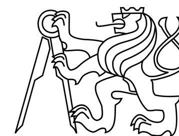


ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: Benchmark grafových databází pro pot eby data lineage
Student: Bc. Milan Ková
Vedoucí: Ing. Michal Valenta, Ph.D.
Studijní program: Informatika
Studijní obor: Webové a softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2018/19

Pokyny pro vypracování

1. Prove te pr zkum sou asného stavu grafových databází (GDB).
2. Seznamte se s požadavky na datové úložišt projektu Manta.
3. Na základ požadavk z bodu 2 sestavte benchmark specifický pro projekt Manta pro úlohu data lineage. Testovací databázi dodá zadavatel.
4. Na podmnožin GDB z bodu 1 implementujte benchmark a porovnejte výsledky.
5. Na základ výsledk vypracujte doporu ení pro novou GDB v projektu Manta.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Ji ina, Ph.D.
d kan

V Praze dne 13. íjna 2017

Diplomová práce

Benchmark grafových databází pro potřeby data lineage

Bc. Milan Kovář

Katedra softwarového inženýrství
Vedoucí práce: Ing. Michal Valenta, Ph.D.

8. ledna 2018

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu, Ing. Michalu Valentovi, Ph.D., za pomoc a cenné rady při tvorbě této práce. Dále bych chtěl poděkovat RNDr. Lukáši Hermannovi za jeho hodnotné rady a uvedení do kontextu nástroje Manta Flow. Hlavně bych však chtěl poděkovat své rodině za její podporu a trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 občanského zákoníku tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součást´ či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům), vč. možnosti Dílo upravit či měnit, spojit jej s jiným dílem a/nebo zařadit jej do díla souborného. Toto oprávnění je časově, teritoriálně i množstevně neomezené a uděluji jej bezúplatně.

V Praze dne 8. ledna 2018 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Milan Kovář. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Kovář, Milan. *Benchmark grafových databází pro potřeby data lineage*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem této práce je zanalyzovat momentální situaci na poli grafových databází a určit vhodnost jednotlivých možných řešení pro data lineage nástroj projektu Manta.

Klíčová slova Grafové databáze, Java, Titan, JanusGraph, OrientDB, ArangoDB, Cassandra, BerkeleyDB, Manta

Abstract

Main focus of this thesis is an analysis of the current state of graph database environemnt and potential suitability of individual solutions for data lineage tool of project Manta.

Keywords Graph database, Java, Titan, JanusGraph, OrientDB, ArangoDB, Cassandra, BerkeleyDB, Manta

Obsah

Odkaz na tuto práci vi

Úvod **1**

1 Cíl práce **3**

2 Databáze a Manta **5**

2.1 Databáze 5
2.1.1 Navigační databáze 5
2.1.2 Relační databáze 6
2.1.3 NoSQL databáze 7
2.2 Grafové databáze 10
2.3 Data lineage 11
2.4 Manta 12

3 Technologie **15**

3.1 TinkerPop 15
3.1.1 Gremlin 18
3.2 Grafové databáze 18
3.2.1 Titan 18
3.2.2 JanusGraph 20
3.2.3 OrientDB 20
3.2.4 ArangoDB 21
3.2.5 Neo4j 22
3.3 Indexovací nástroje 22
3.3.1 Lucene 22
3.3.2 Solr 23
3.4 Programovací jazyk 23
3.5 Shrnutí 23

4 Analýza a návrh **25**

4.1 Analýza požadavků 25
4.2 Datová struktura nástroje Manta Flow 25
4.2.1 Uzly 26
4.2.2 Hrany 28
4.2.3 Indexy 31
4.3 Návrh testů 33
4.3.1 Import 33
4.3.2 Merge import 34
4.3.3 Chunk import 35
4.3.4 Get Parent 36
4.3.5 Get Attributes 36
4.3.6 Get Children 36
4.3.7 Get Attribute by Name 36
4.3.8 Get Children by Name 37
4.3.9 Get All Parents 37
4.3.10 Get Resource 37
4.3.11 Get Vertices by Edge Type 37
4.3.12 Simple Graph Flow 37
4.3.13 Get Node by Name 38
4.3.14 Export databáze 38
4.3.15 Dump databáze 38
4.4 Shrnutí 38

5 Realizace 39
5.1 Cíle 39
5.2 Nástroje 40
5.3 Implementace 40
5.3.1 Vytváření a konfigurace databáze 40
5.3.2 Vytváření indexů 42
5.3.3 Import dat 44
5.3.4 Merge dat 45
5.3.5 Chunk import 46
5.3.6 Get Parent 46
5.3.7 Get Attributes 47
5.3.8 Get Children 47
5.3.9 Get Attribute by Name 48
5.3.10 Get Children by Name 49
5.3.11 Get All Parents 49
5.3.12 Get Resource 50
5.3.13 Get Vertices by Edge Type 50
5.3.14 Simple Graph Flow 51
5.3.15 Get Node by Name 51

6 Výkonnostní testování 53

6.1 Testovací prostředí 53
6.2 Testovací data 53
6.3 Výsledky testování 53
6.3.1 Import dat 54
6.3.2 Merge import 54
6.3.3 Chunk import 54
6.3.4 Get Parent 55
6.3.5 Get Attributes 55
6.3.6 Get Children 55
6.3.7 Get Attributes by Name 56
6.3.8 Get Children by Name 56
6.3.9 Get All Parents 57
6.3.10 Get Resource 57
6.3.11 Get Vertices by Edge Type 57
6.3.12 Simple Graph Flow 58
6.3.13 Get Node by Name 58
6.4 Diskuze nad výsledky 58
6.5 Doporučení 60

Závěr **63**

Literatura **65**

A Seznam použitých zkratek **71**

B Naměřené hodnoty **73**
B.1 Import dat 73
B.1.1 Titan (Persistit) 73
B.1.2 OrientDB 73
B.1.3 OrientDB Server 74
B.1.4 JanusGraph (Cassandra) 74
B.1.5 JanusGraph (Cassandra) Server 74
B.1.6 JanusGraph (BerkeleyDB) 75
B.2 Merge import 75
B.2.1 Titan (Persistit) 75
B.2.2 OrientDB 75
B.2.3 JanusGraph (Cassandra) 76
B.2.4 JanusGraph (BerkeleyDB) 76
B.3 Chunk import 76
B.3.1 Titan (Persistit) 76
B.3.2 OrientDB 77
B.3.3 JanusGraph (Cassandra) 77
B.3.4 JanusGraph (BerkeleyDB) 77
B.4 Get Parent 78

B.4.1 Titan (Persistit) 78
B.4.2 OrientDB 78
B.4.3 OrientDB Server 79
B.4.4 JanusGraph (Cassandra) 79
B.4.5 JanusGraph (Cassandra) Server 80
B.4.6 JanusGraph (BerkeleyDB) 80
B.5 Get Attributes 81
B.5.1 Titan (Persistit) 81
B.5.2 OrientDB 81
B.5.3 OrientDB Server 82
B.5.4 JanusGraph (Cassandra) 82
B.5.5 JanusGraph (Cassandra) Server 83
B.5.6 JanusGraph (BerkeleyDB) 83
B.6 Get Children 84
B.6.1 Titan (Persistit) 84
B.6.2 OrientDB 84
B.6.3 JanusGraph (Cassandra) 85
B.6.4 JanusGraph (BerkeleyDB) 85
B.7 Get Attributes by Name 86
B.7.1 Titan (Persistit) 86
B.7.2 OrientDB 86
B.7.3 JanusGraph (Cassandra) 87
B.7.4 JanusGraph (BerkeleyDB) 87
B.8 Get Children by Name 88
B.8.1 Titan (Persistit) 88
B.8.2 OrientDB 88
B.8.3 JanusGraph (Cassandra) 89
B.8.4 JanusGraph (BerkeleyDB) 89
B.9 Get All Parents 90
B.9.1 Titan (Persistit) 90
B.9.2 OrientDB 90
B.9.3 JanusGraph (Cassandra) 91
B.9.4 JanusGraph (BerkeleyDB) 91
B.10 Get Resource 92
B.10.1 Titan (Persistit) 92
B.10.2 OrientDB 92
B.10.3 JanusGraph (Cassandra) 93
B.10.4 JanusGraph (BerkeleyDB) 93
B.11 Get Vertices by Edge Type 94
B.11.1 Titan (Persistit) 94
B.11.2 OrientDB 94
B.11.3 JanusGraph (Cassandra) 95
B.11.4 JanusGraph (BerkeleyDB) 95
B.12 Simple Graph Flow 96

B.12.1 Titan (Persistit) 96
B.12.2 OrientDB 96
B.12.3 JanusGraph (Cassandra) 97
B.12.4 JanusGraph (BerkeleyDB) 97
B.13 Get Node by Name 98
B.13.1 Titan (Persistit) 98
B.13.2 OrientDB 98
B.13.3 JanusGraph (Cassandra) 99
B.13.4 JanusGraph (BerkeleyDB) 99

C Obsah přiloženého CD **101**

Seznam obrázků

2.1 Navigační databáze 6
2.2 CAP teorém pro databázi Titan[1] 9
2.3 Škálovatelnost NoSQL databází[1] 10
2.4 Příklad modelu grafové databáze 12

3.1 Struktura frameworku TinkerPop2 17

Seznam tabulek

3.1 Rozdíly mezi TinkerPop2 a TinkerPop3 17

6.1 Konfigurace testovacího prostředí 53
6.2 Vkládání nového grafu do databáze. 54
6.3 Vkládání grafu do zaplněné databáze databáze. 54
6.4 Vkládání grafu do databáze po částech. 55
6.5 Získání předka uzlu. 55
6.6 Získání všech atributů. 56
6.7 Získání všech potomků daného uzlu. 56
6.8 Získání všech atributů daného jména. 56
6.9 Získání všech potomků daného uzlu s určitým jménem. 57
6.10 Získání všech předků daného uzlu. 57
6.11 Získání uzlu typu Resource pro daný uzel. 57
6.12 Získání uzlů, spojených pomocí určitého typu hrany. 58
6.13 Získání uzlů, dostupných pomocí daného typu hrany. 58
6.14 Získání všech uzlů daného jména. 58

B.1 Výsledky měření pro Titan(Persistit), test Import dat. 73
B.2 Výsledky měření pro OrientDB, test Import dat. 73
B.3 Výsledky měření pro OrientDB server, test Import dat. 74
B.4 Výsledky měření pro JanusGraph (Cassandra), test Import dat. . . 74
B.5 Výsledky měření pro JanusGraph (Cassandra) server, test Import dat. 74
B.6 Výsledky měření pro JanusGraph (BerkeleyDB), test Import dat. . 75
B.7 Výsledky měření pro Titan(Persistit), test Merge import dat. . . . 75
B.8 Výsledky měření pro OrientDB, test Merge import dat. 75
B.9 Výsledky měření pro JanusGraph (Cassandra), test Merge import dat. 76
B.10 Výsledky měření pro JanusGraph (BerkeleyDB), test Merge import dat. 76

B.11 Výsledky měření pro Titan(Persistit), test Chunk import dat. . . . 76
B.12 Výsledky měření pro OrientDB, test Chunk import dat. 77
B.13 Výsledky měření pro JanusGraph (Cassandra), test Chunk import dat. 77
B.14 Výsledky měření pro JanusGraph (BerkeleyDB), test Chunk import dat. 77
B.15 Výsledky měření pro Titan(Persistit), test Get Parent. 78
B.16 Výsledky měření pro OrientDB, test Get Parent. 78
B.17 Výsledky měření pro OrientDB server, test Get Parent. 79
B.18 Výsledky měření pro JanusGraph (Cassandra), test Get Parent. . . 79
B.19 Výsledky měření pro JanusGraph (Cassandra) server, test Get Parent. 80
B.20 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Parent. . 80
B.21 Výsledky měření pro Titan(Persistit), test Get Attributes. 81
B.22 Výsledky měření pro OrientDB, test Get Attributes. 81
B.23 Výsledky měření pro OrientDB server, test Get Attributes. 82
B.24 Výsledky měření pro JanusGraph (Cassandra), test Get Attributes. 82
B.25 Výsledky měření pro JanusGraph (Cassandra) server, test Get Attributes. 83
B.26 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Attributes. 83
B.27 Výsledky měření pro Titan(Persistit), test Get Children. 84
B.28 Výsledky měření pro OrientDB, test Get Children. 84
B.29 Výsledky měření pro JanusGraph (Cassandra), test Get Children. 85
B.30 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Children. 85
B.31 Výsledky měření pro Titan(Persistit), test Get Attributes by Name. 86
B.32 Výsledky měření pro OrientDB, test Get Attributes by Name. . . . 86
B.33 Výsledky měření pro JanusGraph (Cassandra), test Get Attributes by Name. 87
B.34 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Attributes by Name. 87
B.35 Výsledky měření pro Titan(Persistit), test Get Children by Name. 88
B.36 Výsledky měření pro OrientDB, test Get Children by Name. 88
B.37 Výsledky měření pro JanusGraph (Cassandra), test Get Children by Name. 89
B.38 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Children by Name. 89
B.39 Výsledky měření pro Titan(Persistit), test Get All Parents. 90
B.40 Výsledky měření pro OrientDB, test Get All Parents. 90
B.41 Výsledky měření pro JanusGraph (Cassandra), test Get All Parents. 91
B.42 Výsledky měření pro JanusGraph (BerkeleyDB), test Get All Parents. 91
B.43 Výsledky měření pro Titan(Persistit), test Resource. 92
B.44 Výsledky měření pro OrientDB, test Get Resource. 92
B.45 Výsledky měření pro JanusGraph (Cassandra), test Get Resource. 93
B.46 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Resource. 93
B.47 Výsledky měření pro Titan(Persistit), test Vertices by Edge Type. 94

B.48 Výsledky měření pro OrientDB, test Get Vertices by Edge Type. . 94
B.49 Výsledky měření pro JanusGraph (Cassandra), test Get Vertices by Edge Type. 95
B.50 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Vertices by Edge Type. 95
B.51 Výsledky měření pro Titan(Persistit), test Simple Flow. 96
B.52 Výsledky měření pro OrientDB, test Simple Flow. 96
B.53 Výsledky měření pro JanusGraph (Cassandra), test Simple Flow. . 97
B.54 Výsledky měření pro JanusGraph (BerkeleyDB), test Simple Flow. 97
B.55 Výsledky měření pro Titan(Persistit), test Get Node by Name. . . 98
B.56 Výsledky měření pro OrientDB, test Get Node by Name. 98
B.57 Výsledky měření pro JanusGraph (Cassandra), test Get Node by Name. 99
B.58 Výsledky měření pro JanusGraph (BerkeleyDB), test Get Node by Name. 99

Úvod

Strukturovaná úložiště dat předcházejí vznik výpočetní techniky o řadu let. Systémy jako kartotéky a rejstříky byly užívané už v minulých stoletích. Byl to ale nástup a rozvoj výpočetní techniky, který umožnil jednak ukládání dat v objemech, které byly do té doby nemyslitelné, tak jejich získání ve zlomku času, který by k tomu potřebovala lidská obsluha úložiště.

Stejně, jako většina odvětví výpočetních technologií, jsou i databáze velice různorodé z technologického a implementačního hlediska a v průběhu jejich existence vznikla řada přístupů a modelů, jak s konkrétními problémy a požadavky na jejich řešení naložit. Od svého vzniku v sedmdesátých letech minulého století hrál prim model relační. Postupně se ale začaly objevovat případy užití, pro které relační model nebyl úplně optimální. Tento fakt vedl k rozvoji alternativních modelů, mezi které se řadí mimo jiné modely objektové, dokumentové a grafové.

Grafové databáze sice nejsou z hlediska historie nikterak novou technologií, ale v posledních několika desetiletích došlo k výraznému zvýšení jejich popularity a též četnosti užití. Mohou za to nové trendy v informatice, zejména rozšíření sociálních sítí jako komunikačního kanálu vývojářské komunity či nové metody analýzy datových toků. Ty výrazně těží z lepších vlastností a silných stránek grafových databází. Díky tomu se na trhu objevila řada nových řešení, které se snaží vývojářům daných služeb nabídnout produkty, plnící tyto nové požadavky.

Relativní mládí odvětví grafových databází (oproti databázím relačním) je ale důvodem toho, že dosud neexistuje obecně přijatý standard, který by všechny tyto produkty naplňovaly. Mnozí výrobci navíc přistupují k logicky stejné problematice různými způsoby a zavádějí i vlastní implementační (a proprietární) standardy. Tento fakt má za následek to, že migrace z jednoho produktu na druhý může být složitá či přímo nemožná bez výrazného zásahu do jejich kódu či vnitřních procesů. Mojí prvotní motivací v rámci této práce bylo prozkoumat existující grafové databáze, vhodné k použití v data lineage projektech a poskytnout porovnání výkonnosti, které může ulehčit rozhodo-

vání při výběru vhodného produktu.

Kapitola 1

Cíl práce

Použitý nástroj Manta Flow momentálně využívá k ukládání dat grafovou databázi Titan v kombinaci s backendem Persistit. Tyto technologie ale už nejsou nadále vyvíjeny, což sebou přináší problémy spojené s těžkou odstranitelností možných vnitřních chyb a ztěžuje přidání nových funkcionalit či zlepšení výkonu.

Hlavním cílem této práce je (v návaznosti na výše řečené) analýza dostupných grafových databází na trhu a určení vhodného kandidáta pro užívání v data lineage nástrojích, konkrétně pak v projektu Manta Flow od společnosti Manta Tools.

Rámec této práce zahrnuje následující témata:

- Seznámení s projektem Manta Flow, jeho datovou strukturou a požadavky
- Průzkum momentální situace na trhu grafových databázi
- Výběr podmnožiny vhodných řešení
- Návrh benchmarků, vycházejících z hlavních use cases nástroje Manta Flow a demonstrace výkonnosti a vhodnosti vybraných řešení na této podmnožině problémů
- Argumentované doporučení pro grafovou databázi, vhodnou pro nástroj Manta Flow

Kapitola 2

Databáze a Manta

V této kapitole je popsán princip grafových databází, aktuální situace na trhu s řešeními grafových databází, koncept data lineage a stručný popis nástroje Manta Flow.

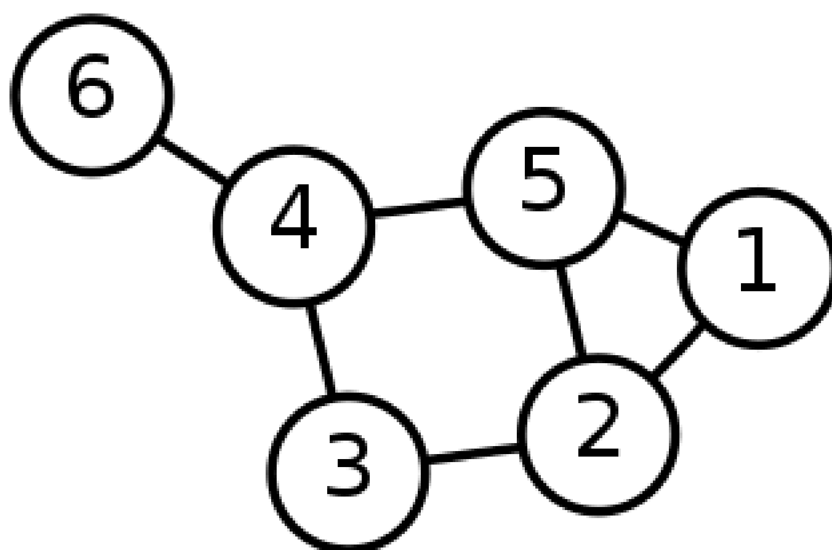
2.1 Databáze

Jak již bylo zmíněno v úvodu, databáze a systémy, ze kterých moderní databáze vznikly, slouží k ukládání, třídění a získávání kolekcí dat. Tato data byla nejdříve ukládána na papíru či jiném neelektronickém nosiči dat metodami administrativní práce (například kartotéky), ale s rostoucím objemem informací bylo nutné přijít s novými způsoby zpracování dat. Jako hlavní alternativa se dlouhou dobu jevil systém děrných štítků, se kterým v roce 1890 přišel Herman Hollerith pro sčítání lidu v USA [2].

První počítačové databáze se objevily v šedesátých letech minulého století, kdy se na tehdejší již dostatečně výkonné počítače přestalo nahlížet jako na pouhé velké kalkulačky sloužící primárně numerickým výpočtům ve vědecké sféře. Díky rozmachu užívání počítačů v komerční sféře, dostupnosti přímých datových úložišť a zájmu o uchování a zpracování komerčních informací silně vzrostl zájem o řešení, která by poskytovala funkcionality databází v počítačovém prostředí.

2.1.1 Navigační databáze

Navigační databáze byly mezi prvními, které se na trhu objevily. Jejich princip je založen na manuální „navigaci“ ve spojeném datovém seznamu, který tvoří rozsáhlou síť vzájemně pospojovaných uzlů. Lze prohlásit, že zde šlo o elementární základní graf, jak je patrné z obrázku 2.1. Získávání dat z tohoto typu databáze probíhalo pomocí primárního klíče, sekvenčním testováním všech záznamů a nebo „navigováním“ po grafu pomocí uživatelem vytvořené cesty. Tato cesta vznikla zřetězením názvů nebo adres uzlů za sebe[3]. V případě,



Obrázek 2.1: Navigační databáze

že cesta mezi danými uzly neexistovala, databáze vrátila chybovou zprávu. Tento přístup byl kodifikován skupinou vystupujícím pod označením „Data Base Task Group“ a byl nazýván „CODASYL“[4].

2.1.2 Relační databáze

Popularita navigačních databází začala postupně upadat v sedmdesátých a osmdesátých letech minulého století, kdy se na trhu objevilo alternativní řešení. To umožňovalo výrazně intuitivnější přístup k zacházení s databází a její obsluze. Tento přístup se nazýval relační databáze. Ideu, která vyústila ve vytvoření relačních databází, předložil Edgar Codd. Ten ve své práci „A Relational Model of Data for Large Shared Data Banks“[5] představil novou metodu ukládání dat v databázích. Data v tomto případě už nebyla pouze listem spojených záznamů, ale byla uložena ve strukturovaných tabulkách fixní šířky. Každý řádek pak korespondoval s jedním datovým záznamem, každý sloupec určoval typovou a sémantickou strukturu dat. Jednotlivé řádky byly mezi sebou propojovány převážně pomocí primárních klíčů. Relační databáze oddělily data od aplikací k nim přistupujících a umožnily získávání obsahu a manipulaci s těmito daty v podobě, která umožňovala uživatelům vyhledávat mezi daty skryté vztahy.

Datové modely, vytvářené v relačních databázích se podle Codda[5] měly řídit skupinou pravidel, která měla zajistit to, že data mohou být získávána a manipulována s pomocí univerzálního datového jazyka.

Tato pravidla byla tvořena následujícími požadavky:

1. Oprostit kolekce vztahů od nežádoucích závislostí, plynoucích z vložení, mazání a úprav
2. Omezit nutnost úprav struktury databáze při přidání nových datových typů a tím prodloužit životnost napojených programů
3. Učinit relační model přehlednější pro uživatele
4. Učinit kolekce vztahů nezávislé na dotazech

Nástroj, který tyto možnosti poskytoval, vznikl původně jako „Project R“[6] ve firmě IBM. Jednalo se o prvního předchůdce dnešního jazyka **SQL** - Structured Query Language. SQL vychází z matematických základů a relační algebry. Dotazy v něm vytvořené mohou využívat mimo jiné kartézský součin, projekci, sjednocení, agregace a transformace.

V moderním pojetí databází se postupně objevilo 6 normálních forem (označovaných jako *0NF* až *5NF*), které mohou databáze splňovat a které usnadňují algoritmizaci a formalizaci zadání. Nejčastěji vyskytujícími se formami jsou[7]:

- **0NF**: Atributy obsahují více než jednu hodnotu
- **1NF**: Atributy obsahují pouze jednu atomickou hodnotu
- **2NF**: Každý neklíčový atribut je silně závislý na celém primárním klíči
- **3NF**: Všechny neklíčové atributy tabulky jsou vzájemně nezávislé

Jednotlivé formy jsou vždy závislé na splnění podmínek všech předcházejících forem, které jsou s každým další stupněm stále striktnější a restriktivnější. Efektem implementace vyšších normálových forem je rozpad datového modelu do více tabulek. V praxi se nejvíce využívá třetí normální forma, která zaručuje, že databáze nebude obsahovat duplicitní data (menší náklady na uložení dat) a umožňuje jednoduše měnit uložené hodnoty záznamů. Vztahy mezi jednotlivými záznamy (řádky) jsou řešeny pomocí primárních a cizích klíčů, které se používají ke spojování dat pomocí příkazu JOIN.

2.1.3 NoSQL databáze

Relační databáze jsou stále nejpoužívanějším typem databází[8]. S akumulací více a více dat ve stále složitějších systémech ale začaly vyvstávat problémy a úlohy, na které relační model není úplně ideální. Důsledkem vyššího objemu zpracovávaných úloh je fakt, že se v praxi používají dotazy a pohledy na data se značným počtem operací JOIN. Ty mají za následek skutečnost, že přestože databáze splňuje všechny formální náležitosti, stejně tyto operace začínají být značně neefektivní z pohledu výkonnosti. Situace se ještě více zhoršuje v případě, kdy je kvůli objemu dat nebo infrastruktuře systému nutná

decentralizace databáze a její rozdělení mezi více fyzických strojů. V takovém případě je mimo faktický výpočet samotného složitého dotazu nutno přenést všechna potřebná data přes síť, což do procesu přidává další úzké hrdlo. To pak výrazně ovlivňuje rychlost provedení dotazu[9].

Odpovědí na tyto problémy se staly NoSQL databáze („Non SQL“ nebo také „Not only SQL“)[10]. Jedná se o databáze, které poskytují možnosti pro uchovávání a získávání dat pomocí jiných, než relačních způsobů. Takovéto databáze existovaly už v šedesátých letech minulého století (viz například 2.1.1), ale označení NoSQL se začalo používat až na přelomu dvacátého a jedenadvacátého století[11]. „Not only“ část v názvu tohoto druhu databází odkazuje na to, že mnohé z těchto databází podporují jazyky podobné SQL nebo přímo jeho nadmnožinu.

Jednou z vlastností, které činí NoSQL databáze atraktivní pro určité způsoby použití, je jejich dobrá podpora distribuovatelnosti. NoSQL databáze bez problému fungují zároveň na mnoha fyzicky oddělených strojích („cluster“) a umožňují tak ukládání dat v takových objemech, které by byly pro jednotlivé dedikované databázové stroje nemyslitelné. Tento fakt sebou ale nese i nevýhody v podobě porušení **ACID**[1] principu[9].

Dalším z principů, kterými se NoSQL databáze také řídí je **CAP** teorém[12] (viz obrázek 2.2). Ten tvrdí, že pro distribuované zdroje dat je nemožné splňovat více než dvě z následujících podmínek:

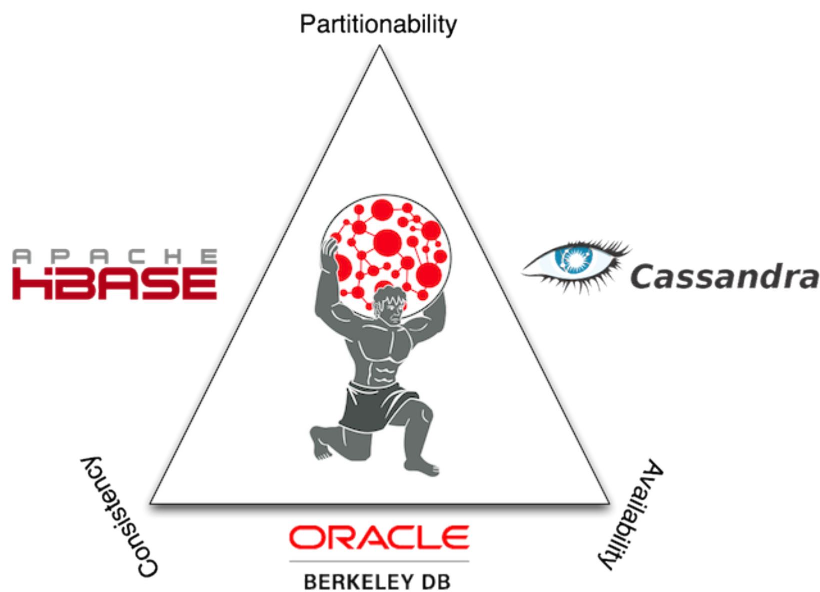
- **Konzistence** - Každé čtení vrátí výsledek posledního zápisu nebo chybu
- **Dostupnost** - Každé čtení vrátí nechybovou odpověď
- **Odolnost k rozdělení** - Systém funguje dál i v případě, že dojde ke zdržení čí ztrátě dat v rámci sítě

Většina NoSQL databází řeší tento problém omezením konzistence dat a to tak, že nabízí takzvanou „eventuální konzistenci“, díky které jsou data zapsána do všech uzlů eventuálně. To ale může vyústit v přečtení neaktuálních dat. Tomuto přístupu se také říka BASE - Basically Available Soft-state services with Eventual-consistency[13].

NoSQL databází existuje mnoho typů. Mimo jejich dělení na základě CAP teorému je také možné rozdělit je podle datových modelů, které používají[9]. Tyto modely jsou:

- **Sloupcový model**: Má řádky podobně jako u relační databáze, ale každý řádek má kolekci sloupců, které se skládají z dvojice klíč-hodnota, kde klíč je název sloupce. Sloupce mohou být pro každý řádek různé.
- **Model klíč-hodnota**: Jeden klíč, jedna hodnota uložená jako BLOB. Přístup k datům podle klíče přes hash tabulky výrazně zrychluje čtení. Neefektivní pro získávání pouze částí hodnoty.

[1] Atomicita, Konzistence, Izolovanost, Trvalost



Obrázek 2.2: CAP teorém pro databázi Titan[1]

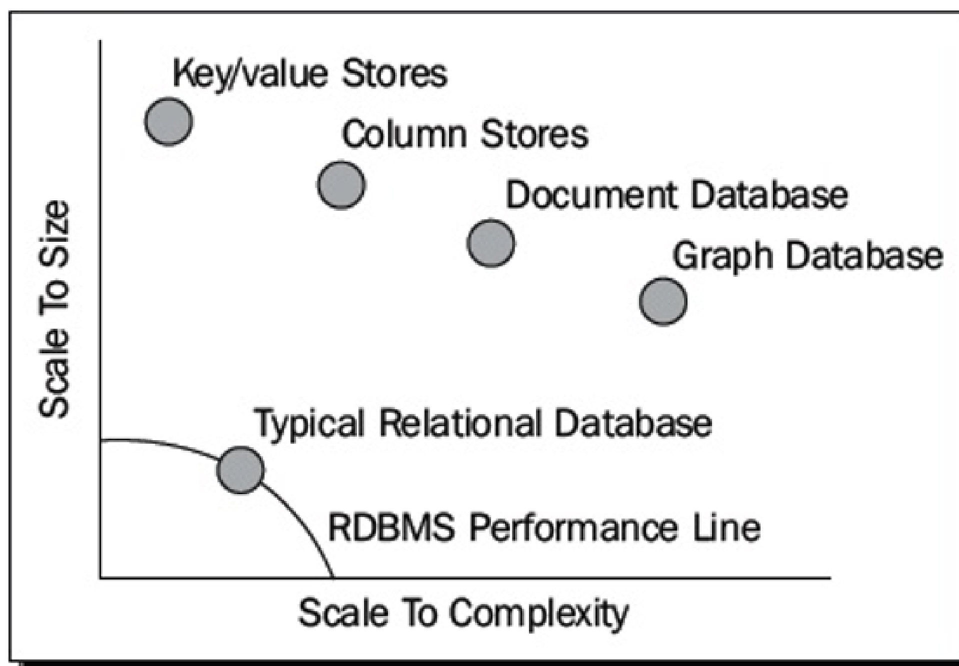
- **Dokumentový model**: Podobný jako model klíč-hodnota, ale hodnota je strukturovaná způsobem, o kterém má databáze přehled (JSON, XML). Umožňuje složitější dotazy nežli dotaz jen přes klíč.

- **Grafový model**: Datový model vycházející z teorie grafů (viz 2.2).

- **RDF model**: Jedná se o specializovaný model vycházející z grafového modelu. Data jsou uložená pomocí trojice („triple“) předmět-predikát-subjekt[14].

- **Multi model**: Databáze podporující multi model umožňují použití více výše zmíněných modelů v rámci jedné databáze[15].

Obrázek 2.3 demonstruje škálovatelnost velikosti a složitosti jednotlivých NoSQL přístupů v porovnání s relačními databázemi. To ale neznamená, že by se do budoucna přestaly relační databáze používat. Spíše poukazuje na jejich omezenou škálovatelnost v porovnání s NoSQL řešeními. Relační databáze zůstanou i nadále využívány v řadě systémů a řešení, zatímco NoSQL databáze budou využívány pro systémy, vynucující si distribuovanost a pracující s enormní velikostí dat, což jsou typicky systémy zabývající se problémy z oblasti Big Data.



Obrázek 2.3: Škálovatelnost NoSQL databází[1]

2.2 Grafové databáze

Grafové databáze se v průběhu let vyvinuly z navigačních databází (viz 2.1.1). Model grafových databází vychází z teorie grafů, kde data jsou reprezentována jako graf vztahů mezi entitami. Příklad jednoho takového grafového modelu je uveden na obrázku 2.4. Jednotlivými složkami modelu v tomto případě jsou:

- **Uzly**: Uzly v modelu reprezentují jednotlivé entity nebo záznamy.
- **Hrany**: Hrany (nebo také vztahy) spojují mezi sebou jednotlivé uzly. Reprezentují tak vztahy mezi entitami. Hrany mohou být orientované, obousměrné nebo bez orientace. Každá hrana může spojovat jen a pouze dva uzly. Hrana může dále obsahovat popis („label"), který blíže určuje, co daný vztah znamená.
- **Vlastnosti**: Vlastnosti reprezentují relevantní informace o entitách.

Oproti relačním databázím, ve kterých se pro dohledání jednotlivých záznamů či vztahů ve velkých objemech dat často musí používat indexy, grafové databáze díky své struktuře ukládají tyto informace pomocí hran. Proto není nutné prohledávat tabulky nebo aplikovat náročné JOIN operace, ale stačí pouze pomocí hrany přejít z jednoho uzlu do druhého[16].

Optimálním případem pro použití grafových databází jsou domény, silně závislé na vztazích[15]. Typickým příkladem jsou sociální sítě. Grafové databáze vynikají v dotazech typu „přátelé přátel“, ve kterých se daný problém dá jednoduše přeložit do průchodu grafem. Oproti tomu relační databáze budou výhodnější pro dotazy type „všichni dodavatelé z mimoevropských zemí s obratem vyšším než n“.

Grafové databáze se velice často používají ve spojení s dalšími NoSQL modely v rámci multi-modelových databází. Důvodem k tomu může být případ, kdy se daný datový model skládá z dat, jejichž podmnožiny nejsou jednoduše modelované jedním způsobem, kde díky tomuto přístupu nevzniká nutnost použití více nezávislých modelů[15].

Způsob fyzického uložení dat se může mezi jednotlivými grafovými databázemi značně lišit. Různé databáze používají různé NoSQL modely, jako je model klíč hodnota nebo dokumentový model[15]. V tomto smyslu tedy grafové databáze slouží jako další rozhraní nad těmito modely.

Vzhledem k relativnímu mládí tohoto typu NoSQL databázi neexistuje pro komunikaci s grafovými databázemi ucelený a obecně přijímaný jazyk, jako je SQL u relačních databází. Existuje několik jazyků, které se snaží o standardizaci, ale zatím (v době zpracování této práce) neexistuje žádný, který by byl podporován všemi významnými výrobci. Mezi tyto jazyky patří:

- **Cypher**: Jazyk byl původně vytvořený pro Neo4j, ale později byl uvolněn pro veřejnost. Jedná se deklarativní jazyk, který mimo dotazování umožňuje i vkládání, úpravu a mazání dat[17].

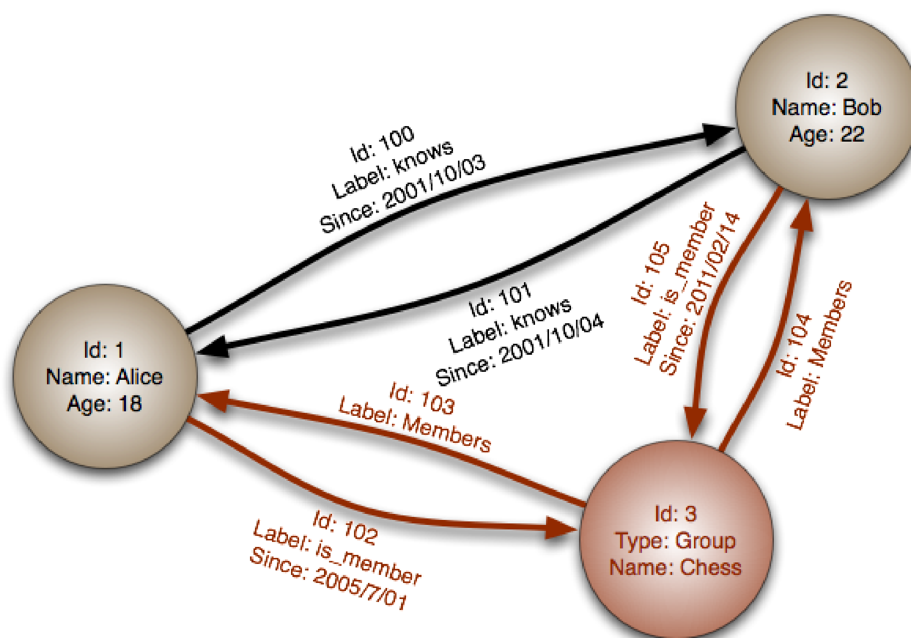
- **SPARQL**: Grafový jazyk, primárně zaměřený pro dotazování nad RDF NoSQL databázemi[18].

- **Gremlin**: Funkcionální jazyk, vytvořený pro práci s grafovými databázemi podporujícími framework TinkerPop (viz 3.1.1).

Řada grafových databází umožňuje mimo vlastních proprietárních grafových jazyků aplikacím třetích stran přístup k databázi pomoci zveřejněného API.

2.3 Data lineage

Data lineage je pojem zahrnující metodologie, spadající do oboru životního cyklu dat („data life cycle“). Data lineage procesy zkoumají, kde se nachází výchozí bod pro původ dat, zaznamenávají, jakými procesy za dobu jejich existence daná data projdou, a udržují informace o tom, jaké transformace tato data prodělala[19]. Tyto metodiky mohou výrazně pomoci s pochopením a analýzou toho, odkud specifická data pocházejí, jak se dané informace používají a jaká data mají vliv na výsledky vnitřních procesů[20].



Obrázek 2.4: Příklad modelu grafové databáze

Metodiky data lineage jsou úzce spojeny s oblastí bussiness inteligence, kde mohou napomoci tvorbě rozhodnutí na základě nasbíraných dat nebo dokonce odhalit chyby a nebezpečí a tím zlepšit firemní dohled nad daty („data governance“).

Vzhledem k tomu, že data nejsou statická, ale stále přibývají a transformují se, data lineage nástroje začínají být více a více důležité. V rámci systému velkých firem přibývají nové zdroje dat relativně často a začíná být čím dál tím složitější udržovat přehled o tom, jak se tato data chovají při prostupu různými firemními systémy[20].

Velkou výhodou data lineage nástrojů je možnost shromažďovat informace z více systémů a vizuálně je prezentovat uživateli v rámci jednoho programu. Díky tomu má uživatel jasně prezentovány datové toky, které v rámci jeho podniku existují, a snáze udrží přehled nad tím, k jakým manipulacím s daty dochází v rámci ETL procesů.

Mezi nástroji, které se zabývají metodikou data lineage, hraje významnou roli nástroj Informatica Metadata Manager nebo Manta Flow.

2.4 Manta

Manta Flow je data lineage nástroj vyvinutý společností Manta Tools s.r.o., známou jako Manta. Manta je dceřiná společnost české konzultační společnosti

Profinit s.r.o, od které se oddělila v roce 2013. Projekt, ze kterého výsledně vznikl nástroj Manta Flow původně vznikl v roce 2008 jako interní nástroj společnosti Profinit. Tento vývoj probíhal ve spolupráci s Fakultou Informačních Technologií ČVUT v Praze a za podpory Technologické Agentury České Republiky (TAČR), od které získal v období mezi roky 2013 a 2017 v rámci programů ALFA a EPSILON granty v celkové výši 1.49 milionů dolarů[21].

Nástroj Manta Flow se dostal do širšího mezinárodního povědomí po vítězství v soutěži „Czech ICT Incubator @ Silicon Valley“, kterou pořádala Czech ICT Alliance v roce 2014[22]. Po vítězství Manta založila v USA svojí první pobočku. Momentálně působí Manta celosvětově prostřednictvím vlastních zahraničních poboček a regionálních partnerů. Mezi významné zákazníky, užívající nástroj Manta Flow, patří společnosti Paypal, OBI, Vodafone nebo Comcast[23].

Hlavní devizou nástroje Manta Flow je automatická analýza programovacího kódu, napsaného v jazycích SQL nebo Java. V kontrastu k ostatním podobným řešením je nástroj unikátní díky své schopnosti rozpoznat i obtížně čitelný a na míru napsaný kód. Díky tomu pak nástroj dokáže v řádu hodin přečíst klientské databáze, čítající statisíce až miliony záznamů a sestavit z nich přehlednou mapu datových toků napříč klientským prostředím. Tato funkcionalita je v praxi užívána pro optimalizaci datových skladů, provádění dopadových analýz, dokumentování prostředí pro potřeby regulačních úřadů a v neposlední řadě i snižování nákladů na vývoj softwaru[24].

Kapitola 3

Technologie

V této kapitole jsou popsány technologie, které byly v rámci vypracování práce zkoumány a případně použity.

3.1 TinkerPop

TinkerPop je open-source framework, určený pro prácí s grafovými databázemi. Framework byl původně vyvinut jako proprietární nástroj pro projekt Titan, nyní je ale používán napříč celou řadou různých grafových databází[25].

První verze TinkerPopu (známá též jako TinkerPop0) vznikla v roce 2009 jako podpůrný projekt ve společnosti Aurelius a jejich grafové databáze Titan. Verze TinkerPop1 a TinkerPop2 vznikly v roce 2011, respektive 2012, a právě verzi TinkerPop2 používá pro správu své databáze nástroj Manta Flow. V roce 2015 přešel TinkerPop pod hlavičku Apache Software Foundation, která v roce 2016 vydala verzi TinkerPop3[25].

Samotný framework TinkerPop se ve verzi 2 skládal z několika nezávislých modulů, které pod sebou integroval. Jednotlivé moduly tvořily od sebe oddělené vrstvy, které bylo možné v rámci vlastního projektu použít libovolně dle potřeb vývojáře. Tyto moduly byly:

- **Blueprints**: Blueprints je kolekce rozhraní, implementací testovacích prostředí pro grafové modely dat. Na Blueprints je možné nahlížet jako na JDBC pro grafové databáze. Díky tomu poskytuje vývojáři sadu rozhraní, které může použít pro vytvoření backendu pro práci s grafovou databází. Program napsaný s pomocí Blueprints je navíc kompatibilní se všemi grafovými databázemi podporujícími TinkerPop. V ramci TinkerPop frameworku se jedná o podkladovou technologii pro všechny ostatní moduly[26].

- **Rexster**: Rexter je grafový server s podporou vyhledávání, doporučování a rankingu. Rexter vystavuje Blueprints grafy přes HTTP s po-

mocí RESTful koncových bodů nebo binárně pomocí RexPro protokolu. Rexter je značně rozšiřitelný pomocí Rexter Extensions a Gremlin konzole[27].

- **Furnace**: Furnace je balíček grafových algoritmů pro Blueprints grafy. Účelem modulu Furnace je umožnit použití existujících grafových algoritmů pro grafy s více-vztahovými uzly a vlastnostmi. Tento balíček mimo jiné obsahuje různé varianty jednotlivých grafových algoritmů, které jsou optimalizované pro různé scénáře (jeden stroj nebo distribuované prostředí)[28].

- **Frames**: Modul Frames poskytuje datovou vrstvu pro Blueprints grafy. Blueprints je reprezentován pouze jako kolekce vrcholů, vlastností a hran. S pomocí Frames je možné tyto tyto entity vystavit jako doménové objekty a jejich vztahy. Díky tomu může programátor pracovat přímo s objekty a nikoliv jednotlivými vrcholy. Jedná se tedy o objektové mapování[29].

- **Gremlin**: Gremlin je funkcionální jazyk vytvořený pro práci s grafovými databázemi podporujícími framework TinkerPop (viz 3.1.1)[30].

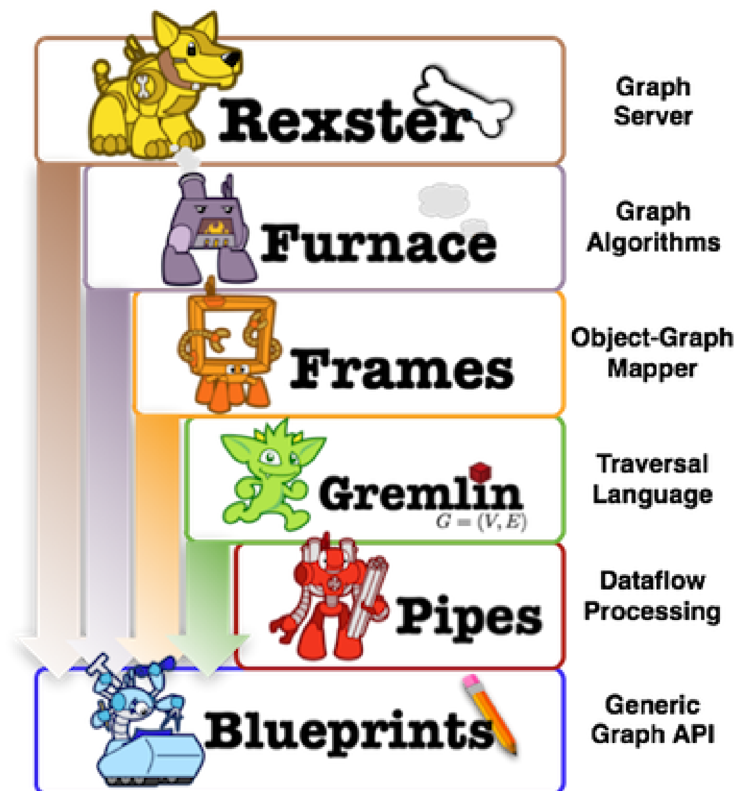
- **Pipes**: Pipes je dataflow framework pro procesové grafy. Umožňuje skládání Pipe uzlů (jeden výpočetní krok) a komunikačních hran do větších výpočetních celků. Takto vytvořený tok dat umožňuje všemožné transformace mezi vstupem a výstupem[31].

Celá struktura frameworku TinkerPop2 je znázorněna na obrázku 3.1.

S příchodem verze TinkerPop3 došlo ke značné změně jak ve struktuře frameworku, tak ve filosofii projektu[25]. Jednotlivé moduly byly přejmenovány (viz tabulka 3.1), spojeny a v důsledku toho jsou nyní obecně označovány jako Gremlin. Ačkoliv je snaha o konsolidaci tohoto projektu pod jednu jasně definovanou značku pochopitelná, tak tato změna sebou přinesla řadu komplikací. Grafové databáze vytvořené pod Blueprints z Tinkerpop2 nejsou vždy kompatibilní s Gremlinem (záleží na implementaci frameworku). Datová migrace mezi verzemi není nikterak náročná. Mnohem větším problémem je ale nekompatibilita Blueprints z Tinkerpop2 s novým Gremlin Structure API. API jednotlivých tříd bylo upraveno tak, aby více odpovídalo syntaxi Gremlina, ta ale vychází spíše z automatů a funkcionálních jazyků[32]. Tato změna má za následek fakt, že všechny programy, jejichž backend pro komunikaci s grafovými databázemi byl vytvořen s pomocí rozhraní, obsažených v Blueprints modulu, nejsou nově kompatibilní s TinkerPop3 a není možné je používat pro komunikaci s databázemi, implementujícími tento framework. Jednoduché řešení tohoto problému neexistuje a uživatelé TinkerPopu jsou tak postaveni do situace, kdy musí přepsat značnou část svého kódu do TinkerPop3, a nebo zůstat u používání starších a mnohdy už nepodporovaných verzích svých grafových databázi.

TinkerPop2	TinkerPop3
Rexster	GremlinServer
Furnace	GraphComputer a VertexProgram
Frames	Traversal
Gremlin	Gremlin
Pipes	GraphTraversal
Blueprints	Gremlin Structure API

Tabulka 3.1: Rozdíly mezi TinkerPop2 a TinkerPop3



Obrázek 3.1: Struktura frameworku TinkerPop2

3.1.1 Gremlin

Gremlin je funkcionální jazyk, uzpůsobený pro průchod po grafech v rámci TinkerPop frameworku a práci s datovými toky. Přitom umožňuje uživateli jednoduše vytvářet složité dotazy v rámci jejich grafového modelu[32]. Příkaz napsaný v jazyce Gremlin se skládá ze sekvence kroků, kde každý krok vykonává atomickou operaci na datech. Většina těchto kroků spadá do jedné z těchto kategorií:

- **Map**: Transformuje objekty, které se momentálně nacházejí na vstupu
- **Filter**: Vyfiltruje objekty, splňující určitou podmínku
- **SideEffect**: Vypočítá hodnotu z objektů na vstupu

Gremlin v sobě obsahuje knihovnu řady různých kroků, které může uživatel použít k vytvoření prakticky libovolného dotazu. Je tomu tak proto, že Gremlin je turingovsky kompletní jazyk[33].

Příkladem příkazu v jazyce Gremlin může být následující příkaz, který vrátí seznam jmen všech přátel uživatele jménem Gremlin.

```
g.V().has("name","Gremlin")
.out("friends")
.values("name")
```

3.2 Grafové databáze

Na základě analýzy možných řešení na poli grafových databází a konzultace požadavků k výslednému doporučení byly k dalšímu průzkumu pro účely této práce vybrány pouze takové grafové databáze, které implementují framework TinkerPop a které současně poskytují Java API aplikacím třetích stran.

3.2.1 Titan

Titan je open-source grafová transakční databáze, určena pro uchovávání grafových modelů s mohutností až stovky miliard uzlů, rozprostřených mezi více nezávislých zařízení („multi-machine clusters“). Podpora transakcí znamená, že se každá grafová operace vykonává v kontextu jedné transakce. Díky tomu je tak možné na jedné databázi provádět více paralelních dotazů ve stejnou chvíli[34].

Titan nemá dedikované interní úložiště dat. Místo toho používá pro uchovávání dat jiné databáze, které je možné k Titanu připojit. Díky této vlastnosti umožňuje svému uživateli výběr takové databáze, která bude nejvíce vyhovovat uživatelově potřebám. Další výhodou tohoto přístupu je fakt, že uživatel může jednoduše změnit úložnou databázi bez toho, aniž by musel jakkoliv

upravovat svůj kód. Jediné, co se v takovém případě mění, je konfigurace dané databáze[35].

Jednotlivá datová úložiště mohou být k Titanu připojena několika různými způsoby:

- **Embedded mode**: Pokud je dané úložiště připojené k Titanu pomocí embedded módu, znamená to, že jak Titan, tak databáze, běží ve stejném JVM a komunikují spolu pomocí procesních volání. Díky tomu odpadají náklady na přesun dat po síti a jejich následnou serializaci či deserializaci. Výsledkem je značné zlepšení výkonu[36].
- **Local server mode**: Databáze může být spuštěna jako samostatný proces, běžící na stejném stroji jako samotný Titan. V této konfiguraci mezi sebou databáze s Titanem komunikují pomocí síťového socketu[36].
- **Remote server mode**: V této konfiguraci běží Titan a databáze na rozdílných strojích. Titan si udržuje síťový socket na databázi a umožňuje tak koncovým aplikacím zároveň běžet na stejném JVM jako Titan a komunikovat s vzdálenou databází[36].
- **Remote server mode with Gremlin server**: Jedná se o variantu předcházejícího řešení, která instanci Titanu obalí Gremlin serverem. Díky tomu koncová aplikace nemusí být psaná v Javě, ale v jakémkoliv jiném (Gremlinem podporovaném) jazyku[36].

Omezení možností připojení je záležitostí jednotlivých úložišť, nikoliv Titanu.

Podporované databáze pro ukládání dat jsou:

- **Cassandra**: Apache Cassandra je NoSQL databáze, navržená pro uchovávání velkého množství dat napříč více stroji. Poskytuje podporu pro ukládání grafů o miliardách vrcholů a bilionech hran. Cassandra je známa svojí podporou replikace, která je považována za jednu z nejlepších v daném segmentu. V rámci propojení s Titanem umožňuje embedded, local server, remote server i gremlin server mód[36].
- **HBase**: Apache HBase je open-source NoSQL databáze, vytvořená po vzoru Google BigTable. HBase běží na HDFS a díky tomu nabízí velkou odolnost vůči chybám při ukládání velkého množství dat. V rámci propojení s Titanem umožňuje local server, remote server i gremlin server mód[37].
- **BerkeleyDB**: Oracle BerkeleyDB je komerční řešení společnosti Oracle, které poskytuje vysoce produktivní embedded úložiště pro data v modelu klíč-hodnota. BerkeleyDB běží ve stejném JVM jako Titan a tudíž vyžaduje, aby se všechna uložená data vešla na lokální disk. Díky tomu je omezena na grafy se řádově 10 až 100 miliony vrcholů, se kterými

ale dokáže pracovat velice rychle. Tato databáze neumožňuje spuštění v samostatném procesu či vzdáleném serveru[38].

Každá z těchto tří databází spadá do jedné z částí CAP teorému (viz 2.2).

Mimo výše zmíněné databáze podporuje Titan i uchovávání dat v proprietárním úložišti v paměti, které ale není nikterak optimalizováno a je doporučeno používat ho pouze pro testovací účely. Data v tomto úložišti zanikají v okamžiku ukončení procesu s databází[35].

Do verze 0.4.4 podporoval Titan i úložiště **Persistit**[2], které používá nástroj Manta Flow. Jedná se o embedded řešení, které je podobně jako BerkeleyDB omezeno velikostí lokálního disku a dostupnou pamětí. Díky své embedded podstatě dokáže ale běžet ve stejném JVM jako Titan a tím značně urychlit komunikaci mezi databází a Titanem.

Titan byl vyvíjen společností Aurelius, kterou ale v roce 2015 odkoupila společnost DataStax a získala tak celý vývojový tým nástroje Titan[39]. Zřejmě v důsledku toho se následně vývoj nástroje Titan zastavil.

3.2.2 JanusGraph

JanusGraph vznikl jako odvozený fork nástroje Titan po ukončení jeho vývoje. Ukončení se připisuje odkoupení vývojářského týmu společností DataStax[39]. Jedná se o škálovatelnou open-source transakční grafovou databázi[40].

Podobně jako Titan i JanusGraph nemá dedikované interní úložiště dat, pokud nepočítáme krátkodobé inmemory úložiště, sloužící primárně k testování. Jelikož se jedná o fork nástroje Titan, je s ním v mnohým věcech totožný. Proto pro podrobnější popis jeho vlastnosti viz 3.2.1.

Na rozdíl od nástroje Titan umožňuje JanusGraph navíc použití **Google Cloud Bigtable** jako datového úložiště[41]. Google Cloud Bigtable je NoSQL Big Data databázová služba od společnosti Google. Jedná se o stejnou databázi, kterou využívají stejné služby jako Google Search, Google Analytics, Google Maps nebo Gmail. Tato databáze je navržena pro silné zatížení a krátkou komunikační prodlevu. Bigtable využívá HBase rozhraní pro všechny svoje operace. Pro připojení k Bigtable je nutné JanusGraph nakonfigurovat podobně, jako pro HBase a danou konfiguraci doplnit o pár specifických nastavení[42].

Dalším rozdílem oproti nástroji Titan je fakt, že JanusGraph podporuje pouze TinkerPop3[43]. Proto je migrace z nástroje Titan na JanusGraph složitější, než by se na první pohled mohlo zdát.

3.2.3 OrientDB

OrientDB je open-source transakční NoSQL databáze. Jedná se o jednu z prvních masivněji rozšířených multi-model databázi. V rámci multi modelu pod-

[2]informace sdílena na GitHub stránce projektu - https://github.com/thinkaurelius/titan/issues/773

poruje grafový model, dokumentový model, model klíč-hodnota a objektový model. Hlavní ideou za tímto přístupem je skutečnost, že uživatel bude moci v rámci jedné databáze uchovávat různé druhy dat, které se hodí pro jednotlivé podporované modely, a nebude se muset starat o vhodnost vybraného řešení. Ve své dokumentaci OrientDB tvrdí, že na rozdíl od ostatních dostupných multi-model řešení, které často přidávají další API vrstvy pro multi-model sloužící jako rozhraní do jednotlivých databázových jader („engine“) je jádro OrientDB postavené tak, aby podporovalo všechny čtyři modely[15].

Oproti nástrojům Titan a JanusGraph OrientDB neumožňuje výběr z různých databázových úložišť. Namísto toho obsahuje vlastní proprietární úložiště, které se nemusí pro svoje použití nijak speciálně konfigurovat. Na druhou stranu OrientDB nabízí několik způsobů propojení s tímto úložištěm[44]:

- **plocal**: Paginated Local Storage. Jedná se o persistentní ukládání dat na disk v rámci stejného JVM, ve kterém běží OrientDB. Díky tomu odpadají náklady na přesun dat po síti a jejich následnou serializaci či deserializaci. Výsledkem je značné zlepšení výkonu.

- **remote**: Data jsou uložena na vzdáleném stroji, ke kterému se OrintDB připojuje pomocí socketu.

- **memory**: Všechna data jsou držena v paměti. Práce s nimi je tedy velice rychlá, ale nejedná se o persistentní uložení dat. Vhodné pro testování a pro případy, kdy je pro implementátora důležitý pouze výsledek dotazů.

Jednou z nových vlastností OrientDB je přidaná SQL vrstva, která umožňuje zadávání SQL příkazů i nad NoSQL databázi. Tím ulehčuje aklimatizaci uživatelům, navyklým na relační databáze. Vzhledem k tomu, že OrientDB obsahuje grafový model však nedochází k provádění operací JOIN, ale namísto toho je daný příkaz přeložen jako průchod grafem.

OrientDB existuje ve dvou edicích. Community edice je poskytována bezplatně a je udržována vývojářskou komunitou. Enterprise edice je placená a poskytuje navíc funkcionality, které nejsou obsaženy v community edici[45]. OrientDB podporuje jak TinkerPop2, tak TinkerPop3[43], díky čemuž je migrace z jiných grafových databází na OrientDB velice jednoduchá.

3.2.4 ArangoDB

ArangoDB (dříve známá také jako AvocadoDB) je multi-model NoSQL databáze podporující grafový model, dokumentový model a model hodnota-klíč. Podobně jako OrientDB i ArangoDB používá databázové jádro, které bylo od počátku navržené jako multi-modelové a nedochází tak vrstvení dodatečných rozhraní, která by následně komunikovala se svými databázovými jádry. Díky této vlastnosti je možné v ArangoDB data v jakékoliv podporované podobě a následně s nimi všemi pracovat v rámci jednoho dotazu[46].

ArangoDB nabízí dva možné způsoby ukládání dat[47]:

- **mmfiles**: Tradiční úložiště dat založené na paměťově mapovaných souborech („memory-mapped files“). Jedná se o výchozí datové úložiště. Tato varianta je vhodná pro data, která se vejdou do hlavní paměti. Důvodem k tomu je, že indexy se vytvářejí pouze v paměti a při opakovaném nahrání dat z disku dojde k tomu, že je nutné indexy znovu postavit. Tento návrh způsobuje delší časy na startu databáze, ale zároveň zrychluje většinu následných operací s indexy.

- **rocksdb**: Tato varianta je určena pro data, která zabírají více místa, než může obsáhnout hlavní paměť. Databáze si udržuje v paměti cache, kterou postupně plní požadovanými daty z disku. Indexy jsou v tomto případě ukládány na disku a tudíž není nutné je znovu vytvářet při startu databáze. To má za následek rychlejší start. S postupným zaplňováním cache v paměti se také postupně zvedá výkon databáze.

ArangoDB nabízí vlastní strukturovaný jazyk nazývaný ArangoDB Query Language (**AQL**). Jedná se o jazyk, který je svým zaměřením podobný SQL. AQL je deklarativní jazyk, který je nezávislý na prostředí nebo jazyku klienta. AQL umožňuje čtení a úpravu dat, ale neumožňuje operace měnící strukturu databáze. Ačkoliv syntaxe AQL není úplně shodná s SQL, sdílí spolu oba jazyky mnoho klíčových slov. Osvojení tohoto jazyka by tak uživateli, seznámenému s SQL nemělo dělat větší problémy[48].

Jako jedinou z vybraných databází není možné ArangoDB používat v embedded módu.

3.2.5 Neo4j

Neo4j je jednou z nejznámějších a nejpopulárnějších grafových databází. Jedná se o transakční a ACID splňující databázi. Jelikož se jedná o čistě grafovou databázi, jsou všechna data uložena jako uzel, hrana nebo atribut[49].

Mimo podpory jazyka Gremlin umožňuje Neo4j používání jazyka Cypher, který byl původně pro tuto databázi vyvinut. Neo4j podporuje framework TinkerPop ve verzi Tinkerpop2[26].

Problémem Neo4j je to, že je pro komerční účely zpoplatněn[50].

3.3 Indexovací nástroje

3.3.1 Lucene

Apache Lucene je open-source knihovna pro získávání informací z dat. Konkrétně se zaměřuje na full-textové indexování a vyhledávání v textech. V rámci grafových databází je často používána pro indexaci určitých atributů, přes které bude očekáváno nadprůměrné množství dotazů[51].

3.3.2 Solr

Apache Solr je open-source vyhledávací platforma postavená na Apache Lucene. Mimo základní full-textové vyhledávání poskytuje řadu dalších funkcionalit, které nacházejí užití zejména v podnikové sféře[52]. Solr je součástí řady enterprise řešení, jako například řady distribucí Hadoopu[53] nebo DataStax DSE[54].

3.4 Programovací jazyk

Jelikož je projekt Manta Flow implementovaný v jazyce Java a tato práce z daného projektu vychází, je výběr programovacího jazyka pro další části této práce jasně dán. Konkrétně se jedná o verzi 1.7, i když mnohé z výše zmíněných databází podporují už i Javu verze 1.8.

Java ovšem není jediným jazykem, který umožňuje práci s grafovými databázemi. Framework TinkerPop podporuje řadu dalších jazyků - Python, Ruby, Scalu, .NET-#C, Javasript (který je nativně podporován i OrientDB a ArangoDB), Go, PHP a další[43]. V důsledku toho není nutné soustřeďovat se pouze na Javu. V případě tvorby nového nástroje si tedy vývojář může vybrat technologii, která mu nejvíce vyhovuje nebo nejlépe zapadá do daného projektu, případně je nejkompatibilnější s firemními zvyklostmi a procesy.

3.5 Shrnutí

V této kapitole byli diskutování možní kandidáti pro grafovou databázi projektu Manta Flow. Na základě požadavků projektu Manta Flow, mezi které (kromě podpory frameworku TinkerPop a dostupnosti JAVA API) patří i požadavek na embedded řešení a vhodné licenční podmínky, budou v dalších částech práce použity následující grafové databáze:

- Titan
- JanusGraph
- OrientDB

Neo4j je diskvalifikován vysokými poplatky za komerční licenci a ArangoDB nabízí pouze remote řešení.

Pro full-text indexování uzlů bude využita knihovna Lucene. Jedná se o osvědčenou a v Manta Tools již používanou technologii. Pro implementaci bude použita Java verze 1.7.

Kapitola 4

Analýza a návrh

V této kapitole je popsána analýza požadavků této práce, datová struktura projektu Manta Flow a návrh vhodných testů.

4.1 Analýza požadavků

Hlavním požadavkem této práce je zmapovat momentální situaci na trhu řešení grafových databází, otestovat je a rozhodnout, zda momentálně existuje řešení, které by splňovalo všechny požadavky projektu Manta Flow, nabízelo dostatečný výkon a rychlost, případně poskytovalo garanci, že se daný projekt bude do budoucna rozvíjet a nedojde k jeho stagnaci tak, jak se stalo u momentálního řešení (viz 3.2.1).

V předchozí kapitole byla popsána možná technologická řešení, která splňují minimální požadavky pro použití v Manta Flow. Na konci dané kapitoly došlo k výběru podmnožiny řešení, která bude použita pro následující testování. Před tím, než ale bude přistoupeno k samotnému testování, je nutné důsledně pochopit strukturu dat, používaných v projektu Manta Flow, a na základě těchto dat vytvořit skupinu úloh, které budou nejlépe reprezentovat reálné použití daných dat v praxi.

4.2 Datová struktura nástroje Manta Flow

Data jsou v rámci projektu Manta Flow uložena v grafu. Tento datový model vychází z principu „vlastnostního“ grafu („property graph“). Jedná se o takový graf, který je podle frameworku TinkerPop[43] definován jako graf, mající následující vlastnosti:

- Principu klíč-hodnota - hodnoty atributů jednotlivých entit (uzlů i hran) jsou přístupné přes klíče
- Směrovost - každá hrana grafu má definovaný směr

- Více-vztahovovost - mezi dvěma různými uzly může existovat více hran

Příkladem „vlastnostního“ grafu může být graf na obrázku 2.4.

V této sekci jsou uvedeny informace, které vycházejí z dokumentace kódu projektu Manta Flow a z informací uvedených na interní Confluence firmy Manta Tools. Proto není možné některé z následujících částí textu ocitovat.

4.2.1 Uzly

V rámci teorie grafů a struktury dat v projektu Manta Flow se uživatel může setkat s dvojicí termínů „**node**“ a „**vertex**“. Oba dva tyto termíny se do české terminologie překládají jako „uzel“, ale z hlediska projektu Manta Flow se jedná o dva rozdílné termíny. „Vertex“ znamená obecný uzel grafu, kdežto „node“ označuje speciální typ uzlu. Proto pro přehlednost a snazší pochopení budou v následujícím textu jednotlivé druhy uzlů popisovány jako „uzel typu Node“ či „uzel typu Resource“, kdežto obecný uzel grafu („vertex“) bude označován jednoduše jako „uzel“.

Uzel v rámci grafu projektu Manta Flow reprezentuje určitou entitu. Uzel může mít libovolný počet vlastností. Každá tato vlastnost je určena svým jménem a v rámci její definice je možné určit i její datový typ. Různá řešení grafových databází se liší v tom, zda je možné definovat vlastní typy uzlů nebo ne, ale v rámci datového modelu nástroje Manta Flow je typ uzlu vždy určen jeho vnitřní vlastností.

Typy jednotlivých uzlů vyskytujících se v datové struktuře projektu Manta Flow jsou:

- **Node**: Uzel typu Node reprezentuje objekt v rámci toku dat prezentovaného projektem Manta Flow uživateli. Může se jednat o databázi, schéma, tabulku, sloupec, proceduru, skript, soubor, složku nebo některý z jiných uživatelem definovaných objektů.

  Tento typ uzlu má následující vlastnosti:

  – **Node type**: Tato vlastnost určuje, jaký objekt daný uzel reprezentuje (například databázovou tabulku).

  – **Node name**: Tato vlastnost určuje, jaký je název reprezentovaného objektu (například TABLE_USERS). Přes tuto vlastnost je také vytvořený fulltext index (viz 4.2.3).

- **Resource**: Uzel typu Resource reprezentuje specifickou technologii, ke které se vážou určitá metadata v grafu. Může se jednat o Oracle, Teradata, Informatica nebo o některou z dalších podporovaných technologií. Jedná se také o vstupní bod do podstromu dat, vázaných na danou technologii. Každý uzel typu Node je umístěn v podstromu některého z uzlů typu Resource.

  Tento typ uzlu má následující vlastnosti:

  - **Resource type**: Tato vlastnost určuje, jakou technologii daný uzel reprezentuje. Od jedné technologie může existovat více uzlů typu Resource (data byla nahrána například z více Oracle databází).
  - **Resource name**: Tato vlastnost určuje, jak se daný uzel jmenuje.

- **Attribute**: Uzel typu Attribute slouží k upřesnění a doplnění informací o uzlu typu Node, se kterým je ve vztahu (více viz 4.2.2).

  Tento typ uzlu má následující vlastnosti:

  - **Attribute name**: Tato vlastnost určuje, jaký je název atributu, který tento uzel reprezentuje. Pokud například asociovaný uzel typu Node reprezentuje sloupec databázové tabulky, tak tato vlastnost může reprezentovat vlastnost tohoto sloupce (například zda může daný sloupec nabývat hodnoty „null“).
  - **Attribute value**: Tato vlastnost určuje, jaká je hodnota tohoto atributu. Pokud opět použijeme příklad se sloupce tabulky, tak tato vlastnost může reprezentovat boolean hodnotu „true“.

- **Super Root**: Uzel typu Super Root reprezentuje hlavní kořen celého grafu. V rámci celého databáze existuje pouze jeden tento uzel. Na tento uzel jsou napojeny všechny uzly typu Resource.

  Tento typ uzlu obsahuje pouze jedinou vlastnost **Super Root**, která je unikátní a zaručuje, že nebude existovat žádný další uzel stejného typu.

- **Layer**: Uzel typu Layer reprezentuje logickou vrstvu dat uložených v databázi. Pomocí těchto uzlů je možné rozlišovat různé pohledy na uložená data a modely, které reprezentují. Uživatel tak má například možnost na stejnou množinu dat nahlížet z technického pohledu (databáze, tabulky,..) nebo z bussiness pohledu, kdy místo toků mezi tabulkami a databázemi sleduje toky mezi bussiness entitami, které tyto objekty reprezentují.

  Tento typ uzlu má následující vlastnosti:

  - **Layer name**: Tato vlastnost určuje, jak se daná vrstva bude nazývat.
  - **Layer type**: Tato vlastnost určuje, jakému typu daná vrstva náleží.

- **Revision Node**: Uzel typu Revision Node reprezentuje jednotlivé revize, kterými prošla data uložená v grafu. Díky tomu je možné v rámci projektu Manta Flow sledovat změny, kterými data prošla v čase.

  Tento typ uzlu má následující vlastnosti:

  - **Revision number**: Tato vlastnost určuje číslo dané revize.

  - **Revision commited**: Tato vlastnost určuje, zda byla příslušná revize provedena („commited“) nebo ne. Může nabývat pouze hodnot „true“ nebo „false“.
  - **Revision end**: Tato vlastnost určuje čas, kdy byla příslušná revize provedena.

- **Revision Root**: Uzel typu Revision Root reprezentuje hlavní kořen pro všechny uzly typu Revision Node, které na něj musí být napojeny. Podobně jako uzel typu Super Root může v databázi existovat pouze jeden uzel tohoto typu. Jedinou jeho možnou vlastností je opět analogicky vlastnost **Revision Root**, která je unikátní a zaručuje, že nebude existovat žádný další uzel stejného typu.

- **Source Node**: Uzel typu Source Node reprezentuje soubor, obsahující zdrojový kód. Většinou se jedná o soubory obsahující databázové skripty upravující data, která jsou v rámci projektu Manta Flow zpracovávána.

  Tento typ uzlu má následující vlastnosti:

  - **Source Node ID**: Tato vlastnost určuje unikátní náhodné generované označení, podle kterého je daný uzel v grafu identifikován.
  - **Source Node Local**: Tato vlastnost obsahuje jméno a cestu k souboru, který je tímto uzlem reprezentován.
  - **Source Node Hash**: Tato vlastnost obsahuje hash hodnotu obsahu souboru, který je tímto uzlem reprezentován. Tato vlastnost se používá pro kontrolu změn, které v reprezentovaném souboru mohly nastat.

- **Source Root**: Uzel typu Source Root reprezentuje hlavní kořen pro všechny uzly typu Source Node, které na něj musí být napojeny. Podobně jako uzel typu Super Root nebo Revision Root může v databázi existovat pouze jeden uzel tohoto typu. Jedinou jeho možnou vlastností je opět analogicky vlastnost **Source Root**, která je unikátní a zaručuje, že nebude existovat žádný další uzel stejného typu.

Z výše popsaných druhů uzlů je patrné, že výsledný datový model v projektu Manta Flow se skládá ze 3 na sobě nezávislých grafů, které nejsou mezi sebou nijak propojeny. Jedná se o hlavní graf, graf revizí a graf zdrojových uzlů.

4.2.2 Hrany

Hrany v rámci grafu projektu Manta Flow reprezentuje vztah mezi dvojicí uzlů v grafu. Hrana může mít podobně jako uzel libovolný počet vlastností určených jménem a hodnotou. Stejně jako u uzlů nabízejí různé grafové databáze možnost vytváření speciálních typů hran. Tento typ hrany se nazývá

štítek„label“. Základní podmínkou je to, že každá hrana smí mít pouze jeden štítek.

Pomocí štítku rozlišuje v datové struktuře projektu Manta Flow následující druhy hran:

- **Parent edge**: Hrana typu Parent spojuje mezi sebou dva uzly typu Node. Jedná se o jednosměrnou hranu, která reprezentuje vztah potomka s předkem. V rámci hierarchie grafu vede z uzlu směrem k jeho předkovi. Každý uzel typu Node má právě jednu takovouto hranu. Příkladem může být vztah dvou uzlů, kdy uzel reprezentující databázi je předkem uzlu reprezentujícího tabulku.

  Tento typ hrany má následující vlastnosti:

  – **Child Name**: Tato vlastnost určuje jméno uzlu (potomka), ze kterého daná hrana vychází. Tato vlastnost je indexována pro rychlejší vyhledávání v grafu (viz 4.2.3).
  – **Tran Start**: Tato vlastnost určuje číslo revize, ve které byl daný objekt poprvé přidán do grafu. Tato hodnota se v průběhu existence objektu nemění.
  – **Tran End**: Tato vlastnost obsahuje číslo poslední revize, ve které byl daný objekt přidán do grafu. Tato hodnota se mění při každém dalším přidáním objektu do grafu.

- **Resource edge**: Hrana typu Resource spojuje mezi sebou uzel typu Node a uzel typu Resource nebo uzel typu Resource a uzel typu Super Root. Jedná se o jednosměrnou hranu vycházející z v první případě z uzlu typu Node a v druhém případě z uzlu typu Resource. V rámci potomků daného uzlu typu Node mají většinou všichni potomci hranu na stejný Resource, ale nemusí to být pravidlem.

  Tento typ hrany má stejné vlastnosti jako hrana typu Parent.

- **Attribute edge**: Hrana typu Attribute spojuje uzel typu Attribute s uzlem typu Node. Jedná se o jednosměrnou hranu vycházející z uzlu typu Node, která reprezentuje vztah objekt-vlastnost.

  Tento typ hrany má vlastnosti **Tran Start** a **Tran End**, které byly již popsány výše.

- **Direct edge**: Hrana typu Direct spojuje mezi sebou dva listové uzly typu Node. Jedná se o jednosměrnou hranu reprezentující přímý tok dat mezi dvěma objekty. Může jít například o reprezentaci faktu, že daný sloupec tabulky je parametrem v určité proceduře. V rámci tohoto toku mohou být data upravena.

  Tento typ hrany má následující vlastnosti:

  - **Target ID**: Tato vlastnost určuje vygenerované databázové ID cílového uzlu hrany. Tato vlastnost je indexována pro urychlení vyhledávání v grafu (viz 4.2.3).
  - **Interpolated**: Tato vlastnost určuje, zda byla daná hrana interpolována nebo ne. Interpolace hrany znamená to, že daná hrana v dané vrstvě byla odvozena na základě datových toků z jiné vrstvy.

  Mimo zmíněné vlastnosti má tento druh hrany i vlastnosti **Tran Start** a **Tran End**, které byly již popsány výše.

- **Filter edge**: Hrana typu Filter spojuje mezi sebou dva listové uzly typu Node. Jedná se o jednosměrnou hranu reprezentující nepřímý tok dat mezi dvěma objekty. Pod nepřímým tokem dat je zde chápán fakt, že zdrojový uzel ovlivňuje, jaká data a za jakých podmínek budou téci do cílového uzlu. Příkladem takového vztahu může být IF podmínka v určitém skriptu.

  Tento typ hrany má stejné vlastnosti jako hrana typu Direct.

- **Revision edge**: Hrana typu Revision spojuje mezi sebou uzel typu Revision Node a uzel typu Revision Root. Jedná se o jednosměrnou hranu, vycházející z uzlu typu Revision Root a směřující do uzlu typu Revision Node. Tato hrana nemá žádné další vlastnosti a slouží pouze k udržení dostupnosti všech nahraných revizí.

  Tento typ hrany má vlastnosti **Tran Start** a **Tran End**, které byly již popsány výše.

- **Source edge**: Hrana typu Source spojuje mezi sebou uzel typu Source Node a uzel typu Source Root. Jedná se o jednosměrnou hranu, vycházející z uzlu typu Source Root a směřující do uzlu Source Node. Tato hrana slouží pouze k udržení dostupnosti všech uzlů typu Source Node.

  Tento typ hrany má vlastnost **Source Local Name**, která určuje název souboru se zdrojovým kódem z cílového uzlu. Tato vlastnost je indexována pro rychlejší vyhledávání pro zrychlení vyhledávání v grafu. Mimo to obsahuje vlastnosti **Tran Start** a **Tran End**, které byly již popsány výše.

- **Layer edge**: Hrana typu Layer spojuje mezi sebou uzel typu Resource a uzel typu Layer. Jedná se o jednosměrnou hranu vycházející z uzlu typu Resource, která specifikuje, do jaké „vrstvy“ data asociovaná s daným uzlem Resource náleží.

  Tento typ hrany má vlastnosti **Tran Start** a **Tran End**, které byly již popsány výše.

- **MapTo edge**: Hrana typu MapTo spojuje mezi sebou dva listové uzly typu Node z různých „vrstev“. Jedná se o jednorozměrnou hranu, která reprezentuje fakt, že cílový uzel je mapovaný na koncový uzel v jiné „vrstvě“.

  Tento typ hrany má vlastnosti **Tran Start** a **Tran End**, které byly již popsány výše.

4.2.3 Indexy

Pod indexem je v rámci grafových databází chápána datová struktura, která napomáhá k zrychlení přístupu k datům z databáze. Indexy fungují na základě principu klíč-hodnota, kdy pro daný klíč umožňuje rychle dohledat hodnoty k němu náležející. Existuje více druhů indexů a v rámci datového modelu nástroje Manta Flow budou popsány následující indexy:

- **Standardní index**: Standardní indexy se mohou používat k indexování vlastností jak uzlů, tak hran. Vzhledem k tomu, že se jedná o vrozenou vlastnost jednotlivých databází, není nutné databázi nijak speciálně konfigurovat pro jejich použití. Standardní indexy se mohou použít jak pro rychlejší vyhledávání entit splňující jimi definované vlastnosti, tak pro zaručení unikátnosti určitých vlastností databáze. Pomocí unikátního indexu lze například garantovat, že se v daném grafu bude vyskytovat uzel s danou vlastnosti pouze jednou, což se v případě nástroje Manta Flow používá pro zajištění unikátnosti jednotlivých kořenů.

  Různé grafové databáze nabízejí vlastní implementace indexů, kdy k základním vlastnostem přidávají vlastni funkcionality, které mohou uživatelům pomoci lépe strukturovat jejich data. Jedním z takových příkladů může být například složený index.

  Jedním z omezení standardního indexu je fakt, že pro mapování hodnot je nutná kompletní shoda s definicí indexu. Tento nedostatek je ale možné obejít pomocí použití externích indexů.

  V rámci datového modelu nástroje Manta Flow jsou definovány následující standardní indexy:

  - **Child Name**: Index vytvořený nad hranami grafu. Index využívá hranovou vlastnost „childName“ pro rychlejší dohledání potomka daného uzlu na základě hodnoty vlastnosti „Node name“ potomka.
  - **Target ID**: Index vytvořený nad hranami grafu. Index využívá hranovou vlastnost „targetID“ pro rychlejší dohledání cílového uzlu dané hrany na základě unikátního ID cílového uzlu.
  - **Super Root**: Unikátní index vytvořený nad uzly grafu, který slouží k rychlému dohledání uzlu typu Super Root.

- **Source Root**: Unikátní index vytvořený nad uzly grafu, který slouží k rychlému dohledání uzlu typu Source Root.
- **Revision Root**: Unikátní index vytvořený nad uzly grafu, který slouží k rychlému dohledání uzlu typu Revision Root.
- **Source Local Name**: Index vytvořený nad hranami grafu. Index využívá hranovou vlastnost „sourceLocalName“ pro rychlejší dohledání uzlu typu Source Node na základě hodnoty jeho vlastnosti „sourceLocalName“.

• **Vertex centric index**: Vertex centrix indexy jsou indexy, vlastní jednotlivým uzlům grafu. Na rozdíl od standardních nebo externích indexů, které existují globálně nad celou databází a slouží k rychlému vyhledání určitých uzlů, vertex centric indexy existují lokálně nad jednotlivými uzly.

  Vertex centric indexy slouží k rozdělení hran, asociovaných s daným uzlem podle hodnoty určité vlastnosti hrany. S jistou měrou zjednodušení lze tvrdit, že každý uzel má přehled o tom, s jakými typy hran je ve vztahu a jaké jsou hodnoty vlastností těchto hran bez toho, aniž by bylo nutné všechny dané hrany procházet a tyto vlastnosti testovat. Tím dochází k značnému zrychlení vyhledávání dalších uzlů s ním spojeným, které je patrné hlavně pří průchodu grafem. Tento typ indexu je zvláště vhodný pro grafy, ve kterých uzly mají velké množství asociovaných hran.

  Přístup k vytváření vertex centric indexů se liší v závislostí na databázi.

  V rámci datového modelu nástroje Manta Flow jsou definovány následující vertex centric indexy:

  - **Has Parent**: Index vytvořený nad vlastností „childName“ hran grafu typu Parent.
  - **Has Resource**: Index vytvořený nad vlastností „childName“ hran grafu typu Resource.
  - **Direct Flow**: Index vytvořený nad vlastností „targetID“ hran grafu typu Direct.
  - **Filter Flow**: Index vytvořený nad vlastností „targetID“ hran grafu typu Filter.
  - **Has Source**: Index vytvořený nad vlastností „sourceLocalName“ hran grafu typu Source.

• **Externí index**: Externí indexy jsou indexy, pro jejichž použití je nutné použít nástroje třetích stran. Aby bylo možné tyto indexy nad danou databází použít, je nutné, aby ona databáze daný nástroj podporovala.

Následně je nutné upravit nastavení databáze. Jedním z takových externích indexovacích nástrojů je Lucene, který je používán i v rámci nástroje Manta Flow.

Externí indexy nabízejí rozšířenou funkcionalitu oproti standardním indexům. Mimo indexování pozice uzlu v grafu a indexování hodnot v určitém rozsahu umožňují externí indexy také fulltextové vyhledávání, kdy nad danou množinou hodnot dokáží velice rychle zasáhnout hledaný text. Další užitečnou funkcionalitou některých externích indexovacích nástrojů je schopnost vyhledávat i takové textové hodnoty, které úplně neodpovídají požadované hodnotě, ale určitým způsobem z ni vycházejí. Mezi způsoby, kterými tohoto dosahuje je například takzvaný „stemming“, pomocí kterého jsou od kořene slova odděleny předpony, přípony nebo koncovky, nebo odstranění takzvaných „stop-words“ (většinou se jedná o předložky, členy a podobná slova) v případě víceslovného dotazu.

V rámci datového modelu nástroje Manta Flow jsou definovány následující externí indexy:

- **Node Name**: Externí fulltext index vytvořený nad uzly grafu. Tento index využívá vlastnost uzlu „nodeName“ pro rychlejší vyhledání uzlů s daným jménem v grafu.
- **Trans Start**: Externí index vytvořený nad hranami grafu. Index využívá hranovou vlastnost „transStart“ a umožňuje intervalové vyhledávání nad prvním číslem revize.
- **Trans End**: Externí index vytvořený nad hranami grafu. Index využívá hranovou vlastnost „transEnd“ a umožňuje intervalové vyhledávání nad nejnovějším číslem revize.

4.3 Návrh testů

Pro účely zhodnocení vhodnosti vybraných databázových řešení je nutné provést důkladné výkonnostní testování. Po konzultaci se zadavatelem práce a analýze potřeb a častých scénářů nástroje Manta Flow byly navrženy následující testy.

4.3.1 Import

První věcí, kterou je nutné práci s daty v rámci nástroje Manta Flow udělat, je tato data do databáze nahrát. V případě prvotního nahrání dat do prázdné databáze se jedná o postupné načítání dat z CSV souborů reprezentující jednotlivé entity v grafu. Každá entita je v rámci CSV souboru identifikována svým proprietárním ID, které umožňuje udržovat vztahy mezi entitami v různých CSV souborech. Toto načítání probíhá v následujícím pořadí:

1. **Resource**: Nejdříve jsou nahrány uzly typu Resource.
2. **Node**: Následně jsou nahrány uzly typu Node. V rámci tohoto nahrávání dochází ke kontrole existence asociovaného uzlu typu Resource a uzlu typu Node, který je v rámci definice záznamu uveden jako předek daného uzlu. Mezi těmito uzly je následně vytvořena hrana typu Parent, která demonstruje hierarchický vztah těchto dvou uzlů. Stejně tak je vytvořena hrana typu Resource na asociovaný uzel typu Resource.
3. **Node Attribute**: Po přidání uzlů typu Node je nutné doplnit jejich atributy. Pro každý atribut je vytvořen uzel typu Attribute, u kterého se kontroluje existence asociovaného uzlu typu Node. Následně je mezi těmito uzly vytvořena hrana typu Attribute.
4. **Edge**: Ve chvíli, kdy byly v databázi nahrány všechny uzly, je možné databázi doplnit o hrany typu Direct a Filter. Před přidáním dané hrany se ověří existence jak zdrojového, tak koncového uzlu hrany. Až poté dochází k přidání samotné hrany.

Výsledkem tohoto testu je databáze s grafem reprezentujícím načtená data.

4.3.2 Merge import

Merge import je speciálním případem nahrávání dat do databáze. Uživatel má možnost do databáze data nahrávat opakovaně, aby model, se kterým nástroj Manta Flow pracoval co nejvíce odpovídal aktuálnímu stavu. V takové situaci je nutné zajistit, aby v databázi nevznikaly duplicity. Jednou z možností je starý model smazat a na základě nově poskytnutých dat vytvořit nový, který by se nahrál stejným způsobem jako v minulém testu. Tímto způsobem by se ale ztratily informace o změnách, které v daném modelu nastaly. Proto je nutné způsob nahrávání upravit tak, aby se předešlo ztrátě informací a zabránilo se možným duplicitám. Postup pro tento způsob nahrávání je následující:

1. **Resource**: Pro každý načtený uzel typu Resource zkontroluje, zda v dané databázi již neexistuje. Tato kontrola probíhá tak, že získáme všechny potomky uzlu typu Super Root a následně zkontrolujeme, zda-li daný uzel v databázi již neexistuje. Tato kontrola probíhá pomocí porovnání vlastností „Resource type“ a „Resource name“ vůči načteným hodnotám. Pokud v databázi již existuje uzel, který má shodnou dvojici vlastností, načtený uzel nebude nahrán. V opačném případě je vytvořen nový uzel typu Resource s načtenými hodnotami.
2. **Node**: Pro každý načtený uzel typu Node zkontrolujeme, zda v dané databázi již neexistuje. Tato kontrola probíhá pomocí „přirozeného klíče“, jelikož způsob, kterým identifikujeme záznamy v rámci CSV souboru nemá v rámci databáze žádný význam. Tento klíč získáme tak, že nejdříve

nalezneme předka načteného uzlu. Jelikož se graf modelu staví hierarchicky od vrchu, existuje jistota, že takový uzel v databázi existuje (pokud takový uzel neexistuje, jedná se o vadná data). Následně získáme všechny jeho potomky - všechny uzly, které jsou s předkem spojeny pomocí hrany typu Parent. Obdobně jako u kontroly duplicity pro uzel typu Resource zkontrolujeme, zda v kolekci získaných potomků neexistuje uzel, jehož vlastnosti „Node name“ a „Node type“ se rovnají načteným hodnotám. Pokud takový uzel neexistuje, je nutné ještě před jeho přidáním zkontrolovat, že uzel typu Resource, se kterým je nově přidávaný uzel ve vztahu opravdu existuje. Až poté je možné daný uzel nahrát do databáze. Pokud ale takový uzel typu Node existuje, ještě to neznamená, že se jedná o duplicitu. V tomto případě zbývá ještě jeden krok, a to kontrola, zda je načtená hodnota asociována se stejným uzlem typu Resource jako nalezený uzel typu Node. Až v této chvíli lze rozhodnout, že se jedná o shodné uzly.

V případě přidání načteného uzlu následně vytvoříme hrany stejným způsobem, jako tomu bylo u předchozího testu.

3. **Node Attribute**: Pro každý načtený uzel typu Attribute zkontrolujeme, zda daný atribut asociovaný s daným uzlem typu Node už v databázi neexistuje. Nejdříve je nutné zkontrolovat existenci asociovaného uzlu typu Node. Následně si načteme všechny uzly typu Attribute, které jsou s daným uzlem spojeny pomocí hrany typu Attribute. Následně podobně jako u kontroly existence uzlu typu Resource zkontrolujeme, zda se v kolekci získaných atributů nevyskytuje takový, jehož vlastnosti „Attribute name“ a „Attribute value“ se rovnají načteným datům. Pokud žádný takový uzel nenalezneme, je možné přidat nový uzel typu Attribute a vytvořit potřebou vazbu.

4. **Edge**: Pro každou načtenou hranu je nutné ověřit, zda se již v databázi nevyskytuje. Tato kontrola probíhá tak, že si nejdříve z databáze získáme počáteční a koncový uzel hrany a ověříme si jejich existenci. Pokud oba dva uzly existují, je nutné ještě zkontrolovat, zda mezi nimi hrana stejného typu již neexistuje. Pouze pokud taková hrana neexistuje, je možné ji do databáze přidat.

Výsledkem tohoto testu je databáze obsahující aktuální model dat a zároveň držící informace o předchozí verzi.

4.3.3 Chunk import

Chunk import je dalším speciálním typem vkládání dat do databáze. Na rozdíl od předchozích dvou případů, kdy byl do databáze vkládán celý datový model, dochází v tomto případě k postupnému přidávání dat po malých inkrementál-

ních částech, které reprezentují rozšiřování původního datové modelu o nové entity a vztahy.

Důležité při vkládání dat tímto způsobem je to, aby daná část dat v kombinaci se všemi předchozími částmi, které byly už do databáze vloženy, tvořila validní datový model.

Samotný import dat probíhá stejným způsobem, jako při předchozím merge importu.

4.3.4 Get Parent

Na rozdíl od předchozích testů, které testovaly schopnosti a vlastnosti databáze při vkládání dat, je tento test založen na schopnosti data z databáze číst. Cílem tohoto testu je otestovat schopnost databáze pracovat s nadefinovanými hranami. Proto se pro určitý uzel typu Node v grafu pokusíme najít uzel, který je v rámci hierarchie datového modelu jeho předkem.

Výsledkem tohoto testu je získání uzlu typu Node, který je opravdu předkem uzlu typu Node na vstupu.

4.3.5 Get Attributes

Podobně jako u minulého testu i u tohoto testujeme čtecí vlastnosti databáze a její zacházení s nadefinovanými hranami a uzly. Cílem tohoto testu je pro určitý uzel typu Node najít všechny uzly typu Attribute, které jsou s ním spojeny pomocí hrany typu Attribute.

Výsledkem tohoto testu je získání kolekce uzlů typu Attribute, které jsou asociovány s uzlem typu Node na vstupu.

4.3.6 Get Children

Poslední základní test pro ověření čtení z databáze. Cílem tohoto testu je pro určitý uzel typu Node najít všechny uzly typu Node, ze kterých do daného uzlu vychází hrana typu Parent.

Výsledkem tohoto testu je získání kolekce uzlů typu Node, které jsou přímými potomky uzlu typu Node na vstupu.

4.3.7 Get Attribute by Name

V rámci tohoto testu bude testována schopnost databáze filtrovat získané hodnoty z dotazu bez toho, aniž by daná vlastnost byla indexovaná.

Výsledkem tohoto testu je získání kolekce hodnot, které odpovídají hodnotám atributů daného jména asociovaných s uzlem typu Node na vstupu.

4.3.8 Get Children by Name

V rámci tohoto testu bude testována efektivita databáze při práci s vertex centric indexem „Has Parent“, který by měl optimalizovat vyhledávání potomku daného uzlu typu Node na základě vlastnosti „Child Name“ hrany typu Parent.

Výsledkem tohoto testu je získání uzlu typu Node, který je přímým potomkem s daným jménem uzlu typu Node na vstupu.

4.3.9 Get All Parents

Tento test je prvním případem, kdy je testována efektivita databáze pro průchod grafem. Cílem tohoto testu je projít graf od určitého uzlu typu Node přes hrany typu Parent až do jeho nejvyššího předka (uzlu typu Node, který nemá svého předka).

Výsledkem tohoto testu je kolekce uzlů typu Node, které tvoří hierarchickou strukturu od kořene po daný uzel na vstupu.

4.3.10 Get Resource

V rámci tohoto testu bude opět testován průchod grafem. V tomto případě je nutné nalézt uzel typu Resource, který je asociován s daným uzlem typu Node.

Výsledkem tohoto testu je uzel typu Resource, který je asociován s uzlem typu Node na vstupu.

4.3.11 Get Vertices by Edge Type

Cílem tohoto testu je získání všech sousedících uzlů, které jsou s daným uzel spojeny hranou určitého typu. Primárně se jedná o získání uzlů spojených hranami typu Direct nebo Filter. Díky tomu se v rámci tohoto testu testuje efektivita práce databáze s vertex centric indexy „Direct Flow“ a „Filter Flow“.

Výsledkem tohoto testu je kolekce uzlů typu Node, které jsou spojeny s daným uzlem hranami typu Direct nebo Filter.

4.3.12 Simple Graph Flow

Tímto testem je také testována efektivita práce databáze s vertex centric indexy „Direct Flow“ a „Filter Flow“ a schopnost databáze přes tyto hrany rychle traverzovat. Cílem tohoto průchodu grafu je nalezení všech možných uzlů typu Node, do kterých je možné dostat se přechodem z uzlu do uzlu pomocí hran typu Direct nebo Filter.

Výsledkem tohoto testu je kolekce uzlů typu Node, která reprezentuje podgraf všech uzlů, které je možné s výchozím uzlem spojit pomocí opakovaného přechodu přes hrany typu Direct nebo Filter.

4.3.13 Get Node by Name

Poslední test na práci s uzly testuje efektivitu práce databáze s externím indexem „Node Name“. Cílem tohoto testu je nalezení všech uzlů typu Node, které mají určitou hodnotu vlastnosti „Node Name“.

Výsledkem tohoto testu je kolekce uzlů typu Node, které mají hodnotu vlastnosti „Node Name“ rovnou hodnotě na vstupu.

4.3.14 Export databáze

Cílem tohoto testu je kompletní export všech entit v databázi do CSV souborů. Bude tak testována schopnost databáze poskytovat obsažená data ve snadno dostupné a zpracovatelné formě.

Výsledkem tohoto testu je kolekce CSV souborů reprezentující všechny potřebné entity z databáze. Opětovným zpracováním těchto souborů by vznikla databáze totožná s databází exportovanou.

4.3.15 Dump databáze

Cílem tohoto testu je získání momentálního stavu databáze v binární formě a její opakované nahrání z této reprezentace.

Výsledkem tohoto testu je binární soubor reprezentující stav databáze v době získání dat. Opětovným zpracováním tohoto souboru by vznikla databáze totožná s databází exportovanou.

4.4 Shrnutí

V této kapitole byla zanalyzována datová struktura projektu Manta Flow. Na základě této analýzy byla navržena množina testů, kterou budou implementovány a použity pro výkonnostní testování zvoleny databázových řešení.

Vzhledem k formě a obsahu testů nebude v rámci realizace implementována datová struktura projektu Manta Flow ve své úplnosti, jelikož některé typy uzlů, hran a indexu do testování nijak nezasahují.

Kapitola 5

Realizace

V této kapitole je popsána realizace nástroje sloužícího pro testování výkonnosti grafových databází, konfigurace jednotlivých řešení a implementace testů navržených v předchozí kapitole.

5.1 Cíle

Cílem této realizační části je vytvořit nástroj, který bude snadno použitelný pro testování funkcionality a výkonnosti různých řešení grafových databází. Dalším důležitým požadavkem je snadná rozšiřitelnost implementovaných funkcionalit o nové testy, které by do budoucna mohly sloužit k hodnocení výkonnosti a vhodnosti daných grafových databází v rámci jiných scénářů, než které byly navrženy v sekci 4.3.

K implementaci rozhraní mezi vytvořeným nástrojem a jednotlivými grafovými databázemi byl použit framework TinkerPop. Jak bylo popsáno v sekci 3.1, framework TinkerPop se nyní používá ve dvou nekompatibilních verzích. Vzhledem k tomu, že zkoumané nástroje vyžadují různé verze tohoto frameworku, bylo nutné přistoupit k určitým ústupkům od původního sjednoceného plánu. Původním záměrem bylo vytvořit nástroj, který bude zcela universální pro všechna použitá řešení. Prvním problémem, bránícím realizaci tohoto záměru byl značný rozdíl mezi API verzí TinkerPop3 a TinkerPop2. Tento nedostatek by se sice dal relativně snadno překonat (za cenu dramatického zhoršení čitelnosti a přehlednosti výsledného kódu), daleko větším problémem se ale ukázaly závislosti důležitých knihoven, které je nutné použít pro práci s jednotlivými grafovými databázemi. Proto nakonec bylo přijato rozhodnutí tento projekt rozdělit na dva moduly, z nichž první umožňuje práci s databázemi prostřednictvím framework TinkerPop2 a modul druhý, podporující framework TinkerPop3. Díky tomu sice došlo k implementaci duplicitních funkcionalit, ale díky dobrému návrhu testovacího prostředí šlo často většinu kódu použít opakovaně s minimem úprav, které se týkaly specifických vlastností jednotlivých frameworků.

5.2 Nástroje

Pro implementaci výsledného nástroje byla použita Java verze 1.7 z důvodů popsaných v sekci 3.4. Jako IDE byl použit „Spring Tool Suite“ ve verzi 3.9, ale vzhledem k tomu, že oba dva projekty nejsou postaveny na žádné z částí frameworku Spring, je možné pro práci s nimi použít jakékoliv jiné IDE s podporou nástroje Maven.

5.3 Implementace

V této části budou nastíněny nejdůležitější části implementace jednotlivých funkcionalit. Vzhledem k faktu, že je nutné pracovat s více verzemi API a dochází tak k duplikaci některých částí kódu, bude v této sekci uváděn relevantní kód pouze jednou a případné změny způsobené rozdílnosti API nebo různým přístupem jednotlivých databází budou doplněny v co nejkonkrétnější podobě.

5.3.1 Vytváření a konfigurace databáze

Před tím, než je možné s grafovou databází jakkoliv pracovat, je nutné vytvořit instanci této databáze. V rámci vytváření dané databáze může uživatel specifikovat mód, ve kterém vytvořená instance poběží, nastavit přístupové údaje a nakonfigurovat zacházení se zdroji, strategii nahrávání a mnohé další vlastnosti databáze.

Jelikož tvorba a nastavení databáze nijak nespadá pod framework TinkerPop, jednotlivá řešení grafových databází přistupují k těmto krokům z různých stran. OrientDB umožňuje uživateli nastavení databáze pomocí speciálního výčtového typu `OGlobalConfiguration`, díky kterému je možné nastavit hodnoty pro jednotlivá nastavení v průběhu běhu instance dané databáze. Mimo tento způsob umožňuje OrientDB konfiguraci databáze pomocí XML souboru s nastavením databáze nebo pomocí argumentů z příkazové řádky při spuštění JVM[55].

Uživatel může chování databáze za běhu dále ovlivnit pomocí dodatečných nástrojů, které OrientDB nabízí. Mezi tyto nástroje patří například možnost vypnout validaci vkládaných dat nebo deklarace záměru uživatele databáze, která umožňuje za běhu změnit vnitřní nastavení databáze tak, aby lépe reflektovalo potřeby následujících úkonů. Tato dodatečná nastavení jsou použita při plnění databáze daty.

```
//...
//Username a Password neni pri plocal pripojeni vyzadovano
OrientGraph internalGraph = new OrientGraph("plocal:" + dbPath);
OGlobalConfiguration.WAL_SYNC_ON_PAGE_FLUSH.setValue(false);
//dočasné nastavení databáze pro vložení velkého množství dat
internalGraph.declareIntent(new OIntentMassiveInsert());
```

```
//vypne validaci vkládaných dat
internalGraph.getRawGraph().setValidationEnabled(false);
//...
```

Titan a JanusGraph mají díky své příbuznosti podobnou filosofii vytváření a konfigurace instancí databáze. Obě řešení používají pro vytváření nových instancí „factory“ třídu, jejíž metoda pro vytvoření instance přijímá jako argument cestu k `.properties` souboru s konfigurací, nebo objekt třídy `Configuration`, který v sobě obsahuje mapu s názvy a hodnotami jednotlivých nastavení[56][57].

Pomocí této konfigurace tak může uživatel specifikovat, které datové úložiště použije pro ukládání dat. Jednotlivá úložiště se dají dále nastavit pomocí svých specifických možností konfigurace. Mimo tyto možnosti, závislé na jednotlivých technologiích, existuje řada nastavení, které upravují grafovou databázi jako celek. Podobně jako u OrientDB je i zde možné pro urychlení vkládání do databáze upravit validaci dat, určit rozsah načítaných dat, nebo upravit práci se zdroji. Titan a JanusGraph sdílejí řadu podobných možností nastavení, ale nejedná se o totožné možnosti.

V případě, že je použito uložiště Cassandra, je nutné nakonfigurovat ještě zvlášť tento nástroj. Tato konfigurace probíhá pomocí speciálního `.yaml` souboru, jehož umístění se specifikuje v rámci konfigurace instance grafové databáze. Tento soubor umožňuje přímé nastavení možností daného úložiště, které JanusGraph pomocí své konfigurace nedovoluje[58].

V následujícím kódu je vidět příklad vytvoření a konfigurace instance JanusGraph.

```
//...
configuration.setProperty("storage.directory", dbPath);
configuration.setProperty("storage.backend", "embeddedcassandra");
//toto nastavení vypne validaci dat při vkládání
configuration.setProperty("storage.batch-loading", "true");
//při čtení uzlu či hrany získej i jejich vlastnosti
configuration.setProperty("query.fast-property", "true");
//specifické nastavení pro Cassandru
configuration.setProperty("storage.cassandra.compression", "true");
configuration.setProperty
("storage.cassandra.replication-factor", "1");
configuration.setProperty
("storage.cassandra.read-consistency-level", "ONE");
configuration.setProperty
("storage.cassandra.compaction-strategy-class",
    "LeveledCompactionStrategy");
configuration.setProperty
("storage.cassandra.write-consistency-level", "ONE");
configuration.setProperty("storage.conf-file",
    "file:\\cassandra.yaml");
//nastavení cache vlastností databáze
configuration.setProperty("cache.db-cache", "true");
```

```
configuration.setProperty("cache.db-cache-size", "0.6");
configuration.setProperty("cache.db-cache-time", "0");
//vytvoření instance databáze
JanusGraph internalGraph = JanusGraphFactory.open(configuration);
//...
```

5.3.2 Vytváření indexů

Vytváření indexů není stejně jako konfigurace databáze nijak standardizováno a jednotlivá řešení k němu přistupují různými způsoby.

Řešení pro Titan a JanusGraph se v tomto případě rozcházejí významně víc, než v případě konfigurace, ale přesto mají některé aspekty podobné. Pro použití externího indexu je nutné v rámci konfigurace databáze určit příslušný backend a umístění pro tento index.

```
// Doplneni indexovaciho enginu
configuration.setProperty("storage.index.INDEX_NAME.backend",
    "lucene");
configuration.setProperty("storage.index.INDEX_NAME.directory",
    dbPath + "/searchindex");
```

Povšimněte si řetězce „INDEX_NAME“ ve jméně nastavení dané konfigurace. Jedná se o název externího indexu, který bude v rámci aplikace používán. Samotné vytvoření externího indexu vypadá následovně:

```
// Titan - vytvoření externího indexu
internalGraph.makeKey(MainConfig.NODE_NAME)
                    .dataType(String.class)
                    .indexed("INDEX_NAME", Vertex.class,
                        Parameter.of(Mapping.MAPPING_PREFIX,
                        Mapping.TEXT))
                    .make();
//...
// JanusGraph - vytvoření externího indexu
JanusGraphManagement mgmt = internalGraph.openManagement();
PropertyKey nodeName = mgmt.makePropertyKey(MainConfig.NODE_NAME)
                .dataType(String.class).make();
mgmt.buildIndex("nodeName", Vertex.class).addKey(nodeName,
    Mapping.TEXT.asParameter()).buildMixedIndex("INDEX_NAME");
```

Pro vytvoření indexů je nutné v rámci databáze definovat jednotlivé vlastnosti („properties“) a štítky („labels“), se kterými se bude následně z pohledu datového modelu pracovat. V následujícím kódu je prezentováno vytvoření vertex centric indexu pro hranu typu Parent. Ostatní indexy jsou vytvořeny obdobně.

```
// Titan - vytvoření indexu
```

```
TitanKey childKey =
    internalGraph.makeKey(MainConfig.EDGE_CHILD_NAME)
                    .dataType(String.class)
            .indexed(Edge.class)
            .make();
internalGraph.makeLabel(MainConfig.EDGE_PARENT_LABEL)
        .sortKey(childKey).make();
//...
// JanusGraph - vytvoření indexu
EdgeLabel parLabel =
    mgmt.makeEdgeLabel(MainConfig.EDGE_PARENT_LABEL).make();
PropertyKey childName =
    mgmt.makePropertyKey(MainConfig.EDGE_CHILD_NAME)
        .dataType(String.class).make();
mgmt.buildEdgeIndex(parLabel, "parentChild", Direction.OUT,
    childName);
```

Vytváření indexů pro OrientDB má stejnou podmínku pro definici vlastností a štítků. Na rozdíl od předcházejících dvou řešení má ale OrientDB ještě jednu podmínku, a to fakt, že indexy se na databázi mohou vytvářet pouze mimo transakci. Vytvoření indexu tedy vypadá následovně:

```
// ...
internalGraph.executeOutsideTx(new OCallable<Object,
    OrientBaseGraph>() {
        @Override
    public Object call(OrientBaseGraph iArgument) {
        OrientEdgeType parentType = internalGraph
        .createEdgeType(MainConfig.EDGE_PARENT_LABEL);
        OProperty parentProp = parentType
        .createProperty(MainConfig.EDGE_CHILD_NAME, OType.STRING);
        parentProp.createIndex(INDEX_TYPE.FULLTEXT);
//...
```

Vytvoření externího indexu je mírně složitější, jelikož v některých konfiguracích OrientDB existuje bug, kdy při použití JAVA API není externí index vytvořen. Tento problém lze ale obejít pomocí spuštění vlastního SQL příkazu, který požadovaný index vytvoří. I v tomto případě je nutné vlastnost, nad kterou je index vytvořen, nejdříve definovat.

```
// ...
nodeType.createProperty(MainConfig.NODE_NAME, OType.STRING);
internalGraph.command(new OCommandSQL("create index Node.name ON
    V(" + MainConfig.NODE_NAME + ") FULLTEXT ENGINE
    LUCENE")).execute();
//...
```

5.3.3 Import dat

Prvním krokem pro import dat je načtení jednotlivých entit z CSV souborů. Toto načítání je momentálně nastaveno tak, aby dokázalo pracovat s formátem poskytnutých vzorových dat. V případě, kdy by bylo nutné použít data formátovaná jiným způsobem, je možné upravit zdroj jednotlivých atributů v načteném řádku pomocí změny hodnot konstant v hlavní konfigurační třídě `MainConfig`.

Data pro uložení do databáze jsou rozdělena do čtyř souborů. Každý soubor reprezentuje jeden druh entit. Jako první jsou načítány uzly typu Resource, po nichž následují uzly typu Node, uzly typu Attribute a nakonec hrany typu Direct a Filter. Jednotlivé vztahy mezi vkládanými entitami jsou reprezentovány pomocí proprietárních ID, která ale nenesou z hlediska databáze smysl. Jelikož jednotlivé grafické databáze přiřazují svým entitám vlastní identifikační údaje, je nutné v rámci vkládání dat udržovat překladovou tabulku mezi vlastními ID a ID entit v databázi. Kvůli tomu vznikla třída `Translator`, která udržuje jak tuto překladovou tabulku ID, tak tabulku referencí na všechny entity přidané v aktuální transakci, čímž urychluje případné získávání dat.

Pro vkládání dat byl vytvořen interface, který odděluje proces importu dat od jednotlivých atomických databázových operací. Důvodem tohoto oddělení je fakt, že ačkoliv jednotlivá řešení implementují standard frameworku TinkerPop pro tyto operace, tak v některých případech nad tímto standardem dále staví a umožňují optimalizaci celého procesu. Příkladem takové situace je funkcionalita OrientDB, která umožňuje vytvoření daného uzlu se všemi jeho vlastnostmi[59]. Tím dochází k urychlení celého procesu, jelikož není nutné daný uzel několikrát updatovat. Díky tomuto rozdělení zůstává celý proces importu dat pro všechny databáze totožný.

Prvními přidanými entitami jsou uzly typu Resource, které jsou pomocí hrany typu Parent navázány na uzel typu Super Root. Následně dochází k přidání uzlu typu Node, u kterých se při vkládání testuje existence jejich předka a příslušného uzlu typu Resource v databázi. Následně je i tento uzel spojen s grafem pomocí hrany typu Parent. Přidání uzlů typu Attribute funguje obdobně, ale místo kontroly existence předka se kontroluje existence asociovaného uzlu typu Node. Jako poslední dochází k vložení hran typu Direct a Filter. Před vložením každé této hrany dochází ke kontrole existence výchozího a cílového uzlu.

Vkládání jednotlivých typů uzlů a hran je velice podobné. Hlavním rozdílem jsou nastavované vlastnosti. V tomto příkladu je implementováno vkládání uzlu typu Node:

```
// ...
Map<String, String> properties = new HashMap<>();
properties.put(MainConfig.NODE_NAME, parts[MainConfig.NODE_I_NAME]);
properties.put(MainConfig.NODE_DESC, parts[MainConfig.NODE_I_DESC]);
```

```
properties.put(MainConfig.NODE_TYPE, MainConfig.VERTEX_NODE_TYPE);
Vertex node = db.addVertexWithProperties(properties);
//...
String parentString = trans.get(parts[MainConfig.NODE_I_PARENT]);
parentNode = db.getVertex(parentString);
if (parentNode != null) {
        Edge edge = db.addEdge(node, parentNode,
            MainConfig.EDGE_PARENT_LABEL);
        //Add child name attribute to Edge (for indexing purposes)
    db.setEdgeProperty(edge, MainConfig.EDGE_CHILD_NAME,
        parts[MainConfig.NODE_I_NAME]);
}
//...
```

5.3.4 Merge dat

Jak bylo popsáno v sekci 4.3.2, při vkládání dat do neprázdné databáze je nutné kontrolovat, zda nedochází k duplikaci dat v grafu. K této kontrole dochází před samotným vložením entity do databáze. Jelikož samotný algoritmus této kontroly byl popsán ve výše zmíněné sekci, demonstruje tato sekce obecný způsob kontroly duplicity pro uzly typu Resource a hrany. Kontrola duplicity pro ostatní typy uzlů je až na názvy a hodnoty vlastností uzlů obdobná.

Jelikož dané kontroly provádíme pomocí dotazů nad databází, je zde rozdíl mezi implementací v jednotlivých verzích frameworku TinkerPop. Jako první uvádím kód v TinkerPop2:

```
//... kontrola duplicity uzlu typu Resource
Iterator<Vertex> vertexIterator = parentNode.query()
        .has(MainConfig.EDGE_CHILD_NAME,
            parts[MainConfig.RESOURCE_I_NAME])
    .direction(Direction.IN).vertices().iterator();
Vertex existingResourceVertex = null;
while (vertexIterator.hasNext()) {
        Vertex checkedVertex = vertexIterator.next();
    if (checkedVertex.getProperty(MainConfig.NODE_NAME)
        .equals(parts[MainConfig.RESOURCE_I_NAME])
        && checkedVertex.getProperty(MainConfig.NODE_TYPE)
        .equals(MainConfig.VERTEX_RESOURCE_TYPE)) {
        existingResourceVertex = checkedVertex;
        //...
        }
}
if (existingResourceVertex == null) {
        addResourceNode(parts, parentNode);
}
//... kontrola duplicity hrany
Iterator<Edge> existingEdges =
    node1.query().has(MainConfig.EDGE_TARGE_ID, node2.getId())
                    .direction(Direction.OUT)
                    .labels(type).edges().iterator();
```

```
Edge createdEdge = null;
while (existingEdges.hasNext()) {
        Edge existingEdge = existingEdges.next();
        if (existingEdge.getVertex(Direction.IN).getId()
                .equals(node2.getId())) {
                //...
        }
}
if (createdEdge == null) {
//...
}
```

V syntaxi TinkerPop3 hlavní dotazy těchto kontrol vypadají následovně:

```
//... kontrola duplicity uzlu typu Resource
Iterator<Vertex> vertexIterator =
    db.getTraversal().V(parentNode.id())
        .inE(MainConfig.EDGE_PARENT_LABEL)
        .has(MainConfig.EDGE_CHILD_NAME,
parts[MainConfig.RESOURCE_I_NAME]).outV();
//... kontrola duplicity hrany
Iterator<Edge> existingEdges =
    db.getTraversal().V(node1.id()).outE(type)
        .has(MainConfig.EDGE_TARGE_ID,node2.id());
```

5.3.5 Chunk import

Pro účely tohoto testu byla testovací data rozdělena na části pomocí vlastního programu „Slicer“, který je možné najít v elektronické příloze. CSV soubory, používané v předchozích testech byli rozděleny do čtveřic, kdy každá čtveřice obsahuje všechny entity vázané na přibližně 5000 uzlů typu Node. Každá čtveřice v kombinaci se všemi předcházejícími tvoří validní graf.

Samotné vkládání dat je totožné s předcházejícími implementacemi. Jediným rozdílem je zdroj dat.

5.3.6 Get Parent

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.4, tato metoda má za úkol získat přímého předka daného uzlu. Probíhá zde i kontrola na počet vrácených uzlů, jelikož přímým předkem může být pouze jeden uzel.

Jako první je uveden kód v TinkerPop2:

```
public static Vertex getParent(Vertex node) {
        Iterable<Vertex> parentIt = node.getVertices(Direction.OUT,
            MainConfig.EDGE_PARENT_LABEL);
    Vertex parent = parentIt.iterator().next();
    //...
```

```
    return parent;
}
```

V TinkerPop3 je kód obdobný, pouze se pro průchod grafem použije objekt třídy `GraphTraversalSource`. Jedná se o objekt, který nad databází umožňuje spouštět dotazy v podobném formátu, jaký se používá přímo pro jazyk Gremlin.

V syntaxi TinkerPop3 tedy hlavní dotaz této metody vypadá následovně:

```
Iterator<Vertex> parentIt =
    traversal.V(node.id()).out(MainConfig.EDGE_PARENT_LABEL);
```

5.3.7 Get Attributes

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.5, tato metoda má za úkol získat kolekci všech uzlů typu Attribute, které jsou vázány na daný uzel.

Jako první je uveden kód v TinkerPop2:

```
public static List<Vertex> getAllNodeAttributes(Vertex node) {
        List<Vertex> attributes = new ArrayList<>();
        Iterable<Vertex> it = node.query()
                .labels(MainConfig.EDGE_ATTRIBUTE_LABEL)
                .direction(Direction.IN).vertices();
       //...
        return attributes;
}
```

Stejně jako v minulém testu se i zde pro průchod grafem používá objekt třídy `GraphTraversalSource`. Tento způsob bude použit i u většiny následujících testů a proto tento fakt už nebude dále zmiňován.

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Iterator<Vertex> it =
    traversal.V(node.id()).in(MainConfig.EDGE_ATTRIBUTE_LABEL);
```

5.3.8 Get Children

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.6, tato metoda má za úkol získat kolekci všech uzlů typu Node, které jsou potomky daného uzlu.

Jako první je uveden kód v TinkerPop2:

```
    public static  List<Vertex> getChildren(Vertex node) {
        List<Vertex> childList = new ArrayList<>();
        Iterable<Vertex> children = node.query()
                .labels(MainConfig.EDGE_PARENT_LABEL)
                .direction(Direction.IN).vertices();
        //...
        return childList;
    }
```

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Iterator<Vertex> children =
    traversal.V(node.id()).in(MainConfig.EDGE_PARENT_LABEL);
```

5.3.9 Get Attribute by Name

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.7, tato metoda má za úkol získat kolekci všech hodnot atributů určitého názvu asociovaných s daným uzlem typu Node.

Jako první je uveden kód v TinkerPop2:

```
public static List<String> getNodeAttribute(Vertex node, String
    key) {
        List<String> results = new ArrayList<String>();
    Iterable<Vertex> it = node.query()
        .labels(MainConfig.EDGE_ATTRIBUTE_LABEL)
        .direction(Direction.IN).vertices();
    //...
        if(key.equals(attribute
        .getProperty(MainConfig.ATTRIBUTE_NAME))) {
                results.add(attribute
                .getProperty(MainConfig.ATTRIBUTE_VALUE));
        }
    //...
    return results;
}
```

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Iterator<Vertex> it =
    traversal.V(node.id()).in(MainConfig.EDGE_ATTRIBUTE_LABEL);
```

5.3.10 Get Children by Name

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.8, tato metoda má za úkol získat kolekci všech potomků daného uzlu typu Node s určitým názvem.

Jako první je uveden kód v TinkerPop2:

```
public static List<Vertex> getChildrenByName(Vertex node, String
    name) {
        List<Vertex> childList = new ArrayList<>();
        Iterable<Vertex> children = node.query()
                .has(MainConfig.EDGE_CHILD_NAME, name)
                .labels(MainConfig.EDGE_PARENT_LABEL)
                .direction(Direction.IN)
                .vertices();
        //...
        return childList;
}
```

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Iterator<Vertex> children = traversal.V(node.id())
                .inE(MainConfig.EDGE_PARENT_LABEL)
                .has(MainConfig.EDGE_CHILD_NAME, name)
                .outV();
```

5.3.11 Get All Parents

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.9, tato metoda má za úkol získat kolekci všech předků daného uzlu typu Node.

Jako první je uveden kód v TinkerPop2:

```
public static List<Vertex> getAllParents(Vertex node) {
    //...
    List<Vertex> parentsList = new ArrayList<>();
    Iterator<Vertex> it = node.getVertices(Direction.OUT,
        MainConfig.EDGE_PARENT_LABEL).iterator();
    while(it.hasNext()) {
        actualNode = it.next();
        parentsList.add(actualNode);
        it = actualNode.getVertices(Direction.OUT,
            MainConfig.EDGE_PARENT_LABEL).iterator();
    }
    //...
    return parentsList;
}
```

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Iterator<Vertex> it =
    traversal.V(node.id()).out(MainConfig.EDGE_PARENT_LABEL);
```

5.3.12 Get Resource

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.10, tato metoda má za úkol získat uzel typu Resource asociovaný s uzlem na vstupu.

Tento test funguje skoro totožně s testem 5.3.11, ale na závěr přidává ještě získání uzlu typu Resource z kořene podstromu všech předků.

Jako první je uveden kód v TinkerPop2:

```
Vertex resource = lastNode.getVertices(Direction.OUT,
    MainConfig.EDGE_RESOURCE_LABEL).iterator().next();
```

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Vertex resource = traversal.V(actualNode.id())
        .out(MainConfig.EDGE_RESOURCE_LABEL).next();
```

5.3.13 Get Vertices by Edge Type

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.11, tato metoda má za úkol získat kolekci uzlů asociovaných s daným uzlem pomocí daného typu a směru hrany.

Jako první je uveden kód v TinkerPop2:

```
    public static List<Vertex> getVerticesByEdgeType(Vertex node,
        String edgeType, Direction dir) {
        List<Vertex> vertices = new ArrayList<>();
        Iterable<Vertex> it = node.getVertices(dir, edgeType);
//...
        return vertices;
    }
```

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Iterator<Vertex> it = traversal.V(node.id()).toE(dir,
    edgeType).toV(dir);
```

5.3.14 Simple Graph Flow

Metoda pro měření výkonnosti testu se nachází ve třídě `GraphOperations`. Jak bylo popsáno v sekci 4.3.12, tato metoda má za úkol získat všechny uzly typu Node, do kterých je možné se dostat opakovaným průchodem po grafu pomocí hran daného typu a daného směru.

Jako první je uveden kód v TinkerPop2:

```
public static List<Vertex> simpleFlow(Vertex node, String edgeType,
    Direction dir) {
        List<Vertex> reachable = new ArrayList<>();
        Deque<Vertex> stack = new ArrayDeque<>();
        Map<String, Boolean> visited = new HashMap<>();
        stack.add(node);
        while(!stack.isEmpty()) {
            Vertex currNode = stack.pop();
            if(visited.containsKey(currNode.getId())) {
                continue; //this node was already visited
            }
            Iterator<Edge> edges = currNode.query().labels(edgeType)
            .direction(dir).edges().iterator();

            while (edges.hasNext()) {
                Edge outgoingEdge = edges.next();
                Vertex neighbour =
                    outgoingEdge.getVertex(dir.opposite());
                stack.push(neighbour);
            }
            visited.put(currNode.getId().toString(), true);
            reachable.add(currNode);
        }
        return reachable;
    }
```

V syntaxi TinkerPop3 hlavní dotaz této metody vypadá následovně:

```
Iterator<Edge> edges = traversal.V(currNode.id()).toE(dir,
    edgeType);
```

5.3.15 Get Node by Name

Metody implementující tento test se nachází přímo v korespondující třídě dané grafové databáze. Důvodem k tomu je to, že dotaz indexovaný externím nástrojem se spouští přímo na instanci databáze nebo vytvořené transakci.

Pro databázi Titan vypadá implementace tohoto testu takto:

```
public List<Vertex> getVertexByName(String name) {
        Iterable<Result<Vertex>> it =
            internalGraph.indexQuery("search", "v." +
            MainConfig.NODE_NAME + ":(/" + name + "/)").vertices();
```

```
        List<Vertex> result = new ArrayList<Vertex>();
        //...
                return result;
    }
```

Implementace pro JanusGraph je prakticky totožná. Jediným rozdílem je to, že metoda `vertices()` je v poslední verzi již „deprecated“, a proto je nutné jednotlivé získané uzly přečíst ze streamu.

OrientDB díky vlastní implementaci fulltext indexu umožňuje používání tohoto indexu přímo v Gremlin příkazu nebo pomocí spuštěného SQL příkazu, jak bylo demonstrováno při vytváření indexů.

Kapitola 6

Výkonnostní testování

V této kapitole jsou popsány výsledky výkonnostního testování jednotlivých databázových řešení nad sadou testů definovaných v sekci 4.3.

6.1 Testovací prostředí

Všechny testy byly spouštěny na notebooku s parametry uvedenými v tabulce 6.1. Během testování bylo spuštěno pouze IDE a programy nutné pro běh operačního systému.

6.2 Testovací data

Testovací data byla poskytnuta zadavatelem práce. Jedná se o CSV soubory, obsahující 9 uzlů typu Resource, 1617799 uzlů typu Node, 2378760 hran typu Direct nebo Filter a 2703157 uzlů typu Attribute.

6.3 Výsledky testování

Hodnoty uvedené v následujících tabulkách vznikly zprůměrováním naměřených hodnot. V rámci testů pro ukládání dat je průměr vypočítán ze tří naměřených hodnot, kdežto v testech pro dotazy nad databází je průměr vypočí-

Notebook	Dell Latitude E6530
Operační systém	Windows 10 (64-bit)
Procesor	Intel Core i7-3740QM (2.7 GHz)
RAM	8 GB
Disk	HDD

Tabulka 6.1: Konfigurace testovacího prostředí

tán z deseti naměřených hodnot. Všechny naměřené hodnoty je možné nalézt v příloze B. Všechny testy pro dotazy nad databází byly spouštěny sekvenčně na nově nastartované databázi, aby se tak předešlo zkreslení výsledků díky načtení dat do vyrovnávací paměti („cache“).

6.3.1 Import dat

Zprůměrované výsledky vkládání kompletního grafu do nové databáze pro jednotlivá řešení jsou shrnuty v tabulce 6.2. Pro demonstraci výhodnosti embedded řešení oproti řešení s local serverem byly do testování zahrnuty i varianty s local server nasazením.

Grafová databáze	**Čas**
Titan (Persistit)	9 m 34 s
OrientDB	20 m 29 s
OrientDB server	41 m 57 s
JanusGraph (Cassandra)	14 m 57 s
JanusGraph (Cassandra) server	21m 22 s
JanusGraph (BerkeleyDB)	13 m 6 s

Tabulka 6.2: Vkládání nového grafu do databáze.

6.3.2 Merge import

Zprůměrované výsledky vkládání kompletního grafu do zaplněné databáze pro jednotlivá řešení jsou uvedeny v tabulce 6.3.

Grafová databáze	**Čas**
Titan (Persistit)	17 m 14 s
OrientDB	1h 15m 58s
JanusGraph (Cassandra)	10h+
JanusGraph (BerkeleyDB)	2h 2m

Tabulka 6.3: Vkládání grafu do zaplněné databáze databáze.

Oproti předchozímu testu je zde dobře patrné znatelné zpomalení, které je způsobeno prohledáváním existujícího grafu pro možné duplicity.

6.3.3 Chunk import

Zprůměrované výsledky vkládání rozdrobeného grafu do nové databáze pro jednotlivá řešení jsou uvedeny v tabulce 6.4.

Grafová databáze	Čas
Titan (Persistit)	20 m 47 s
OrientDB	33 m 30 s
JanusGraph (Cassandra)	1h 10m 12s
JanusGraph (BerkeleyDB)	27 m 59 s

Tabulka 6.4: Vkládání grafu do databáze po částech.

Oproti předchozím testům je zde patrné zpomalení, které je zapříčiněno opakovaným odpojováním a připojováním k databázi.

6.3.4 Get Parent

Zprůměrované výsledky hledání předka uzlu pro jednotlivá řešení jsou shrnuty v tabulce 6.5. Pro demonstraci vhodnosti embedded řešení oproti local server řešení byly do testování zahrnuty i varianty s local server nasazením.

Grafová databáze	Čas (ms)
Titan (Persistit)	0,4891
OrientDB	1,9753
OrientDB server	8,9333
JanusGraph (Cassandra)	87,2443
JanusGraph (Cassandra) server	91,3885
JanusGraph (BerkeleyDB)	10,8480

Tabulka 6.5: Získání předka uzlu.

6.3.5 Get Attributes

Zprůměrované výsledky hledání všech uzlu typu Attribute asociovaných s daným uzlem pro jednotlivá řešení jsou prezentovány v tabulce 6.6. Pro demonstraci vhodnosti embedded řešení oproti local server řešení byly do testování zahrnuty i varianty s local server nasazením.

6.3.6 Get Children

Zprůměrované výsledky hledání všech potomků daného uzlu pro jednotlivá řešení jsou uvedeny v tabulce 6.7.

Řešení s lokálním serverem se ve všech předchozích testech ukázala jako řešení pomalejší. Proto již nebudou v následujících testech uvažována.

Grafová databáze	Čas (ms)
Titan (Persistit)	6,9096
OrientDB	9,4444
OrientDB server	15,9076
JanusGraph (Cassandra)	158,0481
JanusGraph (Cassandra) server	164,8281
JanusGraph (BerkeleyDB)	43,5917

Tabulka 6.6: Získání všech atributů.

Grafová databáze	Čas (ms)
Titan (Persistit)	0,1306
OrientDB	0,4707
JanusGraph (Cassandra)	11,1907
JanusGraph (BerkeleyDB)	0,7447

Tabulka 6.7: Získání všech potomků daného uzlu.

6.3.7 Get Attributes by Name

Zprůměrované výsledky hledání všech uzlu typu Attribute o dané hodnotě, asociovaných s daným uzlem pro jednotlivá řešení jsou uvedeny v tabulce 6.8.

Grafová databáze	Čas (ms)
Titan (Persistit)	1,5983
OrientDB	1,3334
JanusGraph (Cassandra)	124,2800
JanusGraph (BerkeleyDB)	8,3837

Tabulka 6.8: Získání všech atributů daného jména.

6.3.8 Get Children by Name

Zprůměrované výsledky hledání všech potomků daného uzlu s daným jménem pro jednotlivá řešení jsou uvedeny v tabulce 6.9.

Grafová databáze	Čas (ms)
Titan (Persistit)	1,0487
OrientDB	1,0642
JanusGraph (Cassandra)	3,5161
JanusGraph (BerkeleyDB)	3,7108

Tabulka 6.9: Získání všech potomků daného uzlu s určitým jménem.

6.3.9 Get All Parents

Zprůměrované výsledky hledání všech předků daného uzlu pro jednotlivá řešení jsou shrnuty v tabulce 6.10.

Grafová databáze	Čas (ms)
Titan (Persistit)	0,5492
OrientDB	1,7219
JanusGraph (Cassandra)	90,5023
JanusGraph (BerkeleyDB)	4,5880

Tabulka 6.10: Získání všech předků daného uzlu.

6.3.10 Get Resource

Zprůměrované výsledky hledání uzlu typu Resource, asociovaného s daným uzlem pro jednotlivá řešení jsou uvedeny v tabulce 6.11.

Grafová databáze	Čas (ms)
Titan (Persistit)	0,9702
OrientDB	0,5966
JanusGraph (Cassandra)	109,0458
JanusGraph (BerkeleyDB)	8,7528

Tabulka 6.11: Získání uzlu typu Resource pro daný uzel.

6.3.11 Get Vertices by Edge Type

Zprůměrované výsledky hledání všech uzlů, spojených s daným uzlem pomocí hrany určitého typu pro jednotlivá řešení jsou uvedeny v tabulce 6.12.

Grafová databáze	Čas (ms)
Titan (Persistit)	0,2390
OrientDB	0,9599
JanusGraph (Cassandra)	61,9114
JanusGraph (BerkeleyDB)	3,1846

Tabulka 6.12: Získání uzlů, spojených pomocí určitého typu hrany.

6.3.12 Simple Graph Flow

Zprůměrované výsledky hledání všech uzlů, dostupných z daného uzlu pomocí průchodu grafu přes hrany daného typu pro jednotlivá řešení jsou uvedeny v tabulce 6.13.

Grafová databáze	Čas (ms)
Titan (Persistit)	8,5768
OrientDB	26,9291
JanusGraph (Cassandra)	668,5310
JanusGraph (BerkeleyDB)	57,4749

Tabulka 6.13: Získání uzlů, dostupných pomocí daného typu hrany.

6.3.13 Get Node by Name

Zprůměrované výsledky hledání všech uzlů daného jména pro jednotlivá řešení jsou uvedeny v tabulce 6.14.

Grafová databáze	Čas (ms)
Titan (Persistit)	244,2358
OrientDB	83,9629
JanusGraph (Cassandra)	335,3559
JanusGraph (BerkeleyDB)	112,3308

Tabulka 6.14: Získání všech uzlů daného jména.

6.4 Diskuze nad výsledky

Z výše uvedených výsledků je patrné, že nástroj Titan v kombinaci s úložištěm Persistit podává nejlepší výkony v rámci testů na ukládání dat. Ačkoliv výsledky vkládání dat v kombinaci nástroje JanusGraph s úložištěm BerkeleyDB

byly přiměřené, výsledky testů pro dotazy nad databází a merge vkládání neodpovídaly tomu, že se při rozsahu testovacích dat jedná o optimální případ použití tohoto úložiště[60]. Stejně tak prvotní uložení při použití úložiště Cassandra proběhlo neočekávaně rychle vzhledem k tomu, že je tato databáze primárně používána pro data řádově větší[61].

Nástroj OrientDB se pro ukládání testovacích dat ukázal být znatelně pomalejší, než stávající řešení. V rámci snahy o zrychlení procesu vkládání dat bylo vkládání dat upraveno tak, aby lépe využívalo rozhraní poskytované touto databází [59]. Ze stejného důvodu došlo k úpravě velikosti cache, vypnutí validace, přípravě databáze pro vložení většího množství dat a implementaci dalších tipů, uvedených v dokumentaci jako postupy vhodné pro zlepšení výkonu [62]. Tyto změny sice přinesly jistý dopad do produktivity, ale jednalo se pouze o zrychlení v řádu pár minut. Možným důvodem pro tento výkonnostní propad může být fakt, že (dle dokumentace) OrientDB nedoporučuje vkládání vlastností na hrany, což je funkcionalita, která se v rámci datového modelu nástroje Manta Flow využívá velice často. V rámci doporučených úprav je v této dokumentaci uvedena také rada o znovupoužití Vertex objektů pro vkládání uzlů do databáze. Po studiu Java API [63] se ale ukázalo, že tuto metodu není možné implementovat a nejspíše se jedná o neopravená rezidua z některé ze starších verzí dokumentace, která byla v rámci vývoje opomenuta. Na druhou stranu, testování dotazů nad databází dopadlo pro OrientDB mnohem lépe. Jednalo se o druhé nejrychlejší řešení.

Jak bylo uvedeno výše, kombinace databáze JanusGraph a úložiště Cassandra se pro základní vkládání ukázala jako neočekávaně rychlá. Tohoto výsledku bylo dosaženo zejména povolením nastavení `storage.batch-loading` a `query.fast-property`, díky jimž došlo k řádovému urychlení. První nastavení vypíná vnitřní kontrolu konzistence databáze při vkládání, zatímco druhé nastavení upraví čtení z databáze tak, že při čtení objektu jsou získány i všechny vlastnosti daného objektu. Problém zde ale nastává při vkládání dat pomocí metody merge. V tomto případě dochází k takovému propadu rychlosti vkládání dat, že celý proces je při použití testovacích dat neúnosně pomalý. K tomuto zpomalení dochází během kontroly případné duplicity objektů, kdy každé čtení z databáze trvá v průměru 15 ms. Ve snaze o urychlení byla konfigurace databáze JanusGraph[58] a konfigurace samotné Cassandry[64] upravena tak, aby co nejlépe odpovídala konfiguraci testovacího prostředí. Došlo k úpravě konzistence čtení a zapisování pro transakce, úpravám komprese dat, replikačního faktoru, nastavení cache a vertex cache. Tyto změny ale přinesly pouze minimální dopad. Možným důvodem tohoto výkonnostního propadu může být nedostatečný výkon testovacího prostředí. V dokumentaci Cassandry[65] je uvedeno jako doporučené množství paměti pro heap 8 GB. Dalším možným důvodem ke zpomalení je časté spouštění proprietárního garbace collectoru, jehož nastavení je verzi 2.1.x oproti novějším verzím značně omezené[66]. Cassandra od verze 3.0 nabízí mimo jiné nový proprietární garbage collector, který snižuje zpozdění oproti standardnímu garbage collectoru,

který je používán v předchozích verzích. Jelikož je JanusGraph ve své poslední verzi v embedded módu kompatibilní pouze s Cassandrou verze 2.1.x, nemohlo dojít k porovnání případného zlepšení výkonu v nových verzích. Vzhledem k značnému propadu výkonu při přidání kontroly duplicity pomocí dotazů do databáze je tedy možné, že testovací prostředí není dostatečně výkonné pro to, aby úložišti Cassandra umožňovalo bezproblémově zapisovat a zároveň nad databází provádět dotazy.

Kombinace databáze JanusGraph a úložiště BerkeleyDB v testech pro základní vkládání dat poskytuje výsledky jen o málo pomalejší, než stávající řešení. V základním nastavení docházelo k častým pauzám při vkládání entit do databáze, ale těmto pauzám se podařilo prakticky zabránit pomocí snížení hodnoty konfigurace `storage.berkeleyje.cache-percentage` na 30 procent z důvodu rozsahu vkládáných dat. Dále mírně pomohlo zvýšení nastavení `cache.tx-cache-size` na pětinásobek základní hodnoty[58]. Výsledky ostatních testů už tolik příznivé nebyly. Možné konfigurace úložiště BerkeleyDB jsou bohužel v porovnání s možnými nastaveními úložiště Cassandra značně omezené[58] a tak je nepravděpodobné, že by se na testovací konfiguraci podařilo dosáhnout lepších výsledků.

Řešení, založené na nasazení databází OrientDB a JanusGraph v kombinaci s úložištěm Cassandra na lokálně spuštěném serveru se v obou případech ukázalo jako značně pomalejší oproti embedded řešení napříč všemi případnými testy. Tento fakt není nijak překvapivý, jelikož v případě takového nasazení je nutná dodatečná komunikace mezi spuštěnými procesy. V embedded nasazení tato komunikace není nutná, jelikož všechny části aplikace běží pod jedním JVM.

6.5 Doporučení

Vzhledem k naměřeným výsledkům se jako nejlepší řešení z hlediska výkonnosti stále jeví momentálně užívaná kombinace databáze Titan s úložištěm Persistit. Toto řešení ale rozhodně není optimální. Vinou ukončení vývoje obou nástrojů zde již neexistuje očekávání možného přidání nových funkcionalit či oprav případných chyb. Jedním z východisek ze vzniklé situace by v tomto případě bylo dedikování interních lidských zdrojů pro vytvoření a rozvoj vlastních nástrojů, vycházejících z publikovaných zdrojových kódů, což je ale jak lidsky, tak finančně velmi náročné.

JanusGraph je ještě relativně mladá technologie s nemalým potenciálem pro budoucí vývoj. Vzhledem k licenční politice datového úložiště BerkeleyDB a jeho naměřeným výsledkům v testech pro dotazy nad databází ale tuto technologii nelze doporučit k používání. Naproti tomu úložiště Cassandra má i přes naměřené výsledky možný potenciál, který by zcela jistě stálo za to prozkoumat na výkonnějších konfiguracích.

OrientDB se svými výsledky v dotazovacích testech nejvíce blíží výkonu

databáze Titan. Jeho užitnou hodnotu ale degraduje produktivita v testech pro vkládání dat. Je možné, že tento nedostatek může být vyřešen na výkonnějších konfiguracích. Velkou výhodou OrientDB je velmi dobře zpracovaná dokumentace a kompatibilita se stávající implementací funkcionalit nástroje Manta Flow díky frameworku TinkerPop2, nad kterým dodává vlastní vylepšení a přídavky. Toto řešení by bylo vhodné v případě, kdy operace vkládání dat neprobíhá příliš často a prioritou je rychlost čtení z databáze.

Z výkonnostního testování oproti očekávání nevzešel žádný jasný kandidát pro náhradu stávajícího řešení. Existuje však několik možných postupů do budoucna, kde každý z postupů má svá pro a proti.

Závěr

Cílem této práce bylo zanalyzovat momentální situaci na trhu grafových databází. Tato analýza proběhla v kapitole 3.2. Z této analýzy možných řešení bylo následně nutné vybrat podmnožinu případných alternativ pro datové úložiště nástroje Manta Flow.

Dalším cílem této práce bylo seznámení se s datovou strukturu nástroje Manta Flow, jeho potřebami a požadavky. Díky hlubšímu poznání tohoto nástroje bylo možné navrhnout množinu testů, které budou mít jednoznačnou vypovídající hodnotu ohledně vhodnosti použití konkrétního řešení jako datového úložiště pro nástroj Manta Flow. Tato analýza proběhla v kapitole 4. Následně došlo k implementaci testů a k jejich výkonnostnímu otestování v kapitole 5, respektive v kapitole 6. Na základě naměřených hodnot bylo doporučeno několik případných řešení a možností pro budoucí rozvoj.

Výsledkem implementační části je kód, který je s drobnými úpravami znovupoužitelný i pro práci s dalšími případnými grafovými databázemi. Úplné kompatibilitě zabraňují rozdíly v přístupových vrstvách a API.

Možnosti dalšího rozvoje

V rámci analýzy byly definovány také testy 4.3.14 a 4.3.15, které nebyly v rámci této práce implementovány a otestovány. Mohou tak posloužit jako základ další případné práce, která by mohla pokračovat v následné analýze rozebraných řešení a jejich dalšímu testování na prostředích se silnější konfigurací zdrojů. Ta by měla lépe splňovat výkonnostní nároky prostředí a více se blížit reálné produkční konfiguraci.

Vzhledem k faktu, že obor grafových databází je mladý a dynamicky se rozvíjející, je možné, že se v dohledné době objeví nová databázová řešení, která mohou být perspektivní pro možné použití v nástroji Manta Flow.

Literatura

[1] Peroutka, M.: *Optimální struktura a indexy modelu metadatového úložiště v grafové databázi.* Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

[2] Rogers, F. B.: The Development of Punch Card Tabulation in the Bureau of the Census, 1890-1940; with Outlines of Actual Tabulation Programs. *Bulletin of the Medical Library Association*, April 1966. Dostupné z: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC198417/pdf/mlab00175-0113b.pdf

[3] Takizawa, M.; Itoh, H.; Moriya, K.: Logic interface system on navigational database systems. In *Logic Programming*, 1986. Dostupné z: https://link.springer.com/chapter/10.1007/3-540-18024-9_23

[4] Foote, K. D.: A Brief History of Database Management. 2017. Dostupné z: http://www.dataversity.net/brief-history-database-management/

[5] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, June 1970. Dostupné z: http://www.morganslibrary.net/files/codd-1970.pdf

[6] Chamberlin, D. D.; Astrahan, M. M.; Blasgen, M. W.; aj.: A History and Evaluation of System R. *Communications of the ACM*, October 1981. Dostupné z: https://people.eecs.berkeley.edu/~brewer/cs262/SystemR.pdf

[7] Codd, E. F.: Further Normalization of the Data Base Relational Model. *IBM Research Report RJ909*, August 1971.

[8] DB-Engines Ranking. Cit: 2018-01-03. Dostupné z: https://db-engines.com/en/ranking

[9] Leavitt, N.: Will NoSQL Databases Live Up to Their Promise? *Computer*, ročník 43, č. 2, February 2010: s. 12–14. Dostupné z: http://www.leavcom.com/pdf/NoSQL.pdf

[10] No-SQL Databases. Cit: 2018-01-03. Dostupné z: http://nosql-database.org/

[11] Fowler, M.: Nosql Definition. 2017. Dostupné z: https://martinfowler.com/bliki/NosqlDefinition.html

[12] Gilbert, S.; Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, ročník 33, č. 2, Červen 2002: s. 51–59, ISSN 0163-5700, doi:10.1145/564585.564601. Dostupné z: http://doi.acm.org/10.1145/564585.564601

[13] Pritchett, D.: BASE: An Acid Alternative. *Queue*, ročník 6, č. 3, Květen 2008: s. 48–55, ISSN 1542-7730, doi:10.1145/1394127.1394128. Dostupné z: http://doi.acm.org/10.1145/1394127.1394128

[14] Rusher, J.: Triple Store. 2002-2004. Dostupné z: https://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html

[15] OrientDB Ltd: Optimising Graph Data with a Multi-Model Engine. Cit: 2018-01-03. Dostupné z: http://orientdb.com/graph-database/

[16] Neo4j, I.: From Relational to Neo4j. Dostupné z: https://neo4j.com/developer/graph-db-vs-rdbms/

[17] Neo4j, I.: From Relational to Neo4j. Dostupné z: https://neo4j.com/blog/open-cypher-sql-for-graphs/

[18] Herman, I.: SPARQL is a Recommendation. Dostupné z: https://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation/

[19] Haupt, M.: *Data lineage*. Diplomová práce, Masarykova universita, Fakulta informatiky, 2009.

[20] King, T.: What is Data Lineage? Dostupné z: https://solutionsreview.com/data-integration/what-is-data-lineage/

[21] CzechCrunch: Akademik a manager Tomáš Krátký: Čeští studenti přemýšlí v souvislostech, firmy jim ale moc nepomáhají. Dostupné z: http://www.czechcrunch.cz/2017/06/akademik-a-manager-tomas-kratky-cesti-studenti-premysli-v-souvislostech-firmy-jim-ale-moc-nepomahaji/

[22] Tyinternety.cz: Rozhodnuto. S Czech ICT Alliance jedou do Silicon Valley dva startupy – Manta a realPad. Dostupné z: http://tyinternety.cz/startupy/rozhodnuto-s-ict-alliance-jedou-silicon-valley-dva-startupy-manta-realpad/

[23] Manta Tools, s.: Manta Flow. Dostupné z: https://getmanta.com/

[24] Lupa.cz: Čeští Manta Tools získali zákazníky PayPal a Comcast, zajímá je Wall Street. Dostupné z: https://www.lupa.cz/clanky/cesti-manta-tools-ziskali-zakazniky-paypal-a-comcast-zajima-je-wall-street/

[25] The Apache Software Foundation: TinkerPop3 Documentation. Cit: 2018-01-03. Dostupné z: http://tinkerpop.apache.org/docs/3.3.1/reference/

[26] tinkerpop/blueprints Wiki. Cit: 2018-01-03. Dostupné z: https://github.com/tinkerpop/blueprints/wiki

[27] tinkerpop/rexter Wiki. Cit: 2018-01-03. Dostupné z: https://github.com/tinkerpop/rexter/wiki

[28] tinkerpop/furnace Wiki. Cit: 2018-01-03. Dostupné z: https://github.com/tinkerpop/furnace/wiki

[29] tinkerpop/frames Wiki. Cit: 2018-01-03. Dostupné z: https://github.com/tinkerpop/frames/wiki

[30] tinkerpop/gremlin Wiki. Cit: 2018-01-03. Dostupné z: https://github.com/tinkerpop/gremlin/wiki

[31] tinkerpop/pipes Wiki. Cit: 2018-01-03. Dostupné z: https://github.com/tinkerpop/pipes/wiki

[32] The Apache Software Foundation: The Gremlin Graph Traversal Machine and Language. Cit: 2018-01-03. Dostupné z: http://tinkerpop.apache.org/gremlin.html

[33] Rodriguez, M. A.: The Gremlin Graph Traversal Machine and Language. *ACM Database Programming Languages Conference*, May 2015.

[34] Titan homepage. Cit: 2018-01-03. Dostupné z: http://titan.thinkaurelius.com/

[35] Titan Storage Backends. Cit: 2018-01-03. Dostupné z: http://s3.thinkaurelius.com/docs/titan/0.5.4/storage-backends.html

[36] Titan Storage Backends Cassandra. Cit: 2018-01-03. Dostupné z: http://s3.thinkaurelius.com/docs/titan/0.5.4/storage-backends.html

[37] Titan Storage Backends HBase. Cit: 2018-01-03. Dostupné z: http://s3.thinkaurelius.com/docs/titan/0.5.4/storage-backends.html

[38] Titan Storage Backends BerkeleyDB. Cit: 2018-01-03. Dostupné z: http://s3.thinkaurelius.com/docs/titan/0.5.4/storage-backends.html

[39] Enterprise, D.: DataStax Acquires Aurelius, The Experts Behind TitanDB. Cit: 2018-01-03. Dostupné z: https://www.datastax.com/2015/02/datastax-acquires-aurelius-the-experts-behind-titandb

[40] JanusGraph homepage. Cit: 2018-01-03. Dostupné z: http://janusgraph.org/

[41] JanusGraph Storage Backends Bigtable. Cit: 2018-01-03. Dostupné z: http://docs.janusgraph.org/latest/bigtable.html

[42] JanusGraph Storage Backends. Cit: 2018-01-03. Dostupné z: http://docs.janusgraph.org/latest/storage-backends.html

[43] The Apache Software Foundation: Apache TinkerPop Homepage. Cit: 2018-01-03. Dostupné z: http://tinkerpop.apache.org/

[44] OrientDB Ltd: OrientDB Storages. Cit: 2018-01-03. Dostupné z: http://orientdb.com/docs/last/Storages.html

[45] OrientDB Ltd: OrientDB Enterprise - Multi-Model NoSQL Enterprise Database. Cit: 2018-01-03. Dostupné z: http://orientdb.com/orientdb-enterprise/

[46] ArangoDB GmbH: ArangoDB - a native multi-model database. Cit: 2018-01-03. Dostupné z: https://www.arangodb.com/why-arangodb/multi-model/

[47] ArangoDB GmbH: Arango Storage Engines. Cit: 2018-01-03. Dostupné z: https://docs.arangodb.com/3.3/Manual/Architecture/StorageEngines.html

[48] ArangoDB GmbH: ArangoDB query language. Cit: 2018-01-03. Dostupné z: https://docs.arangodb.com/3.3/AQL/index.html

[49] Neo4j, Inc.: Neo4j products. Cit: 2018-01-03. Dostupné z: https://neo4j.com/product/

[50] Neo4j, Inc.: About Neo4j Licenses. Cit: 2018-01-03. Dostupné z: https://neo4j.com/licensing/

[51] The Apache Software Foundation: Apache Lucene Core. Cit: 2018-01-03. Dostupné z: https://lucene.apache.org/core/

[52] The Apache Software Foundation: Apache Solr. Cit: 2018-01-03. Dostupné z: http://lucene.apache.org/solr/

[53] Hortonworks Inc.: Bringing Enterprise Search to Enterprise Hadoop. Cit: 2018-01-03. Dostupné z: https://hortonworks.com/blog/bringing-enterprise-search-enterprise-hadoop/

[54] DataStax: DataStax Enterprise: Cassandra with Solr Integration Details. Cit: 2018-01-03. Dostupné z: https://www.datastax.com/dev/blog/datastax-enterprise-cassandra-with-solr-integration-details

[55] OrientDB Ltd: Orient Configuration. Cit: 2018-01-03. Dostupné z: http://orientdb.com/docs/last/Configuration.html

[56] Titan Configuration. Cit: 2018-01-03. Dostupné z: http://s3.thinkaurelius.com/docs/titan/0.5.4/configuration.html

[57] JanusGraph Configuration. Cit: 2018-01-03. Dostupné z: http://docs.janusgraph.org/latest/configuration.html

[58] JanusGraph Configuration Reference. Cit: 2018-01-03. Dostupné z: http://docs.janusgraph.org/latest/config-ref.html

[59] OrientDB Ltd: Orient Vertices and Edges. Cit: 2018-01-03. Dostupné z: http://orientdb.com/docs/last/Graph-VE.html

[60] JanusGraph Oracle Berkeley DB Java Edition. Cit: 2018-01-03. Dostupné z: http://docs.janusgraph.org/latest/bdb.html

[61] JanusGraph Apache Cassandra. Cit: 2018-01-03. Dostupné z: http://docs.janusgraph.org/latest/cassandra.html

[62] OrientDB Ltd: Orient Performance Tuning. Cit: 2018-01-03. Dostupné z: http://orientdb.com/docs/last/Performance-Tuning-Graph.html

[63] OrientDB Ltd: Orient Javadoc. Cit: 2018-01-03. Dostupné z: http://orientdb.com/javadoc/2.2.x/

[64] Foundation, T. A. S.: Cassandra Configuration File. Cit: 2018-01-03. Dostupné z: http://cassandra.apache.org/doc/latest/configuration/cassandra_config_file.html

[65] Enterprise, D.: Cassandra 2.1 Tuning Java resources. Cit: 2018-01-03. Dostupné z: https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_tune_jvm_c.html

[66] Enterprise, D.: Cassandra 3.0 Tuning Java resources. Cit: 2018-01-03. Dostupné z: https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsTuneJVM.html

Příloha **A**

Seznam použitých zkratek

XML Extensible markup language

CSV Comma-separated values

JVM Java Virtual Machine

CAP Consistency, Availability, Partition tolerance

ACID Atomicity, Consistency, Isolation, Durability

BASE Basically Available Soft-state services with Eventual-consistency

SQL Structured Query Language

AQL ArangoDB query language

CODASYL Conference/Committee on Data Systems Languages

JSON JavaScript Object Notation

BLOB Binary large object

ETL Extract, Transform, Load

API Application programming interface

HTTP Hypertext Transfer Protocol

HDFS Hadoop distributed file system

IDE Integrated development environment

Příloha B

Naměřené hodnoty

V této příloze jsou popsány naměřené hodnoty všech provedených testů. Naměřené časy pro dotazovací testy jsou uvedeny v nanosekundách pro co největší přesnost.

B.1 Import dat

B.1.1 Titan (Persistit)

Měření	**Čas**
č. 1	9m 34s
č. 2	9m 41s
č. 3	9m 27s

Tabulka B.1: Výsledky měření pro Titan(Persistit), test Import dat.

B.1.2 OrientDB

Měření	**Čas**
č. 1	20m 14s
č. 2	20m 40s
č. 3	20m 34s

Tabulka B.2: Výsledky měření pro OrientDB, test Import dat.

B.1.3 OrientDB Server

Měření	**Čas**
č. 1	41m 57s
č. 2	42m 13s
č. 3	41m 40s

Tabulka B.3: Výsledky měření pro OrientDB server, test Import dat.

B.1.4 JanusGraph (Cassandra)

Měření	**Čas**
č. 1	15m 5s
č. 2	14m 48s
č. 3	14m 51s

Tabulka B.4: Výsledky měření pro JanusGraph (Cassandra), test Import dat.

B.1.5 JanusGraph (Cassandra) Server

Měření	**Čas**
č. 1	21m 15s
č. 2	21m 32s
č. 3	21m 19s

Tabulka B.5: Výsledky měření pro JanusGraph (Cassandra) server, test Import dat.

B.1.6 JanusGraph (BerkeleyDB)

Měření	Čas
č. 1	13m 5s
č. 2	13m 2s
č. 3	13m 10s

Tabulka B.6: Výsledky měření pro JanusGraph (BerkeleyDB), test Import dat.

B.2 Merge import

B.2.1 Titan (Persistit)

Měření	Čas
č. 1	16m 59s
č. 2	17m 24s
č. 3	17m 20s

Tabulka B.7: Výsledky měření pro Titan(Persistit), test Merge import dat.

B.2.2 OrientDB

Měření	Čas
č. 1	1h 16m 5s
č. 2	1h 15m 47s
č. 3	1h 16m 2s

Tabulka B.8: Výsledky měření pro OrientDB, test Merge import dat.

B.2.3 JanusGraph (Cassandra)

Měření	Čas
č. 1	10h+
č. 2	10h+
č. 3	10h+

Tabulka B.9: Výsledky měření pro JanusGraph (Cassandra), test Merge import dat.

B.2.4 JanusGraph (BerkeleyDB)

Měření	Čas
č. 1	2h 2m 5s
č. 2	2h 2m 1s
č. 3	2h 1m 55s

Tabulka B.10: Výsledky měření pro JanusGraph (BerkeleyDB), test Merge import dat.

B.3 Chunk import

B.3.1 Titan (Persistit)

Měření	Čas
č. 1	20m 41s
č. 2	20m 57s
č. 3	20m 42s

Tabulka B.11: Výsledky měření pro Titan(Persistit), test Chunk import dat.

B.3.2 OrientDB

Měření	Čas
č. 1	33m 24s
č. 2	33m 29s
č. 3	33m 37s

Tabulka B.12: Výsledky měření pro OrientDB, test Chunk import dat.

B.3.3 JanusGraph (Cassandra)

Měření	Čas
č. 1	1h 9m 57s
č. 2	1h 10m 24s
č. 3	1h 10m 13s

Tabulka B.13: Výsledky měření pro JanusGraph (Cassandra), test Chunk import dat.

B.3.4 JanusGraph (BerkeleyDB)

Měření	Čas
č. 1	28m 2s
č. 2	27m 55s
č. 3	27m 59s

Tabulka B.14: Výsledky měření pro JanusGraph (BerkeleyDB), test Chunk import dat.

B.4 Get Parent

B.4.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	439846
č. 2	598891
č. 3	503388
č. 4	520129
č. 5	458109
č. 6	420061
č. 7	433378
č. 8	439466
č. 9	654443
č. 10	424246

Tabulka B.15: Výsledky měření pro Titan(Persistit), test Get Parent.

B.4.2 OrientDB

Měření	Čas (ns)
č. 1	1983492
č. 2	1964087
č. 3	1933406
č. 4	1944683
č. 5	1991482
č. 6	1943230
č. 7	1937903
č. 8	2045512
č. 9	1997191
č. 10	2012029

Tabulka B.16: Výsledky měření pro OrientDB, test Get Parent.

B.4.3 OrientDB Server

Měření	Čas (ns)
č. 1	6292916
č. 2	6092018
č. 3	5940584
č. 4	13040138
č. 5	6306234
č. 6	13114334
č. 7	14209764
č. 8	5702397
č. 9	5993852
č. 10	12640243

Tabulka B.17: Výsledky měření pro OrientDB server, test Get Parent.

B.4.4 JanusGraph (Cassandra)

Měření	Čas (ns)
č. 1	87964913
č. 2	91283160
č. 3	82191362
č. 4	83629172
č. 5	87540595
č. 6	90647453
č. 7	90701863
č. 8	76455953
č. 9	91262323
č. 10	90766545

Tabulka B.18: Výsledky měření pro JanusGraph (Cassandra), test Get Parent.

B.4.5 JanusGraph (Cassandra) Server

Měření	Čas (ns)
č. 1	91089581
č. 2	91293524
č. 3	91672871
č. 4	91497846
č. 5	91032508
č. 6	91405007
č. 7	91527905
č. 8	91337280
č. 9	91039737
č. 10	91446861

Tabulka B.19: Výsledky měření pro JanusGraph (Cassandra) server, test Get Parent.

B.4.6 JanusGraph (BerkeleyDB)

Měření	Čas (ns)
č. 1	11003420
č. 2	10544445
č. 3	11892582
č. 4	9943995
č. 5	9885399
č. 6	9853819
č. 7	10914317
č. 8	10992697
č. 9	13348716
č. 10	10100857

Tabulka B.20: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Parent.

B.5 Get Attributes

B.5.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	7113256
č. 2	6469847
č. 3	6517027
č. 4	6941274
č. 5	7015850
č. 6	7108308
č. 7	7551580
č. 8	7096513
č. 9	6579428
č. 10	6702706

Tabulka B.21: Výsledky měření pro Titan(Persistit), test Get Attributes.

B.5.2 OrientDB

Měření	Čas (ns)
č. 1	9104735
č. 2	9076579
č. 3	9265302
č. 4	9554096
č. 5	9463540
č. 6	9583010
č. 7	9879415
č. 8	9448700
č. 9	9417120
č. 10	9651883

Tabulka B.22: Výsledky měření pro OrientDB, test Get Attributes.

B.5.3 OrientDB Server

Měření	Čas (ns)
č. 1	18624583
č. 2	18621920
č. 3	18122337
č. 4	11672278
č. 5	19251250
č. 6	12380369
č. 7	12190505
č. 8	17813760
č. 9	18953706
č. 10	11444746

Tabulka B.23: Výsledky měření pro OrientDB server, test Get Attributes.

B.5.4 JanusGraph (Cassandra)

Měření	Čas (ns)
č. 1	154329547
č. 2	162614177
č. 3	154533717
č. 4	157970681
č. 5	159908082
č. 6	157286094
č. 7	158482680
č. 8	159354736
č. 9	160616439
č. 10	155384354

Tabulka B.24: Výsledky měření pro JanusGraph (Cassandra), test Get Attributes.

B.5.5 JanusGraph (Cassandra) Server

Měření	**Čas (ns)**
č. 1	164413724
č. 2	162317201
č. 3	166548652
č. 4	166032708
č. 5	169346011
č. 6	162733602
č. 7	174690747
č. 8	161010219
č. 9	175715406
č. 10	163686989

Tabulka B.25: Výsledky měření pro JanusGraph (Cassandra) server, test Get Attributes.

B.5.6 JanusGraph (BerkeleyDB)

Měření	**Čas (ns)**
č. 1	43440951
č. 2	45161084
č. 3	47934093
č. 4	42749540
č. 5	43007893
č. 6	42216855
č. 7	42147590
č. 8	43003311
č. 9	42132581
č. 10	44123245

Tabulka B.26: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Attributes.

B.6 Get Children

B.6.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	122898
č. 2	141923
č. 3	129747
č. 4	131269
č. 5	122898
č. 6	126704
č. 7	122518
č. 8	129366
č. 9	138118
č. 10	140401

Tabulka B.27: Výsledky měření pro Titan(Persistit), test Get Children.

B.6.2 OrientDB

Měření	Čas (ns)
č. 1	440987
č. 2	446315
č. 3	484744
č. 4	464197
č. 5	486646
č. 6	496159
č. 7	480559
č. 8	454685
č. 9	476754
č. 10	476374

Tabulka B.28: Výsledky měření pro OrientDB, test Get Children.

B.6.3 JanusGraph (Cassandra)

Měření	Čas (ns)
č. 1	10273664
č. 2	13179783
č. 3	13312574
č. 4	9995845
č. 5	13638648
č. 6	3247098
č. 7	15130548
č. 8	15500383
č. 9	15147670
č. 10	2480412

Tabulka B.29: Výsledky měření pro JanusGraph (Cassandra), test Get Children.

B.6.4 JanusGraph (BerkeleyDB)

Měření	Čas (ns)
č. 1	750666
č. 2	821857
č. 3	758316
č. 4	732929
č. 5	729884
č. 6	733309
č. 7	709719
č. 8	793045
č. 9	709994
č. 10	706949

Tabulka B.30: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Children.

B.7 Get Attributes by Name

B.7.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	1463744
č. 2	1747208
č. 3	1815316
č. 4	1489998
č. 5	1459559
č. 6	1624691
č. 7	1542886
č. 8	1771560
č. 9	1359109
č. 10	1708780

Tabulka B.31: Výsledky měření pro Titan(Persistit), test Get Attributes by Name.

B.7.2 OrientDB

Měření	Čas (ns)
č. 1	1502173
č. 2	1277304
č. 3	1335138
č. 4	1275782
č. 5	1332855
č. 6	1346173
č. 7	1283392
č. 8	1386124
č. 9	1300894
č. 10	1294426

Tabulka B.32: Výsledky měření pro OrientDB, test Get Attributes by Name.

B.7.3 JanusGraph (Cassandra)

Měření	**Čas (ns)**
č. 1	128750706
č. 2	128619355
č. 3	117167026
č. 4	125817429
č. 5	116996929
č. 6	124795249
č. 7	124905591
č. 8	124950109
č. 9	125329076
č. 10	125469096

Tabulka B.33: Výsledky měření pro JanusGraph (Cassandra), test Get Attributes by Name.

B.7.4 JanusGraph (BerkeleyDB)

Měření	**Čas (ns)**
č. 1	8416843
č. 2	8159967
č. 3	8484156
č. 4	7985497
č. 5	8408880
č. 6	8501720
č. 7	8217874
č. 8	8798501
č. 9	8187374
č. 10	8675923

Tabulka B.34: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Attributes by Name.

B.8 Get Children by Name

B.8.1 Titan (Persistit)

Měření	**Čas (ns)**
č. 1	999546
č. 2	983566
č. 3	1177235
č. 4	1301656
č. 5	1071459
č. 6	1009058
č. 7	982805
č. 8	951605
č. 9	1021995
č. 10	988512

Tabulka B.35: Výsledky měření pro Titan(Persistit), test Get Children by Name.

B.8.2 OrientDB

Měření	**Čas (ns)**
č. 1	1085918
č. 2	931438
č. 3	1070698
č. 4	1057381
č. 5	1038737
č. 6	1053957
č. 7	1040639
č. 8	1144894
č. 9	1090103
č. 10	1128533

Tabulka B.36: Výsledky měření pro OrientDB, test Get Children by Name.

B.8.3 JanusGraph (Cassandra)

Měření	**Čas (ns)**
č. 1	3970410
č. 2	3694482
č. 3	3165221
č. 4	3424949
č. 5	3866462
č. 6	3247098
č. 7	3261176
č. 8	3286288
č. 9	3536270
č. 10	3709013

Tabulka B.37: Výsledky měření pro JanusGraph (Cassandra), test Get Children by Name.

B.8.4 JanusGraph (BerkeleyDB)

Měření	**Čas (ns)**
č. 1	4164036
č. 2	3019800
č. 3	3283245
č. 4	4212782
č. 5	3955951
č. 6	3738692
č. 7	3888223
č. 8	4274800
č. 9	3297322
č. 10	3272971

Tabulka B.38: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Children by Name.

B.9 Get All Parents

B.9.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	562149
č. 2	598430
č. 3	574244
č. 4	513266
č. 5	535766
č. 6	479841
č. 7	628569
č. 8	450119
č. 9	668140
č. 10	482080

Tabulka B.39: Výsledky měření pro Titan(Persistit), test Get All Parents.

B.9.2 OrientDB

Měření	Čas (ns)
č. 1	1605286
č. 2	1774223
č. 3	1652466
č. 4	2139492
č. 5	1654749
č. 6	1647901
č. 7	1903970
č. 8	1656651
č. 9	1566095
č. 10	1618603

Tabulka B.40: Výsledky měření pro OrientDB, test Get All Parents.

B.9.3 JanusGraph (Cassandra)

Měření	**Čas (ns)**
č. 1	90563885
č. 2	88660911
č. 3	89570354
č. 4	87349290
č. 5	81209028
č. 6	93547924
č. 7	93699360
č. 8	93695554
č. 9	93604997
č. 10	93121395

Tabulka B.41: Výsledky měření pro JanusGraph (Cassandra), test Get All Parents.

B.9.4 JanusGraph (BerkeleyDB)

Měření	**Čas (ns)**
č. 1	4481319
č. 2	4013212
č. 3	4830315
č. 4	3956899
č. 5	4337875
č. 6	4400275
č. 7	4662285
č. 8	5248386
č. 9	5136990
č. 10	4812812

Tabulka B.42: Výsledky měření pro JanusGraph (BerkeleyDB), test Get All Parents.

B.10 Get Resource

B.10.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	872083
č. 2	998024
č. 3	912795
č. 4	957312
č. 5	1136142
č. 6	879692
č. 7	1020093
č. 8	1026561
č. 9	904043
č. 10	995361

Tabulka B.43: Výsledky měření pro Titan(Persistit), test Resource.

B.10.2 OrientDB

Měření	Čas (ns)
č. 1	603076
č. 2	832131
č. 3	538773
č. 4	716462
č. 5	523934
č. 6	543339
č. 7	529261
č. 8	561602
č. 9	552852
č. 10	565408

Tabulka B.44: Výsledky měření pro OrientDB, test Get Resource.

B.10.3 JanusGraph (Cassandra)

Měření	Čas (ns)
č. 1	105096461
č. 2	115158040
č. 3	115669038
č. 4	102636154
č. 5	106020212
č. 6	109130113
č. 7	108709672
č. 8	109190231
č. 9	109307421
č. 10	109540281

Tabulka B.45: Výsledky měření pro JanusGraph (Cassandra), test Get Resource.

B.10.4 JanusGraph (BerkeleyDB)

Měření	Čas (ns)
č. 1	8947346
č. 2	8265509
č. 3	8172462
č. 4	9079449
č. 5	9091244
č. 6	9084015
č. 7	9039878
č. 8	9182182
č. 9	8231818
č. 10	8434238

Tabulka B.46: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Resource.

B.11 Get Vertices by Edge Type

B.11.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	236596
č. 2	278830
č. 3	271601
č. 4	253026
č. 5	178831
č. 6	180733
č. 7	259494
č. 8	214596
č. 9	289933
č. 10	226011

Tabulka B.47: Výsledky měření pro Titan(Persistit), test Vertices by Edge Type.

B.11.2 OrientDB

Měření	Čas (ns)
č. 1	890726
č. 2	1033029
č. 3	1017049
č. 4	896052
č. 5	958454
č. 6	998785
č. 7	1031507
č. 8	938287
č. 9	906326
č. 10	929155

Tabulka B.48: Výsledky měření pro OrientDB, test Get Vertices by Edge Type.

B.11.3 JanusGraph (Cassandra)

Měření	Čas (ns)
č. 1	62979031
č. 2	65954387
č. 3	62744356
č. 4	66959918
č. 5	60512219
č. 6	49726942
č. 7	62418552
č. 8	62542210
č. 9	62646465
č. 10	62630103

Tabulka B.49: Výsledky měření pro JanusGraph (Cassandra), test Get Vertices by Edge Type.

B.11.4 JanusGraph (BerkeleyDB)

Měření	Čas (ns)
č. 1	3332309
č. 2	2716636
č. 3	3054570
č. 4	3276235
č. 5	3243894
č. 6	3316567
č. 7	3170006
č. 8	2970248
č. 9	3356679
č. 10	3408806

Tabulka B.50: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Vertices by Edge Type.

B.12 Simple Graph Flow

B.12.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	9154235
č. 2	10265846
č. 3	8713595
č. 4	10646862
č. 5	7507823
č. 6	7614740
č. 7	6940893
č. 8	7025742
č. 9	9452886
č. 10	8444969

Tabulka B.51: Výsledky měření pro Titan(Persistit), test Simple Flow.

B.12.2 OrientDB

Měření	Čas (ns)
č. 1	26630084
č. 2	27472488
č. 3	27782586
č. 4	26144198
č. 5	26668894
č. 6	26818426
č. 7	26518980
č. 8	27696976
č. 9	27354536
č. 10	26203935

Tabulka B.52: Výsledky měření pro OrientDB, test Simple Flow.

B.12.3 JanusGraph (Cassandra)

Měření	Čas (ns)
č. 1	103582772
č. 2	117031343
č. 3	109381432
č. 4	105219543
č. 5	1018727316
č. 6	1049141849
č. 7	1034464364
č. 8	1048600692
č. 9	1049479623
č. 10	1049680522

Tabulka B.53: Výsledky měření pro JanusGraph (Cassandra), test Simple Flow.

B.12.4 JanusGraph (BerkeleyDB)

Měření	Čas (ns)
č. 1	60223348
č. 2	62203942
č. 3	68279613
č. 4	43175786
č. 5	49172362
č. 6	52285747
č. 7	58415220
č. 8	49868864
č. 9	66330745
č. 10	64793567

Tabulka B.54: Výsledky měření pro JanusGraph (BerkeleyDB), test Simple Flow.

B.13 Get Node by Name

B.13.1 Titan (Persistit)

Měření	Čas (ns)
č. 1	209684072
č. 2	257481471
č. 3	352708250
č. 4	260858796
č. 5	207762599
č. 6	225221749
č. 7	226690439
č. 8	224406359
č. 9	226355990
č. 10	251188650

Tabulka B.55: Výsledky měření pro Titan(Persistit), test Get Node by Name.

B.13.2 OrientDB

Měření	Čas (ns)
č. 1	78237405
č. 2	87782366
č. 3	79735012
č. 4	92150006
č. 5	79380396
č. 6	80498654
č. 7	101526791
č. 8	80615464
č. 9	79219449
č. 10	80483435

Tabulka B.56: Výsledky měření pro OrientDB, test Get Node by Name.

B.13.3 JanusGraph (Cassandra)

Měření	Čas (ns)
č. 1	320725087
č. 2	331617687
č. 3	322937354
č. 4	320540209
č. 5	327464003
č. 6	368502580
č. 7	340759221
č. 8	324554932
č. 9	331397656
č. 10	365060709

Tabulka B.57: Výsledky měření pro JanusGraph (Cassandra), test Get Node by Name.

B.13.4 JanusGraph (BerkeleyDB)

Měření	Čas (ns)
č. 1	115077939
č. 2	108316174
č. 3	109573425
č. 4	92739419
č. 5	106582395
č. 6	101477746
č. 7	134873656
č. 8	111687764
č. 9	120957430
č. 10	122022140

Tabulka B.58: Výsledky měření pro JanusGraph (BerkeleyDB), test Get Node by Name.

Příloha C

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
  data ............................ testovací data
text .............................. text práce
  thesis.pdf ...................... text práce ve formátu PDF
```