FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Využití grafových databází pro pokročilou analýzu dat
Student:	Bc. Juraj Polačok
Vedoucí:	Ing. Marek Sušický
Studijní program:	Informatika
Studijní obor:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Pro vyšetřování podezřelých vazeb či finančních toků se typicky používají standardní databázové stroje, ale v poslední době se objevují implementace grafových databází.
Cílem práce je provést rešerši existujících grafových databází a na základě výsledků vybrat vhodnou databázi pro pokročilou analýzu velkých grafů. Výběr musí brát v úvahu, že očekávané datasety budou obsahovat miliony uzlů a desítky milionů hran. Do vybrané databáze budou následně importována data a provedena analýza úloh detekce anomalit, opakování, clustering a výpočet vybraných SNA metrik. Z provedených úloh bude provedeno měření a zhodnocení možnosti využití v komerčním projektu.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Karel Klouda, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 15. ledna 2019

Diplomová práca

Využitie grafových databáz pre pokročilú analýzu dát

Bc. Juraj Polačok

Katedra . . . Aplikované matematiky
Vedúci práce: Ing. Marek Sušický

9. mája 2019

Poďakovanie

Chcel by som poďakovať Ing. Marekovi Sučickému, vedúcemu diplomovej práce, za odborné vedenie, pomoc a ochotu pri tvorbe tejto práce. Ďalej by som chcel poďakovať firme Profinit EU, s.r.o. za možnosť realizácie tejto práce. V neposlednom rade ďakujem svojim rodičom a priateľke za podporu počas celého štúdia.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 9. mája 2019 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Juraj Polačok. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu

Polačok, Juraj. *Využitie grafových databáz pre pokročilú analýzu dát.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práca sa zaoberá rešeršou dostupných grafových databáz, ktoré podporujú horizontálne škálovanie. Práca sa snaží stručne vysvetliť základné technológie, ktoré sa využívajú v spomínaných grafových distribuovaných databázach. Na základe rešerše je následne vybraný vhodný kandidát, na ktorom bol prevedený benchmark jednotlivých grafových algoritmov. V poslednej časti je zhrnutie, ktoré obsahuje potenciálne prínosy pre komerčné použitie.

Klíčová slova Apache Hadoop, distribuované úložisko, grafové databázy, distribuované grafové databázy, JanusGraph, YARN, SparkGraphComputer

Abstract

This thesis deals with research of available graph database, which supports horizontal scaling. Thesis explains basic technologies which are used in today's distributed graph databases. Based on this research one candidate is selected. On this candidate, the benchmark was made, which tested various graph algorithms. Last part of this thesis deals with potentional benefits of using this graph database in commerce projects.

Keywords Apache Hadoop, distributed storage, graph databases, distributed graph databases, JanusGraph, YARN, SparkGraphComputer

Obsah

Úvod **1**

1 Motivácia a cieľ práce **3**
- 1.1 Motivácia . . . 3
- 1.2 Cieľ práce . . . 3

2 Úvod k distribuovanému spracovávaniu dát **5**
- 2.1 Big Data a ich spracovávanie . . . 5
- 2.2 História vzniku Hadoopu . . . 5
- 2.3 Apache Hadoop . . . 7
- 2.4 CAP teorém . . . 14

3 Rešerš grafových databáz pre veľké grafy **17**
- 3.1 Vymedzenie pojmov . . . 17
- 3.2 Motivácia použitia grafových databáz pre analýzu . . . 33
- 3.3 Grafové databázy . . . 33
- 3.4 Sumarizácia databáz . . . 47

4 Grafové algoritmy **49**
- 4.1 Grafové metriky . . . 49

5 Realizácia **53**
- 5.1 Dataset . . . 53
- 5.2 OLAP - SparkGraphComputer, YARN . . . 56
- 5.3 Zhodnotenie výsledkov . . . 60
- 5.4 Možnosti použitia v komerčnej sfére . . . 60

Záver **63**

Literatúra **65**

A Zoznam použitých skratiek **71**

B Obsah priloženého CD **73**

Zoznam obrázkov

2.1 Interakcia medzi NameNode a Datanode . . . 9
2.2 MRV1 . . . 11
2.3 YARN . . . 12

3.1 Apache Giraph - Sharded Aggregator . . . 24
3.2 Apache Giraph - Facebook benchmark . . . 25
3.3 TinkerPop2 - jednotlivé moduly . . . 27
3.4 TinkerPop3 - previazané moduly . . . 28
3.5 Ukážka prechádzania grafom v Gremline . . . 31
3.6 CAP teórem . . . 37
3.7 Benchmark databáz . . . 45
3.8 Nevhodná distribúcia uzlov . . . 46
3.9 Vhodná distribúcia uzlov . . . 47

5.1 Benchmark malého datasetu . . . 59
5.2 Benchmark veľkého datasetu . . . 60

Zoznam tabuliek

3.1 Rozdiely medzi TinkerPop2 a TinkerPop3 29
3.2 Benchmark - načítanie dát - graph 500 39
3.3 Benchmark - načítanie dát - Twitter 40
3.4 Benchmark - veľkosť dát 40
3.5 Benchmark - K skoková cesta - Twitter 40
3.6 Benchmark - škálovanie TigerGraph 41

5.1 Rozhranie uzlu fyzická osoba 55
5.2 Rozhranie uzlu právnická osoba 55
5.3 Rozhranie spojenia 55

Úvod

Pre rôzne analýzy dát (vyšetrovanie podozrivých väzieb, podvodov, vylepšenie služieb, . . .) sa typicky používajú štandardné relačné databázové stroje. V dnešnej dátovej dobe sa generuje obrovské množstvo dát, ktoré má zmysel analyzovať a spracovávať.

Klasické relačné databázové stroje narážajú na limitácie pri spracovávaní veľkého množstva dát, znamých pod názvom "big data". V poslednej dobe vznikajú nástroje, pomocou ktorých je možné tieto dáta spracovávať. Vzhľadom na veľké množstvo dát by analýza týchto dát na jednom výpočtovom uzle trvala veľmi dlhú dobu (častokrát neprijateľne dlhú dobu). Táto problematika bola vyriešená spôsobom, v ktorom sa dáta rozdistribuovali do viacerých výpočtových uzlov, čo umožnilo dané dáta spracovávať paralelne. Výpočty na distribuovaných systémoch vo väčšine prípadov neprebiehajú nad relačnými databázami ale nad tzv. NoSQL databázami. Jedným z typov takýchto databáz sú práve grafové databázy, ktorými sa táto práca bude zaoberať.

Grafové databázy je vhodné použiť na situácie, v ktorých hlavnú úlohu zohrávaju väzby medzi entitami. Hlavným rozdielom grafových databáz oproti iným databázam je práve to, že väzby sú fyzicky uložené, teda nie je potrebné cez časovo náročné výpočty riešiť spájanie tabuliek (prípadne duplikovať dáta).

Hlavným cieľom tejto práce je vytvoriť rešerš existujúcich grafových databáz a na základe výsledkov vybrať vhodnú databázu pre pokročilú analýzu veľkých grafov. Pri výbere je podstatným atribútom to, že datasety v komerčnom projekte budú obsahovať milióny uzlov a desiatky miliónov hrán. Práca sa snaži detailne zhrnúť použité pojmy nakoľko pri výbere jednotlivých databáz sa častokrát spomínajú pojmy z "Big data" sveta, preto pre komplexné pochopenie problematiky sú v tejto práci vysvetlené všetky súvisiace pojmy.

Po vybraní vhodnej databázy budú do tejto databázy naimportované datasety a budú vytvorené analýzy - detekcia anomalít, clusteringu a výpočet SNA

metrík. Z vytvorených analýz bude vytvorené zhodnotenie možnosti využitia v komerčnom projekte.

Kapitola 1

Motivácia a cieľ práce

1.1 Motivácia

Táto práca má za úlohu porovnať dostupné grafové databázy pre spracovanie veľkých grafov a následne vybrať najvhodnejšiu z nich. Pri výbere databázy je dôležité zohľadniť fakt, že táto databáza bude využívaná na grafy obsahujúce milióny uzlov a hrán. Ako príklad možno uviesť bankové transakcie, ktoré každým dňom pribúdajú vysokým tempom. Na spracovanie týchto dát (za účelom poskytovania lepších služieb, detekovania podvodov, ...) je možno využiť grafové databázy. Výsledok tejto práce má potenciál umožniť tvorbu pokročilej analýzy nad veľkým datasetom, ktorá by bez využita grafových databáz nebola možná (spracovanie dát by trvalo radovo týždne).

1.2 Cieľ práce

Cieľom tejto diplomovej práce je:

- Vytvoriť rešerš existujúcich grafových databáz
- Na základe výsledkov vybrať vhodnú databázu pre pokročilú analýzu dát
  - detekcia anomalít
  - clustering uzlov
  - výpočet SNA metrik
- Zhodnotiť využite v komerčnom projekte

Kapitola 2

Úvod k distribuovanému spracovávaniu dát

2.1 Big Data a ich spracovávanie

Pojem big data možno preložiť ako veľké dáta. Presná odpoveď na otázku, čo možno klasifikovať ako big data, nie je známa. V článku [1] autor uvádza, že do kategórie big data možno zaradiť dáta, ktoré sú natoľko veľké, že náklady na ich uloženie, prípadne spracovanie sú cenovo vyššie ako samotná hodnota týchto dát. Autor ďalej uvádza, že najlepšou definíciou vyššie spomínaného názvu je množina akýchkoľvek dát, ktoré sú natoľko veľké, rýchlo vznikajúce alebo premenlivé, že nie je rentabilné ich ukladať, alebo spracovávať v bežných nástrojoch, alebo úložiskách.

Veľké spoločnosti, napríklad Google, Yahooo! sa s týmito problémami potýkali už v minulosti a danú situáciu vyriešili vytvorením platformy, na ktorej je možné tieto dáta efektívne a lacnejšie spracovávať. V roku 2006 firma Yahoo vytvorila framework Hadoop, ktorý je opensource a ktorý sa masívne v nasledujúcich rokoch rozšíril. Táto platforma je ideálna pre spracovávanie big data a niektoré grafové databázy na spracovanie veľkých dát sú postavené priamo nad týmto frameworkom Hadoop.

2.2 História vzniku Hadoopu

V priebehu roku 1997 začal Doug Cutting implementovať prvú verziu programu Lucene. Program Lucene slúži ako vyhľadávacia fulltext knižnica. Vyhľadávač funguje na princípe vytvárania indexov jednotlivých termov vo vete. Vďaka tomuto vyhľadávaču je možné rýchlo a jednoducho odpovedať na otázku, v akých dokumentoch sa daný text vyskytuje. Autor Doug Cutting vydal tento program v roku 2000 ako opensource, pričom v roku 2011 sa Lucene presunul pod licencie Apache Software Foundation. [2]

V nasledujúcom roku začali práce na podprojekte s názvom Apache Nutch, ktorý sa zaoberal indexovaním webových stránok. Jednalo sa o projekt, ktorý prehľadával webové stránky a ich obsah zaindexoval ku príslušnej stránke, ktorá bola aktuálne prehľadávaná. Na tomto podprojekte Doug Cutting spolupracoval s Mike Cafarellom. [2]

Hlavným problémom ale bola rýchlosť. Na jednom počítači bolo možné dosiahnúť rýchlosť indexovania 100 stránok za sekundu. Pri veľkosti webu (stovky miliónov stránok) by pri použití jedného počítača trvalo zaindexovanie príliš dlhú dobu. Pri snahe zvýšiť rýchlosť indexovania rozšírili počet počítačov z jedného na štýri, pričom sa jednalo o manuálnu konfiguráciu (pridanie ďalších počítačov nebolo jednoduché a vyžadovalo značný zásah do logiky aplikácie). [2]

Oporným bodom pri návrhu distribuovaného výpočtu bolo distribuované úložisko, ktoré potrebovalo spĺňať tieto podmienky:

- bez schémy
- stále
- dokáže sa vysporiadať so zlyhaním komponentov
- dokáže automatický vyrovnávať záťaž

V roku 2003 Google vydal paper s názvom "Google File System Paper", ktorý spĺňal všetky body, ktoré boli popísané vyššie. V roku 2004 začali implementovať vlastný file systém, v ktorom vychádzali práve zo špecifikácie, ktorú popisoval Google. Novovzniknutý súborový systém nazvali Nutch Distributed File System (NDFS). [2]

Problematika ukladania dát bola vyriešená novým NDFS, no paralelné spracovanie stále predstavovalo problém. Bolo potrebné vytvoriť výpočtový model, ktorý dokáže tento novovzniknutý súborový systém efektívne využívať. Výpočtový model potreboval dosahovať čo najvyššiu mieru paralelizmu - ideálne lineárnu. Podobne ako v minulom kroku, Google v decembri 2004 vydal paper "MapReduce: Simplified Data Processing on Large Clusters". Kľúčovou vlastnosťou tohoto výpočtového modelu bolo to, že výpočet prebieha tam, kde sa nachádzajú dáta. Paper riešil tri základné problémy: [2]

- paralelizáciu - ako paralelizovať výpočet
- distribúciu - ako zvládať distribúciu dát
- odolnosť proti zlyhaniu

V júli 2005 bol tento výpočtový model s názvom MapReduce integrovaný do programu Nutch. Vo februári 2006 Cutting použil NDFS a MapReduce z programu Nutch a vytvoril nový projekt s názvom Hadoop. V januári

2006 spoločnosť Yahoo! zamestnala Doug Cuttinga aby im pomohol premigrovať ich systém do Hadoopu. Tento framework sa rýchlo uchytil, nakoľko bol zasadený do ekosystému Yahoo!, v ktorom sa úspešne presadil. Rok na to veľké spoločnosti ako Facebook, Twitter, LinkedIn začali tento framework tiež využívať. [2]

2.3 Apache Hadoop

Apache Hadoop je opensourcový softvér, ktorý umožňuje spoľahlivý, škálovateľný a distribuovaný výpočet. Jedná sa o framework, ktorý umožňuje distribuovaný výpočet na veľkých dátach, ktoré sú rozdistribuované naprieč clusterom. Na výpočet sa využíva jednoduchý progamovací model. [3]

Dôležité je podotknúť, že sa jedná o model, ktorý dokáže operovať nad stovkami komoditných výpočtových uzlov, tj nie je potrebné využívať drahé, vysoko dostupné komponenty. Systém počíta s tým, že sa jednotlivé jednotky môžu v priebehu výpočtu odpojiť/pokaziť. Ak počas výpočtu dôjde k takej situácii, nebude to viesť ku zrušeniu aktuálne bežiaceho výpočtu, ale výpočet dobehne v poriadku. Jedná sa o framework, ktorý zaobaľuje tieto moduly: [3]

- Hadoop Common - knižnice potrebné pre podporu ostatných modulov
- HDFS, viď 2.3.1
- Hadoop MapReduce, viď 2.3.4
- Hadoop YARN, viď 2.3.5
- Hadoop Ozone
  - objektové úložisko pre Hadoop
  - jedná sa o alternatívny file system ku HDFS, ktorý slúži na ukladanie malých objektov
  - dostupné od verzie 3.2
- Hadoop Submarine - jadro pre strojové učenie v Hadoope

2.3.1 HDFS - Hadoop distributed file system

HDFS je distribuovaný súborový systém, ktorý je navrhnutý pre komoditný hardware. Má mnoho podobností s už existujúcimi distribuovanými súborovými systémami. Hlavným rozdielom v porovnaní s konkurenciou je to, že systém je vysoko tolerantný voči poruchám, tým pádom je ideálnym kandidátom na beh na nízkonákladovom hardwari. HDFS poskytuje vysokú priepustnosť dát, preto je vhodný pre aplikácie, ktoré pracujú s veľkými dátami. [4]

Medzi hlavné vlastnosti HDFS patrí:

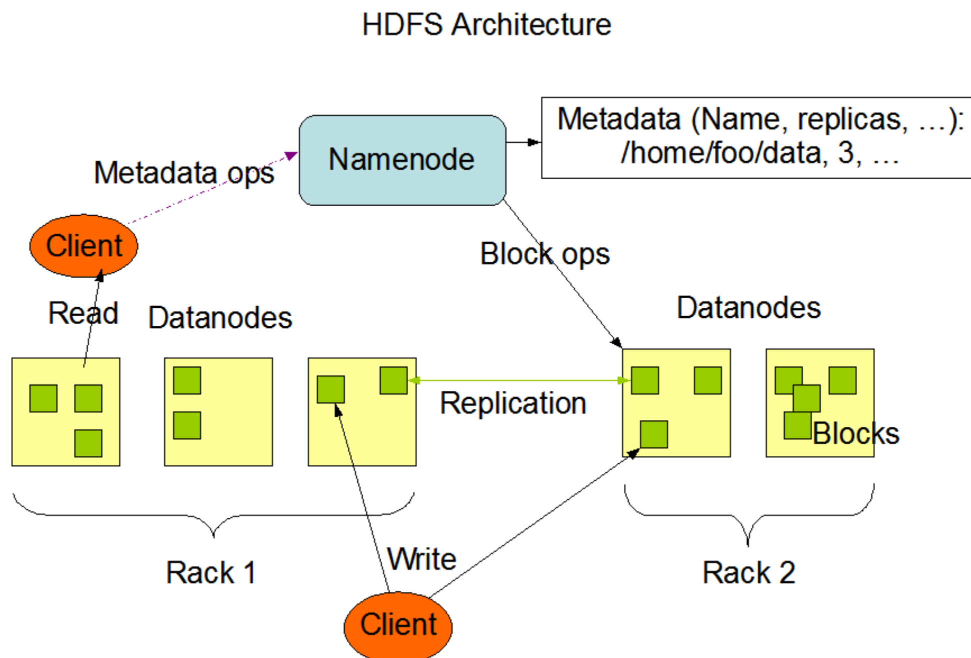
- **Porucha hardwaru** - na poruchu hardwaru sa pozerá skôr ako na pravidlo než na výnimku. Detekcia porúch a následná oprava a zotavanie je rýchle. Táto vlastnosť je jednou z hlavných architektonických cieľov HDFS
- **Streamovanie dát** - HDFS je predovšetkým určené na dávkové spracovávanie. Uprednostňuje sa veľká priepusnosť dát pred nízkou odozvou získavania konkrétných dát.
- **Veľké datasety** - Typicky sa pracuje so súbormi rádovo gigabajtových až terabajtových veľkostí.
- **Jednoduchý koherenčný model** - Model s princípom zapíš raz, čítaj mnohokrát
- **Presun výpočtu je lacnejší ako presun dát** - Výpočet nad dátami je výrazne efektívnejší ak prebieha na výpočtovom uzle, ktorý obsahuje potrebné dáta. Samotný presun dát z jedneho uzla na druhý je časovo náročná operácia, hlavne ak sa jedná o veľké objemy dát.
- **Prenosnosť naprieč rôznym hardwarovým aj softwarovým platformám** - HDFS je navrhnuté na jednoduchú prenosnosť medzi rôznymi platformami.

2.3.2 NameNode a DataNody

HDFS je postavený na architektúre master/slave. V úlohe master je NameNode, ktorý zodpovedá za systémový menný priestor (anglicky známe pod názvom namespace) a reguluje prístupy k súborom klientom. Uzol, ktorý pracuje s dátami sa nazýva "DataNode". HDFS poskytuje súborový systém, ktorý umožňuje klientom ukladať súbory do adresárov. Interne je ale každý súbor rozdelený na jeden alebo viac blokov, ktoré sú následne spracované v DataNodoch. NameNode je zodpovedný za otváranie, zatváranie, premenovanie súborov a priečinkov. Tiež je zodpovedný za mapovanie jednotlivých blokov jednotlivým DataNodom. Na druhú stranu, DataNody sa starajú o čítanie, zapisovanie dát, vytváranie nových blokov, mazanie aj replikáciu. NameNode je v podstate len správca. Klient sa pripojí ku NameNode s požiadavkou na zápis. NameNode následne pridelí klientom dostupný DataNode, na ktorý klient priamo začne zapisovať súbory. DataNode sa následne postará o replikáciu blokov na iné DataNody. NameNode eviduje, ktoré bloky sú kde zapísané. Pre lepšiu predstavu, viď 2.1. [4]

2.3.3 Replikácia

Vzhľadom k tomu, že HDFS je navrhnutý tak, aby fungoval na lacnom hardwari, je štandardné, že môže dôjsť k výpadku nejakého stroja. Z toho dôvodu



Obr. 2.1: Interakcia medzi NameNode a Datanode. Prevzaté z [4]

je dôležité riešiť replikáciu, aby nedošlo k strate dát, ktoré obsahuje stroj, ktorý sa odpojil. Typicky sa používa replikačný faktor 3. V praxi to znamená, že každý blok je súčasne replikovaný na 3 rôznych DataNodoch. Životnosť jednotlivých DataNodov sa zisťuje prostredníctvom "Heartbeat". NameNode periodicky kontroluje životnosť svojich DataNodov práve cez spomínany Heartbeat. Ak nejaký DataNode nevráti Heartbeat späť Namenodu, tak ho NameNode označí sa mŕtvy. Vzhľadom k tomu, že NameNode presne vie, aké dáta sa nachádzali na mŕtvom DataNode, začne všetky tieto dáta replikovať na živý DataNode. DataNody zvyknú byť umiestnené v jednotlivých rackoch. HDFS cluster môže pozostávať z viacerých rackov. Pri replikačnom faktore 3 sú dáta replikované spôsobom, v ktorom sa dve repliky nachádzajú na jednom racku a posledná, tretia, na inom racku. (Z pohľadu bezpečnosti by bolo ideálne, aby každá replika bola v inom racku, ale táto cesta bola zvolená ako kompromis medzi bezpečnosťou a rýchlosťou zápisu.) [4]

2.3.4 Hadoop MapReduce

Hadoop MapReduce je softwarový framework, ktorý slúži na jednoduchú implementáciu aplikácie, ktorá potrebuje spracovať veľké množstvo dát paralelne na clusteri, ktorého výpočtové uzly pozostávajú z komoditného hardwaru, pričom výpočet dobehne spoľahlivo aj v prípade, že dôjde k výpadku jedného

z použitých výpočtových uzlov. [5]

MapReduce framework je postavený na master/slave architektúre. Master sa nazýva ResourceManager, ktorý je len jeden. Slave sa označuje NodeManager (typicky je tento výpočtový uzol zhodný s dátovým nodom - DataNode). Klient pre vytvorenie mapreduce programu potrebuje špecifikovať vstupné dáta, výstupné dáta, logiku pre map fázu a pre reduce fázu. Po spustení tejto aplikácie ju prevezme ResourceManager a zistí, nad akými dátami aplikácia pracuje a rozdistribuje výpočet na uzly, ktoré dané dáta obsahujú (dáta sú uložené v HDFS). [5]

Logika MapReduce je založená na dvoch fázach: Map a Reduce. V map fáze prebehne výpočet, ktorý vytvorí mapu pozostávajúcu z kľúča dátového typu K1 a hodnoty dátového typu K2. Táto mapa sa vytvorí na každom výpočtovom uzle, na ktorom prebiehal výpočet. Dôležité je podotknúť, že map fáza vo väčšine prípadov prebieha nad DataNodom, ktorý obsahuje potrebné dáta k vytvoreniu tejto mapy. Následne prichádza fáza reduce, v ktorej sa agregujú jednotlivé mapy (dôjde k spojeniu (merge) cez kľúče). Hodnotou teda bude list pozostávajúci z prvkov dátového typu K2. [5]

2.3.4.1 Názorná ukážka

Tento princíp sa dá názorne ukázať na jednoduchom príklade. Úlohou programu je zistiť, koľkokrát sa dané slovo vyskytuje v texte.

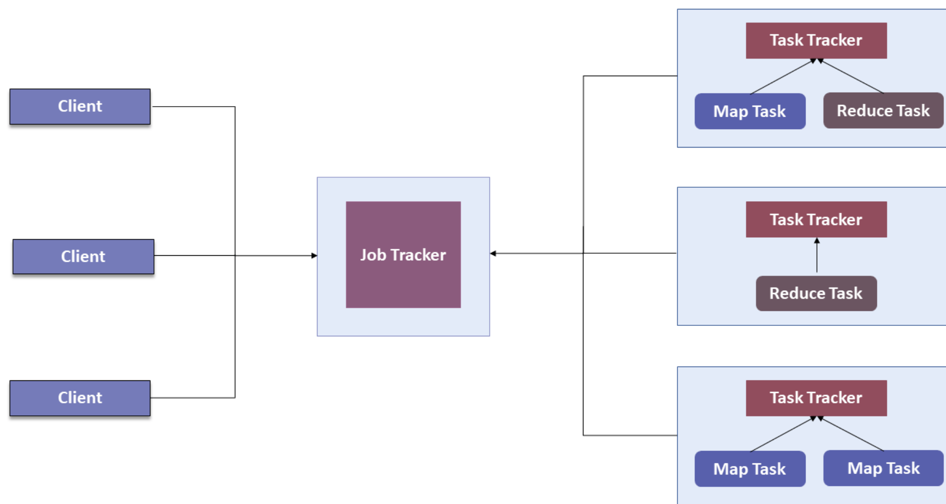
Predpokladajme, že existuje súbor s týmto obsahom: `"Hello World Bye World Hello Hadoop Goodbye Hadoop"`, pričom prvá polovica je uložená na DataNode1 `("Hello World Bye World")` a zvyšná časť na DataNode2 `("Hello Hadoop Goodbye Hadoop")`.

Po spustení programu sa ResourceManager postará o to, aby spustil výpočet na NodeManageri, ktorý sa nachádza na rovnakom clusteri ako DataNode, ktorý obsahuje príslušné dáta. NodeManager, ktorý je nainštalovaný na výpočtovom uzle, ktorý obsahuje DataNode1, pomenujme NodeManager1, pre DataNode2 ho pomenujme NodeManager2.

Výsledkom map fázy na NodeManager1 bude mapa: {`"Hello": 1, "World": 1, "Bye": 1, "World": 1`}. Výsledkom map fázy na NodeManager2 bude: {`"Hello": 1, "Hadoop": 1, "Goodbye": 1, "Hadoop": 1`}.

Po úspešnom skončení map fázy začne prebiehať reduce fáza. Vstupom do reduce fázy bude táto mapa: {`"Hello":[1,1], "World":[1,1], "Bye":[1], Hadoop":[1,1], "Goodbye":[1]`}.

Výstupom bude agregácia - súčet hodnôt v poli pre každý kľúč. Výsledok je {`"Hello":2, "World":2, "Bye":1, "Hadoop":2, "Goodbye":1`}, čo predstavuje náš očakávaný výsledok.



Obr. 2.2: MapReduce Verzia 1. Prevzaté z [6]

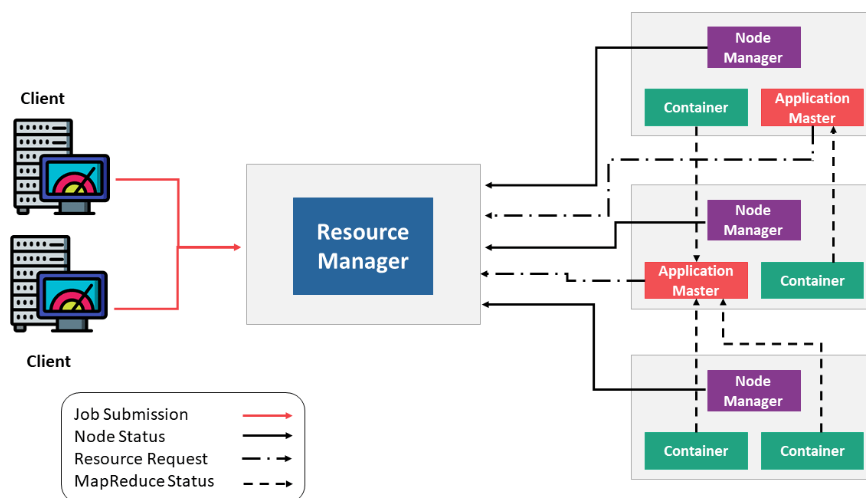
2.3.5 YARN - Yet another resource manager

S príchodom verzie Hadoop 2.0 bol upravený framework na spracovanie MapReduce programov. Novoupravený framework sa nazýva YARN. Dôvod vzniku YARN-u bol ten, že mechanizmus akým sa spracovávali tieto programy vo verzii Hadoop 1.0 obsahoval dizajnovú chybu. [6]

MapReduce v Hadoop 1.0 (označovaný aj MRV1 (MapReduce Version 1)) pozostával z dvoch komponentov: Job Tracker a Task Tracker. Klient pri zadaní novej úlohy (job) komunikoval s Job Trackerom. Ten následne zistil, na ktorých DataNodoch sa majú spustiť map, prípadne reduce fázy. Na týchto zvolených DataNodoch sa spustili procesy s názvom "Task Tracker", ktorých úlohou bolo reportovať informácie o priebehu daných jobov hlavnému Job Trackerovi. Je zrejmé, že pri veľkých clusteroch začalo vznikať úzke hrdlo na výpočtovom uzle, ktorý obsluhoval Job Tracker. Názorná ilustrácia je zobrazená na obrázku 2.2. Spoločnosť Yahoo! uvádzala, že prakticky bude výpočet brzdený týmto mechanizmom pri používani clusteru o veľkosti 5000 výpočtových uzlov, na ktorých je spustených 40 000 jobov paralelne. [6]

Obmedzenie spomínané v minulom odseku bolo vyriešné zavedením YARN-u v roku 2012 s príchodom Hadoop verzie 2.0. Úlohou YARNu bolo odbremeniť hlavný uzol, aby aj pri väčších clusteroch nenastával problém úzkeho hrdla. Tento problém bol vyriešený tak, že kompentencie Job Trackeru boli rozdelné do dvoch úrovní. Druhú úlohu, ktorú mal YARN poskytovať, bola podpora pre aplikácie, ktoré nevyužívajú len MapReduce (jedná sa u úlohy typu grafové spracovanie, interaktívne spracovanie, streamové spracovanie).

Mechanizmus fungovania YARN je znázornený na obrázku 2.3. Vysvetlenie



Obr. 2.3: Mechanizmus YARN-u. Prevzaté z [6]

jednotlivých komponentov je následovné:

- **Resource Manager** - komponent, s ktorým komunikuje klient, zodpovedá za prideľovanie zdrojov
- **Node Manager** - komponent, ktorý je zodpovedný za vykonávanie konkrétnej úlohy na jednom DataNode
- **Application Master** - spravuje životný cyklus aplikácie - komunikuje s Node Managerom, ktorý mu reportuje informácie o behu úlohy
- **Container** - Balíček, ktorý pozostáva zo zdrojov ako RAM, CPU, sieť, HDD . . .

Princíp fungovania YARN je následovný:

1. Klient zadá požiadavku na beh aplikácie (klient komunikuje s *Resource Manager-om*)
2. *Resource Manager* spracuje požiadavku a vytvorí *Application Master* na voľnom *Node Manager-y*
3. *Application Master* požiada *Resource Manager-a* o pridelenie zdrojov (a vytvorenie *container-ov*)
4. *Resource Manager* vytvorí nové *container-y* a túto informáciu odovzdá *Application Master*

5. O kontrolu jednotlivých *container-ov* sa stará výlučne *Application Master.*

    - v porovnaní s MRV1 možno vidieť ako sa rozloží záťaž a prečo nebude vznikať úzke hrdlo. V MRV1 sa o túto činnosť staral "*Resource Manager*".

2.3.6 Apache Spark

Autor v práci [7] detailne popisuje ako funguje Apache Spark, prečo vznikol a v čom sa odlišuje od klasického MapReduce modelu. Hlavným problémom klasického MapRedcuce bolo to, že všetky výsledky map fázy sa ukladali na disk, čo spôsobovalo značné spomalenie. V roku 2009 vznikol projekt Apache Spark, ktorý si stanovil cieľ vyriešiť:

- dáta budú ukladané buď v pamäti alebo na disku
- dáta v pamäti budú uložené ako Javovské štruktúry
- projekt Apache Spark bude obsahovať knižnice pre rôzne druhy výpočtov
- dáta budú rozdelené na menšie časti a tieto časti budú rozdistribuované

Samotný Apache Spark obsahuje submoduly: Submodul na všeobecné výpočty, Mllib (knižnica určená na strojové učenie), Streaming, SQL a GraphX (GraphX je framework ktorý disponuje knižnicami obsahujúcimi grafové algoritmy). V tejto práci sa ale využíva framework SparkGraphComputer, viď 3.1.17).

Spark je známy tým, že podporuje funkcionalné programovanie (jedná sa o podobný koncept, aký využívajú napríklad streamy v Jave 8). Hlavným rozdielom je to, že Spark pracuje nad dátami typu RDD ("resilient distributed dataset"). Jedná sa o datatyp, ktorý dokáže zaobaliť javovský objekt (napríklad kolekciu, načítanie súboru) a následne robiť výpočet paralelne. Objekt typu RDD dokáže byť rozdistribuovaný naprieč clusterom. [7]

Aplikáciu Spark možno spúšťať v troch režimoch: lokálne (je možné definovať, na koľkých vláknach má výpočet prebiehať), cez YARN, viď 2.3.5, prípadne cez Mesos.

Pri špecifikovaní novej úlohy je možné definovať podrobné špecifikácie (koľko pamäte môže využívať hlavný Spark proces (master), koľko má vytvoriť executorov (vlákna, ktoré budú vykonávať daný výpočet), koľko operačnej pamäti prideliť každému executorovi . . .) [7]

2.3.6.1 Cache

Dôležitý atribút je samotná cache a ako sa správať k dátam, ktoré sa nezmestia do pamäte. Existujú tieto základné modely [7]

1. `MEMORY_ONLY` - v JVM sú uložené deserializované objekty. V prípade, že sa nejaký objekt nezmestí do pamäte, staré objekty budú z pamäte vyhodené (ak budú potrebné v budúcnosti, znova sa vytvoria)
2. `MEMORY_AND_DISK` - rovnaké správanie ako v 1 s tým rozdielom, že staré objekty nebudú vyhadzované z pamäte, ale ukladané na disk
3. `MEMORY_ONLY_SER` - rovnaké správanie ako v 1 s rozdielom, že v pamäti sú namiesto deserializovaných objektov, objekty serializované
4. `MEMORY_AND_DISK_SER` - rovnaké správanie ako v 2 s tým rozdielom, že ukladané dáta sú uložené v serializovanej forme
5. `DISK_ONLY` - Dáta sú ukladané len na disk

2.4 CAP teorém

CAP teorém uvádza, že je nemožné pre distribuované úložisko poskytnúť súčasne C,A,P pričom význam jednotlivých písmen je:

- **C - Consitency (konzistencia)** - Systém je vždy konzistentný
- **A - Availability (dostupnosť)** - Systém je vždy dostupný
- **P - Partition tolerance (odolnosť k prerušeniu)** - Systém dokáže fungovať aj v prípade, že sa cluster rozdelí na 2 partície.

2.4.1 Dôkaz

Dôvod, prečo nie je možné súčasne poskytovať všetky 3 požiadavky je možné znázorniť na tomto príklade. Predpokladajme, že máme rozdistrubuovanú databázu na dvoch centrách (`centrum A` a `centrum B`). Medzi týmito centrami existuje prepojenie. Databáza je v konzistentnom stave a obsahuje záznam o počte dostupných produktov (táto informácia je rovnaká pre `centrum A` aj pre `centrum B`). V čase `t` dôjde k rozpojeniu spojenia. Klient komunikuje s `centrum B` a objedná si produkt. Počet dostupných produktov teda klesne o 1. V čase `t+1` sa pripojí nový klient na `centrum A` a zadá požiadavku na počet produktov. `centrum A` vie, že došlo k strate spojenia, tým padom vie, že môže existovať situácia, v ktorej došlo k zmene počtu produktov, a tak nevie zaručiť konzistentnú odpoveď klientovi. To, ako `centrum A` zareaguje definuje, ktorá podmienka bude porušená.

- ak odpovie informáciu, ktorou disponuje, zachová síce **dostupnosť** no poruší **konzistenciu**
- ak neodpovie klientovi nič, pretože mu nevie zaručiť **konzistenciu** tak poruší **dostupnosť**

- ak by sytém predišiel tejto situácií a v momente rozpadu siete by neodpovedal, tak by porušil **odolnosť k prerušeniu**

Kapitola 3

Rešerš grafových databáz pre veľké grafy

3.1 Vymedzenie pojmov

3.1.1 Graf

Graf je základným objektom teórie grafov. Samotný graf pozostáva z prvkov, ktoré môžu byť vzájomne prepojené. Jednotlivé prvky sa nazývajú vrcholy (anglicky Vertex - častokrát značené V), vzájomné väzby sa nazývajú hrany (anglicky Edge - značené E).

3.1.1.1 Uzol

V grafových databázach možno jednotlivé uzly zaradiť do kategórie (napríklad Osoba). Uzly môžu obsahovať aj rôzne atribúty (napríklad uzol typu Osoba môže niesť informáciu o mene, priezvisku, ...).

3.1.1.2 Hrana

Vlasnosti hrán možno popísať následovne:

- jednotlivé hrany možno zaradiť do kategórie rovnako ako uzly. Medzi dvoma uzlami typu PERSON môže existovať väzba typu PAYMENT, ktorej význam možno definovať ako vykonanú platbu medzi dvoma osobami. Medzi dvoma uzlami typu PERSON môže existovať súbežne aj iný typ väzby, ktorá predstavuje iný význam, napr. väzba IS_FRIEND, ktorá má iný význam, tj či daná osoba pozná druhú osobu
- orientovanosť - hrany môžu byť orientované alebo neorientované. Orientovanú hranu si možno predstaviť ako šípku so smerom, tj je možné odpovedať na otázku z akého uzlu vychádza a do akého uzlu hrana smeruje. Príklad - väzba typu PAYMENT spojuje dva uzly typu PERSON.

Vďaka tomu, že hrana môže byť orientovaná, je možné odpoveďať na otázku kto zaplatil komu

- atribúty - v niektorých grafových databázach možno hranám priradzovať parametre. Hrana typu PAYMENT môže obsahovať informácie o tom, kedy bola platba vykonaná, o akú veľkú sumu sa jednalo. . . .

3.1.2 Horizontálne škálovanie

Pojem horizontálne škálovanie predstavuje pridanie nových výpočtových prvkov. Samotný výpočet sa rozdistribuuje medzi viacero výpočtových uzlov. Tento proces je cenovo efektívnejší ako vertikálne škálovanie. Hlavnou výhodou je to, že s narastajúcim objemom dát je možné pridať ďalšie výpočtové uzly.

3.1.3 Vertikálne škálovanie

Pojem vertikálne škálovanie v oblasti big data možno definovať ako vylepšenie jedného výpočtového uzla, na ktorom bude prebiehať výpočet. V praxi to znamená, že sa urobí vylepšenie RAM, CPU, diskov, prípadne grafickej karty. Vertikálne škálovanie je cenovo nákladnejšie ako horizontálne škálovanie. Pri veľkých dátach je častokrát použitie vertikálneho škálovania nedostačujúce, nakoľko existujujú obmedzenia, ktoré nám neumožňujú vylepšovať počítač donekonečna.

3.1.4 PageRank

Algoritmus PageRank vznikol behom vysokoškolského štúdia Larryho Pagea (spoluzakladateľ Google) a jedná sa o paralelu k citačnému indexu. T.j. čím viac akademických práci (webových stránok) odkazuje na prácu A (webovú stránku A'), tým je práca A (webová stránka A') dôležitejšia. [8] Výsledkom algortimu Page Rank je ohodnotenie jednotlivých stránok, pričom hodnota sa pohybuje v rozmedzí [0,1].

3.1.5 Personalizovaný PageRank

PageRank udáva v podstate pravdepodobnosť, že sa človek ocitne na stránke (uzle) X. Jedná sa o iteratívny algoritmus, ktorý v prvej iterácií všetkým uzlom priradí rovnakú pravdepodbnosť, ktorá sa v priebehu ďalších iterácií v závislosti od jednotlivých väzieb mení. Personalizovaný PageRank je ale taký, v ktorom iniciálna pravdepodobnosť návštevy uzlov nie je rovnaká pre všetky uzly.

3.1.6 Fork

Anglické slovo fork možno preložit ako rázcestie. Jedná sa o kópiu repozitára. Fork repozitára užívateľovi umožňuje voľne experimentovať s vlastnými zmenami bez toho, aby ovplyvnil pôvodny projekt. [9]

Častokrát sa fork používa aj na návrh zmien pre iné projekty ako vlastné. Ďalším použitím je použitie iného projektu ako štartovacieho bodu pre vlastnú myšlienku. [9]

3.1.7 Počítačový cluster

Počítačový cluster je zoskupenie viacerých počítačov, ktoré spolu úzko spolupracujú, tým pádom sa navonok tvária ako jeden počítač. Cluster poskytuje služby ako paralelné spracovávanie, rozloženie záťaže, dostupnosť ... [10]

3.1.8 Transakčná databáza

Transakčná databáza je taký typ databázy, ktorý dokáže spracovávať transakcie, viď 3.1.8.1. Transakcia sa buď vykoná celá alebo sa nevykoná vôbec. V prípade, že počas jedného z príkazov dôjde k chybe, vrátia sa všetky vykonané zmeny do stavu na začiatku transakcie.

3.1.8.1 Databázová transakcia

Databázová transakcia je skupina príkazov, ktoré prevedú databázu z jedného konzistentného stavu do druhého.

3.1.8.2 HBase

Apache HBase je distribuovaná, škálovateľná, NoSQL databáza určená na ukladanie big data, ktorá beží nad Hadoopom. HBase dokáže uložiť miliardy riadkov, pričom každý riadok môže obsahovať milióny stĺpcov. Táto databáza umožňuje čítanie a zapisovanie v reálnom čase. Databáza patrí do rodiny stĺpcových databáz (v porovnaní s relačnou databázou, dáta jedného riadku nemusia byť nutne uložené sekvenčne za sebou.) [11] Jedná sa o databázu, ktorá splňuje CA z CAP teorému, viď 2.4.

Tým, že patrí do rodiny stĺpcových databáz, je vhodným kandidátom na ukladanie grafovej štruktúry pre JanusGraph, prípadne Titan, nakoľko tento koncept umožňuje ukladať pre rôzne riadky rôzne stĺpce (v porovnaní s relačnou databázou kde takéto správanie nie je možné). Vzhľadom k tomu, že JanusGraph ukladá dáta pre každý uzol v zozname (každý atribút, ako aj prichádzajúca, alebo odchádzajúca hrana predstavuje nový stĺpec).

Článok [12] názorne vysvetľuje, ako interne funguje databáza HBase. H-Base databáza je založená na master/slave architektúre. Master Server sa

nazýva HMaster, slave - RegionServer. Samotny HBase využíva taktiež Zookeeper (služba, ktorá je zodpovedaná za notifikovanie ohľadom životnosti jednotlivých serverov - odpovedá na otázku, či je server funkčný, alebo je mŕtvy).

HMaster Server - Jedná sa o server, ktorý je zodpovedný za exekúciu DDL príkazov (vytvorenie, zmazanie tabuľky). Je zároveň zodpovedný aj za priraďovanie regiónov pri štarte databázy, rieší problémy spojené s obnovou dát alebo rozložením záťaže. Prostredníctvom Zookeepera udržuje informácie o tom, ktoré RegionServery sú aktívne. [12]

RegionServer - Jedná sa o servery, ktoré obsahujú samotné dáta a ktoré s nimi pracujú. V prípade, že klient pracuje s dátami, tak posiela požiadavky priamo RegionServeru. Tým pádom nevzniká úzke hrdlo (nie je potrebné, aby všetka komunikácia prechádzala cez HMaster Server). To, kam (do akého RegionServeru) sa dáta uložia, definuje index riadku. Každý región má definované rozpätie indexov, ktoré vlastní. RegionServer dokáže obsahovať až 1000 regiónov. Štruktúra RegionServeru je následovná. [12]

- **WAL - Write Ahead Log** - pri zápise nových dát sú dáta najprv uložené do toho segmentu (aby bolo možné rekonštruovať dáta v prípade náhlého výpadku systému)
- **BlockCache** - jedná sa o čítaciu cache. Nachádzajú sa v nej dáta, ktoré sú často čítané. V prípade, že je cache plná, odstraňujú sa najstaršie dáta.
- **MemStore** - jedná sa o zapisovaciu cache. Ukladá nové dáta, ktoré ešte neboli zapísané na disk. Pred zápisom dáta zoradí a následne zapíše. Každý región môže obsahovať viac týchto MemStore
- **Hfiles** - samotné dáta sú uložené v Hfiles súboroch v HDFS systéme, viď 2.3.1. Tieto Hfiles sú následne replikované naprieč clusterom.

Priebeh spojenia - HBase obsahuje špeciálnu tabuľku, ktorá udržuje informácie o RegionServeroch (a to, akými dátami disponuje). Tabuľka sa nazýva .META a je uložená v Zookeeperi. Klient sa pripojí do .META tabuľky, stiahne si túto tabuľku a uloží ju do cache. Následne zistí, na aký RegionServer sa má pripojiť. Pri druhom dotaze už nepotrebuje kontaktovať Zookeeper, pretože môže využiť cache. Ak sa pripojí na RegionServer a zistí, že sa tam daný záznam nenachádza (došlo k zmazaniu, prípadne presunu), vymaže cache a znovu kontaktuje Zookeeper. [12]

3.1.8.3 Cassandra

Podobne ako HBase, aj Cassandra sa radí medzi distribuované a škálovateľné databázy. Táto databáza nebude rozoberaná tak podrobne ako HBase, vzhľadom k tomu, že pre účely tejto diplomovej práce táto databáza nebola využívaná. Táto kapitola sa zamierava predovšetkým na rozdiely medzi HBase a Cassandrou. Článok [13] sa zaoberal týmto porovnaním. Informácie čerpané v tejto kapitole vychádzajú zo spomínaného zdroja.

Jednotlivé databázy sá odlišujú v tom, ako dané dáta ukladajú. Dôležité je uviesť, aké závery z toho plynú a čím sa z pohľadu výkonnosti odlišujú. Cassandra v porovnaní s HBase využíva masterless architektúru. Z toho dôvodu sa Cassandra radí k menej rizikovým riešeniam, nakoľko by HBase pri strate HMaster servera prišiel o všetky dáta (anglicky známe pod pojmom "single point of failure").

Cassandra zaručuje vysokú dostupnosť tým, že replikuje a duplikuje dáta, čo môže viesť ku problémom z pohľadu konzistencie dát. V porovnaní s HBase, táto databáza splňuje AP z CAP teorému, viď 2.4. HBase využíva na ukladanie dát HDFS, zatiaľ čo Cassandra využíva svoj vlastný mechanizmus.

Rýchlosť zápisu - Vzhľadom k tomu, že užívateľ potrebuje najprv komunikovať so Zookeeper-om aby získal informácie o regiónoch, viď 3.1.8.2, je zápis v Cassandre rýchlejší, nakoľko tento krok nie je v Cassandre nutné vykonať. Rozdiely ale nie sú markantné. Bol vykonaný benchmark na 32 uzlovom clusteri, Cassandra dokázala spracovať 326 500 transakcií za sekundu, zatiaľ čo HBase 297 000.

Rýchlosť čítania - Vzhľadom k tomu, že HBase ukladá dáta stále na konkrétne vybraný RegionServer, nie je nutné pre načítanie dát prechádzať viac dátových zdrojov. Cassandra na druhú stranu umožňuje ukladať dáta na rôzne servery, teda pri dotaze na čítanie potrebuje zistiť, ktorý údaj je najaktuálnejší. Z toho vyplýva, že čítanie zvláda rýchlejšie HBase.

3.1.8.4 BerkeleyDB

Pôvodná verzia bola vydaná v roku 1994 na univerzite v Berkley, pričom bola implementovaná v jazyku C. V súčasnosti už BerkeleyDB patrí pod Oracle. BerkeleyDB je dostupná v troch rôznych verziách - pôvodná, verzia v jave a Oracle Berkeley DB XML, ktorá je určená pre prácu s XML dokumentami. Narozdiel od Cassandry a HBase je táto databáza zdarma len pre nekomerčné použitie. Autor ďalej uvádza, že pre komerčné účely je databáza spoplatnená čiastkou 26.000 USD/procesor. [14]

V porovnaní s predošlými dvoma kandidátmi sa jedná o databázu, ktorá nedokáže operovať v distribuovanom režime, tým pádom tento databázový backend nebol ďalej zvažovaný.

3.1.9 ACID

Databáza, ktorá spĺňa ACID vlastnosti je taká, v ktorej platia tieto body:

- **Atomicity / Atomicita** - Databázová transakcia je ako operácia nedeliteľná. Buď sa vykoná celá, alebo sa nevykoná vôbec
- **Consitency / Konzistencia** - Pred a po uskutočnení transakcie je databáza stále v konzistentom stave (nie je porušené žiadne integritné obmedzenie)
- **Isolation / Izolácia** - Operácie v rámci jednej transakcie sú z pohľadu iných transakcií skryté.
- **Durability / Trvalosť** - Zmeny, ktoré boli vykonané (napríklad akciou commit) sú trvalo uložené a nemôžu byť vrátené.

3.1.10 NoSQL

Skratka NoSQL znamená "non relational and not SQL". Do tejto skupiny patria novovzniknuté databázové systémy, ktoré fungujú na inom princípe ako klasické relačné databázy. Dôvod vzniku týchto databáz bol ten, že existujú situácie, v ktorých je výhodnejšie použiť iný typ databázy. Jedným z hlavných dôvodov bol vznik big data, čo viedlo k potrebe vedieť horizontálne škálovať databázy (čo pri relačných modeloch nie je úplne jednoduché). Medzi zástupcov NoSQL možno zaradiť:

- Databázy kľúč-hodnota - jedná sa o jednoduchú databázu, ktorá obsahuje index nad kľúčom, pričom hodnotou môže byť čokoľvek (možno prirovnať k paralele s hashovacou mapou)
- Dokumentové databázy - podobný princíp ako kľúč-hodnota s tým rozdielom, že objekty uložené v tejto databáze majú štruktúru (napr MongoDB ukladá jednotlivé riadky v JSON formáte)
- Stĺpcové databázy - dáta nie sú ukladané sekvenčne po riadkoch, ale po stĺpcoch, napríklad Cassandra, viď 3.1.8.3 alebo HBase, viď 3.1.8.2
- Grafové databázy - dáta sú uložené vo forme grafu

Niektorí autori uvádzajú skratku aj v spojení "not only SQL".

3.1.11 CRUD

CRUD sú štyri základné operácie nad dátovým úložiskom. Jedná sa o tieto operácie:

- **Create / Vytvoriť** - vytvorí nový záznam

- **Read / Čítať** - prečíta záznam
- **Update / Aktualizovať** - aktualizuje záznam
- **Delete / Zmazať** - zmaže záznam

3.1.12 Apache Giraph

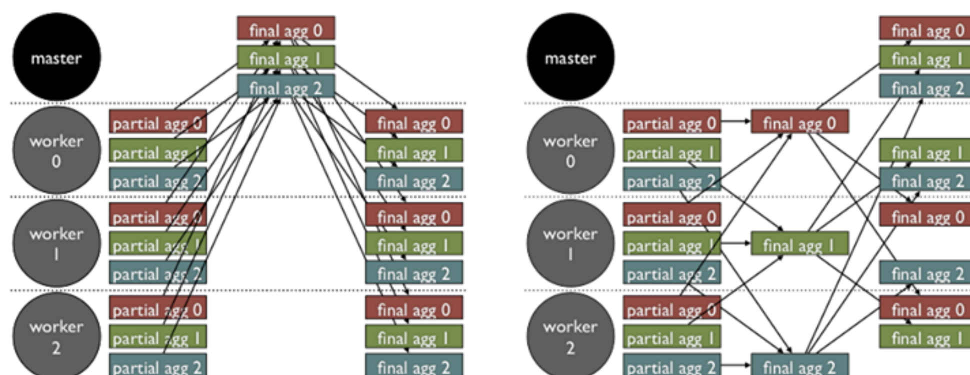
Apache Giraph je opensourcový projekt založený na rovnakom princípe, na akom je postavený framework Pregel, viď 3.3.3. Apache Giraph využíva k behu komponent Apache Hadoop-u, viď 2.3. Tento vznikol v roku 2012, pričom na vývoji sa podieľali spoločnosti ako Facebook, Twitter, Linkedin.

Apache Hadoop disponoval frameworkom MapReduce, viď 2.3.4, preto vzniká otázka, prečo neyvužiť tento princíp k implementovaniu Pregel frameworku. Hlavným problémom bolo to, že MapReduce ukladá všetky medzivýpočty do HDFS, tým pádom je potreba veľkého množstva diskových operácií, čo môže značne spomaliť výpočet.

Apache Giraph beží defaultne v takzvanom režime ïn-memory", čo znamená, že všetky dáta sú uložene v hlavnej pamäti.

Samotný beh aplikácie pozostáva z týchto 4 fáz [15]:

1. **Načítanie**
   - načítanie dát z disku - možno využiť rôzne konektory na HBase viď 3.1.8.2, Hive, HDFS viď 2.3.1
   - vykoná sa mapovanie uzlov na výpočtové uzly - worker nody (pre lepšie pochopenie, viď 3.3.3, kde je popísaný koncept "mysli ako uzol")
2. **Výpočet**
   - jednotlivým uzlom sa priradia správy z minulého *superstep-u*
   - vykoná sa iterácia naprieč aktívnymi uzlami a nad nimi sa zavolá užívateľom definovaná funkcia, viď 3.1.13
3. **Synchronizácia**
   - dôjde k výmene správ medzi výpočtovými uzlami
   - dôjde k agregácií potrebných položiek
   - vytvorí sa checkpoint
   - Ak ešte existujú aktívne uzly, tak z tejto fázy následuje skok do **Výpočet** fázy, inak do **Konečnej fázy**
4. **Konečná fáza / fáza zápisu** - dôjde k zápisu potrebných dát, viď 3.1.13



Obr. 3.1: Apache Giraph - Sharded Aggregator. Prevzaté z [16]

V článku [16] je popísané, akým spôsobom využíval Facebook tento framework, s akými problémami bojovali a ako ich vyriešili. Konkrétne sa venovali problematike s pamäťou.

Jednalo sa o problematiku, v ktorej často dochádzalo k chybám Out of Memory (nedostatok pamäti), čo bolo spôsobené tým, že všetky dáta boli uložené ako separátne Javovské objekty, prípadne k výkonnostným problémom, v ktorých veľkú časť výpočtu spotreboval samotný Garbage Collector (algortimus, ktorý určuje, ktoré bloky pamäte už nie sú odkazované v rámci aplikácie a či je bezpečné toto miesto dealokovať).

Ďalší problém, ktorý bol v prvej verzii Apache Giraph bol ten, že dáta, ktoré boli agregované a následne zdieľané všetkým uzlom boli agregované na *Master* uzle, tým pádom vznikalo úzke hrdlo, pretože *Master* musel v každom kroku prijať enormné množstvo dát, spracovať ho a následne túto informáciu odovzdať *Worker* uzlom. Facebook tento problém vyriešil takzvanými "Sharded aggregators".

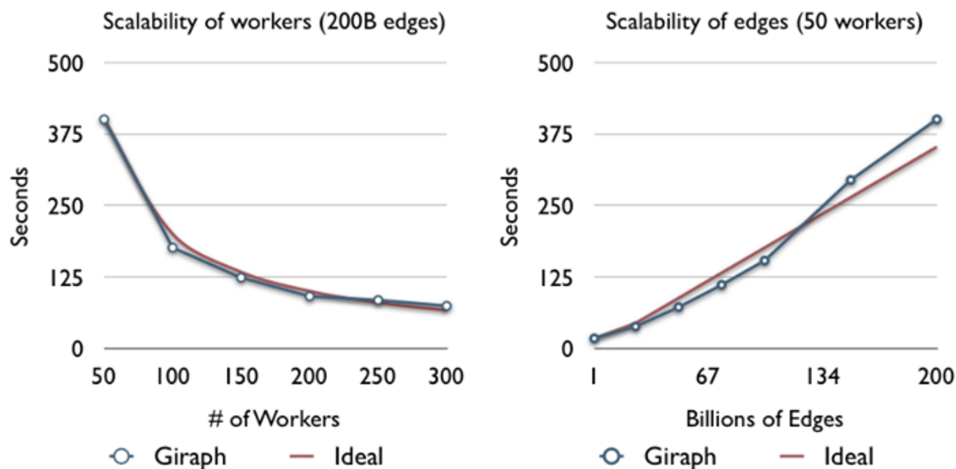
Sharded aggregators sú agregátory, ktoré sa nenachádzajú na *Master* uzle ale na *Worker* uzle. Každá premenná, ktorá sa agreguje, sa môže nachádzať na inom výpočtovom uzle, čím sa rozloží záťaž. Ideu názorne zobrazuje obrázok, viď 3.1. [16]

Ako je viditeľné na obrázku, viď 3.2, tento framework je takmer lineárne škálovateľný vzhľadom k počtu výpočtových uzlov.

3.1.13 Apache Giraph - API

Pre spustenie výpočtu je potrebné implementovať 3 Javovské triedy. Prvou z nich je implementácia výpočtového modulu. Jedná sa o logiku, ktorú bude uzol v každom kroku vykonávať (ako naložiť s prijatými správami, aké správy následne poslať susedom ...). [15]

Druhou triedou, ktorú je potrebné implementovať, je trieda, ktorá definuje, ktoré dáta predstavujú uzol a ktoré dáta predstavujú hrany medzi týmito



Obr. 3.2: Apache Giraph - Škálovateľnosť z pohľadu počtu výpočtových uzlov. Prevzaté z [16]

uzlami. [15] V prípade, že vstupné dáta sú evidované vo forme relačnej databázy, väzby sú typicky definované prostredníctvom tabuľky, ktorá obsahuje aspoň dva stĺpce, napríklad zdrojový uzol, ktorého hodnota je cudzí kľúč nejakej tabuľky a cieľový uzol, ktorého hodnota je cudzí kľúč rovnakej alebo inej tabuľky. Táto trieda bude obsahovať informáciu o tom, ktoré atribúty možno považovať za identifikáciu zdrojového uzla, prípadne cieľového uzla, kde sú dáta, ktoré možno spájať s jednotlivými uzlami. Pri načítávaní z JanusGraph za využitia OLAP, viď 3.1.16 je samotná databáza navrhnutá tak, aby dokázala tomuto frameworku ponúknuť potrebné dáta rýchlo a efektívne. V porovnaní s iným riešením môže predstavovať samotné načítavanie istú réžiu - napríklad vo forme netriviálnych join operácií.

Treťou triedou je trieda, ktorá obsahuje informáciu o tom, ako s výsledkom naložiť. Jedná sa o informáciu, ktoré dáta zapísať, kam ich zapísať a v akom formáte. Ako príklad možno uviesť výpočet PageRank, viď 3.1.4 nad grafom. V prípade, že je žiadúce odpovedať na otázku aký PageRank má aký uzol, táto trieda bude definovať rozhranie výsledného súboru napríklad vo formáte `vertexID`, `pageRankScore`. [15]

Poslednou triedou, ktorá je voliteľná, je takzvaný Combiner. Jedná sa o triedu, ktorá dokáže istým spôsobom agregovať prichádzajúce správy pre konkrétny uzol. V prípade implementácie PageRank algoritmu nepotrebuje každý uzol dostať informáciu o novom prírastku, stačí mu súčet týchto hodnôt. Ak je možné využít Combiner, je možné ušetriť značné množstvo sieťového prenosu, ktorý predstavuje výrazné spomalenie celého výpočtu. [15]

3.1.14 Quorum

Pri práci s distribuovaným úložiskom je potrebné myslieť na to, že s pribúdajúcimi dátovými uzlami sa zvyšuje pravdepodnosť, že nastane situácia kedy sa nejaký uzol pokazí. Jedným z riešení je zaviesť do systému princíp replikácie dát, čo znamená, že dáta sú uložené reduntantne na viacerých uzloch. Ak dôjde k situácií, že jeden z dátových uzlov vypadne, nedôjde k strate dát, pretože tieto dáta boli replikované. Nech N definuje replikačný faktor (číslo udáva, na koľkých uzloch budú reduntantne uložené dáta).

Predpokladajme, že klient chce zadať požiadavku na vytvorenie nových dát. Vzniká otázka, kedy mu má server vrátiť odpoveď "dáta boli v poriadku uložené". Hodnota parametru *Quorum zápisu* udáva presne túto hodnotu. Podobne ako pri zápise je to aj s čítaním.

Ak klient zadá požiadavku na čítanie, vzniká otázka, koľko replík je potrebné kontaktovať, aby server mohol užívateľovi vrátiť odpoveď. Túto hodnotu definuje parameter *Quorum čítania*.

V rámci tejto kapitoly bude *Quorum zápisu* značené ako W a *Quorum čítania* R. K tomu, aby bola zaručená konzistencia, je potrebné spĺňať podmienky 3.1 a 3.2.

$$R + W > N \tag{3.1}$$

$$W > N/2 \tag{3.2}$$

3.1.14.1 Význam jednotlivých rovníc

Vzťah 3.1 možno vysvetliť na názornom príklade. Predpokladajme situáciu, v ktorej platí následové: $N = 5$, $W = 3$, $R = 3$. Jednotlivé uzly, na ktorých budú dáta uložené si označme `A`, `B`, `C`, `D`, `E`. Pri zápise nových dát sa dáta replikujú na uzly `A`, `B`, `C` (tým, že je parameter $W = 3$ je garantované, že sa dáta nachádzajú aspoň na 3 uzloch). Pri následnom čítani dát uvažujme situáciu, v ktorej by boli kontaktované repliky `D`, `E`, `C`. Tým, že platí vzťah 3.1 je garantované, že pri čítaní bude zaručene kontaktovaná replika, ktorá obsahuje aktuálne dáta. Ak by platilo, že $R = 2$, tj spomínaná podmienka by nebola platná a mohlo by dôjsť k situácií, v ktorej by boli načítané neaktuálne (nekonzistentné) dáta, boli by kontaktované len repliky `D`, `E`.

Vzťah 3.2 je dôležitý kvôli tomu, aby v prípade detekovania nekonzistencie bolo možné rekonšturovať skutočné dáta (na základe hodnoty, ktorú vlastní väčšina replík). V prípade, ak by platilo, že $W = \lfloor N/2 \rfloor$, staré dáta získajú väčšinový hlas, tým pádom užívateľ dostane nekonzistentné dáta.

3.1.15 TinkerPop

Autor v práci [17, str. 15] popisuje históriu vzniku TinkerPopu a vysvetľuje, aké komponenty obsahuje. TinkerPop je opensourceový framework, určený na

Obr. 3.3: TinkerPop2 - jednotlivé moduly. Prevzaté z [18]

prácu s grafovými databázami. Framework bol pôvodne vyvíjaný ako nástroj pre Projekt Titan, viď 3.3.4, ale v súčasnosti je využívaný rôznymi databázami.

Prvá verzia vyšla v roku 2009, označovaná pod názvom TinkerPop0. Verzie 1, 2 vznikli v rokoch 2011, respektíve 2012. V roku 2015 prešiel projekt TinkerPop pod Apache Software Foundation a rok na to bola vydaná nová verzia číslo 3. Nová verzia 3 predstavovala výraznejšie zmeny, ktoré spôsobili to, že niektoré fragmenty implementované v staršej verzii neboli kompatibilné. [17, str. 15]

Práca [17, str. 15] pekne vysvetľuje jednotlivé moduly, viď 3.1.15.1, z ktorých pozostával TinkerPop2 ako aj mapovaciu tabuľku, viď 3.1, ktorá zachycuje to, ako sa jednotlivé moduly premenovali, prípadne rozdelili v novovydanej verzii 3. Dôvod týchto zmien bol hlavne ten, aby bolo nové rozhranie jednotnejšie a viac odpovedalo konceptom Gremlin jazyka, viď 3.1.15.2. Situáciu názorne znázorňujú obrázky 3.3 a 3.4, čo predstavuje spojenie jednotlivých modulov do jedného celku, obrazne definovaného ako ”Gremlintron”.

3.1.15.1 Moduly TinkerPopu

TinkerPop pozostával z týchto modulov:

- **Blueprints** - Jedná sa o kolekciu rozhraní pre grafové modely dát. Tento modul možno prirovnať k JDBC pre grafové databázy. Program

Obr. 3.4: TinkerPop3 - previazané moduly. Prevzaté z [18]

napísaný v tomto module je kompatibilný so všetkými grafovými databázami, ktoré podporujú TinkerPop. [17, str. 15]

- **Rexter** - Jedná sa o grafový server, ktorý vystavuje grafy vytvorené za pomoci modulu Blueprints cez HTTP cez RESTové rozhranie. [17, str. 16]
- **Furnace** - Balíček grafových algoritmov pre Blueprints grafy. Algoritmy je možné spúštať v distribuovanom režime, viď 3.1.16. [17, str. 16]
- **Frames** - Modul, ktorý je zodpovedný za mapovanie dátových objektov vytvorených cez Blueprints modul. V praxi to znamená, že programátor môže pracovať s objektami, ktoré predstavujú jednotlivé uzly, vlastnosti týchto uzlov. V podstate sa jedná o objektové mapovanie. [17, str. 16]
- **Gremlin** - Gremlin je funkcionálny jazyk, vytvorený pre prácu s grafovými databázami. [17, str. 16]
- **Pipes** - Modul, ktorý je zodpovedný za skladanie jednotlivých krokov prechodu grafu. Tieto *Pipes* je možno reťaziť. [17, str. 16]

Tabuľka 3.1: Rozdiely medzi TinkerPop2 a TinkerPop3. Prevzaté z [17, str. 117])

TinkerPop2	**TinkerPop3**
Rexter	GremlinServer
Furnace	GraphComputer, viď 3.1.17, VertexProgram
Frames	Traversal
Gremlin	Gremlin
Pipies	GraphTraversal
BluePrints	Gremlin Structure API

3.1.15.2 Gremlin

Vo svete relačných databáz je používaný jazyk SQL, pri grafových databázach sa o to snaží Gremlin. Pri dotazovaní nad grafovou databázou je potrebné nad dotazmi rozmýšľať iným spôsobom ako v relačnom svete.

Spôsob ako definovať dotazovanie v grafových databázach pozostáva z prechádzania grafu (anglicky známe pod pojmom *Traversal*). Jazyk Gremlin je nástroj, ktorým je možné túto podstatu zachytiť. Gremlin možno popísať ako funkcionálny jazyk, v ktorom je možné aj zložité *traversal*-y jednoducho vyjadriť.

Traversal v Gremline začína stále z objektu s názvom `GraphTraversalSource`. Z tohoto objektu je možné definovať začiatok buďto cez dané väzby (definované konkrétným ID), alebo cez dané uzly (rovnako ako pri hranách je možnosť ich explicitne vymenovať prostredníctvom ID). Syntax pre uzly, respektíve hrany je `GraphTraversalSource.V(Object... ids)`, `GraphTraversalSource.E(Object... ids)`. Obe tieto metódy vrátia objekt typu `GraphTraversal`. Nasledovný prechod grafom pozostáva z jednotlivých krokov, pričom každý krok spadá do jednej z týchto kategórií: [18]

- **map** - transformácia z prichádzajúceho stavu do nového stavu ($S \rightarrow E$)
- **flatMap** - transformácia z prichádzajúceho stavu do nových stavov (týchto stavov môže byť viac) ($S \rightarrow E^*$)
- **filter** - funkcia, ktorá definuje, či sa v danom stave má pokračovať do ďalšieho kroku alebo tento konkrétny stav má byť ukončený ($S \rightarrow E \subseteq S$)
- **sideEffect** - jedná sa o funkciu, ktorá dokáže vykonať nad daným stavom istú operáciu, pričom nezmení aktuálny stav priechodu ($S \rightarrow S$)
- **branch** - rozdelenie stavu do nových stavov v závislosti od hodnoty (napríklad sa jedná o príkaz choose, ktorý možno definovať ku `IF`, `ELSE` vetve) ($S \rightarrow \{S_1 \rightarrow E^*, \ldots, S_n \rightarrow E^*\} \rightarrow E^*$)

Názorná ukážka Pre lepšie pochopenie uvažujme graf, ktorý je znázornený na obrázku 3.5. Z grafu chceme zistiť, aké osoby pozná osoba (uzol), ktorá ma meno "marko". [18] Nižšie spomínaný fragment kódu je prevzatý z [18].

```
$ bin/gremlin.sh
gremlin> graph = TinkerFactory.createModern() //1
==>tinkergraph[vertices:6 edges:6]
gremlin> g = graph.traversal()          //2
==>graphtraversalsource[...]
gremlin> g.V().has('name','marko').\
         out('knows').values('name') //3
==>vadas
==>josh
```

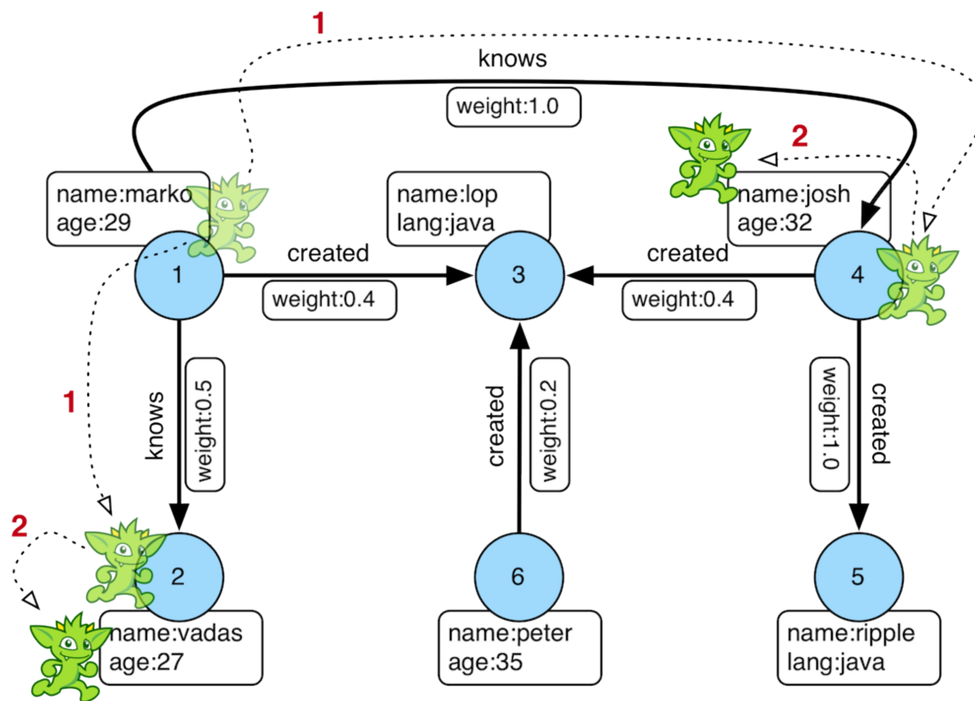
1. Vytvorí graf, ktorý obsahuje dáta, ktoré sú viditeľné na obrázku 3.5
2. vytvorí iniciálny objekt, z ktorého môže začať prechádzanie grafom, ide o situáciu OLTP, pozri 3.1.16
3. príkaz, ktorého výsledkom sú mená osôb, ktoré pozná (knows) marko
   - `g.V()` - vráti všetky uzly
   - `has('name','marko')` - vykonanie *filter* fázy, z grafu je viditeľné, že touto fázou prejde len jeden uzol
   - `out('knows')` - vykonanie *filter* a *flatMap* fázy, výsledkom tejto operácie sú dva uzly
   - `values('name')` - vykonanie map fázy, nakoľko sa uzly premapujú na hodnotu parametru `name`

3.1.16 OLAP vs OLTP

Framework Tinkerpop poskytuje dva prostriedky pre komunikáciu s grafom:

- **OLTP** - online transakčné spracovanie (online transaction processing)
- **OLAP** - online analytické spracovanie (online analytical processing)

OTLP spracovanie je vhodné pre dotazy, ktoré pozostávajú z takých *traversal*-ov, ktoré pracujú s malou podmnožinou grafov. Výsledky týchto dotazov sú typicky zodpovedané v radoch milisekúnd, prípadne sekúnd. Na druhej strane OLAP dotazy sú určené na zložitejšie dotazy (z pohľadu časovej náročnosti). Jedná sa o dotazy, ktoré potrebujú interagovať so všetkými uzlami, všetkými hranami, dokonca prechádzať týmito prvkami opakovane. Príkladom môžu byť dotazy, ktoré majú určiť PageRank jednotlivých uzlov,



Obr. 3.5: Ukážka prechádzania grafom v Gremline. Prevzaté z [18]

prípadne clusterovacie algoritmy. Vzhľadom k obrovskému množstvu spracovávaných dát sú výsledky týchto dotazov prístupné v radoch minút, hodín. [18]

Princíp, na akom funguje OLTP je výrazne odlišný od OLAP spracovania. Pri Gremlin OLTP dotaze je graf prechádzaný postupne z uzlu na uzol cez hrany. Gremlin OLAP využíva takzvané `VertexProgram`-y, ktoré obsahujú logiku pre každý jeden uzol (veľmi podobný princíp ako bol popísaný v Apache Giraph, viď 3.1.12, konkrétne prvá fáza popísaná v 3.1.13). OLAP spracovanie je možné zakončiť MapReduce fázou, ktorá dokáže agregovať potrebné výsledky. [18]

TinkerPop v sebe obsahuje radu algoritmov, ktoré je možné spustiť v OLAP režime, tj obsahuje implementáciu `VertexProgram`-ov.

Framework TinkerPop disponuje aj špeciálnym `TraversalVertexProgram`, ktorý slúži na simulovanie klasických OLTP dotazov, vykonaných v OLAP režime. Princíp je založený na tom, že jednotlivý *traversal* sa spustí na každom uzle. Situáciu si možno predstaviť tak, že obsah `VertexProgramu` je samotný *traversal*. [18]

Názorná ukážka V kapitole 3.1.15.2 bola ukážka jednoduchého OTLP dotazu. V tejto sekcii je znázornená ukážka OLAP dotazu, konkrétne dotazu na

výpočet PageRank-u. Nižšie spomínaný fragment kódu je prevzatý z [18].

```
gremlin> result = graph.compute()
                .program(PageRankVertexProgram.build()
                .create()).submit().get()
=>result[tinkergraph[vertices:6 edges:0],memory[size:0]]
gremlin> result.memory().runtime
=>23
gremlin> g = result.graph().traversal()
=>graphtraversalsource[tinkergraph[vertices:6 edges:0],
 standard]
gremlin> g.V().valueMap()
=>[gremlin.pageRankVertexProgram
        .pageRank:[0.11375510357865538],name:[marko],
        age:[29]]
=>[gremlin.pageRankVertexProgram
        .pageRank:[0.14598540152719103],name:[vadas],
        age:[27]]
=>[gremlin.pageRankVertexProgram
        .pageRank:[0.3047200907912249],name:[lop],
        lang:[java]]
=>[gremlin.pageRankVertexProgram
        .pageRank:[0.14598540152719103],name:[josh],
        age:[32]]
=>[gremlin.pageRankVertexProgram
        .pageRank:[0.17579889899708231],name:[ripple],
        lang:[java]]
=>[gremlin
        .pageRankVertexProgram.pageRank:
        [0.11375510357865538]
        ,name:[peter],age:[35]]
```

Ako je viditeľné v ukážke do parametru funkcie `program` stačí odovzdať náš `VertexProgram`. K implementovaniu vlastného programu je potrebné vytvoriť Javovskú triedu, ktorá bude implementovať rozhranie `org.apache.tinkerpop.gremlin.process.computer.VertexProgram`

3.1.17 SparkGraphComputer

Gremlin obsahuje modul Hadoop-Gremlin, ktorý bol navrhnutý na exekúciu OLAP operácií cez tzv. GraphComputer. Jedným z podporovaných Graph-Computerov je **SparkGraphComputer**. Jedná sa o výpočtový engine, ktorý mapuje OLAP dotaz do Apache Spark, viď 2.3.6. Implementáciu tohoto enginu poskytuje Spark-Gremlin.

Spracovanie OLAP dotazu s využitím Sparku možno vykonať aj prostredníctvom YARN-u, viď 2.3.5.

Hlavnou výhodou pri použití SparkGraphComputera je využívanie cachovacích schopností Sparku, čo vedie k zníženiu prenosu dát, ktoré je potrebné posielať cez sieť pri každej iterácií VertexProgram-u. V iniciálnom kroku je graf načítaný do Spark RDD objektu, viď 2.3.6.

3.2 Motivácia použitia grafových databáz pre analýzu

Z povahy datasetu, ktorý je analyzovaný v tejto diplomovej práci je vhodné použiť grafovú databázu. Hlavným dôvodom sú väzby medzi jednotlivými uzlami. Väzieb je násobne viac ako samotných uzlov. Z podstaty veci grafových databáz (väzby sú reálne uložené) je tento typ databázy najvhodnejším riešením.

3.3 Grafové databázy

V tejto kapitole sú stručne zhrnuté grafové databázy, ktoré je možno použiť na spracovávanie big data, tj ide predovšetkým o databázy, ktoré dokážu operovať nad distribuovaným filesystémom a dokážu spracovávať operácie paralelne. Tento typ databáz bol vybraný predovšetkým kvôli možnosti horizontálneho škálovania, viď 3.1.2.

3.3.1 Problémy pre návrh distribuovanej grafovej databázy

Na trhu sa aktuálne nenachádza veľa grafových databáz, ktoré dokážu pracovať na distribuovanom file systéme. Hlavným problémom je to, že navrhnúť správne rozdistribuovanie grafu na viacero výpočtových uzlov je náročná výzva. Autor [19] uvádza, že súvisiace uzly musia byť na spoločnom výpočtovom uzle, aby prechádzanie grafom mohlo byť efektívne realizované. Na druhej strane je tento spôsob v rozpore s tým, že uzly by mali byť distribuované na rôznych výpočtových uzloch (aby sa predišlo situácii, v ktorej by aktívne počítal len jeden výpočtový uzol a zvyšok by nečinne čakal).

3.3.2 Vznik grafových distribuovaných databáz

Potrebu vzniku grafových databáz, prípadne prvých priekopníkov v tejto oblasti popisuje zdroj [20]. Spoločnosť Google sa venovala tejto problematike pri potrebe výpočtu PageRanku jednotlivých stránok, viď 3.1.4. Google prezentuje "internet" ako obrovský graf, v ktorom uzly grafu predstavujú jednotlivé webové stránky a hrany predstavujú prepojenie stránok (odkazy z jednej

stránky na druhú). V roku 2010 Google predstavil prácu s názvom Pregel, v ktorom popísal koncepty grafovej distribuovanej databázy.

Spoločnosť Facebook potrebovala veľké grafy, rovnako ako Google, tiež spracovávať. Facebook obsahuje graf, v ktorom jednotliví užívatelia predstavujú uzly a väzby priateľstva predstavujú hrany. Pre odporúčanie priateľstiev, počítanie SNA metrík, viď 4.1.1 a mnoho ďalších analýz potreboval Facebook tento graf zanalyzovať. K tomuto používal produkt Äpache Giraph", viď 3.1.12.

Projekt "Apache Giraph" vznikol ako prvý na distribuovanom filesystéme HDFS, viď 2.3.1, pričom koncept grafovej databázy bol postavený na práci Pregel. Projekt "Apache Giraph" umožňoval pracovať s grafmi obsahujúcimi miliardy väzieb. [20]

V jari roku 2014 bola analýza na Hadoop-e, viď 2.3 stále v ranných štádiách. Pri príchode YARN-u, viď 2.3.5 sa stala grafová analýza populárna. Mnoho sociálnych webových stránok začalo využívať na analýzu grafov práve Apache Giraph, viď 3.1.12. Úspech tohoto projektu možno pripísať tomu, že túto službu využíval Facebook, preto sa časom pridali ďalšie veľké spoločnosti.

Dôležité je podotknúť, že tento projekt má svoje limitácie. Nástroj je určený predovšetkým ako výpočtový nástroj, pretože dáta načíta ako graf, uloží ich do pamäte clusteru Hadoopu a následne vykoná požadované dotazy. [20]

Ďalším alternatívnym riešením je projekt, ktorý pôvodne pochádzal od firmy Aurelius, spoločnosti, ktorá vytvorila viacero opensourceových projektov zaoberajúcich sa grafovou analýzou nad Hadoopom. Jedná sa o grafovú databázu Titan, viď 3.3.4. [20]. Tento projekt v roku 2015 skončil a vznikol nový projekt, ktorý vychádzal z projektu Titan s názvom JanusGraph (jednalo sa o fork, viď 3.1.6 Titanu). [21]

Apache Spark, viď 2.3.6, vytvoril projekt GraphX, ktorý umožňuje vygenerovať dáta grafu a následne ich spracovať, všetko interne, vo vnútri Spark frameworku. [20]

3.3.3 Pregel - Google

3.3.3.1 Motivácia

Pri analýze grafu častokrát potrebujeme poznať celú štruktúru grafu. V prípade, že sa jedná o veľké grafy, vzniká otázka ako vyriešiť výpočet nad grafom, ktorý nie je možné uložiť na jeden výpočtový uzol (či už z kapacitných dôvodov alebo z dôvodov výkonnostných, viď 3.1.2).

Pre výpočet rôznych grafových algoritmov je potrebné ukladať medzivýpočty k jednotlivým úzlom (príklad môže byť algoritmus BFS, a príznak, či sme už uzol navštívili). Predpokladajme jednoduchý výpočet najkratšej cesty medzi uzlom A a uzlom B, pričom uzol A je uložený na výpočtovom uzle 1 a uzol B je uložený na výpočtovom uzle 2. V momente keď narazíme na

hranu, ktorá spája uzol X z výpočtového uzla 1 s uzlom Y z výpočtového uzla 2 vzniká otázka ako si preniesť informácie. Tento problém je možné vyriešiť nasledovným spôsobom. Zistiť, do ktorých grafových uzlov sa vieme v rámci výpočtového uzla 2 dostať z grafového uzla Y. Ak máme kompletnú informáciu, tento podgraf na uzle 2 je možné prekopírovať na uzol 1 (prípadne, ak sa jedná o veľkú množinu, možno prekopírovať podgraf z uzla 1 (vrátane vypočítaných hodnôť) na uzol 2). Ak by sa jednalo o malý graf, tento spôsob by fungoval, no pri veľkých grafoch môže jednoducho dôjsť k situácii, že na vyhodnotenie vybraného algoritmu budeme potrebovať pracovať s celým grafom a ten sa nám nezmestí na jeden výpočtový úzol.

3.3.3.2 Princíp

V roku 2010 Google vytvoril Pregel. Pregel možno prirovnať k paralele MapReduce algoritmu pri spracovávaní distribuovaných grafov.

Pregel pracuje s uzlami a hranami na výpočtovom uzle, ktorý výkonáva výpočet a sieťovú komunikáciu medzi výpočtovými uzlami používa len na správy. Prostredníctvom správ pretvára štruktúru grafu z iniciálneho stavu do nového stavu. Tento proces možno prirovnať k MapReduce fáze, viď 2.3.4.

Proces, v ktorom sa spracuvávajú správy a následne sa odošlú, sa nazýva "superstep" (ďalej označovaný len ako krok). Pregel iteratívne vykonáva tieto kroky. V priebehu jedného kroku S sa vykoná užívateľom definovaný kód na každom uzle grafu paralelne. Užívateľsky definovaný kód pracuje s uzlom V a správami, ktoré mu prišli v kroku $S-1$, čím môže zmeniť internú štruktúru uzlu, nad ktorým pracuje, môže editovať odchádzajúce hrany, môže vytvoriť nové uzly (napríklad pre potreby clusteringu), môže deaktivovať uzly, môže poslať správu uzlom, s ktorými je prepojený prichádzajúcimi alebo odchádzajúcimi hranami. [22]

Správy, ktoré boli odoslané v kroku S budú následne spracovávné uzlami v kroku $S+1$. Po každom kroku nastáva synchronizácia. Ak sú správy posielané v rámci jedného výpočtového uzla, spracujú sa na danom výpočtovom uzle. Ak je potrebná komunikácia medzi uzlami, naagregujú sa všetky správy posielané na rovnaký výpočtový uzol, skomprimujú sa a následne sa odošlú. [22]

Každý uzol je v každom kroku buď aktivovaný alebo deaktivovaný. Na začiatku je každý uzol aktivovaný. Ak sa uzol v kroku S deaktivoval, zostane deaktivovaný až do momentu, kým mu nepríde správa. Výpočet končí, ak sa v kroku X nenachádza žiaden aktivovaný uzol. [22]

Výpočtové uzly možno rozdeliť do dvoch kategórií: worker uzly a master uzol. Master uzol je zodpovedný za riadenie celého procesu, pričom worker pracuje nad jednotlivými uzlami grafu. Na začiatku každého kroku uloží každý uzol svoj interný stav do grafového úložiska (checkpoint). Ak v kroku S master uzol zistí, že worker uzol neodpovedá (neodpovedá na ping od mastera), prehlási ho za mŕtveho. Master uzol informuje funkčné worker uzly aby začali

obsluhovať časť grafu, ktorú mal na starosti mŕtvy výpočtový uzol. Výpočet sa vráti k prechádzajúcemu checkpointu a pokračuje sa. Týmto mechanizmom je zabezpečená odolnosť voči poruchám. [23]

Pri implementácii tohoto konceptu sa z pohľadu programátora dá na problematiku pozerať spôsobom ”mysli ako uzol”.

3.3.4 Titan

Titan je škálovateľná grafová databáza optimalizovaná na ukladanie a dotazovanie nad grafmi obsahujúcimi stovky miliárd uzlov a hrán nad distribuovaným cluster-om, viď 3.1.7. Titan spadá do transakčných databáz, viď 3.1.8, a dokáže obsluhovať radovo tisícky súčasne pripojených užívateľov vykonávajúcich zložité grafové operácie v reálnom čase. Databáza podporuje ACID, pozri 3.1.9 a eventuálnu konzistenciu. [24]

3.3.4.1 Možné typy úložiska

Samotné dáta môžu byť uložené v jednej z týchto databáz:

- Apache Cassandra, viď 3.1.8.3
- Apache HBase, viď 3.1.8.2
- Oracle BerkeleyDB, viď 3.1.8.4

Na základe požiadavok na databázu je možné vybrať si vhodný typ úložiska, viď 3.6.

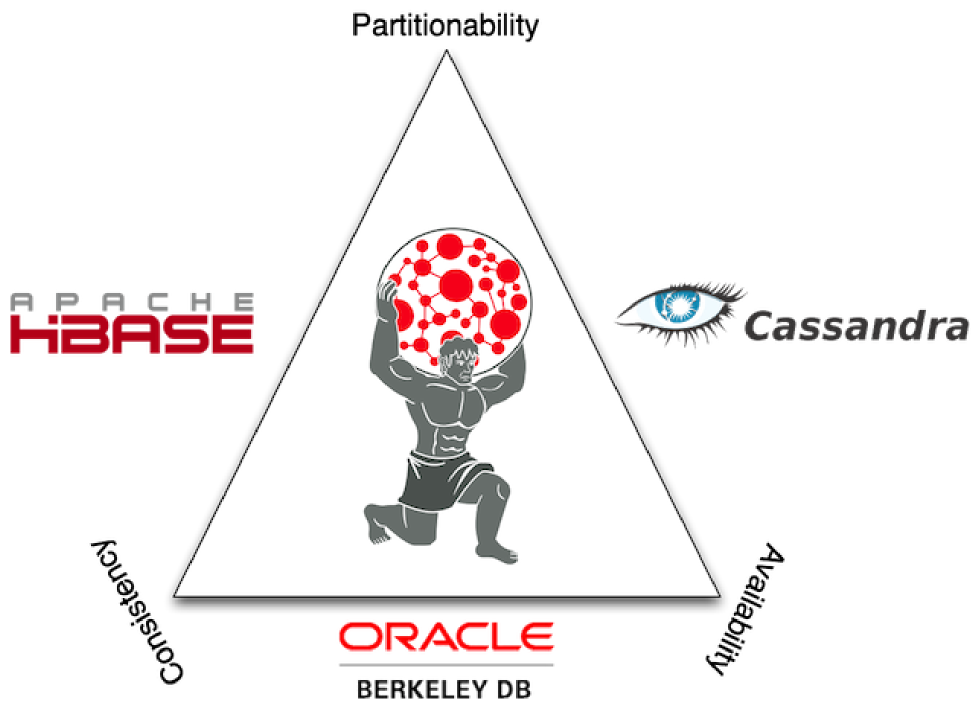
3.3.4.2 Podpora pre spracovanie grafu

Titan podporuje integráciu so známymi frameworkami na spracovanie dát, konkrétne sa jedná o:

- Apache Spark, viď 2.3.6
- Apache Giraph, viď 3.1.12
- Apache Hadoop, viď 2.3

3.3.4.3 Integrácia s TinkerPop

Spomínaná databáza podporuje integráciu s frameworkom Apache TinkerPop (grafový výpočtový engine, ktorý je zároveň opensource) a s jazykom Gremlim, viď 3.1.15.2.



Obr. 3.6: Výber vhodnej databázy podľa CAP teorému. Prevzaté z [25]

3.3.4.4 Problémy

V roku 2015 projekt TitanDb odkúpila firma DataStax, ktorá túto službu integrovala do svojich projektov. Od verzie 1.0, ktorá bola vydaná v septembri roku 2015 už nepripudli žiadne aktualizácie a tým pádom projekt skončil.

V roku 2016 bol vydaný produkt DataStax Enterprise Graph (DSE). Spoločnosť DataStax stojí aj za databázou Cassandra. Nový produkt bol inšpirovaný TitanDb (predpokladá sa, že 10% kódu je použítého z projektu TitanDb [26]), v ktorom bola bližšia integrácia s úložiskovým backendom Cassandra. [27]

V roku 2017 vznikol nový projekt s názvom JanusGraph, ktorý pokračoval tam, kde projekt Titan skončil. Jednalo sa o fork, viď 3.1.6. Nový projekt je zastrešovaný Linux Foundation, pričom projekt aktívne vyvíjajú veľké spoločnosti ako Google, IBM, Expero, Hortonworks. Kód je voľne dostupný na URL `https://github.com/JanusGraph/janusgraph`. [21] Bližšie sa projektu venuje kapitola 3.3.5.

3.3.5 JanusGraph

Projekt JanusGraph vychádzal z projektu Titan. Samotný názov "Janus" pochádza z mena rímskeho Boha, ktorý hľadí súčasne jak na minulosť (k dobe

Titánov), tak do budúcnosti. [21]

V porovnaní s databázou Titan bola rozšírená integrácia o nový typ úložiska Google Cloud Bigtable. JanusGraph podporuje len TinkerPop3, pozri 3.1.15, čo môže spôsobovať problémy pri prechode z databázy Titan.

Oproti Titan datábaze je možné JanusGraph integrovať s frameworkami, ktoré dokážu graf vizualizovať. Jedná sa konkrétne o tieto nástroje: [28]

- Cytoscape
- Gephi - doplnok pre Apache Tinkerpop
- Graphexp
- KeyLines by Cambridge Intelligence
- Linkurious

3.3.6 TigerGraph

V septemberi 2017 bola pre verejnosť uvoľnená grafová databáza TigerGraph. V porovnaní s grafovou databázou JanusGraph sa jedná o distribuovanú grafovú databázu, ktorá používa vlastný dátový model na ukladanie grafu. TigerGraph sa sústreďuje na cieľ, ktorý v IT svete chýbal, a to na nutnú potrebu tvorby analýzy (v reálnom čase) pre enterprise podniky, ktoré majú obrovské množstvo dát. [29]

Dôvod vzniku tejto databázy bol ten, že na trhu chýbali grafové databázy, ktoré boli vytvorené na spracovávanie big data. Grafové databázy, ktoré boli na trhu, dokázali operovať len na jednom vypočetnom uzle, tj neboli navrhované s myšlienkou distribuovaného rozloženia dát. [29]

Pred samotným uvedením pre verejnosť bol tento projekt testovaný na grafoch, ktoré obsahovali 100 miliárd uzlov, 600 miliárd hrán na 20 výpočtových uzloch s denným prírastkom, ktorý predstavoval 2 miliardy udalostí. [29]

Jednou z hlavných výhod je schopnosť preskúmať stovky miliónov uzlov/hrán za sekundu na každom výpočtovom uzle pri 3 a viac skokoch (1 skok (hop) predstavuje prechod cez jednu hranu). Táto funkcionalita je pre mnoho databáz problematická, nakoľko každý ďalší skok predstavuje náročnú výpočtovú zložitosť ($\theta(n^x)$, kde n predstavuje počet hrán, a x počet skokov).

3.3.6.1 GSQL

Spoločnosť TigerGraph vytvorila dotazovací jazyk s názvom GSQL. Autori tohoto jazyka ho nazývajú najlepším moderným jazykom pre dotazovanie nad grafovou databázou. V porovnaní s ostatnými (Gremlin a Cypher) vyzdvihujú jeho schopnosť vyhovieť skutočným požiadavkám biznisu. [30]

Jazyk bol tvorený s cieľom poskytnúť paralelné načítavanie, dotazovanie a ultrarýchle analýzy. Silu jazyka vyzdvihujú v porovnaní s databázou Neo4j

Tabuľka 3.2: Benchmark - načítanie dát - graph 500. Prevzaté z [32])

	TG [1]	**Neo4J**	**Neptune**	**JDB** [2]	**Arr.m** [3]	**Arr.r** [4]
hrana	56	37.3	1571.0	574.6	599.6	944.9
uzol	-	-	45.0	43.9	11.5	44.5
predzpr.[5]	-	54.4	93.2	54.4	54.4	54.4
index	-	7.99	-	-	-	-
celkovo	56	99.7	1709.2	672.9	665.9	1043,8
relatívne	1.0	1.8	30.5	12	11.9	18.6

a DataStax, kde výsledky danej požiadavky boli v TigerGraphe vykonané v sekundách, zatiaľ čo v Neo4j alebo DataStaxe bolo možné rovnaký výsledok dosiahnúť v rádoch hodín, ak vôbec. [30]. Názov GSQL má s jazykom SQL podobnú syntax hlavne preto, aby bola zaručená rýchla učiaca krivka pre vývojárov, ktorí poznajú SQL. Jazyk tiež podporuje MapReduce interpretáciu, ktorá je preferovaná NoSQL vývojármi a dôležítá pre škálovateľné a masové paralelné vyhodotenie. [31]

3.3.6.2 Benchmark

Firma TigerGraph vytvorila benchmark, v ktorom porovnáva databázy Neo4J, Neptune (grafová databáza od Amazonu - funguje výlučne na AWS serveroch), JanusGraph, viď 3.3.5 a ArangoDB, viď 3.3.8.

Benchmarky, ktoré boli vykonané, sú popísané v nasledujúcich tabuľkách. Benchmarky obsiahnuté v tabuľkách boli vykonané na 1 výpočtovom uzle, obsahujúcom 32vCPUs, 244 GB RAM.

Test pozostával z dvoch datasetov

1. graph 500 - Syntetický Kroneckerov graf (2.4 miliónov uzlov, 67 miliónov hrán)

2. Twitter - graf užívateľov a prepojenie medzi nimi (kto koho nasleduje) (41.6 miliónov uzlov, 1.47 miliardy hrán)

Benchmark pozostával z testu načítania, veľkosti vytvorenej databázy po načítaní, dotaz obsahujúci K skokovú cestu a test škálovania.

Test načítania Niektoré databázy potrebujú predspracovanie, prípadne pozpracovanie pred samotným nahraním dát. Tieto položky sú v tabuľke označené ako "index" a "predspracovanie". Výsledky benchmarkov zachycujú tabuľky 3.2 a 3.3

Tabuľka 3.3: Benchmark - načítanie dát - Twitter. Prevzaté z [32])

	TG	Neo4J	Neptune	JDB	Arr.m	Arr.r
hrana	785	685	42248	12192	N/A	20137
uzol	-	-	733	839	206	272
predzpr	-	1270	2259	1270	-	1270
index	-	126	-	-	-	-
celkovo	785	2081	45240	14301	N/A	21679
relatívne	1.0	2.7	57.6	18.2	N/A	27.6

Tabuľka 3.4: Benchmark - veľkosť uloženej databázy. Prevzaté z [32])

	Data	TG	Neo4J	Neptune	JDB	ArrangoDB
graph500	967	482	2300	3850	2764	6657
twitter	24375	9500	49000	91140	55296	126970

Tabuľka 3.5: Benchmark - K skoková cesta dát - graph500. Prevzaté z [32])

	K-skok	TG	Neo4J	Nep. [1]	JDB	Arr.m	Arr.r
graph500	1-skok	1	3.3	2.5	4.3	14.6	3.7
	2-skok	1	59	109	216	337	234
	3-skok	1	125	5	4477	8391	4497
	6-skok	1	737	N/A	N/A	N/A	N/A
twitter	1-skok	1	9	14	16	69	N/A
	2-skok	1	40	57	60	63	N/A
	3-skok	1	44	6	642	580	N/A
	6-skok	1	N/A	N/A	N/A	N/A	N/A

Veľkost databázy po uložení Po uložení datasetu do databázy bola zaznamenaná veľkosť databázových súborov v MB. Výsledky sú v tabuľke 3.4.

K skoková cesta Benchmark sa zameral na dotaz, ktorého výsledkom je počet uzlov, ku koľkým sa viem dostať za 1 (testované pre 300 náhodných uzlov), 2 (testované pre 300 náhodných uzlov), 3 (testované pre 10 náhodných uzlov), 6 skokov (testované pre 10 náhodných uzlov). Tabuľka 3.5 neuvádza presné časy, ale hodnoty sú normalizované proti TigerGraph výsledkom. Hodnota N/A v tabuľke znamená, že výpočet neskončil v priebehu 24 hodín.

[1] TigerGraph
[2] JanusGraph
[3] ArangoDB.m
[4] ArangoDB.r
[5] Predzpracovanie
[1] Neptune

Tabuľka 3.6: Benchmark - škálovanie TigerGraph. Prevzaté z [32])

Počet vyp. uzlov	**1**	**2**	**4**	**6**	**8**
priemerný čas dotazu (s)	969.8	535.5	263.4	209.1	144.8
zrýchlenie	1.0	1.81	3.68	4.64	6.70

3.3.6.3 PageRank, Weakly Connected Components

V rámci benchmarku boli testované aj algoritmy, ktoré potrebujú pracovať so všetkými uzlami a hranami. Konkrétne sa jednalo o algoritmus PageRank, viď 3.1.4 a Weakly Connected Components, viď 4.1.1.7.

Test škálovania Tabuľka 3.6 zachycuje škálovateľnost len pre grafovú databázu TigerGraph, nakoľko ostatné testované databázy nepodporujú horizontálne škálovanie, s výnimkou databázy JanusGraph. Nad touto databázou neboli vykonané benchmarky.

3.3.7 OrientDB

OrientDB je prvou NoSQL databázou, ktorá patrí do rodiny tzv. "Multimodel", teda kombinuje model grafu s dokumentami v jednej databáze. Databáza je vyvíjaná od roku 2011 firmou OrientDB Ltd., pričom v roku 2017 bol projekt odkúpený firmou CallidusCloud. [33] Firma CallidusCloud bola chvíľku na to odkúpená firmou SAP. [34]

Samotný vývoj tejto databázy začal už v roku 2010. OrientDB podporuje na ukladanie rôznych parametrov až 20 rôznych dátových typov, pričom umožňuje definovať obmedzenia na každý parameter (test na minimálnu/maximálnu hodnotu, regex, povinnosť atribútu, to, či je hodnota len čitateľná).

Samotná schéma je benevoletná vzhľadom k tomu, že vývojár môže definovať, či bude schéma striktná, čiastočne striktná alebo žiadna. OrientDB pracuje s objektovo-orientovaným návrhom. Samotná databáza umožňuje pracovať s dedičnosťou (rodič - potomok). [34]

Rovnako ako TigerGraph, aj táto databáza disponuje vlastným jazykom na dotazovanie - OrientDB SQL. Dotazy a CRUD, pozri 3.1.11, je jednoduché napísať, ako aj sa v nich orientovať. Samotné SQL nemá dostatočnú vyjadrovaciu schopnosť na prechádzanie grafu. OrientDB SQL rozširuje SQL práve o schopnosť prechádzať grafom. [34] OrientDB podporuje tiež jazyk Gremlin.

3.3.7.1 Distribuovaný režim

Distribúcia je postavená na tzv. Multi-master architektúre, teda architektúre, v ktorej sú všetky výpočtové uzly ekvivalentné. V prípade, že od klienta príde požiadavka, databáza mu priradí výpočtový uzol, s ktorým bude komunikovať (o priradenie sa stará opensource projekt Hazelcast). Replikáciu a sharding rieši databáza automaticky, opäť prostredníctvom Hazelcast-u.

Pôvodne bolo možné databázu spustiť v distribuovanom režime, ktorý bol založený na architektúre Master-Slave, no pri veľkom množstve konkurenčných dotazov dochádzalo k výraznému spomaleniu, nakoľko všetky dotazy prechádzali cez Master uzol (jednalo sa o úzke hrdlo, tzv "bottleneck"). V septembri 2014 došlo k zmene prístupu na vyššie spomínanú Multi-master architektúru, ktorá prišla s verziou OrientDB 2.0. Nová architektúra predstavovala zrýchlenie o 300%. [35]

Transakcie v distribuovanom režime V distribuovanom režime databáza dokáže spracovať transakcie. Princíp je nasledovný: [35]

- ak je transakcia potvrdená (commit), táto informácia sa pošle všetkým master uzlom, t.j. každý uzol je zodpovedný za commitnutie transakcie
- v prípade, že potvrdenie transakcie na niektorých uzloch neprebehlo, skontroluje sa quorum
  - ak väčšina uzlov transakciu potvrdila, transakcia sa považuje za úspešne dokončenú
  - ak väčšina uzlov transakciu nedokončila, transakcia sa považuje za neúspešnú a transakcia sa vráti (rollback)

MapReduce OrientDB podporuje model MapReduce bez Hadoopu. Existuje konektor, ktorý integruje OrientDB na Spark. Natívne sa odporúča používať vstavaný jazyk OrientDB SQL.

Operácie MapReduce sú úplne transparentné pre vývojára. Ak optimalizátor zistí, že daný dotaz je možné vykonať paralelne, vykoná ho tak. [36]

Limitácie V dobe písania tejto diplomovej práce bola vydaná verzia 3.0, ktorá mala tieto nedostatky: [35]

- importovať databázu nie je možné v distribuovanom režime
- za niektorých okolností môžu byť porušené obmedzenia, ktoré boli definované na špecifických uzloch. K tomuto javu môže dochádať pri konkurenčných dotazoch na vytvorenie uzla a aktualizáciu naprieč clusterom
- Sharded indexy nie sú podporované. Unikátnu hodnotu nie je možné zaručiť v prípade, že je skupina uzlov rovnakej triedy rozdistribuovaná naprieč clusterom
- Nie všetky agregujúce funkcie sú podporované v paralelnom režime. Príkladom môže byť funkcia AVG (výpočet priemeru).

3.3.7.2 Licencia

Projekt OrientDB spadá pod licenciu Apache 2.0 (Community edícia). K dispozicii je aj platená varianta Enterprise. Medzi hlavné výhody platenej verzie možno zaradiť: [37]

- Inkrementálne zálohy
- Bezpečnosť
- Podpora 24/7
- Query Profiler
  - nástroj na monitorovanie dotazov. Kto vykonal aký dotaz, koľkokrát za sebou, metriky - dĺžka vyhodnotenia dotazu
- Konfigurácia/prehľad pre distribuovaný clustering
  - prehľad aktuálneho clusteru
    * štatistiky záťaže pre CPU, RAM, DISK CACHE, DISK
    * aktívne pripojenia
    * sieťové požiadavky
    * upozornenia
  - Konfigurácia databázy - zmenu je možné vykonať v reálnom čase (nie je potreba reštartovať server)
    * Quorum zápisu, pozri 3.1.14
    * Quorum čítania, pozri 3.1.14
    * Roly serverov (Master alebo Replica)
- Monitoring v reálnom čase s možnosťou konfigurácie

3.3.8 ArangoDb

Iniciálny koncept databázy vznikol pred ôsmimi rokmi v momente, keď nemeckí inžinieri Claudius Weinberger a Frank Celler navrhovali databázu pre veľke verejné spoločnosti v Europskej Únii prostredníctvom konzultačnej spoločnosti triAGENS GmbH. Vo vtedajšej dobe bolo zvykom používať na rôzne problémy rôzne dátové modely, čo viedlo k používaniu mnohých databáz. Autori sa rozhodli ubrať sa novou cestou, v ktorej vytvoria jeden model, ktorý bude všeobecný pre väčšinu použití - Multi-model databáza (rovnako ako OrientDB). Na tejto myšlienke vznikla databáza s názvom AvocadoDB, ktorá sa o zopár rokov neskôr premenovala na dnes známu databázu ArangoDb (kvôli právnym sporom ohľadom použitia názvu AvocadoDB). [38]

Podobne ako databázy, ktoré boli spomínané v minulých kapitolách, autori vytvorili nový jayzk, prostredníctvom ktorého je možné pracovať s databázou. Názov jazyka je ArangoDB Query Language (AQL). [38]

3.3.8.1 Dátový backend

ArangoDb je možné prevádzkovať v dvoch módoch (podľa spôsobu uloženia dát): [39]

- mmfiles (memory-mapped files)
  - dataset sa musí zmestiť do hlavnej pamäte
  - indexy sa vytvárajú v pamäti a zostanú v nej uložené
  - reštart databázy spôsobí opätovnú rekonštrukciu indexov
  - štart databázy je výrazne dlhší, viď predcházajúci bod
- rocksdb
  - dataset sa nemusí zmestiť do hlavnej pamäte
  - indexy sa nachádzajú aj v hlavnej pamäti, aj na disku
  - rýchly štart databázy

3.3.8.2 Benchmark

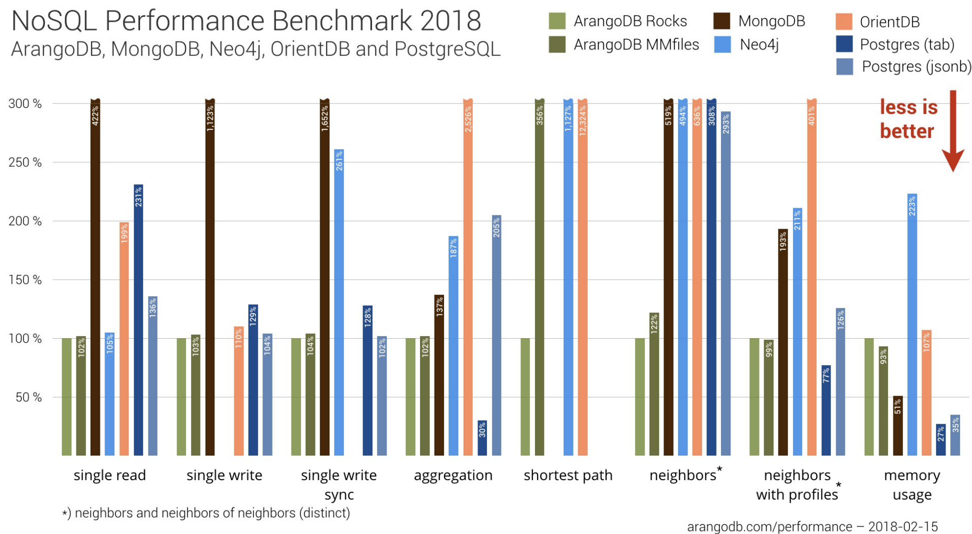
Spoločnosť ArangoDb začiatkom roku 2018 vytvorila benchmark, v ktorom porovnávala výkonnosť rôznych databáz, konkrétne [40]

- Neo4j 3.3.1
- MongoDB 3.6.1
- PostgreSQL 10.1 (tabular & jsonb)
- OrientDB 2.2.29
- ArangoDB 3.3.3

Výsledky benchmarku zachytáva obrázok 3.7.

3.3.8.3 Dataset

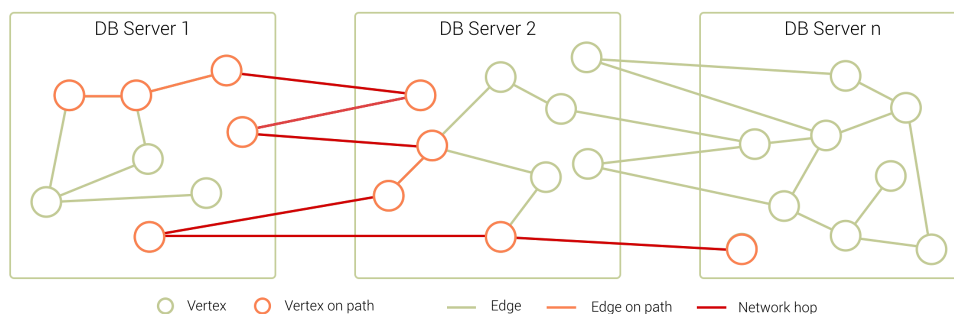
Benchmark bol vykonaný nad datasetom Pokec, ktorý je dostupný na adrese https://snap.stanford.edu/data/soc-pokec.html. Obsahuje 1.6 milióna ľudí (uzlov) a 30.6 milióna hrán. [40]



Obr. 3.7: Benchmark databáz. Prevzaté z [40]

Popis testovaných akcií Benchmark pozostával z nasledovných testov: [40]

- **sigle - read:** výber konkrétneho užívateľa s jednoznačným identifikátorom (opakované pre 100 000 rôznych užívateľov)
- **single - write:** pridanie konkrétneho užívateľa do databázy (opakované pre 100 000 nových užívateľov)
- **single write sync:** rovnaký postup ako v prvom bode s tým rozdielom, že sa meralo, kým sa dáta uložili na disk
- **aggregation:** agregácia podľa atribútu vek
- **shortest path:** hľadanie najkratšej cesty medzi dvomi náhodne zvolenými uzlami (opakované 1000)
- **neighbors:** pre konkrétneho užívateľa zistiť jeho susedov a susedov ich susedov (opakované pre 1000 rôznych užívateľov). Výsledkom boli ID užívateľov
- **neighbors with profiles:** rovnaký postup ako v minulom bode s tým rozdielom, že výsledkom dotazu neboli len ID ale celé profily užívateľov
- **memory usage:** priemerné využitie RAM



Obr. 3.8: Nevhodná distribúcia uzlov. Prevzaté z [42]

3.3.8.4 Distribuovaný režim

Databáza podporuje distribuovaný režim. Na stránke [41] sa vysvetľuje, že pre efektívne využite horizontálneho škálovania je potrebné aby boli dáta rozdistribuované správnym spôsobom. Na tieto účely sa používa platená funkcia Smartgraphs, pozri ako funguje funkcia "'SmartGraphs"', viď 3.3.8.5. ArangoDb poskytuje aj funkcie na vytvorenie bežných grafových metrík. Grafové metriky, ktoré využívajú paralelný výkon clusteru sú detailnejšie popísané v 4.1. Algoritmy sú implementové priamo v ArangoDB, pričom sú postavené práve nad myšlienkou Pregel, pozri 3.3.3. Jedná sa o tieto algoritmy:

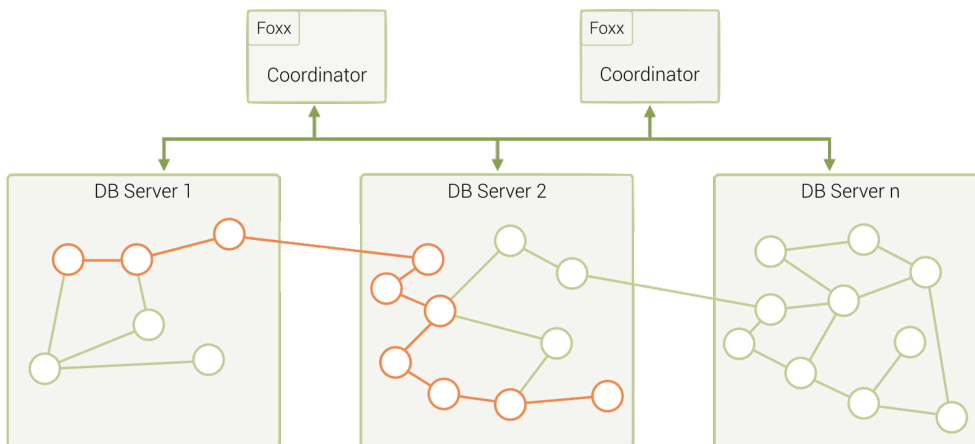
- PageRank, viď 3.1.4
- Slabo/Silno prepojené komponenty, viď 4.1.1.7, 4.1.1.6
- Closeness/Betweenes centrality, viď 4.1.1.3, 4.1.1.4
- Community detection

3.3.8.5 SmartGraphs

Pri distribuovanom režime sa jednotlivé uzly rozložia na rôzne výpočtové uzly. To, akým spôsobom sú uzly uložené, môže do značnej miery ovplyvniť celkovú výkonnosť jednotlivých dotazov. Sieťová komunikácia je drahá operácia (300x násobne drahšia ako komunikácia v rámci jedného výpočtového uzla), preto je dôležité túto činnosť čo najviac obmedziť. [42].

Článok [42] uvádza, že správnym rozdistribuovaním možno získať až 4-8 násobne väčšie zrýchlenie.

Obrázok 3.8 a obrázok 3.9 zobrazujú ako rôznymi spôsobmi možno graf rozdistribuovať na cluster.



Obr. 3.9: Vhodná distribúcia uzlov. Prevzaté z [42]

3.3.8.6 Licencia

ArangoDB je dostupná v 3 edíciách. Community edícia spadá pod licenciu Apache 2.0. Druhou edíciu je Basic edícia, čo predstavuje rozšírenie Community edície o SLA 9x5. Poslednou edíciou je Enteprise, ktorá už nepatrí pod licenciu Apache 2.0. [43] Enterprise edícia je rozšírená o:

- SmartGrapsh, viď 3.3.8.5
- SLA 24x7
- SatelliteCollections
- šifrovanie

3.4 Sumarizácia databáz

Kapitola 3.3 opisovala aktuálne dostupné grafové databázy, ktoré dokážu operovať v distribuovanom režime.

Väčšina uvedených databáz obsahuje edíciu, ktorá spadá pod licenciu Apache 2.0 (výnimkou je TigerGraph, pozri 3.3.6). Vzhľadom k tomu, že TigerGraph neponúka edíciu, ktorá nie je spoplatená, bola vyradená z potenciálnych kandidátov.

Zo zvyšných kandidátov je najvhodnejšie použiť databázu JanusGraph, vzhľadom k tomu, že podporuje najlepšie nástroje na OLAP analýzu, pozri 3.1.16. Táto práca sa zaoberá využitím databázy na pokročilú analýzu, na ktorú sú potrebné grafické procesory ako napríklad Apache Spark alebo Apache Giraph. Výhodou je, že samotná databáza JanusGraph podporuje integráciu s hadoop-gremlin, čo konkurenčné databázy neponúkajú.

V neposlednom rade stojí za zmienku, že za touto databázou stoja veľké firmy ako napríklad Uber, Netflix, IBM a Amazon. [44]

Kapitola 4

Grafové algoritmy

V tejto kapitole sú vysvetlené známe grafové algoritmy. Jedná sa o vysvetlenie toho ako daný algoritmus funguje a k čomu je užitočná táto informácia.

4.1 Grafové metriky

4.1.1 SNA Metriky

SNA pochdádza z anglického slova Social Network Analysis, čo možno preložiť ako analýzu sociálných sietí. Jedná sa o vednú disciplínu, ktorá sa zaoberá rôznymi vlastnosťami grafov.

Medzi známe metriky SNA patria algoritmy ako napríklad rôzne centrality (centralita uzlu (*degree centrality*), blízkosť polohy v strede (*closeness centrality*), clusterové metriky (priradenie jednotlivých uzlov do skupín), detekcia slabých, prípadne silných komponentov).

Tieto rôzne metriky priradzujú jednotlivým uzlom hodnotu. V závistosti od tejto hodnoty možno posudzovať významosť jednotlivého uzlu vzhľadom k vybranej metrike. Niektoré metriky definujú hodnotu len na základe blízkeho okolia, tým pádom nezohľadňujú komplexnú štruktúru grafu.

4.1.1.1 Degree centrality

Jedná sa o najzákladnejšiu a najjednoduchšiu metriku. Táto metrika pridelí každému uzlu počet hrán, s ktorými je uzol prepojený. Vzťah je vyjadrený v 4.1, prevzaté z [45, str. 890], kde $deg(v)$ vyjadrí súčet vstupných a výstupných hrán.

$$C_D(v) = deg(v) \tag{4.1}$$

4.1.1.2 Eigenvector centrality

Jedná sa o sofistikovanejšiu centralitu v porovnaní s 4.1.1.1. Narozdiel od *degree centrality*, táto metrika zohľadňuje aj kvalitu jednotlivých väzieb. Ak je

uzol spájaný s dôležitým uzlom, táto hrana môže zohrávať dôležitejšiu úlohu. Variáciou tejto metriky je napríklad PageRank algoritmus, viď 3.1.4.

Táto centralita je všeobecne definovaná v 4.1.1.2, prevzaté z [45, str. 890], kde λ predstavuje konštantu, $M(v)$ predstavuje množinu susedných uzlov uzla v. Premenná $a_{v,t}$ má hodnotu 1 ak existuje väzba medzi uzlom v a t, inak obsahuje hodnotu 0.

$$C_E(v) = x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t = \frac{1}{\lambda} \sum_{t \in G} a_{v,t} x_t \tag{4.2}$$

4.1.1.3 Closeness centrality

Úlohou tejto centrality je zodpovedať otázku ako veľmi v strede sa daný uzol nachádza. Metrika je daná vzťahom 4.3, prevzaté z [45, str. 890], kde n predstavuje počet uzlov, $d(u_i, v)$ predstavuje najkratšiu vzdialenosť medzi uzlom u_i a v.

$$C_C(v) = \frac{n-1}{\sum_{k=\mathrm{i}}^{n} d(u_i, v)} \tag{4.3}$$

4.1.1.4 Betweeness centrality

V porovnaní s vyššie spomínanými metrikami sa jedná o najzložitejšiu (z pohľadu výpočtovej zložitosti). Táto metrika vyjadruje, koľko najkratších ciest medzi všetkými uzlami prechádza práve cez nami vybraný uzol. Uzly, ktoré majú túto metriku vysokú, zohrávajú významnú rolu v informačnom toku. Matematicky je hodnota tejto metriky definovananá v 4.4, prevzaté z [45, str. 891], kde σ_{st} predstavuje počet najkratších ciest medzi uzlom s a t, a $\sigma_{st}(v)$ predstavuje počet najkratších ciest medzi uzlom s a t takých, ktoré obsahujú uzol v.

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4.4}$$

4.1.1.5 Clustering

Účelom clusteringu je vytvoriť zhluky skupín, v ktorých sú rôzne uzly. Platí, že uzol patrí výlučne do jedného clusteru. Uzly v spoločnom clusteri by mali zdieľať podobné vlastnosti. Existujú rôzne algoritmy, ktoré sa zaoberajú otázkou ako detekovať či sa dané uzly podobajú či nie. Jedným zo známych algoritmov je napríklad *Peer pressure Clustering*.

Peer pressure clustering Jedná sa o clusteringový algoritmus. Známy je tiež pod názvom *Label propagation*. Algoritmus možno popísať nasledovne. [18]

1. Každý uzol si vygeneruje vlastné jednoznačné ID.

2. Každý uzol zistí, koľko má výstupných uzlov - d, a na základe tejto informácie priradí jednotlivým hránam váhu - štandardne je používaný jednoduchý vzťah $1.0/d$.

3. Svojim susedným uzlom pošle svoje jednoznačné ID s príslušnou váhou.

4. Každý uzol dostane n (každá prichádzajúca hrana vygeneruje jednu správu) rôznych správ s rôznymi IDčkami a rôznymi váhami. Z prijatých správ vyberie to ID, ktoré má najvyššiu váhu (pri 1. iterácii obsahujú všetky správy rôzne ID, no všeobecne pri n-tej iterácií je možné, že dostane správy s rovnakým ID (ale inými váhami) - v takom prípade je potreba tieto váhy sčítať) a svoje jednoznačné ID prepíše touto hodnotou.

5. Krok 4 sa iteratívne opakuje až do momentu, v ktorom je dosiahnutý maximálny počet iterácií alebo výsledok dokonvergoval (za istých okolností je možná istá čiastočná konvergencia - napr $x\%$ uzlov nezmenilo svoj cluster), t.j. pri nových iteráciách nedochádza k priradeniu do nových clusterov

6. ID jednotlivých uzlov predstavujú cluster, do ktorého bol prvok zaradený.

4.1.1.6 Silno prepojené komponenty

Nech G je orientovaý graf. Graf G môže obsahovať viacero silno prepojených komponentov. Pojem silno prepojený komponent je taká podmnožina grafu H, v ktorej platia tieto pravidlá

- existuje cesta medzi všetkými uzlami grafu H
- jedná sa o maximálny podgraf, t.j. neexistuje podgraf H', ktorý by bol silne súvislý a platilo by $H \subset H'$

4.1.1.7 Slabo prepojené komponenty

Jedná sa o rovnaký princíp ako v 4.1.1.6 s tým rozdielom, že platia tieto pravidlá:

- existuje neorientovaná (orientácia hrán je ignorovaná) cesta medzi všetkými uzlami grafu H
- jedná sa o maximálny podgraf, tj neexistuje podgraf H', ktorý by bol slabo súvislý a platilo by $H \subset H'$

4.1.1.8 Anomálie

Pojem anomália možno vysvetliť ako niečo neštandardné, niečo, čo sa odlišuje od normálu. Anomálie môžu predstavovať netypické štruktúry špecifického podgrafu alebo odklon od typického správania. Tieto javy môžu v rôznych doménach predstavovať zaujímavé zistenia.

V doméne bankového sveta sú anomálie vo väčšine prípadov zisťované za účelom odhaliť bankové podvody. Existuje viacero prístupov ako tieto anomálie detekovať. Jednému zo spôsobov sa venuje článok [46]. V článku je uvedené, že bankové podvody (anomálie) možno zisťovať práve s využitím personalizovaného PageRanku, viď 3.1.5. Princíp je pomerne jednoduchý. Hlavná myšlienka pozostáva z toho, že vstupnou množinou uzlov budú tie uzly (právnicke osoby), ktoré v minulosti spáchali nejaký bankový podvod. Týmto uzlom bude v prvej iterácií PageRank algoritmu pridelená pomerná pravdepodobnosť ($1/n$), kde n je počet označených uzlov, pričom ostatným uzlom bude pridelená hodnota 0.

Uzly, ktoré budú mať vysokú hodnotu po dobehnutí takto definovaného personalizovaného PageRanku, budú označené za anomálie. Využíva sa predpoklad, ktorý tvrdí, že spoločnosti, ktoré spolupracujú so zločincami, sú tiež zločinci. Tento postup bol použitý aj pri detekcii podvodov v zdravotnej starostlivosti. Účelom bolo zistiť, ktorí lekári vykonávajú podvodné činnosti.

Kapitola 5

Realizácia

V tejto kapitole je popísaný test grafových algoritmov nad distribuovanou databázou JanusGraph. Jedná sa o spustenie známych algoritmov popísaných v 4.1 a zistenie aký vplyv má veľkosť datasetu na dobu trvania týchto výpočtov, ako aj vplyv horizontálneho škálovania.

5.1 Dataset

Dataset pozostáva z prepojenia jednotlivých právnických a fyzických subjektov. Tieto prepojenia sú čerpané z obchodného registra Českej republiky a Slovenskej republiky, ktorý je verejne dostupný. Táto štruktúra je sama o sebe grafom a dáva zmysel túto štruktúru skúmať a vyvodzovať z nej rôzne závery (podvody, biele kone, . . .). Túto štruktúru by bolo zaujímavé rozšíriť o hrany, ktoré by predstavovali bankové transakcie. Vzhľadom k tomu, že existujú informácie o registrovaných číslach účtov DPH, bankové inštitúcie majú všetky potrebné dáta, k tomu aby mohli rozšíriť väzby v obchodnom registri.

Rozšírenie obchodného registra o tieto dáta by mohlo odhaliť nové závislosti. Bankové inštitúcie disponujú relevantnými dátami a majú prístup k českému obchodnému registru, teda bolo by možné tieto dva svety prepojiť. Hlavným problémom ale je, že transakčné dáta medzi jednotlivými entitami nie sú verejné, sú vlastnené bankovými inštitúciami a nemôžu byť ďalej šírené.

Dáta finančných inštitúcií sú chránené prísnymi NDA (zmluva medzi aspoň dvoma stranami o zdieľaní dôverných informácií, ktoré nebudú zdieľané so žiadnou treťou stranou), preto nebolo možné použiť tieto dáta na účely tejto diplomovej práce. V rámci tejto diplomovej práce boli dáta obchodného registra rozšírené o syntetické hrany, ktoré majú znázorňovať bankové transakcie. Dataset je rozdelený na 4 podmnožiny, pričom jednotlivé podmnožiny sa líšia veľkosťou. Štruktúra datasetu je popísaná v 5.1.1, spôsob tvorenia fiktívnych väzieb v 5.1.2.

5.1.1 Rozhranie datasetu

Pre účely tejto diplomovej práce bol vytvorený dataset, ktorý pozostáva z prepojenia fyzických a právnických osôb (možné kombinácie), pričom informácie boli čerpané z obchodných registrov Českej republiky a Slovenskej republiky.

Dataset pozostáva z dvoch typov uzlov: **Fyzická osoba**, viď 5.1 a **Právnická osoba**, viď 5.2. Väzby medzi týmito entitami sa delia na:

- **väzba z obchodného registra** - jedná sa o skutočnú väzbu medzi dvomi subjektami. Môže ísť o väzbu medzi dvoma fyzickými osobami, právnickými osobami alebo o ich vzájomnú väzbu, ktorá sa nachádza v obchodnom registri.
  - jedná sa o orientovanú väzbu, obsahujúcu parametre popísané v 5.3
- **syntetická väzba** - jedná sa o fiktívnu väzbu predstavujúcu platobnú transakciu medzi dvomi subjektami, či už právnickými, fyzickými alebo ich kombináciou, viac v 5.1.2

5.1.1.1 Veľkosti datasetu

Pre vytvorenie benchmarku bol dataset rozdelený na 3 množiny.

- Všetky fyzické a všetky právnické osoby
- Polovica fyzických a polovica právnických osôb
  - vzťahy typu **väzba z obchodného registra**, ktoré odkazovali na neexistujúcu entitu boli odstránené
  - vzťahy typu **syntetická väzba**, ktoré odkazovali na neexistujúcu entitu boli odstránené
- Štvrtina fyzických a štvrtina právnických osôb
  - vzťahy typu **väzba z obchodného registra**, ktoré odkazovali na neexistujúcu entitu boli odstránené
  - vzťahy typu **syntetická väzba**, ktoré odkazovali na neexistujúcu entitu boli odstránené

Selekcia menších datesetov bola implementovaná prostredníctvom knižnice `pandas` a jej funkcie `sample` (jedná sa o funkciu, ktorá náhodne vyberie podmožinu o veľkosti n. Selekcia nie je úplne náhodná - je definovaná počiatočným seedom, to znamená, že túto selekciu možno zopakovať). Predspracovanie jednotlivých datasetov sa nachádza na priloženom CD.

Dôvod, prečo bol zvolený tento spôsob zmenšovania datasetu je ten, že OLAP analýzy v nami vybranej databáze využíva princíp "mysli ako uzol", viď 3.3.3, 3.1.12, počet uzlov teda výraznejšie ovplyvní dobu behu programu.

Tabuľka 5.1: Rozhranie uzlu fyzická osoba

Názov	**Typ**	**Popis**
Entity_id	Long	Jednoznačný identifikátor
Name	String	Meno osoby
Surname	String	Priezvisko osoby
FullAddress	String	Celá adresa osoby
Birthday	Date	Dátum narodenia osoby
Preffix	String	Tituly pred menom
Suffix	String	Tituly za menom
rchash	String	Hash rodného čísla
rcc	String	Zašifrované rodné číslo
country_code	String	Číselníková hodnota - krajina pôvodu
city_txt	String	Mesto
zip_txt	String	Zip

Tabuľka 5.2: Rozhranie uzlu právnická osoba

Názov	**Typ**	**Popis**
Entity_id	Long	Jednoznačný identifikátor
Name	String	Názov právnickej osoby
strt_dt	String	Začiatok vzniku firmy
endt_dt	String	Ukončenie činnosti firmy
country_code	String	Číselníková hodnota - krajina pôvodu
city_txt	String	Mesto
zip_txt	String	Zip

Tabuľka 5.3: Rozhranie spojenia medzi právnickými a fyzickými osobami

Názov	**Typ**	**Popis**
strt_dt	Date	Začiatok platnosti väzby
endt_dt	Date	Koniec platnosti väzby
relation_code	String	Číselníková hodnota - definícia vzťahu
description_cz	String	Definícia vzťahu
relation_reliability	Integer	Indikátor spoľahlivosti väzby

5.1.2 Syntetické väzby

Syntetické väzby predstavujú bankové transakcie medzi jednotlivými uzlami. Myšlienka pre generovanie týchto väzieb bola postavená na tom, že právnické osoby sídliace v rovnakom meste budú obchodovať s právnickými osobami sídliacimi v rovnakom meste.

Tieto väzby boli generované za využitia knižnice `pandas`, pričom princíp bol nasledovný:

1. spočítaj celkový počet väzieb typu *väzba z obchodného registra* pre každú právnickú osobu
2. pre každú **právnickú osobu** vypočítaj 1/6 z celkového počtu väzieb typu *väzba z obchodného registra*. Túto hodnotu označ ako X
3. Pre každú **právnickú osobu** vyber X právnických alebo fyzických osôb, ktoré zdieľajú s právnickou osobou adresu a vytvor medzi nimi syntetickú (transakčnú) väzbu
    - rovnako ako 5.1.1.1 bola použitá funkcia `sample` z knižnice `pandas`. Dôležité je podotknúť, že pre každú spoločnosť bol použitý iný seed (aby nedošlo ku generovaniu rovnakých podmnožín)
    - označme počet osôb (fyzických a právnických), ktoré majú rovnakú adresu ako **právnická osoba** Y. Ak platí, že $X > Y$, vygeneruje sa X náhodných väzieb s fyzickými a právnickymi osobami, ktoré pochádzajú z Českej republiky.

Kód zodpovedný za generovanie týchto syntaktických väzieb je priložený na CD.

5.2 OLAP - SparkGraphComputer, YARN

Ako bolo spomínane v 3.1.16, TinkerPop podporuje dva spôsoby komunikácie s grafom. V tejto kapitole je popísane akým spôsobom je možné vykonať OLAP dotazy prostredníctvom SparkGraphComputer, viz 3.1.17.

TinkerPop podporuje integráciu s frameworkom Apach Spark na vykonanie OLAP dotazu, no proces ako toho docieliť nie je dostatočne zdokumentovaný a nie je triviálny. Využiť Apache Spark je možné v týchto režimoch:

- **spark.master=local[x]** - predstavuje spustenie výpočtu lokálne, pričom parameter x určuje počet threadov.
- **spark.master=IP** - IP predstavuje IP adresu vzdialeného servera, na ktorom je nainštalovaný Apache Spark.
- **spark.master=yarn** - spustenie Apache Sparku prostredníctvom YARN, viz 2.3.5. Týmto spôsobom je možné naplánovať využitie výpočtového výkonu celého clusteru.

5.2.0.1 Prostredie clusteru

Samotný benchmark prebiehal na výpočtovom clusteri pozostávajúcom zo 4 výpočtových uzlov (označme si tieto uzly `uzol1`, `uzol2`, `uzol3`, `uzol4`). Každý uzol obsahoval 11GB RAM a procesor s 2 jadrami. Na clusteri bola nainštalovaná distribúcia HDP vo verzii 2.6.3.0. HBase ako dátovy backend pre JanusGraph bol nainštalovaný vo verzii 1.1.2, pričom obsahoval Region-Servery na všetkých 4 výpočtových uzloch. Zookeeper bol nakonfigurovaný na výpčtovom uzle `uzol1` a `uzol2`. Cluster obsahoval inštaláciu Apache Spark 1.6.3 a Apache Spark 2.2.0. Analýza bola spúštana z gremlin konzoly, ktorá je súćasťou aplikácie JanusGraph 0.3.1.

5.2.0.2 SparkGraphComputer v režime YARN

Informácie zhrnuté v tejto kapitole vychádzajú zo zdrojov [47] a [48]. Najväčším problémom pri definovaní úlohy cez YARN je spôsob, akým sa načítavajú jednotlivé knižnice v jednotlivých kontajneroch. Aplikácia je totiž spúštaná v troch rôznych režimoch, pričom je potrebné správne nastaviť cesty k potrebným knižniciam (jar) súborom pre každú aplikáciu zvlášť.

- aplikácia **gremlin-console** je klientská aplikácia, ktorá vystupuje v roli yarn clienta
- YARN vytvorí **ApplicationMaster**, viď 2.3.5, pričom tento Application Master potrebuje prístup k JanusGraph knižniciam
- YARN vytvorí separátne **containery**, ktoré zohrávajú úlohu workerov, pričom rovnako ako Application Master potrebujú pristup k JanusGraph knižniciam

Cestu k rozbalenému adresáru JanusGraph 0.3.1 označme `$JANUS_HOME`. V tomto adresári je potrebné vytvoriť súbor `init.sh`, ktorý bude obsahovať nastavenia premenných, ktoré sú nevyhnutné pre jeho správne fungovanie. Obsah súboru je následovný:

```
#!/bin/bash

export GREMLIN_HOME=$JANUS_HOME
export CLASSPATH=/usr/hdp/current/hadoop-client/conf:
/usr/hdp/current/hbase-client/conf:$GREMLIN_HOME/lib/*
export HADOOP_GREMLIN_LIBS=$GREMLIN_HOME/empty
export JAVA_OPTIONS="-Djava.library.path=
/usr/hdp/current/
hadoop-client/lib/native:
/usr/hdp/current/hadoop-client/lib/native/Linux-amd64-64
-Dhdp.version=2.6.3.0-235"
export SPARK_HOME=/usr/hdp/current/spark2
```

```
export HDP_VERSION=2.6.3.0-235
export SPARK_JAVA_OPTS="-Dhdp.version=2.6.3.0-235"
export SPARK_MAJOR_VERSION=2
```

Do adresára *$JANUS_HOME/lib* je potrebné skopírovať jar súbory, ktoré sú potrebné pre vytvorenie Spark aplikácie. Konkrétne sa jedná o tieto súbory: *hadoop-yarn-server-web-proxy.jar*, *scala-reflect.jar*, *spark-yarn.jar*. V HDP distribúcií sa tieto súbory nachádzajú v adresári */usr/hdp/${hdp_version}/ spark2/jars*.

Ako ďalší krok je potrebné vytvoriť zip súbor, ktorý obsahuje všetky jar súbory prítomné v adresári *$JANUS_HOME/lib*. Takto vytvorený súbor je potrebné uložiť do adresára *$JANUS_HOME* pod názvom *lib.zip*. Dôvod prečo je tento krok potrebný je ten, aby si Spark ApplicationMaster mohol tieto knižnice bez problémov stiahnuť. [47].

Následne je nevyhnutné definovať súbor, ktorý obsahuje informácie potrebné pre nadviazanie spojenia s databázou, ako aj definície potrebné pre spustenie SparkGraphComputera cez YARN. V princípe sa jedná o súbor, ktorý je definovaný v [48]. V porovnaní s referenčným súborom bol atribút `spark.yarn.dist.archives` premenovaný na `spark.yarn.archive`. Dôležité je uviesť, že parameter `janusgraphmr.ioformat.conf.storage.hostname` vyjadruje spojenie so Zookeeper serverom, ktorý komunikuje s HBase a nie na HMaster prípadne RegionServer, viď 3.1.8.2.

Parametre, ktoré boli pri benchmarkoch použité a nie sú obsiahnúte v referenčnom súbore sú:

- **spark.executor.instances** - udáva počet inštancií vytvorených YARN-om. Tento parameter definuje nakoľko paralelne výpočet pobeží.

Finálne je možné spustiť gremlin konzolu *$JANUS_HOME/ bin/gremlin.sh* a výpočet PageRank metriky iniciovať týmito príkazmi:

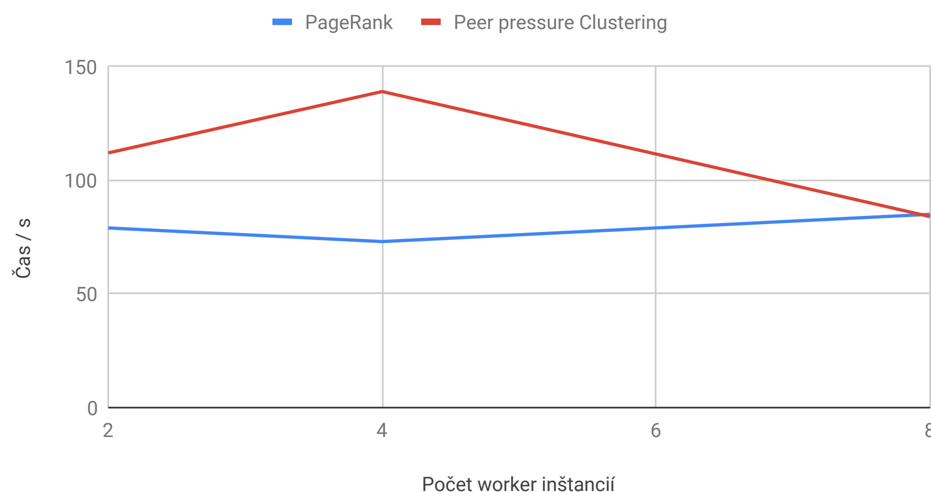
1. graph = GraphFactory.open('conf/hadoop-gremlin/graph.properties')

2. result = graph.compute(SparkGraphComputer) .program(PageRankVertexProgram.build().create()).submit().get()

5.2.1 Benchmark OLAP dotazov

V tejto kapitole je vytvorený benchmark, ktorý testuje ako je možné zvládať paralelne spracovanie grafu.

Pri spúšťaní jednotlivých algoritmov bolo zistené, že výkon testovacieho clusteru nebol dostatočný, viz 5.2.0.1. Limitujúcim faktorom bola nedostatočná pamäť RAM. Počas behu aj na najmenšom datasete často dochádzalo k pádu vytvorených containerov kvôli nedostatku pamäte. Preto boli vytvorené menšie datasety, ktoré obsahovali 10 000 a 100 000 uzlov. Nad týmito



Obr. 5.1: Benchmark malého datasetu

datasetmi bol testovaný algoritmus Peer pressure clustering, viz 4.1.1.5 a PageRank. V rámci práce bol implementovaný rozšírený PageRank, konkrétne sa jednalo o personalizovaný PageRank. Z pohľadu časovej náročnosti je tento algortimus zhodný s natívne implementovaným algoritmom v TinkerPop module.

Benchmark skúmal aký vplyv má počet worker containerov na dobu trvania výpočtu. Metrika, ktorá bola zvolená na meranie potrebného času bola definovaná životným cyklom aplikácie v YARN. T.j. čas je definovaný ako rozdiel medzi vznikom aplikacie v YARNe a jej úspešným dokončením.

5.2.1.1 Benchmark menšieho datasetu

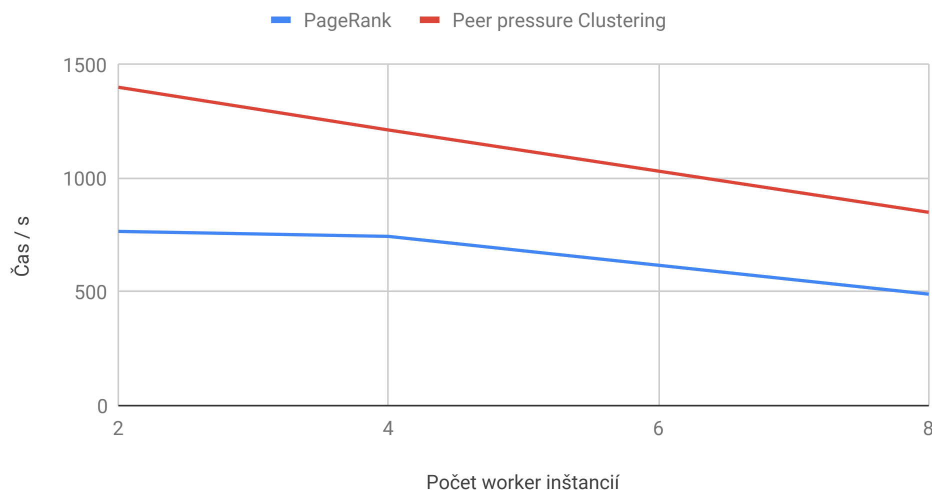
Vzhľadom k tomu, že testovací cluster pozostával z 2*4 CPU jadier, kde 2 je počet procesorov na každom výpočtovom uzle a 4 je počet výpočtových uzlov, nemalo zmysel vytvárať viac ako 8 workerov.

Postupne boli spomínane algoritmy spúšťané s 2, 4 a 8 workermi (zmena parametru **spark.executor.instances**). Test pozostával z 3 exekúcií s každým parametrom a hodnota bola následne spriemerovaná.

Doba výpočtu spomínaných algoritmov je znázornená na grafe 5.1.

5.2.1.2 Benchmark väčšieho datasetu

Benchmark väčšieho datasetu (100 000 uzlov) bol vykonaný rovnako ako benchmark menšieho datasetu, viď 5.2.



Obr. 5.2: Benchmark veľkého datasetu

5.3 Zhodnotenie výsledkov

Výsledky behu nad malým datasetom boli celkom prekvapivé, nakoľko pri väčšom výkone bola doba výpočtu dlhšia. Tento jav bol pravdepodobne spôsobený tým, že dataset bol príliš malý a réžia vytvárania viacerých containerov a následná synchronizácia prevyšovala benefity.

Pri väčšom datasete už bolo viditeľné mierne zrýchlenie. Grafové algoritmy sú výpočtovo časovo náročné, preto sa vo väčšine prípadov používajú výkonné clustery. Vzhľadom k zisteniu, že spracovanie grafu je prostredníctvom SparkGraphComputera škálovateľné, je možné predikovať potrebný výkon na spracovanie väčšieho množstvo dát.

5.4 Možnosti použitia v komerčnej sfére

Dáta, ktoré boli použité v tejto diplomovej práci boli kvôli NDA tvorené synteticky. Podstatné je, že benchmark ukázal, že výpočty sú škálovateľné a rôzne metriky je možné spočítať aj nad ostrými dátami a následne z týchto metrík vyvodiť rôzne závery. Čo sa týka detekovania anomalít, bude možné vybrať malú množinu podozrivých subjektov, ktoré bude možné ďalej individuálne posudzovať a zistovať, či sa jedná o skutočnú anomáliu (v kontexte data science označované ako ”true positive”).

Hlavným problémom pri behu týchto algoritmov je vysoká náročnosť na RAM nakoľko SparkGraphComputer natívne funguje v `MEMORY`. Pri použíti

cachovania na disk stále nie je zaručené, že výpočet úspešne dobehne.

Výsledný projekt je možno spustiť nad ostrými dátami, pričom je potrebné upraviť len zopár konfiguračných parametrov (napríklad spojenie s databázou, autentifikáciu, parametre určujúce paralelný výpočet (koľko workerov, s akými zdrojmi) ...).

Záver

Cieľom tejto práce bolo vytvoriť rešerš dostupných grafových databáz, ktoré dokážu spracovávať veľké dáta (uzly a hrany v radoch miliónov). Táto práca popisuje základné technológie, ktoré sa spájajú s distribuovaným spracovávaním dát, ktoré sú využívané v grafových databázach.

Práca ďalej vysvetľuje princípy, ktoré opisujú ako masívne paralelne spracovávať graf (Pregel). Následne sú v práci vysvetlené grafové nástroje na spracovanie grafov, ako je napríklad Giraph alebo Spark.

Ďalšia časť sa zaoberá grafovými databázami. V práci sú spomínané štyri grafové databázy, ktoré podporujú beh v distribuovanom režime a to JanusGraph (Titan), OrientDB, ArrangoDB, TigerGraph. Z vymenovaných databáz bol zvolený JanusGraph z dôvodu, že umožňuje prepojenie s TinkerPop Frameworkom a jeho doplnkom Hadoop gremlin, v ktorom je možné spúšťať dotazy, ktoré efektívne pracujú s celým grafom. Databáza JanusGraph využíva na ukladanie grafu rôzne databázy (označované ako dátové backendy).

Práca popisuje HBase a Cassandru ako dvoch kandidátov, pričom vysvetľuje ako funguje HBase a akým spôsobom sa odlišuje od databázy Cassandra. Pre účely tejto práce bola zvolená databáza JanusGraph s dátovým backendom HBase najmä preto, že HBase poskytuje rýchlejšie čítanie dát ako Cassandra.

V neposlednom rade sú vysvetlené základné grafové metriky (SNA) ako aj jeden z clusterovacích algoritmov. Na detekciu anomalít bol zvolený personalizovaný PageRank, pomocou ktorého možno detekovať podozrivé väzby.

V poslednej časti bol vytvorený benchmark zvolenej databázy nad datasetom, ktorý predstavuje jak reálne tak syntetické väzby. Dataset pozostáva z prepojenia jednotlivých právnických subjektov a fyzických subjektov. Tieto prepojenia sú čerpané z obchodného registra Českej republiky a Slovenskej republiky, pričom je rozšírený o syntetické väzby, ktoré predstavujú bankové transakcie medzi entitami. Dataset bol vo finalnej fáze zmenšený na dva datasety (10 000 a 100 000 uzlov) z dôvodu nedostatočného výpočetného výkonu testovacieho clustera. Nad týmito dvoma datasetami bol vytvorený bench-

mark, ktorý potvrdil, že OLAP dotazy sú horizontálne škálovateľné.

Posledná časť obsahuje zhodnotenie využita tejto grafovej databázy v komerčnom projekte.

Ciele tejto práce boli splnené.

Literatúra

[1] Augustín, J.: Big data a možnosti jejich využití. *Adastra*, september 2014, [cit. 2019-01-31]. Dostupné z: https://www.adastra.cz/clanky/big-data-a-moznosti-jejich-vyuziti

[2] Bonaci, M.: The history of Hadoop. *Medium*, apríl 2015, [cit. 2019-01-31]. Dostupné z: https://medium.com/@markobonaci/the-history-of-hadoop-68984a11704

[3] Apache Hadoop: *Apache Hadoop*. [cit. 2019-04-19]. Dostupné z: https://hadoop.apache.org

[4] Apache Hadoop: *HDFS Architecture*. [cit. 2019-04-19]. Dostupné z: http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[5] Apache Hadoop: *MapReduce Tutorial*. [cit. 2019-04-19]. Dostupné z: http://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[6] Subramaniam, A.: Hadoop YARN Tutorial – Learn the Fundamentals of YARN Architecture. *edureka!*, December 2018, [cit. 2019-04-16]. Dostupné z: https://www.edureka.co/blog/hadoop-yarn-tutorial/

[7] Ondřej, M.: *Rozšíření projektu Spark o podporu jazyka Ruby*. Diplomová práce, České vysoké učení technické v Praze, 2015.

[8] Šimek, M.: SEO mýtus #3: Čím vyšší ranky (PageRank, S-rank), tím lepší pozice ve vyhledávačích. *Blog Martina Šimka*, január 2015, [cit. 2019-01-31]. Dostupné z: http://blog.martinsimko.cz/2015/01/19/ranky-pozice-vyhledavace/

[9] Github Inc: *Fork a repo.* [cit. 2019-02-01]. Dostupné z: https://help.github.com/articles/fork-a-repo/

[10] Rouse, M.: cluster. *WhatIs*, apríl 2006, [cit. 2019-02-04]. Dostupné z: https://whatis.techtarget.com/definition/cluster

[11] MapR Technologies: *WHAT IS APACHE HBASE?* [cit. 2019-04-24]. Dostupné z: https://mapr.com/products/apache-hbase/

[12] McDonald, C.: An In-Depth Look at the HBase Architecture. *MapR Technologies*, August 2015, [cit. 2019-04-24]. Dostupné z: https://mapr.com/blog/in-depth-look-hbase-architecture/

[13] Bekker, A.: Cassandra vs. HBase: twins or just strangers with similar looks? *ScienceSoft USA Corporation*, Jún 2018, [cit. 2019-04-24]. Dostupné z: https://mapr.com/blog/in-depth-look-hbase-architecture/

[14] Medek, J.: *Anal7za DB knihoven a technologií pro Javu.* Diplomová práce, Západočeská univerzita v Plzni, 2015.

[15] Schelter, S.: Large Scale Graph Processing with Apache Giraph. [online], Máj 2012, [cit. 2019-04-24]. Dostupné z: https://researcher.watson.ibm.com/researcher/files/us-heq/Large%20Scale%20Graph%20Processing%20with%20Apache%20Giraph.pdf

[16] Bekker, A.: Scaling Apache Giraph to a trillion edges. *Facebook*, August 2013, [cit. 2019-04-24]. Dostupné z: https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920/

[17] Kovář, M.: *Benchmark grafových databázi pro potřeby data lineage.* Diplomová práce, České vysoké učení technické v Praze, 2017.

[18] The Apache Software Foundation: *TinkerPop Documentation.* [cit. 2019-02-01]. Dostupné z: http://tinkerpop.apache.org/docs/3.4.1/reference/

[19] Grolinger, K.; Higashino, W. A.; Tiwari, A.; aj.: Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, December 2013, [cit. 2019-01-31]. Dostupné z: https://doi.org/10.1186/2192-113X-2-22

[20] deRoos, D.: Graph processing in Hadoop. *dummies*, [cit. 2019-01-31]. Dostupné z: https://www.dummies.com/programming/big-data/hadoop/graph-processing-in-hadoop/

[21] Harris, D.: Google, IBM back new open source graph database project, JanusGraph. *Architect*, január 2017, [cit. 2019-02-01]. Dostupné z: https://architecht.io/google-ibm-back-new-open-source-graph-database-project-janusgraph-1d74fb78db6b

[22] Colyer, A.: Pregel: A System for Large-Scale Graph Processing. *the morning paper*, máj 2015, [cit. 2019-02-01]. Dostupné z: https://blog.acolyer.org/2015/05/26/pregel-a-system-for-large-scale-graph-processing/#respond

[23] Ho, R.: Google Pregel Graph Processing. *DZone*, júl 2010, [cit. 2019-02-01]. Dostupné z: https://dzone.com/articles/google-pregel-graph-processing

[24] TITAN. *TITAN*, [cit. 2019-02-04]. Dostupné z: http://titan.thinkaurelius.com/

[25] The Benefits of Titan. *TITAN*, [cit. 2019-02-04]. Dostupné z: http://s3.thinkaurelius.com/docs/titan/1.0.0/benefits.html/

[26] Schumacher, R.: Introducing DataStax Enterprise Graph. *Datastax*, apríl 2016, [cit. 2019-02-04]. Dostupné z: https://www.datastax.com/2016/04/introducing-datastax-enterprise-graph

[27] Waples, J.: Visualizing JanusGraph: the new TitanDB fork. *Cambridge Intelligence*, december 2017, [cit. 2019-02-04]. Dostupné z: https://cambridge-intelligence.com/visualizing-janusgraph-new-titandb-fork/

[28] *JanusGraph*. [cit. 2019-02-08]. Dostupné z: http://janusgraph.org/

[29] Xu, Y.: INTRODUCING TIGERGRAPH, A NATIVE PARALLEL GRAPH DATABASE. *TigerGraph*, september 2017, [cit. 2019-02-05]. Dostupné z: https://www.tigergraph.com/2017/09/29/introducing-tigergraph-a-native-parallel-graph-database/

[30] Xu, Y.: IT IS TIME FOR A MODERN GRAPH QUERY LANGUAGE. *TigerGraph*, máj 2018, [cit. 2019-02-05]. Dostupné z: https://www.tigergraph.com/2018/05/22/its-time-for-a-modern-graph-query-language/

[31] GSQL: An SQL-Inspired Graph Query Language. *TigerGraph*, 2018, [cit. 2019-02-05]. Dostupné z: https://info.tigergraph.com/gsql

[32] Graph Database Benchmark Report. *TigerGraph*, 2018, [cit. 2019-02-05]. Dostupné z: https://www.tigergraph.com/benchmark/

[33] Building the database that powers today's strongest applications. *OrientDb*, 2017, [cit. 2019-02-06]. Dostupné z: https://orientdb.com/about-us/

[34] Staff, E.: 2018 is the Year of the Graph. *expero*, február 2018, [cit. 2019-02-05]. Dostupné z: https://www.experoinc.com/post/2018-is-the-year-of-the-graph

[35] Garulli, L.: *Distributed Architecture*. máj 2014, [cit. 2019-02-07]. Dostupné z: http://www.orientdb.com/docs/last/Distributed-Architecture.html

[36] Garulli, L.: *Sharding*. máj 2014, [cit. 2019-02-07]. Dostupné z: http://sorientdb.com/docs/3.0.x/distributed/Distributed-Sharding.html

[37] Why a Multi-Model Database? *OrientDb*, február 2018, [cit. 2019-02-07]. Dostupné z: http://orientdb.com/multi-model-database/

[38] Woodie, A.: ArangoDB Reaping the Fruits Of Its Multi-Modal Labor. *Datanami*, marec 2018, [cit. 2019-02-08]. Dostupné z: https://www.datanami.com/2018/03/28/arangodb-reaping-the-fruits-of-its-multi-modal-labor/

[39] *Storage Engines*. [cit. 2019-02-07]. Dostupné z: https://docs.arangodb.com/3.3/Manual/Architecture/StorageEngines.html

[40] How We Wronged Neo4j & PostgreSQL: Update of ArangoDB Benchmark 2018. *ArangoDB*, február 2018, [cit. 2019-02-08]. Dostupné z: https://www.arangodb.com/2018/02/how-we-wronged-neo4j-postgresql-update-arangodb-benchmark-2018/

[41] *Single Instance vs. Cluster*. [cit. 2019-02-08]. Dostupné z: https://docs.arangodb.com/3.3/Manual/Architecture/SingleInstanceVsCluster.html

[42] *Using SmartGraphs in ArangoDB*. [cit. 2019-02-08]. Dostupné z: https://www.arangodb.com/using-smartgraphs-arangodb/

[43] *Using SmartGraphs in ArangoDB*. [cit. 2019-02-08]. Dostupné z: https://www.arangodb.com/subscriptions/

[44] Plurad, J.: Community-Driven Graphs with JanusGraph. *IBM*, november 2017, [cit. 2019-02-11]. Dostupné z: https://www.slideshare.net/JasonPlurad/communitydriven-graphs-with-janusgraph-82388397

[45] Arif, T.: The Mathematics of Social Network Analysis: Metrics for Academic Social Networks. *International Journal of Computer Applications Technology and Research*, ročník 4, 11 2015: s. 889–893, doi: 10.7753/IJCATR0412.1003.

[46] Moreau, A.: How to Perform Fraud Detection with Personalized Page Rank. *Sicara's Vlog*, január 2019, [cit. 2019-04-30]. Dostupné z: https://blog.sicara.com/fraud-detection-personalized-page-rank-networkx-15bd52ba2bf6

[47] Configuring Apache Tinkerpop for Spark-Yarn. *Yet Another Analytics & Intelligence Communication Series*, jún 2017, [cit. 2019-05-09]. Dostupné z: http://yaaics.blogspot.com/2017/06/configuring-apache-tinkerpop-for-spark.html

[48] Configuring JanusGraph for Spark-Yarn. *Yet Another Analytics & Intelligence Communication Series*, júl 2017, [cit. 2019-05-09]. Dostupné z: http://yaaics.blogspot.com/2017/07/configuring-janusgraph-for-spark-yarn.html

Dodatok **A**

Zoznam použitých skratiek

GUI Graphical user interface

XML Extensible markup language

HDFS Hadoop distributed file system

YARN Yet Another Resource Negotiator

RDD Resilient Distributed Dataset

CRUD Create/read/update/delete

ACID Atomicity/Consitency/Isolation/Durability

NoSQL No SQL/Not only SQL

SQL Structured Query Language

SNA Social Network Analysis

Dodatok B

Obsah priloženého CD

```
readme.txt ........................ stručný popis obsahu CD
src
  impl ............................ zdrojové kódy implementácie
  thesis .......................... zdrojová forma práce vo formáte LaTeX
text .............................. text práce
  thesis.pdf ...................... text práce vo formáte PDF
  thesis.ps ....................... text práce vo formáte PS
```