klastri a úrovňou konzistencie.

7.1 Testovacie prostredie

Pre výkonnostné testovanie som mal 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 softvarovú konfiguráciu testovaných aplikácií popíšem jednotlivo v nasledujúcich podkapitolách.

Hardvérová konfigurácia

- 4 jádrový 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 systém Debian GNU/Linux Lenny x64, Sun Java JDK 1.6.0_+88. Na každom uzle bol deaktivovaný odkladací priestor (angl. swap). Za účelom monitorovania bol použitý softvér Zabbix, VisualVM, htop, iostat a dstat.

Sieťová konfigurácia

Hodnota maximálnej reálnej sieťovej priepustnosti medzi dvoma uzlami bola zmeraná pomocou aplikácie nuttcp[1] s výslednou hodnotou 940 Mbit.

7.2 Popis testovacej metodológie

Nad oboma distribuovanými databázovými systémami sme vykonali testy na základe, ktorých sme sa snažili identifikovať ako tieto systémy ovplyvňuje rôzna úroveň replikácie, konzistencie, pozorovali sme ich schopnosť horizontálneho škálovania a chovanie sa pod záťažou. Testy boli zamerané na zápis dát, ktorý je kritickým prvkom pre potreby našej aplikácie.

7.2.1 Testovací klient

Testovací klient bol multivláknová aplikácia založená na princípe producent - konzument, kde konzumenti predstavovali jednotlivé vlákna vykonávajúce zápis do databázových systémov. Optimálny počet paralelne zapisujúcich vlákien bol stanovený na hodnotu 50, pri ich zvyšovaní sa zvyšovala latencia a nedošlo k zvýšeniu dátového toku pre zápis. Kritickým bodom bolo v tomto prípade, zabezpečiť počas celej doby jednotlivých testov rovnaké zaťaženie každého uzla v klastri. 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 8,000,000 riadkov. Každý riadok obsahoval jeden stĺpec o veľkosti 1000 B, ktorého obsah tvorili náhodné data. Počet celkovo zapísaných dát bol 7,6 GB. Uzly obsahovali 4 GB fyzickej operačnej pamati a v prípade, zápisu malého množstva dát by tieto dáta nemuseli byť zapísané počas testu na pevný disk, časť ich obsahu by mohla byť uložená v štrukúrach Memtabe a Memstore. Zvolili sme preto cca dvojnásobnú veľkosť zapisovaných dát v porovnaní s veľkosťou operačnej pamati, čím počas testu docielime so 100% pravdepodobnosťou zaťaženie pevných diskov. Za zmienku stojí fakt, že na Internete sú dostupné články, testujúce tieto systémy, ktoré počas testu neprekročia 1/3 RAM?? [].

7.2.3 Testovací prípad pre čítanie dát

Dôvodom tohto testovacieho prípadu bolo určiť ako dané systémy zvládajú čítanie dát, pretože pre výpočet štatistík pomocou MapReduce bude mať rýchlosť čítania dát výrazný vplyv na celkovú dobu trvania výpočtov.

[1] http://www.wcisd.hpc.mil/nuttcp/

7.2.4 Zaťažovací test

Cieľom bolo zistiť stabilitu klastra v prípade, keď bude pod sústavným zápisom, budú v ňom prebiehať časté operácie pre zápis štruktúr Memtable a Memstore na pevný disk a ich následné zlievanie. Tento test sme vykonali po dobu piatich hodín. Počas niektorých testovacích prípadov sme použili dvoch klientov z dôvodu aby sme zaručili maximálnu saturáciu 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 pre zápis 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 (ang. Java Virtual Machine) bola nastavená na na 1 GB.

Meranie maximálnej rýchlosi zápisu sme testovali v troch konfiguráciach. 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 klastrov bola následovná:

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

Počas jednotlivých testov sa na súborový systém zapisovali tri rôzne veľkosti súborov, kde v HDFS bola zachovaná štandardná veľkosť bloku tj. 65MB. TODO:rozpisat Pomocou MapReduce sa na každom uzle typu slave paralelne vykonával zápis súborov o danej veľkosti. Testy boli vykonané nástrojom TestDFSIO a na každý uzol sa zapisovalo 12 GB dát. Každý test bol vykonnaný trikrát a výsledná hodnota bola určená ako aritmetický priemer. Výsledky testu, ktoré zobrazuje tabuľka 7.1 poukazujú na fakt?,

~~Veľkosť súboru [MB]~~	Klaster		
	A	B	C
128	287 MB/s	102 MB/s	190 MB/s
812	371 MB/s	85 MB/s	162 MB/s
4096	433 MB/s	85 MB/s	163 MB/s

Tabuľka 7.1: Výkonnosť HDFS pre zápis dát

že zvýšenie faktoru replikácie ma 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 (prípad klastrovv konfigurácii B a C), jeho výkonnosť vzrastie

lineárne, čo potvrdzuje vysokú škálovateľnosť systému. Rôzne veľkosti zapisovaných súborov sme volil s cieľom určiť či daný systém dosahuje lepšie výsledky pre zápis malých alebo veľkých súborov. Počas testov systém nevykazoval žiadne známky preťaženia, nebolo indetifikované žiadne úzke hrdlo.

7.4 HBase

popis toho ze sme pomocou API vytvorili tabulku s tym ze sme ju predrozdelili na viacero prazdnych regionov - kazdy na inom node - load balancing HBase currently does not do well with anything about two or three column families so keep the number of column families in your schema low – nezmyselne testy v paperoch Compaction is currently triggered by the total number of files under a column family. Its not size based. ako to je v C ? preto vela compactions a write bottleneck?

bloom filtre default nope .. zapli sme

Počet uzlov	Replikácia	Čas	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: Zápis riadkov o veľkosti 1000 B

Počet uzlov	Replikácia	Čas	Riadok/sek	Priepustnosť [MB/s]
1	1	246	4069	4
3	1	117	8561	8
3	3	127	7936	8
6	3	190	11904	12

Tabuľka 7.3: Čítanie riadkov o veľkosti 1000 B

Počet uzlov				
Veľkost riadku	3	4	5	6
1 KB	32	35	36	
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: Maximálna priepustnosť klastru v MB/s

7.5 Cassandra

Distribuovaný databázový systém Cassandra som podrobil viacerým testom, ktorých zámer a výsledky popíšem v následujúcej časti. Testy boli zamerané hlavne na stabilitu systému, sledovali rýchlosť zápisu dát a ako sa táto rýchlosť mení na základe rôzneho počtu uzlov, úrovne konzistencie a replikácie. Pri testovaní bola použitá verzia Cassandra-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 120 MB. Každý uzol zaberal na hašovom kruhu rovnaký úsek.

Tabuľka 7.5 obsahuje výsledky z viacerých meraní, pri ktorých sme do systému zapisoval osem miliónov riadkov s jedným stĺpcom o veľkosti 1000 B. Pre zapisované riadky boli použité ako kľúče hodnoty v rozsahu nula až celkový počet riadkov. Počas testovania boli všetky uzly rovnako zaťažené.

Počet uzlov	Replikácia	Konzistencia	Čas	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: 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 som zdvojnásobil počet uzlov z troch na šesť (riadok 3,5) vzrástla priepusnosť zápisu o cca 50%.

Replikácia

V prípade zvýšenia úrovne replikácie z 1 na 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 byt dáta zapísané na dve repliky.

Čítanie dát

Nad dátami uloženými v databázovom systéme budem vykonávať výpočet štatistík, preto je cieľom určiť ako Cassandra zvláda čítanie dát. Počas testu, ktorého výsledky sú zaznamenané v tabuľke 7.6 som zo systému načítal milión riadkov o veľkosti 1000 B. Riadky boli čítané náhodne a kešovanie kľúčov a riadkov, ktoré Cassandra podporuje bolo vypnuté.

Počet uzlov	Replikácia	Konzistencia	Čas	Riadok/sek	Priepustnosť [MB/s]
1	1	One	383	261	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: Čítanie riadkov o veľkosti 1000 B

Zaťažovací test

V tomto teste som zaťažil klaster po dobu zápisu 20 min. V určitých prípadoch som použili pre zápis do klastra dvoch klientov z dôvodu aby sa vylúčilo úzke hrdlo na strane klienta. Cieľom bolo zistiť stabilitu a maximálna priepustnosť klastra počas sústavného zápisu dát. Tabuľka 7.7 zobrazuje priemerný dátový tok počas doby testovania v MB/s, ktorý sa zapisoval do klastra.

Počas testu som klaster monitorovali a zistil viacero závažných dôsledkov. Na všetkých uzloch prebiehali veľmi časté GC kolekcie (angl. Garbage collections) z dôvodu častého zápisu štruktúr memtable na pevný disk. Podľa hardverovej špecifikácie všetky uzly disponovali minimálnou veľkosťou operačnej pamäti (4 GB), preto z dôvodu 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 disku, ktoré Cassandra zlievala na pozadí (ang. compactions), čo spôsobovalo záťaž vstupno výstupných operácií (angl. I/O wait). V prípade, takto zaťaženého systému a veľkého množstva SSTable súborov by bola operácia čítania náročná na diskové operácie, pretože dáta patriace do jednej rodiny stĺpcov by boli uložené vo veľko množstve samostatných súborov a ich načítanie by vyžadovalo zvýšené množstvo diskových operácií (angl. seek).

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 zohrávajú hlavnú úlohu V/V diskové operácie.

7.6 Voľba databázového systému

Počet uzlov				
Veľkost 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: Maximálna priepustnosť klastru v MB/s

Kapitola 8

Návrh systému

V tejto kapitole popíšeme návrh systému, ktorý bude slúžiť na archiváciu emailov a spĺňať požiadavky, ktoré sme pre tento systém definovali. V prvej časti sa zameriame na výber vhodných open source nástrojov pre implementáciu prototypu a následne 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 archivovať 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 viacerý príjemcovia. Emailová správa predstavuje semištrukturovaný dokument [21] v textovej podobe, ktorého syntax popisuje štandard RFC 2822 [1] z roku 2001 pod názvom Formát Internetovej správy (angl. Internet Message Format). Dokument RFC 2822 nahradzuje a upravuje pôvodné RFC 822 pod názvom Štandard pre formát Internetových textových správ ARPA z roku 1982 (angl. Standard for the Fromat 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 rožšírujú štandardy, pod názvom MIME (ang. Multipurpose Internet Mail Extensions), RFC 2045, RFC 2046[10, 11], RFC 2047, RFC 2048 a RFC 2049. Tieto štandardy pridávajú možnosť použitia iných znakových sád ako US-ASCII, ďalej umožnujú štruktúrovať telo správy (vnorené správy rfc822), definujú formát a typy pre zasielanie príloh atď.

Zber emailových objektov je realizovaný na unixových serveroch, ktoré budú používať emailový server Qmail[1]. Tento server bude zároveň realizovať antispamovú kontrolu kontrolu pomocou modulu Qmail-scanner[2], ktorý je naprogramovaný v jazyku Perl[3]. Po doručení emailového objektu na server je emailový objekt spracovaný programom Qmail, ktorý volá obslužný modul qmail-scanner a následne dokončí doručenie správy. Tento modul sme vhodne

[1] http://cr.yp.to/qmail.html
[2] http://qmail-scanner.sourceforge.net
[3] http://www.perl.org

modifikovali pre potreby nášho systému. Modifikovaný súbor je súčasťou zdrojových kódov tejto práce. Výstupom je dvojica súborov a to obálka, ktorá obsahuje viacero štatistických údajov a samotný textový súbor reprezentujúci emailovú správu v pôvodnej podobe. Príklad obálky znázorňuje obrázok 8.1. Detailný popis tejto štruktúry sa nachádza na webovej adrese http://qmail-scanner.sourceforge.net/.

```
Tue, 15 Mar 2011 10:12:09 CET Clear:RC:1(88.208.65.55):SA:1 0.007811 9508
odosielatel@server prinemca@server2 predmet <13001802281039145460aq>
1300180329.16836-0.forid1:5987 priloha1:134
```

Obr. 8.1: Obsah obálky z programu qmail-scanner

8.2 Analýza dát

Jedný z hlavých požiadavkov 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 (angl. media type) a bližšie špecifikovaný podtyp, napríklad „image/gif“. Norma definuje základných päť typov médií a to text, image, audio, video a application. V prípade, našej aplikácie má zmysel využiť deduplikáciu na všetky tieto typ s výnimkou typu „text/plain“, kde predpokladáme, že sa jedná o bežnú textovú správu napísanú uživateľom.

Program pre analýzu a deduplikáciu emailovej správy bol napísaný v programovacom jazyku Python[4]. 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 datasetu ukázala, že XY % emailov tento požiadavok nespĺňa. Z celkovej množiny emailov o veľkosti 98000 bolo 0.022% emailov, ktoré nespĺňali štruktúru definovanú normou RFC 2045. Metódu deduplikácie sme riešili nasledujúcim postupom:

- analýzou 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 emaile sme nahradili značkou v tvare MARK:H
- dáta reprezentujúce prílohu sme do databáze 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 návhru sme využili poznatky získané štúdiom architektúry databázového systému Cassandra. Schéma je tvorená pomocou štyroch rodín stĺpcov a to:

[4] http://python.org

- 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 emailov
- lastInbox v chronologickom časovom poradí, podľa hodnoty Date v hlavičke emailu, zaznamenáva emaily daného užívateľa

Tradičné techniky pre popis databázových schémat nie je možné aplikovať na databázové systémy ako napríklad 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 schémat. Článok pod názvom techniky pre definíciu štruktúr pomocou diagramov v cloude a návrhové vzory (angl. Cloud data structure diagramming techniques and design patterns [6]) definuje stereotypy pre diagramy v jazyku UML a obsahuje vzory pre popis štruktúry týchto dát. Obrázok 8.2 znázorňujúci databázovú schému nášho modelu aplikuje tieto techniky.

Ako jedinečný identifikátor emailovej správy v databáze využivame nasledujúcu schému:

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

V tejto schéme reprezentuje:

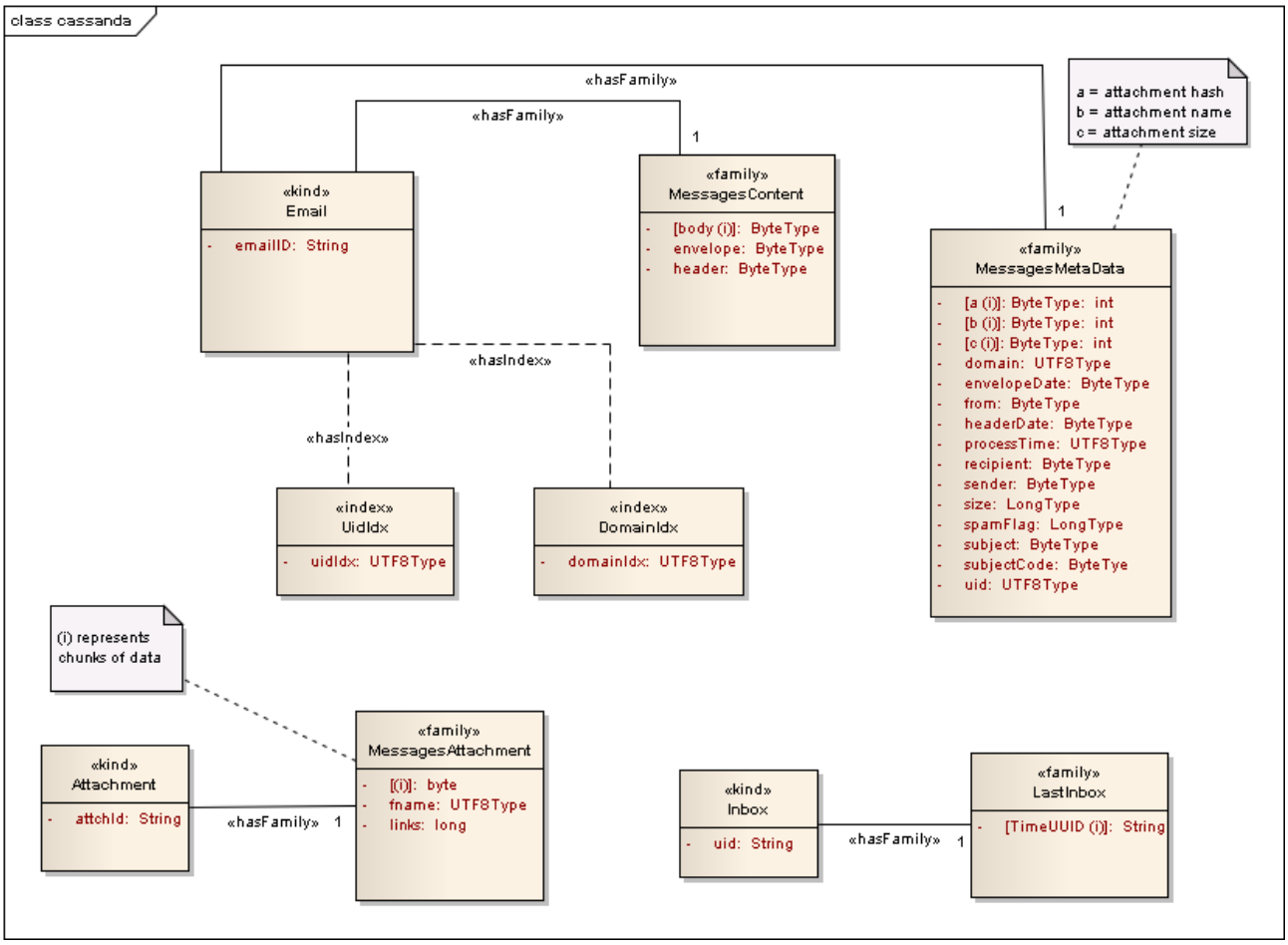
- uid emailovú adresu príjemcu, v tvare jan@mak.com
- MessageID je ide identifikátor z hlavičky daného emailu
- date je časová značka reprezentujúca čas kedy bol email prijatý emailovým serverom (formát: rok, mesiac, deň, hodina, minúta, sekunda)
- emailID je výstup hašovacej funkcie SHA2 v hexadecimálnom tvare

V predchádzajúcej kapitole sme zistili, že Cassandra nie je optimalizovaná pre zápis blokov dát (ang. blob), ktorých veľkosť prevyšuje 5 MB avšak optimálne výsledky pre zápis dosahuje pri veľkosti blokov 512 kB. Preto prílohy a dáta reprezentujúce email v prípade, že ich veľkosť presahuje 1 MB zapisujeme do samostatných stĺpcov o veľkosti 512 kB. Toto rozdeľovanie dát na menšie bloky vykonávame na aplikačnej úrovni na strane klienta. Názvy stĺpcov číslujeme vzostupne v rozmedzí 0-N, kde N je počet blokov. Spätnú rekoštrukciu dát vykonáva klient.

8.4 Fultextové vyhľadávanie

Fultextové vyhľadávanie realizujeme pomocou samostatného NoSQL systému Elasticsearch[5]. Hlavný index, ktorý obsahuje všetky zaindexované dáta ma názov emailArchive, a delíme ho na dva typy s názvom email a envelope. Schéma týchto typov obsahuje polia poďla, ktorých chceme v emailovom archíve vyhľadávať a jej reprezentáciu zapísanú vo formáte JSON znázorňuje obrázok 8.3.

[5] http://elasticsearch.org



Obr. 8.2: Databázová schéma

8.5 Implementácia

V programovacom jazyku Python sme implementovali klienta pre zápis dát do databáze Cassandra a ElasticSearch. Jednou z najdoležitejších vlastností týchto klientov je voľba úrovne konzistencie pri zápise. Našou prioritou je integrita dát a od databáze požadujeme silnú konzistenciu. Zvolili sme mód kvóra (ang. Quorum), ktorá zabezpečí zápis dát na N / 2 + 1 replík a klient následne obdrží potvrdenie o úspešnosti zápisu, inak sa zápis zopakuje. Klient, ktorý slúži na čítanie dát z databáze využíva taktiež mód kvóra. Tieto vlastnosti nám zabezpečujú, v prípade použitia faktoru replikácie tri (dáta sa v databázovom systéme nachádzajú trikrát), silnú úroveň konzistencie na strane klienta a databáze. Analýza emailovej správy a jej deduplikácia spotrebúva hlavne CPU zdroje. Moderné procesory obsahuju viacero jadier, tento fakt môžeme využiť pre paralelizované spracúvanie emailov, teda každé jadro CPU bude spracúvať súčasne jednu emailovú správu.

```
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 fultextové vyhľadávanie

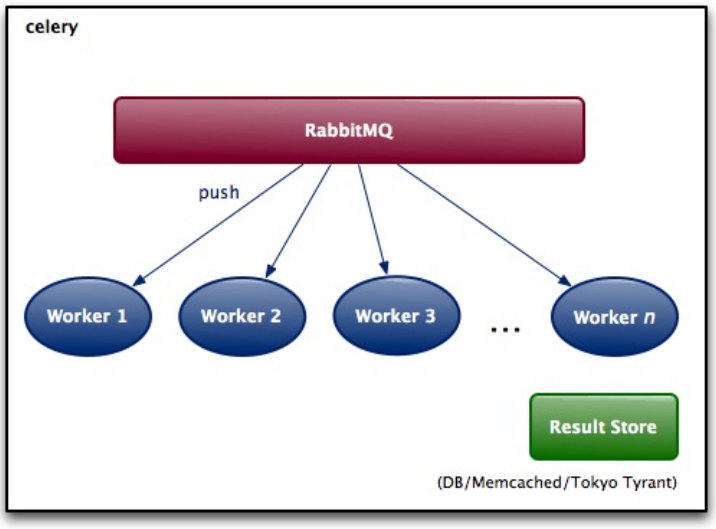
Celery

Paralelizáciu našej aplikácie sme zabezpečili pomocou využitia asynchrónnej fronty úloh pod názvom Celery[6], ktorá využíva architektúru distribuovaného predávania správ (ang. distributed message passing). Architektúru znázorňuje obrázok 8.4. „Pracovníci“ (angl. workers) reprezentujú samostatné procesy v našom prípade proces pre analýzu a deduplikáciu emailu, ktoré môžu bežať paralelne. Ako sprostredkovateľ (ang. broker) je použitá aplikácia RabbitMQ[7], ktorý obdrží od klienta správu a uloží ju do fronty. Správa obsahuje identifikátor emailovej správy, v tomto prípade cestu na lokálnom súborovom systéme k súboru reprezentujúcom email. Táto správa je následne zaslaná ľubovoľnému pracovníkovy (v našom prípade klient kykonávajúci analýzu a deduplikáciu emailu), ktorý ju spracuje. Táto architektúra je plne distribuovaná, dokáže odolávať chybám (napr. v prípade výpadku elektrickej energie, správy nadalej pretrvávajú vo fronte).

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 dispozícií ľubovoľný pracovník, je emailová správa spracovaná pomocou nášho analyzátora a následne zapísaná do databáze Cassandra a fultextového systému Elasticsearch. Na testovacie účely sme nemali k dispozícii reálny dátový tok emailov. Vrstvu reprezentujúcu Qmail sme nahradili modulom, vytvárajúcim nové úlohy

[6] http://celeryproject.org
[7] http://www.rabbitmq.com



Obr. 8.4: Architektúra Celery, Zdroj: [online], http://ask.github.com/celery/getting-started/introduction.html

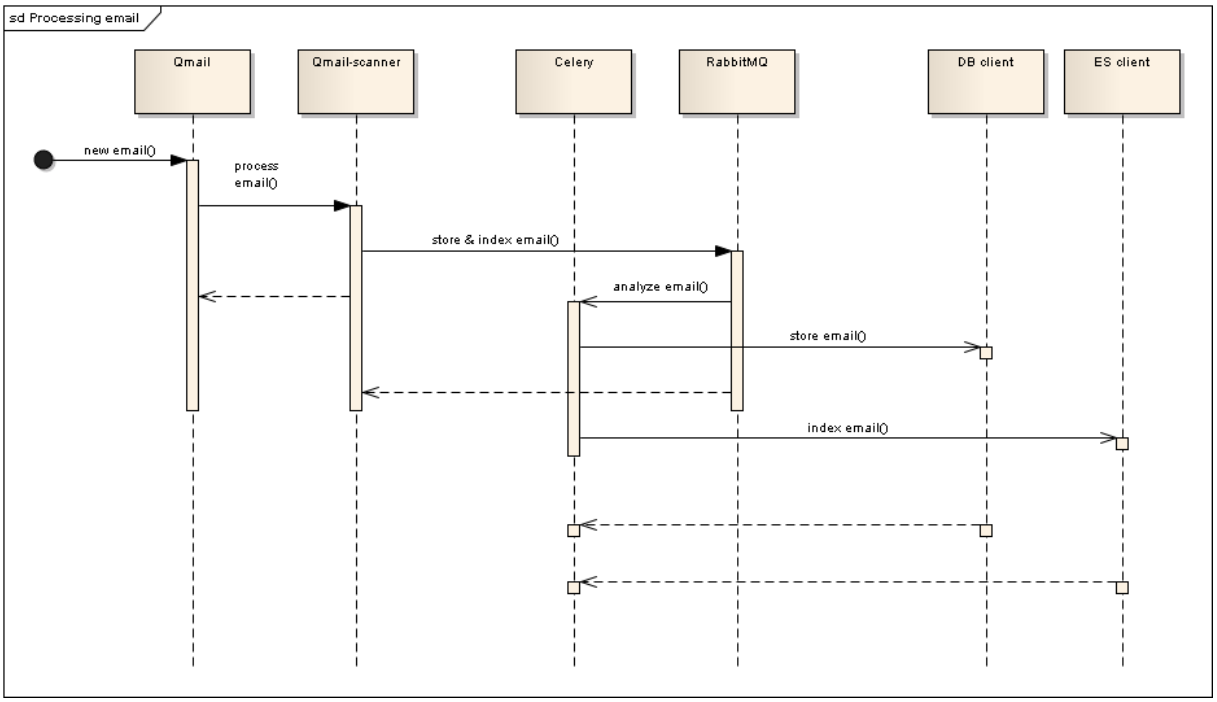
prechádzaním lokálneho súborového systému, ktorý obsahoval testovaciu množinu emailový správ.

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. Tvorba aplikácií v tomto frameworku prebieha v Jave, je náročná a okrem toho programový model MapReduce obsahuje viaceo problémov. Model napríklad neobsahuje primitíva na filtrovanie, agregáciu, spájanie dát a je potrebná ich vlastná implementácia. Tieto nedostatky rieši nástroj Pig[] vďaka, ktorému sme boli schopný spracúvať meta dáta uložené v databáze. Obrázok 8.6 zobrazuje programovú ukážku, ktorá slúži na výpočet veľkosti najvačsieho emailu pre domény, ktorých emaily archivujeme. Pre jednoduchosť sme vynechali časti, ktoré slúžia na načítanie dát z databáze a obsahujú uživateľsky definovanú funkciu v programovacom jazyku Java, ktorá slúži na predprípravu vstupných dát. Podpora uživateľom definovaných funkcií je jednou z ďalších výhod nástroja Pig.

8.5.3 Webové rozhranie

TODO web IFACE – pristup koncovych uservo k archivu a diskusia ohladne bezpecnosti - strucny popis django aplikacie.



Obr. 8.5: Spracovanie emailu

```
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

8.6 Overenie návrhu

Pomocou vyšie popísaného návrhu a implementovaných nástrojov sme overili funkčnosť nami navrhovaného modelu. Vhodná voľba daných verzií u aplikácií Celery a RabbiMQ 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 aplikácii. 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 aktuálna verzia 0.8.? a medzitým vznikla kvalitná dokumentácia od spoločnosti Datastax[8].

[8] https://datastax.com

Konfigurácia

Hardverová konfigurácia bola totožná s konfiguráciou z kapitoly ???. Na šiestich serveroch bola nainštalovaná databáza Cassandra 0.7.3, Hadoop 0.20.2, dvojica serverov obsahovala klientskú aplikáciu a Celery 2.6. V úlohe sprostredkovateľa bola použitá aplikácia RabbitMQ 2.1.1, ktorá bola nainštalovaná na samostatnom serveri.

Overenie integrity dát

Databázový kluster sme naplnili testovacími dátami obsahujúcimi emaily o objeme 300 GB. Následne sme nasimulovali 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. Tento test prebehol bez akejkoľvek chyby.

Dosiahnuté výsledky

Doležitým pozorovaním bol fakt, že z celkového objemu emailových správ 300 GB sa po deduplikácií príloh tento objem znížil na 99 GB, teda došlo k 30% úspore diskovej kapacity. Štruktúry do ktorých sme ukladali dáta pre potrebu štatistík zaberali 0,4% z celkového objemu dát, čo je zanedbateľná položka.

ES rychlost pre vyhladavanie bola do XY ms.

Kapitola 9

Záver

Literatúra

[1] Internet message format, 2001.

[2] Simple mail transfer protocol, 2001.

[3] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, Paul Gauthier. Cluster-base scalable network services, 1997.

[4] A. B. Bondi. Characteristics of Scalability and Their Impact on Performance, 2000.

[5] Brewer. Towards robust distributed systems., 2000.

[6] David Salmen, Tatiana Malyuta, Rhonda Fetters, Normert antunes. Cloud Data Structure Diagramming Techniques and Design Patterns, 2010.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[8] E. F. Codd. A Relational Model of Data for Large Shared Data Banks, Jún, 1970. `www.seas.upenn.edu/~zives/03f/cis550/codd.pdf`.

[9] Eben Hewitt. Cassandra: The Definitive Guide, November 2010.

[10] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies, 1996.

[11] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part two: Media types, 1996.

[12] 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.

[13] J. F. Gantz, J. Mcarthur, and S. Minton. The expanding digital universe. *Director*, 285(6), 2007.

[14] 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.

[15] 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.

[16] Michael Stonebraker. The Case for Shared Nothing Architecture, 1986. `http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf`, stav z 28. 2. 2011.

[17] A. O. R. W. Paper. Why Cloud-Based Security and Archiving Make Sense, March 2010. `www.google.com/postini/pdf/why_cloud_based_wp.pdf`, stav z 28. 2. 2011.

[18] D. Pritchet. BASE: An Acid Alternative.

[19] R. Shoup. The eBay Architecture, Striking a balance between site stability, feature velocity, performance, and cost, November 2006. `www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf`, stav z 28. 2. 2011.

[20] 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.

[21] J. Udell. *Practical Internet GroupWare*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[22] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[23] Xavier Défago, Péter Urbán, Naohiro Hayashibara, and Takuya Katayama. The Phi accrual failure detector., 2004.

Dodatok A

Zoznam použitých skratiek

WPAN Wireless Personal Area Network

Dodatok B

Inštalačná a užívateľská príručka

B.0.1 Inštalácia simulátoru OMNeT++ pre platformu Linux

1. Stiahnutie archívu obsahujúceho zdrojový kód zo stránok
   http://www.omnetpp.org/omnetpp
2. Prekopírovanie archívu do adresára /usr/local/
3. Rozbalenie archívu pomocou príkazu tar zxvf omnetpp-4.0.src.tgz
4. Do užívateľského profilu .bash_profile alebo .profile pridáme riadok
   export PATH=$PATH:/usr/local/omnetpp-4.0/bin
5. Je potreba zabezpečiť prítomnosť následujúcich balíkov v systéme

   sudo apt-get install build-essential gcc g++ bison flex perl tcl8.4 tcl8.4-dev
   tk8.4 tk8.4-dev blt blt-dev libxml2 libxml2-dev
   zlib1g zlib1g-dev libx11-dev

6. Prevedieme následujúce príkazy:
   cd /usr/local/ometpp-4.0
   ./configure
   ./make
7. Spustenie OMNeT++ s IDE pomocou príkazu omnetpp

B.0.2 Inštalácia mnou modifikovaného Mobility Frameworku

1. Stiahnutie súborov Mobility frameworku z svn http://my-svn.assembla.com/svn/mframework/, poprípade prekopírovanie adresára mf2o4 z priloženého CD do adresára /usr/local/
2. Import MF do aplikácie OMNeT++
   (a) Po spustení aplikácie Omnet, klikneme na oblasť „Project explorer“, pravým tlačítkom a zvolíme položku „Import...“
   (b) Zvolíme „General->Existing project into Workspace“

(c) V položke „Select root directory“, zvolíme cestu k adresáru mf2o4, tj. /usr/local/mf2o4

(d) Pomocou CTRL+B, preložíme zdrojové súbory

B.0.3 Práca s modelom IEEE 802.15.4

Vo vývojom prostredí Omnetu si otvoríme v oblasti „Project explorer“ adresárovú štruktúru mf2o4, kde si následne otvoríme adresár networks a v ňom adresár ieee802.15.4. V tomto adresári sa nachádzajú aj xml súbory popisujúce antény. Otvoríme si súbor omnetpp.ini, tento súbor je hlavným konfiguračným súborom modelu simulácie. Zahrnul som do neho ukážkové nastavenia viacerých modelov, ktoré som simuloval. Samotná simulácia sa potom spustí otvorením súboru omnetpp.ini a následným kliknutím na tlačítko „Run“ z menu aplikácie.

Dodatok C

Obsah priloženého CD

Následujúci obrázok C.1 zobrazuje štruktúru priloženého CD.

```
.
|-- data
|   |-- XBee.xlsx                    - popis charakteristik antén z meraní
|   |-- Graphs.xls                   - grafy z realnych meraní
|   `-- Graphs2.xls                  - grafy zo simulácie
|-- mf2o4                            - modifikovaný Mobility framework
|   `-- networks
|       `-- ieee802154               - model IEEE 802.15.4
|           `-- results              - výsledky zo všetkých uskutočnených simulácii
|-- readme.txt                       - popis adresárovej štruktúry
|-- text
|   `-- Lenart-thesis-2009.pdf       - text bakalárskej práce vo formáte PDF
`-- zoznamCD
```

Obr. C.1: Výpis priloženého CD