

Big Data a NoSQL databáze

EDICE
PROFESSIONAL

 GRADA

Irena Holubová, Jiří Kosek
Karel Minařík, David Novák

Tato elektronická kniha byla zakoupena v internetovém knihkupectví **Grada.cz**

Jméno a příjmení kupujícího: **Jiří Fišer**

E-mail: **fiserj@g.ujep.cz**

Upozorňujeme, že elektronická kniha je dílem chráněným podle autorského zákona, a je určena jen pro osobní potřebu kupujícího. Kniha jako celek ani žádná její část nesmí být volně šířena na internetu, ani jinak dále zveřejňována. V případě dalšího šíření neoprávněně zasahujete do autorského práva s důsledky podle platného autorského zákona a trestního zákoníku.

Velmi si vážíme, že e-knihu dále nešíříte. Jen díky Vašim nákupům dostanou autoři, vydavatelé a knihkupci odměnu za svou práci. Děkujeme, že tak přispíváte k rozvoji literatury a vzniku dalších skvělých knih.

Máte-li jakékoli otázky ohledně použití e-knihy, neváhejte nás prosím kontaktovat na adresu **eknihy@grada.cz**

Tuto knihu bychom rádi věnovali:

Kryštofovi.

– Irena

*Rodině, která mne podpořila při práci na knize,
i když dobře věděla, co ji čeká.*

– Jirka

*Mým učitelům z Ústavu filosofie a religionistiky FF UK.
– Karel*

Sofince, která mi rostla před očima spolu s knihou.

– David

Upozornění pro čtenáře a uživatele této knihy

Všechna práva vyhrazena. Žádná část této tištěné či elektronické knihy nesmí být reprodukována a šířena v papírové, elektronické či jiné podobě bez předchozího písemného souhlasu nakladatele.

**Doc. RNDr. Irena Holubová, Ph.D., Ing. Jiří Kosek, Mgr. Karel Minařík
a RNDr. David Novák, Ph.D.**

Big Data a NoSQL databáze

Kniha je monografie

Vydala Grada Publishing, a.s.
U Průhonu 22, 170 00 Praha 7
tel.: +420 234 264 401, fax: +420 234 264 400
www.grada.cz
jako svou 6041. publikaci

Odborní recenzenti:
Doc. Ing. Michal Krátký, Ph.D.
RNDr. Jiří Materna, Ph.D.

Vydání knihy schválila Vědecká redakce nakladatelství Grada Publishing, a.s.

Odpovědný redaktor Petr Somogyi
Sazba Jiří Kosek
Grafické zpracování obrázků Milan Vokál
Návrh a zpracování obálky Vojtěch Kočí
Počet stran 288
První vydání, Praha 2015

Kniha byla připravena v XML formátu DocBook
a vysázena pomocí XSL-FO v programu XEP.

Vytiskla Tiskárna v Ráji, s.r.o., Pardubice

© 2015 Grada Publishing, a.s.

Cover Photo © fotobanka allphoto

ISBN 978-80-247-5939-5 (ePub)
ISBN 978-80-247-5938-8 (pdf)
ISBN 978-80-247-5466-6 (print)

Stručný obsah

O autorech	13
Předmluva	15

I. Pojem Big Data a principy distribuovaného zpracování dat

1. Úvod	19
2. Datové formáty	29
3. Základní principy	47
4. Zpracování dat pomocí MapReduce	63

II. NoSQL databáze

5. Základní principy NoSQL databází	87
6. Databáze typu klíč-hodnota	95
7. Dokumentové databáze	109
8. Sloupcové databáze	127
9. Grafové databáze	143

III. Pokročilé aspekty zpracování Big Data

10. Další aspekty zpracování Big Data	171
11. Dotazování nad NoSQL databázemi	193
12. Transakce v distribuovaném prostředí	205
13. Pokročilé aspekty grafových databází	217
14. Další databáze pro Big Data	243
Závěr	261
Použitá literatura	263
Rejstřík	273

Obsah

O autorech	13
Předmluva	15

I. Pojem Big Data a principy distribuovaného zpracování dat

1. Úvod	19
1.1 Jak velká jsou Big Data?	20
1.2 Historie a vznik NoSQL databází	22
1.2.1 Konec relačních databází?	25
1.3 O čem bude kniha	27
2. Datové formáty	29
2.1 JSON	30
2.1.1 JSON schéma	33
2.2 XML	35
2.2.1 XML schémata	37
2.3 YAML	39
2.4 Formáty Linked Data	41
2.4.1 RDF/XML	41
2.4.2 JSON-LD	42
2.5 CSV	43
2.6 Optimalizace ukládání a přenosu dat	44
2.6.1 Protocol Buffers	44
2.6.2 Apache Thrift	45
2.6.3 BSON	45
2.6.4 EXI a Fast Infoset	45
2.6.5 ASN.1	46
2.7 Jaký formát vybrat	46
3. Základní principy	47
3.1 Škálovatelnost	48
3.2 Konzistence	49
3.2.1 Souběh transakcí	50
3.2.2 CAP teorém	52
3.2.3 Občasná konzistence	54

3.3 Distribuce	56
3.3.1 Rozdělení dat (sharding)	57
3.3.2 Master-slave replikace	58
3.3.3 Peer-to-peer replikace	59
3.3.4 Replikace + sharding	61
4. Zpracování dat pomocí MapReduce	63
4.1 Funkce Map a Reduce	66
4.1.1 Další příklady	68
4.2 MapReduce framework	68
4.2.1 Další vlastnosti	70
4.3 Hadoop	72
4.3.1 HDFS	72
4.3.2 Hadoop MapReduce	74
4.3.3 Další nadstavby systému Hadoop	76
4.4 Kritika a ústup od MapReduce	82
 II. NoSQL databáze	
5. Základní principy NoSQL databází	87
5.1 Společné principy NoSQL databází	88
5.2 Datové modely v NoSQL databázích	89
5.3 Typologie NoSQL databází	93
6. Databáze typu klíč-hodnota	95
6.1 Principy	96
6.1.1 Základní operace a práce s klíči	96
6.1.2 Jmenné prostory klíčů	97
6.1.3 Druhy úložišť typu klíč-hodnota	98
6.2 Realizace a vlastnosti	99
6.2.1 Distribuce dat	99
6.2.2 Konzistence a dostupnost dat	102
6.2.3 Lokální organizace dat	105
6.3 Práce s daty	105
6.3.1 Sekundární indexy	106
6.3.2 Redis	106
7. Dokumentové databáze	109
7.1 Datový model „dokument“	109
7.2 Dotazování a manipulace s daty	115
7.2.1 Dotazy	115
7.2.2 Modifikace databáze	116
7.2.3 Agregované dotazy a MapReduce	117
7.2.4 Shrnutí	117

7.3	Vlastnosti dokumentových databází	118
7.3.1	Indexy	118
7.3.2	Replikace dat a dostupnost systému	119
7.3.3	Rozdělení dat	122
7.3.4	ACID pro jednotlivé operace a transakce	123
7.4	Závěr	124
8.	Sloupcové databáze	127
8.1	Datový model	128
8.2	Cassandra: datový model sloupců v praxi	132
8.2.1	Data jako multidimenzionální pole	133
8.2.2	Data jako řídké tabulky	134
8.3	Struktura a vlastnosti systému	136
8.3.1	Distribuce a replikace dat	136
8.3.2	Lokální organizace dat	137
8.4	Dotazy, indexy a transakce	138
8.4.1	Dotazy	139
8.4.2	Indexy	140
8.4.3	Transakce	141
9.	Grafové databáze	143
9.1	Typy grafů a související pojmy	145
9.2	Databáze Neo4j	146
9.2.1	Datový model Neo4j	146
9.3	Přístup k databázi Neo4j	147
9.3.1	Java API	147
9.3.2	Gremlin	149
9.3.3	Cypher	152
9.4	Pokročilé rysy Neo4j	156
9.4.1	Neo4j HA	156
9.4.2	Transakce	157
9.4.3	Indexy	158
9.5	Další grafové databáze	162
9.5.1	Sparksee	163
9.5.2	InfiniteGraph	163
9.5.3	OrientDB	163
9.5.4	Titan	164
9.6	RDF databáze	164
9.7	Srovnání úložišť pro grafy	165
9.8	Závěr	167

III. Pokročilé aspekty zpracování Big Data

10. Další aspekty zpracování Big Data	171
10.1 Analytické zpracování Big Data	172
10.1.1 Schéma dat	172
10.1.2 Tvorba datových skladů	175
10.1.3 Analytické zpracování	176
10.2 Vizualizace Big Data	178
10.2.1 Vizualizace propojených dat	179
10.2.2 Nástroje pro vizualizaci	181
10.3 Invertovaný index jako databáze	182
10.3.1 Apache Lucene a jeho nástavby	183
10.3.2 Zpracování logů	186
10.4 Cloud computing	187
10.4.1 Cloud computing a Big Data	189
11. Dotazování nad NoSQL databázemi	193
11.1 Přímý přístup pomocí programového rozhraní	194
11.2 MapReduce	196
11.3 Specifické dotazovací jazyky	196
11.3.1 Elasticsearch Query DSL	197
11.4 Univerzální dotazovací jazyky	199
11.4.1 Deriváty SQL	199
11.4.2 Rozšíření SQL	199
11.4.3 XQuery	202
11.4.4 JSONiq	203
11.4.5 SPARQL	203
11.5 Závěr	204
12. Transakce v distribuovaném prostředí	205
12.1 Vlastnosti CAP podrobněji	205
12.2 Základní transakční modely	206
12.2.1 Ploché transakce	207
12.2.2 Zřetězené transakce	207
12.2.3 Hnězděné transakce	207
12.3 Transakce v distribuovaném prostředí	208
12.3.1 2PC protokol	209
12.3.2 3PC protokol	210
12.4 Optimistické a pesimistické off-line zámky	210
12.4.1 Optimistický přístup	211
12.4.2 Pesimistický přístup	211
12.5 Uspořádání časových razítek	213
12.5.1 Pesimistické uspořádání	213
12.5.2 Optimistické uspořádání	214

12.6 MVCC	215
12.7 Závěr	216
13. Pokročilé aspekty grafových databází	217
13.1 Reprezentace grafů	217
13.1.1 Matice sousednosti	218
13.1.2 Seznam sousedů	218
13.1.3 Matice incidence	219
13.1.4 Laplaceova matice	219
13.2 Lokalita dat	220
13.3 Distribuce grafu	221
13.4 Dotazování nad grafy	223
13.4.1 Typy dotazů	224
13.4.2 Vyhodnocování dotazů a indexace grafových dat	225
13.4.3 Dotazovací jazyky pro grafy	234
13.5 Závěr	242
14. Další databáze pro Big Data	243
14.1 Hybridní databáze	243
14.1.1 PostgreSQL	244
14.1.2 MarkLogic	249
14.2 Databáze ve webovém prohlížeči	250
14.2.1 Web Storage	250
14.2.2 Indexed Database	252
14.3 NewSQL databáze	254
14.3.1 VoltDB	255
14.4 Array databases	256
14.4.1 SciDB	257
Závěr	261
Použitá literatura	263
Rejstřík	273

O autorech

Doc. RNDr. Irena Holubová, Ph.D. se habilitovala v roce 2014 v oboru informatika na MFF UK v Praze, kde v současné době působí jako docent na Katedře softwarového inženýrství. Současně externě působí na Katedře počítačů FEL ČVUT. Je autorkou více než 80 původních článků, které byly publikovány na mezinárodních konferencích a v impaktovaných časopisech, z oblasti analýz reálných dat a operací, odvozování XML schémat, XML benchmarkingu, generování testovacích dat a efektivní propagace změn v komplexních systémech týkajících se převážně semi-strukturovaných dat. Čtyři z nich získaly významná mezinárodní ocenění. Je spoluautorkou knihy „Technologie XML“ vydané v roce 2008 v nakladatelství Grada, která v tomtéž roce získala Cenu děkana MFF UK za nejlepší monografii. V rámci svého pedagogického působení (spolu)vytvářela na MFF UK a FEL ČVUT předmět „Technologie XML“ a „Pokročilé aspekty a nové trendy v XML“. V nedávné době vytvořila nový předmět „Big Data management a NoSQL databáze“. Na tyto oblasti také zaměřuje vedené bakalářské, diplomové a dizertační práce. Více informací je možné nalézt na stránce <http://www.ksi.mff.cuni.cz/~holubova/>.



Ing. Jiří Kosek již více než 15 let poskytuje školení a konzultace v oblasti webových a XML technologií. Celá generace tvůrců webu vyrostla na jeho knížkách o HTML a PHP a je i autorem řady článků vydaných jak v Česku, tak v zahraničí. Na půdě Vysoké školy ekonomické v Praze vytvořil a učí předměty zaměřené na webové technologie a XML. Ve svém volném čase spolupořádá a programově zajišťuje konferenci XML Prague.¹ Jirka se podílí na tvorbě a údržbě důležitých standardů v několika organizacích – zejména W3C, OASIS a ISO a přispívá do několika open source projektů. Více se o jeho aktivitách můžete dozvědět na jeho stránkách <http://www.kosek.cz> a <http://xmlguru.cz>.



¹ <http://xmlprague.cz>

Mgr. Karel Minařík je webový designér a vývojář. Vystudoval filosofii na FF UK. Věnuje se programovacímu jazyku Ruby, využití nerelačních databází a vizualizaci dat. V současnosti pracuje pro společnost Elastic.² Žije v Praze. Více informací naleznete na webových stránkách <http://karimi.cz>.



RNDr. David Novák, Ph.D. získal doktorát z informatiky v roce 2008 na Fakultě informatiky MU v Brně, kde nyní pracuje jako vědecký pracovník. Ve svém výzkumu se věnuje zejména technikám pro podobnostní vyhledávání, vyhledávání v multimédialních datech a distribuovaným datovým strukturám. Pracoval na více než deseti národních a evropských výzkumných projektech a je autorem třiceti publikací na mezinárodních odborných fórech. V roce 2014 zavedl na FI MU předmět o NoSQL databázích. V roce 2015 získal Fulbrightovo stipendium na semestrální pobyt na University of Massachusetts Amherst v USA. Více informací naleznete na jeho stránce <http://disa.fi.muni.cz/david-novak/>.



² <http://elastic.co>

Předmluva

Kniha *Big Data a NoSQL databáze* má tři hlavní cíle: vysvětlit pojem Big Data, představit svět NoSQL databází a objasnit jeho souvislost s Big Data. Rozhodli jsme se ji napsat, protože zatím žádná ucelená publikace na dané téma v češtině neexistovala.

Knihu jsme rozdělili do tří částí. V první části vysvětlujeme fenomén Big Data a principy distribuovaného zpracování dat. Ve druhé představujeme několik typů databázových systémů označovaných jako NoSQL databáze. Poslední část knihy přináší přehled dalších nových typů databázových systémů, popisuje pokročilejší aspekty distribuovaného zpracování dat a nabízí přehled dalších souvisejících technologií.

Na první pohled různorodý autorský kolektiv spojuje právě dlouhodobý zájem o oblast zpracování dat. Irena Holubová a David Novák působí v akademickém prostředí. Big Data a související technologie vysvětlují v širších souvislostech. Přináší srovnání s tradičními technologiemi a ukazují, jak fenomén Big Data ovlivnil přístup ke zpracování dat a vývoj databázových systémů. Jiří Kosek se ve své praxi i v této knize věnuje zpracování strukturovaných dat, formátům pro jejich ukládání a možnostem dotazování. Karel Minařík sleduje vývoj NoSQL databází od jejich počátku a aktivně se ho účastní. Knihu obohatil zejména o cenné postřehy z praxe. Konkrétně se na obsahu knihy autoři podíleli následovně: Irena Holubová – kapitoly 1, 3, 4, 9, 12, 13 a sekce 10.2, 10.4, 14.3, 14.4; Jiří Kosek – kapitoly 2, 11 a sekce 14.1.2, 14.2; Karel Minařík – sekce 6.3.2, 10.3, 14.1.1 a většina šedých doplňujících rámečků; David Novák – kapitoly 5, 6, 7, 8 a sekce 10.1.

Na tomto místě bychom rádi poděkovali prof. RNDr. Jaroslavu Pokornému, CSc., doc. RNDr. Vlastislavu Dohnalovi, Ph.D. a RNDr. Martinu Svobodovi, Ph.D. za přečtení rukopisu textu a řadu cenných připomínek, které přispěly ke zkvalitnění výsledku. Dále děkujeme RNDr. Davidu Hokszovi, Ph.D., RNDr. Filipu Zavoralovi, Ph.D., RNDr. Jakubu Klímkovi, Ph.D., RNDr. Leu Galambošovi, Ph.D., Ing. Vladimíru Kyjonkovi, RNDr. Jakubu Lokočovi, Ph.D., Mgr. Jindřichu Mynarzovi a Lence Koskové Třískové za kontrolu vybraných kapitol textu a odborné konzultace k nim. V neposlední řadě pak patří velký dík recenzentům, doc. Ing. Michalu Krátkému, Ph.D. z Vysoké školy báňské – Technické univerzity Ostrava a RNDr. Jiřímu Maternovi, Ph.D. ze společnosti Seznam.cz. Děkujeme také sponzorům za významnou finanční pomoc. Autoři byli při přípravě knihy částečně financováni z Programu rozvoje vědních oblastí na Univerzitě Karlově (PRVOUK) č. 204-04/1204 (Irena Holubová).

Do knihy se promítají dlouholeté zkušenosti autorů z výuky kurzů zaměřených na zpracování dat, Big Data a NoSQL databáze. Látka byla Irenou Holubovou zpracována pro jednosemestrální kurz v informatické sekci MFF UK, který vznikl v roce 2012 a dnes je povinně volitelnou součástí magisterské výuky. Obdobný kurz vytvořil David Novák v roce 2014 na FI MU v Brně. V roce 2016 bude kurz v upravené podobě Irenou Holubovou vyučován i na FEL ČVUT. Studentům uvedených i obdobných kurzů bude kniha sloužit jako studijní opora.

Kniha obsahuje velké množství příkladů, které je možné nalézt i na webové stránce <http://www.ksi.mff.cuni.cz/bigdata>. Na stránkách naleznete i opravy případných chyb a další informace. Připomínky a dotazy ke knize můžete zaslat na adresu bigdata@ksi.mff.cuni.cz.

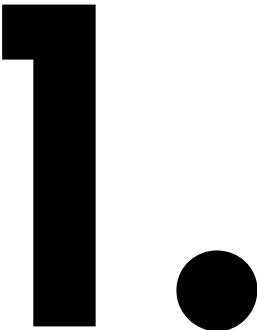
Přejeme vám příjemné čtení.

Autoři

Praha, Brno, Fryšava, Oldřichov v Hájích a Amherst, MA, USA, 25. září 2015

Část I.

**Pojem Big Data a principy
distribuovaného
zpracování dat**



Úvod

*Big Data is like teenage sex:
everyone talks about it,
nobody really knows how to do it,
everyone thinks everyone else is doing it,
so everyone claims they are doing it...*

—Dan Ariely

Slovní spojení Big Data naznačuje, že budeme mluvit o datech, jež jsou velká. Tiše předpokládáme, že data jsou digitální data – žijeme přece ve 21. století. Otázkou ale zůstává, jak velká musejí být data, aby se z nich stala Big Data.¹ Formální, přesnou a všemi přijímanou definici nenajdeme. Společnost Gartner,² v oblasti informačních technologií uznávaná výzkumná a poradenská společnost se sídlem v USA, definuje Big Data jako „data, jejichž velikost (**volume**), rychlosť nářstu (**velocity**) a různorodost (**variety**) neumožňují zpracování pomocí dosud známých a ověřených technologií v rozumném čase“ [67]. Tyto tři základní vlastnosti bývají označovány jako „3 V“. Postupně k nim přibývají i další „V“, jako např. nejistá věrohodnost (**veracity**) a vysoká hodnota (**value**) pro firmu, která je vlastní

¹ Stejně jako mnoho jiných pojmu z oblasti informačních technologií se ani pojem Big Data nepřekládá. Nebudeme ho tedy překládat ani my.

² <http://www.gartner.com>

[145], nebo limitovaná doba platnosti (**validity**) pro jejich využití a s tím související přechodná doba jejich nutného ukládání (**volatility**) [132].

Co tedy jsou Big Data? Jedná se o revoluci v IT, která znamená konec všech dosud užívaných nástrojů, nebo je to jen bublina, jež brzy splaskne? Přinesou nové aplikace a přístupy očekávané obrovské zisky a převratné vědecké výsledky, nebo sledujeme novodobou zlatou horečku? A odkud se berou Big Data?

Big Data se objevila především s příchodem nových technologií, služeb a jejich kombinací. Příkladem mohou být senzorové sítě nebo vědecké přístroje zkoumající přírodní jevy, sociální sítě nebo mobilní technologie a související aplikace. Tyto typy technologií a aplikací, s nemalou pomocí svých uživatelů, generují každou vteřinu obrovská množství dat, která potřebujeme efektivně uložit a účelně zpracovat. Myšlenku dobrě vystihuje např. společnost IBM:³ „V závislosti na odvětví a organizaci zahrnují Big Data informace z interních a externích zdrojů, jako jsou transakce, sociální média, podniková data, senzory a mobilní zařízení. Firmy mohou tato data využívat, aby lépe přizpůsobily své výrobky a služby potřebám zákazníka, dále optimalizovaly provoz a infrastrukturu a/nebo nalezly zcela nové zdroje příjmů“ [45].

Podívejme se blíže na jednotlivá „V“ z uvedené charakteristiky společnosti Gartner. Hovoříme-li o *velikosti* dat, máme na mysli datové kolekce takových objemů, které nedokážeme uložit na jeden databázový server, ale potřebujeme jich několik desítek či stovek. Množství dat v Big Data kolekci typicky *naruštá* v čase velmi rychle, často exponenciálně. Navíc velké objemy přibývajících a obměňujících se dat potřebujeme zpracovávat velmi *rychle*. Dalším rozměrem problému je *různorodost* dat. Na rozdíl od klasických strukturovaných dat např. v relačních databázích (*relational database management systems*, RDBMS), v oblasti Big Data hovoříme o datech semi-strukturovaných (např. obecně textové dokumenty nebo data ve formátech XML nebo JSON) nebo zcela nestrukturovaných (např. multimediální data). Protože kolekce Big Data často vznikají z různých veřejně dostupných zdrojů, mohou se potýkat s nižší *věrohodností*. Například data získaná vytěžováním textů ze sociálních sítí nemůžeme považovat za tak konzistentní, úplná a přesná jako data z databázových záznamů v uzavřených firemních systémech.

1.1 Jak velká jsou Big Data?

Abychom mohli objektivně měřit v přesných číslech, je dobré získat měřítko pro velikost dat. Uvědomme si nejprve, že v současné době má pevný disk v běžném osobním počítači kapacitu řádově až několik terabajtů (TB), tj. 10^{12} bajtů (B).⁴ Zajímavé odhady týkající se Big Data pak uvádí společnost IBM [44]. Např. odhaduje, že v roce 2016 bude existovat 18,9 miliard internetových připojení, tj. 2,5 připojení na každou osobu na Zemi. V roce 2020 pak podle jejich odhadů bude 6 miliard lidí na zemi vlastnit mobilní telefon, denně vznikne 2,8 kvintilionů

³ <http://www.ibm.com>

⁴ Dříve předpony kilo-, mega-, giga- atd. odpovídaly násobkům 1 024, tedy 2^{10} . Dnes pro ně ale používáme speciální zkratky KiB, MiB, GiB atd.

$(2,8 \times 10^{18})$ bajtů dat⁵ a celkově bude na discích uloženo 40 zettabajtů dat (kde 1 zettabajt je 10^{21} bajtů, tedy ekvivalent miliardy pevných disků o velikosti 1 TB).

Častým zdrojem Big Data jsou sociální sítě. Dle zjištění společnosti IBM [44] např. uživatelé serveru YouTube⁶ každý měsíc shlédnou více než 4 miliardy hodin videa. Sít Twitter⁷ má v současné době 200 milionů uživatelů, kteří každý měsíc vyprodukuje 400 milionů krátkých komentářů (*tweets*). A podle společnosti Zephoria [46] má Facebook⁸ (v době psaní této knihy, tedy jaro 2015) 900 milionů uživatelů aktivních každý den, kteří sdílí nějaký obsah více než 4,75 miliardy krát denně a na Facebook nahrají přes 300 milionů fotografií denně. V roce 2008 pro tyto účely využíval Facebook síť 10 000 serverů, v roce 2009 už to bylo 30 000 a odhadly z roku 2012 mluví o více než 180 000 serverech.

Dalším zajímavým příkladem je obchodování na burze. Zde dnes namísto lidí obchodují převážně počítače využívající komplexní matematické algoritmy. Rychlosť takových obchodů omezuje jen výkonnost hardwaru, tedy fyzikální vlastnosti použitých materiálů. Dle již uvedeného zdroje od IBM burza v New Yorku v současné době zachytí přibližně 1 TB dat během jednoho dne obchodování.

Obchodování na burze

Obrovský objem strojově generovaných dat, jako např. dat o obchodování na burze, přitom představuje problém nejen z prostého pohledu jejich ukládání a zpracování, ale především z pohledu jejich interpretace a analýzy. Dobrým příkladem může být projekt studia Stamen⁹ [136] [142], který vizualizuje data z jediného dne obchodování na burze NASDAQ¹⁰ (National Association of Securities Dealers Automated Quotations), po burze v New Yorku druhé největší svého druhu na světě.

Posledním pojmem, který zmíníme v souvislosti s příklady velkých dat, jsou *proudy dat (streams)*. Jedná se o specifický typ zdrojů, z nichž data do aplikace přicházejí ve formě nikdy nekončícího rychlého toku dat, který typicky není možné opakovat procházet. Data můžeme dočasně uložit, nicméně vzhledem k jejich neustálému nárůstu není možné uložit a zpracovávat celou množinu, ale pouze vybrané časové okno. S proudy dat se setkáváme např. v prostředí sítí – at už senzorových, počítačových nebo mobilních.

⁵ Při překladu názvů velkých čísel si musíme dát pozor na to, kterou škálu měl autor na mysli. Krátká škála, používaná ve většině anglicky mluvících zemích, mění předpony názvů čísel s násobkem tisíc (tedy např. bilion je tisícinásobek milionu). Dlouhá škála, používaná v kontinentální Evropě, používá pro každou předponu vždy dvě předpony (milion, miliarda) a předpony mění s násobkem milion. Tedy např. kvintilion v krátké škále odpovídá 10^{18} , zatímco v dlouhé škále 10^{30} .

⁶ <https://www.youtube.com>

⁷ <https://www.twitter.com>

⁸ <https://www.facebook.com>

⁹ <http://stamen.com>

¹⁰ <http://www.nasdaq.com>

Zdroje Big Data

Z uvedeného je zřejmé, že klíčovou charakteristikou Big Data není oproti běžné představě ani tak celkový objem dat, ale právě trvalost a rychlosť jejich nárůstu (velocity). Lidmi vytvářená data, jako jsou uváděné příklady sociálních sítí, přitom mají nějakou představitelnou, byť velmi vzdálenou mez; na celém světě je zkrátka pouze určitý počet lidí vybavených mobilním telefonem s fotoaparátem, připojeným k internetu a účtem na Facebooku. Ale např. již datům získávaným z obchodování na burze tento praktický limit do značné míry chybí. Dalším z rapidně rostoucích zdrojů dat jsou dnes informace o provozu strojů samotných, tedy např. logy webových a e-mailových serverů, sítových směrovačů, webových API (Application Programming Interface) a aplikací, atd.

V současnosti a blízké budoucnosti budou přitom strojově generovaná data představovat mnohem větší objem celkově ukládaných a zpracovávaných dat. „Každých 30 minut provozu vygeneruje jediný motor letounu Boeing 10 TB dat. Jeden zaoceánský let čtyřmotorového letounu tak vygeneruje asi 640 TB dat. Vynásobíme-li to asi 25 tisíci lety, které se uskuteční každý den, uděláme si představu o obrovském množství dat, které existuje“. [143]

Dat získávaných ze senzorů přitom bude jen přibývat, ať už se bude jednat o měříče zatížení vozovky zabudované v moderních dálnicích, nositelná zařízení v oblasti medicíny, RFID (Radio Frequency Identification) čipy v poštovních zásilkách nebo domácí spotřebiče propojené v tzv. *Internet of Things* (IoT). Ty všechny odesílají data prakticky neustále a často bez vědomí svého nositele či majitele.

S problémem rychlosti nárůstu je těsně spjat též problém *granularity*. Většina systémů dnes požaduje ukládat „surová data“ tak, jak přicházejí, protože nejsou dopředu známy všechny požadavky na jejich využití a stále častěji tak není možné „šetřit“ pomocí ukládání agregovaných dat.

1.2 Historie a vznik NoSQL databází

Donedávna nejpoužívanějším způsobem pro efektivní ukládání a správu dat byly tradiční databázové systémy, ať už relační, objektové, objektově relační, XML nebo další. Nejpopulárnějšími z nich jsou pak bezpochyby databáze relační, které vycházejí z jednoduchého, zato robustního matematického pojmu relace, který si většinou představujeme jako tabulku s řádky a sloupci. Všechny systémy tohoto typu mají mnoho společných rysů, jako je persistentní ukládání dat, řízení souběžného přístupu více uživatelů, dotazovací jazyky pro přístup k datům apod. Typicky jsou založeny na architektuře klient/server, která předpokládá, že máme veškerá data uložena na jednom výkonného uzlu (serveru), k němuž přistupují klientské programy. Pokud tedy potřebujeme uložit větší množství dat, zvýšíme diskovou kapacitu serveru. Pro zálohování nám obvykle stačí 1–2 další záložní servery (tzv. *zrcadla*) serveru primárního, na nichž jsou veškerá data duplikována. Jak je ale z vlastností Big Data zřejmé, pro jejich zpracování standardní databázové sys-

témy nestačí a potřebujeme přístupy nové. Ty jsou dnes souhrnně označovány jako *NoSQL databáze*, i když, jak si ukážeme později, nejedná se o tak jednoznačný termín jako v případě relačních nebo objektových databází.

Pojem „NoSQL“ pravděpodobně jako první použil Carlo Strozzi v článku [152] z roku 1998. Označil tak open source relační databázi, která nepodporovala klasický přístup k datům přes SQL [21]. V roce 2009 stejný pojem použil Eric Evans pro označení konference pro zastánce různých ne-relačních databází [82]. V současné době se jako NoSQL označují nové databázové systémy, které začaly vznikat začátkem nového tisíciletí pro podporu efektivního zpracování Big Data. Podle [81] se jedná o „novou generaci databázových systémů pro správu velkého množství dat, které jsou převážně ne-relační, distribuované, škálovatelné a podporují replikaci. Často jsou to databáze bez datového schématu, s jednoduchým rozhraním pro práci s daty a open source přístupem“.

V minulých desetiletích jsme mohli pozorovat vznik několika nových databázových konceptů, které s odstupem času můžeme vnímat spíše jako módní trendy, jež se příliš neprosadily. Např. objektové databáze začaly vznikat jako přirozenější úložiště pro struktury objektového programování. V dnešní době sice objektové programovací paradigma stále dominuje, ale použitost objektových databází je oproti relačním podstatně nižší.

Naopak postupný vznik NoSQL databází přichází jako odpověď na praktické potřeby společností internetové éry. NoSQL technologie jsou nyní buď vyvýjeny interně přímo pro potřeby společností jako je Google,¹¹ Amazon¹² či Facebook, nebo jsou rozvíjeny v režii open source komunit a relativně malých nově vzniklých firem. Vývoj NoSQL technologií probíhá většinou bez zásadnější účasti akademických komunit, ale tyto databáze často staví na solidních akademických výsledcích (např. v oblasti udržování konzistence dat v distribuovaných systémech). Často se jedná o teoretické výsledky publikované před desítkami let, pro jejichž širší využití dozrála doba až nyní, a to právě díky technologiím popisovaným v této knize.

Mezi hlavní uváděné výhody NoSQL databází patří především:

1. Flexibilní škálovatelnost: Zatímco klasické databázové systémy využívají *vertikální škálovatelnost*, tedy využívají výkonnější hardware, NoSQL databáze *škálují horizontálně*. Zpracování každé úlohy tyto systémy mohou distribuovat v rámci množiny uzlů (zvané *cluster*¹³), kterou je možné dle potřeby rozšiřovat přidáváním dalších uzlů.
2. Flexibilní datový model: NoSQL databáze nevyžadují určení žádného databázového schématu anebo je schéma dat volné. Dodržování tohoto schématu je často ponecháno na aplikaci a jeho změna neznamená pro databázi významnou zátěž.

¹¹ <http://www.google.com/about/company/>

¹² <http://www.amazon.com>

¹³ Pojem cluster může mít několik významů. V oblasti distribuovaného zpracování dat rozlišujeme především dva – konkrétní hardwarové řešení, kdy jsou jednotlivé výpočetní jednotky fyzicky umístěny v jednom nebo více stojanech a propojeny pomocí vysokorychlostních kabelů, popř. sdílející např. diskové pole, nebo obecně jakoukoli množinu síťově propojených počítačů. Pokud nebude řešeno jinak, budeme v textu nadále používat slovo cluster v tom druhém, obecnějším významu.

3. Orientace na efektivní čtení: NoSQL databáze vybízejí své uživatele k vytváření takových datových struktur, které budou nejlépe odpovídat často pokládaným dotazům. Každý takový dotaz je pak zpracován velmi rychle a systém může zvládat obrovské množství dotazů za jednotku času.
4. Ekonomický aspekt: Vzhledem k horizontální škálovatelnosti NoSQL databází neklademe na uzly, na něž ukládáme data, žádné speciální požadavky. Může se jednat o běžně dostupné, relativně levné počítače (*commodity hardware*).

Dynamické datové schéma

Přestože NoSQL databáze jsou v běžném pojetí téměř synonymní s „bezschématovými“ (*schema-less*) databázemi, v praxi samozřejmě data vždy nějaké schéma mají, jen nemusí být přesně definováno. Spíše se jedná o meta informace k datům, ne o schéma ve svém klasickém pojetí. Např. článek má autora a datum publikace. Záznam v logu webového serveru obsahuje URL a návratový status. Na rozdíl od relačních databází NoSQL databáze často přenáší zodpovědnost za správu schématu na aplikační logiku: přidáme-li do datového modelu článku nový atribut, musíme jej bud doplnit i k existujícím záznamům, což může být od určitého objemu dat ekonomicky nebo technicky vyloučené. Druhou možností je upříslit aplikaci tak, aby se vyrovnala s *heterogenním* datovým modelem.

V praxi existuje ještě další řešení, které zcela opouští tradiční model „data mají jeden autoritativní datový model a jedno jediné úložiště“. Zdrojová data ukládáme v surovém formátu a dávkovým zpracováním je pravidelně obohacujeme, transformujeme nebo agregujeme do podoby vyžadované aplikací či aplikacemi. Zdrojová data jsou neměnná (*write-once* či *immutable*) a aplikace pracuje s odvozenými datovými modely, kterých může být několik. (Konkrétní příklad s počítáním přístupů na webové stránky viz první a druhá kapitola knihy od Nathana Marze [126].)

Hlavními výzvami pro NoSQL databáze jsou pak:

1. Zralost: I když jsou existující NoSQL databáze velmi efektivní, ještě nedosahují léty prověřené robustnosti, na kterou jsme zvyklí u relačních databází.
2. Uživatelská podpora: Výhodou, ale současně i nevýhodou NoSQL databází je jejich vznik v rámci tzv. *start-up* projektů a fakt, že jsou typicky open source. Díky tomu je sice jejich vývoj překotný, ale na druhou stranu nemáme vždy k dispozici poskytovatele s dobrým jménem a silným zázemím (opět srovnáme-li situaci se světem relačních databází).
3. Administrace: NoSQL databáze jsou často masivně distribuovanými systémy, což a priori znamená složitější instalaci a také následnou údržbu.
4. Standardizace přístupu k datům: Zatímco relační databáze nabízejí standardizovaný přístup pomocí jazyka SQL, v NoSQL světě si prakticky každá databáze vyvinula vlastní dotazovací jazyk a programátorské rozhraní (API). Zadávání složitějších dotazů často vyžaduje netriviální programátorské znalosti.

5. Experti: Na trhu je stále poměrně velký nedostatek odborníků v oblasti NoSQL databází.

Zdá se, že pro čím dál větší počet aplikací a IT společností vychází bilance uvedených výhod a nevýhod pozitivně, protože technologie popisované v této knize jsou v dnešní době již široce používány. Tím nemáme na mysli pouze internetové giganty, kteří stáli u zrodu těchto databází, ale také stovky společností uváděných v sekcích „Our Customers“ webových prezentací různých NoSQL databází.¹⁴ Záplava dat nutí čím dál více běžných firem zamýšlet se nad možnostmi jejich databázové vrstvy, často se pak poohlížejí po technologiích popisovaných v této knize.

1.2.1 Konec relačních databází?

Jak je někdy chybně uváděno, pojem NoSQL neznamená „No to SQL“, neboli „Ne jazyku SQL“. Tyto databáze nemají nahradit relační (SQL) ani jiný typ databází. Jejich cílem je nabídnout řešení pro nové typy aplikací, pro něž stávající databázové systémy nebyly navrženy, a tudíž jim nevyhovují. NoSQL ale neznamená ani „Not only SQL“, čili „Nejen jazyk SQL“ nebo „Něco víc než SQL“. Například systémy Oracle DB¹⁵ nebo PostgreSQL¹⁶ podporují širokou škálu jiných principů než jen jazyk SQL, ale do skupiny NoSQL databází je neřadíme. Na druhou stranu existuje klasifikace NoSQL databází [81] na *pravé* (core), kterým bude primárně věnována tato kniha, a *nepravé* (non-core), kam můžeme zahrnout i všechny ostatní systémy, které nejsou založeny na relačním modelu, tedy např. objektové nebo XML databáze.

Příchod NoSQL databází neznamená konec tradičních relačních databází. Tyto systémy mají velké množství cílových aplikací, pro které je standardizovaný relační model vhodný. Desítky let vývoje přinesly velká množství efektivních technologií využívaných v relačních databázích – například techniky fyzické organizace dat na disku, vyhledávací indexy, optimalizace zpracování dotazů, implementace vyhledávacích operací a další. Nezanedbatelná hodnota existujících relačních databází je i v jejich prověřenosti časem, stabilitě, spolehlivosti a robustnosti, existenci standardizovaných rozhraní a dotazovacích jazyků pro komunikaci s nimi a v neposlední řadě také v záruce zachování silné konzistence dat. Těmito vlastnostmi se většina současných NoSQL úložišť pochlubit nemůže. Je ovšem pravděpodobné, že postupem času budou jednotlivé NoSQL databáze „dozrávat“ a tato situace se bude měnit.

V tabulce 1.1 na následující straně vidíme srovnání požadavků na zpracovávaná data, resp. předpokladů o těchto datech v prostředí databází relačních a NoSQL. Pochopitelně se v obou případech najdou výjimky a speciální případy, tabulkou je tedy třeba chápat jako krátký shrnující přehled obecných tezí.

¹⁴ Viz například <http://basho.com/about/customers/>, <http://planetcassandra.org/companies/>, <https://www.mongodb.com/who-uses-mongodb> nebo <http://neo4j.com/customers/>.

¹⁵ <https://www.oracle.com/database/>

¹⁶ <http://www.postgresql.org>

Tabulka 1.1: Relační vs. NoSQL databáze – předpoklady o datech a aplikaci

Relační databáze	NoSQL databáze
Integrita dat je zásadní.	Stačí, pokud je většina dat většinu času v pořádku.
Datový formát je konzistentní a dobře definovaný.	Datový formát nemusí být známý nebo konzistentní.
Předpokládáme dlouhodobé uložení dat.	Vzhledem k velkému množství dat často ukládáme pouze určité „časové okno“ (např. poslední měsíc, poslední rok).
Aktualizace dat jsou časté.	„Write-once/read-many“, tedy vložená data už typicky nejsou dále modifikována (nebo alespoň ne příliš často). Obvykle data neustále přibývají, aniž by byla modifikována. Již nepotřebné záznamy jsou pak smazány.
Předvídatelný (lineární) nárůst velikosti dat.	Nepředvídatelný (exponenciální) nárůst velikosti dat.
Nástroje pro dotazování dat umožňují přístup i ne-programátorům.	Typicky pouze programátoři píší (implementují) zpracování dat.
Probíhají pravidelné zálohy dat.	Pro řešení výpadků je využívána replikační dat.
Přístup k datům zajišťuje jediný server.	Data jsou umístěna na více serverech, přistupujeme tedy ke clusteru uzlů.

Na relační, resp. NoSQL databáze bychom tedy měli nahlížet jako na dvě různé možnosti pro ukládání dat. Každá má své výhody a nevýhody a je určena pro jiný typ aplikací. Z obecnějšího hlediska používáme pojem *polyglotní persistence*, který znamená, že nastala doba, kdy máme mnohem více možností využívat různé typy úložišť pro různé typy úloh a dat – a to například i pro různé části jednoho systému [144].

Svět před relačními databázemi

Prestože se může zdát, že relační databáze tu byly „odjakživa“, a NoSQL systémy je přicházejí „nahradit“, relační databáze byly ve své době revoluční změnou v pohledu na data a vyzývatelem *tehdy* tradičních databází, jako byl např. IBM IMS, který využíval pro datový model hierarchickou strukturu, nebo síťové databázové systémy, jako byl např. systém IDMS.

1.3 O čem bude kniha

Naším cílem je představit čtenáři problematiku zpracování Big Data zejména pomocí NoSQL databází. Podíváme se na obecné základní principy, na různé druhy NoSQL databází včetně bližšího pohledu na konkrétní populární systémy i na teoretičtější pozadí využívaných metod. Kapitoly knihy jsou rozděleny do tří částí.

První část knihy je věnována oblasti Big Data a distribuovaného zpracování dat obecně. Konkrétně v kapitole 2 nejprve ukážeme, jaké datové formáty typicky využíváme v oblasti Big Data a jaké jsou jejich výhody a nevýhody. V kapitole 3 si vysvětlíme základní pojmy a techniky využívané při práci s Big Data, jako je distribuce, replikace, škálování, konzistence apod. A v kapitole 4 si blíže představíme také programový model MapReduce, který je často používaným principem při distribuovaném zpracování dat.

Druhá část knihy je věnována vlastním NoSQL databázím. Nejprve v kapitole 5 podrobněji rozebereme společné prvky databázových systémů tohoto typu, principy datového modelování v NoSQL světě a uvedeme nyní již poměrně standardizovanou klasifikaci NoSQL databází. V následujících čtyřech kapitolách se podrobnejší seznámíme s hlavními třídami NoSQL databází, tedy systémy typu klíč-hodnota (kapitola 6), dokumentovými databázemi (kapitola 7), sloupcovými databázemi (kapitola 8) a grafovými databázemi (kapitola 9). Budeme se zabývat jejich hlavním zaměřením, vlastnostmi a technikami, pomocí nichž je zajištěna funkcionality těchto systémů. V každé z těchto kapitol typicky zvolíme jeden konkrétní systém využívaný v příkladech, kterými je text bohatě prokládán. Samozřejmě se budeme zamýšlet také nad tím, pro které typy úloh je daný typ databází vhodný a pro které méně.

V poslední části knihy se budeme věnovat pokročilejším problémům a přístupům v oblasti zpracování Big Data pomocí NoSQL databází. Abychom čtenáři netvrdili, že Big Data znamenají pouze databáze, úvodem si v kapitole 10 krátce vysvětlíme i další související pojmy, jako je analytické zpracování Big Data, jejich vizualizace, cloud computing nebo fulltextové vyhledávání. Dále se podíváme na oblasti jako je např. efektivní dotazování (kapitola 11) nebo transakce v distribuovaném prostředí (kapitola 12). Zaměříme se také podrobněji na oblast grafových NoSQL databází (kapitola 13), které se od ostatních tří typů výrazně liší, a představíme si také hybridní databáze, NewSQL databáze a další speciální typy databází pro Big Data (kapitola 14). V závěru se pokusíme shrnout aktuální stav oblasti a diskutovat předpokládaný další vývoj.

2.

Datové formáty

2. Datové formáty

Možná přemýšlít, proč kniha o databázích začíná kapitolou o datových formátech. Je to proto, že práce s moderními databázemi NoSQL znalost datových formátů vyžaduje. Na rozdíl od relačních databází, které staví na abstraktním relačním modelu dat, u NoSQL databází často formální datový model neexistuje nebo je tak jednoduchý, že vnitřní strukturu ukládaných dat řídí sama aplikace. Typicky k tomu používá právě některý existující široce rozšířený formát. Například databáze typu klíč-hodnota pracují s daty jako s nestrukturovaným binárním objektem a pro ukládání strukturovaných hodnot používají nejčastěji formát JSON. Ve světě dokumentově orientovaných databází jsou typickými kandidáty formáty JSON a XML.

Znalost datových formátů je důležitá i pro celkový návrh aplikací používajících NoSQL databáze. Většina aplikací dnes pracuje ve webovém prostředí. Klíčové části aplikace jsou implementovány jako služby, které jsou využívány dalšími částmi aplikace, například klientskou aplikací napsanou v Javascriptu a běžící přímo ve webovém prohlížeči. Jednotlivé části aplikace si data předávají opět v přesně definovaném formátu. V ideálním případě je datový formát pro komunikaci shodný s formátem pro ukládání, aby se omezila režie potřebná pro konverzi dat.

Nyní se stručně seznámíme s jednotlivými běžně používanými datovými formáty.

2.1 JSON

Formát *JSON (Javascript Object Notation)* [68] [11] původně vznikl pro předávání dat mezi serverovou a klientskou částí webové aplikace. Jedná se o podmnožinu jazyka Javascript, která dovoluje reprezentovat základní datové struktury a umožňuje jejich přímočaré použití v prohlížeči. Postupem času se však JSON stal rozšířeným datovým formátem a knihovny umožňující jeho použití existují pro všechny běžně používané jazyky. Použití JSON v aplikaci je velice přímočaré, protože JSON lze snadno mapovat přímo na objekty daného jazyka.

JSON umožňuje reprezentovat čtyři jednoduché datové typy (řetězec, číslo, logická hodnota a `null`) a dva strukturované (objekty a pole). Pole a objekty mohou jako své prvky obsahovat libovolný další datový typ – jednoduchý i strukturovaný. JSON je tak velmi flexibilní a lze v něm reprezentovat v podstatě jakoukoliv datovou strukturu.

Příklad 2.1: Ukázka dokumentu JSON

```
{  
    "conferences": [  
        {  
            "name": "XML Prague 2015",  
            "start": "2015-02-13",  
            "end": "2015-02-15",  
            "web": "http://xmlprague.cz/",  
            "price": 120,  
            "currency": "EUR",  
            "topics": ["XML", "XSLT", "XQuery", "Big Data"],  
            "venue": {  
                "name": "VŠE Praha",  
                "location": {  
                    "lat": 50.084291,  
                    "lon": 14.441185  
                }  
            }  
        },  
        {  
            "name": "DATAKON 2014",  
            "start": "2014-09-25",  
            "end": "2014-09-29",  
            "web": "http://www.datakon.cz/",  
            "price": 290,  
            "currency": "EUR",  
            "topics": ["Big Data", "Linked Data", "Open Data"]  
        }  
    ]  
}
```

Logické hodnoty se zapisují jako `true` nebo `false`. Speciální hodnotou je rovněž `null`.

Řetězce se zapisují do uvozovek a mohou obsahovat jakýkoliv znak Unicode.¹ Speciální znaky je nutné přepisovat pomocí escape sekvencí.

Znak	Escape sekvence
"	\"
\	\\
/	\^a
BACKSPACE (U+0008)	\b
FORM FEED (U+000C)	\f
LINE FEED (U+000A)	\n
CARRIAGE RETURN (U+000D)	\r
TAB (U+0009)	\t

^a Lomítko lze zapisovat i klasickým způsobem, escape sekvence existuje jen pro úplnost.

Podobně je nutné přepisovat řídicí znaky (U+0000 až U+001F). Vkládají se pomocí speciální sekvence \u«kód», kde «kód» je Unicode kód znaku zapsaný v šestnáctkové soustavě. Tímto způsobem lze zapisovat i znaky, které by se jinak obtížně zadávaly na klávesnici. Následující dva řetězce tedy obsahují stejný text:

```
"@ Grada Publishing"
"\u00A9 Grada Publishing"
```

Situace je komplikovanější u přepisování znaků, které mají v Unicode kód větší než U+FFFF, tam je nutné použít zápis pomocí náhradních páru (*surrogate pairs*). Použijí se tedy dvě escape sekvence \u«kód» a z jednotlivých částí čísel za nimi se poskládá výsledný kód znaku v Unicode. Náhradní páry se používají v kódování UTF-16. V době vzniku Javascriptu se bohužel tento způsob zápisu přenesl i přímo do zápisu textových řetězců. Například znak žolíku ☺ (U+1F0DF) se přepisuje jako "\ud83c\udcfd". Ideální je proto podobné znaky zapisovat do dokumentů JSON přímo v kódování UTF-8.

Pro zápis čísel platí podobná pravidla jako ve většině programovacích jazyků pro datový typ **double**. Rozsah a přesnost čísel je dána tímto typem. Zápis je klasický, pro oddělení celé a desetinné části se používá tečka, před číslem je možné uvést znak - , případně +. Čísla je možné zapisovat i v exponenciálním tvaru, např. milion lze napsat jako 1e6 (tj. 1×10^6). Definice číselného datového typu v JSON je poměrně vágní. V Javascriptu, který má jen jeden číselný typ, je situace jednoduchá, použije se vždy ten. V ostatních jazycích se v závislosti na použitém parseru JSON mohou číselné hodnoty mapovat na různé datové typy – celočíselné, desetinné, navíc s různou přesností.

¹ <http://www.unicode.org>

Interoperabilita číselného typu ve formátu JSON

Specifikace JSON neřeší, na jaký datový typ se mají číselné hodnoty mapovat při dekódování. Záleží tak na tom, jak se chová daný parser JSON a jaké číselné typy nabízí daný programovací jazyk. Např. předpokládejme následující jednoduchý dokument JSON:

```
{ "n": 12345678901234567890 }
```

Kdybychom takový objekt načetli v Javascriptu, dostaneme hodnotu **12345678901234567000** (poslední tři platné číslice jsou zcela ignorovány). Oproti tomu například v Pythonu bude pracovat se správnou hodnotou z dokumentu JSON.

Většina jazyků používá pro reprezentaci čísel datový typ **double**, který umí reprezentovat zhruba 15 platných číslic. Potřebujeme-li přenášet čísla s větší přesností nebo rozsahem, je vhodnější hodnoty ukládat jako řetězec a v aplikaci je pak konvertovat na odpovídající číselný datový typ s větší přesností nebo rozsahem. Jinak se můžeme dočkat podobných překvapení a nechtěných oříznutí hodnot jako ve výše uvedeném příkladě.

Pole se v JSON zapisují do hranatých závorek a jednotlivé prvky pole se oddělují čárkou a mohou mít odlišné datové typy. Ukázka dvou různých polí by mohla vypadat takto:

```
[1, 2, 3]
[true, "Ahoj", 123, "Nazdar"]
```

Posledním podporovaným datovým typem jsou objekty. Objekt je neuspořádaná množina dvojic jméno-hodnota. Jako hodnotu je možné použít jakýkoliv další datový typ, takže není problém do sebe objekty zanořovat nebo použít jako hodnotu celé pole atd. Svým pojetím je tak objekt JSON podobný spíše asociativnímu poli.

Objekt se ohraničuje složenými závorkami, jednotlivé dvojice jméno a hodnota jsou odděleny čárkou. Jméno objektu je řetězec, a proto se zapisuje do uvozovek. Od hodnoty se odděluje dvojtečkou. Často se v souladu s terminologií objektově orientovaných jazyků říká těmto dvojicím vlastnosti. Např. následující objekt má tři vlastnosti **name**, **age** a **student**.

```
{
  "name": "Jan Novák",
  "age": 42,
  "student": false
}
```

Obecně je tedy formát JSON poměrně jednoduchý, ale přesto v něm lze snadno reprezentovat rozličné druhy údajů. Jednoduchost formátu má však i své nevýhody. Za významný nedostatek lze považovat zejména to, že syntaxe JSON nepodporuje

komentáře.² Není tak možné komentovat například složitější dokumenty JSON nebo pro potřeby ladění část dokumentu dočasně pomocí komentáře „vypnout“.

Proč se JSON stal tak populárním

Po roce 2000 se při vývoji webových aplikací stále častěji používala technika dnes známá jako AJAX (Asynchronous Javascript and XML) [89]. Ta spočívala v tom, že webová aplikace byla napsaná v Javascriptu a běžela v prohlížeči. Při potřebě zobrazení dalších dat na stránce se neposílal klasický požadavek na webový server, který by vrátil celou novou stránku, jež by překreslila tu stávající. Místo toho se přímo z Javascriptu vyvolal požadavek HTTP (pomocí objektu XMLHttpRequest) a server vrátil data, která se mají zobrazit. Data zachytily programy v Javascriptu a zobrazily je v požadovaném místě stránky. Nemusela se tak přenášet a překreslovat celá webová stránka a práce s aplikací se jevila jako mnohem plynulejší a příjemnější.

Jak už samotná zkratka AJAX napovídá, pro přenos dat mezi serverem a prohlížečem se používal formát XML. Bohužel webové prohlížeče pro práci s XML nabízejí pouze rozhraní DOM [50], které není moc pohodlné. Místo XML se tak pro přenos dat začal používat JSON – přenášená data šlo snadno načíst rovnou do javascriptového objektu a přímo s nimi pracovat. Nejprve se k načítání dat používala funkce eval() pro dynamické spouštění kódu, ale kvůli pomalému výkonu a bezpečnostním rizikům byla v prohlížečích nahrazena dedikovaným parserem JSON (metoda JSON.parse()).

2.1.1 JSON schéma

Vzrůstající obliba JSON pro výměnu dat vyvolala potřebu jazyka, který dovolí popsat strukturu zasílaných zpráv. Takový popis pak slouží jako jednoznačná definice formátu, ve kterém nějaká služba přijímá nebo poskytuje data. Obecně se takovému popisu struktury dat říká schéma. JSON Schema [87] [88] je jazyk pro zápis schémat pro dokumenty ve formátu JSON. JSON Schema umožňuje popsat přípustnou strukturu dokumentu – u objektů lze definovat jejich vlastnosti a jejich datové typy, podobně tak i u polí lze popsat datové typy jejich prvků. Máme-li k dispozici JSON Schema, můžeme vůči němu validovat jakýkoliv dokument JSON a zjistit, zda schématu vyhovuje nebo ne.

Příklad 2.2: Ukázka JSON Schema

```
{
  "$schema": "http://json-schema.org/schema#",
  "type": "object",
  "properties": {
    "conferences": {
      "type": "array",
      "items": {
        "type": "object"
      }
    }
  }
}
```

² Kuriózní na komentářích je to, že první verze JSON je podporovaly. Ale autor JSON Douglas Crockford je později odstranil. O vývoji formátu JSON a mimo jiné i o důvodech pro odstranění komentářů hovoří Crockford ve své přednášce The JSON Saga [72]. My i přesto budeme v některých příkladech v knize komentáře pro zvýšení přehlednosti používat.

```
"type": "object",
"properties": {
    "name": { "type": "string" },
    "start": { "type": "string", "format": "date" },
    "end": { "type": "string", "format": "date" },
    "web": { "type": "string" },
    "price": { "type": "number" },
    "currency": { "type": "string",
        "enum": ["CZK", "USD", "EUR", "GBP"] },
    "topics": {
        "type": "array",
        "items": {
            "type": "string"
        }
    },
    "venue": {
        "type": "object",
        "properties": {
            "name": { "type": "string" },
            "location": {
                "type": "object",
                "properties": {
                    "lat": { "type": "number" },
                    "lon": { "type": "number" }
                }
            }
        },
        "required": ["name"]
    }
},
"required": ["name", "start", "end", "web", "price", "topics"]
}
}
```

Samotné schéma je opět JSON dokument reprezentující objekt. Vlastnost `$schema` určuje, jakou verzi JSON Schema používáme. Pomocí vlastnosti `type` určujeme datové typy pro jednotlivé části dokumentu. U objektů navíc ve vlastnosti `properties` popíšeme všechny vlastnosti (dvojice klíč a hodnota) objektu, u polí k tomu slouží vlastnost `items`.

Protože datové typy podporované přímo v JSON jsou velice omezené, nabízí JSON Schema prostředky pro lepší definici a kontrolu formátu dat. Kromě určení typu tak můžeme u řetězcových datových typů určit i formát – v našem příkladu tuto možnost používáme například pro datum. Tvar přípustných hodnot lze omezit i pomocí regulárních výrazů. A JSON Schema nabízí také další možnosti, které jsou popsány v [88].

Standardizace JSON Schema

Původně JSON vznikl a začal se používat pro výměnu dat, kde jednoduchost jeho použití převážila nad pokročilejšími vlastnostmi jiných technologií. Jeho oblíbá rostla a používal se ke stále složitějším účelům, kde již samotný JSON přestával stačit. Pro kontrolu a popis dat tak vzniklo JSON Schema, jehož standardizace probíhala na půdě IETF (Internet Engineering Task Force).³ V roce 2015 však standard stále nebyl dokončen, poslední verze draft-04 byla ze začátku roku 2013. Při výběru knihoven podporujících JSON Schema (např. pro validaci dat) je tak důležité sledovat, jakou verzi JSON Schema podporují. Rozhodně není vhodné používat nástroje, které nepodporují draft-04, ale jen starší verze.

2.2 XML

Značkovací jazyk *XML* (*eXtensible Markup Language*) [13] vznikl jako zjednodušení komplexního značkovacího jazyka SGML [26]. Značkovací jazyky umožňují doplnit prostý text o strukturu elementů, která mu dá význam. Původně se SGML a XML používalo zejména v oblasti elektronického publikování pro reprezentování rozsáhlých dokumentů s pevnou strukturou a potřebou lepšího propojení pomocí odkazů a prohledávání. Typickou oblastí nasazení XML tak byla například reprezentace právních textů (zákon, precedenty, smlouvy) nebo technická dokumentace.⁴

V době, kdy XML vzniklo, byla zároveň velká poptávka po formátu, který by umožnil snadné propojení systémů a výměnu dat mezi nimi. XML šlo použít i k těmto účelům, a tak se stalo nejpoužívanějším formátem pro výměnu dat. Kromě výměny prostého XML vznikly i složitější nadstavby, jako jsou webové služby (například formát SOAP [39] nebo WSDL [51]), které samotné XML doplnily o další vrstvy jako popis rozhraní, autentizace atd. – viz např. [119]. XML se používalo i v odlehčenějších scénářích. Jak již bylo řečeno, před nástupem JSON to byl právě formát XML, který používaly AJAXové aplikace pro zasílání dat ze serveru do prohlížeče.

Samotný princip XML je velice intuitivní. Jednotlivé části dokumentu, které obsahují důležité informace pro další zpracování, označíme pomocí elementů. Např. cenu můžeme reprezentovat jako:

```
<price>120</price>
```

Řetězec 120 tvoří obsah elementu **price**. Element je v dokumentu vyznačen pomocí počátečního tagu **<price>** a koncového tagu **</price>**.

³ <https://www.ietf.org>

⁴ Ve formátu XML byla ostatně připravena i kniha, kterou právě držíte v ruce. Konkrétně se jednalo o formát DocBook [154].

Příklad 2.3: Ukázka dokumentu XML

```
<?xml version="1.0" encoding="UTF-8"?>
<events>
    <conference>
        <name>XML Prague 2015</name>
        <start>2015-02-13</start>
        <end>2015-02-15</end>
        <web>http://xmlprague.cz/</web>
        <price currency="EUR">120</price>
        <topic>XML</topic>
        <topic>XSLT</topic>
        <topic>XQuery</topic>
        <topic>Big Data</topic>
        <venue>
            <name>VŠE Praha</name>
            <location lat="50.084291" lon="14.441185"/>
        </venue>
    </conference>
    <conference>
        <name>DATAKON 2014</name>
        <start>2014-09-25</start>
        <end>2014-09-29</end>
        <web>http://www.datakon.cz/</web>
        <price currency="EUR">290</price>
        <topic>Big Data</topic>
        <topic>Linked Data</topic>
        <topic>Open Data</topic>
    </conference>
</events>
```

Elementy mohou obsahovat buď přímo hodnotu, nebo další elementy. XML samo o sobě nemá datové typy, veškeré hodnoty v dokumentu se chápou jako text. XML se však běžně používá společně s XML schématy, které popisují nejen povolenou strukturu dat, ale právě i datové typy jednotlivých elementů.

Kromě elementů mohou dokumenty XML obsahovat i atributy. V ukázce se jedná například o atribut **currency** u elementu **price**:

```
<price currency="EUR">120</price>
```

Dokumenty mohou obsahovat i komentáře, které se zapisují stejnou syntaxí, jakou používá jazyk HTML [17]:

```
<!-- Komentář -->
```

Jako znakovou sadu používá XML Unicode a pro zajištění maximální interoperability je nejlepší pro přenos a ukládání dokumentů používat kódování UTF-8. Pokud nějaký znak nemůžeme zapsat přímo na klávesnici, můžeme využít odkaz na znakovou entitu. Dříve zmíněný znak žolíku tak můžeme zapsat jako `🃟`.

Některé znaky jsou v XML vyhrazené a zapisují se odkazem na znakovou entitu.

Znak	Přepis pomocí znakové entity
<	<
&	&
>	> ^a
"	" ^b
'	' ^c

^a Většítko se musí přepisovat, pouze pokud mu předchází posloupnost znaků]].

^b Uvozovky se musí přepisovat, pouze pokud jsou uvedeny v hodnotě atributu, který uvozovky používá pro ohraničení hodnoty.

^c Apostrof se musí přepisovat, pouze pokud je uveden v hodnotě atributu, který apostrofy používá pro ohraničení hodnoty.

Formát XML obsahuje ještě několik možností, jako jsou sekce CDATA, instrukce pro zpracování, entity a DTD. Pro praktické použití XML však nejsou úplně podstatné, takže se jimi zde nebudeme zabývat.

2.2.1 XML schémata

XML již od svého vzniku umožňovalo použití schémat.⁵ Pomocí schématu lze přesně definovat, jaké elementy a atributy se v dokumentu mohou vyskytovat, v jakém pořadí a kolikrát se mají opakovat. Definovat lze i datové typy pro jejich obsah (co je řetězec, co číslo, co datum).

Využití XML schémat je poměrně široké. V první řadě samozřejmě přesně popisují určitý formát dat a lze je tak považovat za kontrakt, na jehož základě probíhá výměna dat mezi systémy. Schéma pak můžeme kdykoliv použít k *validaci* – kontrole toho, zda dokument vyhovuje všem omezením popsaným ve schématu.

Schémata však mají ve světě XML mnohem širší využití. Slouží pro přiřazení datových typů jednotlivým částem dokumentu XML – tyto informace pak využívají dotazovací jazyky nebo databáze při indexování obsahu. Ze schématu lze také snadno generovat přehlednou dokumentaci. A pokud je XML formát, ve kterém přímo autoři pořizují data, mohou využívat specializované editory XML, jež na základě schématu napovídají a dovolují vytvářet jen validní dokumenty XML.

Kontrola a čistota dat byla ve světě XML považována vždy za velmi důležitou. Postupem času tak vzniklo několik různých jazyků, ve kterých lze schéma zapsat. Ty se liší svými schopnostmi i elegancí zápisu. Podrobněji se o nich můžete dozvědět například v [120].

My si zde pro ilustraci ukážeme schéma pro náš ukázkový dokument ve dvou jazyčích pro popis XML schématu. Prvním z nich bude RELAX NG [23] v kompaktní syntaxi [24]. To je jazyk oblíbený a rozšířený mezi opravdovými XML geeky.

⁵ Historický předchůdce SGML použití schémat (v podobě jazyka DTD) dokonce vyžadoval. Jednou z revolučních myšlenek XML tedy bylo, že schéma pro dokument nemusí existovat.

Příklad 2.4: Ukázka schématu v kompaktní syntaxi RELAX NG

```
element events {  
    element conference {  
        element name { text },  
        element start { xsd:date },  
        element end { xsd:date },  
        element web { xsd:anyURI },  
        element price {  
            attribute currency { "CZK" | "USD" | "EUR" | "GBP" },  
            xsd:decimal  
        },  
        element topic { text }+,  
        element venue {  
            element name { text },  
            element location {  
                attribute lat { xsd:decimal },  
                attribute lon { xsd:decimal }  
            }?  
        }?  
    }+  
}
```

Zápis je velice jednoduchý a intuitivní. Klíčová slova **element** a **attribute** definují nové elementy, resp. atributy a ve složených závorkách se pak definuje jejich obsah. Jako obsah lze definovat různé kombinace dalších vnořených elementů a atributů. Má-li element či atribut obsahovat již jen prostou hodnotu, určí se jeho datový typ. Můžeme použít bud některý z datových typů W3C XML Schema (např. **xs:date**, **xs:decimal**) [54], definovat výčet povolených hodnot (v našem příkladu je výčet použit pro kód měny), nebo říci, že nás další kontrola nezajímá a hodnota může být libovolná (**text**). Jazyk RELAX NG nabízí i prostředky pro definici vlastních datových typů včetně implementace odpovídajícího validačního kódu.

Za složenou závorkou je pak možné uvést indikaci toho, kolikrát se daný element může opakovat, zda je povinný nebo volitelný. Používají se obvyklé znaky jako ?, * a +, které známe například z regulárních výrazů.

Druhý příklad ukazuje jazyk W3C XML Schema. Ten je dnes ze všech jazyků pro popis XML schématu nejpoužívanější a zároveň má širokou podporu ve všech možných nástrojích.

Příklad 2.5: Ukázka schématu v jazyce W3C XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xss: schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xss:element name="events">
    <xss:complexType>
      <xss:sequence>
        <xss:element name="conference" maxOccurs="unbounded">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="name" type="xss:string"/>
              <xss:element name="start" type="xss:date"/>
              <xss:element name="end" type="xss:date"/>
              <xss:element name="web" type="xss:anyURI"/>
              <xss:element name="price">
```

```

<xs:complexType>
  <xs:simpleContent>
    <xs:extension base="xs:decimal">
      <xs:attribute name="currency" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="CZK"/>
            <xs:enumeration value="USD"/>
            <xs:enumeration value="EUR"/>
            <xs:enumeration value="GBP"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
</xs:element>
<xs:element name="topic" type="xs:string" maxOccurs="unbounded"/>
<xs:element name="venue" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="location" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="lat" use="required" type="xs:decimal"/>
          <xs:attribute name="lon" use="required" type="xs:decimal"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

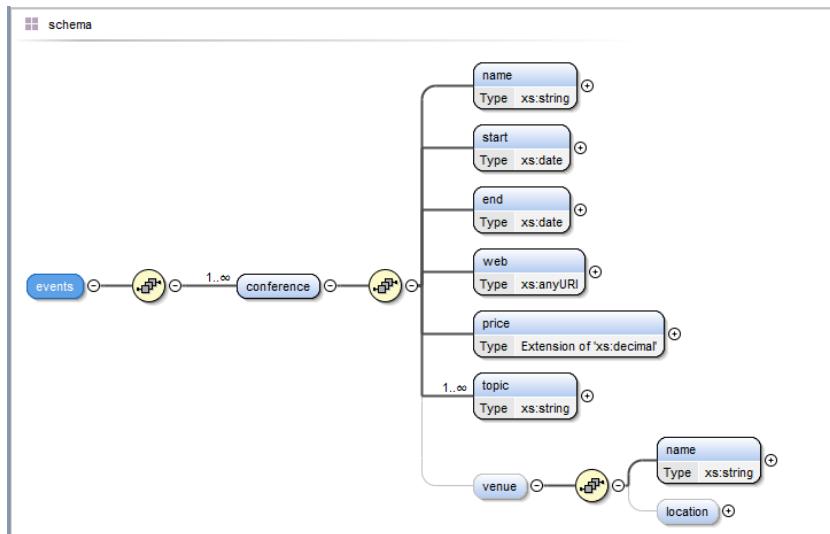
```

W3C XML Schema pro svůj zápis používá přímo formát XML, který není zcela elegantní a úsporný na zápis. Schéma je tak poměrně dlouhé a u rozsáhlých schémat nebývá snadné se v zápisu orientovat. Existuje proto mnoho nástrojů, jež umožňují pracovat se schématem ve vizuální podobě, která může být pro mnoho uživatelů přehlednější. Na obrázku 2.1 na následující straně je ukázka našeho schématu vizualizovaná pomocí editoru oXygen XML Editor.⁶

2.3 YAML

Formát YAML (YAML Ain't Markup Language) [64] byl navržen pro zápis dat v podobě dobře srozumitelné a čitelné pro člověka. Stále je přitom zachována možnost mapovat YAML na datové struktury běžně používané v programovacích

⁶ <http://oxygenvxml.com>



Obrázek 2.1: XML schéma znázorněné v grafické podobě

jazycích. V porovnání s JSON a XML není YAML zdaleka tak rozšířen, nicméně i tak je možné se celkem často setkat se systémy, kde se YAML používá zejména pro zápis konfiguračních souborů.

Mnohé jednodušší dokumenty YAML jsou velmi podobné dokumentům JSON a odpovídají tomu i základní datové typy – mapy (obdoba objektů), pole a skalární hodnoty. Pole a mapy lze do sebe samozřejmě zanořovat. YAML se snaží ulehčit ruční zápis souborů, a proto jeho syntaxe obsahuje mnoho užitečných zkratek. Například prvky pole nemusí být jen hodnoty oddělené čárkami uvnitř hranatých závorek, ale může se jednat o samostatné řádky uvozené znakem -. Vzájemné zanoření objektů a polí je v tomto případě dánno odsazením hodnot od začátku řádky. V mnoha případech není nutné řetězce uzavírat do uvozovek, nebo prvky pole či mapy oddělovat čárkami, stačí použít konec řádky. Zápis tak může být poměrně kompaktní a přehledný, jak ukazuje následující ukázka.

Příklad 2.6: Ukázka dokumentu YAML

conference:

```

- name: XML Prague 2015
  start: 2015-02-13
  end: 2015-02-15
  web: http://xmlprague.cz/
  price: 120
  currency: EUR
  topics: [XML, XSLT, XQuery, Big Data]
  venue:
    name: VŠE Praha
    location:
      lat: 50.084291
      lon: 14.441185
- name: DATAKON 2014
  start: 2014-09-25
  end: 2014-09-29

```

```
web: http://www.datakon.cz/
price: 290
currency: EUR
topics: [Big Data, Linked Data, Open Data]
```

Kromě jednoduchého zápisu nabízí YAML i pokročilé funkce, které v XML a JSON nenajdeme. V jednom souboru je možné mít více YAML dokumentů, data lze kromě přímého vložení načíst i referencí z jiné části dokumentu. Lze používat komentáře, datové typy a mapa může jako klíč používat i složenou hodnotu. Tyto pokročilé vlastnosti YAML mohou být v některých případech velice užitečné, ale na druhou stranu komplikují syntaxi a implementaci jazyka. Pro výměnu dat se tak jazyk YAML příliš neujal a tomuto poli stále dominuje JSON a XML, v oblastech s extrémními požadavky na rychlosť pak binární formáty dat, na které se podíváme v sekci 2.6.

2.4 Formáty Linked Data

Linked Data⁷ je iniciativa navazující na myšlenky sémantického webu. Snaží se propojit různé zdroje dat a mít pak možnost nad nimi provádět dotazy. Data se reprezentují pomocí datového modelu RDF (Resource Description Framework) [35], který je ve svém základu velice jednoduchý – skládá se z výroků složených ze tří částí – subjektu, predikátu a objektu. Například v klasické slabikářové věti „Ema má maso“ je „Ema“ subjektem, predikátem je vztah vlastnictví („má“) a objektem je „maso“. Aby bylo možné propojovat různé zdroje dat, je však potřeba, aby si pod pojmy jako „Ema“ nebo „má“ všichni představovali totéž. Ve světě Linked Data se proto všechny subjekty a predikáty reprezentují pomocí URI, která jsou unikátní a umožňují sdílení jednoho identifikátoru pro jednu věc napříč doménami. Pro běžně používané pojmy a druhy vztahů navíc postupně vznikají standardizované slovníky pojmu a vztahů (tzv. *ontologie*) – v současné době je jedním z takových nejpoužívanějších slovníků <http://schema.org>.

Datový model RDF je graf složený z jednotlivých výroků – subjekty a objekty tvoří uzly grafu a predikáty jsou hrany mezi uzly. Proto jsou ideálním úložištěm pro RDF grafové databáze. S nimi se blíže seznámíme v kapitole 9 a podrobněji se jim budeme věnovat v kapitole 13. Při přenosu RDF se používá některý ze serializačních formátů.

2.4.1 RDF/XML

RDF/XML je nejstarším formátem, který umožňuje uložení RDF grafu v podobě dokumentu XML. Jedná se o způsob zápisu, který není moc úsporný a snadno čitelný, a proto se v dnešní době často dává přednost dalším formátům jako Turtle [37] nebo N-triples [36].

Příklad 2.7: Ukázka informace o události, reprezentovaná pomocí RDF/XML

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
```

⁷ <http://linkeddata.org>

```

xmlns:schema="http://schema.org/">
<schema:Event>
  <schema:name>XML Prague 2015</schema:name>
  <schema:description>XML, XSLT, XQuery, Big Data</schema:description>
  <schema:startDate rdf:datatype="http://schema.org/Date">
    >2015-02-13</schema:startDate>
  <schema:endDate rdf:datatype="http://schema.org/Date">2015-02-15</schema:endDate>
  <schema:url rdf:resource="http://xmlprague.cz/">
  <schema:location>
    <schema:Place>
      <schema:name>VŠE Praha</schema:name>
      <schema:geo>
        <schema:GeoCoordinates>
          <schema:latitude rdf:datatype="http://www.w3.org/2001/XMLSchema#double">
            >50.084291</schema:latitude>
          <schema:longitude rdf:datatype="http://www.w3.org/2001/XMLSchema#double">
            >14.441185</schema:longitude>
        </schema:GeoCoordinates>
      </schema:geo>
    </schema:Place>
  </schema:location>
  <schema:offers>
    <schema:Offer>
      <schema:price rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
        >120</schema:price>
      <schema:priceCurrency>EUR</schema:priceCurrency>
    </schema:Offer>
  </schema:offers>
</schema:Event>
</rdf:RDF>

```

2.4.2 JSON-LD

Formát JSON-LD [27] reagoval na vznášející oblibu formátu JSON ve webovém prostředí a definoval standardní způsob, jak RDF graf přenášet v podobě dokumentu JSON. Většina dokumentů JSON-LD vypadá celkem přirozeně, používá se pouze několik vlastností se speciálními klíči jako `@context` a `@type` pro identifikaci použité ontologie a typu.

Příklad 2.8: Ukázka informace o události zachycené v podobě JSON-LD

```
{
  "@context": "http://schema.org",
  "@type": "Event",
  "name": "XML Prague 2015",
  "startDate": "2015-02-13",
  "endDate": "2015-02-15",
  "url": "http://xmlprague.cz/",
  "location": {
    "@type": "Place",
    "name": "VŠE Praha",
    "geo": {
      "@type": "GeoCoordinates",
      "latitude": 50.084291,
      "longitude": 14.441185
    }
  }
}
```

```

    },
    "offers": {
        "@type": "Offer",
        "price": 120,
        "priceCurrency": "EUR"
    },
    "description": "XML, XSLT, XQuery, Big Data"
}

```

Oproti příkladu 2.1 na straně 30 nepoužíváme námi zvolené názvy (klíče) jednotlivých vlastností, ale používáme standardizovaná jména definovaná na <https://schema.org/Event>.

Takto strukturovaná data lze navíc zařazovat přímo do webových stránek, kde je mohou využít vyhledávače. Kromě formátu JSON-LD lze přímo kód v jazyce HTML obohatit o několik speciálních atributů, které do stránky doplní potřebnou sémantiku. V současné době existují dva konkurenční způsoby zápisu – standardy RDFa [38] a Microdata [16].

2.5 CSV

Nakonec jsme si nechali formát *CSV* (*Comma-separated Values*). Jedná se o triviální formát pro ukládání dat, která mají podobu tabulky. Každá řádka textového souboru obsahuje jeden záznam s několika položkami, které jsou odděleny nějakým oddělovačem – typicky středníkem nebo čárkou. První řádka souboru může obsahovat názvy sloupců (položek).

Příklad 2.9: Ukázka souboru CSV

```

name;start;end;web;price;currency;topics;venue;lat;lon
XML Prague 2015;13.2.2015;15.2.2015;http://xmlprague.cz/;120;EUR;XML, XSLT, XQuery, ▶
Big Data;VŠE Praha;50.084291;14.441185
DATAKON 2014;25.9.2014;29.9.2014;http://www.datakon.cz/;290;EUR;Big Data, Linked Data, ▶
Open Data;;

```

Formát CVS je ale až příliš primitivní a neexistuje jeho standardizovaná podoba. Používají se tak různé oddělovače položek, různě se řeší situace, kdy položka obsahuje jako svoji hodnotu znak oddělovače. Neexistuje ani způsob, jak do souboru uložit informaci o použitém kódování.

Z těchto důvodů je dobré se formátu CSV pokud možno vyhnout. Nicméně z historických důvodů se jedná o formát, ve kterém je dostupné obrovské množství informací. Importu dat z tohoto formátu se tak v praxi často nevyhneme. CSV je častým zdrojem dat například v systému Hadoop, se kterým se seznámíme v sekci 4.3.

2.6 Optimalizace ukládání a přenosu dat

Všechny dosud zmíněné formáty jsou textově založené. Díky tomu jsou člověku snadno srozumitelné a při ladění aplikací, kde se tyto formáty používají, není potřeba používat žádné speciální nástroje na dekódování dat. Nicméně v aplikacích s extrémními nároky na rychlosť načítání dat nebo velikost přenášených dat nejsou textově založené formáty nejlepším řešením. V následujícím textu se proto stručně seznámíme s dalšími formáty, které data obvykle neukládají v textové, ale v kompaktnější binární podobě. Kromě vyšší efektivity přenosu a načítání dat naleznou tyto formáty uplatnění i v dalších oblastech. Například v databázích typu klíč-hodnota je často potřeba jako hodnotu ukládat nějakou složitější strukturu. Můžeme proto samozřejmě použít například formát JSON, ale při vyšších požadavcích na rychlosť načítání a ukládání dat můžeme využít následující technologie.

2.6.1 Protocol Buffers

Protocol Buffers [34] je formát pro serializaci strukturovaných dat vyvinutý firmou Google. Původně jej Google používal pro komunikaci mezi svými službami, ale dnes je implementace pro mnoho jazyků dostupná v podobě open source.

Strukturu dat musíme nejprve definovat v `.proto` souboru. Ten plní podobnou funkci jako schéma pro dokumenty JSON nebo XML. V Protocol Buffers je však použití schématu (`.proto` souboru) nutné. Samotná data uložená ve formátu Protocol Buffers nenesou informaci o své struktuře, tak jako v případě JSON nebo XML. Ze souboru `.proto` se pak dá vygenerovat kód, který dovoluje načítání a ukládání dat v požadovaném programovacím jazyce.

Příklad 2.10: Ukázka `.proto` souboru – schéma pro Protocol Buffers

```
Message Events {
    Message Location {
        required double lat = 1;
        required double lon = 2;
    }

    Message Venue {
        required string name = 1;
        optional Location location = 2;
    }

    Message Conference {
        required string name = 1;
        required uint32 start = 2;
        required uint32 end = 3;
        required string web = 4;
        required double price = 5;
        enum Currency {
            CZK = 0; USD = 1; EUR = 2; GBP = 3;
        }
        required Currency currency = 6;
        repeated string topic = 7;
        optional Venue venue = 8;
    }
}
```

```

    }
repeated Conference conference = 1;
}

```

Jak je vidět, definice formátu pomocí Protocol Buffers je podobná definici tříd nebo struktur v programovacích jazycích. Pomocí klíčových slov `optional`, `required` a `repeated` lze určit povinnost a počet opakování daného údaje. Za povšimnutí stojí to, že kromě datového typu a jména je každému údaji přiřazeno číslo – to se pak při přenosu používá pro identifikaci dané položky. Při aktualizaci `.proto` souboru (například přidávání nových údajů) se tato čísla nesmí měnit, jinak staré aplikace (napsané vůči předchozí verzi `.proto` souboru) nebudou schopné pracovat s novějšími zprávami.

2.6.2 Apache Thrift

Za vznikem Apache Thrift⁸ stojí společnost Facebook, která tento framework vyvinula pro snadné vytváření služeb, které spolu komunikují. Podobně jako Protocol Buffers byla tato technologie později uvolněna jako open source. Thrift kromě samotného popisu struktur zpráv nabízí i prostředky pro definici služeb a jejich rozhraní. Jinak je ale princip a použití podobné jako u Protocol Buffers – z popisu struktury dat (v tomto případě soubor `.thrift`) se automaticky vygeneruje potřebný kód.

Thrift navíc umožnuje kromě binárního formátu používat i textovou nebo JSON reprezentaci dat, což může usnadnit ladění aplikací. Oproti Protocol Buffers jsou podporovány i složitější datové typy jako seznam, mapa (asociativní pole) a množina.

2.6.3 BSON

BSON (Binary JSON)⁹ [4] je binární způsob zápisu JSON dokumentů. Původně vznikl jako způsob reprezentace dat v databázi MongoDB,¹⁰ ale lze jej používat i pro přenos dat. Kromě větší kompaktnosti a rychlejšího zpracování nabízí oproti JSON některá vylepšení – například podporuje binární data a má speciální datový typ pro datum a čas.

2.6.4 EXI a FastInfoSet

EXI (Efficient XML Interchange) [12] a FastInfoSet [25] jsou formáty, které definují binární způsob reprezentace dokumentů XML. S jejich použitím se setkáme jen v případech, kdy jsou požadavky na rychlosť načítání a úsporu přenášených dat opravdu extrémní. Některé implementace webových služeb například umějí automaticky přepnout na formát FastInfoSet, pokud jej podporují obě strany komunikace. EXI se používá zejména v úsporných komunikačních protokolech nasazených v průmyslovém a vojenském odvětví.

⁸ <http://thrift.apache.org>

⁹ <http://bsonspec.org>

¹⁰ <https://www.mongodb.org>

2.6.5 ASN.1

ASN.1 [20] je ze standardů zmíněných v této části nejstarší a nejpoužívanější. I když formát ASN.1 není mezi vývojáři příliš známý a málokdo jej používá přímo, mnoho současných protokolů a formátů používaných v oblasti internetu a telekomunikací je založeno právě na ASN.1. Při použití ASN.1 musíme pro danou zprávu nejprve vytvořit její definici (opět se jedná v podstatě o schéma). Pro ukládání dat pak musíme vybrat vhodnou notaci ASN.1 – těch existuje několik a liší se opět čitelností a efektivitou přenosu a načítání. Kromě několika různých binárních notací tak existuje i notace, kdy se data ukládají do XML.

2.7 Jaký formát vybrat

Seznámili jsme se s dnes nejpoužívanějšími datovými formáty. Přesto vám nedáme jednoznačnou odpověď na otázku, jaký formát je vhodný pro vaši aplikaci. Záleží vždy na konkrétní aplikaci. Ve světě NoSQL se dnes nejčastěji setkáme s formátem JSON, at při ukládání či přenosu dat. Zejména v kapitole 7 o dokumentových databázích uvidíte mnoho ukázek použití formátu JSON. V některých relačních databázích je formát JSON dostupný jako nový datový typ (viz sekce 14.1.1). Při použití databází typu klíč-hodnota, kde je rychlosť a kompaktnost ukládaných dat prioritou, však sáhneme nejspíše po binárních formátech jako Apache Thrift. Pro data s hodně složitou strukturou, která se navíc v čase mění a rozšiřuje, je stále nejlepší volbou XML.

Volba správného nástroje

Při řešení každého problému by měla být volba vhodného postupu a nástrojů samozřejmostí. Zní to velmi jednoduše, ale v praxi bývá toto pravidlo často porušováno. Někdy z neznalosti, jindy z nemožnosti prosadit byť sebemenší změnu ve velké setrvačné organizaci, jindy nás omezuje snaha ušetřit (většinou však jen krátkodobě).

Vzpomeňte si na toto pravidlo, až někde uvidíte naprostoto nesmyslné použití technologií. Například aplikaci spravující texty smluv krásně strukturovaných ve formátu XML, která je ale ukládá do textových polí relační databáze. Proč nebyla použita nativní XML databáze nebo alespoň nativní XML datový typ v relační databázi, aby šlo v dokumentech rozumně vyhledávat? Možná proto, že relační databáze už byla na nějakém serveru nainstalována a vývojář aplikace se nechtěl učit nic jiného.

Existují však i příklady z opačné strany, kde příliš aktivní vývojář zvolí zcela nevhodnou technologii, protože je zrovna v módě. Jak jinak si vysvětlit situaci, kdy se RDF trojice ukládají jako JSON dokumenty do dokumentové NoSQL databáze? Opět se jedná o neefektivní řešení, s pokrouceným datovým modelem a bez možnosti vyhledávání. Škoda, že se vývojář ve svém studiu módních trendů zřejmě bohužel nedostal až k databázím grafovým či přímo RDF.

3.

Základní principy

Jak jsme již zmínili v úvodu, jednou ze základních myšlenek zpracování Big Data je distribuce problému na cluster vzájemně propojených uzlů. Velikost clusteru, tedy počet uzlů, pak můžeme přizpůsobovat potřebám dané aplikace, přičemž jako jednotlivé uzly mohou sloužit běžné počítače. Přesto totiž dokáže výpočetní kapacita clusteru uzlů mnohonásobně převýšit výpočetní kapacitu jediného sebevýkonnéjšího serveru. Tuto výhodu ovšem pochopitelně nezískáme jen tak. Distribuované zpracování dat s sebou přináší i řadu problémů, které je třeba řešit, a omezení, s nimiž je třeba počítat.

Základním problémem distribuovaného zpracování jsou samozřejmě výpadky spojení uzlů. Díky nim můžeme (dočasně) ztratit přístup k celé podmnožině uzlů a tudíž i k datům na nich uložených. Typicky tento problém řešíme pomocí replikace, tedy uložení dat na více uzlech, pokud možno v různých částech sítě. Kromě problému, jak nalézt optimální způsob distribuce dat, ale s existencí více kopií dat vyvstává otázka udržování jejich konzistence. Z obecnějšího hlediska se jedná o problematiku transakčního zpracování požadavků na operace s distribuovanými daty.

Obecně se tedy snažíme řešit tento problém: Máme množinu různých objektů (např. souborů), které jsou uloženy na množině uzlů v clusteru. Navíc je každý objekt replikován na více uzlech. Pro jednoduchost předpokládejme pouze dvě

operace s našimi daty – **PUT** (neboli **WRITE**), která zapíše daný objekt na všechny repliky, a **GET** (neboli **READ**), která přečte požadovaný objekt z některé z jeho replik (přičemž v ideálním případě jsou na všech replikách totožné kopie). V této kapitole ukážeme, jakým způsobem je možné zajistit předpokládané korektní chování. Už předem můžeme prozradit, že budeme muset z některých požadavků obvyklých ve světě klasických relačních databází něco slevit. Bude to ale malá daň za velkou výhodu efektivního zpracování Big Data. Jak také ukážeme, z kompromisů, které distribuované zpracování dat vyžaduje, můžeme vždy vybrat ty, jež dané aplikaci působí nejmenší problém.

3.1 Škálovatelnost

Škálovatelnost je z hlediska problematiky zpracování dat vlastnost systému (popř. sítě, procesu apod.) flexibilně reagovat na měnící se požadavky, v našem případě zejména na zvyšující se objemy dat a zátěž systému. V tradičních databázových systémech se využívá *vertikální škálování* (*vertical scaling*, nebo také *scaling up*), tedy zvyšování výpočetního výkonu příslušného (jediného) serveru prostřednictvím robustnějšího HW. Tuto strategii je možné využít v mnoha aplikacích, ale jak už jsme naznačili v sekci 1.2, neumožňuje dosáhnout výpočetního výkonu, jaký vyžaduje zpracování Big Data. Časem se navíc může projevit problém nazývaný *proprietární uzamčení* nebo také *uzamčení poskytovatelem* (*vendor lock-in*). Výkonný hardware pro servery totiž vyrábí pouze omezené množství firem a velmi často se jedná o jejich proprietární řešení. Zákazník je tak nucen další výkonnější prvky nakupovat stále u stejného výrobce. Vertikální škálování má tedy především následující hlavní nevýhody:

1. Vyšší náklady: Výkonné servery jsou o několik řádů dražší než běžný HW.
2. Omezení nárůstu dat: I nejvýkonnější server má horní hranici svých možností.
3. Nutnost proaktivního přístupu: Při implementaci aplikace je třeba předem plánovat maximální velikost dat a tomu odpovídající HW.

Protipólem vertikálního škálování je *škálování horizontální* (*scaling out*), tedy distribuce problému na více uzlů. Odpadají nám všechny tři hlavní nevýhody vertikálního škálování, jelikož můžeme začít s clusterem několika běžných počítačů a postupně ho dle potřeby rozšiřovat, aniž bychom museli měnit architekturu a logiku systému – ty budou stejné nad clusterem 10, 100 i 10 000 uzlů. Tedy v podstatě umožníme zpracování neomezeně velkých dat. Mohli bychom říci, že horizontální škálování je optimálním řešením. To by ovšem platilo pouze v případě, že by platily i následující body [78]:

1. Síť je 100% spolehlivá.
2. Zpoždění (*latency*) je nulové.
3. Šířka pásma (*bandwidth*) je neomezená.
4. Komunikace po síti je bezpečná.

5. Topologie sítě se nemění.
6. Síť má jediného administrátora.
7. Náklady na přenos dat jsou nulové.
8. Síť je homogenní.

Jak je zřejmé, většina z těchto bodů je spíše utopií, ostatní jsou málo pravděpodobné nebo dosažitelné jen ve speciálních případech. V následujících sekcích proto ukážeme, jakým způsobem se distribuované zpracování dat snaží tomuto ideálu alespoň přiblížit.

3.2 Konzistence

Abychom mohli korektně a efektivně zpracovávat jakákoli data, potřebujeme především, aby byla *konzistentní*. Ve světě tradičních databázových systémů to znamená, že naše data v databázi splňují *integritní omezení*, tedy omezující podmínky, které si na ně z hlediska dané aplikace klademe. Mohou to být pravidla pro jednotlivé hodnoty (např. „věk osoby musí být hodnota z intervalu [0, 99]“) nebo jejich kombinace (např. „datum narození musí být menší než datum úmrtí“). Velmi častým omezením v tradičních databázových systémech je také *referenční integrita*, tedy korektnost odkazu cizího klíče v jedné tabulce na primární klíč v tabulce druhé.

Databázový systém pak pracuje na úrovni *transakcí*, což jsou sekvence logicky souvisejících operací, které převádí data z jednoho konzistentního stavu do druhého. Operacemi mohou být výše uvedené operace **READ** a **WRITE**, nejrůznější modifikace načtených dat a také dvě speciální operace – **COMMIT** (úspěšné ukončení transakce) a **ROLLBACK** (zrušení transakce při neúspěchu). V rámci transakce sice může být integrita dočasně narušena, pokud je to pro danou sekvenci operací potřeba, ale po skončení transakce musí být opět veškerá integritní omezení splněná. U transakcí proto vyžadujeme následující vlastnosti, souhrnně označované podle jejich počátečních písmen jako vlastnosti ACID:

- Atomicita (**atomicity**) neboli nedělitelnost databázové transakce – transakce budě proběhne celá, nebo neproběhne vůbec. Jinými slovy, operace **ROLLBACK** zajistí, že všechny doposud provedené změny nijak neovlivní stav databáze, zatímco operace **COMMIT** naopak zajistí trvalé zanesení změn do databáze.
- Konzistence (**consistency**), tedy zajištění přechodu z konzistentního do konzistentního stavu.
- Izolovanost (**isolation** nebo **independence**) znamená skrytí operací pobíhajících uvnitř transakce před ostatními běžícími transakcemi.
- Trvalost (**durability**), tedy zajištění, že se výsledky úspěšně provedených transakcí skutečně do databáze uloží.

Na základě vlastností ACID máme jistotu, že transakce proběhne korektně a nenařší tedy konzistenci dat v databázi. Mějme například transakci, která převádí částku X z účtu A na účet B . Transakce se bude skládat z několika kroků:

1. Zkontroluje, že je zůstatek na účtu A větší než X .
2. Z účtu A odečte částku X .
3. Přičte částku X na účet B .

Díky vlastnostem ACID máme zajištěno, že transakce např. při pádu celého systému neskončí po druhém kroku, tj. finance se „neztratí“.

Účetní v záznamech negumují

Je poměrně zvláštní, že tradičním příkladem a mentálním modelem pro koncept databázové transakce je transakce finanční, tedy převod peněz z jednoho účtu na druhý. Tento historický příklad totiž mlčky předpokládá, že jsou oba účty v jedné bance, což je jistě stále reálné, nicméně různé banky pravděpodobně nesdílí jednu databázi a tudíž se v tomto případě ve skutečnosti podobná technika při zacházení s penězi nepoužívá. Architektonicky i technicky bezpečnější a spolehlivější je v tomto případě ukládat všechny transakce v sekvenci a zrušení transakce (**ROLLBACK**) provést uložením kompenzační transakce: kupříkladu, pokud operace „přičti částku X na účet B “ neproběhne korektně, provede se a uloží se operace „přičti částku X (zpět) na účet A “. Ověření konzistence pak provádí další operace, která všechny transakce „sečeť“ (což, jak ukážeme v kapitole 12, vyžaduje další opatření). Jinými slovy: „účetní v záznamech negumují“ [58]. Pro technickou diskusi tohoto přístupu, zvaného např. *event sourcing*, viz [84].

3.2.1 Souběh transakcí

Vzhledem k tomu, že s daty na serveru typicky pracuje více uživatelů, z nichž každý spouští vlastní transakce, dochází k *souběhu transakcí*, tj. situaci, kdy server prokládá jednotlivé operace pro zvýšení celkové propustnosti.¹ Pokud každá transakce pracuje s jinými daty nebo pokud jsou data pouze čtena, můžeme jejich operace prokládat libovolně. Problém ale pochopitelně nastává v případě, že dvě různé transakce pracují se stejnými daty a alespoň jedna z nich data modifikuje. Pak může dojít k jedné ze tří *konfliktních situací*:

- *write-read konflikt*: V tomto případě dojde k situaci, že transakce T_1 zapíše do záznamu A novou hodnotu a ještě než provede zbyvající operace, transakce T_2 tato data přečte. Dojde tedy ke čtení nepotvrzených dat (*dirty read*). Je-li T_1 později zrušena, transakce T_2 pracuje s neplatnými daty. Situace je naznačena v následujícím příkladu:

Transakce T_1 :	Transakce T_2 :
<code>READ(A); // A = 12000</code>	
<code>A:=A-1000;</code>	

¹ Máme-li vícejádrový systém, může server na každém jádru provádět jinou transakci.

```

WRITE(A);
    READ(A); // čtení nepotvrzených dat
    READ(B);
    A := 1.01 * A;
    B := 1.01 * B;
    WRITE(A);
    WRITE(B);
    COMMIT;

READ(B);
B:=B+1000;
WRITE(B);
COMMIT;

```

- *read-write konflikt*: V tomto případě dojde k situaci, že transakce T_1 přečte hodnotu záznamu A a ještě než provede zbývající operace, transakce T_2 tato data přepíše. Problémem je, že transakce T_1 by při následném opakováném čtení záznamu A dostala jinou hodnotu. Operaci čtení tedy není možné zopakovat se stejným výsledkem (*non-repeatable read* neboli neopakovatelné čtení). Transakce T_1 tedy pracuje s hodnotou, která už není platná. Situace je naznačena v následujícím příkladu:

Transakce T_1 :	Transakce T_2 :
READ(A); // A = 12000	READ(A); READ(B); A := 1.01 * A; B := 1.01 * B; WRITE(A); // změna hodnoty A WRITE(B); COMMIT;
A:=A-1000; // A už má jinou hodnotu než v prvním kroku	
WRITE(A);	
READ(B);	
B:=B+1000;	
WRITE(B);	
COMMIT;	

- *write-write konflikt*: V tomto případě dojde k situaci, že transakce T_1 zapíše do záznamu A novou hodnotu a ještě než provede zbývající operace, transakce T_2 tato data přepíše. Dojde tedy k přepsání nepotvrzených dat (*lost update* nebo také *blind write*), tedy situaci, kdy se změna dat provedená v transakci T_1 v databázi vůbec neprojeví. Situace je naznačena v následujícím příkladu:

Transakce T_1 :	Transakce T_2 :
A:=10; WRITE(A);	B:=15; WRITE(B);
B:=10; WRITE(B); COMMIT;	A:=15; WRITE(A); // přepsání nepotvrzených dat COMMIT;

- Přidáme-li k operacím čtení a zápisu ještě možnost přidávání a mazání záznamů, může nastat další problém, tzv. *fantom*. V tomto případě dojde k situaci, že transakce T_1 položí dotaz nad určitou množinou záznamů a příslušné informace přečte. Ještě než dojde k jejímu potvrzení, transakce T_2 do databáze přidá nový záznam, který by tomuto dotazu také vyhovoval, a T_2 úspěšně skončí. V tomto okamžiku transakce T_1 nevrátí správný výsledek.

Operace v transakcích je tedy třeba prokládat obezřetně. Později ukážeme různé strategie, které se k tomuto účelu využívají právě v NoSQL databázích. Podrobněji se této problematice budeme věnovat v kapitole 12.

3.2.1.1 Úrovně izolace transakcí

Z obecného pohledu existuje několik *úrovní izolace transakcí*. Ne vždy totiž trváme na tom, aby se databázový systém snažil vyhnout všem možným konfliktním situacím. Typicky čím volnější pravidla stanovíme, tím efektivnější práci s daty získáme – samozřejmě ale za cenu výše uvedených problémů, s nimiž musíme počítat nebo je ošetřit jiným způsobem. Rozlišujeme následující úrovně izolace transakcí.

- Read uncommitted*: Nejnižší úroveň, při které transakce okamžitě vidí všechny změny provedené ostatními transakcemi. Může tedy nastat problém čtení nepotvrzených (a později zrušených) dat, problém neopakovatelného čtení i výskyt fantomů.
- Read committed*: Jedna transakce vidí data změněná druhou transakcí okamžitě po jejich potvrzení (operaci **COMMIT**). Nicméně stále může nastat problém neopakovatelného čtení nebo výskytu fantomů.
- Repeatable read*: Je zaručeno, že v průběhu transakce nemůže dojít k problému neopakovatelného čtení. Stále ale může dojít k výskytu fantomů.
- Serializable*: Nejpřísnější úroveň odpovídající situaci, jako by všechny paralelně zpracovávané transakce běžely ve skutečnosti jedna po druhé (tedy sériově). Nemůže tedy dojít ani k výskytu fantomů.

Pro zajištění příslušné úrovně izolace se využívají různé metody. Konkrétní přístupy představíme u jednotlivých typů NoSQL databází ve druhé části knihy, resp. v kapitole 12.

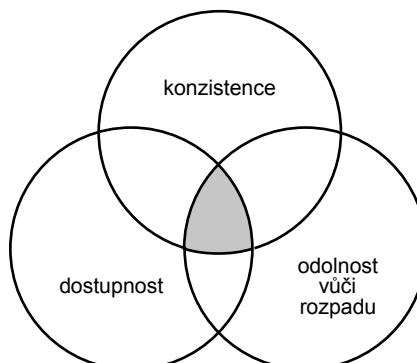
3.2.2 CAP teorém

I ve světě distribuovaných databázových systémů bychom jistě vlastnosti ACID uvítali. Nicméně jejich zajištění je vzhledem k distribuci, replikaci a výpadkům sítě velmi obtížné a systém by významně zpomalovalo. Používáme tedy jiný přístup a rozdíly si ukážeme pomocí tří ideálních vlastností (distribuovaného) databázového systému:

- Konzistence (**consistency**): Existuje jediná aktuální verze dat.

- Dostupnost (**availability**): Všechny požadavky na čtení, resp. zápis dat jsou vždy systémem obsluženy.
- Odolnost vůči rozpadu sítě (**partition tolerance**): Distribuovaný systém funguje, i když se rozpadne na několik izolovaných částí.

Jak je vidět na obrázku 3.1, naším ideálem je dosáhnout všech tří vlastností současně, tj. průniku uprostřed. Jak ale ukázal Eric Brewer v [70], v (distribuovaném) systému jsme schopni dosáhnout vždy maximálně dvou z těchto vlastností současně. Toto tvrzení se podle počátečních písmen vlastností nazývá *CAP teorém* nebo také *Brewerův teorém* podle svého autora.



Obrázek 3.1: CAP teorém

Např. tradiční centralizovaná databáze s podporou ACID transakcí zaručuje pouze konzistenci a dostupnost. Pokud však uvažujeme distribuovaný systém, potřebujeme především odolnost vůči rozpadu sítě a tím pádem nám nezbývá než do určité míry omezit konzistenci a/nebo dostupnost. (Přesněji řečeno, distribuovaný systém by byl bez odolnosti vůči rozpadu sítě jen těžko prakticky realizovatelný i využitelný.)

CAP teorém ovšem nelze brát doslova a jeho kritikové uvádějí hned několik bodů k diskusi. Především je to fakt, že původně použité definice nejsou zcela přesné a konkrétní teorém, který byl dokázán později, má další omezující podmínky. Dále jsou diskutabilní některé vlastnosti. Např. zatímco chybějící konzistence předpokládáme po celou dobu práce systému, chybějící dostupnost máme pouze v případě rozpojení sítě, tedy pouze někdy. Tyto dvě podmínky tudíž nemají symetrické vlastnosti. Nebo kdybychom chápali teorém doslovně, potom říká, že zajištění kombinace podmínek konzistence a odolnosti vůči rozpadu sítě znamená, že systém po rozpojení není dostupný vůbec. To je pochopitelně velmi špatná vlastnost.

Ve skutečnosti nám tedy CAP teorém spíše říká, že v distribuovaném prostředí musíme ze svých nároků na databázový systém v kontextu těchto tří podmínek poněkud slevit. Jak uvidíme ve druhé části knihy, různé NoSQL databáze nabízejí odolnost vůči rozpadu sítě a vhodný kompromis mezi zbylými dvěma vlastnostmi. Např. můžeme oželet silnou konzistence a získat tím částečnou dostupnost, nebo naopak.

Svět není černobílý

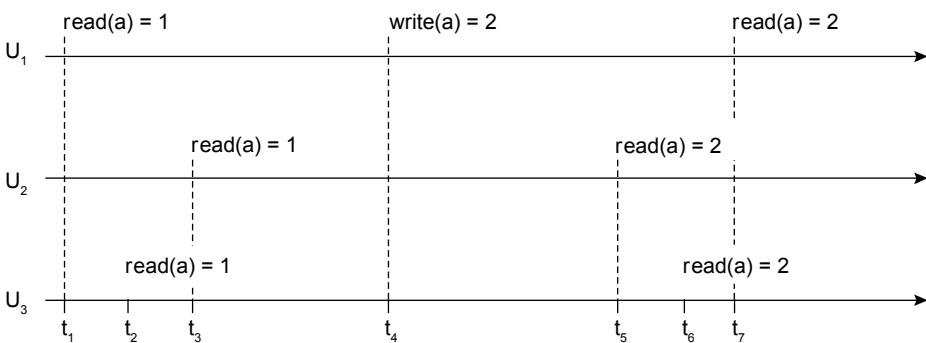
V praktickém uvažování o CAP teorému je třeba si uvědomit, že vlastnosti *konzistence* a *dostupnosti* nejsou černobílé, ale mohou být též „šedé“. Systém může například vyžadovat úplnou konzistenci pro určité kritické operace, ale slabou nebo žádnou pro operace ostatní. Jako příklad může sloužit procedura vybírání peněz z bankomatu, kterou uvádí Eric Brewer [69]: autor systému může upřednostnit silnou konzistenci před dostupností a zcela znemožnit výběry z bankomatu, pokud je bankomat offline (tedy došlo k „rozpadu sítě“). Tento „bezpečný“ přístup ale jistě neocení zákazníci, kteří si chtějí vybrat peníze. Autor systému proto může stanovit určitý umělý limit na výběr, např. 500 korun, a do jeho výše povolit výběry, i když je bankomat offline a nemůže tedy stoprocentně ověřit, zda daný zákazník na účtu daných 500 korun vůbec má – počítá přitom s tím, že kdyby se tak stalo, banka peníze získá v budoucnu vyrovnaným se zákazníkem. Výběry vyšších částek přitom může vázat na ověření konzistence, aby minimalizoval riziko vysoké pohledávky. (Uvedený příklad mimochodem dobře ilustruje, proč podobný systém nebude používat databázové transakce v naivním slova smyslu, ale kupříkladu event sourcing zmíněný v rámečku *Účetní v záznamech negují* na straně 50.)

3.2.3 Občasná konzistence

Alternativou k ACID vlastnostem tradičních databázových systémů je ve světě distribuovaných systémů model BASE. Díky němu za cenu nižší míry konzistence dat získáme škálovatelnost, která by v ACID prostředí nebyla možná. Jeho charakteristiky jsou tyto:

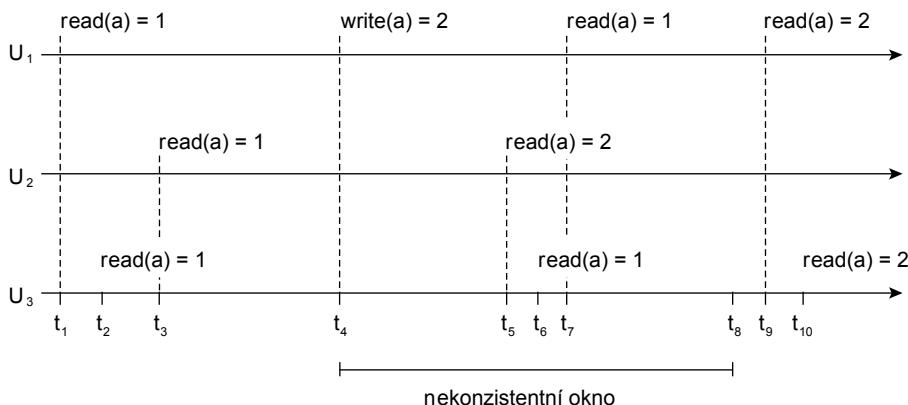
- Převážná dostupnost (**basically available**): Systém je převážně dostupný po celou dobu. Mohou se vyskytnout částečné výpadky, ale nikdy nedojde k výpadku celého systému.
- Volný stav (**soft state**): Systém je dynamický a nedeterministický, neustále dochází ke změnám.
- Občasná konzistence (**eventual consistency**): Čas od času je systém uveden do konzistentního stavu, nicméně konzistenci nemáme zaručenou neustále.

Občasná konzistence je tedy vlastně jakousi alternativou silné konzistence, kterou známe z tradičních databázových systémů. Uvažme nejprve příklad na obrázku 3.2 na následující straně, kde na ose X vidíme čas, na ose Y jednotlivé uživatele U_1 , U_2 , U_3 a zaznamenáváme operace, které provedli uživatelé nad systémem s podporou silné konzistence dat. V časech t_1-t_3 postupně jednotliví uživatelé přečetli hodnotu proměnné a , která je replikována na více uzlech v clusteru, přičemž všichni zjistili, že je 1. V čase t_4 uživatel U_1 zapsal do proměnné a novou hodnotu 2. V časech t_5-t_7 opět všichni uživatelé přečetli hodnotu a , přičemž dle očekávání všichni zjistili, že je 2.



Obrázek 3.2: Silná konzistence dat – příklad

Nyní uvažme obrázek 3.3, kde budou probíhat stejné operace, ale v systému s podporou občasné konzistence dat. V časech t_1-t_3 opět došlo ke čtení vícenásobně replikované proměnné a , která má pro všechny uživatele stejnou hodnotu 1. V čase t_4 uživatel U_1 opět zapsal do proměnné a novou hodnotu 2. Jak ale vidíme, v časech t_5-t_7 vrací operace čtení jak původní, tak novou hodnotu proměnné. Nyní je totiž systém v dočasně nekonzistentním stavu, tedy různé kopie proměnné a v distribuovaném systému obsahují různé hodnoty. Na každé replice je totiž zapsána jiná hodnota, kterou dostávají jako výsledek operace čtení jednotlivých uživatelů. V čase t_8 dochází k synchronizaci replik a systém se opět dostává do konzistentního stavu, což nám ukazují i následné operace čtení z proměnné a . Časový úsek mezi časem t_4 a t_8 nazýváme *nekonzistentní okno* (*inconsistency window*).



Obrázek 3.3: Občasná konzistence dat – příklad

Takové chování systému se může zdát zvláštní. Uvědomme si ale, že v distribuovaném systému o desítkách tisíc uzlů a milionech operací za vteřinu nemůžeme za každých okolností očekávat, že bude k synchronizaci dat docházet okamžitě. To by znamenalo příliš vysokou zátěž. Připomeňme také, že v předpokládaných aplikacích očekáváme vlastnost write-once/read-many, tedy nečekáme, že by často docházelo k modifikaci uložených dat, ale spíše k zápisu stále nových. Někdy se také tento přístup označuje za *optimistický*, tedy předpokládáme, že k problémům s konzistencí dat příliš často nedochází. A když už k problému dojde, umí ho daný systém řešit (nebo přinejmenším oznámit uživateli odpovídající chybu).

Naopak přístupy s podporou ACID transakcí bývají označovány za *pesimistické*, tj. v daných aplikacích je pravděpodobnost nekonzistence vysoká a je třeba jí preventivně předcházet.

Ukažme si ještě jeden praktický příklad z [144], kdy se můžeme setkat s občasní konzistencí dat. Předpokládejme, že máme systém pro rezervaci hotelů. Uživatelé U_1 a U_2 shodou okolností hledají stejný typ pokoje ve stejném městě a ve stejnou dobu. Oběma vrátí vyhledání v systému stejný výsledek – pokoj v hotelu H_1 , H_2 a H_3 . Uživatel U_1 se rychle rozhodne a zarezervuje hotel H_2 . Uživatel U_2 se k výběru vrátí později, provede sice aktualizaci výsledků hledání, ale vzhledem k tomu, že nás systém zajišťuje pouze občasní konzistenci, dostane tentýž výsledek, přestože v hotelu H_2 už pokoj k dispozici není. Když se přesto uživatel U_2 rozhodne pro tento pokoj a pokusí se o rezervaci, jeho operace nyní vyvolává požadavek na zápis a tím vynutí synchronizaci dat. Data jsou uvedena do konzistentního stavu přinejmenším vzhledem k datům, s nimiž aktuálně pracujeme. Díky tomu je zjištěna kolize a zápis neproběhne. Uživatel U_2 dostává chybové hlášení a aktualizovaný seznam výsledků původního hledání.

Silná vs. občasná konzistence

Myšlenka občasné konzistence vychází z náhledu, že pro mnoho aplikací je rychlosť mnohem důležitější vlastnosti než dokonalá konzistence (neboli, laicky řečeno, „správnost“ či „aktuálnost“ dat). Každá forma vynucené konzistence bude z principu zpomalovat načtení dat (at již ve formě zamýkání záznamů nebo získávání dat z více úložišť). Slovy Wernera Vogelse, CTO (Chief Technology Officer) společnosti Amazon.com: „Pokud provádíte operace pod vysokým zatížením a potřebujete dosáhnout nějaké formy shody, jste ztraceni“ [153].

3.3 Distribuce

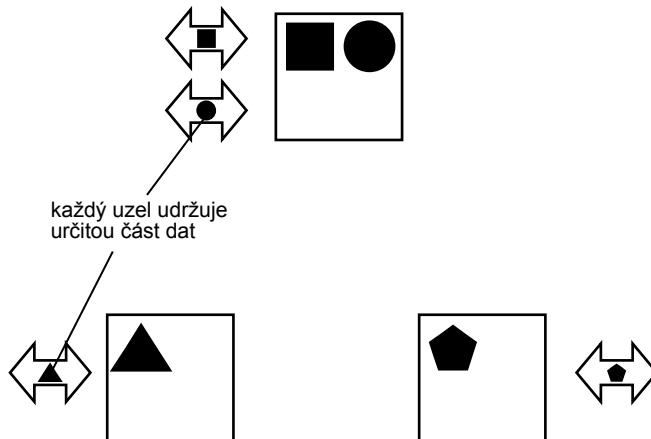
Již jsme několikrát zmínili, že Big Data svým rozsahem neumožňují zpracování v rámci jediného databázového serveru. Pro jejich optimální zpracování využíváme a kombinujeme dvě ortogonální techniky distribuce dat:

1. *Rozdělení (sharding)*, tj. rozmístění různých částí dat (*shards*) na různé uzly v clusteru, a tedy zvýšení kapacity systému.
2. *Replikaci*, tj. vytvoření kopí dat na více uzlech v clusteru, a tedy zvýšení dostupnosti a propustnosti systému.

Než vysvětlíme podstatu těchto principů, je třeba zmínit, že i ve světě NoSQL databází může mít využití také případ, kdy data nedistribuujeme, ale máme je uložena na jediném místě. Konkrétně v kapitole 9 se seznámíme s tzv. *grafovými databázemi*, u nichž je distribuce dat velmi obtížná (zvlášť je-li graf téměř úplný). Pokud je to vzhledem k velikostem dat možné, snažíme se je distribuovat pouze minimálně, ideálně vůbec, jelikož distribuce dat přináší také nemálo problémů.

3.3.1 Rozdělení dat (sharding)

Chceme-li horizontálně škálovat, musíme data rozdělit na vhodné (ne nutně disjunktní) podmnožiny a tyto uložit na různé uzly v clusteru, jak je naznačeno na obrázku 3.4 převzatém z [144]. Uživatelé pak tedy nepřistupují k jedinému serveru, ale každý k jinému uzlu (resp. k několika uzlům), podle toho, jaká data potřebují.



Obrázek 3.4: Sharding

Jak je patrné, strategie rozmístění dat je klíčová a výrazně ovlivňuje efektivitu systému. Typicky se snažíme o dosažení kompromisu mezi následujícími protichůdnými cíli (nebo vybereme jeden z nich):

1. Rovnoměrné rozmístění dat na uzlech.
2. Minimalizace počtu uzlů, na které musí uživatel přistoupit, aby požadovaná data získal.
3. Optimalizace fyzického rozmístění dat vzhledem k jejich geografické příslušnosti (např. měst, zemí, kontinentů, firem apod.).

Jak ukážeme ve druhé části knihy, většina NoSQL databází řeší rozdělení dat automaticky (má tzv. *auto-sharding*) a/nebo nabízí uživatelům volbu preferované strategie. Pokud ovšem dojde k výpadku sítě, k dané části dat ztrácíme přístup. Sharding je tedy často kombinován s replikací. Při replikaci obvykle volíme jednu ze dvou osvědčených strategií popsaných v sekcích 3.3.2 a 3.3.3.

Heterogenní cluster

Ve většině případů uvažujeme uzly clusteru jako rovnocenné. V mnoha případech ale můžeme cluster modelovat jako *heterogenní*, tedy přidělit jednotlivým uzlům různou výpočetní kapacitu, např. co se výkonu procesoru nebo disku týče. Praktickým důvodem může být kupříkladu rozdělení dat na „horká“ a „studená“: za „horká“ budeme považovat data z posledního měsíce, protože drtíva většina uživatelů se zajímá právě o ně, jelikož jsou zobrazována na domovské stránce aplikace atd. Umístíme je tedy na stroje se SSD (*solid-state drive*) disky a výkonnými procesory. Starší, „studená“ data pak můžeme umístit na slabší servery s běžnými magnetickými disky – načítání dat tak bude sice vzhledem k odlišné technologii nepoměrně pomalejší, ale jelikož k této datům uživatelé nepřistupují tak často, nebude je zvýšená latence příliš obtěžovat.

Podobně je tomu i s rovnoměrným rozložením dat na jednotlivých uzlech: v určitých případech můžeme chtít explicitně určit umístění konkrétních dat na konkrétní *shard*. Například když uživatelé systému přistupují výhradně ke „svým“ dokumentům, je výhodné dokumenty jednoho konkrétního uživatele uložit vždy v rámci jednoho uzlu a minimalizovat tak síťovou komunikaci při čtení dat.

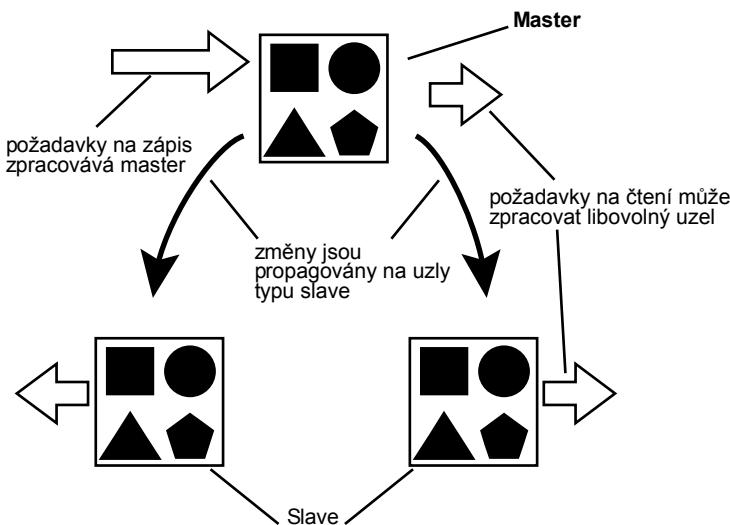
Pro podrobnější vysvětlení těchto přístupů na konkrétních příkladech např. v systému Elasticsearch² viz [62].

3.3.2 Master-slave replikace

Master-slave replikace zobrazená na obrázku 3.5 na následující straně, který jsme si opět vypůjčili z [144], předpokládá, že máme data replikována na určité podmnožině uzlů. Jeden z uzlů je určen jako *primární* neboli *master*, ostatní uzly jsou *sekundární* neboli *slaves*. Zatímco operaci čtení může obsloužit kterýkoli z uzlů, který má požadovaná data k dispozici, požadavky na zápis jsou předány na primární uzel, který je smí jako jediný provést a o této změně pak informuje sekundární uzly.

Tato strategie je vhodná pro data, jež jsou především čtena a minimálně modifikována. Když se zvyšují požadavky na čtení, prostě zvýšíme počet replik nebo sekundárních uzlů. Selže-li primární uzel, mohou sekundární uzly stále zpracovávat požadavky na čtení. A navíc vzhledem k tomu, že sekundární uzly mají kopii dat uzlu primárního, může se v případě potřeby z kteréhokoli z nich stát nový master poměrně snadno. Vlastní určení nového primárního uzlu provádí bud správce systému, nebo si ho uzly v clusteru zvolí na základě určitého algoritmu. Konkrétní algoritmy předvedeme v rámci popisu příslušných databázových systémů ve druhé části knihy.

² <https://www.elastic.co>



Obrázek 3.5: Master-slave replikace

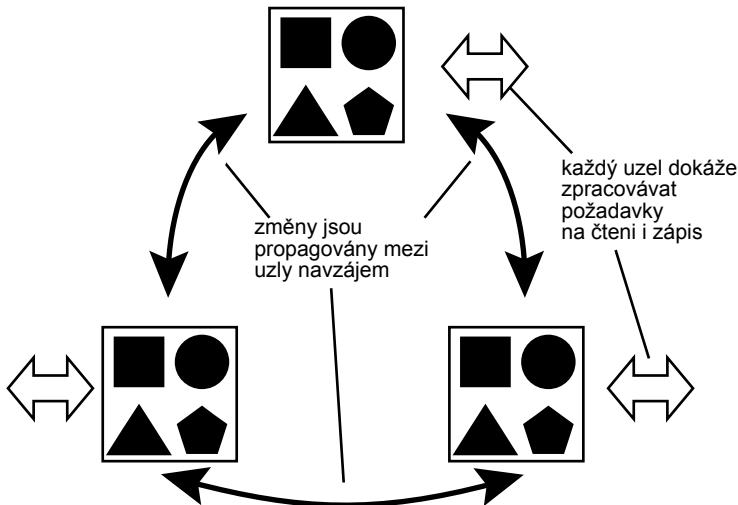
3.3.3 Peer-to-peer replikace

Centralizovaná správa požadavků na zápis při master-slave replikaci má zásadní nevýhodu, a tou je úzké hrdlo systému na uzlu master. Je-li požadavků na zápis více, nedokážeme je výrazně škálovat – jsme opět omezeni možnostmi jediného uzlu. Tento problém se snaží řešit druhá technika replikace dat, která se nazývá *peer-to-peer*³ nebo také *p2p*. Jak je znázorněno na obrázku 3.6 na následující straně, opět pocházejícím z [144], všechny uzly jsou si rovný, tj. mohou obsluhovat požadavky na čtení i na zápis. V případě výpadku některého z uzlů neztrácíme možnost čtení ani zápisu dat.

V peer-to-peer replikaci nám tedy odpadá problém s úzkým hrdlem na uzlu master a tedy malou mírou škálovatelnosti zápisů. Na druhou stranu vznikají problémy s udržením konzistence dat. Především vzniká hrozba write-write konfliktu, pokud se sejdou dva různé požadavky na aktualizaci stejných dat. Než se pustíme do rozboru možných řešení, ještě doplníme, že u obou přístupů (tedy master-slave i peer-to-peer replikace) hrozí také read-write konflikt, tedy čtení neaktualizovaných dat v době nekonzistentního okna. Ale zatímco read-write konflikt je dočasný do aktualizace dat, write-write konflikt může data poškodit trvale.

Konflikty můžeme řešit dvěma způsoby. V prvním z nich předpokládáme, že se při každém zápisu dat uzly v síti koordinují (domluví), což ale výrazně zvyšuje nároky na komunikaci a zpomaluje systém. Druhou možností je využití klasického volebního principu „rozhoduje většina“, který se realizuje prostřednictvím *kvóra*, tedy stanovujeme minimální počet uzlů, na nichž musí operace proběhnout, abychom ji mohli prohlásit za úspěšnou. Počet požadovaných replik dat na různých uzlech označujeme jako *replikační faktor*. Předpokládejme, že je jeho hodnota rovna N .

³Tento pojem by se dal přeložit jako rovnocenná replikace, ale obecně se v IT slovo peer-to-peer nepřekládá.



Obrázek 3.6: Peer-to-peer replikace

Uvažme nejprve zápis dat a hrozbu write-write konfliktu. Jedná se o situaci, kdy se dva uživatelé snaží provést zápis na stejném (replikovaném) místě databáze. Tako-vému konfliktu předcházíme tím, že si obě souběžné operace vyžádají potvrzení zápisu od několika (konkrétně W) replik a úspěšně dokončenou bude jen jedna z konfliktních operací. Na první pohled by se mohlo zdát, že potřebujeme potvrzení úspěšného zápisu od všech N replik. Ve skutečnosti však stačí, aby platila nerovnost $W > N / 2$, tzv. *kvórum zápisu (write quorum)*, tedy aby se podařilo úspěšně zapsat data na více než $N / 2$ uzlů. Pak už můžeme prohlásit operaci zápisu za úspěšnou, neboť nadpoloviční většina může získat jen jednu z konfliktních operací. Druhá bude odmítnuta jako neúspěšná.

Podobně můžeme uvažovat také při snaze o přečtení nejnovější verze dat. V tomto případě ale předpokládáme, že máme spolu s daty uložena také odpovídající časová razítka, která nás informují o stáří dat, tedy o tom, kdy byla naposledy zapsána nebo modifikována. Opět není třeba kontaktovat všechny uzly, ale stačí úspěšně přečíst data z R uzlů tak, aby platila nerovnost $R + W > N$, tzv. *kvórum čtení (read quorum)*, kde W je počet uzlů, které musí potvrdit úspěšný zápis dat. Tato nerovnost nám zajistí, že získáme aktuální data, neboť neumožňuje, aby současně proběhla operace čtení i operace zápisu.

Uvažujme například cluster s replikačním faktorem $N = 3$. Pro zápis dat nám stačí uspět na dvou uzlech, tedy $W = 2$ a je splněna nerovnost $W > N / 2$. Pro přečtení poslední verze dat pak potřebujeme kontaktovat dva uzly, tedy $R = 2$, abychom měli jistotu, že máme aktuální data (jeden nám nestačí, mohli bychom trefit zrovna ten, který úspěšný zápis nepotvrdil a obsahuje starší verzi dat). To odpovídá nerovnosti $R + W > N$. Kdybychom předpokládali $W = 3$, stačí nám naopak při čtení kontaktovat pouze jedinou repliku.

3.3.4 Replikace + sharding

Jak již bylo řečeno, typicky replikaci a sharding (rozdělení) dat kombinujeme. Nejprve tedy data rozdělíme dle námi zvolené strategie a pak zvolíme jeden z možných typů replikace. Kombinace *master-slave* replikace a rozdělení dat je pak realizována tak, že máme v clusteru více uzlů master, ale pro konkrétní data existuje master vždy jeden, zatímco současně pro jiná data může být uzlem typu slave. Kombinujeme-li peer-to-peer replikaci a rozdělení dat, potom je například při základním replikačním faktoru 3 každá část dat uložena na třech uzlech v clusteru, přičemž jednotlivé uzly obsahují různé části dat. Využití obou uvedených postupů podrobněji ukážeme u jednotlivých typů databází NoSQL ve druhé části knihy.

4.

Zpracování dat pomocí MapReduce

Jedním ze základních principů využívaných v oblasti paralelního distribuovaného zpracování Big Data je programovací model zvaný *MapReduce*. Tento princip byl představen firmou Google v článku [75].¹ Je zde popsán přístup, s jehož pomocí autoři efektivně rozdistribuovali, zpracovali a opět sloučili enormní množství dat. Tento princip poté začaly využívat různé systémy, od systému Hadoop, jež je pravděpodobně nejpopulárnější implementací a který představíme v sekci 4.3, až po nejrůznější NoSQL databáze, které tento princip využívají jako jeden z možných způsobů, jak zpracovávat uložená data.

Google definuje MapReduce jako „jednoduché a výkonné rozhraní, které umožňuje automatickou paralelizaci a distribuci výpočtů nad rozsáhlými daty, a k němu příslušející implementaci tohoto rozhraní, jež umožňuje dosáhnout vysoký výkon s využitím velkého clusteru běžně dostupných počítačů“ [75]. Taková implementace se obvykle nazývá MapReduce framework. Pravděpodobně nejznámější z nich

¹ První verze tohoto článku byla publikována o tři roky dříve na konferenci OSDI.

je systém Hadoop² organizace Apache,³ na který navazuje celá řada projektů vycházejících z principu MapReduce, jako je např. Mahout⁴ pro dolování dat (*data mining*), ZooKeeper⁵ pro řešení distribuované koordinace služeb nebo Hive⁶ pro datové sklady (*data warehousing*). Mezi další známé implementace tohoto principu patří např. Phoenix,⁷ Spark⁸ nebo DISCO.⁹ A můžeme jej také nalézt jako přímou součást NoSQL databází jako např. CouchDB,¹⁰ Riak¹¹ nebo MongoDB.¹²

Historie projektu Hadoop

Hadoop vznikl původně jako součást projektu Nutch,¹³ jehož smyslem bylo umožnit komukoliv provádět rozsáhlé indexování webu tak, jak je v té době byly schopny provádět pouze velké společnosti typu Google nebo YAHOO! [155]. Podobně jako v případě Lucene,¹⁴ prvního důležitého open source projektu Douga Cuttinga, který umožnil široké vývojářské veřejnosti indexovat a efektivně prohledávat textové dokumenty, byla originální vizí systému Hadoop demokratizace technologie – v tomto případě distribuovaného vykonávání úloh.

Hadoop v současné době tvoří silně heterogenní ekosystém obsahující jak samotný open source projekt vyvíjený pod záštitou Apache Software Foundation, tak komerční distribuce vyvíjené a udržované zejména společnostmi Cloudera,¹⁵ Hortonworks¹⁶ a MapR.¹⁷ Ty se liší svým přístupem k open source „zdrojovému“ projektu, tedy jeho poskytovanými rozšířeními. Např. MapR nabízí proprietární implementaci HDFS (Hadoop Distributed File System) – viz sekce 4.3.1, která je několikrát rychlejší než open source verze.

Společnost Amazon nabízí Hadoop jako plně spravovanou službu s grafickým rozhraním a sofistikovanou integrací s ostatními službami (jako např. Amazon Elastic Compute Cloud – Amazon EC2¹⁸ nebo Amazon Simple Storage Service – Amazon S3²⁰) v rámci produktu Elastic MapReduce²¹ a umožňuje tak širší vývojářské veřejnosti zkoumat a používat Hadoop bez vstupních nákladů na infrastrukturu a bez nutnosti instalace a konfigurace všech jeho komponent.

² <http://hadoop.apache.org>

³ <http://www.apache.org>

⁴ <http://mahout.apache.org>

⁵ <http://zookeeper.apache.org>

⁶ <http://hive.apache.org>

⁷ <http://mapreduce.stanford.edu>

⁸ <http://spark.apache.org>

⁹ <http://discoproject.org>

¹⁰ <http://couchdb.apache.org>

¹¹ <http://basbo.com/riak/>

¹² <http://www.mongodb.org>

¹³ <http://nutch.apache.org>

¹⁴ <http://lucene.apache.org>

¹⁵ <http://www.cloudera.com>

¹⁶ <http://hortonworks.com>

¹⁷ <https://www.mapr.com>

¹⁸ <http://aws.amazon.com/ec2/>

²⁰ <http://aws.amazon.com/s3/>

²¹ <http://aws.amazon.com/elasticmapreduce/>

MapReduce je využíván v mnoha známých aplikacích pracujících s Big Data. Např. podle článku [46] již v roce 2010 využíval Facebook právě systém Hadoop. Na druhou stranu ale později ukážeme, že některé systémy a společnosti od principu MapReduce zase upouští.

Využití systému Hadoop ve společnosti Seznam.cz

Jedním z nejvýznamnějších uživatelů systému Hadoop v českém prostředí je společnost Seznam.cz,²² která jej využívá k mnoha úlohám přibližně od roku 2009.

Jedno z hlavních využití je v kontextu fulltextového vyhledávání: robot, který prochází český a zahraniční internet, ukládá webové stránky do databáze HBase,²³ přičemž klíčem je URL a hodnotou jsou její různé reprezentace, např. extrahovaný hlavní obsah stránky, metadata a jejich historie. Hadoop je využíván pro počítání *zpětných odkazů*, tedy odkazů, které vedou na danou stránku, a rovněž k počítání *signálů*, tedy atributů stránky, které ovlivňují její pozici při řazení výsledků hledání nebo chování indexujícího robota. Pro mnoho výpočtů přitom společnost Seznam.cz ustupuje od „čistého“ systému Hadoop: např. pro počítání *page ranku*, tedy důležitosti stránky z hlediska počtu odkazů, které na ni vedou, využívá nástroj Apache Giraffe,²⁴ pro jiné operace Apache Spark.²⁵

Dalším využitím je analýza provozních dat služby Email.cz: Informace o provozu v síti jsou ukládány do HDFS a pomocí MapReduce úlohy jsou sestavovány analýzy a statistiky, např. objem e-mailů, které uživatelé smažou bez přečtení, počet e-mailů odeslaných z určité domény, detekce uživatelských účtů využívaných roboty apod. Pro implementaci MapReduce úlohy je používán jazyk Python v nadstavbě pydoop,²⁶ který je výkonnostně pomalejší než Java, ale úlohy jsou v něm mnohonásobně rychleji napsány. Hadoop přitom tvoří jen dočasné úložiště, agregovaná data jsou uložena do systému Elasticsearch. Jejich grafickou a interaktivní reprezentaci poskytuje upravená verze nástroje Kibana.²⁷

Seznam.cz využívá Hadoop v distribuci Cloudera (CDH²⁸) a provozuje několik nezávislých clusterů, které v současné době obsahují několik stovek uzlů.

²² <https://www.seznam.cz>

²³ <http://hbase.apache.org>

²⁴ <http://giraph.apache.org>

²⁵ <https://spark.apache.org>

²⁶ <https://pypi.python.org/pypi/pydoop>

²⁷ <http://www.elastic.co/products/kibana>

²⁸ <http://www.cloudera.com/content/cloudera/en/products-and-services/cdb.html>

4.1 Funkce Map a Reduce

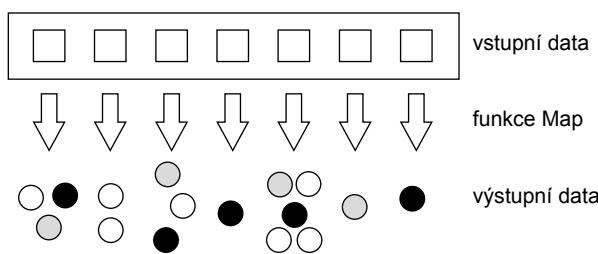
Základní myšlenka programovacího modelu MapReduce vychází z programovacího principu „rozděl a panuj“. Je založena na dvou základních funkcích *Map* a *Reduce*. Funkce Map slouží ke zpracování každého objektu z množiny vstupních dat a jejím výstupem je jedna nebo více dvojic (**klíč**, **hodnota**) pro každý zpracovaný objekt. Funkce Reduce slouží ke sloučení výsledků (hodnot) pro jednotlivé klíče z předchozí fáze Map do celkového výsledku. Mezi aplikací funkcí Map a Reduce je tedy třeba zajistit vytvoření množiny všech hodnot pro každý jednotlivý klíč z mezikvě sledku.

Přestože je možné princip MapReduce použít pro lokální zpracování dat, jeho hlavní síla je především v distribuovaném zpracování velkého množství dat. Právě z tohoto důvodu mají funkce Map a Reduce pevně dané rozhraní. Ukážeme je na oblíbeném učebnicovém příkladu zjištění počtu výskytů jednotlivých slov v dané množině dokumentů. Obecně má funkce Map dvojici vstupních parametrů tvořenou klíčem z domény K_1 a hodnotou z domény H_1 , což může být v našem případě např. dvojice (**název**, **obsah**), tedy název a obsah zpracovávaného souboru. Výstupem funkce Map je množina dvojic, přičemž každá dvojice je tentokrát tvořená klíčem z domény K_2 a příslušející hodnotou z domény H_2 . V našem příkladu by to tedy byla dvojice (**slovo**, **počet**), kde hodnota **počet** reprezentuje počet výskytů slova v daném souboru, nebo případně pouze dvojice (**slovo**, **1**) pro každý výskyt slova. Naše funkce Map by v pseudokódu²⁹ vypadala takto:

Příklad 4.1: Funkce Map

```
Map(String key, String value):
    // key: název dokumentu
    // value: obsah dokumentu
    foreach word in value:
        return (word, 1);
```

Klíč (proměnná **key**) tedy odpovídá názvu zpracovávaného dokumentu, hodnota (proměnná **value**) obsahuje celý text dokumentu. Funkce Map prochází jednotlivá slova dokumentu a pro každé z nich vrátí dvojici (**slovo**, **1**). Schematicky bychom funkci Map a její aplikaci mohli zobrazit tak, jak je vidět na obrázku 4.1.

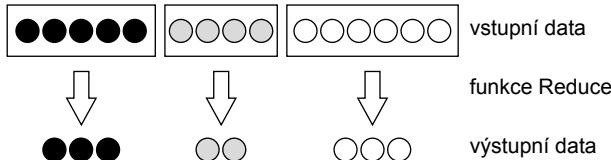


Obrázek 4.1: Funkce Map

Funkce Reduce má také dvojici vstupních parametrů, tentokrát tvořenou klíčem z domény K_2 a příslušející množinou *všech* hodnot z domény H_2 vytvořených v rámci fáze Map. Jedná se o klíče a hodnoty ze stejné domény, jako jsou výstupní

²⁹ Později ukážeme její implementaci v jazyce Java pro konkrétní MapReduce framework.

klíče a hodnoty funkce Map, přesněji řečeno přímo ty, které byly ve fázi Map vygenerovány. V našem případě by to tedy bylo slovo a seznam počtu jeho výskytů ve vstupních souborech, resp. pouze seznam jedniček, jedna za každý výskyt slova. Výstupem funkce Reduce je dvojice parametrů klíč z domény K_2 a redukovaná množina hodnot z domény H_2 . V našem příkladu by tedy tato funkce pro dané slovo všechny počty výskytů sečetla. Schéma funkcí je vidět na obrázku 4.2.



Obrázek 4.2: Funkce Reduce

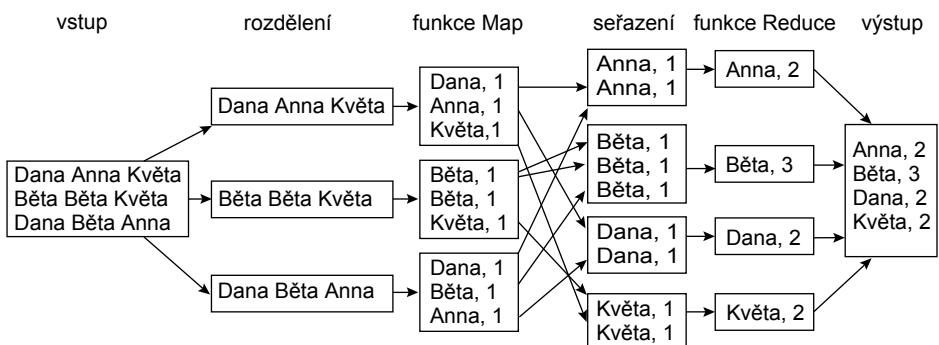
Konkrétní funkce Reduce by opět v pseudokódu vypadala takto:

Příklad 4.2: Funkce Reduce

```
Reduce(String key, Iterator values):
    // key: slovo
    // values: seznam počtu výskytů
    int result = 0;
    foreach v in values:
        result += v;
    return (key, result);
```

Tentokrát klíč (proměnná `key`) obsahuje jedno z nalezených slov dokumentu. Hodnota (proměnná `values`) obsahuje seznam všech hodnot, se kterými se ve dvojici daný klíč vyskytl na výstupu funkce Map. Konkrétně v našem případě je to tolik hodnot 1, kolikrát se slovo v některém z dokumentů vyskytlo. Funkce Reduce hodnoty seče a výsledek spolu s daným slovem vypíše na výstup.

Jak již bylo řečeno, aby tyto dvě funkce skutečně zajistily předpokládaný výsledek, musí MapReduce framework sloučit příslušným způsobem výstupní hodnoty mezi fází Map a Reduce.



Obrázek 4.3: Počet výskytů slov pomocí MapReduce

Podívejme se nejprve, jak by zpracování z našeho příkladu vypadalo v distribuovaném prostředí. Jeho grafické znázornění je vidět na obrázku 4.3. Nejprve jsou vstupní data rozdělena na menší celky, v našem případě tři. Na každý z nich je

aplikována funkce Map, která emittuje dvojice (*slovo*, 1). Tyto dvojice jsou následně rozděleny do čtyř skupin odpovídajících čtyřem různým slovům, která byla v datech nalezena, a tedy čtyřem různým klíčům. Následuje aplikace funkce Reduce, která pro každé slovo sloučí jeho počty výskytů a přispěje jedním záznamem do výsledku. Díky rozdělení vstupu na menší části a jejich nezávislému zpracování je možné jak funkci Map, tak Reduce spouštět paralelně na různých uzlech v clusteru.

4.1.1 Další příklady

Dalšími příklady úloh a jejich realizací pomocí principu MapReduce mohou být například:

- Výpočet frekvence přístupů na webové stránky: Na vstupu předpokládáme logy přístupů na webové stránky. Funkce Map prochází logy a podobně jako v předchozím příkladu pro každé nalezené URL vygeneruje dvojici (*URL*, 1). Funkce Reduce pro každé URL seče počet výskytů a výsledek vypíše.
- Graf zpětných odkazů: Na vstupu předpokládáme množinu webových stránek s odkazy. Funkce Map prochází webové stránky a pro každý odkaz na cílovou stránku v aktuálně zpracovávané stránce vygeneruje dvojici (*cíl*, *zdroj*). Pro každý cíl funkce Reduce vytvoří seznam všech zdrojových odkazů.
- Vytvoření invertovaného seznamu, tedy indexu výskytů slov: Funkce Map prochází vstupní dokumenty a pro každé slovo vygeneruje dvojici (*slovo*, *identifikátor dokumentu*). Funkce Reduce pak pro každé slovo vytvoří celkový seznam dokumentů, v nichž se vyskytuje.

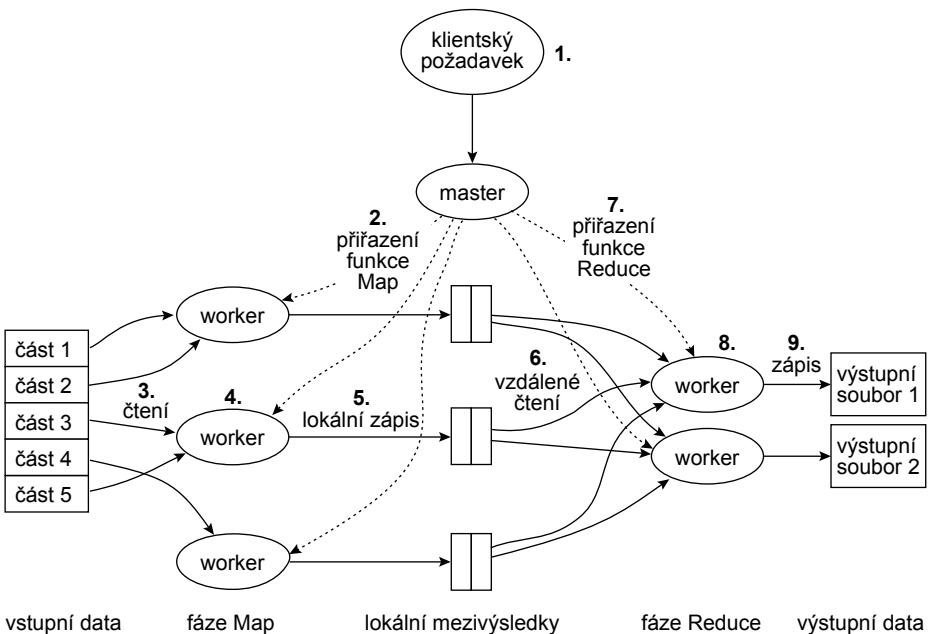
4.2 MapReduce framework

Ukažme nejprve jednotlivé fáze práce MapReduce frameworku na obrázku 4.4 na následující straně. Předpokládejme, že máme k dispozici cluster uzlů. Jeden z nich, říkem mu *master*, bude řídit celý výpočet. Master nejprve přijme MapReduce požadavek od klienta. Požadavek je tvořen kódem funkce Map, kódem funkce Reduce, vstupními daty a případně dalšími parametry.

V první fázi (nazývané také *Map*, podle příslušné funkce) master rozdělí zpracovávaná data na M částí³⁰ a připraví seznam M čekajících (*idle*) Map úloh. Pak postupně předává jednotlivé čekající úlohy spolu s daty ostatním uzlům v clusteru – ty bývají označovány jako *workers* (dělníci).³¹ Každý uzel následně aplikuje danou úlohu Map na přidělená data (po tuto dobu je příslušná úloha označena jako *běžící*), výsledek uloží na lokální disk a informaci o tomto umístění předá uzlu master. Konkrétní umístění dat na lokálním disku je dáno funkcí *Partition*, která určuje rozdělení dat do R částí disku odpovídajících budoucím R úlohám Reduce. Úspěšně dokončená úloha je označena jako *zpracovaná*.

³⁰Tento parametr, stejně jako další, které ukážeme, je možné nastavit dle nároků konkrétní aplikace.

³¹V závislosti na konkrétní implementaci sám sobě master také případně přidělí část úloh, tj. dokáže pracovat jako master i worker současně.



Obrázek 4.4: Fáze práce MapReduce frameworku

V další fázi (nazývané *Reduce*) master vytvoří R čekajících úloh *Reduce*. Následně jejich zpracování spolu s příslušnými daty (z fáze *Map* ví, kam která úloha *Map* uložila svůj výsledek) přiděluje jednotlivým uzlům. Úlohy *Reduce* nejprve pomocí vzdáleného volání načtou data z míst, která jim byla uzlem master přidělena. Poté každá úloha tato data seřadí podle klíče (tedy vytvoří množiny hodnot odpovídajících jednotlivým klíčům) a postupnou aplikací funkce *Reduce* na jednotlivé vstupní dvojice vytvoří příslušnou část výsledku MapReduce požadavku.

Jak je tedy vidět, máme-li implementaci MapReduce frameworku, programátor má za úkol jediné – implementovat funkce *Map* a *Reduce* a zadat parametry M , R a případně další parametry. Všechno ostatní, tedy především distribuci dat, plánování jednotlivých výpočtů, řízení komunikace mezi jednotlivými servery, řešení výpadků atd., zajišťuje MapReduce framework. V sekci 4.2.1.1 vysvětlíme některé principy, které k tomu využívá.

Díky rozdělení úlohy na M , resp. R částí můžeme problém zpracovávat paralelně a tedy mnohem efektivněji. Na druhou stranu existují úlohy, kdy např. parametr R nastavíme na 1, tedy výstup bude uložen pouze do jediného souboru. Framework také typicky řeší více MapReduce úloh najednou, resp. parametry M a R nemusí přesně odpovídat počtu uzlů v clusteru (bývá jich mnohem větší). Jeden uzel tedy obvykle zpracovává více (různých) úloh *Map*, resp. *Reduce*.

4.2.1 Další vlastnosti

Podívejme se nyní podrobněji na hlavní části MapReduce frameworku. Jsou to především tyto:

- Modul pro načítání vstupu (*input reader*): Úkolem tohoto modulu je načítání zpracovávaných dat ze zadaného úložiště, kterým může být např. distribuovaný souborový systém nebo NoSQL databáze, jejich rozdělení na M částí a předání jednotlivým funkcím Map.
- Funkce Map: Uživatelem daná funkce, která mapuje dvojice (klíč, hodnota) na množinu jiných dvojic (klíč, hodnota), formálně vyjádřeno jako Map: $K_1 \times H_1 \rightarrow \text{set}(K_2 \times H_2)$.
- Funkce pro rozdělování dat (funkce *Partition*): Tato funkce má na starost rozdělení výstupních hodnot funkce Map pro jednotlivé funkce Reduce. Na vstupu postupně dostává klíče z domény K_2 , popř. jim odpovídající hodnoty z domény H_2 , a parametr R , který nese informaci o tom, kolik funkcí Reduce bude spuštěno. Jejím výstupem je index funkce Reduce, která bude daný klíč zpracovávat.
- Funkce pro porovnávání (funkce *Compare*): Tato funkce zajistí požadované seřazení vstupu přiřazeného funkčním Reduce, které bude odpovídat specifickovanému způsobu porovnávání.
- Funkce Reduce: Uživatelem daná funkce, která zredukuje dvojice (klíč, množina hodnot) na dvojice (klíč, množina hodnot), přičemž výstupní množina hodnot může být menší, formálně vyjádřeno jako Reduce: $K_2 \times \text{set}(H_2) \rightarrow K_2 \times \text{set}(H_2)$
- Modul pro zápis výstupu (*output writer*): Tento modul zajišťuje zápis výstupu funkce Reduce do zadанého úložiště.

Další uživatelsky definovanou funkcí může být funkce *Combine*. Není-li uživatelem upravena, potom jen předá svůj vstup beze změny na výstup. Funkci můžeme využít například tehdy, když chceme redukovat množství dat předávaných mezi fází Map a Reduce. Funkce Combine funguje podobně jako funkce Reduce, ale pouze nad lokálními výstupy jednoho uzlu worker ve fázi Map. Uvažme opět náš první příklad s počtem výskytů slov. Funkce Map generuje na výstupu dvojice (slovo, 1). Je zjevné, že pokud se slova v dokumentech opakují, je tento výstup významně komprimovatelný. Pomocí funkce Combine můžeme tedy tyto hodnoty redukovat na dvojice (slovo, počet), kde proměnná počet obsahuje počet výskytů slova pouze v dané lokální podmnožině dokumentů.

4.2.1.1 Tolerance k výpadku

Obecně u úlohy typu MapReduce předpokládáme, že velké množství uzel zpracovává velké množství dat. Je tedy nezbytně nutné, aby měl v sobě MapReduce framework zabudovaný robustní mechanismus pro řešení výpadků a chyb. Selhat samozřejmě může libovolná část clusteru.

Selhání uzel typu worker řeší master. Ten pravidelně posílá dotaz na aktuální stav všem uzelům worker, kterým byla přidělena nějaká úloha Map nebo Reduce.

Jestliže se příslušný worker neozve do zadáno časového limitu, označí jej master za *nedostupný* a všechny jeho nedokončené úlohy označí opět jako čekající. Průběžně je potom rovnoměrně přiděluje dostupným uzlům worker.

Selhání uzlu master se obvykle řeší jedním ze dvou přístupů. V prvním z nich, pesimistickém, master v pravidelných intervalech zapisuje do logů aktuální stav celého clusteru (tj. informace o dostupných uzlech, čekajících/běžících/dokončených úlohách Map a Reduce atd.). Když master selže, je možné nastartovat novou verzi, která pokračuje v řízení systému od posledního bodu stavového zápisu. Ve druhém, optimistickém přístupu předpokládáme, že master je spolehlivý a jeho selhání je nepravděpodobné. Pro zvýšení efektivity žádný záznam aktuálního stavu neprobíhá. Když master přece jen selže, je třeba znova spustit veškeré nedokončené MapReduce úlohy.

Efektivitu systému mohou ovlivňovat nejen výpadky jednotlivých uzlů, ale také jejich neefektivita, která může být způsobena např. výrazně méně výkonným nebo poškozeným HW některých uzlů. Takový uzel bývá označován jako *straggler* (opozdilec) a čekání na něj může zbytečně zpomalit celou MapReduce úlohu. V okamžiku, kdy už je většina Map, resp. Reduce úloh dokončena, proto MapReduce framework spustí ještě *záložní* (backup) výpočty běžících *primárních* úloh na jiných uzlech. Výpočet, který úspěšně doběhne jako první (bez ohledu na to, jestli se jedná o primární nebo záložní), zajistí, že je úloha označená jako zpracovaná a ostatní výpočty stejně úlohy jsou ukončeny.

4.2.1.2 Parametrizace

Jak jsme již ukázali, MapReduce framework má množství parametrů, kterými může uživatel kontrolovat jeho chování. Hlavními jsou parametr M (předpokládaný počet paralelně běžících funkcí Map) a R (předpokládaný počet paralelně běžících funkcí Reduce a tedy i výstupních souborů). Jak ale tyto parametry nastavit? Uvědomme si nejprve, že master musí spustit $O(M + R)$ výpočtů, tedy M -krát Map a R -krát Reduce, popř. znova spustit úlohy z nedostupných uzlů a/nebo záložní výpočty. V paměti udržuje $O(M \times R)$ informací o stavu. Konkrétně pro každou úlohu Map, resp. Reduce udržuje její stav (čekající, běžící, zpracovaná). Pro každou nečekající úlohu udržuje identifikaci uzlu, ke kterému byla přiřazena. A pro každou dokončenou úlohu Map udržuje až R odkazů na umístění a velikost příslušné části výstupních dat, která jsou vstupem odpovídající úlohy Reduce.

Doporučení pro konkrétní hodnoty parametrů jsou dle článku [75] tyto: Parametr M je vhodné nastavit tak, aby každá úloha pracovala s 16–64 MB dat, což je předpokládané rozumné množství vzhledem k očekávanému výpočetnímu výkonu jednotlivých uzlů a lokální optimalitě zpracování. Nemáme-li specifické úmysly například s počtem výstupních souborů, je pro parametr R doporučován malý násobek počtu fyzických uzlů.

Pro lepší představu ještě z článku [75] uvedme dva reálné příklady nastavení a doby výpočtu na clusteru 1800 uzlů:³² Nejprve předpokládejme, že máme 1 TB dat, která prohledáváme příkazem grep s regulárním výrazem o třech znacích.

³² Každý se dvěma 2GHz procesory Intel Xeon, aktivovanou funkcí Hyper-Threading, 4 GB paměti, dvěma 160GB IDE disky a gigabitovým Ethernetovým připojením.

Hledaná hodnota se v datech vyskytuje 92 337krát. Při nastavení $M = 15\ 000$ a $R = 1$ byl dle článku vstup rozdělen na části o velikosti zhruba 64 MB. Celý výpočet trval 150 vteřin, tedy 2,5 minuty, přičemž zhruba minutu spotřebovaly přípravné kroky.

Druhým příkladem je seřazení zhruba 1 TB dat na stejném clusteru. Konkrétně funkce Map z každého řádku extrahuje klíč o velikosti 10 B a ten spolu s daty vrací na výstup. Dále se využije přirozeného mechanismu frameworku MapReduce, který data z fáze Map seřadí podle jejich klíčů. Funkce Reduce pouze vrací daná vstupní data. Při nastavení $M = 15\ 000$ a $R = 4\ 000$ celý výpočet trval 891 vteřin.

4.3 Hadoop

V současné době je pravděpodobně nejznámější implementací programovacího modelu MapReduce open source framework od firmy Apache s názvem Hadoop.³³ Jeho autorem je Doug Cutting, tvůrce knihovny Lucene pro fulltextové vyhledávání. Jak už bylo řečeno, Hadoop sám vznikl v rámci projektu Nutch, open source webového vyhledávače založeného právě na Lucene.

Jeho základní myšlenky vycházejí z MapReduce frameworku firmy Google, který ale open source není. Hadoop se skládá z několika modulů, těmi základními jsou:

- Hadoop Common: Sada společných základních balíčků a funkcí, které jsou využívány v ostatních modulech.
- Hadoop Distributed File System (HDFS): Distribuovaný souborový systém, který umožňuje efektivně ukládat a načítat rozsáhlá data.
- Hadoop YARN: Modul pro plánování úloh a správu zdrojů celého clusteru.
- Hadoop MapReduce: Implementace modelu MapReduce využívající předchozí tři moduly.

V následujících sekcích blíže představíme nejen podporu MapReduce, ale také HDFS. Ten totiž slouží jako úložiště, ke kterému přistupují běžící úlohy Map a Reduce.

4.3.1 HDFS

HDFS je open source distribuovaný platformově nezávislý škálovatelný souborový systém. Jeho hlavním rysem je odolnost vůči výpadkům, založená na předpokladu, že v clusteru o řádově tisících uzlech jsou selhání pravidlem, nikoli výjimkou. Systém tedy dokáže výpadky nejen detektovat, ale také se z nich efektivně zotavit.

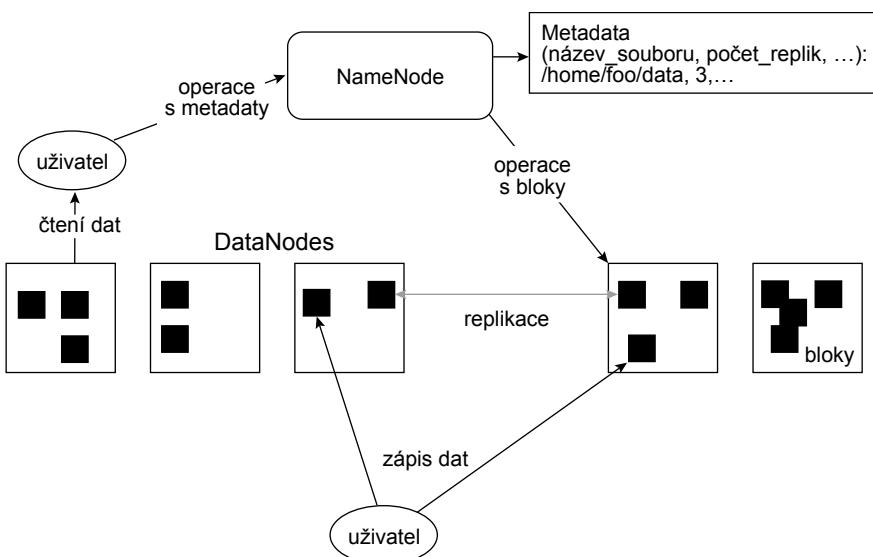
HDFS je optimalizovaný především pro dávkové zpracování, namísto interaktivního uživatelského přístupu. Dalším předpokladem je vlastnost write-once/read-many,

³³ Slovo Hadoop není zkratka, jak bývá u SW produktů zvykem, ale jedná se o smyšlené jméno pro plyšového žlutého slona autorových dětí. Ten je dnes také součástí loga produktu.

tedy soubor, který je jednou vytvořen, naplněn a uložen, už není dále modifikován, nebo alespoň ne příliš často. Typická aplikace nad HDFS tedy může být založena právě na principu MapReduce.

HDFS zpřístupňuje uživatelům svůj *jmenný prostor* (namespace), který je tvořený hierarchií adresářů a souborů, podobně jako u klasických lokálních souborových systémů. Stejně tak nabízí funkce pro jejich vytvoření, smazání, přejmenování, přesunutí apod. Z tohoto hlediska se práce s distribuovaným souborovým systémem nijak výrazně nelíší od lokálních souborových systémů. Zásadní rozdíl je ale v tom, jak je tato funkcionality implementována.

Architektura HDFS, zobrazená na obrázku 4.5, je založena na principu master/slave. Uzel typu master se v HDFS nazývá *NameNode* a má na starosti správu jmenného prostoru a přístupu k souborům a adresářům, tedy jejich otevření, zavírání, přejmenování apod. Tyto operace spravuje jediný uzel a nemůže tedy dojít k nekonzistencím. Uzly typu slave se v HDFS nazývají *DataNodes* (datové uzly). Jak už jejich název napovídá, právě na nich jsou v lokálních souborových systémech uložena samotná data. Typicky jeden datový uzel odpovídá jednomu fyzickému uzlu v clusteru. Konkrétní rozmístění dat na datových uzlech řídí NameNode.



Obrázek 4.5: Architektura HDFS

Jelikož předpokládáme ukládání velkých souborů, jsou tyto v HDFS rozděleny do tzv. *bloků*, typicky velikosti 64 nebo 128 MB, a distribuovány na více uzlů. Rozdělení dat a jejich distribuci má opět na starosti NameNode. Pro zvýšení odolnosti vůči výpadkům jsou data také replikována, přičemž počet replik můžeme určit pro každý soubor zvlášť. Typicky je replikační faktor nastaven na hodnotu 3 a pro jejich optimální distribuci se využívají různé techniky v závislosti na vlastnostech sítě a požadavcích dané aplikace (více viz [15]). Všechny informace o souborech, jejich aktuální distribuci do bloků, replikaci atd. (souhrnně *metadata*) také spravuje NameNode.

Datové uzly obsluhují požadavky klientů na čtení a zápis dat z/do bloků na nich uložených a zajišťují vytváření a mazání bloků na základě pokynů uzlu NameNode. Obvykle ukládají každý blok do samostatného souboru. Soubory jsou uloženy v adresárové struktuře optimální pro daný lokální typ souborového systému.

Pro zajištění rychlého zotavení z výpadků využívá HDFS několik technik:

NameNode pravidelně ukládá na lokální disk *kontrolní bod* (*checkpoint*), tj. obraz celého systému, který tvoří jmenný prostor, veškerá metadata a případně další nastavení systému. Dále udržuje (tj. ukládá a při uložení nového kontrolního bodu promazává) log transakcí, které se v souborovém systému proběhly od posledního kontrolního bodu (jako např. vytvoření souboru, změna replikačního faktoru souboru apod.). V případě výpadku se tedy do posledního známého stavu dostane snadno – nejprve načte poslední uložený obraz celého systému a následně aplikuje operace z logu transakcí.

Datové uzly pravidelně informují NameNode o svém aktuálním stavu, tj. posílají tzv. *heartbeat*, tedy zprávu o tom, že nedošlo k výpadku spojení. Pokud NameNode zprávu v zadaném časovém intervalu nedostane, označí bloky dat na daném uzlu za nedostupné a takovému uzlu neposílá požadavky na ukládání/načítání dat. Kdyby tím klesl replikační faktor některého ze souborů (ale i v případě poškození replik nebo HW, zvýšení požadovaného replikačního faktoru apod.), dojde k dodatečné replikaci.

4.3.2 Hadoop MapReduce

Implementace MapReduce v nástroji Hadoop primárně pracuje s daty uloženými v HDFS. Umožňuje veškerou funkcionality, kterou od MapReduce frameworku očekáváme, tedy plánování a spouštění úloh, sledování průběhu jejich zpracování, řešení výpadků a agregaci dlouhých výstupů do požadovaného celkového výsledku. Řízení výpočtu zajišťuje uzel master (tzv. *JobTracker*), výpočet provádí uzly typu worker (tzv. *TaskTrackers*). Systém je implementován v jazyce Java, který se tedy předpokládá i pro funkce Map, Reduce, popř. Combine a další. Nicméně prostřednictvím nástroje Hadoop Streaming³⁴ je možné využívat i jiné programovací jazyky jako např. Python nebo Ruby.

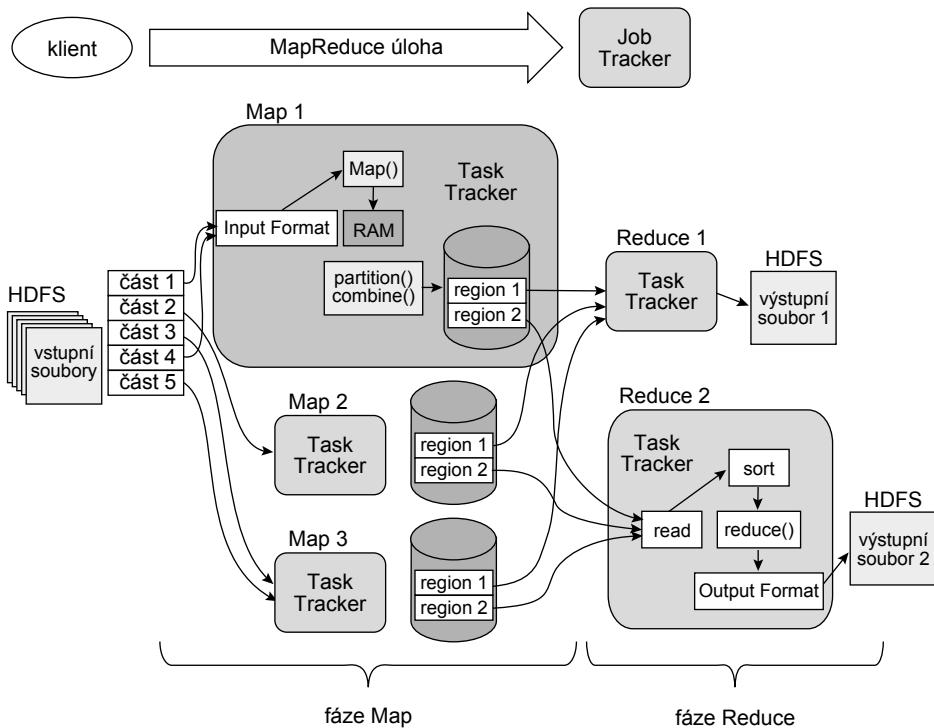
Jak je schematicky znázorněno na obrázku 4.6 na následující straně, MapReduce požadavek, tedy především funkce Map, Reduce a cesty ke zpracovávaným datům (uložené do parametrů třídy *JobConf*), je předán uzlu JobTracker (třídě *JobClient*). Ten nejprve od řídicího uzlu NameNode příslušného HDFS zjistí, které uzly typu TaskTracker leží „nejblíže“ zpracovávaným datům a které by tedy měly primárně jednotlivé úlohy zpracovávat. Následně pak řídí celý výpočet.

Každý TaskTracker (worker) přijímá jednotlivé MapReduce úlohy, tj. konkrétní funkce Map/Reduce/Combine a cesty ke vstupům, resp. výstupům. Udržuje tabulkou volných pozic pro zpracování úloh, přičemž pro každou úlohu spouští samostatnou JVM (Java Virtual Machine). Ve zprávě heartbeat pro JobTracker posílá mimo jiné počet volných pozic, aby mohl JobTracker distribuovat zátěž rovnoměrně. Vstupy

³⁴ <https://hadoop.apache.org/docs/r1.2.1/streaming.html>

a výstupy úloh zajišťují třídy `InputFormat` a `OutputFormat`, které definují způsob načítání uložených dat. Můžeme buď využít některou z nabízených implementací, nebo vytvořit vlastní.

Funkce Map ukládá výsledky do lokální paměti. Odtud jsou periodicky ukládány na lokální disk příslušného uzlu rozdělený na R regionů. V tomto kroku tedy dochází k určení úlohy Reduce, která bude klíče zpracovávat. Seznam a umístění regionů jsou předány na uzel JobTracker, který informace sloučí a předá jednotlivým uzlům vykonávajícím funkce Reduce, jež data načtou pomocí vzdáleného volání procedur. Po načtení všech dat tato seřadí podle jejich klíče a zpracuje. Je-li dat příliš velké množství, je pro řazení použit některý z algoritmů vnějšího řazení.



Obrázek 4.6: Zpracování MapReduce požadavku v systému Hadoop

Hadoop podporuje i mnoho dalších vstupních parametrů, které umožňují modifikovat výpočet. Tyto parametry je možné předat třídě `JobConf` před spuštěním úlohy. Také lze např. implementovat vlastní funkci `Partition`, která určuje úlohu `Reduce`, jež bude zpracovávat výstupy pro daný klíč a hodnotu na výstupu úlohy `Map`. Implicitně Hadoop v tomto kroku používá hašovaná hodnota klíče. Stejně tak je možné doplnit funkci `Combine` pro lokální redukci výstupů funkce `Map`. Způsob, jakým bude probíhat třídění klíčů před fází `Reduce`, lze určit implementací funkce `Compare`. Pro sledování průběhu je dále možné definovat různé vlastní čítače a generovat zprávy o aktuálním stavu výpočtu. Pro všechny tyto parametry Hadoop nabízí velké množství tříd a rozhraní. Podrobnější informace je možné nalézt např. v [28] nebo [155].

Funkce Map z příkladu na zjištění počtu výskytů jednotlivých slov by přímo pro Hadoop odpovídala kódů v příkladu 4.3, funkce Reduce pak kódů v příkladu 4.4.

Příklad 4.3: Funkce Map v Java API Hadoop MapReduce

```
public class Map
    extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key,
                    Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
```

Příklad 4.4: Funkce Reduce v Java API Hadoop MapReduce

```
public class Reduce
    extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key,
                      Iterator<IntWritable> values,
                      OutputCollector<Text, IntWritable> output,
                      Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

4.3.3 Další nadstavby systému Hadoop

Kromě MapReduce frameworku nabízí systém Hadoop také velké množství dalších modulů a nástrojů. Tři z nich, konkrétně Pig pro analýzy a dotazování dat, Cascading pro práci ve vyšších programovacích jazycích a Hive pro podporu datových skladů, ještě krátce představíme v této sekci. Existuje ale i množství dalších souvisejících nástrojů pro zpracování dat s využitím systému Hadoop, jako např. Hadoop³⁵ pro iterativní zpracování dat, databázový systém HadoopDB,³⁶ množina nástrojů pro analytické zpracování dat SQL-on-Hadoop³⁷ a další.

³⁵ <https://code.google.com/p/hadoop/>

³⁶ <http://db.cs.yale.edu/hadoopdb/badoopdb.html>

³⁷ <https://www.mapr.com/why-hadoop/sql-hadoop/sql-hadoop-details>

4.3.3.1 Pig

Apache Pig³⁸ je ukázkou využití principu MapReduce pro analýzy Big Data (viz sekce 10.1). Sestává se z jazyka vyšší úrovně nazývaného Pig Latin,³⁹ který umožňuje definovat požadované analytické operace nad daty, a infrastruktury, která skripty v jazyce Pig Latin provádí.⁴⁰ Zásadní výhodou je, že tyto programy je možné paralelizovat a tedy aplikovat na velká data. Paralelizovatelnost je zajištěna překladačem jazyka Pig Latin, který vytvoří sadu MapReduce programů spustitelných přímo pomocí Hadoop MapReduce. Pig Latin má za cíl poskytovat jednoduché programovací rozhraní, možnosti optimalizace a rozšířitelnost.

Příkazy v jazyce Pig Latin jsou obvykle seřazeny takto:

- Příkaz LOAD načte data z (distribuovaného) souborového systému. Předpokládáme textová data, která jsou při načtení rozdělena podle zvoleného nebo implicitního oddělovače na množinu n-tic. Je tedy vlastně možné pracovat s relacemi, podobně jako v relačních databázích. Příslušné příkazy se podobají jazyku SQL.
- Série transformací zpracovávajících a analyzujících data, tedy některé z následujících příkazů:
 - Příkaz FILTER pracuje s celými n-ticemi a umožňuje z nich vybírat podmnožiny dle zadané podmínky. Příkaz FOREACH pracuje s jednotlivými prvky n-tic a na základě jejich hodnot generuje požadovaný výstup.
 - Příkaz GROUP umožňuje seskupit data jedné relace (resp. příkaz COGROUP více relací) do skupin dle zadané podmínky.
 - Příkazy INNER JOIN a OUTER JOIN umožňují provádět vnitřní a vnější spojení dat z více relací. Příkaz UNION sjednocuje data z více relací.
 - Příkaz SPLIT umožňuje data rozdělit do více relací.
- Příkaz DUMP slouží pro zobrazení výsledků, příkaz STORE pro jejich uložení.

V příkladu 4.5 jsou nejprve data ze souboru `student.txt` pomocí příkazu LOAD načtena do relace A se třemi sloupcí (name, age a avg). Jak zjistíme díky příkazu DESCRIBE, není-li uveden datový typ, použije se implicitní `bytearray`. Následně je z této tabulky do proměnné B vypsán výsledek dotazu, který provede operaci GROUP přes věk jednotlivých studentů, tj. vytvoří skupiny studentů se stejným věkem.

Příklad 4.5: Příklad použití jazyka Pig Latin

```
-- kontrolní výpis obsahu souboru
cat student.txt;
John    18  4.0
Mary    19  3.8
Bill    20  3.9
Joe     18  3.8
```

³⁸ <https://pig.apache.org>

³⁹ <http://pig.apache.org/docs/r0.14.0/basic.html>

⁴⁰ Poznamenejme, že Pig Latin je v anglicky mluvících zemích také dětská slovní hra, která mění slova s použitím speciálních přípon a změn pořadí písmen.

```
-- načtení souboru do relace A
A = LOAD 'student.txt' AS (name:chararray, age:int, avg);

-- kontrola dat v relaci A
DUMP A;
(John,18,4.0)
(Mary,19,3.8)
(Bill,20,3.9)
(Joe,18,3.8)

-- struktura relace
DESCRIBE A;
A: {name: chararray, age: int, gpa: bytearray}

-- dotaz a uložení jeho výsledku do relace B
B = GROUP A BY age;

-- kontrola dat v relaci B
DUMP B;
(18,{(John,18,4.0),(Joe,18,3.8)})
(19,{(Mary,19,3.8)})
(20,{(Bill,20,3.9)})
```

Příkazy je možné v systému Pig spouštět ve více režimech:

- Lokální režim předpokládá jediný server, na němž jsou uložena veškerá data, s nimiž chceme pracovat.
- Režim MapReduce předpokládá přístup k frameworku Hadoop MapReduce a HDFS. Data jsou uložena i zpracovávána distribuovaně. Tento režim je v systému Pig výchozí.

Oba režimy umožňují interaktivní nebo dávkové zpracování.

Kromě uvedených vlastností, které lze považovat za výhody systému, je jeho hlavní nevýhodou především efektivita zpracování. Ta úzce souvisí s obecně nízkou efektivitou MapReduce frameworku pro nevhodné úlohy a vysokými nároky na inicializaci frameworku.

4.3.3.2 Cascading

Další zajímavou nadstavbou nad Hadoop MapReduce je Apache Cascading.⁴¹ Má podobný cíl jako nástroj Pig – umožnit pohodlnou a rychlou implementaci zpracování lokálních i distribuovaných dat pomocí Hadoop MapReduce. Cascading ovšem pro tyto účely využívá vyšších programovacích jazyků jako např. Java. Lokální mód je určen především pro testovací účely, zatímco distribuovaný mód je určený k efektivnímu zpracování Big Data.

Cílem nadstavby Cascading je zjednodušit práci s Hadoop MapReduce prostřednictvím nové vrstvy. Práce s daty pomocí Apache Cascading je založena na myš-

⁴¹ <http://www.cascading.org/>

lence vytváření datových toků (streams) složených z rour (pipes) a datových filtrů (tedy operací s daty). Podobně jako Pig i Cascading pracuje s daty ve formě n-tic, které „tečou“ rourami a jsou modifikovány prostřednictvím filtrů. Na vstupu tedy typicky předpokládáme textové soubory, z nichž jsou n-tice načítány.

Roury je možné spojovat (obdoba klasické operace vnitřního/přirozeného spojení z relačních databází), slévat (obdoba operace sjednocení), seskupovat (obdoba operace GROUP BY z SQL), ale i rozdělovat (split) do několika rour. Na data v rourách lze aplikovat různé operace jako filtrace, transformace, aritmetické operace nebo uspořádání. Aby bylo možné definované propojení rour spustit, je třeba počáteční roury připojit na „kohouty“ (*taps*), tedy datové zdroje n-tic, a koncové na „odtoky“ (*sinks*), tedy úložiště výstupních dat (typicky výstupní soubory).

V příkladu 4.6 vidíme, jak jsou roury vytvářeny a vzájemně spojovány. Výsledek příkladu, tedy struktura definovaného toku, je graficky znázorněn na obrázku 4.7 na následující straně.

Příklad 4.6: Příklad použití Hadoop Cascading

```
// vytvoření první vstupní roury p1
Pipe p1 = new Pipe( "p1" );

// aplikace funkce a filtru na každou n-tici v rouře p1
p1 = new Each( p1, new SomeFunction() );
p1 = new Each( p1, new SomeFilter() );

// vytvoření druhé vstupní roury p2
Pipe p2 = new Pipe( "p2" );

// aplikace funkce na každou n-tici v rouře p2
p2 = new Each( p2, new SomeFunction() );

// přirozené spojení výstupů rour p1 a p2
Pipe join = new CoGroup( p1, p2 );

// aplikace agregací na každou n-tici ve výsledku spojení = rouře join
join = new Every( join, new SomeAggregator() );

// seskupení n-tic do skupin s ekvivalentnímu hodnotami sloupců
join = new GroupBy( join );

// další aplikace agregací
join = new Every( join, new SomeAggregator() );

// aplikace funkce a vygenerování výstupu
join = new Each( join, new SomeFunction() );

// specifikace vstupních dat (kohoutů)
Tap p1Source = new Hfs( new TextLine(), "p1.txt" );
Tap p2Source = new Hfs( new TextLine(), "p2.txt" );

// specifikace výstupu (odtoku)
Tap sink = new Hfs( new TextLine(), "output.txt" );

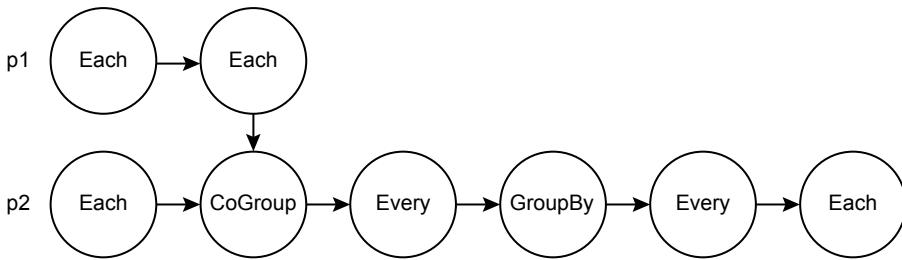
// definice a vytvoření celého toku a připojení vstupů a výstupů
```

```

FlowDef f1 = new FlowDef()
    .setName( "flow1" )
    .addSource( p2, p2Source )
    .addSource( p1, p1Source )
    .addTailSink( groupBy, sink );
Flow flow = new HadoopFlowConnector().connect( f1 );

```

Příklad demonstruje hlavní výhodu nadstavby Cascading, díky níž autor kódů pracuje s abstraktními objekty vrstvy Cascading tvořícími datový tok, nikoli s funkcemi Map a Reduce. Mapování na ně zajišťuje systém Cascading připojený k Hadoop MapReduce frameworku.



Obrázek 4.7: Roury v Hadoop Cascading

4.3.3.3 Hive

Poslední nadstavbou nad Hadoop MapReduce, kterou představíme, je Apache Hive.⁴² Na rozdíl od předchozích nástrojů neslouží ke zpříjemnění práce s Hadoop MapReduce frameworkem. Jedná se o infrastrukturu zajišťující funkcionality datového skladu. O tomto tématu budeme hovořit podrobněji v sekci 10.1. Nyní si vystačíme s konstatováním, že datový sklad [114] je speciální typ datového úložiště s podobným rozhraním, jako mají relační databáze. Datový sklad předpokládá uložení velkého množství dat a dotazy zaměřující se především na jejich analýzu. Na rozdíl od klasických relačních databázových systémů datový sklad dále předpokládá málo změn dat, ukládání historických dat (tj. namísto změn dat spíš ukládání nových verzí), integraci velkého množství dat z různých zdrojů a snahu o uložení, které vyhovuje analytickým dotazům i za cenu vyšších nároků na prostor, případně i redundance v datech.

Apache Hive umožňuje definovat a přiřadit k datům uloženým v distribuovaném úložišti strukturu podobnou relačnímu modelu a pak se nad nimi dotazovat prostřednictvím jazyka HiveQL,⁴³ vycházejícího z jazyka SQL (i když specifikaci standardu SQL přesně nedodržuje). Každý dotaz v HiveQL je komplítorem přeložen na sérii (přesněji řečeno orientovaný acyklický graf) MapReduce úloh, které jsou následně spuštěny nad Hadoop MapReduce frameworkem. Programátor může v těchto úlohách specifikovat také své vlastní MapReduce úlohy, není tedy omezen pouze na jazyk HiveQL.

Podobně jako SQL má i HiveQL dvě části – DDL (Data Definition Language), tedy jazyk pro definici dat, a DML (Data Manipulation Language), tedy jazyk pro mani-

⁴² <https://hive.apache.org/>

⁴³ <https://cwiki.apache.org/confluence/display/Hive/LanguageManual>

pulaci s daty. HiveQL DDL zahrnuje známé příkazy jako **CREATE TABLE** pro definici struktury tabulky, **ALTER TABLE** pro modifikace tabulky jako např. přidání/smazání/přejmenování sloupců a **DROP TABLE** pro smazání tabulky.

Hlavní operací HiveQL DML je **LOAD DATA**, která umožňuje načíst do tabulky data buď z lokálního souboru, nebo ze souboru uloženého v HDFS. Fyzicky tento příkaz způsobí přesunutí nebo překopírování souboru do místa souborového systému, kde jsou uloženy tabulky. Operace je tudíž poměrně rychlá. Navíc dochází k (alespoň základním) kontrolám struktury dat, u nichž předpokládáme, že řádky souboru odpovídají záznamům tabulky s definovanými oddělovači a datovými typy. Další možností je naplnění tabulky výsledkem dotazu v HiveQL.

Pro dotazování existuje v HiveQL příkaz **SELECT**, který má obdobnou strukturu jako stejnojmenný příkaz v jazyce SQL, ale není tak bohatý. Zahrnuje následující klauzule a operátory:

- **SELECT-FROM-WHERE** pro selekci a projekci vybraných dat,
- **GROUP BY** a **HAVING** pro práci se skupinami dat v tabulce,
- (v omezené míře) vnořené poddotazy,
- **JOIN** pro vnitřní (**INNER**) a vnější (**OUTER**) spojení tabulek,
- operace pro modifikaci výsledku, jako např. omezení počtu výsledků (**LIMIT**), zjištění počtu výsledků (**COUNT**) nebo seřazení výsledků dle vybraných hodnot (**ORDER BY**, **SORT BY**, **CLUSTER BY**)
- a mnohé další.

V následujícím příkladu uvedeme ukázku vytvoření a naplnění tabulky, položení dotazu a smazání tabulky. Jak je vidět, od dotazů v jazyce SQL se příliš neliší.

Příklad 4.7: Příklad HiveQL

```
-- vytvoření tabulky včetně určení oddělovače položek ve vstupním souboru
CREATE TABLE u_data (
    userid INT,
    movieid INT,
    rating INT,
    weekday INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';

-- načtení dat ze zadанého souboru s případným přepsáním již existujících
LOAD DATA LOCAL INPATH './u.data'
OVERWRITE INTO TABLE u_data;

-- zjištění počtu záznamů pro každý den
SELECT weekday, COUNT(*)
FROM u_data
GROUP BY weekday;

-- smazání tabulky
DROP TABLE u_data;
```

Nástupci systému Hadoop

Přestože je Hadoop stále nejpoužívanějším systémem pro zpracování netřívních objemů dat, lze v současnosti pozorovat ústup od na něm založených řešení. Hlavním důvodem je přitom samotná podstata systému Hadoop: dávkové zpracování úloh (*batch processing*) předpokládá, že data jsou někde „uložena“ a periodicky „zpracována“. Hadoop tedy stále vyhovuje pro ETL (*extract-transform-load*) řešení, ale jak jsme uvedli v kapitole 1, data často přibývají tak rychle, že je dávkové zpracování přírůstku nemožné. Proto se již před několika lety začaly objevovat projekty vycházející z odlišného konceptu: zpracování dat v reálném čase (*realtime stream processing*), jako je např. Apache Storm.⁴⁴ V současné době je populární projekt Apache Spark,⁴⁵ který obsahuje integraci s komponentami systému Hadoop, ale je jednoznačně orientován na zpracovávání „živých proudů“ dat (*live data streams*): poskytuje možnost spojování různých datových proudů (např. pro porovnání aktuálních a historických dat) či operace v pohyblivém okně (*sliding window*) v rámci datového toku [40]. Ve srovnání se systémem Hadoop je zpracování dat pomocí Spark mnohonásobně rychlejší mimo jiné proto, že Spark drží mezivýsledky v paměti. Dále nabízí rozhraní SQL i API pro práci s daty a obsahuje zabudované algoritmy strojového učení (*machine learning*), nebo funkce pro práci se vztahy mezi prvky. (Mezi další podobné projekty můžeme v současnosti počítat také např. Apache Ignite⁴⁶ nebo Apache Geode⁴⁷).

4.4 Kritika a ústup od MapReduce

Přestože je myšlenka MapReduce stále velmi populární a umožnuje efektivní reálnou realizaci odpovídající třídy úloh, setkáme se také s kritikou. Již v roce 2008 uvádí David DeWitt a Michael Stonebraker⁴⁸ v článku [79] následující hlavní nevýhody MapReduce frameworku z hlediska světa databází:

- MapReduce je krokem zpět v databázovém přístupu k datům, který stojí na třech klíčových vlastnostech: (1) schématech striktně popisujících strukturu dat, (2) oddělení schématu od vlastní aplikace a (3) pokročilých dotazovacích jazyků.
- MapReduce je suboptimálním řešením, které namísto efektivních ověřených indexů používá hrubou sílu.
- MapReduce není žádná novinka, jedná se o implementaci technik starých přes 20 let.

⁴⁴ <http://storm.apache.org>

⁴⁵ <http://spark.apache.org>

⁴⁶ <http://ignite.incubator.apache.org>

⁴⁷ <http://geode.incubator.apache.org>

⁴⁸ Michael Stonebraker je jedním z předních databázových expertů současnosti, který v roce 2014 získal Cenu A. M. Turinga, považovanou za jakousi Nobelovu cenu informatiky. Jak uvidíme v dalších kapitolách knihy, podílel se na vzniku velkého množství různých typů databázových systémů.

- MapReduce postrádá množství vlastností, které jsou samozřejmou součástí databázových systémů, jako jsou indexy, transakce, aktualizace, integritní omezení, referenční integrita, pohledy apod.
- MapReduce není kompatibilní s nástroji, které databázoví uživatelé nad databázemi implementují, jako např. nástroje pro data mining nebo business intelligence (přesněji řečeno, pro jejich implementaci je třeba značné implementační úsilí).

Obecně jsou za hlavní nevýhody principu MapReduce považovány především dvě skutečnosti:

1. Princip MapReduce je vhodný pouze pro specifickou množinu úloh. Ne vždy je vhodné nebo možné daný problém řešit právě tímto způsobem.
2. Přes veškeré optimalizace může být efektivita zpracování dané úlohy velmi nízká.

Další související událostí nedávné doby byla zpráva, že sám autor MapReduce, společnost Google, nyní přechází na zcela nový nástroj pro analýzu rozsáhlých dat s názvem *Google Cloud Dataflow* [77]. Potřebu vzniku nového přístupu odůvodňují požadavky na složitější způsoby zpracování dat. Ty by měla umožnit nová myšlenka založená na kombinaci cloudového a proudového zpracování dat umožňující data načít, transformovat a analyzovat bez nutnosti udržovat složitou infrastrukturu. Cloud computing je podrobněji popsán v sekci 10.4.

Část II.

NoSQL databáze

5.

Základní principy NoSQL databází

Ještě před nedávnem platilo, že pod slovem „databáze“ si člověk mohl představit „centralizovanou relační databázi“ a neudělal chybu, protože tyto dva pojmy v podstatě splývaly. V průběhu minulých desetiletí samozřejmě vznikaly nové typy databází, např. objektové nebo XML databáze. Některé z těchto typů se oproti očekávání neprosadily, jiné se stále rozvíjejí a mají své specifické aplikace. Žádný z těchto druhů databází se ale nestal tak široce používaným, aby se s odstupem času dalo mluvit o zásadnějším ohrožení hegemonie klasických relačních databází.

V posledních letech se ovšem v některých aplikačních oblastech poměrně zásadně posunuly požadavky na systémy pro správu dat, což vedlo k živelnému rozmachu zcela nových typů databázových technologií. Právě díky různorodým požadavkům na funkcionality a výkon databází nyní žijeme v době, kdy je nabídka databázových systémů tak pestrá, že pojem „databáze“ již neříká vše a často raději mluvíme o konkrétním databázovém systému. Dá se bez nadsázky říci, že každý z těchto nově vzniklých systémů je unikátní svou nabídkou funkcí, vnitřním fungováním, efektivitou a také záměrem, se kterým vznikl. Cílem druhé části knihy je vnést řád do chaosu tím, že identifikujeme několik různých typů těchto novodobých databázových systémů a popíšeme jejich společné a rozdílné rysy.

5.1 Společné principy NoSQL databází

Klasické relační databáze vychází z předpokladu, že dobře známe strukturu ukládaných dat, a tato data jsou často klienty a aplikacemi dotazována různými i předem neznámými způsoby. Proto jsou datové struktury rozděleny na co nejmenší kompaktní celky, každý je uložen v samostatné tabulce a odpověď na dotaz je pak sestavena z obsahu téhoto tabulek. Efektivita databázového systému je pak silně ovlivněna výběrem a implementací operací pro sestavování odpovědi, např. s využitím sekundárních indexů. Relační databáze, tak jak je známe, také poskytují záruky zachování konzistence databáze tím, že plně implementují transakční zpracování dotazů (vlastnosti ACID) s nejvyšší úrovní izolace transakcí (viz sekce 3.2.1.1).

Sestavování odpovědí pomocí spojování tabulek a transakční zpracování s vlastnostmi ACID jsou tedy základními kameny, na kterých stojí klasické relační databáze. Potřeba efektivní implementace téhoto dvou vlastností je důvodem, proč relační databáze nebyvají plně distribuované a tedy horizontálně škálovatelné. Zdá se ale, že čím dál více aplikací je ochotných se téhoto dvou základních kamenů do jisté míry vzdát a umožnit tím výrazné zvýšení efektivity a škálovatelnosti databázového systému.

Maximální atomizace datových záznamů do jednotlivých tabulek je přístup, který umožňuje efektivně realizovat jednotlivé operace zápisu dat, ale sestavování odpovědi z různých tabulek může být velmi náročné. U databázových systémů typu NoSQL vidíme naopak často posun směrem k trendu, kdy se vynaloží více úsilí při návrhu struktury dat a při jejich ukládání proto, aby pak vyhodnocení vybraných typů dotazů bylo rychlejší a systém jich zvládal vyhodnotit více najednou. NoSQL databáze tak poskytují specializovaná úložiště pro konkrétní typy dat, umožňují seskupování různých záznamů, na které se bude poté většinou přistupovat společně, a také replikaci dat na více uzlech.

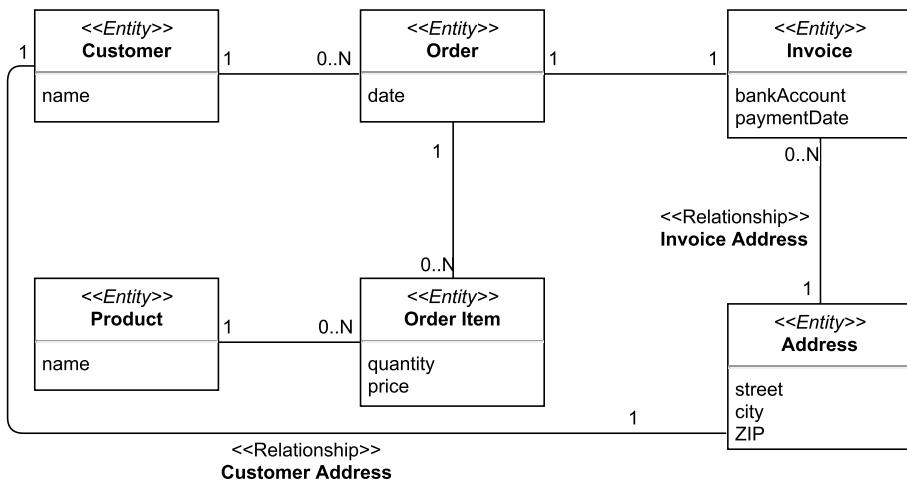
Další motivací pro vznik systémů odlišných od tradičních databází je flexibilita schématu uložených dat. Samozřejmě, že u relačních databází je možné měnit schéma jednotlivých tabulek i „za běhu“, ale tyto operace jsou v SQL realizovány pomocí *jazyka pro definici dat* (DDL), který není součástí sady příkazů běžně přístupných všem uživatelům. Změny ve schématu jsou většinou realizovány administrátorem systému a opět mohou být poměrně náročné.

U NoSQL databází se naopak od základu počítá s tím, že schéma spravovaných dat může být nejednotné a proměnlivé. Úroveň této flexibility se pro různé typy NoSQL databází liší. U některých systémů schéma neexistuje vůbec a je tedy zcela v režii aplikace, aby „věděla“, jaká data do systému ukládá. U jiných systémů se předpokládá, že záznamy stejného typu mají podobné schéma. U dalšího typu databází jsou pravidla na rozšiřování schématu určena přesněji, ale jsou jedním ze základních nástrojů, které mají uživatelé a aplikace v rukou. V následující sekci se budeme věnovat obecným principiálním rozdílům v uvažování datového návrháře SQL a NoSQL databází.

5.2 Datové modely v NoSQL databázích

Vytváříme-li databázové schéma, prvním a zásadním krokem je identifikace entit a jejich vzájemných vztahů. Výsledkem tohoto procesu bývá tzv. *entity-relationship diagram* (*E-R diagram*). Uvažujeme-li v intencích relačních databází, existuje přímočáry, téměř algoritmizovatelný způsob, jak takový diagram poté převést na relační databázové schéma. Jak velí teorie i praktická zkušenosť, výsledné schéma by mělo splňovat jistá pravidla, např. *třetí normální formu* (3NF) [74]. Při použití NoSQL technologií se těchto pravidel často nedržíme a zvažujeme i jiná než *normalizovaná* databázová schémata. Nalezení vhodného schématu je přitom pro efektivnost zpracování databázových operací zásadní, obzvlášť pokud má být databáze distribuovaná. Špatné rozhodnutí v této fázi může efektivitě výrazně uškodit.

Na obrázku 5.1 je zobrazen E-R diagram v notaci UML [47] zachycující zjednodušený systém pro správu zákazníků (*Customer*), objednávek (*Order*) a faktur (*Invoice*). Jednotlivé objednávky mají vždy několik položek (*Order Item*) odpovídajících různým výrobkům (*Product*). V diagramu je dále zachycena entita adresa (*Address*), která má jednak vazbu na zákazníky (*Customer Address*) a jednak na faktury coby fakturační adresa (*Invoice Address*). U entit jsou naznačeny nejdůležitější atributy a diagram ukazuje vztahy mezi entitami včetně jejich kardinalit.



Obrázek 5.1: Příklad E-R diagramu systému pro správu zákazníků a objednávek

Převodem tohoto diagramu na relační databázové schéma dostaneme sadu relací (tabulek) např. tak, jak je vidět na obrázku 5.2 na následující straně. Pro každou entitu vznikne právě jedna relace obsahující atributy dané entity, často s přidaným atributem pro primární klíč. Atributy tvořící primární klíč jsou ve schématu podtrženy. Vztahy jsou realizovány provázáním přes cizí klíče (značené jako FK), kdy jedna tabulka obsahuje jako atribut primární klíč jiné tabulky. Např. atribut `addressID` relace `Customer` obsahuje hodnoty primárního klíče `addressID` z tabulky `Address`. Intuitivně vzato, u takto normalizovaného databázového schématu je datový model rozložen do co nejmenších smysluplných celků. Výhodou takového schématu je flexibilita dotazování, protože umožňuje jednak přímočáre vyhodnocení jednodu-

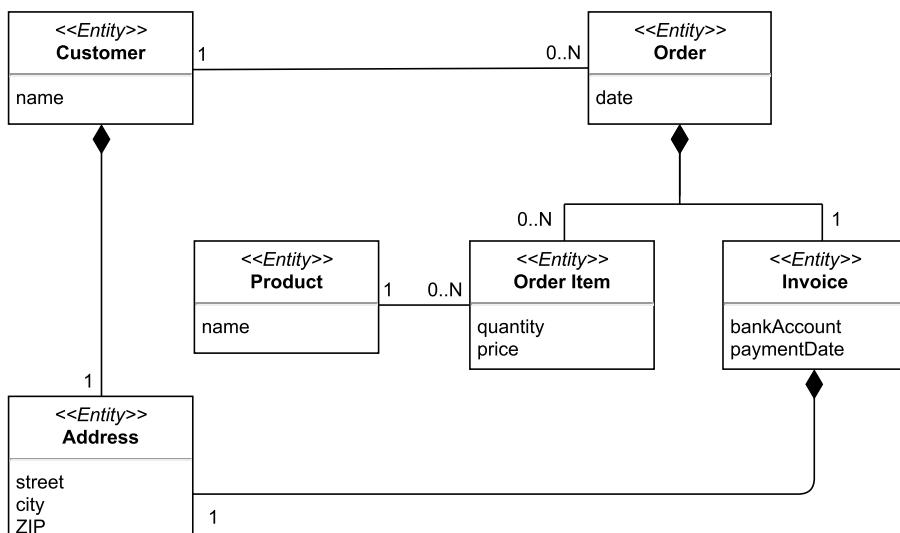
Customer	Order	Invoice
<u>customerID</u>	<u>orderNumber</u>	<u>invoiceID</u>
name	date	bankAccount
addressID (FK)	customerID (FK)	paymentDate
		addressID (FK)
		orderNumber (FK)

Product	OrderItem	Address
<u>productID</u>	<u>orderNumber (FK)</u>	<u>addressID</u>
name	<u>productID (FK)</u>	street
	quantity	city
	price	ZIP

Obrázek 5.2: Normalizované relační databázové schéma systému pro správu zákazníků a objednávek

chých dotazů nad jednotlivými tabulkami a jednak sestavování komplexních odpovědí pomocí spojování záznamů z různých relací.

NoSQL databáze vybízí k trochu jinému uvažování o datech a k jinému přístupu k datovému modelování, který vychází z pojmu agregací (*aggregates, celky*) [144]. Vyjdeme-li z E-R diagramu na obrázku 5.1 na předchozí straně, můžeme v něm nalézt entity, které s některými jinými entitami tvoří logické celky. S využitím operátora agregace (znak diamantu v UML notaci) můžeme toto schéma překreslit např. tak, jak je zobrazeno na obrázku 5.3. Zde vidíme klíčové entity *Customer* a *Order*, které v sobě sdružují ostatní data. Entita *Invoice* dále tvoří celek s entitou *Address*.



Obrázek 5.3: E-R diagram využívající koncept agregací

Pří modelování pro NoSQL databáze se před námi v tuto chvíli otevří několik možností, jak takový E-R diagram převést na databázové schéma. Můžeme se například přesně držet navrženého schématu s celky zákazník, objednávka a faktura a sdružit ostatní data do těchto celků. V notaci jazyka JSON by pak data s tímto schématem mohla vypadat např. jako v příkladu 5.1.¹

Příklad 5.1: Příklad schématu s využitím agregací v notaci JSON

```
// v kolekci "Customer"
{
    "customerID": 1,
    "name": "Jan Novák",
    "address": {
        "city": "Praha",
        "street": "Krásná 5",
        "ZIP": "111 00"
    }
}

// v kolekci "Order"
{
    "orderNumber": 11,
    "date": "2015-04-01",
    "customerID": 1,
    "orderItems": [
        {
            "productID": 111,
            "name": "Vysavač ETA E1490",
            "quantity": 1,
            "price": 1300
        },
        {
            "productID": 112,
            "name": "Sáček k vysavači ETA E1490",
            "quantity": 10,
            "price": 300
        }
    ]
}

// v kolekci "Invoice"
{
    "invoiceID": 2015003,
    "orderNumber": 11,
    "bankAccount": "17-5364640439/0100",
    "paymentDate": "2015-04-16",
    "address": {
        "city": "Brno",
        "street": "Slunečná 7",
        "ZIP": "602 00"
    }
}
```

¹ V tomto i následujících příkladech budeme používat zápis // komentář, který ale není validní konstrukcí formátu JSON (viz sekce 2.1).

Entity *Address*, *Order Item* a *Product* nyní nejsou uloženy v samostatných kolekcích (tabulkách), ale jsou součástí ostatních entit. Provázání přes cizí klíče mezi hlavními entitami zůstává. Toto schéma je výhodné, pokud bude aplikace pracovat s daty jako celky, které jsme určili jako hlavní (zákazník, objednávka, faktura). Pokud bude aplikace například chtít přistoupit k jedné objednávce, kolekce *Order* obsahuje všechna potřebná data na jednom místě. V případě rozdělení dat objednávky mezi tři tabulky *Order*, *OrderItem* a *Product* (jak je uvedeno v příkladu 5.2 na straně 90) se budou při dotazu data jedné objednávky skládat z těchto tří tabulek. Obecně vzato, pokud jsou data distribuována na několika uzlech, efektivní implementace propojování záznamů je složitá, a proto MySQL databáze vybízí právě ke sdružování dat do logických celků podle toho, jak se k nim bude přistupovat.

Použití *denormalizovaných* schémat databáze může ale vést k redundantnímu uložení některých dat. Např. v příkladu 5.1 na předchozí straně je adresa uložena na dvou místech databáze. Toto je daň, kterou se můžeme rozhodnout zaplatit, a v některých případech je tato redundance dokonce z hlediska aplikace žádoucí. Podrobněji se důsledkům použití denormalizovaných schémat budeme věnovat zejména v kontextu dokumentových databází v kapitole 7.

Svět nepodléhá třetí normální formě

Uvedený příklad s adresou je výbornou ilustrací toho, že dle třetí normální formy „správně“ navržená struktura dat nemusí zaručit skutečnou správnost v reálném světě. Uvažujme případ, kdy zákazník (*Customer*) změní sídlo firmy (*Address*). Pokud bude faktura (*Invoice*) jednoduše provázána s adresou zákazníka jako v normalizovaném schématu, změní se adresa i na všech v minulosti vytvořených fakturách, které vzniknou typicky pomocí operace spojení tabulek. To není žádoucí, protože takto vzniklé faktury budou obsahovat neplatné údaje. (Mimořadem, mnoho služeb pro správu faktur provozovaných na internetu je náhylných právě k této chybě.)

Správným řešením v prostředí relačních databází je v tomto případě zachytit tuto časovou složku vztahu adresy a zákazníka: vztah mezi entitami *Customer* a *Address* je uložen v separátní entitě (v případě relační databáze tedy v separátní tabulce) a obsahuje nejenom identifikátory zákazníka a adresy, ale též časovou platnost od–do. Taková struktura nám pak dovoluje nejenom reálně správný datový model (faktura je vázána na datum, a proto můžeme získat správnou adresu v daný moment), ale rovněž uchovává historické informace, at již pro účely reportingu nebo auditu (viz též rámeček *Účetní v záznamech negumují* na straně 50).

V prostředí NoSQL databází popsaný problém vyřešíme jednoduše pomocí denormalizovaného schématu databáze – pokud je adresa uložena spolu s fakturou, nemůže dojít k vygenerování neplatné faktury ani ve chvíli, kdy zákazník změní sídlo firmy. Dále nad takto strukturovanými daty budeme moci efektivně provést např. agregaci typu „žebříček deseti měst, kde sídlí naši zákazníci, seřazený podle počtu faktur jím vystavených v tomto roce“ v systémech jako je MongoDB nebo Elasticsearch. V distribuovaných databázových systémech má struktura dat vždy poměrně zásadní vliv na výkon.

Datový model a databázové schéma lze vždy navrhnout několika různými způsoby a NoSQL databáze nám k tomu poskytují silný aparát. V našem příkladu bychom např. mohli všechny objednávky ukládat přímo v datové struktuře zákazníků, faktury ukládat pod objednávkami a mít tedy všechna data jen v jedné hierarchické struktuře typu *Customer*. Na druhou stranu, u jakéhokoli typu databáze máme vždy možnost použít normalizované schéma (viz příklad 5.2 na straně 90) a tabulky provázat přes cizí klíče.

V následující části budeme hovořit o čtyřech typech NoSQL databází. Výše představený způsob uvažování o datech a jejich modelování je možné využít u prvních třech typů. Čtvrtým typem jsou grafové databáze, které pracují se zcela odlišným modelem dat.

5.3 Typologie NoSQL databází

Databázové systémy označované jako NoSQL zažívají v současnosti velký rozmach a jednotlivé systémy procházejí často poměrně překotným vývojem. V porovnání s tradičními relačními systémy se konkrétní NoSQL databáze navzájem mnohem více liší na mnoha úrovních a některé z nich je těžké zařadit do jedné kategorie. Abychom se ale mohli bavit o společných a rozdílných vlastnostech jednotlivých systémů, pokusme se nejprve vymezit základní typy tak, jak se v současnosti používají v literatuře i mezi vývojáři NoSQL systémů. Tato typologie se opírá zejména o datový model jednotlivých systémů.

- *Databáze typu klíč-hodnota*: Tyto databázové systémy mohou ukládat prakticky jakékoli objekty na základě jejich unikátních klíčů. Operace nad úložiště typu klíč-hodnota jsou pak poměrně jednoduché a primárně neposkytují žádný způsob, jak s daty manipulovat nebo je vyhledávat na základě uloženého obsahu – jen podle určeného klíče. Datový model u těchto systémů je tedy absolutně svobodný.
- *Dokumentové databáze*: Systémy tohoto typu ukládají a spravují různé druhy strukturovaných *dokumentů*, o kterých se předpokládá, že mají *samopopisný* charakter.² Typickým příkladem je formát JSON, XML a jiné hierarchické struktury (viz kapitola 2). Dokumentové databáze umožňují přistupovat k dokumentům a vyhledávat v nich podle jejich obsahu.
- *Sloupcové databáze*: Datový model těchto systémů si můžeme představit jako tabulku, u které je možné do každého řádku volně přidávat sloupce bez nutnosti přidat je i do řádků ostatních. Toto volné nakládání se sloupcí nesnížeje výkonost sloupcových databází, které bývají masivně distribuované.
- *Grafové databáze*: Tento typ databázových systémů se výrazněji liší od předchozích tří. Je určen pro data, která je vhodné modelovat a hlavně dotazovat jako grafy, tedy objekty a vztahy mezi nimi. Data jsou vnitřně strukturována

² Samopopisným charakterem máme na mysli, že jednotlivé fragmenty dat s sebou přímo nesou také informaci o svém významu – typickým příkladem je název pole a jeho hodnota. U relačních databází je tato informace odtržená od dat samotných a je obsažena v názvech sloupců a názvech tabulek.

tak, aby bylo možné efektivně vyhodnocovat různé grafové úlohy, jako je hledání sousedů nebo cest v grafu.

Společným a rozdílným vlastnostem jednotlivých typů NoSQL databázových systémů a jejich konkrétním zástupcům se budeme podrobně věnovat v následujících čtyřech kapitolách.

6.

Databáze typu klíč-hodnota

Úložiště typu klíč-hodnota (*key-value stores*) si můžeme představit jako asociativní pole (*map*) nebo hašovací tabulkou ukládající *hodnoty* podle unikátního *klíče*. V těchto typech úložišť mohou být obecně uloženy hodnoty jakéhokoli typu – často se v nich ukládají např. data *serializovaná* pomocí protokolů Apache Thrift nebo Protocol Buffers (viz sekce 2.6). Databázový systém primárně neposkytuje žádný způsob, jak data efektivně prohledávat podle obsahu uložených hodnot.

Proč bychom tedy chtěli takový systém použít? Protože mnoha aplikacím přístup k datům přes primární klíč dostačuje, tato úložiště bývají extrémně rychlá a dají se velice efektivně distribuovat. Jak uvidíme v této kapitole, v současnosti existují jistě desítky systémů, které můžeme zařadit do této třídy a které se mohou poměrně zásadně lišit svým zaměřením a vlastnostmi. Při uvádění příkladů se zaměříme na systém Riak,¹ což je jeden ze systémů, jehož síla je zejména v masivní distribuci a replikaci dat a tím dosahované vysoké propustnosti operací čtení a zápisu.

¹ <http://basho.com/riak/>

6.1 Principy

Úložiště typu klíč-hodnota jsou velmi jednoduchá, a to jak strukturu dat, tak i operacemi, které poskytují pro práci s daty. Základní API poskytované všemi systémy tohoto typu obsahuje jen tři operace: vložení hodnoty pro daný klíč (operace **PUT**), získání hodnoty pro daný klíč (operace **GET**) a smazání dvojice klíč-hodnota (operace **DELETE**). Úložiště nijak nezkoumá obsah vkládaných hodnot a je pak tedy zcela na klientské aplikaci, aby obsahu rozuměla a znala klíče pro získání uložených hodnot. Některé systémy ovšem poskytují i další možnosti, jak s uloženými daty pracovat (viz sekce 6.3).

6.1.1 Základní operace a práce s klíči

Uvedme příklad volání základních operací vložení, získání a smazání hodnoty pro daný klíč na databázovém systému Riak přes HTTP REST API [138]. V následujících příkladech používáme populární nástroj *curl*² umožňující HTTP komunikaci s daným serverem. Hlavním parametrem tohoto nástroje je URL adresa, se kterou *curl* naváže spojení a na standardní výstup vytiskne odpověď od daného serveru.

```
curl -X PUT http://localhost:8098/buckets/autori/keys/David -H "Content-Type: application/json" -d '{"name": "David Novák", "affiliation": "Masarykova univerzita"}'
```

Tento příkaz otevře spojení na `localhost` na portu 8098, což je implicitní port systému Riak. Příkaz do úložiště vloží pod klíč `David` JSON řetězec uvedený za prepínačem `-d`. Že se jedná o operaci vkládání, je určeno pomocí *metody* HTTP protokolu, v tomto případě `PUT`.³ Pokud zpracování příkazu proběhne v pořádku, volání API nevrátí žádnou hodnotu.

V následujícím příkladu je kurzívou vyznačena návratová hodnota:

```
curl -X GET http://localhost:8098/buckets/autori/keys/David
{"name": "David Novák", "affiliation": "Masarykova univerzita"}
```

Tento příkaz používá metodu `GET`, čímž získá uloženou hodnotu pro klíč `David` a vrátí ji na standardní výstup.

Další uvedený příkaz hodnotu pro tento klíč z úložiště smaže:

```
curl -X DELETE http://localhost:8098/buckets/autori/keys/David
```

Klíčovou roli pro práci s úložištěm typu klíč-hodnota hraje tedy *klíče*, jakožto jednoznačné identifikátory uložených datových objektů. Existuje mnoho aplikací a datových množin, ve kterých je k dispozici nativní primární klíč přirozeně identifikující objekty daného typu. Často se úložiště typu klíč-hodnota používají např. pro ukládání informací z tzv. *session* (česky relace) mezi klientem a serverem, kde *session ID* je přirozeným klíčem pro data z dané relace. Rychlé a jednoduché úložiště na straně serveru pak udržuje pro dané session ID všechny potřebné in-

² <http://curl.haxx.se>

³ Kromě nejběžněji používaných metod `GET` a `POST` definuje protokol HTTP/1.1 i další metody, např. právě `PUT` a `DELETE` [18].

formace jako jeden objekt, který je vytvořen při zahájení relace a získán vždy, když klient serveru zašle dané session ID. Dalšími klasickými případy použití jsou správa obsahu nákupního košíku v e-shopu, kde každý nákupní košík má vygenerovaný jednoznačný identifikátor, nebo udržování informací o profilech a preferencích uživatelů, kde klíčem je identifikátor uživatele *user ID*.

Pokud aplikace nemá pro datové objekty přirozený klíč, je potřeba klíč odvodit, např. pomocí hašovací funkce a časového razítka. Většinou je možné vygenerování takového klíče nechat na databázovém systému ve chvíli, kdy do něj objekt vkládáme.

6.1.2 Jmenné prostory klíčů

Většina databázových systémů typu klíč-hodnota umožňuje uložená data rozčlenit podle jejich typu do oddělených úložišť typicky nazývaných *buckety* (někdy překládáno jako *příhrádky*). Buckety můžeme přesněji chápat jako jmenné prostory klíčů (*namespaces*), které logicky oddělují záznamy různých typů, ale fyzicky jsou všechny uloženy ve stejném hašovacím prostoru.⁴

Pracujeme-li se systémem Riak, není potřeba jmenný prostor klíčů nijak explicitně vytvářet – Riak jednoduše vypočítá hašovanou hodnotu dvojice (*namespace*, *key*). Tato dvojice spolu tvoří místo (*location*), kam bude hodnota v systému Riak uložena. V následujícím příkladu používáme Java API systému Riak 2.0 k demonstraci použití prostoru klíčů, klíče a uložení JSON řetězce do úložiště pomocí příkazu *StoreValue*:

```
// vytvoření jmenného prostoru klíčů
Namespace namespace = new Namespace("authors");

// místo pro uložení je v Riak tvořeno dvojicí (namespace, key)
Location location = new Location(namespace, "David");
String value =
    "{\"name\": \"David Novák\", \"affiliation\": \"Masarykova univerzita\"}";

// vytvoření a vykonání příkazu pro uložení hodnoty na dané místo
StoreValue storeCmd = new StoreValue.Builder(value).withLocation(location).build();
client.execute(storeCmd);
```

V následující části textu budeme používat příklad databáze, ve které jsou uloženy informace o uživatelsích e-shopu. Řekněme, že aplikace potřebuje o každému uživateli udržovat informace o uživatelském profilu, data z právě probíhající webové relace (session) a obsah nákupního košíku. V úložišti typu klíč-hodnota můžeme ukládat všechny tyto informace na jednom místě (v jednom jmenném prostoru) – viz obrázek 6.1 na následující straně. Tento přístup může být výhodný, pokud naše aplikace bude často potřebovat najednou všechny tři zmíněné druhy infor-

⁴ Hašovací funkce přiřazuje libovolnému klíči *hašovanou hodnotu* z *hašovacího prostoru* (většinou jde o interval celých čísel). Nejčastěji se setkáme s hašovací funkcí při vytváření vyhledávacího indexu v klasických (i jiných) databázích. Takový index je tvořen *hašovací tabulkou*, jejíž jednotlivá pole obsahují vždy všechny klíče se stejnou hodnotou hašovací funkce, tato pole jsou také často označována termínem *bucket*. V našem textu jsme použili termín bucket ve významu „odděleného úložiště“ a dále se budeme raději držet významově jasnějšího termínu jmenný prostor.

Namespace *User*

Klíč:	<i>userID</i>
Hodnota:	<i>userProfile</i>
	<i>sessionData</i>
	<i>shoppingCart</i>
	• <i>item 1</i>
	• <i>item 2</i>

Obrázek 6.1: Uložení všech dat v jednom jmenném prostoru klíčů

mací o daném uživateli a s tímto nastavením se k nim dostane pomocí jednoho přístupu do databáze.

Pokud ale různé části aplikace vyžadují vždy jen informace z jednoho ze tří logických polí (uživatelský profil, relace nebo nákupní košík), tak bude výhodnější data rozdělit do tří jmenných prostorů tak, jak je vidět na obrázku 6.2.

Namespace *UserProfiles*

Klíč:	<i>userID</i>
Hodnota:	<i>userProfile</i>

Namespace *Session*

Klíč:	<i>sessionID</i>
Hodnota:	<i>sessionData</i>

Namespace *ShoppingCart*

Klíč:	<i>userID</i>
Hodnota:	<i>shoppingCart</i>

- *item 1*
- *item 2*

Obrázek 6.2: Rozdělení dat do několika jmenných prostorů klíčů

Pokud bychom potřebovali přistoupit ke všem datům o uživateli, tak tato konfigurace bude vyžadovat tři přístupy do úložiště. Toto dilema je typickým příkladem rozhodnutí o rozdělení agregace tak, jak je popsáno v sekci 5.2.

6.1.3 Druhy úložišť typu klíč-hodnota

Za počátek vývoje distribuovaných databázových systémů typu klíč-hodnota je tradičně považován článek o systému Amazon Dynamo [76] (nyní přístupnému jako placená služba Amazon Dynamo DB⁵). Tento článek identifikuje hlavní výzvy, které je potřeba vyřešit při realizaci distribuovaného, persistentního, efektivního a odolného úložiště typu klíč-hodnota, a navrhuje různá řešení těchto výzv. Samozřejmě již dříve existovala např. distribuovaná paměťová cache Memcached⁶ nebo výkonné lokální úložiště Berkeley DB.⁷ Od té doby vznikl a stále vzniká velký počet systémů, které lze zařadit mezi úložiště typu klíč-hodnota, ať už volně šířitelných nebo komerčních. Mezi ty aktuálně nejpopulárnější patří:

⁵ <http://aws.amazon.com/dynamodb>

⁶ <http://memcached.org>

⁷ <http://www.oracle.com/us/products/database/berkeley-db/index.html>

- persistentní distribuované systémy jako Riak, Redis⁸ nebo Infinispan,⁹
- knihovny pro tvorbu vestavěných diskových úložišť jako Berkeley DB, LevelDB,¹⁰ RocksDB¹¹ nebo MapDB¹²,
- paměťové cache jako Memcached, Ehcache¹³ nebo Hazelcast.¹⁴

Pro aktuální bohatý seznam systémů typu klíč-hodnota doporučujeme například stránky DB-Engines.com.¹⁵

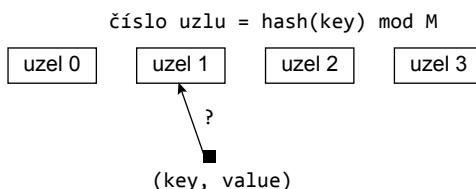
6.2 Realizace a vlastnosti

V této sekci se zaměříme na vybrané vlastnosti a výzvy, které se týkají většiny databázových systémů typu klíč-hodnota, a popíšeme principy jejich řešení v současných implementacích.

6.2.1 Distribuce dat

Většina systémů typu klíč-hodnota může pracovat v distribuovaném režimu, kdy jsou data rozdělena mezi více uzlů distribuovaného systému (viz sekce 3.3.1). Rozhodnutí o tom, na který uzel bude uložena dvojice *(key, value)*, je učiněno buď přímo podle konkrétního klíče *key*, nebo podle hodnoty hašovací funkce *hash(key)*.

Nabízí se použití poměrně standardního přístupu, kdy bychom klíč přidělili danému serveru na základě zbytku po dělení hašované hodnoty počtem uzlů *M*, tedy $\text{hash}(\text{key}) \bmod M$ (viz obrázek 6.3). Tímto přístupem lze dosáhnout rovnoměrného rozložení hodnot mezi uzly a je ho možné dobře použít, pokud se množina uzlů nemění. Pokud ale například přidáme do systému jeden uzel, změní se hodnota *M*, tím i výsledky operace modulo a bylo by potřeba přesunout prakticky všechna data na jiné uzly.



Obrázek 6.3: Princip standardního hašování založeného na operaci modulo

⁸ <http://redis.io>

⁹ <http://infinispan.org>

¹⁰ <https://github.com/google/leveldb>

¹¹ <http://rocksdb.org>

¹² <http://mapdb.org>

¹³ <http://ehcache.org>

¹⁴ <http://bazelcast.com>

¹⁵ <http://db-engines.com/en/ranking/key-value+store>

Využití přístupu „zbytek po dělení“

Přestože rozdelení dat pomocí zbytku po dělení trpí zmíněnou „statičností“, v praxi se efektivně využívá. Např. Elasticsearch při ukládání dokumentu získá zbytek po dělení hašované hodnoty směrovacího klíče (*routing value*) – kterým je implicitně identifikátor dokumentu – počtem primárních oddílů indexu (*shards*), a uloží dokument do příslušného oddílu. Směrovací klíč přitom může uživatel definovat, takže má kontrolu nad distribucí dat, např. vzhledem k dodržení principu lokality, tj. ukládání dat, která k sobě patří, společně.

Zmíněná „statičnost“ uložení dat se konkrétně projevuje nemožností změnit počet primárních oddílů databáze (*primary shards*) po jejím vytvoření – při změně počtu primárních oddílů bychom pro stejnou hodnotu získali jiný zbytek po dělení. Doplňme, že jiné obdobné systémy, např. Solr,¹⁶ disponují funkcí rozdelení primárních oddílů (*shard splitting*), ale platí za to relativně vysokou cenu: taková operace bude vždy náročná na prostředky a výpočetní kapacitu, a od jistého objemu dat bude prakticky neproveditelná.

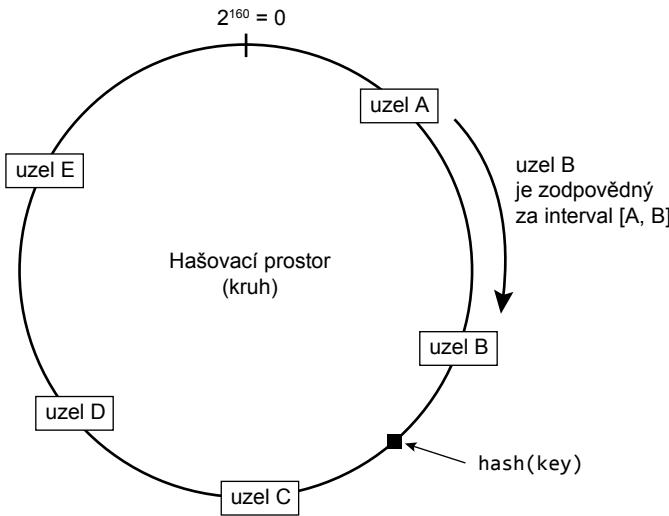
Doplňme, že v kontextu Elasticsearch je zmíněná nevýhoda prakticky neutralizována v okamžiku, kdy si uvědomíme, že data nemusíme ukládat do jediné databáze (v terminologii Elasticsearch *indexu*), ale do několika, a můžeme s nimi pracovat jako s jedním logickým celkem. To nám přináší zcela zásadní výhodu v tom, že můžeme přidávat (a samozřejmě též odebírat) další databáze (*indexy*), podle toho, jak roste objem dat.

Protože v kontextu Big Data často předem nevíme, kolik výpočetní kapacity bude potřeba, je vhodné mít efektivní možnosti, jak dělit data dynamicky. Proto se často využívá princip *konzistentního hašování* (*consistent hashing*) [112], ve kterém je každý uzel zodpovědný za souvislý interval (nebo intervaly) hašovacích klíčů. Jak toho dosáhnout ilustruje obrázek 6.4 na následující straně.

Každému uzlu v systému je přiřazen hašovací klíč ze stejné domény, kterou má hašovací funkce na klíčích (v našem příkladu má doména rozsah $[0, 2^{160} - 1]$). Každý uzel pak spravuje všechny klíče v intervalu mezi klíčem předcházejícího uzlu a svým klíčem. Hašovací prostor na obrázku je zobrazen jako kruh, protože interval mezi dvěma uzly může *přecházet přes nulu* (viz interval mezi uzly E a A). V případě přidání nového uzlu mu systém také přidělí klíč z domény hašovací funkce, čímž se rozštěpí právě jeden interval klíčů mezi dvěma existujícími uzly. Fyzicky jsou pak z existujícího uzlu přesunuta data příslušející nově příchozímu uzlu a žádná další data v systému není nutné přesouvat.

Pro rozšíření informace o změně v množině uzlů se používají tzv. *gossip* protokoly (česky doslova *klepy* nebo *drby*). Název této rodiny protokolů poměrně dobře vystihuje jejich podstatu: každý uzel v pravidelných časových intervalech vybere náhodně jeden ze svých známých uzlů, který kontaktuje a předá mu „novinky“ – v našem případě aktuální informace o uzlech v distribuovaném systému. Podle matematického modelu šíření epidemie [61] se tímto způsobem informace rozšíří

¹⁶ <http://lucene.apache.org/solr/>

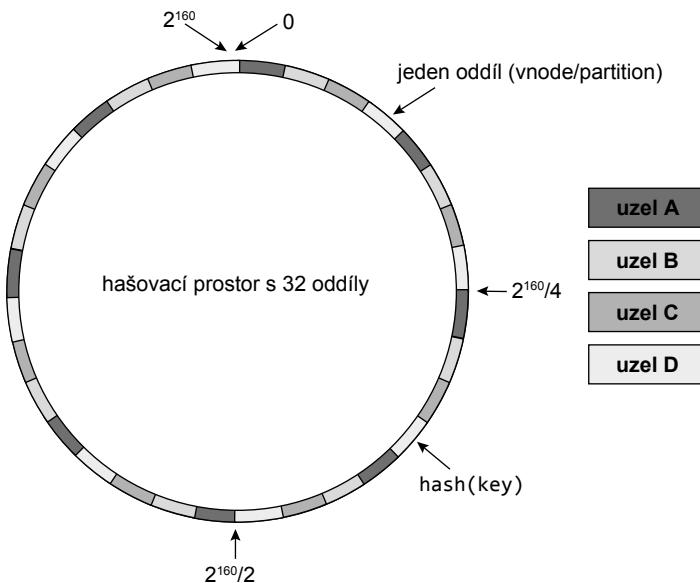


Obrázek 6.4: Princip konzistentního hašování

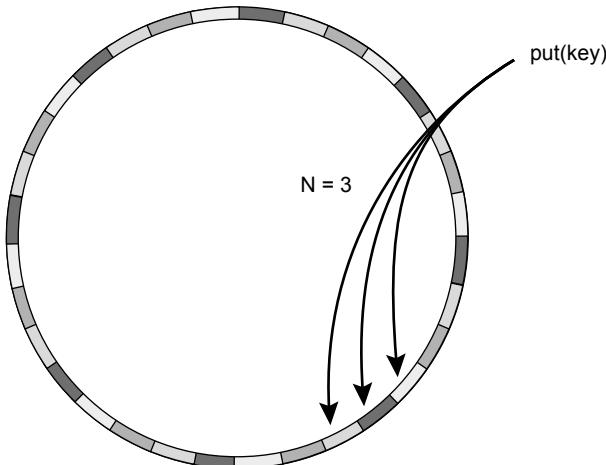
velmi rychle do všech částí distribuovaného systému. Gossip protokoly se v současných systémech využívají mimo jiné k šíření změn různých globálních nastavení systému.

Slabina konzistentního hašování může být v nerovnoměrnosti rozložení dat mezi spolupracující uzly. Hašovací funkce totiž nedokáže zaručit, že intervaly mezi uzly budou stejně dlouhé – obzvlášt pokud je uzlů relativně málo a jejich počet se mění (např. ze dvou na tři). Tento nedostatek je v mnoha systémech řešen pomocí konceptu *virtuálních uzlů*. Např. v systému Riak je tento princip využit tak, jak je naznačeno na obrázku 6.5 na následující straně: doména hašovací funkce je rozdělena na fixní počet stejně velkých intervalů neboli virtuálních uzlů (v systému Riak označovaných jako *vnode* nebo *partition*). Tyto intervaly jsou postupně přiřazovány participujícím fyzickým uzlům (serverům), které jsou v obrázku vyznačeny různým podbarvením. Jak je vidět i z obrázku, při použití této techniky mohou být data na fyzické servery rozdělena rovnoměrně. Počet virtuálních uzlů je parametrem systému, který je potřeba uvést při jeho inicializaci. Tento princip rozdělování dat se používá i v jiných typech NoSQL databází, např. v systému Cassandra (viz kapitola 8).

Jak je zmíněno v sekci 3.3, rozdělení dat se většinou kombinuje s replikací dat kvůli dosažení většího výkonu operací čtení a případně i zápisu a kvůli zvýšení tolerance systému k výpadkům (*fault tolerance*). Pokud použijeme konzistentní hašování, můžeme data ukládat nejen do uzlu zodpovědného za daný interval, ale také do několika následujících intervalů (počet záleží na nastaveném replikačním faktoru N). Na obrázku 6.6 na následující straně je tento princip naznačen pro $N = 3$ v kombinaci s využitím virtuálních uzlů. V systému Riak je možné nastavit hodnotu N jako globální parametr pro celý systém, nebo zvlášt pro každý prostor klíčů.



Obrázek 6.5: Princip využití virtuálních uzlů v systému Riak



Obrázek 6.6: Princip replikace v systému Riak

6.2.2 Konzistence a dostupnost dat

Způsob replikace dat představený na konci předchozí sekce může být využit jak pro master-slave, tak pro peer-to-peer replikaci (viz sekce 3.3) a záleží na konkrétním systému, kterou z variant implementuje. Úložiště typu klíč-hodnota jsou často využívána aplikacemi náročnými na zápis (*write intensive*), které potřebují vložit nový objekt nebo jej změnit třeba každou milisekundu. Pro zvýšení propustnosti operací zápisu tedy nelze využít master-slave replikaci a je nutná peer-to-peer replikace, protože u ní každý z N uzlů nesoucích repliky daného klíče může obsluhovat i operace zápisu. Např. systém Riak využívá peer-to-peer replikaci spolu s kvóry zápisu W a čtení R (viz sekce 3.3.3). Tento princip umožňuje systému

zapisovat, resp. číst data i pokud je $N - W$ resp. $N - R$ uzelů nefunkčních. Správné využití těchto kvůr samozřejmě ovlivňuje zachování konzistence dat.

U systému Riak je možné hodnoty W a R nastavit jednak pro celý systém a jednak pro jednotlivé operace čtení a zápisu. Řekněme, že v našem systému je každá dvojice $(key, value)$ replikována na třech uzlech, tedy $N = 3$, a globálně jsou nastaveny hodnoty $W = 2$ a $R = 2$, což odpovídá pravidlům silné konzistence popsaným v sekci 3.3.3. Nastavením hodnoty $R = 1$ pro konkrétní operaci $read(key)$ říkáme, že operace má přistoupit jen na jeden libovolný uzel obsahující dvojici $(key, value)$. Tím dosáhneme toho, že operace bude zpracována nejrychleji, jak je možné, ale zároveň se tím vzdáváme záruk silné konzistence dat. Riskujeme totiž, že nebude vrácena nejnovější zapsaná hodnota $value$, protože poslední operace zápisu $write(key, value)$ s kvórem zápisu $W = 2$ mohla zapsat novou hodnotu právě jen na zbylé dva uzly nekontaktované naší operací čtení. Použití peer-to-peer replikace a principu kvůr je velmi časté v současných systémech typu klíč-hodnota i v jiných NoSQL databázích.

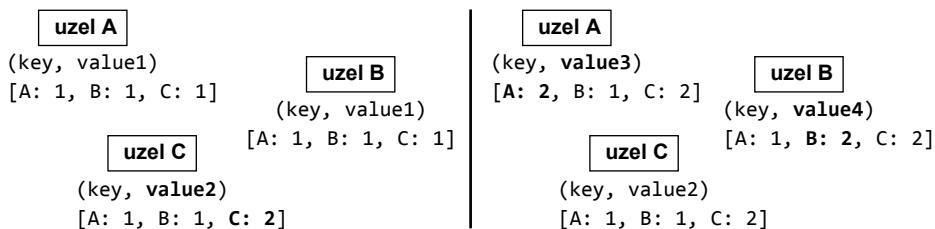
Další často používanou technikou, která dokáže zvýšit dostupnost systému pro zápis (*write availability*) při dočasném výpadku některých uzelů, jsou tzv. *hinted handoffs* (česky nepřesně opožděné přihrávky). Pokud je při operaci zápisu nedostupný některý z uzelů, na který se má daná hodnota zapsat, nedojde k zablokování operace zápisu, ale k jejímu přesměrování na jiný, náhradní uzel. Při návratu nefunkčního uzelu do systému mu náhradní uzel předá všechny změny, které se mezikádem staly.

Při použití peer-to-peer replikace, kde každý z N uzelů může obsluhovat požadavky na zápis pro daný klíč, jsou nutné mechanismy pro předcházení a případné řešení konfliktů typu write-write (viz sekce 3.3.3). Fakticky jde o mechanismy, které umožní rozpoznat, která z hodnot pro daný klíč je novější, a případně říci, že došlo ke změně hodnoty na dvou uzlech zároveň, čímž vznikl konflikt. Obecně se toto řeší pomocí dodatečného atributu uloženého u daného klíče, který se změní vždy, když je změněna hodnota pro daný klíč. I v centralizovaných systémech se používají techniky jako jednoduché čítače, náhodně generovaná unikátní čísla (GUID), hašované aktuální hodnoty pro daný klíč nebo časová razítka (*timestamp*).

Použití každého z těchto přístupů v distribuovaném prostředí má svá pro i proti, ale bohužel ani jejich kombinace (např. čítač + hašovaná hodnota) nedokáže vyřešit naznačený problém v celé jeho obecnosti [144]. Proto se v peer-to-peer NoSQL databázích používají techniky složitější, a to tzv. *vektorová razítka* (*vector stamps*). Obecně řečeno, tyto techniky u každého klíče udržují vektor čítačů – pro každý uzel jeden čítač, který je zvyšen vždy, když daný uzel aktualizuje hodnotu pro daný klíč. Každý uzel udržuje informaci o hodnotách čítačů *všech* uzelů (tedy celý vektor) a uzly si průběžně vyměňují informace o stavu čítačů a aktuálních hodnotách.

Mějme pro názornost tři uzly, A, B a C, které všechny drží hodnotu pro klíč key (replikují tuto hodnotu). Řekněme, že na začátku uzly sdílí stejnou hodnotu pro tento klíč, $value1$ a vektor čítačů je inicializován na všech uzlech na hodnotu $[A: 1, B: 1, C: 1]$. Na obrázku 6.7 na následující straně vlevo je zobrazen stav ve chvíli, kdy uzel C aktualizuje hodnotu na $value2$ a zvýší hodnotu svého čítače.

Pokud v tuto chvíli proběhne synchronizace a uzly budou mít možnost porovnat své aktuální hodnoty vektorů čítačů, zjistí, že vektor [A: 1, B: 1, C: 2] je novější než vektor [A: 1, B: 1, C: 1]. Novější hodnota *value2* a s ní novější hodnota vektoru se tedy rozšíří na všechny tři uzly.



Obrázek 6.7: Příklad vektorových razítek. Vlevo: stav po inicializaci a aktualizaci hodnoty uzlem C. Vpravo: stav, když zároveň aktualizují hodnotu uzly A a B a je detekován konflikt.

Příklad pokračuje v pravé části obrázku situací, ve které uzel A a uzel B chtějí zároveň aktualizovat hodnotu klíče *key* na *value3*, resp. *value4*. Oba uzly v tu chvíli zvýší hodnotu svých čítačů, ale při následné synchronizaci zjistí, že hodnoty vektorů [A: 2, B: 1, C: 2] a [A: 1, B: 2, C: 2] jsou neporovnatelné (podle hodnoty A je první vektor novější a podle hodnoty B je to naopak), což znamená konflikt při zápisu. Záleží na konkrétním systému a jeho nastavení, jak se v tu chvíli zachová – jestli bude hrozící konflikt odhalen ještě před dokončením operace zápisu, nebo jestli budou uloženy obě konfliktní hodnoty a situace bude řešena později.

V současných peer-to-peer systémech se využívá několik konkrétních technik aplikujících tento princip, které se liší zejména v tom, jak probíhá synchronizace vektorů čítačů (komunikační algoritmus mezi uzly). Konkrétně tyto systémy implementují různé varianty následujících algoritmů: vektorové hodiny (*vector clock*) [83], Lamportova časová razítka (*Lamport timestamps*) [121] nebo vektory verzí (*version vector*) [135]. Např. Riak v současné chvíli využívá algoritmus vektorových hodin, ale od verze 2.0 nabízí také možnost použít techniku nazvanou *dotted version vectors* [137], která má větší propustnost operací zápisu. Další podrobnosti o uspořádání časových razítek jsou uvedeny v sekci 12.5.

Obecně existují tři varianty řešení konfliktů při zápisu (často označováno jako proces *anti-entropy*) a všechny tři jsou aktivně využívány v současných systémech typu klíč-hodnota i v dalších typech NoSQL databází: oprava při zápisu (*write repair*), oprava při čtení (*read repair*) a asynchronní oprava (*asynchronous repair*). Oprava při zápisu vyřeší neshodu hned v zárodku, což ale může celou operaci značně zpomalit. Oprava při čtení je příkladem optimistické strategie, kdy je konflikt řešen až při následné operaci čtení, tedy jen když konflikt opravdu nastane. Třetí často využívanou možností je asynchronní proces, který prochází data, odhaluje konflikty a řeší je nezávisle na operacích zápisu či čtení. Např. Riak implementuje jednak opravy při čtení (s několika různými nastaveními) a navíc má funkci *Active Anti-Entropy*,¹⁷ která je vhodná zejména pro data, jež nejsou často čtena. V rámci této techniky si jednotlivé uzly vyměňují komprimovanou informaci

¹⁷ <http://docs.basho.com/riak/latest/theory/concepts/aae/>

o uložených datech, konkrétně strukturu zvanou *Merkle tree* [128], která umožní konflikty efektivně detekovat.

O dalších přístupech a technikách umožňujících hledání kompromisu mezi zachováním konzistence a dostupností systému se dočteme v následujících kapitolách. Samostatná kapitola 12 je věnována zejména transakčnímu zpracování v distribuovaných systémech.

6.2.3 Lokální organizace dat

Z hlediska efektivity datového úložiště je velmi důležitý způsob organizace lokálních dat typu klíč-hodnota na jednotlivých uzlech databáze. V sekci 6.1.3 jsme identifikovali několik druhů systémů typu klíč-hodnota a jedním z uvedených druhů byla lokální vestavěná disková úložiště realizovaná pomocí různých programátorských knihoven. Tyto knihovny se využívají jednak přímo v aplikačních systémech, které nepotřebují distribuovanou databázi, ale také je využívají distribuovaná úložiště typu klíč-hodnota pro lokální organizaci dat na jednotlivých uzlech.

Jednou z nejznámějších knihoven tohoto typu je bezesporu LevelDB. Tuto C++ knihovnu je možné využít jako lokální úložiště pro systémy Riak nebo Infinispan a mimo jiné tvoří základ systému IndexedDB [19] pro uložení dat v prohlížeči Google Chrome (viz sekce 14.2.2). Knihovna LevelDB je oblíbená pro svou rychlosť, jednoduchost a spolehlivost. Byla vytvořena jako open source implementace lokálního úložiště sloupcové databáze Bigtable společnosti Google. Této databázi a dalším systémům vytvořeným na podobném konceptu je věnována kapitola 8. Konkrétně principům lokálního uložení dat, které přebírá i LevelDB, se věnuje sekce 8.3.2.

Kromě LevelDB existují i další populární systémy, jako např. RocksDB vyvíjený původně pro vnitřní účely ve společnosti Facebook nebo systém MapDB implementovaný v jazyce Java a vytvořený českým programátorem. Velmi známou a používanou je také knihovna Berkeley DB, jejíž základ byl implementován v jazyce C již v polovině devadesátých let. Knihovna je nyní např. základem tří databází společnosti Oracle:¹⁸ Oracle Berkeley DB, Berkeley DB Java Edition a Berkeley DB XML.¹⁹

6.3 Práce s daty

Část systémů typu klíč-hodnota se striktně drží základní myšlenky, že jako hodnotu pro daný klíč je možné uložit prakticky cokoli, ale k dispozici jsou pouze základní operace pro práci s daty, popsané výše. Jiná současná úložiště však další možnosti práce s uloženými daty poskytuje.

¹⁸ <http://www.oracle.com>

¹⁹ <http://www.oracle.com/us/products/database/berkeley-db/overview/>

6.3.1 Sekundární indexy

Čím dál více systémů založených na principu klíč-hodnota umožňuje vytváření sekundárních indexů nad uloženými daty. Pomocí indexů pak můžeme vyhledávat datové objekty na základě obsahu jejich vybraných polí nebo atributů. Používají se standardní typy vyhledávacích indexů, které známe z relačních databází. Například systém Riak poskytuje následující druhy indexů:

- *celočíselný index*: standardní atributový index umožňující vyhledávání pomocí zvolené hodnoty nebo intervalu indexovaných hodnot,
- *binární index*: vyhledávací index použitelný na libovolných binárních datech, což koresponduje se záměrem ukládat libovolná data,
- *textový index*: index umožňující fulltextové vyhledávání v textových datech.

Protože obecně není na ukládaných hodnotách definované žádné schéma dat, indexy se nepoužívají automaticky, jak jsme zvyklí z relačních databází. Uložení hodnoty do indexu je nutné explicitně vyvolat. Tento princip je vidět na následujícím příkladu, kde při ukládání hodnoty přes HTTP REST API systému Riak vyvoláme vložení do indexu použitím dodatečných HTTP hlaviček:

```
curl -X PUT http://localhost:8098/buckets/autori/keys/David \
-H 'x-riak-index-surname_bin: Novak' \
-H 'x-riak-index-phone_int: 1234' \
-H "Content-Type: application/json" \
-d '{"name": "David", "surname": "Novák", "phone extension": 1234}'
```

V tomto příkladu jsme do binárního indexu `surname_bin` uložili hodnotu `Novak` a do celočíselného indexu `phone_int` hodnotu `1234` (typ indexu se pozná podle přípony `_bin`, resp. `_int`). Riak udržuje indexy lokálně pro data jednotlivých virtuálních uzlů distribuovaného systému. Po zavolení předcházejícího příkazu jsou tedy tyto indexy aktualizovány na stejném uzlu, na kterém se fyzicky uloží daná hodnota. Vyhledávání v indexu poté vyvoláme např. takto:

```
curl http://localhost:8098/types/indexes/buckets/autori/index/surname_bin/Novak
```

Tento příkaz vyhledá v indexu `surname_bin` všechny záznamy uložené pod klíčem `Novak`. Protože data i indexy jsou v případě Riaku distribuované, toto volání ve skutečnosti vygeneruje a spustí MapReduce úlohu (viz kapitola 4), která provede vyhledávání na lokálních indexech a vrátí celkovou odpověď.

6.3.2 Redis

Zajímavým příkladem databáze postavené na principu klíč-hodnota je Redis (zkratka z *Remote Dictionary Server*). Redis pracuje s mnohem komplexnějšími datovými strukturami, jako je např. seznam (*list*), množina (*set*) nebo asociativní pole (*hash*), a poskytuje sofistikované operace nad těmito strukturami, jako např. práci s prvky na konkrétní pozici seznamu, množinové operace apod.

Redis je zároveň hybridně persistentní databází: výchozím úložištěm je operační paměť (RAM), což zajišťuje masivní propustnost operací (*throughput*), která je ale pravidelně ukládána na disk pro zvýšení odolnosti (*resiliency*). Uživatel má přitom možnost konfigurovat systém tak, aby dosáhl kýžené rovnováhy mezi těmito dvěma vlastnostmi.

Síla databáze Redis přitom tkví v kombinaci jednoduchosti a rychlosti. Jako praktický příklad můžeme uvažovat problém počítání přístupů k určitému zdroji (webová stránka, soubor ke stažení atd.). V případě systému Redis můžeme pro jeho implementaci použít nejjednodušší datovou strukturu: řetězec (*string*) v kombinaci s atomickým zvýšením jeho číselné hodnoty (*increment*). Podobně jako v řadě dalších nerelačních databází přitom využijeme denormalizaci databázového schématu: pro souhrnné čítače (*downloads:total*, *downloads:total:today*, *downloads:total:2011-05-10*) i pro čítače jednotlivých zdrojů (*downloads:/downloads/file1.mpg:total* atd.) budeme udržovat separátní klíče. Při každém jednotlivém přístupu ke zdroji je všechny zvýšíme pomocí operace **INCR**:

```
INCR downloads:total
INCR downloads:total:today
INCR downloads:total:2011-05-10
INCR downloads:/downloads/file1.mpg:total
INCR downloads:/downloads/file1.mpg:today
INCR downloads:/downloads/file1.mpg:2011-05-10
```

Základním rysem tohoto přístupu je využití jedné z hlavních vlastností systému Redis: rychlý zápis s velkou propustností pro operaci **INCR**. Získání souhrnného čítače počtu stažených souborů za dnešek pak provedeme pouze pomocí jednoduché operace **GET** klíče *downloads:total:today*. Pro získání čítače konkrétního souboru pak provedeme opět operaci **GET**, pouze na konkrétním klíči: *downloads:/downloads/file1.mpg:today*. Dobře zde také vidíme, jak název samotného klíče obsahuje jednak „doménu“ dat (*downloads*), tak konkrétní URL (*/downloads/file1.mpg*), kvůli maximálnímu usnadnění čtení ze strany aplikace.

Pravidelné vyprázdnění čítačů pro dnešek pak můžeme provádět pomocí operace **EXPIREAT**, která v zadaný čas příslušný klíč smaže:

```
# Čítač vyprší v: 2015-05-01 23:59:59
EXPIREAT downloads:total:today 1430517599
```

Z uvedeného příkladu je zjevné, jak relativně primitivní vlastnosti systému Redis umožňují implementaci sofistikované funkcionality. Klíčovým faktorem je zde přitom odlišná charakteristika systému Redis: masivní prostupnost operací. Není náhodou, že Redis ve výchozí instalaci obsahuje utilitu **redis-benchmark**, která umožňuje efektivně a transparentně ověřit výkonnost operačního systému, na němž je nainstalován, stejně jako není náhodou, že dokumentace u všech operací uvádí jejich asymptotickou složitost. Na běžném stolním počítači je Redis ve výchozím nastavení schopen provést řádově 70 tisíc operací **INCR** nebo **GET** v padesátinásobné paralelizaci za sekundu, přičemž 98 % operací proběhne pod jednu milisekundu.

Jinou základní datovou strukturou systému Redis, kterou však lze využít pro implementaci sofistikovaných požadavků, je seznam, jenž odpovídá ve většině pro-

gramovacích jazyků poli. Oblíbeným způsobem využití této datové struktury je udržování seznamů úloh pro jejich zpracování na pozadí (*background jobs, queues*). V tomto případě do seznamu pomocí operace `R PUSH` jeden proces či komponenta ukládá data pro provedení úlohy (např. konverzi obrázku nebo generování PDF reportu) a pomocí operace `L POP` je ze seznamu získává jiný proces či komponenta. Výpočetní náročnost je přitom v obou případech $O(1)$, nezáleží tedy na počtu prvků v seznamu: operace bude stejně efektivní pro seznam obsahující několik položek i stovky milionů položek.

7.

Dokumentové databáze

V této kapitole představíme dokumentové databáze, úložiště o něco složitější než ta, popisovaná v kapitole předchozí. Jak název napovídá, pro tyto systémy je klíčový koncept *dokumentu*, jenž v daném kontextu znamená datovou strukturu, která má samopopisný charakter, tedy obsahuje kromě samotných dat i metadata popisující význam jednotlivých částí datové struktury. Typickým příkladem je samozřejmě formát JSON, jeho binární reprezentace BSON nebo XML. Všechny tyto formáty lze vnímat jako stromové datové struktury obsahující mimo jiné asociativní pole (dvojice název a hodnota), seznamy a základní datové typy. Dokumentové databáze používají dokumenty nejen pro ukládání dat, ale také pro komunikaci s klienty a aplikacemi. V této kapitole budeme používat příklady ze systému MongoDB,¹ což je v současné době jeden z nejpopulárnějších databázových systémů tohoto typu.

7.1 Datový model „dokument“

Jak jsme již několikrát zmínili, NoSQL databáze začaly vznikat zejména jako odpověď na požadavky moderních webových aplikací. V prostředí webu se pro komunikaci mezi jednotlivými komponentami a službami často používají právě formáty

¹ <http://www.mongodb.org>

XML a v posledních letech zejména JSON. Při použití relační databáze v aplikaci typicky dochází k častým konverzím dat, a to jednak při ukládání paměťových datových struktur (tzv. *objektově relační mapování*, ORM) a také při převodu relačních dat na JSON (a zpět) pro komunikaci s dalšími komponentami webové aplikace. Pokud ale použijeme dokumentovou databázi, můžeme v ideálním případě uložená data přímo využít pro komunikaci s ostatními komponentami. Dokumentové formáty také přirozeněji odpovídají struktuře tříd objektového programování, což výrazně zjednodušuje jejich vzájemnou konverzi – v tomto případě se mluví o *objektově dokumentovém mapování* (ODM).

Dalším zásadním důvodem pro použití dokumentových databází je volnost datového modelu. Přestože můžeme ukládat v dokumentové databázi libovolné, zcela různorodé dokumenty, častěji ukládáme společně vždy dokumenty *stejného druhu*. Jejich struktura se může podle potřeby lišit nebo v čase vyvíjet, aniž bychom museli explicitně měnit schéma databáze. V příkladu 7.1 jsou uvedeny dva dokumenty (objekty), které mají některá jména hodnot shodná a v některých se liší, ale oba popisují uživatele.

Příklad 7.1: JSON dokumenty pro uložení v dokumentové databázi

```
{
  "login": "honza",
  "firstname": "Jan",
  "surname": "Novák",
  "address": {
    "city": "Praha",
    "street": "Krásná 5",
    "zip": "111 00"
  }
}

{
  "login": "janicka",
  "firstname": "Jana",
  "surname": "Novotná",
  "web": "http://janicka.novotna.cz/",
  "profile": {
    "colorschema": "green",
    "design": "simple"
  }
}
```

V dokumentové databázi bychom tyto JSON objekty nejspíše uložili ve stejné *kolekci*. Kupříkladu databáze MongoDB neposkytuje žádný způsob, jak omezit nebo vynutit nějaké konkrétní schéma pro dokumenty v kolekci (např. jména hodnot, která by musely mít všechny dokumenty v kolekci). V praxi ale často aplikace komunikují s databází pomocí různých knihoven, jež v kolekcích jisté schéma de facto udržují. Explicitní vynucení konkrétního schématu je běžné např. v nativních XML databázích (viz *XML databáze* na straně 114).

Systém MongoDB umožňuje ukládat libovolné JSON objekty s tím, že každý takový dokument musí obsahovat hodnotu se jménem `_id`, která slouží jako *primární klíč* záznamu v kolekci. Hodnota pod tímto jménem tedy musí být v rámci kolekce

unikátní, neměnná, ale může být jakéhokoli typu jiného než pole a může být vygenerována databází při vkládání dokumentu do kolekce (v příkladech budeme používat pro vygenerovaný klíč zápis <ObjectIdx>). Někdy je existence primárního klíče interpretována tak, že dokumentové databáze jsou vlastně typu klíč-hodnota s tím, že v hodnotě jsou ukládány JSON dokumenty. Např. v MongoDB se ale s `_id` pracuje trochu jiným způsobem, jak uvidíme níže. Vnitřně pak MongoDB ukládá dokumenty ve formátu BSON, což je binární podoba formátu JSON (viz sekce 2.6.3).

Obecně vzato máme při návrhu rozdelení dat do kolekcí dokumentů několik možností, které korespondují s diskuzí o datových modelech v NoSQL světě (viz sekce 5.2). V dokumentových databázích jsou k dispozici dva základní přístupy – použití *vnořených dokumentů* neboli *vnořených objektů* (*embedded documents*) nebo použití *odkazů*. Použití prvního principu je vidět v příkladu 7.2, kde uvedený dokument obsahuje dva vnořené dokumenty *address* a *profile*.

Příklad 7.2: Použití konceptu vnořených dokumentů

```
{
  "_id": <ObjectID1>,
  "login": "honza",
  "name": "Jan Novák",
  "address": {
    "city": "Praha",
    "street": "Krásná 5",
    "zip": "111 00"
  },
  "profile": {
    "colorscheme": "green",
    "design": "simple"
  }
}
```

Hlavní výhodou použití takového datového modelu je možnost manipulace se všemi daty v rámci jedné operace (čtení, zápisu, aktualizace). Tento model je vhodný pouze tehdy, pokud modelujeme vztah 1:1 (tedy pokud nadřazený dokument obsahuje vždy právě jeden vnořený dokument daného typu) nebo vztah 1:N (tedy pokud vnořené dokumenty mají přirozeně právě jeden nadřazený dokument). Obecnou nevýhodou tohoto konceptu je, že pokud do nadřazeného dokumentu vkládáme časem další vnořené záznamy, celková velikost dokumentu může výrazně vzrůst a negativně ovlivňovat rychlosť čtení, zápisu a přenosu dat.² Výběr konkrétního schématu dat může zásadním způsobem ovlivnit efektivitu vyhodnocování dotazů – ty operace, které se budou držet vždy v rámci jednoho dokumentu, budou velmi efektivní.

Druhým možným přístupem je provázání dokumentů pomocí odkazů na jejich `_id` podobným způsobem, jakým bývají využívány cizí klíče v relačních databázích. Příklad 7.3 na následující straně ukazuje použití tohoto konceptu, kde jsme rozdě-

² Systém MongoDB má omezení na maximální velikost dokumentu, a to aktuálně 16MB. Pro uložení větších dokumentů je možné využít rozhraní GridFS API, které umí dokumenty před uložením automaticky rozdělovat. Více viz dokumentace MongoDB (<http://docs.mongodb.org/manual/reference/limits/>).

lili informace o uživateli do tří dokumentů (ve třech kolekcích) s tím, že dva podřízené dokumenty obsahují hodnotu s názvem `user_id`, která nese `_id` nadřazeného dokumentu.

Příklad 7.3: Použití konceptu odkazů

```
kolekce "users"
{
    "_id": <ObjectId1>,
    "login": "honza",
    "name": "Jan Novák"
}

kolekce "addresses"
{
    "_id": <ObjectId2>,
    "user_id": <ObjectId1>,
    "city": "Praha",
    "street": "Krásná 5",
    "zip": "111 00"
}

kolekce "profiles"
{
    "_id": <ObjectId3>,
    "user_id": <ObjectId1>,
    "colorscheme": "green",
    "design": "simple"
}
```

Taková struktura dat odpovídá normalizovanému schématu dat známému z relačních databází. Toto schéma je flexibilnější a používáme ho většinou v situacích, kdy by použití vnořených dokumentů vedlo k duplicitnímu uložení dat, tedy pro modelování vztahů $N:M$. Zjevnou nevýhodou je nutnost získávání provázaných záznamů pomocí několika operací, vzhledem k tomu, že dokumentové databáze zřídka umožňují propojování několika záznamů v rámci jednoho dotazu. Současné systémy tohoto typu také většinou neposkytují podporu pro automatickou kontrolu existence referencovaného primárního klíče.

Při návrhu struktury databáze můžeme cíleně udělat rozhodnutí, že pro uložení nějaké informace zvolíme přístup vnořených dokumentů, i když pak bude docházet k duplicitnímu uložení dat. Jako příklad vezměme databázi redakčního systému, ve které ukládáme zejména publikované články. Každý článek patří do jedné či několika kategorií a kategorie tvoří hierarchii (nad- a podkategorie). Informaci o názvech kategorií daného článku můžeme držet přímo v dokumentu s obsahem článku (protože tyto informace budeme potřebovat vždy najednou), i když každý název kategorie pak bude v celé databázi mnohokrát. Dále k článkům potřebujeme přidružit komentáře od čtenářů. Opět se nabízí možnost držet i tyto informace společně s obsahem článku. Musíme vzít ale v úvahu, že komentářů může být hodně a budou často doplňovány – dokument by tedy mohl značně „nabobtnat“ a systém by také musel řešit častý souběh operací zápisu v jednom dokumentu.

Denormalizace a povaha dat

Častou námitkou proti ukládání přidružených informací do nadřazeného dokumentu (tedy denormalizovaně) je obtížnost nebo nemožnost jejich změny. Uložíme-li název kategorie přímo do dokumentu, musíme po přejmenování kategorie provést úpravy třeba milionů či miliard dokumentů.

To může být skutečně nepraktické či přímo neproveditelné. Na druhé straně však stojí otázka, jak často a zda vůbec dochází v daném systému k přejmenování kategorie – v mnoha případech může být cena za jednorázové provedení úprav velkého množství dokumentů stále přijatelná, a to zejména vzhledem k tomu, že denormalizovaný datový model je vždy „levnější“ při přístupu k datům.

Správné rozhodnutí tudíž musí vycházet z povahy dotazů a dat, s nimiž pracujeme. Odlišné vlastnosti budou mít např. atributy *místo narození* a *místo pobytu*. V prvním případě může ke změně atributu dojít pouze při opravě zjevné chyby (do systému byla vložena nesprávná data), tyto incidenty budou občasné a izolované. Ve druhém případě pochopitelně ke změně atributu může docházet relativně často.

Zdánlivě se tedy atribut typu *místo pobytu* nehodí ukládat přímo do nadřazeného dokumentu. Správné řešení však opět vychází z povahy dat. Uvažujme situaci, kdy provozujeme rozsáhlý archiv otázek a odpovědí jako např. Stack Overflow,³ kde každá otázka a odpověď má svého autora, který má své bydliště. Uvažujme, že aplikace umožňuje uživateli pokládat dotazy typu „Seznam měst, jejichž obyvatel položili tento měsíc nejvíce otázek“. Přestěhuje-li se uživatel z Barcelony do Berlína, měli bychom adekvátně upravit dokumenty s jeho otázkami? Odpověď zde nemůže být striktně technická, ale musí vycházet z našeho pojetí světa: je otázka položená uživatelem žijícím v Barceloně stále z Barcelony, i když se daný uživatel přestěhuje? Ve většině podobných systémů bude odpověď znít „ano“ a denormalizovaný datový model tak v tomto případě odolá i své nejvážnější námitce. (Viz též *Svět nepodléhá třetí normální formě* na straně 92.)

Datový model JSON dokumentu se používá v již zmíněném MongoDB i v dalších populárních open source systémech CouchDB,⁴ RavenDB⁵ nebo Elasticsearch, dále v databázích poskytujících několik různých modelů jako OrientDB⁶ nebo PostgreSQL (viz sekce 14.1.1) a také v komerčních cloudových databázích Amazon DynamoDB,⁷ Google Cloud Datastore⁸ nebo Microsoft Azure DocumentDB.⁹ Tento výčet nemá ambici být kompletním – aktuální seznam nejpopulárnějších dokumentových databází je k nalezení např. na stránkách DB-engines.com.¹⁰

³ <http://stackoverflow.com>

⁴ <http://couchdb.apache.org>

⁵ <http://ravendb.net>

⁶ <http://www.orientechnologies.com/orientdb/>

⁷ <http://aws.amazon.com/dynamodb/>

⁸ <https://cloud.google.com/datastore/docs>

⁹ <http://azure.microsoft.com/en-us/services/documentdb/>

¹⁰ <http://db-engines.com/en/ranking/document+store>

Další samostatnou skupinou systémů jsou nativní XML databáze, které začaly vznikat již okolo roku 2000 a jsou tak bezpochyby vůbec nejstaršími dokumentovými databázemi. O těchto databázích a XML technologiích již ale obecně existuje řada publikací [119] [130], ve zbytku této kapitoly se tedy budeme věnovat později vzniklým a méně známým systémům, které staví na formátu JSON.

XML databáze

Pro úplnost ale alespoň stručně představíme typické základní vlastnosti XML databází. Většina XML databází organizuje XML dokumenty do kolekcí, podobně jako dokumentové NoSQL databáze. Zde však podobnost obvykle končí. XML databáze umožňují pro všechny dokumenty v kolekci určit XML schéma, kterému mají vyhovovat. Lze tak automaticky zajistit kontrolu integrity dat v databázi. Přítomnost schématu také umožňuje lepší optimalizaci dotazů a tvorby indexů. Použití schématu je však nepovinné.

Zatímco dokumentové NoSQL databáze mají obvykle omezené možnosti dotazování, všechny XML databáze implementují pokročilý dotazovací jazyk XQuery, o kterém si více povíme v sekci 11.4.3. XQuery je pro XML databáze tím, čím je SQL pro relační databáze – standardizovaným jazykem, jehož znalosti využijeme v různých databázích. Kromě výběru dat můžeme pomocí XQuery Update [57] provádět i modifikaci dat. Není nutné pracovat s celými dokumenty najednou, lze modifikovat nebo vybírat jen části dokumentů uložených v kolekci.

Pro efektivní využití výhodnocování dotazů lze samozřejmě používat indexy. Většina XML databází zcela automaticky vytváří několik základních indexů, které dovolují rychlé hledání elementů a atributů na základě jejich jména, hodnoty a pozice ve stromu XML. Kromě toho pak lze definovat vlastní indexy, pokud chceme optimalizovat konkrétní typy dotazů. Typickým nasazením XML databází je ukládání dokumentů se strukturovanými texty v přirozeném jazyce – texty zákonů, smlouvy, příbalové letáky léků, patenty – většina XML databází proto nabízí i fulltextové vyhledávání, které je dostupné přímo z rozšíření dotazovacího jazyka XQuery Full Text [56].

XML databáze běžně nabízejí transakční zpracování a REST API. Kromě toho jsou dostupná specializovaná API pro jednotlivé programovací jazyky. Pro Java existuje dokonce standardizované API XQJ [131], které je univerzální pro všechny XML databáze (podobně jako např. JDBC pro relační databáze).

Jedinou oblastí, kde XML databáze za těmi NoSQL pokulhávají, je škálovatelnost a dostupnost. Zatímco prostá replikace je dostupná ve většině XML databází, možnost rozdělení celé databáze na několik uzlů a distribuované provádění dotazů je dostupné jen v některých komerčních produktech, jako je MarkLogic.¹¹

Mezi nejznámější open source XML databáze patří BaseX¹² a eXistdb.¹³

¹¹ <http://marklogic.com>

¹² <http://baseinx.org>

¹³ <http://exist-db.org>

7.2 Dotazování a manipulace s daty

Obecně vzato v dokumentových databázích neexistují zavedené standardy pro zadávání dotazů a manipulaci s daty. Prakticky každý systém vyvinul vlastní jazyk a vlastní programátorské API. V porovnání se světem SQL databází toto samozřejmě znamená nutnost u každého systému vstřebat trochu jinou terminologii a jinou syntaxi a také obtížnější převod aplikace z jednoho systému na druhý. Hlavním důvodem této rozdílnosti je rychlý rozvoj jednotlivých databází, přičemž každá z nich má unikátní vlastnosti a možnosti, které vyžadují vlastní způsoby ovládání a přístupu k datům. Výjimkou z tohoto tvrzení jsou do značné míry právě výše zmíněné nativní XML databáze.

7.2.1 Dotazy

Se systémem MongoDB můžeme pracovat pomocí jazyka, jehož syntaxe vychází z konvence formátu JSON. Jeho příkazy můžeme zadávat např. pomocí ovládací konzole `mongo`. Začneme příkladem dotazu, který vyhledá v kolekci uživatelů ty, u nichž evidujeme věk, a ten je větší než 33.

```
db.users.find( { age: { $gt: 33 } } )
```

První část dotazu vždy specifikuje dotazovanou kolekci, prostřední část dotazovací kritéria a poslední volitelná část modifikátory aplikované na odpověď. Tento příkaz v aktuální databázi (`db`) a kolekci s názvem `users` naleze a vrátí všechny JSON dokumenty, jež mají na první úrovni hodnotu s názvem `age`, která je větší (operátor `$gt`) než 33. Odpověď na tento dotaz je databázový kurzor (iterátor) vracející postupně všechny dokumenty splňující podmínu dotazu. Chceme-li, aby byly výsledky seřazeny podle věku, použijeme následující konstrukci:

```
db.users.find( { age: { $gt: 33 } } ).sort( { age: 1 } )
```

Ve výrazu `sort({ age: 1 })` vidíme použití hodnot 0 nebo 1 ve významu nepravda, resp. pravda. V tomto případě výraz znamená, že výsledek bude řazen *vzestupně* a `age: 0` by znamenal řazení sestupně. Podívejme se na příklad složitějšího dotazu, který vrátí jména všech uživatelů z Prahy nebo z Brna, přičemž předpokládáme datový model vnořených dokumentů z příkladu 7.2 na straně 111:

```
db.users.find( { 'address.city': { $in: [ 'Praha', 'Brno' ] } }, { name: 1 } )
```

Oproti předchozímu dotazu zde vidíme zápis omezujícího kritéria s tečkovou notací (`address.city`), která znamená hodnotu s názvem `city` ve vnořeném dokumentu uloženém pod názvem `address`. Tečková notace se často používá pro navigaci v JSON dokumentech. Ve výše uvedeném dotazu dále vidíme operátor `$in` umožňující zadat více zájmových hodnot pro jeden název. Výraz `{ name: 1 }` na konci dotazu je zápis operace projekce, tedy vrácení pouze vybraných hodnot z vyhovujících záznamů (v tomto případě jde o jména uživatelů a primární klíč `_id`, který se vrací implicitně). Výsledek dotazu by tedy mohl vypadat např. takto:

```
{ "_id" : ObjectId("54d22c14476a64b06ae075db"), "name" : "Jan Novák" }
{ "_id" : ObjectId("54d22c1e476a64b06ae075dc"), "name" : "Jana Novotná" }
```

V následujícím příkladu ukážeme, jak bychom stejné zadání vyřešili v případě použití několika kolekcí propojených pomocí referencí. Předpokládáme schéma z příkladu 7.3 na straně 112, kde jsou data uložena ve třech kolekcích. Omezující podmínu na město Praha nebo Brno je v tomto případě potřeba použít v kolekci `addresses` a jména příslušných uživatelů jsou v propojené kolekci `users`. Následuje krátký program v jazyce Javascript, který můžeme pomocí ovládací konzole `mongo` spustit na serverové straně systému MongoDB:

```
cursor = db.addresses.find( { 'city': { $in: [ 'Praha', 'Brno' ] } }, { user_id: 1 } )
while ( cursor.hasNext() ) {
    user_id = cursor.next().user_id
    user = db.users.findOne( { _id: user_id }, { name: 1, _id: 0 } )
    printjson( user )
}
```

Uvedený skript nejdříve vyhledá adresy vyhovující zadané podmínce. Z každého dokumentu ve výsledku tohoto dotazu se přečte pole `user_id` a v kolekci `users` se nalezně jméno uživatele, jehož primární klíč `_id` odpovídá dané hodnotě `user_id`. Všimněte si, že u příkazu vyhledávání v kolekci uživatelů jsme uvedli `_id: 0`, což znamená, že na výstupu nebude hodnota primárního klíče. Výsledkem tedy bude:

```
{ "name" : "Jan Novák" }
{ "name" : "Jana Novotná" }
```

Dotazovací aparát systému MongoDB je poměrně bohatý a naším cílem bylo představit jeho základní principy, které jsou obecně platné i pro další dokumentové databáze.

7.2.2 Modifikace databáze

Dále se podíváme, jak modifikovat obsah databáze MongoDB. Následující příkaz vloží nový JSON dokument do kolekce `users`.

```
db.users.insert( { "login": "honza", "name": "Jan Novák", "address": {
    "city": "Brno-město", "street": "Slunná 5", "zip": "603 00" } } )
```

V tomto případě MongoDB vygeneruje primární klíč `_id`, protože hodnota s tímto názvem není obsažena ve vkládaném záznamu. Jako další uvádíme příkaz pro hromadnou aktualizaci záznamů, které splňují daná kritéria:

```
db.users.update( { 'address.city': 'Brno-město' },
    { $set: { 'address.zip': '602 00' } },
    { multi: 1 } )
```

Tento příkaz `update` změní dokumenty splňující podmínu v prvním argumentu (`'address.city': 'Brno-město'`) a všem těmto dokumentům nastaví hodnotu s názvem `address.zip` na správné PSČ Brna-města, tedy `602 00`. Poslední modifikátor říká, že se daná úprava má opravdu provést na všech vyhovujících záznamech a ne jen na prvním z nich. Uvedený příkaz nastaví hodnotu i v případě, že před tím dokument hodnotu s názvem `address.zip` neobsahoval.

7.2.3 Agregované dotazy a MapReduce

Stejně jako většina dokumentových databází i systém MongoDB umožnuje vyhodnocovat složitější agregované dotazy pomocí principu MapReduce (viz kapitola 4). Použití tohoto principu předvedeme na příkladu kolekce `accesses`, ve které jsou zaznamenány přístupy uživatelů do systému. V každém záznamu (dokumentu) je vždy uložen identifikátor uživatele, který do systému vstoupil, časové razítka vstupu, odchodu a typ přístupu:

```
kolekce "accesses":  
{  
    "user_id": <ObjectId>,  
    "login_timestamp": <čas_vstupu_do_systému>,  
    "logout_timestamp": <čas_odchodu_ze_systému>,  
    "access_type": <typ_vstupu_do_systému>  
}
```

Ukážeme si, jak lze pomocí principu MapReduce z této kolekce získat souhrnnou zprávu o tom, kolik času jednotliví uživatelé strávili přihlášení do systému, ale pouze pokud se jednalo o standardní typ přístupu (`regular`):

```
db.accesses.mapReduce(  
    function() { emit (this.user_id, this.logout_timestamp - this.login_timestamp); },  
    function(key, values) { return Array.sum( values ); },  
    {  
        query: { access_type: "regular" },  
        out: "access_times"  
    }  
)
```

V tomto příkazu je první z uvedených funkcí (*Map*) spuštěna na každém dokumentu z kolekce `accesses`, který splňuje podmítku `query` (tedy `access_type: "regular"`). Každé toto spuštění vygeneruje dvojici (`key`, `value`), kde klíčem je identifikátor uživatele `user_id` a hodnotou je doba, kterou v rámci tohoto přístupu strávil v systému. Další uvedená funkce (*Reduce*) dostane na vstup vždy identifikátor uživatele (`key`) a pole hodnot vygenerovaných v předchozí fázi pro daného uživatele (`values`). Tyto hodnoty se sečtou a uloží do výstupní tabulky (`access_times`).

Pokud je kolekce distribuovaná (viz sekce 7.3.3), je tento příkaz paralelně zpracován na všech příslušných uzlech. V aktuální verzi MongoDB by bylo možné toto zadání vyřešit i pomocí příkazu `db.collection.aggregate`, který je efektivnější a má jednodušší a čitelnější zápis než výše uvedený příkaz `mapReduce`. Tento agregační systém je ale specifický pro systém MongoDB a naším cílem bylo zejména přiblížit použití principu MapReduce, který se často využívá v dokumentových i jiných NoSQL databázích.

7.2.4 Shrnutí

Neklademe si za cíl plně pokrýt možnosti a syntaxi jazyka systému MongoDB, ale pomocí příkladů poodhalit možnosti manipulace s dokumenty poskytované doku-

mentovými databázemi. Dotazovací jazyky těchto systémů umožňují efektivně vyhodnocovat všechny běžně používané operace, ale pouze na jedné kolekci. Jak jsme již uvedli, propojování záznamů v rámci vyhodnocování jednoho dotazu většinou nebyvá u dokumentových databází možné a aplikace se musí na propojené záznamy dodatečně „doptat“. JSON databáze přirozeně umožňují pracovat s JSON formátem v souladu s jeho definicí, tedy zejména navigovat na jednotlivé hodnoty v dokumentu, včetně práce s vnořenými dokumenty a seznamy hodnot.

Jak jsme uvedli výše, jednou z výhod dokumentových databází je relativně přímočaré mapování paměťových struktur objektového programování na dokumentové formáty, zejména JSON (objektově dokumentové mapování, ODM). Pro přístup k dokumentovým databázím se proto velmi často používají různé ODM knihovny, které aplikaci odstíní od konkrétního jazyka pro manipulaci s daty. ODM knihovny umožní automatickou serializaci objektu do formátu JSON a jeho transparentní „uložení“ do databáze. Tyto knihovny se mohou postarat i o případné rozdělování dat do různých kolekcí a jejich zpětné skládání při načítání objektu z databáze.

7.3 Vlastnosti dokumentových databází

V této sekci se detailněji podíváme na technologické vlastnosti dokumentových databází a na způsoby, jak jsou tyto vlastnosti realizovány.

7.3.1 Indexy

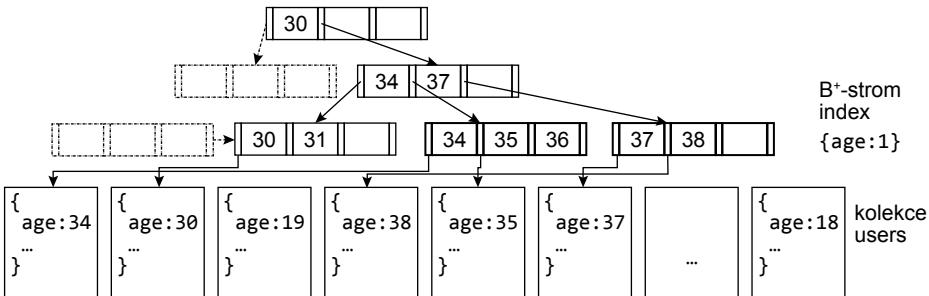
V kapitole 6 jsme zmiňovali, že některá úložiště typu klíč-hodnota umožňují vytvoření sekundárních indexů na uložených hodnotách. Pro dokumentové databáze je vytváření a používání vyhledávacích indexů klíčovou vlastností, bez které by takové systémy nebyly o moc více než množinou souborů obsahujících kolekce dokumentů. Stejně jako v relačních databázích, i zde indexy slouží pro efektivní vyhodnocování dotazů, protože pokud pro výběrové kritérium v daném dotazu neexistuje index, musí databáze projít celou kolekci sekvenčně.

V systému MongoDB jsou standardní vyhledávací indexy implementovány pomocí B⁺-stromů.¹⁴ Na obrázku 7.1 na následující straně vidíme naznačený index na hodnotách s názvem `age`.

Vyhledávací index tohoto typu nám umožní a usnadní následující operace:

- vyhodnocování klasických dotazů, které vyhledávají záznamy s konkrétní hodnotou v indexovaném atributu: `db.users.find({ age: 28 })`,

¹⁴ B-stromy resp. B⁺-stromy organizují množinu klíčů z kolekce dokumentů do hierarchické (stromové) indexační struktury tak, že je možné s logaritmickou složitostí nalézt záznamy vyhovující danému vyhledávacímu klíči. B⁺-stromy také umožňují efektivní nalezení všech dokumentů, jejichž klíč je v intervalu vyhledávacích klíčů (tzv. *rozsahový dotaz*).



Obrázek 7.1: Příklad použití indexu na hodnotách s názvem `age` z kolekce `users` a vyhodnocení dotazu `db.users.find({ age: { $gt: 33 } })`

- efektivní vyhodnocování rozsahových (intervalových) dotazů: `db.users.find({ age: { $gt: 33 } })` (vyhodnocení tohoto dotazu je naznačeno na obrázku 7.1 pomocí silnějšího orámování relevantních uzel indexu),
- pokud má být výsledek dotazu seřazen podle indexovaného atributu, tak se index použije také pro tento seřazený výpis: `db.users.find({ age: { $gt: 33 } }).sort({ age: 1 })`,
- pokud dotaz vrací pouze hodnoty z indexovaného atributu (operace projekce), tak ho databáze vyhodnotí přímo z hodnot v indexu – bez přístupu do samotných dat v kolekci: `db.users.find({ age: { $gt: 33 } }, { age: 1, _id: 0 })`.

V systému MongoDB je na každé kolekci automaticky vybudovaný index na hodnotách `_id`. Dále je možné vytvořit indexy na jakékoli jiné vybrané hodnotě (i vnořené), na kombinaci hodnot, na seznamu hodnot (resp. automaticky na každé hodnotě v seznamu). Místo B^+ -stromu je možné zvolit hašovací index, ale také textové vyhledávací indexy nebo prostorový index pro geografické dotazy.

Obecně je vhodné mít index správného typu na každém atributu, nad kterým budeme chtít v databázi vyhledávat, aby nedocházelo k sekvenčním průchodům celé kolekce. Každý index s sebou ale nese jisté náklady: zabírá místo na disku i v paměti a je ho potřeba aktualizovat společně s příslušnými daty v kolekci. Databázový systém MongoDB umožňuje sledovat efektivitu vyhodnocování dotazů a automaticky do logu hlásí dotazy, které nemohly využít žádný existující index.

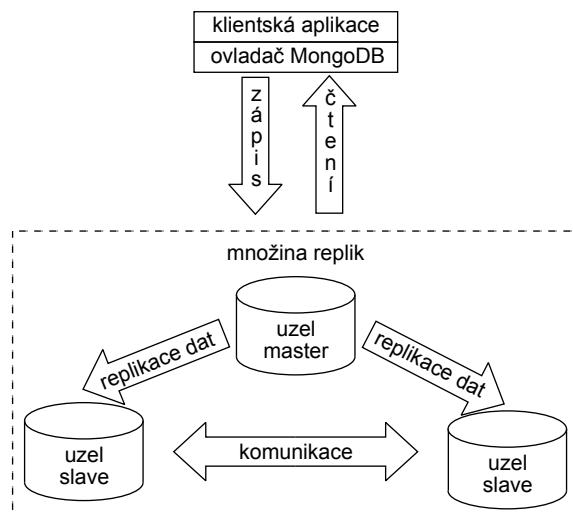
7.3.2 Replikace dat a dostupnost systému

Dokumentové databáze umožňují aplikovat principy rozdělení dat i replikace tak, jak je obecně popsáno v sekci 3.3. Zaměřme se nejprve na replikaci v systému MongoDB, která používá princip master-slave. V tomto systému se používá termín *množina replik (replica set)*, což je několik instancí MongoDB, které kooperují a obsahují stejná data. Jedna z těchto instancí je určena jako master a pouze tato instance může provádět operace zápisu. Jak víme ze sekce 3.3.2, tento princip:

- zvyšuje propustnost operací čtení (počet obslužených operací), protože čtení může být realizováno ze všech uzel v množině replik,

- zvyšuje dostupnost systému a dat při krátkodobém výpadku některého z uzlů, protože klienti mohou dále číst a případně i zapisovat data přes ostatní uzly,
- zvyšuje celkovou odolnost systému vůči ztrátě dat – i při fatálním výpadku některého z uzlů systém o data nepřijde.

Realizace v MongoDB je schematicky naznačena na obrázku 7.2. Nejprve si všimněme, že klientské aplikace nekomunikují přímo s jedním konkrétním uzlem, ale v jistém smyslu s celou množinou replik. Aplikace se sice nejprve připojí k jednomu uzlu, jehož adresu zná, ale ovladač (*driver*) poskytovaný MongoDB může požadovanou operaci přesměrovat (např. operaci zápisu vždy přesměruje na uzel master).



Obrázek 7.2: Princip master-slave replikace v systému MongoDB

Téměř učebnicově zajímavý je způsob, jakým MongoDB řeší vyvážení vlastnosti CAP (viz sekce 3.2.2). Jak již víme, master-slave replikace zvyšuje dostupnost systému jen do jisté míry. Pokud dojde k výpadku uzlu master, tak ostatní uzly mezi sebou zvolí nový uzel master, který bude dále realizovat operace zápisu. Představme si ale, že dojde jen k výpadku sítě mezi uzlem master a ostatními uzly (tedy k rozdělení systému tak, jak ho popisuje CAP teorém). MongoDB tuto situaci řeší tím, že ve chvíli, kdy master zjistí, že s ostatními nemůže navázat spojení a je „sám“, tak automaticky přestane být uzlem typu master a bude dále obsluhovat pouze operace čtení. Naopak zbylé uzly (řekněme dva) spolu dále komunikují, a protože jich je více než polovina původního počtu, tak mezi sebou zvolí nový master. V termínech CAP teorému můžeme říci, že systém replikace v MongoDB preferuje konzistence dat před dostupností systému (např. pokud vypadnou dva uzly ze tří, tak na zbylý uzel není možné zapisovat). Zachování konzistence je nadřazeno i odolnosti vůči rozpadu systému – pokud se systém rozpadne, tak je možné zapisovat pouze do té části, která obsahuje většinu uzlů.

Využití topologie clusteru v klientských knihovnách

Uvedené sofistikované směrování požadavků si může systém MongoDB dovolit díky specializovanému ovladači, který využívá knihovny v konkrétních programovacích jazycích. Systémy, jež pro komunikaci s databází používají standardní HTTP protokol a rozhraní typu REST, jako např. Elasticsearch, tuto možnost pochopitelně nemají – klient nezná topologii clusteru, a požadavek proto směruje vždy na konkrétní uzel, který zajistí jeho případné přeposlání na ostatní uzly. V tomto případě může docházet k přetížení konkrétního uzlu, na který klientská aplikace posílá požadavky, a může tak dojít k nestabilitě celého systému. Jedním z řešení tohoto problému je předřadit databázi HTTP proxy (např. Nginx¹⁵ nebo HAProxy¹⁶), propojit ji s několika uzly clusteru a směrovat požadavky střídavě na každý z nich (tzv. *round-robin*). Klient se pak připojuje nikoliv přímo k databázi, ale k proxy, která zajišťuje přeposílání požadavků a odpovědí. Tento přístup je přitom umožněn díky hlavním principům HTTP protokolu: bezstavovosti, textové reprezentace dat a důležitosti URI, které jsou často považované za nedostatky tohoto protokolu.

Stejně jako MongoDB, tak i třeba Elasticsearch řešení tohoto problému posunuje přímo do klientských knihoven, které obsahují funkcionality pro získání topologie clusteru a nativně podporují střídavé posílání požadavků na jednotlivé uzly, stejně tak jako aktualizaci topologie nebo opakování požadavku, který skončil chybou, na jiném uzlu.

Jak je tedy vidět, je vhodné, aby počet uzlů v množině replik byl lichý. Pokud při konfiguraci systému přirozeně vychází počet databázových serverů jako sudý, umožňuje MongoDB přidat tzv. *arbitra*, tedy uzel, který nenese žádná data, ale účastní se volby nového uzlu typu master v případě výpadku. Vlastní replikace dat je v systému MongoDB realizována pomocí přeposílání *operačního logu* (*oplog*) z uzlu typu master na ostatní.¹⁷ Systém také umožňuje uzel typu slave nastavit jako *zpožděnou kopii*. Změny z logu se pak na tento uzel aplikují až po uplynutí daného časového intervalu. Mít automaticky k dispozici časově zpožděnou zálohu databáze se může hodit např. pro obnovu dat poškozených chybou v aplikaci.

Samotné nastavení replikace je v MongoDB poměrně jednoduché. Pokud např. chceme za běhu změnit jednoserverové nastavení na množinu replik se třemi uzly, můžeme to udělat např. touto sekvencí příkazů v administrátorské konzoli:

```
rs.initiate()
rs.add("10.0.1.2:27017")
rs.add({ "host": "10.0.1.3:27017", "hidden": 1, "priority": 0, "slaveDelay": 3600 })
```

Předpokládejme, že na uzlech 10.0.1.2 a 10.0.1.3 běží další dvě instance MongoDB, obě na implicitním portu 27017. První příkaz z této sekvence inicializuje na aktuálním uzlu množinu replik (rs znamená *replica set*) obsahující pouze aktuální uzel

¹⁵ <http://nginx.org/en/>

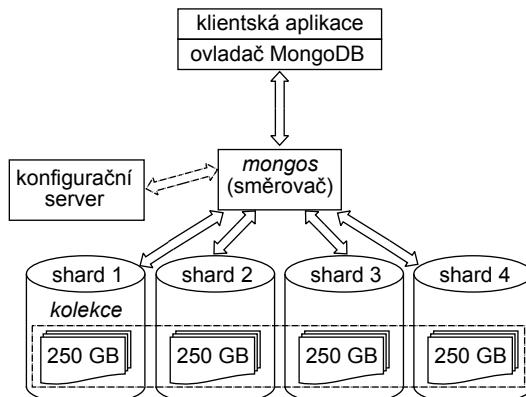
¹⁶ <http://www.haproxy.org>

¹⁷ Do operačního logu (též *write-ahead log*, *WAL* nebo *journal*) jsou zaznamenávány všechny operace zápisu, a to dříve, než jsou promítnuty do vlastního úložiště (více viz např. sekce 8.3.2).

(jako master). Následující příkaz do této množiny přidá druhý uzel, který se tím pádem stane uzlem typu slave. V tu chvíli je na tento uzel překopírován aktuální obsah databáze. Třetí příkaz přidá do této množiny uzel, který je *skrytý*, a proto se z něj nikdy nebude číst ("`hidden`": 1), nemůže být zvolen jako master ("`priority`": 0) a bude udržovat o hodinu zpožděnou kopii dat ("`slaveDelay`": 3600).

7.3.3 Rozdělení dat

Dokumentové databáze obecně umožňují také rozdělení dat na více uzlů. Samotné dělení kolekce na části probíhá podobně jako u distribuovaných systémů typu klíč-hodnota. V případě MongoDB můžeme data rozdělit podle hodnot `_id` nebo jakéhokoli jiného klíče (*sharding key*), který je v rámci kolekce unikátní, neměnný a je na něm postavený index. MongoDB umožňuje rozdělit data z kolekce na jednotlivé uzly buď pomocí rozdělení domény tohoto klíče na intervaly, nebo pomocí hašovací funkce. Schéma dělení kolekce je zobrazeno na obrázku 7.3.



Obrázek 7.3: Princip rozdělení dat v MongoDB

Na tomto obrázku vidíme naznačenou velkou kolekci, kterou rozdělíme na čtyři menší části (*shard 1* až *shard 4*). MongoDB pak aplikaci umožňuje komunikovat s distribuovaným systémem pomocí *směrovače* (procesu s názvem `mongos`), který dostává informace z konfiguračního serveru o tom, kde jsou jaká data, a který směruje dotazy na příslušné uzly. Každá jednotlivá část nemusí být tvořena jen jedním uzlem, ale může být tvořena množinou replik. Takovou kombinaci dělení dat a replikace můžeme vidět v NoSQL databázích často (viz princip popsány v sekci 3.3.4). Pokud si to přejeme, můžeme MongoDB na pozadí provádět vyrovnaní velikosti dat uložených na jednotlivých uzlech (*balancing*); tento proces může provádět dělení a migraci kusů dat tak, aby objemy dat na jednotlivých uzlech byly co nejvyrovnanější.

Standardní operace typu CRUD (*create, read, update, delete*) jsou směrovačem přeposlány na příslušnou část kolekce, případně na několik částí podle toho, kterých dokumentů se dotaz týká, a také podle toho, jak dobře může směrovač zjistit, ve kterých částech jsou relevantní dokumenty uloženy. V rámci jednotlivých částí je dotaz zpracován stejným způsobem jako na samostatné kolekci, částečně odpovědi jsou spojeny a vráceny klientské aplikaci.

7.3.4 ACID pro jednotlivé operace a transakce

Dále se podíváme na to, jak dokumentové databáze zajišťují vlastnosti ACID (viz sekce 3.2). V prostředí distribuovaných NoSQL databází musíme v první řadě řešit otázku, jak jsou tyto vlastnosti zajištěny pro *jednotlivé operace* na replikovaných datech – nikoli pro transakce, jak jsme zvyklí z centralizovaných databází. V sekci 7.3.2 jsme popsali princip master-slave replikace aplikovaný v systému MongoDB. Díky tomu, že v systému je vždy pouze jeden uzel master, je a priori zabráněno vzniku write-write konfliktu, tedy nemůže dojít k souběžné modifikaci dat na dvou uzlech z množiny replik. Jak jsme také uvedli, v tomto smyslu preferuje MongoDB konzistenci dat před dostupností a tolerancí systému k rozpadu.

Další otázkou ale je potenciální read-write konflikt, tedy hrozba toho, že klient bude číst zastaralá (*stale*) data z uzlu typu slave, zatímco na uzlu master již byla data aktualizována (jen se tato informace ještě nestihla rozšířit na uzly typu slave). Pro zajištění (resp. regulaci) tohoto typu konzistence používá MongoDB přístup vycházející z principu kvór (viz sekce 3.3.3). Při zápisu si může aplikace zvolit, jestli má systém nahlásit úspěšné ukončení operace hned po zápisu na master, nebo až po rozšíření změny na (volitelný počet) dalších uzlů typu slave. Tento parametr můžeme nastavit globálně, nebo takto pro jednotlivý příkaz:

```
db.users.insert(
  { "login": "honza", "name": "Jan Novák" },
  { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

Uvedený příkaz vloží nový záznam s tím, že se tato aktualizace musí rozšířit alespoň na dva (*w: 2*) uzly, tedy master a jeden slave. Operace zápisu je blokující, dokud se to nepodaří, ale nejdéle pět sekund (5000 ms). Při čtení můžeme naopak nastavit, jestli se má operace realizovat pouze na uzlu master, nebo jestli se může číst i z ostatních uzlů, které mohou teoreticky obsahovat neaktuální data (v závislosti na nastavení *writeConcern*). Mód pro čtení můžeme nastavit buď globálně, pro spojení, nebo pro konkrétní operaci:

```
db.getMongo().setReadPref('primaryPreferred')

db.users.find({ login: "honza" }).readPref(
  { mode: 'nearest', tags: [ { 'location': 'Europe' } ] }
)
```

První z uvedených příkazů nastaví globálně, že preferujeme čtení z uzlu master, ale čtení z ostatních uzlů je také povoleno. Druhý příkaz zařídí, že se operace čtení provede z „nejbližšího“ uzlu, což je vyhodnoceno jednak podle informace z tagu *'location': 'Europe'* a jednak pomocí měření doby odezvy jednotlivých uzlů z klientské aplikace.

MongoDB umožnuje nastavit různé chování operací zápisu a čtení i lokálně v rámci jednoho databázového serveru. Při zápisu můžeme (opět buď globálně, nebo pro jednotlivé operace) nastavit již známý parametr *writeConcern* takto:

- **w=0**: operace zápisu je ukončena hned, bez jakékoli kontroly, jestli změnu systém opravdu provedl,

- **w=1**: operace zápisu je ukončena po úspěšném provedení změny do paměťového prostoru databázového systému,
- **w=1, j=true**: operace je ukončena až po úspěšném zápisu změny do diskového operačního logu (j jako *journal*).

Použití těchto nastavení je tedy typickým příkladem rozvolnění vlastnosti trvalosti změn ve prospěch rychlosti operací zápisu.

Další důležitou vlastností z ACID je izolovanost transakcí nebo alespoň jednotlivých operací. V MongoDB operace čtení vracejí nejnovější hodnoty zapsané jen do paměťového prostoru systému. Toto chování lze označit za nejnižší úroveň izolovanosti operací (read uncommitted – viz sekce 3.2.1.1), protože operace čtení může vrátit novou hodnotu dokonce ještě před tím, než je dokončena operace zápisu, která změnu provedla (pokud je nastavený parameter `writeConcern` na úrovni **w=1, j=true**). Na druhou stranu, protože jednotlivé operace neprovádí žádné explicitní ukončení „transakce“ pomocí příkazu `commit`, tak se nemůže stát, že by čtecí operace mohla dostat nevalidní data, která by byla později vrácena pomocí `rollback`.

Operace zápisu jsou vždy *atomické* na úrovni jednotlivých dokumentů. Pokud operace provádí hromadnou změnu několika dokumentů, tak jsou operace opět atomické pro jednotlivé dokumenty, ale ne jako celek. Toto chování je možné změnit pomocí parametru `$isolated`, který zaručí, že dokud celá operace hromadné změny nebude dokončena, tak se na modifikovaných dokumentech žádná jiná operace neprovede. Tento parametr ale není funkční globálně na rozdělených kolekcích.

Systém MongoDB zavádí také možnost jistého transakčního zpracování, které umožňuje celou sekvenci operací vrátit (`rollback`), pokud je jedna z operací neúspěšná. Toto je v MongoDB implementováno pomocí algoritmu dvoufázového COMMIT protokolu (viz sekce 12.3.1), což ale může vést k velkému zpomalení celého zpracování.

7.4 Závěr

V celé kapitole jsme se drželi zejména příkladu databáze MongoDB a popisovali její vlastnosti a funkce dostupné v době psaní tohoto textu (jaro 2015). Cílem bylo představit principiální výzvy, se kterými se dokumentové databáze setkávají, vždy jedno vybrané řešení daného problému a vlastnosti tohoto řešení. Věříme, že kapitola čtenáři pomůže rychle se zorientovat i v dalších dokumentových systémech jako RavenDB, CouchDB nebo OrientDB, které se liší v konkrétních řešeních, ale základní principy jsou vždy podobné.

Polyglotní persistence

Dokumentové databáze jsou z pohledu datového modelování vhodné všude tam, kde pracujeme s entitami, které mají přímou vazbu na reálné dokumenty, jako např. články, faktury, kontakty, údaje o zásilkách apod. Ve všech těchto případech využijeme flexibilitu a dynamičnost datového schématu, stejně jako přímou korespondenci mezi datovým modelem a skutečností. Při vytváření datového modelu se tak můžeme nechat vést reálným světem a jeho zákonitostmi, spíše než teoretickými pravidly návrhu jako v případě relačních databází. Je přitom zjevné, že pro některé entity jsou dokumentové databáze nevhodné, kupříkladu pro ukládání vztahů mezi entitami, pro něž jsou vhodné specializované grafové databáze (viz kapitola 9). Jak jsme ale viděli v sekci 6.3.2, netriviální funkcionality můžeme dosáhnout i v relativně jednoduché databázi jako Redis. Volba vhodné databáze tak bude dnes vždy záviset na primárních požadavcích aplikace. Pro určitou aplikaci může být kupříkladu kritická propustnost zápisu, protože čtení dat bude probíhat ze zcela jiné databáze, která je plněna na pozadí. Projinou bude naopak kritické, aby byla data dostupná pro čtení okamžitě po zápisu, nebo redundantní uložení dat v geograficky vzdálených datových centrech.

Z tohoto důvodu je třeba se vzdát myšlenky, že soudobé aplikace zpracovávající rozsáhlé objemy dat si vystačí s jedním typem databáze, nebo dokonce s jednou konkrétní databází; Martin Fowler pro tento fakt zavedl termín *Polyglotní persistence* [85]: „Komplexní aplikace využívá různé druhy dat a již nyní zpravidla integruje informace z různých zdrojů. Takové aplikace budou čím dál častěji spravovat svá vlastní data, s využitím různých technologií, v závislosti na tom, jak jsou data využívána. Tento trend bude probíhat společně s trendem dekompozice aplikačního kódu do separátních komponent, integrovaných skrze webové služby. Hranice jedné takové komponenty udává, kde je správné vyčlenit a ohradit určitou technologii pro ukládání a zpracování dat – technologii zvolenou právě s ohledem na to, jak s daty manipulujeme.“

Pro současné aplikace a projekty je proto typické, že využívají např. MongoDB pro ukládání entit typu článek nebo produkt, Redis pro ukládání uživatelských *sessions* a obsahu košíků, Elasticsearch pro fulltextové vyhledávání a ukládání logů a PostgreSQL pro ukládání informací o dodavatelích a finančních transakcích. Laicky řečeno: doba, kdy na všechny otázky bylo odpověď MySQL, je již minulostí.

8.

Sloupcové databáze

V této kapitole se přesuneme k dalšímu typu NoSQL databází, jež jsou v angličtině označovány jako *column family stores*, *wide column stores*, případně *columnar stores*. Při studiu anglicky psaných zdrojů je důležité tyto pojmy odlišit od termínu *column oriented RDBMS* [149], který označuje relační databáze, jež ukládají záznamy primárně „po sloupcích“ relačních tabulek (např. systémy MonetDB,¹ Vertica² nebo SAP IQ³ pro datové sklady). Pro třídu NoSQL databází, které se věnujeme v této kapitole, budeme dále používat český termín *sloupcové databáze*. Do této kategorie spadají zejména momentálně velmi populární systémy Cassandra⁴ a HBase⁵ a také databáze Hypertable⁶ a Accumulo.⁷ Aktuální seznam databází tohoto typu je k nalezení např. opět na stránkách DB-engines.com.⁸

¹ <http://www.monetdb.org>

² <http://www.vertica.com>

³ SAP IQ byl dříve známý jako Sybase IQ a nyní je součástí platformy SAP (<http://www.sap.com>).

⁴ <http://cassandra.apache.org>

⁵ <http://hbase.apache.org>

⁶ <http://hypertable.org>

⁷ <http://accumulo.apache.org>

⁸ <http://db-engines.com/en/ranking/wide+column+store>

8.1 Datový model

Jak jsme již několikrát uvedli, jednou ze základních odlišností NoSQL technologií od tradičních RDBMS systémů je větší volnost datového modelu. Úložiště typu klíč-hodnota (kapitola 6) obecně nekladou žádná omezení na uložená data – často spravují, z pohledu databáze, pouze nestrukturované binární objekty. Dokumentové databáze (kapitola 7) jednak určují formát ukládaných dat (např. JSON) a jednak předpokládají jisté implicitní schéma dat (záznamy v rámci kolekce mírají stejná pole). Sloupcové databáze jdou v tomto směru dál a umožňují schéma dat definovat přesněji.

Všechny sloupcové systémy zmíněné v úvodu této kapitoly se odvolávají na datový model popsaný v článku z roku 2008, ve kterém vývojáři společnosti Google představili distribuovaný databázový systém *Bigtable* [103]. V této kapitole se nezabýváme přímo touto databází, protože se jedná o interní systém vytvořený v rámci Google, který uživatelé nemají možnost přímo využívat. Při uvádění konkrétních příkladů se proto budeme většinou odvolávat na jeden z nejpopulárnějších systémů – Cassandra.

Základním pojmem popisovaného datového modelu je *řádek (row)* identifikovaný *klíčem řádku (row key)*, který může mít velmi mnoho *sloupců (columns)*. Každý sloupec v daném řádku má svůj *název (column name)*, *hodnotu (column value)* a také *časové razítka*, kdy byla hodnota uložena (*time stamp*). Jednotlivé sloupce jsou sdruženy do *rodin sloupců (column families)*, česky též *skupiny sloupců*. Při návrhu schématu databáze definujeme pouze rodiny sloupců a jednotlivé řádky mohou v rámci dané rodiny volně vytvářet libovolné množství sloupců s libovolnými názvy. Jednu rodinu sloupců můžeme zobrazit tak, jak vidíme na obrázku 8.1. Pro zjednodušení jsou na tomto obrázku vynechána časová razítka hodnot.

user_id (klíč řádku)	název sloupce	název sloupce	...
	hodnota sloupce	hodnota sloupce	...
1	login	first_name	...
	honza	Jan	...
4	login	age	
	david	35	...
5	first_name	last_name	...
	Jana	Novotná	...
...			...

Obrázek 8.1: Příklad rodiny sloupců users

Zobrazená rodina sloupců vypadá téměř jako relační tabulka, ale všimněme si, že každý řádek má různé sloupce. Také počty sloupců v jednotlivých řádcích se mohou i výrazně lišit. Taková data by bylo možné modelovat i pomocí konceptu dokumentu – náš *klíč řádku* odpovídá primárnímu klíči dokumentu (např. hod-

notě s názvem `_id` v systému MongoDB) a dvojice (název sloupce, hodnota sloupce) odpovídají asociativnímu poli `{název : hodnota}` z JSON objektu.

Kromě jednoduchých sloupců umožňují některé sloupcové databáze vytvářet i tzv. *supersloupce* (*super column*), jejichž hodnota je složená z podsloupců (*subcolumns*). Příklad rodiny supersloupců `addresses` je zobrazen na obrázku 8.2.

user_id (klíč řádku)	název supersloupce			název supersloupce			...
	název podsl.	název podsl.	...	název podsl.	název podsl.	...	
	hodnota podsl.	hodnota podsl.	...	hodnota podsl.	hodnota podsl.	...	
1	home_address			work_address			
	city	street	...	city	street	...	
	Brno	Krásná 5	...	Praha	Pracovní 13	...	
4	home_address			temporary_address			
	city	street	...	city	PSČ	...	
	Plzeň	sady Pětatřicátníků 35	...	Praha	111 00	...	
...							

Obrázek 8.2: Příklad rodiny supersloupců `addresses`

Koncept supersloupců de facto odpovídá další úrovni zanoření v JSON objektu. Také bychom mohli říct, že každý supersloupec je zanořenou rodinou sloupců. V rámci rodiny můžeme supersloupce libovolně přidávat (pro vybraný řádek s daným klíčem) – všimněme si, že na obrázku 8.2 má každý řádek supersloupce s různými názvy. Pokud bychom chtěli stejnou situaci modelovat jen pomocí rodin sloupců, museli bychom už při návrhu schématu databáze vytvořit tři rodiny sloupců `work_address`, `home_address` a `temporary_address`. Dalším řešením je vytvořit jednu rodinu sloupců `address` a informaci o druhu adresy přesunout do názvu jednotlivých sloupců, např. `work_city`, `home_street`, `home_city` atd.

Stejný klíč řádku můžeme použít pro řádky v různých rodinách sloupců. Řádky se stejným klíčem pak většinou dohromady tvoří jeden logický datový celek. Každá rodina obsahuje sloupce s hodnotami, které spolu logicky souvisí, bývají stejného typu a předpokládá se, že tyto sloupce budou často dotazovány společně. Jak uvidíme níže, databáze pracují s rodinami sloupců tak, aby právě dotazy na hodnoty z jedné rodiny byly vyhodnocovány efektivně.

Na obrázku 8.3 na následující straně je uveden příklad z článku o Bigtable [103], ve kterém je zachyceno, jak společnost Google interně ukládá (nebo aspoň ukládala v roce 2008) obsah jednotlivých webových stránek, meta informace o těchto stránkách a křízové odkazy mezi stránkami. Klíčem řádku je invertovaná URL adresa stránky a zobrazený řádek je tvořen sloupcem ze tří rodin. V první z nich je pouze jeden sloupec s názvem `contents:html`, ve kterém je uložen přímo HTML obsah stránky; ve druhé rodině jsou sloupce obsahující atributy stránky, jako je jazyk (`param:lang`) a kódování (`param:enc`). Názvy sloupců ve třetí rodině tvoří URL adresy stránek, které obsahují odkazy na danou stránku, a hodnoty těchto sloupců

pak tvoří texty těchto odkazů (obsah HTML tagu `a`). V příkladu vidíme, že např. stránka `iHned.cz` odkazuje na `idnes.cz` s tím, že text tohoto odkazu je `iDNEs`. Dále si všimněme časových razítok t_x u jednotlivých hodnot a toho, že pro sloupec `contents:html` je naznačeno uložení tří různých hodnot s různými časovými razítky t_2 , t_6 a t_8 .

klíč řádku	<i>rodina "contents"</i>	<i>rodina "param"</i>		<i>rodina "a"</i>	
	<code>contents:html</code>	<code>param:lang</code>	<code>param:enc</code>	<code>a:novinky.cz</code>	<code>a:iHned.cz</code>
<code>cz.idnes</code>	$t_2 \rightarrow \boxed{<\text{html}>...}$ $t_6 \rightarrow \boxed{<\text{html}>...}$ $t_8 \rightarrow \boxed{<\text{html}>...}$	$t_2 \downarrow \boxed{\text{CZ}}$	$t_2 \downarrow \boxed{\text{UTF-8}}$	$t_3 \downarrow \boxed{\text{iDNEs.cz}}$	$t_7 \downarrow \boxed{\text{iDNEs}}$

Obrázek 8.3: Příklad použití datového modelu sloupcových databází pro ukládání obsahu webových stránek a meta informací o nich [103]. Příklad obsahuje tři rodiny sloupců a sloupce v těchto rodinách pro klíč řádku `cz.idnes`.

Existují dvě možnosti, jak se dívat na takto strukturovaná data a vlastně na celé sloupcové databáze. Jednak můžeme každou rodinu sloupců vnímat jako relační tabulku s (mnoha) hodnotami `null` (v relační tabulce bychom museli vytvořit všechny požadované sloupce pro každý řádek). Jeden logický záznam by pak byl tvořen řádky se stejným klíčem řádku ze všech různých rodin sloupců (tabulek) – viz obrázek 8.3.

Druhý možný pohled staví do centra zájmu samotné hodnoty v jednotlivých sloupcích a na celou datovou strukturu se dívá jako na multidimenziorní asociativní pole těchto hodnot. Jednotlivé dimenze pole jsou 1) klíč řádku, 2) sloupec, případně 3) supersloupec a 4) časové razítka hodnoty. Toto asociativní pole bývá často seřazené (*sorted map*) podle všech dimenzí.

V programovacím jazyce typu Java bychom takové asociativní pole (bez supersloupců) mohli definovat asi takto:

```
SortedMap<RowKey, SortedMap<ColumnKey, SortedMap<TimeStamp, ColumnValue>>> table
```

Intuitivně bychom pak s daty pracovali např. takto:

```
table['row_key_1']['column_key_1'][timestamp_1] = 'column_value'
```

Ostatně původní článek o Bigtable říká, že „tabulka v Bigtable je řídké, distribuované, persistentní, multidimenziorní asociativní pole. Data jsou organizována ve třech dimenzích: řádky, sloupce a časová razítka“ [103]:

$(row:string, column:string, time:int64) \rightarrow string$.

Dobře je struktura dat vidět ve formátu JSON, ve kterém bychom mohli data ukládat podle následujícího předpisu:

```
{
// řádek
"row_key_1": {
  "column_key_1": {
    "timestamp_1": "column_value_1",
    "timestamp_2": "column_value_2",
    ...
  }
}
```

```
        "timestamp_2": "column_value_2"  
    },  
    "column_key_2": {  
        "timestamp_3": "column_value_3",  
        "timestamp_4": "column_value_4"  
    },  
}  
}
```

Ukažme tento princip na konkrétní příkladu, když rozložíme polynom $x^5 + 2x^3 - x^2 + 1$ na součin dvou členů:

```
// rodina sloupců "users"
{
    // řádek s klíčem "honza"
    "honza": {
        "firstName": {
            "1429268224554000": "Jan"
        },
        "lastName": {
            "1429268224554000": "Novák"
        },
        "email": {
            "1429268226436000": "honza@seznam.cz",
            "1429268234554000": "honza@gmail.com"
        }
    },
    // řádek s klíčem "janicka"
    "janicka": {
        "firstName": {
            "1429268453453040": "Jana"
        },
        "lastName": {
            "1429268453453040": "Novotná",
            "1429239403921900": "Nováková"
        },
        "web": {
            "1429268453453040": "http://janicka.novotna.cz/"
        }
    }
}
```

8. Sloupcové databáze

Pro zápis různých adres uživatele můžeme využít rodinu supersloupců, ve které jsou data uložena o jednu úroveň hlouběji:

```
// rodina supersloupců "addresses"
{
    // řádek s klíčem "honza"
    "honza": {
        // supersloupec "home_address"
        "home_address": {
            "city": {
                "1429268226436000": "Praha"
            },
            "street": {

```

```
"1429268226436000": "Krásná 5",
"1429268786984060": "Slunečná 7"
},
"zip": {
    "1429268226436000": "111 00"
}
},
// supersloupec "temporary_address"
"temporary_address": {
    "city": {
        "1429268226436000": "Brno"
    },
    "street": {
        "1429268348657000": "Nudná 13"
    }
}
}
```

Detailly datového modelu se pro konkrétní systémy liší, ale základní myšlenka zůstává. Existují dvě základní motivace ke vzniku tohoto modelu a s nimi související rozdíly oproti standardnímu relačnímu modelu. Za prvé je to samozřejmě možnost volného přidávání sloupců a schopnost databáze efektivně zvládnout tuto volnost. Použitím rodin sloupců databázi dále sdělujeme, ke kterým hodnotám se bude přistupovat společně. Databázový systém pak může efektivně data distribuovat na různé uzly s tím, že sloupce z jedné rodiny (pro daný řádek) zůstanou vždy na jednom uzlu. Podrobnosti k tomuto principu a jeho realizaci jsou uvedeny dále.

Je nutné za každou cenu distribuovat?

Při volbě databáze je vždy třeba zvážit, zda opravdu potřebujeme a využijeme distribuovanou databázi – distribuovanost vždy přidává další úroveň složitosti, zvyšuje režijní náklady už jen vzhledem k tomu, že data je třeba po síti získávat z více serverů. Platíme tak zvýšenou latencí dotazů, vyšší pravděpodobností selhání některého ze serverů atd. Například v dokumentaci systému Apache HBase se říká [2]: „Systém HBase není vhodný pro každý problém. Za prvé se ujistěte, že máte dost dat. Pokud máte stovky milionů nebo miliardy řádků, pak je pro vás HBase dobrým kandidátem.“

Ostatně výhod nerelačního datového modelu můžeme využít i při použití nedistribuované databáze, jako je Redis (viz sekce 6.3.2) nebo PostgreSQL (viz sekce 14.1.1).

8.2 Cassandra: datový model sloupců v praxi

Představený datový model je v různých systémech lehce modifikován a využíván různými způsoby. V této sekci ukážeme jeho aplikaci v systému Cassandra, což je sloupcová databáze vytvořená původně interně ve společnosti Facebook. Nyní je Cassandra open source projektem v rámci iniciativy Apache Software Founda-

tion.⁹ Systém je implementovaný v jazyce Java, pro definici struktury dat i dotazy používá jazyk CQL (Cassandra Query Language) [7] a umožňuje vyhodnocování agregačních dotazů pomocí MapReduce přístupu s využitím frameworku Apache Hadoop (viz sekce 4.3). V souladu s datovým modelem rodin sloupců umožňuje implementace systému Cassandra mít prakticky neomezený počet sloupců pro každý řádek, což je jeden ze zásadních rozdílů oproti tradičním relačním databázím.¹⁰ Dalšími výhodami je fyzické ukládání hodnot sloupců z jedné rodiny blízko sebe (fyzická kolokace dat) a samozřejmě distribuovaná architektura systému Cassandra.

8.2.1 Data jako multidimenziorní pole

Možnosti datového modelu i jeho zápis se v průběhu vývoje systému Cassandra vyvíjely. Původní přístup (jediný možný do verze 1.2) velmi dobře odpovídá pohledu na data jako na multidimenziorní asociativní pole, které jsme popsali v předchozí části. Proto se tomuto přístupu budeme pro srovnání krátce věnovat, i když v současné verzi systému Cassandra není doporučovaný a nejspíše bude v budoucnu odstraněn. Jednoduchá definice datové struktury v této syntaxi (nyní označované jako Thrift API, viz sekce 2.6.2) vypadá např. takto:

```
create column family users
    with key_validation_class = Int32Type
    and comparator = UTF8Type
    and default_validation_class = UTF8Type;
```

Tento příkaz vytvoří rodinu sloupců s názvem `users` s tím, že jako klíče řádků jsou očekávána celá čísla, názvy sloupců budou UTF-8 řetězce a hodnoty v těchto sloupcích také. Data pak vkládáme tímto způsobem:

```
// hodnoty sloupců v řádku s klíčem 7
set users[7]['login'] = utf8('honza');
set users[7]['name'] = utf8('Jan Novák');
set users[7]['email'] = utf8('jan@novak.name');

// hodnoty sloupců v řádku s klíčem 13
set users[13]['login'] = utf8('fantomas');
set users[13]['name'] = utf8('incognito');
```

K jednotlivým uloženým hodnotám přistupujeme pomocí příkazu `get`, případně můžeme všechny uložené hodnoty vypsat pomocí příkazu `list` (odpovědi systému na jednotlivé příkazy jsou opět vyznačeny kurzívou):

```
get users[7]['login'];
=> (name=Login, value=honza, timestamp=1429268223462000)

get users[13];
=> (name=Login, value=fantomas, timestamp=1429268224554000)
=> (name=name, value=incognito, timestamp=1429268224555000)
```

⁹ <http://apache.org>

¹⁰ Většina relačních databází má fyzické omezení počtu sloupců v dané tabulce. Např. u PostgreSQL je tento limit 250 až 1600 v závislosti na datovém typu sloupců [1].

```
list users;
-----
RowKey: 7
=> (name=email, value=jan@novak.name, timestamp=1429268223473000)
=> (name=login, value=honza, timestamp=1429268223462000)
=> (name=name, value=Jan Novák, timestamp=1429268223471000)
-----
RowKey: 13
=> (name=login, value=fantomas, timestamp=1429268224554000)
```

Jak je vidět, tento přístup dobře odpovídá datovému modelu asociativních polí, a to včetně dimenze časových razítok.

8.2.2 Data jako řídké tabulky

Systém Cassandra se ovšem dále vyvíjel směrem ke druhé zmiňované interpretaci datového modelu, která vnímá jednotlivé rodiny sloupců jako tabulky. Termín *column family* se změnil na *table* (tabulka), pojem *column* na *cell* (buňka) a celý jazyk CQL vychází ze známých a osvědčených konstrukcí jazyka SQL. Základní definice dat v CQL tedy vypadá např. takto:

```
CREATE TABLE users (
    user_id int PRIMARY KEY,
    login text,
    name text,
    email text );
```

Základní manipulace s daty je také prakticky shodná se syntaxí známou z SQL:

```
INSERT INTO users (user_id, login, name) VALUES (3, 'honza', 'Jan Novák');

SELECT * FROM users;
user_id | email | login | name
-----+-----+-----+
 3 | null | honza | Jan Novák
```

Povšimněme si, že v tomto příkladu systém vypsal pro sloupec `email` hodnotu `null`, protože hodnota pro tento sloupec nebyla nastavena. Ve standardní relační databázi by v tomto případě byla hodnota `null` také fyzicky v databázi uložena. V systému Cassandra není uložena hodnota žádná a ve výpisu se `null` objeví proto, že je sloupec `email` v tabulce `users` definován.

Cassandra tedy efektivně pracuje s nedefinovanými hodnotami, ale výše uvedená syntaxe `CREATE TABLE` neumožňuje přímo dynamicky přidávat libovolné sloupce. Jedním ze způsobů, jak dosáhnout požadované dynamičnosti, jsou *kolekce*. Cassandra umožňuje vytvářet sloupce několika typů kolekcí: množina (*set*), seznam (*list*) a zejména asociativní pole (*map*). Syntaxe CQL použitá při definici dat s využitím těchto typů je pak následující:

```
CREATE TABLE users (
    login text PRIMARY KEY,
    name text,
```

```

    emails set<text>,           // sloupec typu množina
    profile map<text, text> // sloupec typu asociativní pole
);

```

Použití kolekcí je v CQL přirozené a pohodlné. Následující příklad vloží nové hodnoty do tabulky `users` včetně jedné hodnoty do množiny `emails` a dvou dvojic (klíč, hodnota) do asociativního pole `profile`:

```

INSERT INTO users (login, name, emails, profile) VALUES
  ('honza', 'Jan Novák', { 'honza@novak.cz' },
   { 'colorschema': 'green', 'design': 'simple' } );

```

Následující příklad přidá do množiny e-mailů uživatele `honza` další e-mail:

```

UPDATE users SET
  emails = emails + { 'jn@firma.cz' } WHERE login = 'honza';

```

Pro všechny kolekce platí, že vnitřně Cassandra pracuje s jejich hodnotami jako s jednotlivými sloupci. Použití kolekcí je tedy plnohodnotnou náhradou za dynamické rodiny sloupců s tím, že asociativní pole (*map*) odpovídá principu supersloupců z výše popsaného datového modelu.

Existuje také další možnost, jak vytvořit v systému Cassandra tabulku s plně dynamickým počtem sloupců, tedy rodinu sloupců. Využijeme k tomu primárního klíče složeného z názvu dvou sloupců:

Příklad 8.1: Příklad vytvoření tabulky se složeným klíčem v CQL

```

CREATE TABLE mytable (
  row_id int,
  column_name text,
  column_value text,
  PRIMARY KEY (row_id, column_name)
);

```

Tímto způsobem dosáhneme kýženého efektu a s daty pak budeme pracovat přirozeně takto:

```

INSERT INTO mytable (row_id, column_name, column_value) VALUES
  ( 3, 'login', 'honza' );
INSERT INTO mytable (row_id, column_name, column_value) VALUES
  ( 3, 'name', 'Jan Novák' );
INSERT INTO mytable (row_id, column_name, column_value) VALUES
  ( 3, 'email', 'honza@novak.cz' );

```

Uvedený přístup lze samozřejmě použít i v jakémkoli tradičním relačním databázovém systému. Rozdíl je v tom, jakým způsobem systém s takto navrženým schématem dat pracuje, zejména jak data rozmiští na uzly v distribuovaném prostředí. K těmto otázkám se dostaneme v následující sekci této kapitoly.

8.3 Struktura a vlastnosti systému

V této sekci se zaměříme na to, jak je ve sloupcových databázích realizována distribuce dat a jak jsou data organizována lokálně na jednotlivých uzlech.

8.3.1 Distribuce a replikace dat

Klíčovou vlastností sloupcových databází je bezesporu jejich distribuovaná architektura. Podobně jako úložiště typu klíč-hodnota a dokumentové databáze rozdělují i sloupcové systémy data mezi jednotlivé uzly podle *dělicího klíče (partition key)* – vybraného sloupce nebo kombinace sloupců. Primárním přístupem k distribuci dat není hašovací funkce, ale intervalové dělení tak, jak je naznačeno na obrázku 8.4.

user_id (klíč řádku)	login	name
1	honza	Jan...
4	david	David...
...		
1000	karel	Karel...
1001	irena	Irena...
1003	jirka	Jiří...
...		
2000		
...		

Obrázek 8.4: Příklad dělení tabulky na jednotlivé části

V jednotlivých sloupcových systémech se setkáme s různými názvy pro vzniklé části tabulky: *tablets* (Bigtable), *regions* (HBase) nebo *partitions* (Cassandra). Pokud není určeno jinak, je v systému Cassandra dělicí klíč tabulky tvořen *prvním názvem sloupce* z primárního klíče tabulky.

Dále budeme vycházet z příkladu 8.1 na předchozí straně, který definuje tabulku se složeným primárním klíčem (`row_id`, `column_name`). Připomeňme, že složený primární klíč znamená, že v tabulce nemohou existovat dva řádky se stejnou dvojicí hodnot těchto sloupců. V našem případě bude dělicím klíčem sloupec `row_id`, což znamená, že všechny záznamy se stejnou hodnotou tohoto sloupce budou umístěny na stejném uzlu systému. Jak víme, uvedená definice tabulky umožňuje dynamickou práci se sloupci a právě nastavení dělicího klíče tímto způsobem je zásadní vlastností z hlediska efektivity zpracování dotazů. Dá se totiž předpokládat, že se dotazy často budou týkat právě záznamů se stejným `row_id`, které dohromady tvoří jeden logický datový celek, a je žádoucí, aby tyto záznamy byly na stejném fyzickém uzlu.

Pokud je primární klíč složen z více sloupců, systém Cassandra použije názvy ostatních sloupců za dělicím klíčem jako tzv. *clustering columns*. V rámci každého uzlu pak budou záznamy fyzicky uloženy seřazené právě podle hodnot těchto sloupců, čehož je také využito pro efektivnější práce s dotazy (viz níže). Pokud bychom chtěli nastavit dělicí klíč jako kombinaci několika sloupců (nikoliv pouze jako první sloupec primárního klíče), můžeme to udělat pomocí uzavření požadovaných názvů sloupců do závorky, např. takto:

```
CREATE TABLE tab ( a int, b text, c text, d text,
    PRIMARY KEY ( (a, b), c )
);
```

V tomto příkladu bude dvojice **(a, b)** dělicím klíčem a sloupec **c** bude tvořit clustering column.

Další důležitou vlastností distribuovaného systému je replikace dat. Obecně vzato jsou v tomto ohledu u sloupcových databází využívány stejné principy a mechanismy jako u úložišť typu klíč-hodnota a dokumentových databází. Cassandra implementuje peer-to-peer replikaci (viz sekce 3.3.3) a poskytuje řadu možností, které ovlivňují chování systému na škále mezi vysokou dostupností systému a garancí zachování konzistence dat. Tyto možnosti vycházejí z principu kvór pro čtení a zápis, které byly podrobně rozebrány v sekcích 3.3.2, 6.2.2 a 7.3.2.

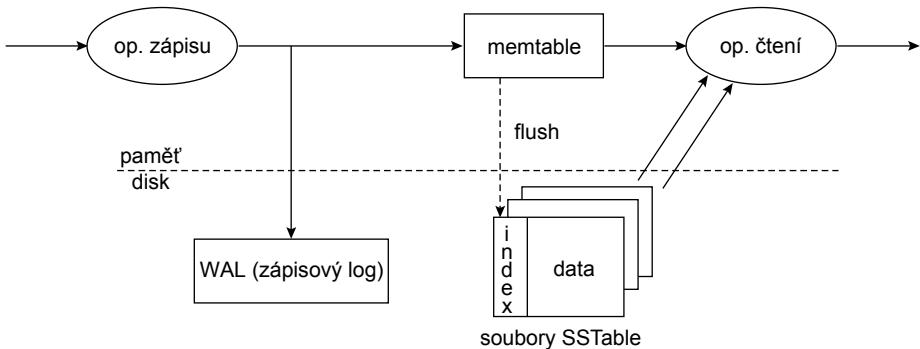
Naproti tomu systém HBase, který nejvěrněji kopíruje původní architekturu popsanou v článku o Bigtable, je postavený nad distribuovaným souborovým systémem HDFS (viz sekce 4.5 na straně 73). Pro replikaci dat tedy HBase využívá vnitřní mechanismy HDFS. Dále HBase umožňuje master-slave replikaci na úrovni celého HBase clusteru, která slouží zejména pro zálohování dat.

8.3.2 Lokální organizace dat

Zaměřme se nyní na to, jakým způsobem je ve zmiňovaných systémech (a nejen v nich) řešeno fyzické ukládání dat na jednotlivých uzlech a zpracování operací na této úrovni. Úkolem lokálního úložiště je na jedné straně zajištění persistenze dat a trvalosti provedených změn, na straně druhé pak dosažení co nejvyšší propustnosti operací zápisu a čtení. Často se proto využívá kombinace technik zápisového logu (*write-ahead log*) a paměťových tabulek (*memtable*), jak je naznačeno na obrázku 8.5 na následující straně.

Na disku jsou data často ukládána ve struktuře zvané *SSTable* (*sorted string table*). Koncept SSTable byl navržen společností Google jako formát pro persistentní soubor typu klíč-hodnota, který má pevně danou velikost, a byl používán např. právě v systému Bigtable [103]. Velmi podobná struktura se používá také v systémech Cassandra (SSTable) a HBase (HFile).

Klíčovou otázkou není přesný formát těchto diskových struktur, ale jejich použití v popisovaných databázových systémech – obsah každé SSTable je totiž vždy neměnný v čase (*immutable*). Všechny operace modifikující data jsou nejprve zaznamenány do zápisového logu, do kterého se zapisuje pouze na konec, a proto bývá velmi rychlý. Změny jsou dále zaznamenány do paměťové struktury memtable, která se uloží na disk do SSTable ve chvíli, kdy je plná. Operace čtení



Obrázek 8.5: Lokální organizace datového úložiště pomocí zápisového logu a struktur SSTable a memtable

berou v úvahu data v paměti i na disku (o každé SSTable se v paměti drží příslušná metadata).

Čas od času je v systému spuštěna operace *konsolidace* (*compaction*, *consolidation*), kdy se struktury SSTable nahrají do paměti a data z nich se přesklupí tak, aby bylo čtení co nejefektivnější. V rámci konsolidace jsou také odstraněny starší hodnoty, které byly nahrazeny novějšími, a také jsou fyzicky odstraněny záznamy, které byly smazány operacemi promítnutými zatím pouze do paměti. Popsaný proces je naznačen na obrázku 8.6 na následující straně.

Tento princip byl poprvé uceleně popsán ve struktuře nazvané *Log-Structured Merge-Tree* (LSM Tree) [133] a využívá se nejen v systémech Cassandra a HBase, ale také např. v úložištích typu klíč-hodnota LevelDB, RocksDB a v dalších soudobých databázích.

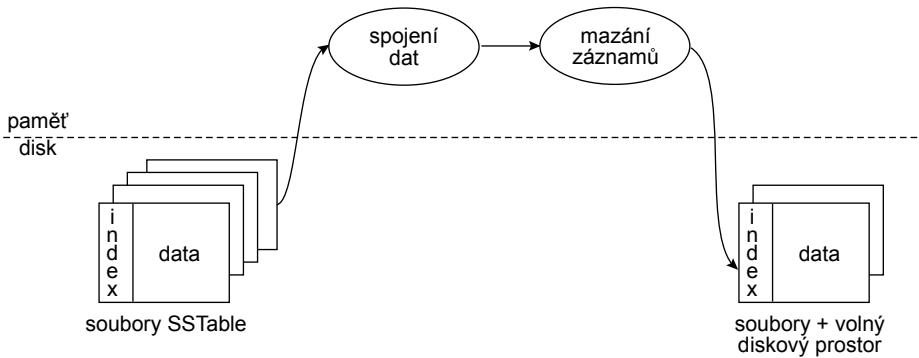
Cassandra v praxi

Pravděpodobně jedním z nejvýznamnějších nasazení sloupcové databáze v praxi je využití databáze Cassandra ve společnosti Netflix,¹¹ která vysílá filmy a seriály prostřednictvím internetu. Netflix v databázi Cassandra ukládá naprostou většinou informací, se kterými aplikace pracuje – hodnocení pořadů, záložky, historie zhlédnutých pořadů a další metadata – a je schopna prostřednictvím interních nástrojů vytvořit nový cluster během 10 minut. V současné době produkční cluster zpracovává 10 milionů operací za sekundu [29]. Pro detailní informace o měření výkonu Cassandra viz [111].

8.4 Dotazy, indexy a transakce

Dále se zaměříme na možnosti dotazování dat ve sloupcových databázích a způsoby, jakými jsou dotazy vyhodnocovány. Opět budeme klást důraz na systém Cassandra a v příkladech budeme využívat konstrukce jeho jazyka CQL.

¹¹ <http://www.netflix.com>



Obrázek 8.6: Schéma procesu konsolidace lokálního úložiště

8.4.1 Dotazy

Po syntaktické stránce dotazovací část CQL vychází ze standardních konstrukcí SQL, ale zdaleka nezahrnuje všechny jeho příkazy. Na druhou stranu má CQL také vlastní konstrukce, které vycházejí z distribuovaného charakteru systému Cassandra a jeho specifických implementačních detailů. Základní struktura dotazu vypadá poměrně standardně takto:

```
SELECT <selectExpr>
FROM [<keyspace>.]<table>
[WHERE <clause>]
[ORDER BY <clustering_colname> [DESC]]
[LIMIT m];
```

Cassandra tedy umožňuje dotazovat pouze jednu tabulkou (v části FROM je striktně uvedeno <table>). Ostatní části příkazu SELECT mají na první pohled standardní sémantiku: v části <selectExpr> je seznam názvů sloupců, jejichž hodnoty mají být na výstupu (operace *projekce*), WHERE obsahuje omezující podmínky na hodnoty jednotlivých sloupců (operace *selekce*), ORDER BY je řazení výsledku a LIMIT omezení počtu řádků na výstupu. Vyjdeme-li opět z definice tabulky z příkladu 8.1 na straně 135, můžeme v CQL napsat např. dotaz:

```
SELECT column_name, column_value
FROM mytable
WHERE row_id=3
ORDER BY column_value;
```

Dotazovací jazyky

Přístup k databázi pomocí dotazovacího jazyka podobného SQL je specifikem systému Cassandra. Např. při používání HBase, druhé nejvíce rozšířené sloupcové databáze, máme dvě možnosti: použít nativní Java API, nebo některou součást ekosystému Hadoop – Hive (viz sekce 4.3.3.3) vycházející ze syntaxe SQL či Pig (viz sekce 4.3.3.1) poskytující doménově specifický jazyk pro načítání a transformaci dat. Tuto funkcionality ovšem nabízí i systém Cassandra.

Z distribuované podstaty sloupcových systémů ale vyplývají další omezení kladená na zápis dotazů. Obecně řečeno, systém Cassandra nevyhodnotí (nebo ani neumí vyhodnotit) dotaz, který by vyžadoval „příliš náročné“ zpracování. Konkrétně, omezující podmínky v části `WHERE` mohou být pouze na některých sloupcích a jejich kombinacích.

- Podmínka může být na názvech sloupců dělicího klíče (tedy první sloupec nebo sloupcy z primárního klíče), ale pouze s využitím operátorů `=` nebo `IN`, např. `WHERE row_id IN (3, 4, 5)`. Tím je mimo jiné zaručeno, že dotaz bude zpracováván pouze na jednom nebo několika málo uzlech systému.
- Podmínka může být na sloupcích tvořících clustering columns (tedy na ostatních sloupcích primárního klíče), např. `WHERE column_name='login'`. Pokud zároveň v podmínce omezujeme i dělicí klíč, tedy `WHERE row_id=3 AND column_name='login'`, pak je vyhodnocení celé podmínky opravdu efektivní. Pokud se podmínka týká pouze sloupců tvořících clustering columns, tak by systém stejně musel projít a přefiltrovat všechny řádky tabulky, což lze výslovně povolit takto:
`SELECT * FROM mytable WHERE column_name IN ('login', 'name') ALLOW FILTERING;`
- Na hodnoty jiných sloupců můžeme klást omezení pouze tehdy, pokud je na nich postavený index (viz sekce 8.4.2).
- Pokud používáme v některých sloupcích kolekce (např. `emails set<text>`, `profile map<text, text>`), tak na těchto sloupcích můžeme také postavit indexy a poté dotazovat hodnoty pomocí klíčového slova `CONTAINS` takto:
`SELECT login FROM users WHERE emails CONTAINS 'jn@firma.cz';`
`SELECT * FROM users WHERE profile CONTAINS KEY 'colorschema';`
- Data v systému Cassandra jsou fyzicky uložena na disku jednotlivých uzlů seřazená podle hodnot prvního sloupee tvořícího clustering column. Použití rozsahových dotazů a řazení výsledku pomocí `ORDER BY` je proto v současné chvíli v systému Cassandra podporováno pouze na tomto sloupci.

V části `<selectExpr>` je možné použít funkci `COUNT`, která vrátí pouze počet řádků ve výsledku. Další agregační funkce v době psaní tohoto textu nejsou v systému Cassandra podporovány. Složitějších analytických funkcí je ale možné dosáhnout využitím MapReduce a dalších Hadoop technologií.

8.4.2 Indexy

Způsob, jímž pracuje Cassandra se sloupci z primárního klíče, jsme vysvětlili podrobnej výše. Systém dále umožňuje vytvořit sekundární indexy na dalších vybraných sloupcích, včetně sloupců obsahujících kolekce. Syntaxe CQL vychází opět z jazyka SQL:

```
CREATE INDEX ON users (emails);
```

Kromě standardních předdefinovaných implementací indexů (jako B⁺-strom) je možné vytvářet také vlastní typy indexů.¹²

Na základě seznamu „omezení“ možností CQL oproti SQL bychom mohli říci, že funkcionality sloupcových databází silně pokulhává za zavedenými relačními databázemi. Uvědomme si ale, že všechna tato omezení jsou pouze přirozenou daní za horizontální škálovatelnost sloupcových systémů, tedy za tu hlavní výhodu, kvůli níž sloupcové databáze začaly vznikat. Samozřejmě platí, že chceme-li používat distribuované databáze místo standardních relačních, je potřeba výrazně více přemýšlet o schématu dat, určení klíčů tabulek, vytváření indexů a o dotazech, které bude databáze vyhodnocovat. V kontextu distribuovaných úložišť popsaných v předchozích kapitolách ale sloupcové databáze přináší možnost, jak přirozeným způsobem pracovat s atributovými daty v distribuovaném prostředí.

8.4.3 Transakce

Jak jsme již zmínili, v distribuovaném prostředí je obtížné a hlavně neefektivní plně podporovat transakční zpracování tak, jak je běžné v centralizovaných RDBMS. Proto distribuované systémy, které cílí na efektivitu a vysokou propustnost operací, často implementují *odlehčené transakce* (*lightweight transactions*). V případě systému Cassandra se tento princip realizuje pomocí *podmíněných operací*, což jsou speciální složené příkazy, které se vyhodnocují atomicky (princip operací „test and set“), například:

```
INSERT INTO users (login, name, emails)
    VALUES ('honza', 'Jan Novák', { 'honza@novak.cz' }) IF NOT EXISTS;

UPDATE mytable SET column_value = 'honza@firma.cz'
    WHERE row_id = 3 AND column_name = 'email'
    IF column_value = 'honza@firm.cz';
```

Pokud by dvě takové operace měly být vyhodnoceny na dvou různých replikách, je potřeba zabránit vzniku konfliktu a dovolit pouze jedné z těchto operací úspěšné dokončení. Pro řešení konfliktů situací při souběhu takových operací využívá systém Cassandra variantu protokolu Paxos¹³ [122], která je méně náročná než např. dvoufázový COMMIT protokol (viz sekce 12.3.1). Více o transakcích v distribuovaném prostředí viz kapitola 12.

¹² Vlastní indexy se implementují v jazyce Java a systému se předá pouze informace o názvu Java třídy, která index definuje [8].

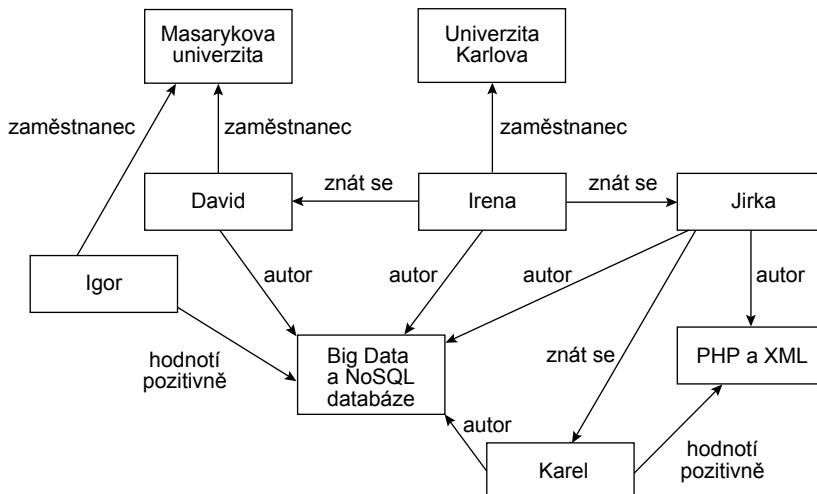
¹³ Paxos je rodina protokolů pro řešení *problému shody* (*consensus*). Protokol Paxos byl původně navržen Leslie Lamportem v roce 1989 a publikován v časopise v roce 1998. Na tyto původní články navazuje řada dalších prací.

9.

Grafové databáze

Tři typy NoSQL databází představené v kapitolách 6, 7 a 8 mají mnoho společného. Předpokládají, že máme data identifikována pomocí klíče, a liší se tím, zda jejich hodnotu považujeme za jediný binární objekt, množinu sloupců nebo složitější datovou strukturu vyjádřenou pomocí formátu XML, JSON apod. Čtvrtý typ NoSQL databází, který představíme v této kapitole, pracuje se zcela jinou datovou strukturou – s grafem, tedy množinou uzlů vzájemně propojených hranami. Předpokládejme, že chceme uložit informace o množině objektů reálného světa a vztazích mezi nimi. Uzel v grafu odpovídá jednomu objektu, který může mít množinu atributů (vlastností) jako např. jméno nebo věk. Hrany, typicky orientované, představují vztahy mezi objekty. Hrany také mohou mít atributy, jako je typ (např. zaměstnanec nebo autor), dobu platnosti vztahu, podmínky platnosti vztahu apod. Jinými slovy řečeno, v grafové databázi můžeme mít několik typů vztahů mezi objekty, přičemž nemáme omezení ani na jejich konkrétní typ, ani na jejich počet.

Uvažme graf na obrázku 9.1 na následující straně. Z názvů uzlů můžeme usoudit, že databáze popisuje osoby, instituce a knihy. Hrany vyjadřují vztahy „zaměstnanec“ mezi osobami a institucemi, „autor“ mezi osobami a knihami, „znát se“ mezi osobami a „hodnotí pozitivně“ mezi osobami a knihami. V takové databázi můžeme hledat odpověď na dotaz „kteří zaměstnanci Masarykovy univerzity pozitivně hodnotí knihu Big Data a NoSQL databáze“, „kdo zná některého z autorů knihy



Obrázek 9.1: Příklad grafové databáze

Big Data a NoSQL databáze“ nebo „kdo z kolegů (tedy zaměstnanců stejného zaměstnavatele) nehodnotí pozitivně knihu, které napsal David“.

Reprezentovat data pomocí grafu není jistě žádná novinka. Stejně tak není žádným objevem, že takové informace chceme a můžeme persistentně ukládat, mimo jiné i pomocí tradičních relačních databází. Proč bychom tedy měli vytvářet nový typ databázových systémů vhodný pouze pro grafy,¹ když do relační databáze můžeme uložit v podstatě cokoli, tedy i strukturu grafu? Relační databáze sice umožňují uložení grafových struktur, ovšem právě pro tento typ dat mají dvě zásadní nevýhody. První z nich je poměrně nízká efektivita implementace průchodu grafem, který typicky vyžaduje velké množství operací spojení tabulek. To je dáno především tím, že v relačních databázích neukládáme primárně data tak, abychom k danému uzlu efektivně získali všechny jeho sousedy, což je základní operace pro průchod grafem. Mohli bychom využít množinu indexů, jež by efektivitu zvýšily, ale za cenu všech nevýhod, které s indexy souvisejí. Grafové databáze naopak tuto vlastnost (tzv. *adjacency property* nebo také *index-free adjacency*) primárně mají, tj. hlavním cílem je právě co nejfektivnější uložení vzhledem k předpokládaným klasickým operacím nad grafy.

Druhou nevýhodou je, že relační databáze vyžadují předem definované schéma, které přirozeně definujeme tak, aby co nejvíce vyhovovalo předpokládaným datazům. Chceme-li ale později např. přidat nový typ vztahu a ten začlenit také do dotazů, znamená to typicky netriviální modifikaci schématu, souvisejících indexů, operací s daty atd. Naproti tomu v grafových databázích průchod grafem nevyhodnocujeme v okamžiku vyhodnocování dotazu, ale máme předpřipravené a uložené vhodné datové struktury, které jsou optimální bez ohledu na strukturu konkrétního grafu. Základní datové struktury pro reprezentaci grafů představíme v sekci 13.1.

¹ Na tomto místě je ale třeba poznamenat, že ani grafové databáze nejsou objevem poslední doby. První návrhy systémů i dotazovacích jazyků se objevovaly již v 80. letech minulého století [156]. Stejně jako v mnoha jiných oblastech IT se tedy i v tomto případě jedná spíše o znovuobjevení a rozšíření úspěšné myšlenky.

Typickou aplikací pro grafové databáze jsou např. sociální sítě, resp. jakákoli doména, která je bohatá na vztahy a reference. Další obvyklou aplikací pro uplatnění grafových databází jsou nejrůznější oblasti pro doručování, směrování, komunikaci a obecně jakákoli data, která hovoří o umístění. A v neposlední řadě jsou to různé doporučovací systémy, jež se snaží predikovat preference nebo ohodnocení, které by pravděpodobně nějakému objektu uživatel dal. Naopak se grafové databáze příliš nehodí, pokud potřebujeme ukládat velká data, která je nutné distribuovat. Distribuce grafových dat se na rozdíl od předchozích tří typů databází dělá obtížně, zvláště pokud je graf téměř úplný, tj. mezi většinou dvojic uzlů vede hrana. Proto existuje pouze několik databázových systémů, které umí pracovat s distribuovaným grafem. Některé z nich představíme v sekci 9.5 a z obecného hlediska se této problematice budeme věnovat v sekci 13.3.

Obecně je cílem této kapitoly seznámit čtenáře s grafovými databázemi především z uživatelského pohledu. Nicméně vzhledem k tomu, že se datový model grafových databází výrazně liší od datového modelu předchozích tří typů databází, liší se významně i problematika reprezentace dat, vyhodnocování dotazů, indexace, distribuce velkých grafů apod. Těmto principům bude proto věnována samostatná kapitola 13.

9.1 Typy grafů a související pojmy

Ještě než se zaměříme na grafové databáze, je třeba se seznámit s typy grafů, s nimiž se v oblasti grafových databází můžeme setkat. V sekci 9.5 pak uvidíme, že různé typy grafových databází podporují různé typy grafů.

Jednou ze základních klasifikací grafů jsou grafy *orientované* (např. v případě formátu XML, RDF nebo dopravních sítí) nebo *neorientované* (např. v případě sociálních sítí nebo chemických sloučenin). Neorientovaná hrana vlastně znamená, že máme hranu vedoucí oběma směry, tj. že daný vztah platí pro oba objekty bez ohledu na pořadí. Např. na obrázku 9.1 na předchozí straně máme hrany orientované. Často také pracujeme s pojmem *cesta*, která je tvořena počátečním uzlem, koncovým uzlem a seznamem hran, jejichž průchodem se dostaneme z počátečního do koncového uzlu. *Délka cesty* je pak počet jejích hran. Jsou-li počáteční a koncový uzel totožné, nejsou spojeny hranou a cesta má délku 0. V orientovaném grafu pak můžeme rozlišovat cesty vedoucí pouze ve/proti směru orientace hran.

Dále rozlišujeme *jednovztažové* grafy, v nichž mají všechny hrany stejný, homogenní a tudíž často nespecifikovaný typ, jako např. existenci dopravního spojení. Ve *vícevztažových* neboli *ohodnocených* (*labeled*) grafech mají hrany typy, kterými rozlišujeme jejich význam, jako např. „autor“ nebo „znát se“ na obrázku 9.1 na předchozí straně. Máme-li hrany různých typů, můžeme vytvářet *multigrafy*, tedy grafy, v nichž mezi dvěma vrcholy může vést více hran (různého typu).

Uvážíme-li klasifikaci hran, pak obvykle předpokládáme, že jedna hrana spojuje právě dva vrcholy, které mohou být případně totožné – pak hovoříme o *smyčce*. Povolíme-li ale, aby hrana spojovala více než dva vrcholy, tedy vyjadřovala vztah mezi více než dvěma uzly, umožníme vytváření *hypergrafů*.

Kromě názvu, resp. typu, které jsou často reprezentovány specifickým způsobem, mohou mít uzly i hrany jakékoli další atributy, jako např. věk nebo váhu. Pak hovoříme o *atributovém grafu* (*property graph*) . Např. na obrázku 9.2 na straně 150 je uveden příklad atributového grafu, v němž mají atributy uzly i hrany.

9.2 Databáze Neo4j

Podívejme se nyní na konkrétní grafovou databázi a na to, jakou funkcionalitu nabízí svým uživatelům. V současné době pravděpodobně nejznámější a dle stránek DB-Engines.com² nejpopulárnější grafovou databází je systém Neo4j³ americko-švédské firmy Neo Technology, jehož první verze pochází z roku 2007. Vzhledem k tomu, že je implementován v jazyce Java, je přenositelný mezi operačními systémy. Je dostupný v open source i komerční verzi. Sami tvůrci systému označují jako jednoduchý, intuitivní, spolehlivý, škálovatelný, efektivní a zabudovatelný do jiných systémů. V kontextu databází pro Big Data je zajímavá především otázka škálovatelnosti, která je ale omezená, jelikož Neo4j nedokáže uložený graf distribuovat. Spolehlivost je spojována s transakčními vlastnostmi ACID, které ale také mají určitá omezení.

9.2.1 Datový model Neo4j

Datový model v Neo4j zahrnuje uzly a hrany. Hrana má vždy typ určený svým názvem a spojuje právě dva uzly. Je tedy možné vytvářet multigrafy, ale nikoli hypergrafy. Hrany jsou vždy orientované, vzhledem k uzlu pak rozlišujeme, zda se jedná o vstupní nebo výstupní hranu, resp. jestli se jedná o počáteční nebo koncový uzel hrany. Nicméně orientaci hran můžeme ignorovat, pokud ji pro danou aplikaci nepotřebujeme. Algoritmy pro průchod grafem stejně efektivně procházejí grafem ve směru i proti směru hrany.

Jak uzly, tak hrany mohou mít atributy. Atribut je v podstatě dvojice (klíč, hodnota), přičemž klíč je typu řetězec (datový typ String z jazyka Java), zatímco hodnota může mít některý z jednoduchých datových typů jazyka Java (boolean, byte, float, char atd.) nebo opět typ String, popř. pole hodnot některého z těchto datových typů.

Poznamenejme ještě, že na rozdíl od relačních databází nemá Neo4j speciální hodnotu null. Pokud chceme vyjádřit, že daný atribut uzel nebo hrana nemá, prostě příslušnou dvojici (klíč, hodnota) neuvedeme. Nicméně později uvidíme, že se s hodnotou null můžeme setkat např. ve výsledcích dotazů, kde nás právě o neexistenci nějakého atributu bude informovat.

² <http://db-engines.com/en/ranking/graph+dbms>

³ <http://www.neo4j.org>

9.3 Přístup k databázi Neo4j

Pro práci s databází Neo4j existuje několik možností. K databázi můžeme přistupovat přímo z programového kódu, tedy např. pomocí Java API nebo REST API, popř. využít rozhraní, která nabízejí speciální jazyky pro práci s grafy. V případě Neo4j je to především jazyk Gremlin a jazyk Cypher.

9.3.1 Java API

Jak již bylo uvedeno v úvodu, databáze Neo4j může být zabudována do jiného systému. Například v rozhraní pro jazyk Java bychom graf z obrázku 9.1 na straně 144 (obohacený o další atributy) vytvořili takto:

Příklad 9.1: Java API v Neo4j

```
// výčet typů vztahů
private static enum RelTypes implements RelationshipType
{
    KNOWS,           // hrana typu "znát se"
    EMPLOYEE,        // hrana typu "zaměstnanec"
    AUTHOR          // hrana typu "autor"
};

// vytvoření nové databáze
GraphDatabaseService graphDb =
    new GraphDatabaseFactory().newEmbeddedDatabase("/home/db/testDB");

// vytvoření dvou uzlů a nastavení jejich atributů name
Node n1 = graphDb.createNode();
n1.setProperty( "name", "Irena" ); // uzel se jménem Irena
Node n2 = graphDb.createNode();
n2.setProperty( "name", "Jirka" ); // uzel se jménem Jirka

// vytvoření hrany typu KNOWS ("znát se") a nastavení jejího atributu since ("od kdy")
Relationship rel1 = n1.createRelationshipTo(n2, RelTypes.KNOWS);
    // vztah "znát se"
rel1.setProperty("since", 2003);
    // atribut vztahu "od kdy"

// další uzly a hrany
Node uk = graphDb.createNode();
uk.setProperty( "name", "Univerzita Karlova" );
    // uzel s názvem Univerzita Karlova
uk.setProperty( "address", "Ovocný trh 5, Praha 1" );
    // ... a adresou Ovocný trh 5, Praha 1

Relationship rel2 = n1.createRelationshipTo(uk, RelTypes.EMPLOYEE);
    // Irena je zaměstnancem Univerzity Karlovy
rel2.setProperty("since", 2007);
    // ... od roku 2007

Node bd = graphDb.createNode();
bd.setProperty( "name", "Big Data a NoSQL databáze" );
    // uzel s názvem Big Data a NoSQL databáze
```

```

Relationship rel3 = n1.createRelationshipTo(bd, RelTypes.AUTHOR);
    // Irena je autorem knihy Big Data a NoSQL databáze
Relationship rel4 = n2.createRelationshipTo(bd, RelTypes.AUTHOR);
    // Jirka je autorem knihy Big Data a NoSQL databáze

// a obdobným způsobem bychom vytvořili celý graf

// ukončení práce s databází
graphDb.shutdown();

```

Samotný průchod grafem je v Neo4j reprezentován objektem nazývaným *Traversal*, který zahrnuje informace o tom, jakou část grafu a jakým způsobem chceme projít. Konkrétně se skládá z těchto charakteristik:

- Ve kterých uzlech průchod začíná.
- Směr hran a případně typy vztahů, které chceme do průchodu zahrnout.
- Pořadí uzel při průchodu (tj. chceme-li procházet do šířky nebo do hloubky).
- Povolení nebo zakázání opakovaného navštěvování uzel a/nebo hran.
- Jakou část grafu chceme projít (např. celý graf, pouze do určité hloubky apod.).
- Jakou část průchodu grafem chceme zahrnout do výsledku.

Např. v části kódu uvedené v příkladu 9.2 definujeme průchod, který bude grafem procházet do šířky (`breadthFirst`), do výsledku nezahrne počáteční uzel (`excludeStartPosition`) a bude procházet pouze hrany typu `KNOWS` v libovolném směru (`Direction.BOTH`). Parametry, které nenastavíme, budou mít příslušnou implicitní hodnotu, např. opakované navštívění uzel tedy není implicitně povoleno. Chceme-li najít všechny osoby, které znají Jirku přímo nebo přes jinou osobu, specifikovaný průchod aplikujeme na počáteční uzel `n2` z příkladu 9.1 na předchozí straně.

Příklad 9.2: Průchod grafem v Neo4j Java API

```

// definice průchodu
TraversalDescription traversalDescription = Traversal.description()
    .breadthFirst()
    .evaluator( Evaluators.excludeStartPosition() )
    .relationships( RoleRels.KNOWS, Direction.BOTH );

// aplikace průchodu
Traverser traverser = traversalDescription.traverse( n2 ); // začínáme v uzlu Jirka

// výpis výsledku
String output = "";
for ( Path path : traverser )
{
    Node node = path.endNode();
    output += "Uzel "
        + node.getProperty( "name" ) + " nalezen v hloubce: "

```

```
+ ( path.length() ) + "\n";
}
```

Jak je vidět ve druhé části ukázky, výsledek průchodu můžeme zpracovat pomocí třídy `Traverser`. Můžeme iterovat přes jednotlivé cesty (jako v našem příkladu), uzly nebo hrany výsledku. Konkrétně výše uvedený kód by nad grafem z obrázku 9.1 na straně 144 vypsal následující výsledek:

```
Uzel Irena nalezen v hloubce: 1
Uzel Karel nalezen v hloubce: 1
Uzel David nalezen v hloubce: 2
```

9.3.2 Gremlin

Jazyk Gremlin [14] navrhl Marko A. Rodriguez pro účely jednoduchého průchodu ohodnoceného atributového grafu, tj. stejného datového modelu, jaký využívá Neo4j. Jediným rozdílem je, že uzlům i hranám ještě přidává jednoznačný identifikátor. Gremlin je součástí rodiny nástrojů a rozhraní pro práci s grafy s názvem Blueprints,⁴ kterou vyvíjí a udržuje skupina TinkerPop.⁵ Obvyklá charakteristika je, že Blueprints znamená pro grafy totéž, co ODBC (Open Database Connectivity)⁶ pro relační databáze. V tomto smyslu by Gremlin odpovídal jazyku SQL, čili standardnímu nástroji pro práci s grafovými daty.

Uvažujme nový příklad grafu na obrázku 9.2 na následující straně, kde již máme uvedeny identifikátory uzelů i hran a také atributy uzelů i hran ve tvaru (klíč, hodnota). Atributem „name“ je určeno jméno/název vrcholu, tedy např. jméno osoby nebo knihy. Atributem „weight“ máme určenou míru platnosti daného vztahu, tedy např. že David je na 100 % expertem v oblasti „podobnost dat“, ale Jirka pouze na 40 % v oblasti „Big Data“.

Krátká ukázka průchodu grafem v jazyce Gremlin spouštěná z konzolového klienta pro Gremlin (včetně kurzívou vyznačeného informačního textu, který databáze k příkazu vrátí) by mohla vypadat takto: Nejprve do proměnné `g` přiřadíme odkaz na příslušnou databázi:

```
gremlin> g = new Neo4jGraph('/home/db/testDB')
==> neo4jgraph[EmbeddedGraphDatabase[I:\tmp\myDB.graphdb]]
```

Pomocí tečkové notace můžeme volat jednotlivé příkazy jazyka, které reprezentují jednotlivé kroky v průchodu grafem. Konkrétně v následujícím příkladu chceme do proměnné `v` dosadit uzel s identifikátorem 1.

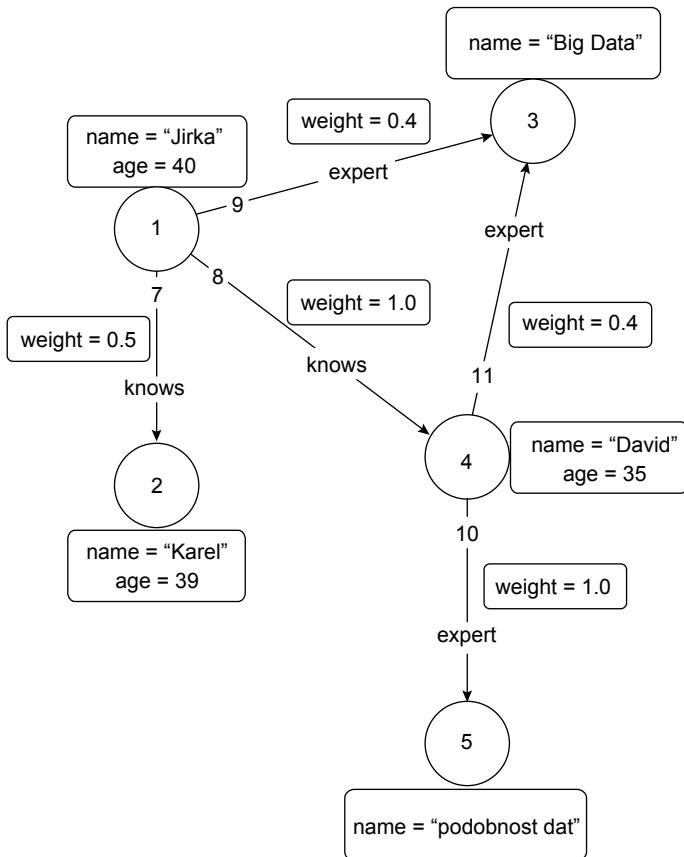
```
gremlin> v = g.v(1)
==> v[1]
```

Ve druhém příkladu pomocí kroku `outE` získáme všechny výstupní hrany uzlu `v`, tedy hrany s identifikátory 7, 9 a 8.

⁴ <http://blueprints.tinkerpop.com>

⁵ <http://www.tinkerpop.com>

⁶ Standardizované softwarové rozhraní pro přístup k databázovým systémům.



Obrázek 9.2: Příklad grafové databáze v Neo4j

```
gremlin> v.outE
==> e[7][1-knows->2]
==> e[9][1-expert->3]
==> e[8][1-knows->4]
```

V dalším příkladu je vidět, že kroky můžeme řetězit, konkrétně nejprve přejdeme pomocí kroku `outE` do výstupních hran uzlu `v` a pomocí kroku `inV` do jejich vstupních uzelů, čímž získáme uzly s identifikátory 2, 3 a 4.

```
gremlin> v.outE.inV
==> v[2]
==> v[3]
==> v[4]
```

V následujících dvou ukázkách demonstrujme, jak můžeme využívat složitější konstrukce, konkrétně iterátory. Oba uvedené příklady dělají totéž – provedou vlastně příkaz `v.out.out`. První z nich ale využívá iteraci přes daný počet kroků, druhý využívá iteraci, dokud je splněna daná podmínka.

V prvním příkazu do proměnné `list` uložíme jednoprvkový seznam uzelů tvořený pouze uzlem `v`. Následující příkaz provádí iteraci pro proměnnou `i` od 1 do 2, přičemž v každém kroku vezme aktuální obsah proměnné `list` a na každý její

prvek aplikuje krok `out`, který vrací uzly, do nichž vede hrana. Tyto uzly pak sloučí do seznamu, jenž se stává novou hodnotou proměnné `list`.

```
gremlin> list = [v]
gremlin> for(i in 1..2)
    list = list._().out.collect{it}
gremlin> list
==> v[5]
==> v[3]
```

Ve druhém příkladu využíváme podmínku `it.loops < 3`, která říká, že se má daný cyklus zopakovat méně než 3x. Proměnná `it` reprezentuje iterátor a číslo 1 říká, kolik kroků zpět je třeba se vrátit. V našem případě máme tedy vzít ten poslední, což je na začátku uzel `v`, v dalších iteracích uzly, do nichž se dostaneme krokem `out`.

```
gremlin> v.out.loop(1){it.loops<3}
==> v[5]
==> v[3]
```

Nyní ukážeme, jak je možné pracovat s atributy uzlů a na základě jejich hodnot procházet grafem. Opět nejprve do proměnné uložíme konkrétní uzel, z něhož budeme vycházet.

```
gremlin> v = g.v(1)
==> v[1]
```

V následujícím příkazu zjištujeme hodnotu jeho atributu `name`.

```
gremlin> v.name
==> Jirka
```

Další příkaz z uzlu `v` přechází krokem `outE` s parametrem `knows`, který říká, že nás zajímají pouze výstupní hrany tohoto typu. Dále pokračujeme krokem `inV` do jejich vstupních uzlů, ale ty jsou funkcí `filter` omezeny pouze na ty, které splňují uvedenou podmínu `it.age < 36`, a od nich vrátíme hodnotu atributu `name`.

```
gremlin> v.outE('knows').inV.filter{it.age < 36}.name
==> David
```

Podobně můžeme v posledním příkazu pomocí funkce `matches` vybrat jen ty uzly, v jejichž atributu `name` je hodnota odpovídající regulárnímu výrazu `d.{4}|D.{4}`. Také zde vidíme možnost uložit aktuální výsledek do proměnné `x` a schopnost se do ní vrátit prostřednictvím funkce `back`.

```
gremlin> v.out('knows').filter{it.age < 40}.as('x').name.
filter{it.matches('d.{4}|D.{4}')}.back('x').age
==> 35
```

A na závěr ještě uvedme příklady metod, které umožňují modifikovat graf. První příkaz demonstruje možnost nastavit (resp. změnit) hodnotu atributu `note`.

```
gremlin> g.v(1).note = "XML guru"
==> XML guru
```

Druhý příkaz přidává do databáze nový uzel s jediným atributem `name` a tento uloží do proměnné `v1`.

```
gremlin> v1 = g.addVertex([name: "Irena"])
=> v[7]
```

Třetí příkaz vytvoří hranu typu `knows` mezi uzlem v proměnné `v1` a uzlem s identifikátorem 1.

```
gremlin> g.addEdge(v1, g.v(1), 'knows')
=> e[7][7->1]
```

9.3.3 Cypher

Jak jsme viděli, Gremlin je založen na myšlence popsat průchod grafem pomocí jednotlivých kroků, popř. jejich iterováním. Naproti tomu jazyk Cypher [9], který byl navržen přímo pro Neo4j, je deklarativní. To znamená, že nepopisujeme, jak chceme grafem projít, ale co chceme průchodem získat. Cílem autorů bylo vytvořit jazyk, který je jednoduchý a uživatelsky příjemný. Inspirovali se tudíž ve známých a ověřených jazycích SQL pro relační data a SPARQL [41] pro RDF data.

Základními stavebními prvky jazyka jsou jednotlivé klauzule, a to například:

- **START**: Určení počátečních uzlů grafu.
- **MATCH**: Vzor navázaný na počáteční uzly, kterému musí požadovaný graf odpovídat.
- **WHERE**: Filtrací kritéria.
- **RETURN**: Návratové hodnoty.
- **CREATE**: Vytváření uzlů a hran.
- **DELETE**: Mazání uzlů, hran a atributů.
- **SET**: Nastavení/změna hodnot atributů uzlů/hran.
- **FOREACH**: Provedení změn nad všemi prvky daného seznamu.
- **ORDER BY**: Seřazení výsledku podle zvoleného kritéria.

Například v následující ukázce vidíme příkaz pro vytvoření jednoho uzlu bez jakýchkoli atributů (včetně kurzívou vyznačeného informačního textu, který databáze k příkazu vrátí):

```
CREATE n
(empty result)
Nodes created: 1
```

Nebo bychom takto vytvořili uzel s jedním atributem a současně bychom vytvořený výsledek vrátili na výstup (přičemž vidíme, že námi vytvořený uzel má automaticky přiřazený identifikátor, v tomto případě s hodnotou 2):

```
CREATE (a {name : 'Jaroslav'})
RETURN a
```

```
a
Node[2]{name:"Jaroslav"}
1 row
Nodes created: 1
Properties set: 1
```

Podobně bychom mohli vytvořit uzel s více atributy (tentokrát nás ale výsledek opět nezajímá, nepoužíváme tedy klauzuli RETURN):

```
CREATE n = {name : 'Jaroslav', title : 'professor'}
```

```
(empty result)
Nodes created: 1
Properties set: 2
```

A můžeme samozřejmě vytvářet také hrany. V následujícím příkladu nejprve do proměnných **a** a **b** přiřadíme již existující uzly s identifikátory 1 a 2. Pak mezi nimi vytvoříme hranu typu FRIEND a výsledek pro kontrolu vrátíme.

```
START a = node(1), b = node(2)
CREATE a-[r:FRIEND]->b
RETURN r
```

```
r
:FRIEND[1] {}
1 row
Relationships created: 1
```

Také u hrany bychom při vytváření mohli přímo specifikovat její atribut:

```
START a = node(1), b = node(2)
CREATE a-[r:FRIEND {name : a.name + '<->' + b.name }]->b
RETURN r
```

```
r
:FRIEND[1] {name:"Jaroslav<->Irena"}
1 row
Relationships created: 1
Properties set: 1
```

A dokonce můžeme vytvářet celé cesty, jak ukazuje následující příklad, v němž jsou vytvořeny všechny části specifikovaného vzoru, které zatím neexistují.

```
CREATE p =
(jarda {name:'Jaroslav'})-[:WORKS_AT]->mff<-[:WORKS_AT]-(irena {name:'Irena'})
RETURN p
```

```
p
[Node[4]{name:"Jaroslav"},:WORKS_AT[2] {},Node[5]{},:WORKS_AT[3] ▶
```

```
{},Node[6]{name:"Irena"}]
1 row
Nodes created: 3
Relationships created: 2
Properties set: 2
```

Samozřejmě můžeme atribut nastavit (nebo změnit) pro již existující uzel:

```
START n = node(2)
SET n.surname = 'Pokorný'
RETURN n

n
Node[2]{name:"Jaroslav",surname:"Pokorný"}
1 row
Properties set: 1
```

A také můžeme atribut smazat – právě pomocí již dříve zmíněné hodnoty `null`:

```
START n = node(2)
SET n.name = null
RETURN n

n
Node[2]{surname:"Pokorný"}
1 row
Properties set: 1
```

Stejně tak můžeme mazat jednotlivé uzly, jak je vidět v prvním příkazu níže, nebo také celé množiny uzlů a hran, jak demonstruje druhý příkaz:

```
START n = node(4)
DELETE n

(empty result)
Nodes deleted: 1

START n = node(3)
MATCH n-[r]-()
DELETE n, r

(empty result)
Nodes deleted: 1
Relationships deleted: 2
```

Klauzule `FOREACH` umožňuje provádět zvolenou editační operaci iterativně přes množinu uzlů. V následujícím příkladu předpokládáme, že z uzlu s identifikátorem 2 (v proměnné `begin`) do uzlu s identifikátorem 1 (v proměnné `end`) vede cesta délky 3. Iterace nastaví atribut `marked` na hodnotu `true` všem uzlům na této cestě (v proměnné `p`), tedy celkem čtyřem uzlům.

```
START begin = node(2), end = node(1)
MATCH p = begin -[*]-> end
FOREACH (n in nodes(p) : SET n.marked = true)
```

```
(empty result)
Properties set: 4
```

A konečně klasické dotazování demonstrujeme na následujícím příkladu, který opět pracuje s grafem na obrázku 9.2 na straně 150. Následující dotaz zjišťuje, na jaké oblasti jsou experty osoby, které zná Jirka. Tedy všechny uzly, do nichž vede cesta délky 2 přes hrany typu `knows` a `expert` z uzlu s identifikátorem 1.

```
START jirka = node(1)
MATCH jirka-[:knows]->()-[:expert]->ex
RETURN jirka, ex
```

jirka	ex
-------	----

Node[1]{name:"Jirka"} Node[3]{name:"Big Data"}	
Node[1]{name:"Jirka"} Node[5]{name:"podobnost dat"}	

Nad stejným grafem bychom mohli vyhodnotit např. následující dotaz na všechny přátele uzelů s identifikátory 1, 2 a 4 starší 30 let, jejichž jméno začíná na D.

```
START user=node(1,2,4)
MATCH user-[:knows]->friend
WHERE friend.age > 30 AND friend.name =~ 'D.*'
RETURN user, friend.name
```

user	friend.name
------	-------------

Node[1]{name:"Jirka"} "David"	
---------------------------------	--

Podobně jako v SQL umožňuje klauzule `ORDER BY` seřadit výsledky podle zadанého kritéria:

```
START n = node(1,2,4)
RETURN n
ORDER BY n.name
```

n	
---	--

Node[4]{name->"David"}	
Node[1]{name->"Jirka"}	
Node[2]{name->"Karel"}	

V jazyce Cypher najdeme také obdobu funkce `count()`, která spočítá počet výsledků dotazu, jak vidíme níže.

```
START n = node(1)
MATCH (n)-->(x)
RETURN n, count(*)
```

n	count(*)
---	----------

Node[1]{name:"Jirka"} 3	
---------------------------	--

A dokonce máme k dispozici zjednodušenou obdobu klauzule `GROUP BY`, jež demonstruje následující příklad, který pro hrany různých typů vycházející z uzlu s identifikátorem 1 spočítá jejich počty.

```

START n = node(1)
MATCH (n)-[r]->()
RETURN type(r), count(*)

TYPE(r)  / count(*)
-----
"knows" / 2
"expert" / 1

```

Jazyk Cypher je ovšem ještě mnohem bohatší. Nabízí např. agregační funkce (`sum`, `avg`, `max`, `min`, ...), klauzule `LIMIT n` a `SKIP n` umožňující vrátit pouze prvních `n` výsledků nebo naopak je přeskočit, kvantifikátory (`ANY`, `ALL`), velké množství funkcí a operátorů a mnoho dalšího.

9.4 Pokročilé rysy Neo4j

Nyní se zaměříme na vlastnosti systému Neo4j z hlediska jeho architektury a klasických databázových vlastností. Oproti ostatním typům NoSQL databází nabízí Neo4j překvapivě vlastnosti ACID. Nicméně, jak už jsme naznačili, není to „zadarmo“. V Neo4j nalezneme také podporu vysoké dostupnosti (*high availability*), indexů a dalších zajímavých principů.

9.4.1 Neo4j HA

Neo4j HA (*Neo4j High Availability*) je speciální část systému, která je založena na klasické master-slave architektuře. Ta umožňuje:

- nakonfigurovat několik uzlů typu slave tak, aby byly přesnými replikami uzlu master, a tudíž zajišťovaly odolnost vůči výpadkům,
- rozložit zátěž na více uzlů, tedy horizontálně škálovatelné čtení dat.

Jinými slovy řečeno, Neo4j může pracovat buď na jediném serveru, nebo na clusteru uzlů. Z hlediska aplikací pracujících s databází se nic nezmění, jelikož obě verze využívají stejná rozhraní. Pokud se rozhodneme využít distribuovanou verzi databáze, budeme mít transakce stále atomické, konzistentní a trvalé, ale do uzlů typu slave budou propagovány pouze občasné. Přesněji řečeno, pokud dojde k zápisu na master, je změna do uzlů typu slave propagována občasné, zatímco pokud dojte k zápisu na uzlu slave, je tato změna propagována na uzel master okamžitě (a na obou musí proběhnout úspěšně).

Každý uzel v clusteru má stejnou logiku, která mu umožňuje koordinaci s jinými uzly. Po startu se instance databáze nejprve pokusí připojit ke clusteru, který má uveden v konfiguraci. Pokud se jí to podaří, stává se uzlem typu slave. Pokud ne, stává se uzlem typu master. Pokud v běžícím clusteru dojde k výpadku, nastane jedna z následujících možností:

- Pokud dojde k výpadku uzlu typu slave, ostatní uzly tuto informaci zjistí díky ztrátě komunikace a master s ním data dále nesynchronizuje.

- Pokud dojde k výpadku uzlu typu master, uzly typu slave zvolí nový uzel master.

Pokud dojde k opětovnému navázání komunikace, tak se původní uzel master chová jako slave, tj. synchronizuje svoje data s aktuální verzí na serveru, stejně jako kterýkoli původní uzel slave.

Jak je z uvedeného popisu patrné, jelikož má každý uzel uloženou (více či méně aktuální) kopii celé grafové databáze, existují omezení na velikost grafu, který je Neo4j schopen uložit. (Dalším omezením jsou samozřejmě možnosti daného serveru, na němž je databáze instalována.) V tabulce 9.1 jsou uvedeny maximální přípustné počty jednotlivých entit grafu.

Tabulka 9.1: Maximální velikost databáze

Entity	Maximální počet
uzly	2^{35}
hrany	2^{35}
atributy	2^{36} až 2^{38} , v závislosti na datových typech
typy vztahů	$2^{15} \sim 32\,000$

Jak je vidět, hodnoty to jsou poměrně velkorysé, nicméně i tak existuje množství aplikací, pro něž jsou nedostačující.

9.4.2 Transakce

Veškeré operace editace grafu (tj. vytváření, mazání a modifikace jakýchkoli dat), tedy operace zápisu, musí probíhat v transakci (v opačném případě Neo4j vyhlásí chybu – např. výjimku `NotInTransactionException`), přičemž systém nabízí také podporu hnízděných transakcí (viz kapitola 12). Pokud dojde ke zrušení některé vnitřní transakce, je zrušena celá transakční hierarchie. Operace tedy probíhají v následujících krocích:

1. Začátek transakce.
2. Operace nad grafem zahrnující zápisy.
3. Označení transakce jako úspěšné, nebo neúspěšné.
4. Ukončení transakce, kdy dojde k uvolnění paměti i zámků.

Tento princip by se např. v jazyce Java realizoval takto:

Příklad 9.3: Transakce v Neo4j

```
Transaction tx = graphDb.beginTx();
try
{
    // editace grafu
    // ...
    tx.success();    // v operaci close() bude proveden COMMIT
```

```

    }
    catch (Exception e)
    {
        tx.failure(); // v operaci close() bude proveden ROLLBACK
    }
    finally
    {
        tx.close();
    }
}

```

Jelikož veškeré operace zápisu probíhají v paměti, je třeba velké editace grafu rozdělit do menších celků. Jakákoli editační operace pak implicitně využívá zámky. Konkrétně:

- Přidání/změna/smazání atributu uzlu/hrany aplikuje zámek pro zápis na příslušném uzlu/hraně.
- Vytvoření/smazání uzlu aplikuje zámek na příslušný uzel.
- Vytvoření/smazání hrany aplikuje zámek na hranu a oba její uzly.

Důsledkem využití zámků může být samozřejmě v určitých případech deadlock. Taková situace je nicméně systémem vždy detekována a uživatel je informován prostřednictvím chybového hlášení, např. výjimky.

Operace smazání má navíc speciální sémantiku:

- Smazání uzlu/hrany znamená smazání všech příslušných atributů.
- Není možné smazat uzel, do/z něhož vedou hrany. Dojde k vyvolání chyby (výjimky).
- Pokus o editaci uzlu/hrany smazané v aktuální transakci, která ještě nebyla potvrzena, způsobí chybu (výjimku). Nicméně odkaz na takový uzel/hranu lze získat a s tímto odkazem je možné pracovat až do operace **COMMIT**.

Operace čtení vždy přečte poslední hodnotu, pro kterou byl úspěšně proveden **COMMIT**. V implicitním nastavení se operace nikdy nezablokuje, jelikož neaplikuje žádné zámky. Důsledkem tohoto chování je ovšem možnost výskytu problému neopakovatelného čtení, kdy dvě po sobě jdoucí operace čtení, které se vyskytnou v jedné transakci, vracejí různou hodnotu (viz sekce 3.2.1). Pokud nám toto chování v dané aplikaci nevyhovuje, můžeme explicitně aplikovat zámky také pro čtení. Přesněji řečeno, v Neo4j máme na výběr mezi implicitní úrovní izolace transakcí **read committed** a vyššími úrovněmi zajištěnými pomocí explicitních zámků (viz sekce 3.2.1.1).

9.4.3 Indexy

Z pohledu Neo4j je index speciální datová struktura sloužící pro efektivnější vyhledávání uzelů, respektive hrani grafu. Každý index má unikátní, uživatelem zadané jméno a umožňuje asociovat libovolné množství dvojic (klíč, hodnota) s libovolným množstvím indexovaných entit, tj. uzelů nebo hrani. Typicky je to jedna ku jedné,

nicméně např. jeden uzel/hrana může být asociována s více dvojicemi (klíč, hodnota) se stejným klíčem. Tudíž chceme-li hodnotu modifikovat, musíme tu starou nejprve smazat, jinak by pouze přibyla nová dvojice (klíč, hodnota).

V následujícím příkladu, částečně převzatém z dokumentace Neo4j, ukážeme základní práci s indexy, tedy vytváření, kontrolu existence a smazání. Všimněme si, že Neo4j rozlišuje dva typy indexů (zvlášť pro uzly a zvlášť pro hrany), kterým odpovídají dvě množiny odpovídajících operací, např. `existsForNodes()` a `existsForRelationships()`.

Příklad 9.4: Vytváření indexů v Neo4j Java API

```
// vytvoření databáze a správce indexů
graphDb = new
GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);
IndexManager index = graphDb.index();

// kontrola existence indexu
boolean indexExists = index.existsForNodes("zaměstnanci");

// vytvoření indexů
Index<Node> employees = index.forNodes("zaměstnanci");
Index<Node> projects = index.forNodes("projekty");
RelationshipIndex roles = index.forRelationships("role");
RelationshipIndex test = index.forRelationships("test");

// smazání indexu
test.delete();
```

Přidávání uzelů/hran do indexu vlastně odpovídá jeho asociaci s alespoň jednou dvojicí (klíč, hodnota):

Příklad 9.5: Naplnění indexů v Neo4j Java API

```
Node emp1 = graphDb.createNode();
emp1.setProperty("name", "Irena Holubová");

Node emp2 = graphDb.createNode();
emp2.setProperty("name", "Jaroslav Pokorný");

// přidání uzlu do indexu = asociace s dvojicí (klíč, hodnota)
employees.add(emp1, "name", emp1.getProperty("name"));

// asociace jednoho uzlu s více dvojicemi (klíč, hodnota)
employees.add(emp2, "name", emp2.getProperty("name"));
employees.add(emp2, "name", "Jarda");

Node project1 = graphDb.createNode();
project1.setProperty("title", "GAČR GA201/09/0990");
project1.setProperty("year", 2009);
projects.add(project1, "title", project1.getProperty("title"));
projects.add(project1, "year", project1.getProperty("year"));

Relationship role1 = emp1.createRelationshipTo(project1, WORKS_ON);
role1.setProperty("name", "člen týmu");
Relationship role2 = emp2.createRelationshipTo(project1, WORKS_ON);
```

```
role2.setProperty("name", "vedoucí");

// přidání hran do indexu
roles.add(role1, "name", role1.getProperty("name"));
roles.add(role2, "name", role2.getProperty("name"));
```

Jak jsme již naznačili, vzhledem k možnosti asociovat s jedním uzlem/hranou více dvojic (klíč, hodnota) máme několik možností pro mazání prvků indexu:

Příklad 9.6: Mazání položek indexu v Neo4j Java API

```
// smazání všech výskytů uzlu emp2 v indexu employees
employees.remove( emp2 );

// smazání všech položek indexu, které se týkají uzlu emp2 a mají klíč "name"
employees.remove( emp2, "name" );

// smazání položky indexu pro uzel emp2, klíč "name" a hodnotu "Jarda"
employees.remove( emp2, "name", "Jarda" );
```

A chceme-li příslušnou dvojici (klíč, hodnota) modifikovat, musíme ji smazat a znova vytvořit s novou hodnotou:

Příklad 9.7: Modifikace položek indexů v Neo4j Java API

```
Node emp3 = graphDb.createNode();
emp3.setProperty("name", "Nečaský");
employees.add(emp3, "name", emp3.getProperty("name"));

// modifikace položky indexu
employees.remove( emp3, "name", emp3.getProperty("name") );
emp3.setProperty ("name", "Martin Nečaský");
employees.add (emp3, "name", emp3.getProperty("name"));
```

Vyhledávání proložek indexu jsme již viděli, nicméně možností je více. Základní operací je funkce `get()`:

Příklad 9.8: Vyhledávání položek indexů v Neo4j Java API – funkce `get()`

```
// nalezení jediného uzlu/hrany
IndexHits<Node> hits = employees.get("name", "Irena Holubová");
Node emp1 = hits.getSingle();

Relationship leader = roles.get("name", "vedoucí").getSingle();
Node emp     = leader.getStartNode();
Node project = leader.getEndNode();

// iterace přes více výsledků
for ( Relationship role : roles.get("name", "člen týmu") ) {
    Node emp = role.getStartNode();
    // ...
}
```

Druhou možností je bohatší funkce `query()`, která umožňuje nezadávat přesné hodnoty, ale také využívat zástupné symboly * (jakýkoliv počet libovolných znaků) a ? (jakýkoliv jeden znak):

Příklad 9.9: Vyhledávání položek indexů v Neo4j Java API – funkce query()

```
for ( Node a : employees.query("name", "*ý") ) {
    // vrátí zaměstnance "Jaroslav Pokorný" a "Martin Nečaský"
}

for (Node m : projects.query("title:*GA201/09/0990* AND year:2009")) {
    // vrátí projekt "GAČR GA201/09/0990" z roku 2009
}
```

Pro vyhledávání hran můžeme také využít možnost vyhledávat přes jejich uzly, opět pomocí přesných hodnot nebo regulárních výrazů:

Příklad 9.10: Vyhledávání hran v indexu pomocí Neo4j Java API

```
// nalezení vztahu podle počátečního uzlu (presný výběr)
IndexHits<Relationship> hits = roles.get("name", "vedoucí", emp2, null);
Relationship rel1 = hits.iterator().next();
hits.close();

// nalezení vztahu podle koncového uzlu (prostřednictvím regulárního výrazu)
IndexHits<Relationship> hits = roles.query( "name", "člen*", null, project1 );
Relationship rel2 = hits.iterator().next();
hits.close();
```

9.4.3.1 Automatická indexace

Kromě explicitně vytvořených indexů máme možnost využít také indexy implicitní – konkrétně jeden index pro uzly a jeden pro hrany, přičemž oba automaticky indexují vybrané atributy uzelů/hran. V implicitním nastavení databáze není automatická indexace povolena, je tedy třeba tuto funkci povolit. Další práce s těmito indexy je obdobná jako u indexů explicitně vytvářených.

Povolení automatické indexace před spuštěním databáze je uvedeno v příkladu 9.11, v němž povolujeme jak indexaci uzelů (`node_auto_indexing`), tak hran (`relationship_auto_indexing`). Pro uzly budeme indexovat atributy `atributUzlu1` a `atributUzlu2`, pro hrany atribut `atributHrany1`.

Příklad 9.11: Neo4j automatická indexace – nastavení před spuštěním databáze

```
GraphDatabaseService graphDb = new GraphDatabaseFactory().
    newEmbeddedDatabaseBuilder(storeDirectory).
    setConfig(GraphDatabaseSettings.node_keys_indexable, "atributUzlu1,atributUzlu2").
    setConfig(GraphDatabaseSettings.relationship_keys_indexable, "atributHrany1").
    setConfig(GraphDatabaseSettings.node_auto_indexing, "true").
    setConfig(GraphDatabaseSettings.relationship_auto_indexing, "true").
    newGraphDatabase();
```

Stejně nastavení můžeme udělat až později, po spuštění databáze:

Příklad 9.12: Neo4j automatická indexace – nastavení po spuštění databáze

```
// start bez speciální konfigurace
GraphDatabaseService graphDb =
    new GraphDatabaseFactory().newEmbeddedDatabase(storeDirectory);

// parametrizace pro automatickou indexaci uzelů
```

```
AutoIndexer<Node> nodeAutoIndexer =
    graphDb.index().getNodeAutoIndexer();
nodeAutoIndexer.startAutoIndexingProperty("atributUzlu1");
nodeAutoIndexer.startAutoIndexingProperty("atributUzlu2");
nodeAutoIndexer.setEnabled(true);

// parametrizace pro automatickou indexaci hran
AutoIndexer<Relationship> relAutoIndexer =
    graphDb.index().getRelationshipAutoIndexer();
relAutoIndexer.startAutoIndexingProperty("atributHrany1");
relAutoIndexer.setEnabled(true);
```

Práce s automatickým indexem je obdobná jako s explicitně vytvořenými indexy. Výhodou je samozřejmě automatická indexace položek:

Příklad 9.13: Neo4j automatická indexace – práce s indexem

```
node1 = graphDb.createNode();
node2 = graphDb.createNode();
rel = node1.createRelationshipTo(node2, DynamicRelationshipType.withName("DYNAMIC"));

// vytvoření indexovaných i neindexovaných položek
node1.setProperty("atributUzlu1", "aaa");
node2.setProperty("atributUzlu2", "bbb");
node1.setProperty("atributUzlu3", "ccc");
rel.setProperty("atributHrany1", "ddd");
rel.setProperty("atributHrany2", "eee");

ReadableIndex<Node> autoNodeIndex = graphDb.index().getNodeAutoIndexer().getAutoIndex();

// test zaindexování
assertEquals(node1, autoNodeIndex.get("atributUzlu1", "aaa").getSingle());
assertEquals(node2, autoNodeIndex.get("atributUzlu2", "bbb").getSingle());

// test nezaindexování
assertFalse(autoNodeIndex.get("atributUzlu3", "ccc").hasNext());
```

9.5 Další grafové databáze

Databáze Neo4j není pochopitelně jediným systém tohoto typu na trhu. Aktuální přehled a především srovnání jejich popularity (vypočtené na základě míry diskutovanosti systému na různých fórech, množství pracovních nabídek apod.) je možné nalézt např. na stránkách DB-Engines.com.⁷ V této sekci krátce představíme další zástupce grafových databázových systémů a jejich hlavní charakteristiky.

⁷ <http://db-engines.com/en/ranking/graph+dbms>

9.5.1 Sparksee

Databázový systém Sparksee,⁸ dříve známý pod jménem DEX, je komerční systém⁹ implementovaný v jazyce Java s jádrem v jazyce C++, jehož první verze pochází z roku 2007. Cílem autorů bylo vytvořit škálovatelné řešení, které umožní pracovat s rozsáhlými grafovými daty. Datový model databáze Sparksee odpovídá orientovanému multigrafu. S daty v databázi je možné pracovat prostřednictvím API v jazyce Java, C++ nebo .NET, podporuje rozhraní Blueprints a vzdálený přístup prostřednictvím REST metod.

Sparksee nabízí částečnou podporu ACID transakcí, která je označována jako aCiD, jelikož izolovanost a atomicita není zaručena vždy. Transakce běží v módu *N čtenářů a 1 zapisující*, tedy souběžně smí běžet libovolné množství transakcí, ale pouze jedna z nich smí zapisovat.

Sparksee má dále rozšířené nazývané SparkseeHA (Sparksee High Availability),¹⁰ které umožňuje horizontálně škálovatelné řešení pro čtení dat, přičemž zápisy jsou vzhledem k architektuře master-slave stále řešeny centrálně, tj. zápis na některý z uzlů slave je okamžitě synchronizován s uzlem master.

9.5.2 InfiniteGraph

Databázový systém InfiniteGraph¹¹ je rovněž komerční systém implementovaný v jazyce Java s jádrem v jazyce C++. První verze byla vydána v roce 2009. InfiniteGraph pracuje s orientovanými multigrafy a nabízí k nim přístup přes API v různých programovacích jazycích jako např. Java, C++, C# nebo Python. Stejně tak podporuje rozhraní Blueprints včetně jazyka Gremlin

Jak už název napovídá, hlavními rysy tohoto systému jsou škálovatelnost, paralelní zpracování a distribuce grafu. Systém nabízí plný ACID model transakcí, který je ale možné pro zvýšení efektivity operací uvolnit. Jelikož jako back-end slouží objektově orientovaný systém Objectivity/DB,¹² využívá právě jeho vlastnosti distribuce, škálování a replikace.

9.5.3 OrientDB

Databázový systém OrientDB¹³ je specifickým databázovým systémem, který je především grafovou databází, ale současně také databází typu klíč-hodnota a databází dokumentovou. Přesněji řečeno, grafová databáze využívá právě dokumentově orientované úložiště jako svůj back-end. Vzhledem k tomu je také možné graf distribuovat na více uzlů. Systém je implementován v jazyce Java a od své první verze z roku 2010 je poměrně populární.

⁸ <http://sparsity-technologies.com/> (<http://sparsity-technologies.com>)

⁹ S akademickou verzí zdarma limitovanou na 1 000 000 uzlů a libovolné množství hran.

¹⁰ <http://sparsity-technologies.com/UserManual/HighAvailability.html>

¹¹ <http://www.objectivity.com/infinitegraph>

¹² <http://www.objectivity.com/products/objectivitydb/>

¹³ <http://orientdb.com/> (<http://orientdb.com>)

Databáze ukládá atributové grafy a nabízí API v mnoha programovacích jazycích, které zahrnují rozhraní *Traverser* v jazyce Java, REST rozhraní pro vzdálený přístup, speciální jazyk *Extended SQL*, který syntaxí připomíná jazyk SQL, rozhraní Blueprints a další.

Transakce mají plnou podporu ACID vlastností díky využití MVCC (viz sekce 12.6), kterou využívá velké množství (nejen NoSQL) databázových systémů.

Systém OrientDB je možné distribuovat na cluster uzlů díky využití systému Hazelcast¹⁴ pro správu distribuce dat, který umožňuje peer-to-peer replikaci dat. Dále je možné nastavit, zda požadujeme okamžitou propagaci změn, nebo pouze občasnou.

9.5.4 Titan

Poslední databázový systém, který představíme, se jmenuje Titan.¹⁵ Jedná se o novější databázi z roku 2012 implementovanou v jazyce Java. Cílem autorů bylo vytvořit systém, který umožňuje škálovatelné zpracování grafů, včetně jejich distribuce. K atributovému grafu, který Titan umožňuje ukládat, lze přistupovat prostřednictvím rozhraní REST nebo Blueprints včetně jazyka Gremlin Pro indexaci uzlů je možné využít nástroje Apache Lucene nebo Elasticsearch které umožňují provádět efektivní fulltextové vyhledávání, hledání podle rozsahu hodnot nebo práci s geografickými souřadnicemi.

Jako back-end podporuje Titan tři databázové systémy typu klíč-hodnota, resp. sloupcové, které nabízejí různé vlastnosti transakčního zpracování a škálovatelnosti. Uživatel může zvolit jeden ze systémů Cassandra, HBase nebo Berkeley DB. Berkeley DB má sice limitovanou horizontální škálovatelnost, ale je nejfektivnější pro využití systému v nedistribuované verzi. Zbylé dva naopak nabízejí nativní podporu pro distribuovaná řešení. Samotný graf je uložen ve formě (případně distribuovaného) seznamu sousedů (viz sekce 13.3).

9.6 RDF databáze

V kapitole o grafových databázích nelze opomenout také pravděpodobně neznámejší typ databází určených pro grafová data reprezentovaná ve formátu RDF (Resource Description Framework) [35], který jsme krátce představili v sekci 2.4. RDF data odpovídají trojicím subjekt-predikát-objekt. Propojením těchto trojic získáváme graf, nad nímž se můžeme dotazovat pomocí jazyka SPARQL (který v kontextu jiných dotazovacích jazyků krátce představíme v sekci 13.4.3.4).

Myšlenka formátu RDF pochází přibližně z roku 1996, přičemž první specifikace konsorcia W3C byla publikována v roce 1999. Jazyk SPARQL byl poprvé specifikován až v roce 2008. V současné době existuje velké množství databázových systémů, které ukládají data ve formátu RDF a umožňují dotazování pomocí jazyka

¹⁴ <http://www.hazelcast.com/> (<http://www.hazelcast.com>)

¹⁵ <http://thinkaurelius.github.io/titan/>

SPARQL. Nejznámějšími reprezentanty jsou pravděpodobně systémy Virtuoso,¹⁶ 4store,¹⁷ Jena TDB,¹⁸ Sesame,¹⁹ AllegroGraph²⁰ a další.

Hlavní sírou RDF dat je možnost vzájemného propojování souvisejících dat z různých datových zdrojů. K tomu slouží tzv. *SPARQL endpoints*, tedy webové přístupové body k RDF úložištím, jejichž prostřednictvím je možné se pomocí jazyka SPARQL nad daty z daného úložiště dotazovat a také je spojovat s daty z jiných úložišť. Takové přístupové body provozují např. DBpedia,²¹ British National Bibliography,²² WordNet²³ a mnohé další instituce a firmy (viz např. [43]).

Otázkou tedy je, zda RDF databáze patří do kategorie NoSQL databází, resp. zda umí pracovat s Big Data. Z hlediska klasifikace jsou obvykle řazeny do nepravých NoSQL databází (viz sekce 1.2.1) a jak je vidět výše, myšlenka RDF je obecně starší než problematika Big Data a NoSQL databází. Z hlediska práce s Big Data je ovšem odpověď složitější. Uvážíme-li např. celý Linked Open Data cloud²⁴, tedy celou datovou množinu veřejně dostupných a případně i vzájemně propojených RDF dat, která neustále narůstá, pak se o Big Data jedná, resp. z jistého pohledu jednat může. Na druhou stranu, každá datová množina, která je jeho součástí, je uložena ve vlastním úložišti a je přístupná přes vlastní koncový bod. Máme tedy opačnou situaci, než jakou jsme řešili doposud – datové množiny přijatelné velikosti jsou uloženy lokálně (dalo by se říci, že jsou od počátku implicitně distribuovaný) a teprve jejich následným vzájemným propojením případně získáváme Big Data. Jedná se tedy vlastně o jednu z myšlenek sémantického webu, jakousi globální kolekci databází, k níž nám jazyk SPARQL umožňuje přistupovat jako k jediné velké databázi. V oblasti distribuovaných databází říkáme, že taková databáze vzniká *zdola-nahoru*, na rozdíl od pravých NoSQL databází, které naopak vznikají *shora-dolů*.

9.7 Srovnání úložišť pro grafy

Na závěr této kapitoly ještě ukážeme, jak se liší uložení grafu v různých typech databázových systémů, abychom si ujasnili výhody a nevýhody těchto možností [6].

Nejprve se podívejme, jak bychom uložili graf do relační databáze. Jak je znázorněno na obrázku 9.3 na následující straně, využili bychom množinu tabulek (v našem případě tří pro tři různé typy uzlů – A, B, a C) a princip klíčů a cizích klíčů, což je na tomtéž obrázku znázorněno pomocí spojnic mezi uzly. Pokud by se jednalo o téměř úplný graf, museli bychom mít navíc pro uložení hran další speciální tabulky. Jak je z tohoto řešení patrné, každý dotaz odpovídající průchodu grafem znamená velké množství operací spojení tabulek. Stejně tak pokud by nám

¹⁶ <http://virtuoso.openlinksw.com>

¹⁷ <http://4store.org>

¹⁸ <http://jena.apache.org/documentation/tdb/index.html>

¹⁹ <http://rdf4j.org>

²⁰ <http://www.franz.com/agraph/allegrograph/>

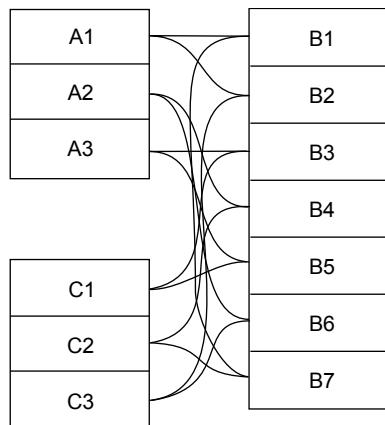
²¹ <http://dbpedia.org/sparql>

²² <http://bnb.data.bl.uk/sparql>

²³ <http://wordnet.rkbexplorer.com/sparql/>

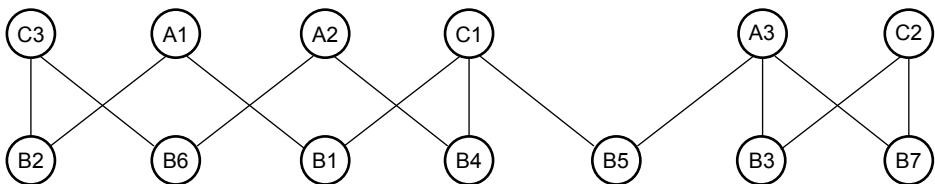
²⁴ <http://lod-cloud.net>

přibyly uzly typu D, E a F, museli bychom vytvořit nové tabulky a příslušným způsobem upravit dotazy.



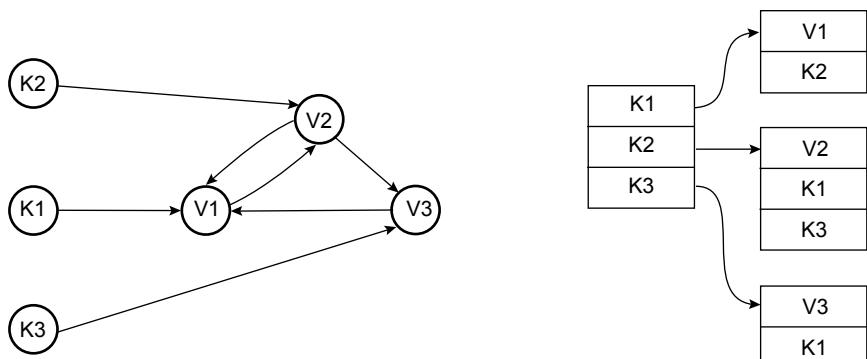
Obrázek 9.3: Uložení grafu v relační databázi

Uložení stejného grafu do grafové databáze je znázorněno na obrázku 9.4. Jak je vidět, nemusíme různé typy uzlů agregovat do tabulek ani vymýšlet komplikované uložení pro hrany. Průchod grafem je optimalizován právě pro tuto datovou strukturu, takže nevyžaduje množství operací spojení. Uzly i hrany můžeme libovolně přidávat i ubírat bez nutnosti měnit strukturu úložiště nebo operací.



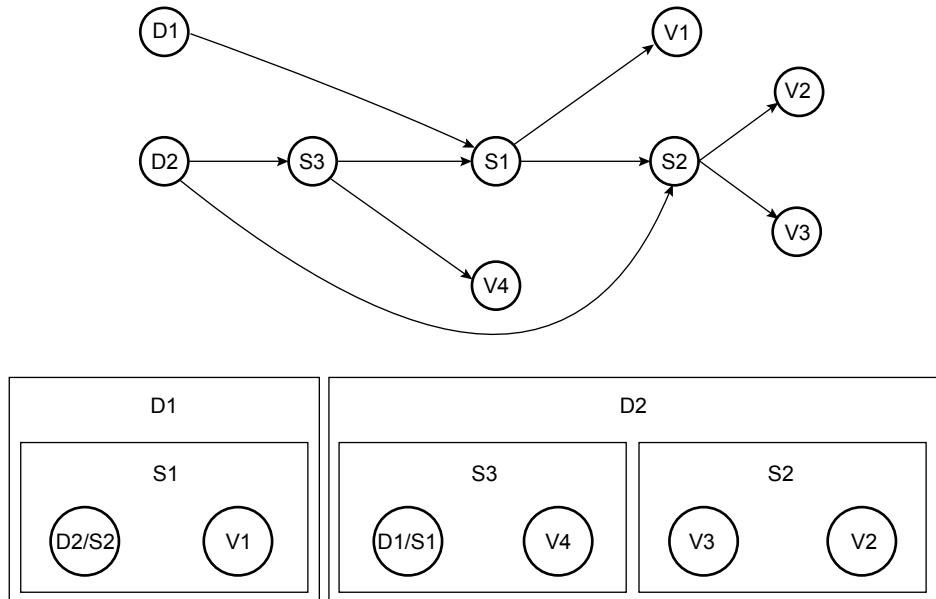
Obrázek 9.4: Uložení grafu v grafové databázi

Na obrázku 9.5 vlevo je ukázka grafu, který bychom mohli uložit do sloupcové databáze tak, jak je vidět na obrázku vpravo. Jak je patrné, pro tento speciální typ grafu uložení vytvoříme, nicméně by průchod grafem opět znamenal neustálé dohledávání klíčů a jím odpovídajících hodnot.



Obrázek 9.5: Uložení grafu ve sloupcové databázi

A konečně na obrázku 9.6 vidíme graf a jeho uložení v dokumentové databázi. Uvědomíme-li si, že dokumentové databáze vlastně v hodnotě k odpovídajícímu klíči ukládají hierarchickou strukturu, tedy strom, stačí v podstatě graf rozložit na množinu stromů a navíc uložit odkazy mezi nimi. A to je přesně to, co vidíme na obrázku. Pochopitelně toto uložení lze využít pouze v případě, že je graf možné rozložit na rozumné množství rozumně velkých stromů a hran mezi těmito stromy nevede příliš mnoho, jelikož jejich rostoucí množství už opět degraduje efektivitu průchodu.



Obrázek 9.6: Uložení grafu v dokumentové databázi

Celkově lze říci, že pro ukládání grafových, resp. hodně propojených dat jsou grafové databáze jistě optimální volbou. Ale na druhou stranu je samozřejmě pořeza dodat, že se nejedná o univerzální řešení pro jakákoli data. Navíc je třeba vzít v potaz také problém s distribucí velkého grafu. Čili i v tomto případě musíme zvážit výhody a nevýhody a pro danou konkrétní aplikaci vybrat vhodný kompromis.

9.8 Závěr

Problematika grafových databází se výrazně liší od předchozích tří typů NoSQL databází. Z hlediska datového modelu je podstatně složitější, jelikož grafová data v obecném případě není možné tak snadno distribuovat jako data typu (klíč, hodnota), at už je hodnota jakékoli složitosti. Stejně tak tento datový model vyžaduje specifické metody reprezentace, efektivního ukládání i vlastní dotazovací jazyky. Na tyto aspekty se proto podrobněji zaměříme v kapitole 13.

Část III.

**Pokročilé aspekty
zpracování Big Data**

10.

Další aspekty zpracování Big Data

V předchozích dvou částech knihy jsme vysvětlili základní principy týkající se Big Data, distribuovaného zpracování dat a hlavních typů NoSQL databází. V tomto okamžiku bychom mohli knihu uzavřít, ale my se podíváme ještě dál. Problematika efektivního zpracování Big Data je totiž složitější a existuje více typů souvisejících databázových technologií. Na ty nejdůležitější se podíváme v kapitolách 11–14. Ještě před tím ale v této kapitole krátce představíme oblasti, které s Big Data také úzce souvisí, přestože se nejedná přímo o databázové zpracování, které je hlavním cílem této knihy. Zmíníme zejména oblast datových analýz, které jsou jedním z častých způsobů zpracování Big Data, a s tím související problematiku vizualizace velkých dat. Představíme také oblast fulltextového vyhledávání v Big Data a krátce se podíváme na oblast cloud computingu. Cílem tohoto přehledu je vysvětlit čtenáři základní principy dalších souvisejících oblastí a především jejich vztah k Big Data.

10.1 Analytické zpracování Big Data

Databázové systémy, včetně těch představených v předchozí části knihy, jsou nejčastěji využívány pro řešení úlohy nazývané *online transaction processing* (OLTP). Zpracování typu OLTP předpokládá, že uložená data jsou průběžně aktualizována, dotazy jsou vyhodnocovány na těchto „živých datech“ a všechny dotazy jsou zpracovávány v reálném čase.

Oproti tomu například datové sklady (*data warehouses*) předpokládají uložení velkých objemů často historických dat a provádění rozsáhlých analytických úloh nad těmito daty [106] [114]. U analytických úloh se obvykle neočekává vyhodnocení v řádu jednotek nebo desítek milisekund a datové sklady bývají benevolentnější k dalším požadavkům standardně kladeným na databázové systémy. Protože NoSQL databáze umožňují distribuované zpracování velkých objemů dat, jsou v poslední době často využívány také právě pro zmiňovaný typ úloh. V této sekci krátce představíme základní pojmy a principy z oblasti analytického zpracování dat a datových skladů a jejich možnou realizaci pomocí NoSQL technologií.

10.1.1 Schéma dat

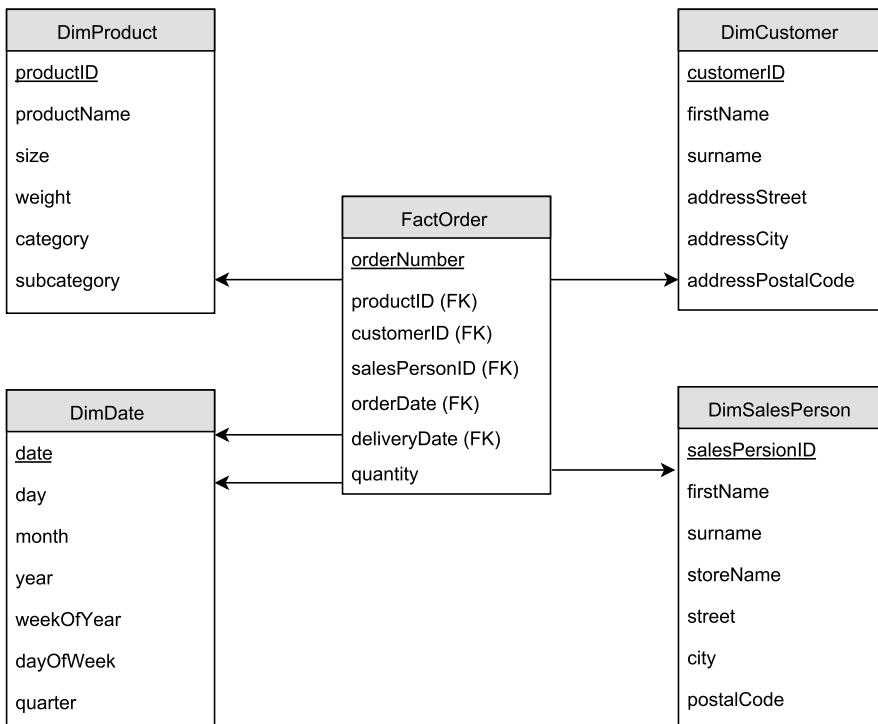
Datové sklady mívají několik datových vrstev. Spodní bývá *relační vrstva*, která udržuje *primární data* strukturovaná pomocí normalizovaného relačního schématu. My se budeme dále zabývat zejména strukturou vrstev odvozených (prezentačních), jejichž schéma často vychází z konceptu *dimenzionálního modelování* [114]. Uložená data jsou zde logicky rozčleněna do *schémat*, která odpovídají jednotlivým analyzovaným oblastem. Základem každého schématu je *faktová tabulka* (*fact table*), která obsahuje „měřitelné údaje“ (metriky, ukazatele, fakta) z jedné analyzované oblasti. Jako příklad vezmeme schéma pro oblast objednávek. Faktová tabulka **FactOrder** bude obsahovat záznamy o jednotlivých objednávkách a prodejích uskutečněných v nějaké síti prodejen – viz schéma na obrázku 10.1 [146].

FactOrder	
<u>orderNumber</u>	
productID	
customerID	
salesPersonID	
orderDate	
deliveryDate	
quantity	

Obrázek 10.1: Příklad faktové tabulky datového skladu objednávek

Data v této tabulce mohou pocházet z různých zdrojů (více o tvorbě datových skladů viz sekce 10.1.2). Datové skladы pak poskytují uživatelům možnosti, jak nahlížet na data „z různých úhlů pohledu“. Tento princip je realizován pomocí tzv. *dimenzi*, tedy atributů (kategorií), podle kterých je možné data filtrovat, agregovat, třídit atd. Pro každou dimenzi existuje tabulka obsahující možné hod-

noty daného atributu pro faktovou tabulkou a další informace. Tabulky dimenzií jsou navázány na faktové tabulky, např. tak, jak je vidět na obrázku 10.2.

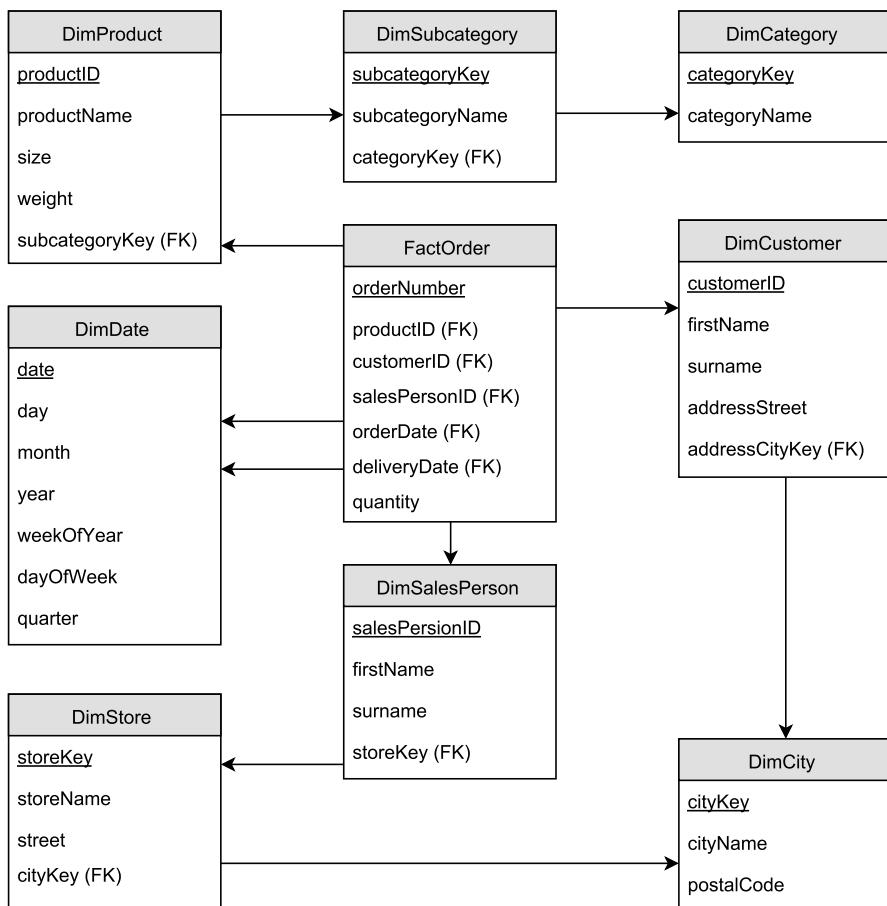


Obrázek 10.2: Příklad schématu datového skladu objednávek

Na tomto schématu jsou zobrazeny tabulky čtyř dimenzií, které odpovídají atributům faktové tabulky uprostřed – vybrané atributy ve faktové tabulce jsou cizími klíči obsahujícími hodnoty primárních klíčů tabulek dimenzií. Tabulky pro dimenze objednaného produktu (**DimProduct**), zákazníka (**DimCustomer**) a prodejce (**DimSalesPerson**) mají atributy, které obsahují dodatečné informace k příslušným hodnotám atributů **productID**, **customerID**, resp. **salesPersonID** faktové tabulky. Dimenze **DimDate** se od ostatních liší ve dvou ohledech:

1. Hodnoty všech jejich neklíčových atributů lze odvodit z hodnoty primárního klíče **date**, ale tyto hodnoty jsou „předpočítané“ kvůli efektivnějšímu procházení této dimenze.
2. Tato dimenze je navázáná na dva atributy faktové tabulky (**orderDate** a **deliveryDate**).

Uvedený příklad odpovídá schématu typu *hvězda* (*star schema*) – tabulky dimenzií jsou svázány vždy přímo s faktovou tabulkou a schéma není normalizované [114]. Např. dimenze **DimProduct** obsahuje kromě atributu **subcategory** i atribut **category**, jehož hodnoty se budou zbytečně opakovat u produktu z této kategorie a všech jeho podkategorií. Na obrázku 10.3 na následující straně je pak uvedeno schéma, které vzniklo normalizací některých dimenzií schématu typu hvězda z obrázku 10.2.



Obrázek 10.3: Příklad normalizovaného schématu datového skladu objednávek

Vidíme, že dimenze jsou vzájemně provázané, mohou tvořit jistou hierarchii a ne všechny jsou navázané přímo na faktovou tabulkou. Např. **DimProduct** nyní obsahuje atribut **subcategoryKey**, který je cizím klíčem obsahujícím hodnoty primárního klíče dimenze **DimSubcategory**, která je dále navázána na dimenzi **DimCategory**. Taktéž vzniklé schéma, u kterého nejsou všechny tabulky dimenzí navázány přímo na faktové tabulky, jsou typu *sněžová vločka* (*snowflake schema*) [114].

Pro základní porozumění struktuře datových skladů budou uvedené příklady po stačovat. Uvedená databázová schémata je možné realizovat pomocí jakékoli relační databáze a specifika datových skladů jsou zejména v jejich tvorbě a analytických funkcích. V NoSQL světě je možné takto strukturovaná data ukládat ve sloupcových databázích typu Cassandra nebo HBase. Jak již víme, pro distribuované sloupcové databáze je typická (vhodná) denormalizace schématu. Při použití výše uvedeného dimenzionálního modelu se v extrémním případě může jednat o tzv. *relační datovou kostku* (*relational datacube*), tedy „jednu velkou analytickou tabulkou“, ve které jsou uložena data faktů i dimenzí. Taková tabulka v podstatě odpovídá výsledku relačního spojení faktové tabulky se všemi tabulkami dimenzí. Dále ale existují specializované typy systémů pro tvorbu datových skladů, jako je Apache Hive (viz sekce 4.3.3.3). V dalším textu se lépe seznámíme s konkrétním použitím tohoto i jiných NoSQL nástrojů pro potřeby datových skladů.

10.1.2 Tvorba datových skladů

Pro proces naplnění datového skladu se tradičně užívá pojem ETL (*extract, transform, load*) podle jednotlivých fází tohoto procesu a jejich pořadí. Ve zkratce řečeno, fáze *extrakce* načte data z různých zdrojů, fáze *transformace* tato data upraví a pozmění dle konkrétních požadavků a fáze *načtení* zpracovaná data vloží do připravených tabulek databázového systému. Různí autoři a různé metodiky tyto fáze dále rozvádří, člení a specifikují konkrétní operace. Například bude me-li vycházet z metodiky zavedené Kimbalem a kol. [113] [114], můžeme fázi transformace rozdělit na dvě a proces popsat takto:

- fáze *extrakce*: načtení dat z různých zdrojů (podnikových databází, ERP systémů,¹ logů, sociálních sítí atd.) a jejich schémat, dočasné uložení těchto vstupních dat na disk a případně průběžné sledování a načítání změn zdrojových dat,
- fáze *čištění*: kontrola konkrétních požadovaných vlastností načítaných dat (jak na úrovni jednotlivých datových hodnot, tak na úrovni struktury dat); nevyhovující data se bud vyrádí, nebo se transformují a požadované vlastnosti se tak vynutí; odstranění duplicit ve zdrojových datech, dočasné uložení vyčištěných dat na disk,
- fáze *sjednocování (conform)* je potřeba, pokud má datový sklad integrovat data z různých zdrojů: sjednocení různých názvů atributů v různých zdrojích pro stejně objekty, dále např. převedení hodnot do společného jazyka, sjednocení (normalizace) numerických hodnot, které mají vyjadřovat stejnou veličinu, a zejména pak příprava tzv. *sjednocených dimenzí (conformed dimensions)*, které jsou navázány na více faktových tabulek a atributy těchto dimenzí mají identický význam pro všechny faktové tabulky (typickým jednoduchým příkladem je dimenze datumu); tato fáze je opět zakončena dočasným uložením dat na disk,
- fáze *doručení (deliver)* nebo častěji *načtení (load)*: fyzické vytvoření tabulek faktů a dimenzí a jejich uložení v cílovém úložišti datového skladu.

Fáze čištění a/nebo sjednocování mohou být ve skutečnosti prováděny již na datech načtených v databázovém systému (pak se někdy používá pojem ELT – *extract, load, transfer*).

Proces ETL může být implementován pomocí libovolných softwarových nástrojů podle specifického zadání pro danou aplikační oblast. Existuje ale samozřejmě množství různých komerčních i volně dostupných ETL nástrojů, které umí pracovat s širokou škálou zdrojových systémů a formátů a data ukládat do různých databázových systémů. Současné ETL nástroje již běžně podporují ukládání dat do systému Cassandra. Proces ETL je možné také realizovat přímo pomocí nástrojů Apache Pig (viz sekce 4.3.3.1) nebo Apache Hive (viz sekce 4.3.3.3). Tyto nástroje umožňují provádět proces ETL distribuovaně, např. s využitím Hadoop HDFS nebo

¹ Pojemem Enterprise Resource Planning (ERP) se označuje provozní (informační) systém, který organizace používá pro správu dat ze všech různých oblastí své činnosti, např. plánování výroby, plánování nákupu materiálu, správu lidských zdrojů atd. Nejznámějšími zástupci ERP systémů jsou SAP (<http://www.sap.com>) nebo např. cloudový systém NetSuite (<http://www.netsuite.com>).

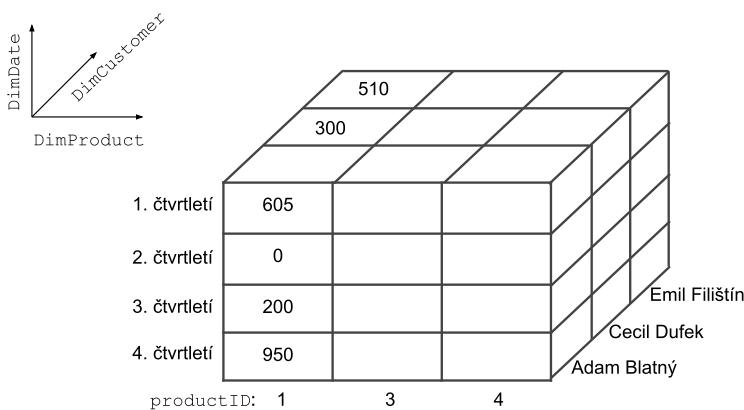
cloudového úložiště Amazon S3² a zpracování pomocí MapReduce. (Příklady práce se systémy Pig a Hive jsou uvedeny v příslušných sekcích.) Pokud jsou data nejdříve načtena do systému a pak transformována, jedná se o typický příklad procesu typu ELT.

10.1.3 Analytické zpracování

Jedním z hlavních důvodů pro tvorbu datových skladů je provádění operací za účelem získávání tzv. *business intelligence*. Pojem *business intelligence* se ale také označuje metody a technologie, které jsou určeny k získávání těchto nových informací [115]. V této oblasti se setkáváme s mnoha různými pojmy, s jejichž významy se ale často nakládá poměrně volně. Uvedeme ty nejčastější:

- online analytické zpracování (*online analytical processing*, OLAP),
- reportování (*reporting*) a přehledové zobrazení (*dashboards*),
- dolování dat (*data mining*) nebo
- prediktivní modelování.

Dále se budeme věnovat zejména analytickému zpracování typu OLAP. Tento typ operací efektivně využívá struktury datového skladu typu hvězda nebo sněhová vločka a umožnuje několik druhů operací s využitím jednotlivých dimenzí. Data faktové tabulky s jejimi dimenzemi můžeme promítnout do tzv. *multidimenziální kostky* (*OLAP cube*) – viz obrázek 10.4 pro tabulku objednávek a tři vybrané dimenze **DimDate** (vertikální osa s datem objednávek), **DimProduct** (horizontální osa se třemi vybranými produkty) a **DimCustomer** (šikmá osa se třemi jmény zákazníků).



Obrázek 10.4: Příklad OLAP kostky pro faktovou tabulku objednávek a tři dimenze

V kostce jsou naznačeny celkové hodnoty pro produkt s identifikátorem 1 – řekněme, že to jsou počty prodaných výrobků danému zákazníkovi v daném časovém úseku (v tomto případě po čtvrtletích vybraného roku). Podle jednotlivých dimenzí takové kostky pak můžeme realizovat zejména následující operace:

² <https://aws.amazon.com/s3/>

- *slice*: filtrování podle jedné zvolené hodnoty vybrané dimenze a tedy snížení dimenze kostky o jednu – doslova „ukrojení jednoho plátku z kostky“ (např. zachování pouze dat týkajících se produktu s *productID=1*, tedy v kostce zůstanou pouze počty produktů s *productID=1* prodaných v jednotlivých čtvrtletích jednotlivým zákazníkům),
- *dice*: výběr pouze několika hodnot z vybraných dimenzí a tím zmenšení OLAP kostky (např. jen první dvě čtvrtletí roku a jen výrobky s čísly 1 a 3, ale kostka zůstane trojrozměrná),
- *roll-up*: vybrané seskupení (agregace) hodnot podle hierarchie elementů jedné dimenze, např. sloučení čtvrtletí po dvojicích do pololetí nebo seskupení produktů podle jejich (pod-)kategorie,
- *drill-down*: inverzní operace k *roll-up*, tedy detailnější pohled podle vybrané dimenze, např. zobrazení výsledků po jednotlivých měsících místo po čtvrtletích.

Je zřejmé, že postupné interaktivní aplikování uvedených operací může být pro uživatele silným nástrojem pro analýzu dat a následný rozhodovací proces.

Z pohledu implementace daných operací jsou v literatuře rozlišovány zejména následující možnosti.

- ROLAP: relační OLAP, kdy jsou data faktových tabulek a dimenzí uložena v relační databázi a uvedené OLAP operace jsou implementovány pomocí standardních agregačních operací relačních databází. Tento přístup má výhodu v relativně úsporné diskové reprezentaci dat a také v tom, že jsou vždy přístupné všechny atributy dimenzí i faktových tabulek. Nevýhodou je nižší efektivita vyhodnocování OLAP operací takového řešení.
- MOLAP: multidimenzionální OLAP, kdy jsou data ukládána v multidimenzionálních databázích, které fyzicky ukládají OLAP kostku jako jednu multidimenzionální matici (včetně hierarchií jednotlivých dimenzí) a fyzicky se adresují přímo buňky této matice. Realizace výše uvedených analytických operací je pak velmi efektivní. Nevýhodou je nutnost „předpřipravení“ takového úložiště pro konkrétní vybrané dimenze a velikost diskové reprezentace.
- HOLAP: hybridní OLAP, který kombinuje oba předchozí přístupy. První z možných kombinací (tzv. datový sklad s duální granularitou) ukládá detailní data v relační databázi a vybrané dimenze jsou duplicitně agregovány pomocí multidimenzionálního úložiště typu MOLAP. Druhou možností je uložení starších dat pomocí úspornějšího ROLAP přístupu a nejnovějších dat pomocí MOLAP.

Ve světě NoSQL technologií existuje několik možností, jak řešit úlohy OLAP nad distribuovanými daty a s využitím distribuované výpočetní kapacity. Již téměř klasickým řešením je výše zmínovaný systém Apache Hive, který ukládá relační data pomocí distribuovaných souborových systémů kompatibilních s HDFS, a analytické funkce realizuje pomocí Hadoop MapReduce [100]. Při přímém použití nástroje Hadoop MapReduce ale musíme vždy počítat s větší časovou režií a výsledný systém prakticky nemůže být zcela *online*.

V roce 2014 byl uvolněn projekt Apache Kylin,³ který je z hlediska výše uvedené klasifikace typickým hybridním řešením. Systém ukládá primární data pomocí Hive (HDFS) s využitím relačních schémat datového skladu. Data připravená pro rychlé OLAP operace jsou uložena zvláště v úložišti typu HBase a celý systém poskytuje komunikační rozhraní na bázi SQL.

Velmi populární jsou řešení využívající pro uložení dat distribuovanou sloupcovou databází Cassandra a budování distribuovaného analytického systému nad těmito daty. Jednou z možností je opět využít Hadoop se všemi jeho výhodami a nevýhodami – několik takových řešení je např. dostupných v rámci systému DataStax Enterprise [10].

Dalším, zřejmě efektivnějším systémem pro vybudování analytického systému nad sloupcovou databází typu Cassandra je Apache Spark.⁴ Spark je framework pro masivně distribuované výpočty nad daty v paměti, který může zdrojová data přebírat mimo jiné z HDFS nebo právě ze systémů HBase a Cassandra (např. pomocí projektu Calliope⁵ nebo konektoru⁶ od firmy DataStax⁷). Spojení systémů Cassandra a Spark se tedy zdá být velmi efektivním řešením pro řadu různých úloh business intelligence [151] [102].

10.2 Vizualizace Big Data

Další oblastí, která souvisí s Big Data a také velmi úzce s jejich výše zmíněným analytickým zpracováním a oblastí business intelligence, je jejich vizualizace [104] [125]. V současné době většina nástrojů pro analýzu dat obsahuje sadu možností jak analyzovaná data vhodně vizualizovat. Vizualizace dat obecně napomáhá pořozumění datům pomocí jejich převedení do vhodné vizuální podoby. Např. máme-li hodnoty naměřené pomocí nějakého zařízení, pak jim mnohem snáz porozumíme, pokud je vizualizujeme ve formě křívy, která zobrazuje naměřené hodnoty v čase a umožňuje sledovat jejich nárůst, pokles, specifické charakteristiky, extrémní hodnoty apod. Po takovém vizuálním předzpracování je pak možné data automaticky analyzovat pomocí vhodných sofistikovaných analytických metod. Podobně můžeme mnohem snáze zkoumat např. vztahy v sítích, hledat určité shluky odpovídající např. komunitám nebo určitým biologickým vlastnostem atd.

V současné době existuje velké množství technik pro zobrazování dat, jako jsou sítové diagramy, teplotní mapy, dendrogramy, sloupcové grafy, koláčové grafy, tag cloud (neboli mrak slov – český překlad se ale nepoužívá), treemap (neboli stromová mapa) a mnohé další, které se dělí a liší v závislosti na tom, s jakými metodami (např. OLAP, dolování dat, statistika apod.), resp. jakým cílovým určením (např. běžná grafika, logistika, burzovní grafika apod.) je vizualizace spojena. Moderní vizualizační software navíc dokáže pro daná data sám navrhnut vhodný typ vizualizace, tedy např. pro jednorozměrná data sloupcový graf, pro data

³ <http://kylin.incubator.apache.org>

⁴ <http://spark.apache.org>

⁵ <http://tuplejump.github.io/calliope/>

⁶ <https://github.com/datastax/spark-cassandra-connector>

⁷ <http://www.datastax.com>

s geografickými souřadnicemi vizualizaci v mapě apod. Tím se uživateli situace výrazně zjednoduší.

V oblasti vizualizace Big Data nejde jen o problém *strukturálního modelování dat* (tj. detekce, extrakce a simplifikace klíčových informací v konkrétních datech) a následný problém vhodné grafické reprezentace, ale přibývají další aspekty:

- Vzhledem k velikosti dat obvykle dokážeme zobrazit pouze jejich určitou část nebo pouze vybrané hodnoty (např. každou 100. položku, průměr z každých 100 položek apod.). Vizualizace této části by nám měla dát odpověď na otázku, kterou z části chceme zobrazit v dalším kroku, nebo zda nás v určité oblasti zajímají přesnější data.
- Důsledkem zobrazování částí dat může být nejasnost trendů nebo ztráta charakteristických vlastností dat. Obecně totiž platí, že uživateli není možné zobrazit příliš mnoho informací, ale současně zobrazíme-li příliš málo, můžeme tím přijít o podstatné informace.
- Stejně jako jakékoli jiné algoritmy, musí být i ty vizualizační škálovatelné pro libovolně velká data.
- Obvyklý problém *redukce šumu*, tedy odstranění nežádoucích chyb a zkreslení, je v oblasti Big Data díky jejich vlastnostem zcela zásadní.

10.2.1 Vizualizace propojených dat

Hlavní výzvou v oblasti vizualizace Big Data jsou data, která jsou nějakým způsobem vzájemně propojena. Máme-li velké množství vzájemně nesouvisejících dat, můžeme je snadno rozdělit na menší části a tyto zobrazovat nezávisle klasickými metodami. Pokud je naopak propojení mezi daty mnoho, vizualizace se nám výrazně komplikuje.

10.2.1.1 Vizualizace grafů

Nejjednodušší problematikou z této třídy je pravděpodobně vizualizace grafů. Cílem je obvykle zajistit takové zobrazení, aby byl graf co nejpřehlednější, tj. aby byly uzly rozprostřeny symetricky, délky hran byly rovnoměrné a hrany se minimálně křížily [118]. Naneštěstí některé kombinace těchto kritérií jsou ve vzájemném sporu – jako např. symetrické rozprostření uzel grafu a minimální křížení hran.

Jedním z nejpopulárnějších řešení je sada přístupů nazývaná *silou řízené rozmístění* (*force-directed placement* nebo *spring embedder*) [86] [116]. V jedné z nejstarších metod tohoto typu je graf modelován tak, že se jeho uzly chovají, jako by byly elektricky nabité, zatímco chování hran odpovídá pružinám. Uzly se tedy odpuzují, zatímco hrany se až do určitého momentu přitahují, resp., pokud jsou příliš blízko sebe, tak se odpuzují. V takovém systému pak hledáme rovnovážný stav, tedy stav s nejnižší energií. Lze ukázat, že nalezení rovnováhy mezi těmito silami pak vede k rovnoměrným délkám hran a symetrickému rozmístění uzel. Rovnováha se hledá iterativně. Existuje i verze tohoto algoritmu pro Big Data, tedy

velké grafy, kdy jsou výpočty přitažlivých a odpudivých sil omezeny na minimální nutnou oblast a dále optimalizovány tak, aby netrvaly neúnosně dlouho.

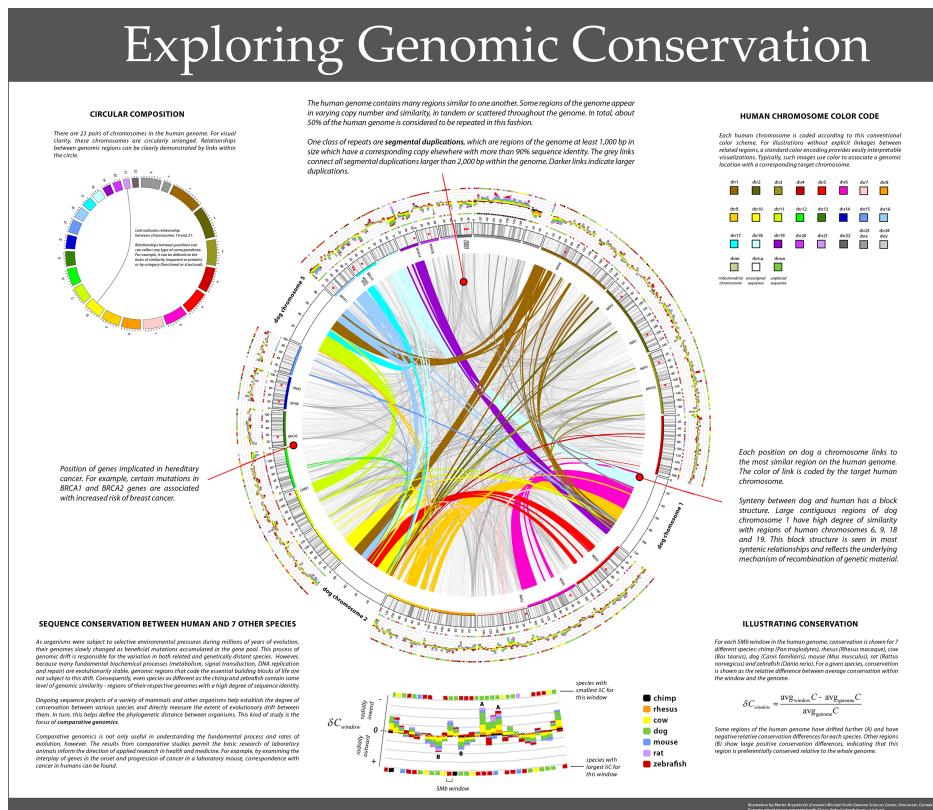
Pro vizualizaci velkých grafů ale stále zůstávají především dvě hlavní výzvy:

- Zatímco velikost dat roste vysokou rychlostí, vlastnosti zobrazovacích médií se vyvíjí podstatně pomaleji (jedná se o tzv. *interface problem*). Často pak můžeme narazit např. na omezení dané rozlišením obrazovky, které neumožní zobrazit více informací, resp. informace jsou příliš zhuštěné.
- Algoritmy jsou pro velké grafy neúnosně pomalé. Obvykle tedy není možné zobrazit celý graf.

Další komplikací vizualizace velkých grafů jsou grafy s různými typy hran a uzelů, jejichž efektivní vizualizace je podstatně složitější. Problémem také zůstává, že většina existujících algoritmů neumí efektivně reagovat na změny v grafu. Typicky je po změně potřeba celý graf zobrazit znovu, což u menších grafů problém není, ale u velkých pochopitelně ano.

Uvedené dva hlavní problémy můžeme řešit několika způsoby. Např. můžeme určité podmnožiny dat vhodně vizuálně zakódovat (třeba pomocí barev nebo tvarů) nebo využívat různé interaktivní metody jako lupa (*zoom*) nebo rybí oko (*fish eye*). Pokud jsou data skutečně velká, často se využívají techniky, které původní graf zjednoduší. Např. může jít o snížení počtu hran (třeba odstraněním hran s nízkou vahou), omezením grafu pouze na jeho kostru (tedy podstrom zahrnující všechny uzly), výběr podgrafu, který obsahuje pouze cesty maximální délky apod. Rychlosť v zobrazování grafů se pak snaží řešit tzv. *multi-scale algoritmy*, které graf zobrazují s různou úrovní detailu, ale se zachováním původního rozmístění. Např. v článku [96] autoři navrhují metodu, která identifikuje shluky uzelů (tedy množiny propojených uzelů, jež leží blízko sebe) a tyto nahradí umělými uzly při zachování jejich vzájemné pozice.

Populární zobrazovací metodou je také tzv. *obloukový graf (arc graph)*, nebo jeho varianta zobrazená na kružnici, tzv. *radiální graf (radial chart)*, jehož příklad je uveden na obrázku 10.5 na následující straně. Oba grafy zobrazují uzly na přímce nebo na kružnici a vztahy mezi nimi jsou reprezentovány oblouky. Jak je vidět na obrázku, cílem není analyzovat každý jednotlivý oblouk, ale spíše jejich typické shluky. Oblouky mohou být navíc rozlišeny barvou nebo tloušťkou, která vyjadřuje význam daného vztahu, např. vzdálenost, množství vzájemných odkazů, nebo pokud se jedná o agregaci informací o skupině objektů, množství aggregované informace. Také je možné data na přímce, resp. na kružnici různými způsoby seřadit, čímž můžeme opět zkoumat zajímavé vlastnosti. A samozřejmě můžeme k datům na přímku/kružnici přidávat další informace jako např. histogramy.



Obrázek 10.5: Radiální graf (zdvoj: <http://circos.ca/images/>)

10.2.2 Nástroje pro vizualizaci

V současné době existuje velké množství nástrojů, které se snaží nabídnout uživateli širokou paletu vizualizačních technik. Mezi nejznámější patří nástroj Tableau,⁸ Many Eyes,⁹ Gephi¹⁰ nebo Circos¹¹ (z něhož pochází graf v obrázku 10.5) a také vizualizační knihovny jako je D3¹² nebo Raphael.¹³ Většina z nich však neumí zpracovávat Big Data. Příkladem aplikace, která Big Data vizualizovat umí, je analytický nástroj Hunk¹⁴ postavený nad Hadoop MapReduce (viz sekce 4.3) nebo cloudové řešení Watson Analytics,¹⁵ které je už ale přímo plnohodnotným analytickým nástrojem.

⁸ <http://www.tableausoftware.com>

⁹ <http://www-958.ibm.com>

¹⁰ <http://gephi.github.io>

¹¹ <http://circos.ca>

¹² <http://d3js.org>

¹³ <http://raphaeljs.com>

¹⁴ <http://www.splunk.com/bunk>

¹⁵ <http://www.ibm.com/analytics/watson-analytics/>

10.3 Invertovaný index jako databáze

Zajímavou událostí ve světě NoSQL a Big Data je využití *invertovaného indexu* (*inverted index*) jako svého druhu databáze. Invertovaný index je datová struktura určená ke specifickému účelu: uložení prvků textu pro efektivní vyhledávání v nestrukturovaném textovém obsahu (*full-text search*, neboli *fulltextové vyhledávání*). Mezi nejznámější vyhledávací systémy (*search engines*) pak v současné době patří systémy Sphinx¹⁶ a Apache Lucene.

Princip invertovaného indexu je jednoduchý: vstupní text je rozdělen na jednotlivé prvky (*tokens*) a každý z nich je uložen společně se seznamem identifikátorů vstupních textů (*postings list*), v nichž se vyskytuje. Obvykle indexačnímu mechanismu na vstup předáváme dokument obsahující několik atributů (nadpis, autor, vlastní text atd.) a pro každý z atributů je vytvořen separátní invertovaný index (obsahující data daného atributu ze všech dokumentů). Uvažujme tedy dva jednoduché dokumenty s jedním atributem odpovídajícím textu dokumentu, uvedené v tabulce 10.1.

Tabulka 10.1: Dokumenty pro fulltextovou indexaci

ID	Text
1	NoSQL databáze a Big Data
2	Relační databáze a integrita dat

Nejjednodušší forma invertovaného indexu pak bude vypadat tak, jak je naznačeno v tabulce 10.2.

Tabulka 10.2: Struktura invertovaného indexu

Token	Postings list
NoSQL	1
databáze	1,2
a	1,2
Big	1
Data	1
Relační	2
integrita	2
dat	2

Již z této jednoduché, naivní podoby indexu je zjevná jeho zásadní výhoda: abychom získali identifikátory dokumentů obsahující slovo **databáze**, provedeme tu nejméně náročnou operaci v jakékoli databázi, vyhledání řádku podle klíče (v našem případě tedy získáme seznam identifikátorů [1,2]). Abychom získali identifikátory dokumentů obsahující zároveň slova **databáze** a **NoSQL**, vyhledáme

¹⁶ <http://sphinxsearch.com>

oba příslušné řádky a provedeme jejich průnik (v tomto případě tedy získáme seznam identifikátorů [1]).

Proces vytváření invertovaného indexu je přitom v praxi pochopitelně složitější, neboť musí vyhovět mnoha dalším požadavkům, jako je odstranění velkých písmen (verzálek), převedení tokenů na jejich základní tvar (*stemming* a *lemmatizace*), odstranění nevýznamných slov (*stopwords*), zohlednění synonym a u mnoha jazyků též odstranění diakritiky. Zároveň může být index kvůli efektivnímu vyhledávání řádků seřazen podle abecedy. Realističtější verze našeho indexu by tedy vypadala přibližně tak, jak vidíme v tabulce 10.3.

Tabulka 10.3: Struktura invertovaného indexu

Token	Postings list	Komentář
big	1	<i>odstranění verzálek</i>
dat	1,2	<i>odstranění verzálek, převedení na základní tvar</i>
databaze	1,2	<i>odstranění diakritiky</i>
integrita	2	
nosql	1	<i>odstranění verzálek</i>
relacni	2	<i>odstranění verzálek a diakritiky</i>
sql	2	<i>synonymum pro prvek „relaci“</i>

Tento proces se ve vyhledávacích programech nazývá *analýza* a důležitým principem pro fungování celého systému je, že je uživatelský dotaz zpracován stejným procesem: dotaz *Big Data* bude tedy převeden na výraz **big dat**.

10.3.1 Apache Lucene a jeho nástavby

Dominantním fulltextovým vyhledávacím nástrojem je v současnosti systém Apache Lucene, který původně navrhl a implementoval Doug Cutting, autor systému Hadoop. Lucene je přitom nízkoúrovňová knihovna, jež poskytuje rozhraní pouze v programovacím jazyce Java, v němž je implementována, a pro efektivní používání vyžaduje netriviální znalosti o své implementaci. Pro pohodlnější používání v ostatních programovacích jazycích a odstínění aplikačního kódu od nízkoúrovňové logiky se proto využívají nadstavby a rozšíření Lucene jako je Apache Solr¹⁷ a Elasticsearch,¹⁸ které poskytují REST rozhraní a zajišťují nízkoúrovňové operace.

V případě systému Elasticsearch tedy indexaci dvou dokumentů z tabulky 10.1 na předchozí straně provedeme následujícím HTTP požadavkem:

¹⁷ <http://lucene.apache.org/solr/>

¹⁸ <https://www.elastic.co>

```
curl -X PUT http://localhost:9200/articles/article/1 -d '{"title" : "NoSQL databáze ▶ a Big Data"}'
curl -X PUT http://localhost:9200/articles/article/2 -d '{"title" : "Relační databáze ▶ a integrita dat"}'
```

Vyhledání dokumentů obsahujících výraz **data** pak provedeme opět HTTP požadavkem:

```
curl http://localhost:9200/articles/article/_search?q=data
```

Výstupem tohoto dotazu bude odpověď obsahující relevantní dokumenty (*hits*) ve formátu JSON:

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.13424811,
    "hits": [
      {
        "_index": "articles",
        "_type": "article",
        "_id": "1",
        "_score": 0.13424811,
        "_source": {
          "title": "NoSQL databáze a Big Data"
        }
      }
    ]
  }
}
```

Konkrétní dokument přitom můžeme získat následujícím jednoduchým dotazem (výsledek je jako obvykle zvýrazněn kurzívou):

```
curl -XGET http://localhost:9200/articles/article/1
```

```
{
  "_index": "articles",
  "_type": "article",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {"title": "NoSQL databáze a Big Data"}
}
```

Jak vidíme, Elasticsearch vrací v obou případech v odpovědi kompletní dokument tak, jak jsme jej do indexu uložili. Může proto sloužit jako výkonná, distribuovaná dokumentová databáze [91]. Navíc však můžeme strukturu invertovaného indexu

využít nikoliv jen k vyhledání relevantních dokumentů, ale i k získání agregovaných informací o dokumentech, nebo jejich částech. V následujícím příkladu získáme seznam jednotlivých slov dle jejich četnosti pro dokumenty obsahující výraz databáze:

```
curl 'http://localhost:9200/articles/article/_search?search_type=count' -d '{
  "query" : {
    "match" : {
      "title" : "databáze"
    }
  },
  "aggregations" : {
    "words" : {
      "terms" : {
        "field" : "title"
      }
    }
  }
}'

{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "words" : {
      "buckets" : [ {
        "key" : "a",
        "doc_count" : 2
      }, {
        "key" : "databáze",
        "doc_count" : 2
      }, {
        "key" : "big",
        "doc_count" : 1
      }, {
        "key" : "dat",
        "doc_count" : 1
      }, {
        "key" : "data",
        "doc_count" : 1
      }, {
        "key" : "integrita",
        "doc_count" : 1
      } ]
    }
  }
}'
```

```

    }, {
      "key" : "nosql",
      "doc_count" : 1
    }, {
      "key" : "relační",
      "doc_count" : 1
    }
  ]
}
}
}

```

Spojení vyhledávání v nestrukturovaném textu, strukturovaných dotazů (např. „všechny články z roku 2015“) a agregovaných analytických informací tak z prostého vyhledávacího programu v případě Elasticsearch vytvoří výkonný nástroj i pro ukládání a zpracování Big Data.

Elasticsearch v praxi

Jedním z největších nasazení Elasticsearch je jeho použití ve službě GitHub,¹⁹ kde zajišťuje vyhledávání ve 2 miliardách dokumentů pro více než 8 milionů repozitářů zdrojových kódů; celkový objem dat se v současnosti blíží 100 TB. GitHub přitom indexuje všechny informace týkající se repozitářů: informace o jejich správcích, zdrojový kód samotný, hlášení chyb (*issues*) či návrhy oprav (*pull requests*). Zároveň GitHub využívá Elasticsearch v celé řadě interních aplikací, které slouží k monitoringu systému, vyhledávání zranitelných částí kódu či pokusů o průnik do systému [49].

10.3.2 Zpracování logů

Rapidně se rozvíjející oblastí, jež využívá principu invertovaného indexu pro ukládání, zpracování, získávání a vyhodnocování dat, je v současnosti zpracování logů. Ty obvykle obsahují textové informace, které nepřetržitě generují softwarové nástroje, jako jsou webové či e-mailové servery, routery, aplikační servery, databáze nebo monitorovací nástroje. Pionýrským nástrojem je v této oblasti zřejmě systém Splunk,²⁰ který obsahuje vlastní implementaci invertovaného indexu a je přímo zaměřen na ukládání časově vázaných dat (*time-based data*). Primárním způsobem získávání dat je doménový jazyk, vytvořený přímo na míru typických využití, výstupem jsou uživatelsky definované vizualizace a grafy (viz příklad na obrázku 10.6 na následující straně).

V současnosti dominantním open source systémem pro ukládání a zpracování logů je právě Elasticsearch, v kombinaci s rozšířenými Logstash²¹ a Kibana²² (které jsou souhrnně označovány zkratkou ELK). Logstash poskytuje propojení se zdrojovými daty (např. ze souborů na disku, monitorovacích nástrojů jako Nagios,²³

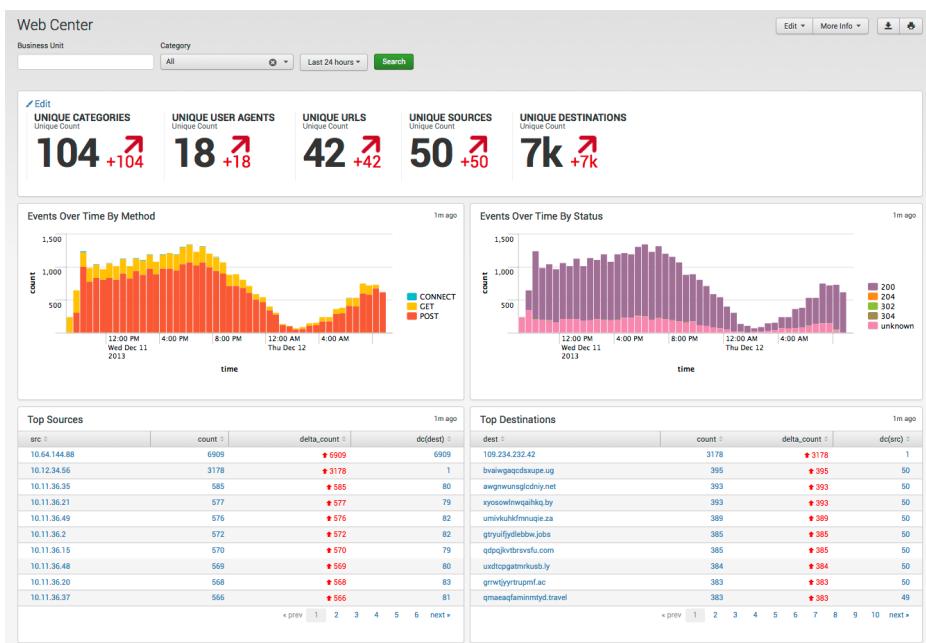
¹⁹ <http://github.com>

²⁰ <http://splunk.com>

²¹ <http://www.elastic.co/products/logstash>

²² <http://www.elastic.co/products/kibana>

²³ <https://www.nagios.org>



Obrázek 10.6: Analytické rozhraní nástroje Splunk

komunikačních kanálů jako RabbitMQ²⁴ apod.) a automatickou konverzi nestrukturovaného textu do strukturovaného dokumentu (např. logů webového serveru Nginx,²⁵ databáze PostgreSQL atd.). Kibana pak poskytuje grafické rozhraní pro vyhledávání a analýzu uložených dat (viz obrázek 10.7 na následující straně).

Úspěšné využití systémů jako Splunk nebo Elasticsearch ukazuje nejen to, že invertovaný index lze využít jako výkonnou databázi, ale zároveň, že fulltextové vyhledávání je velmi efektivním způsobem dotazování v oblasti Big Data. Konečně většina online služeb pro ukládání a vyhodnocování logů jako Loggly²⁶ či Papertrail²⁷ používá právě fulltextové vyhledávání jako primární přístup k informacím.

10.4 Cloud computing

Poslední pojem, který v této kapitole představíme, je *cloud computing*²⁸ [148]. Firma Gartner popisuje cloud computing jako způsob vytváření softwaru, který je založen na myšlence nabízet externím zákazníkům prostřednictvím internetu škálovatelné IT technologie v podobě služeb. IT technologií rozumíme kombinaci hardwaru a softwaru, která může odpovídat aplikaci, platformě nebo infrastrukturě. Pak rozlišujeme tři modely:

- *Software as a Service* (SaaS), tedy software jako služba, kterou obvykle využívají přímo koncoví uživatelé dané aplikace.

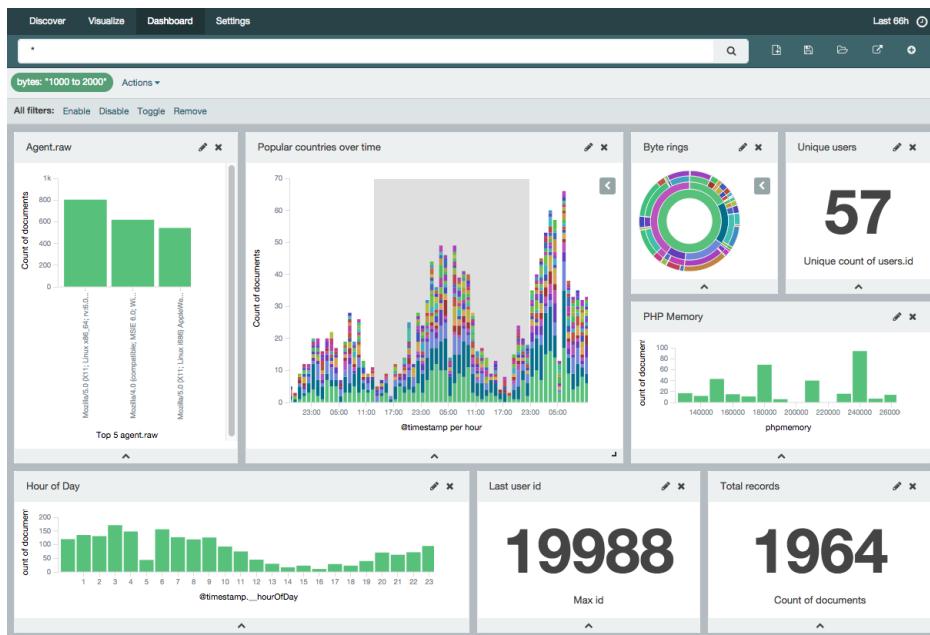
²⁴ <https://www.rabbitmq.com>

²⁵ <http://nginx.org>

²⁶ <https://www.loggly.com>

²⁷ <https://papertrailapp.com>

²⁸ Tento pojem nemá vhodný překlad do češtiny, používá se vždy v anglické verzi.



Obrázek 10.7: Analytické rozhraní nástroje Kibana pro Elasticsearch

- *Platform as a Service* (PaaS), tedy platforma jako služba, která je určena vývojářům, pro něž nabízí sadu nástrojů pro vývoj aplikací, nebo obecně pro provozování vlastního softwaru.
- *Infrastructure as a Service* (IaaS), tedy infrastruktura jako služba, která slouží pro poskytování infrastruktury, tj. obvykle robustního či jinak ne snadno dostupného hardwaru, typicky formou virtualizace.

Ve všech třech modelech uživatel neplatí za konkrétní software nebo hardware, ale za jejich používání, jde tedy o jakousi formu pronájmu. Obvykle se jedná o platbu za dobu využívání, velikost uložených/zpracovaných dat apod. Z hlediska cílové množiny uživatelů rozdělujeme cloudy na *veřejné*, tedy dostupné komukoli, v nichž dochází ke sdílení technologií, *privátní*, tedy provozované určitou organizací pro vnitřní účely nebo výhradně pro vybrané zákazníky, *komunitní*, tedy určené pouze pro určitou vybranou skupinu uživatelů (komunitu), popř. jejich nejrůznější kombinace.

Hlavní výhody clouдовého přístupu jsou především tyto:

- Uživateli odpadá nutnost spravovat (pořizovat, instalovat, upgradovat, udržovat v chodu atd.) příslušné technologie, pouze je využívá. Navíc se nemusí starat ani o případy, kdy dojde např. k výměně určitého hardwaru nebo implementaci efektivnější verze softwaru, pokud rozhraní dané služby zůstává stejně.
- Díky internetu může uživatel služby cloudu využívat odkudkoli.
- Poskytovatel služby je (v rámci svých možností) schopen nabízet zákazníkům různě robustní řešení odpovídající jejich konkrétním požadavkům. Máme tedy

zcela automaticky (i když samozřejmě ne zdarma) zajištěnou škálovatelnost pro měnící se požadavky různých aplikací.

- Vzhledem k tomu, že jsou data uložena na serveru/serverech cloutu, zjednoduší se problematika sdílení dat, které lze zajistit vhodným zpřístupněním dat vybraným uživatelům cloutu.

Jak jsme již ale viděli u mnoha jiných přístupů, i v tomto případě musíme akceptovat několik nevýhod. Mezi hlavní problémy a výzvy cloud computingu patří především tyto:

- Data, jež ukládáme do veřejného cloutu, vlastně ukládáme do místa, které nemáme pod kontrolou. Teoreticky je tedy možné zneužítí dat. Na druhou stranu je ale třeba říci, že u solidních poskytovatelů lze očekávat vysokou míru zabezpečení. Nicméně přestože se od prvotních řešení techniky zabezpečení vylepšují, stále panuje (často ne úplně racionální) obava uživatelů ze zneužití dat.
- V případě, že využíváme platformu cloutu, často nám hrozí již známý problém proprietárního uzamčení (vendor lock-in), které nedovoluje snadný přechod na platformu jinou.
- Ceny za poskytování služeb, u nichž odpadá množství výše uvedených problémů, mohou být nad možnosti menších firem či akademických institucí. Poskytovatelé cloudů sice nabízejí určitou míru využití zdarma nebo za symbolické částky, nicméně tyto kapacity jsou obvykle výrazně omezeny. Využití cloutu má tedy přirozeně smysl pouze v případě, že náklady na koupi, instalaci a údržbu příslušných technologií vysoko převyšují ceny za využití cloudového přístupu.

Výše uvedená obecná specifikace cloud computingu se dá překvapivě aplikovat na mnoho běžných aplikací a nástrojů, jako je např. Google Calendar,²⁹ Dropbox³⁰ ale koneckonců i třeba GMail.³¹ V těchto případech totiž také využíváme službu, která nám umožňuje ukládat a zpracovávat data, aniž bychom se zabývali správou souvisejícího hardwaru a softwaru. Mezi nejznámější zástupce komplexnějších cloudových technologií a služeb, kde už nalezneme robustní realizace modelů PaaS a IaaS, patří např. Microsoft Azure,³² Amazon Web Services,³³ Google App Engine,³⁴ Red Hat OpenShift³⁵ a mnohé další.

10.4.1 Cloud computing a Big Data

Jak tedy vlastně souvisí cloudy a Big Data? Z předchozích kapitol již víme, že efektivní zpracování Big Data vyžaduje cluster uzlů, na nichž máme data distribuovaná, replikována atd. Pro efektivní ukládání a zpracování Big Data pak využíváme různé typy NoSQL databází a nejrůznější nástroje pro distribuované a paralelní zpracování dat. Hlavní výhodou distribuovaného zpracování je právě škálovatelnost,

²⁹ <https://www.google.com/calendar/>

³⁰ <https://www.dropbox.com>

³¹ <http://www.gmail.com>

³² <http://azure.microsoft.com>

³³ <http://aws.amazon.com>

³⁴ <https://appengine.google.com>

³⁵ <https://www.openshift.com>

která nám umožňuje doplňovat do clusteru další uzly dle potřeby dané aplikace. Na druhou stranu je třeba přiznat, že nákup, instalace a údržba takového clusteru je velmi drahá a náročná. A zde se právě dostávají ke slovu *cloudy*.

Cloudy nabízí škálovatelné řešení bez nutnosti spravovat konkrétní hardware nebo software. Jsou tedy přímo určeny pro pohodlné řešení problémů spojených se zpracováním Big Data. S jejich využitím můžeme získat podstatně jednodušší řešení konkrétních aplikací – nutnou infrastrukturu a popř. i konkrétní aplikace máme díky cloudu k dispozici v podstatě okamžitě a navíc za nás vyřeší problém správy, škálovatelnosti atd. My se tudíž můžeme zaměřit pouze na novou funkcionality, kterou daná aplikace vyžaduje, tedy např. na efektivní a komplexní analytické zpracování dat. Z hlediska velkých objemů dat často platí, že je pronájem infrastruktury v clodu levnější než její nákup a údržba. Platí to především v případech, kdy bychom pro naši aplikaci celou infrastrukturu plně nevyužili. Mnoho aplikací právě z oblasti analytického zpracování Big Data potřebuje vysoký výpočetní výkon pouze v určitých časových intervalech (např. jednou za několik hodin). Flexibilita cloudů je právě na takového požadavky připravena a uživatel platí pouze za dobu, kdy cloudové technologie využívá.

Výhod zpracování Big Data pomocí cloudů je mnoho. Existují ale samozřejmě také nevýhody. Zatímco nevýhoda spojená s vysokou cenou nám v tomto případě odpadá, zbylé dvě naopak získávají na významu. Jak jsme již zmínili, Big Data často představují velmi cenný majetek konkrétní firmy a jejich bezpečné uložení je tedy zásadní. A stejně tak problém proprietárního uzamčení může znamenat, že další nároky naší aplikace už nedokáže příslušný poskytovatel clodu uspokojit a přechod k jinému by pro nás znamenal, vzhledem k velikosti dat, velmi drahy krok. Další komplikací může být samozřejmě vůbec samotná velikost dat, která musíme do clodu uložit.

Infrastruktura jako kód

Tou skutečně revoluční změnou, kterou cloud a spřízněné technologie přináší, je *elasticita infrastruktury*, tedy možnost operativně zvětšovat a zmenšovat výpočetní kapacitu. Tento zdánlivě banální fakt má závažné důsledky pro architekturu, administraci i technickou implementaci aplikací a systémů v clodu.

V pojetí clodu je zařízení, na kterém provozujeme určitou službu, nejenom „virtuální“, ale především prchavou entitou: vzniká a zaniká dle potřeby. Z toho plyne, že stroje a zařízení v clodu nemůžeme konfigurovat manuálně, tj. například instalovat jednotlivé softwarové služby, upravovat jejich nastavení, spouštět je, propojovat atd. Takové procesy musíme v prostředí clodu automatizovat. V posledních několika letech se pro tento fakt vžilo označení *infrastruktura jako kód* (*infrastructure as code*). Adam Jacob, spoluautor nástroje Chef,³⁶ shrnuje cíle tohoto přístupu jako [108] „umožnění kompletní rekonstrukce funkčního systému pouze z repozitáře se zdrojovými kódy, zálohy aplikačních dat a surové výpočetní kapacity“. V případě izolovaného incidentu, např. selhání disku, tak oprava nejen neznamená fyzický

³⁶ <http://www.chef.io>

zásah (např. cestu do serverovny), ale ani manuální intervenci (např. připojení nového disku). Jednodušší, spolehlivější a často i rychlejší je kompletní vytvoření nového „serveru“ od nuly, prostřednictvím nástroje na správu konfigurací jako je např. Chef, Puppet,³⁷ Ansible³⁸ a mnohé další.

Dovedeno do důsledků, v prostředí cloudu může jakýkoliv disk kdykoliv selhat, jakýkoliv server vypadnout ze sítě. Extrémní formou přijetí tohoto faktu je open source nástroj Chaos Monkey,³⁹ vyvinutý ve společnosti Netflix⁴⁰ [65]: „Došlo nám, že nejlepší obranou proti závažným a neočekávaným výpadkům a selháním je zkrátka selhávat častěji. Tím, že selhání aktivně způsobujeme, jsme donuceni naše služby budovat jako mnohem odolnější.“ Chaos Monkey způsobuje chaos v infrastruktuře: náhodně vypíná virtuální servery, a to nikoliv v simulaci nebo testovacím režimu, ale přímo v produkčním prostředí. Důležitým faktorem přitom je, že běží pouze v běžné pracovní době, aby správci systému mohli reagovat. Pokud autoři určité služby nechtějí „ohrozit“ její funkce prostřednictvím Chaos Monkey, musí to explicitně deklarovat v konfiguraci a očekává se, že vysvětlí proč. Důsledkem takového radikálního přístupu pak je, že architektura a implementace určitého systému přímo počítá se selháním, které je pro ni běžným faktorem, nikoliv neočekávaným incidentem.

S postupem času Netflix vytvořil další podobné nástroje, např. *Latency Monkey*, který nevypíná celý virtuální server, ale uměle prodlužuje dobu odpovědi na požadavek a testuje tak reakci systému na přetížení sítě, nebo *Chaos Gorilla*, který simuluje výpadek celého datového centra (*availability zone*) v síti Amazon [107]. Důsledkem tohoto přístupu je, že právě Netflix často přečká bez úhony rozsáhlé výpadky sítě Amazon [80], které jinak postihují služby jako Reddit,⁴¹ Foursquare⁴² nebo Heroku.⁴³

³⁷ <http://puppetlabs.com>

³⁸ <http://ansible.com>

³⁹ <https://github.com/Netflix/SimianArmy>

⁴⁰ <https://www.netflix.com>

⁴¹ <https://www.reddit.com>

⁴² <https://foursquare.com>

⁴³ <https://www.heroku.com>



Dotazování nad NoSQL databázemi

Jen málokterá aplikace využívá databázi jen jako černou díru, kam se data jednou uloží a už se s nimi dále nepracuje. Typicky se naopak jednou uložená data opakově načítají nebo se propojují s dalšími daty a různě agregují, aby uživatel získal přístup k dalším zajímavým odvozeným informacím.

Databázové systémy nabízejí různé možnosti, jak pracovat a dotazovat se nad daty v nich uloženými. Začneme u relačních databází – většina z nich podporuje dotazovací jazyk SQL. Alespoň základní znalost jazyka SQL má dnes snad každý vývojář, i když málokdo zná jazyk SQL v celé jeho šíři. Tím, že jazyk SQL není jen dotazovací jazyk, ale nabízí i prostředky pro správu celé databáze a její struktury, řízení přístupových práv, zápis uložených procedur, práci s multimédii a mnoho dalšího, je standard velice rozsáhlý – v současné době se skládá z patnácti částí. Jednotlivé databázové systémy proto obvykle implementují jen vybrané části standardu, na druhou stranu ale často nabízejí vlastní rozšíření nad jeho rámec.

Dotazovací část jazyka SQL je relativně jednoduchá na osvojení a v praxi se můžeme setkat s mnoha systémy, kde pokročilý uživatel může přímo zapisovat dotazy v syntaxi SQL. Mnohé aplikace naopak generují rozsáhlé dotazy zcela automaticky.

Dotazovací modul databáze se tak musí vypořádat i s nečekanými a velmi složitými dotazy. Většina databází dotaz nejprve různě automaticky vnitřně přepisuje tak, aby jeho provedení co nejméně zatěžovalo databázi a v co největší míře využívalo indexy, a přitom se zajistilo, že výsledek bude identický s původním dotazem. Některé databáze dokonce umí za běhu automaticky vytvářet chybějící indexy, aby pomocí nich bylo možné zrychlit příští provedení dotazu.

Tento přístup je pohodlný pro autory dotazů v SQL, ale klade samozřejmě vysoké nároky na inteligenci a výkon databáze. To je ovšem v rozporu s požadavky NoSQL databází, které se snaží nabídnout škálovatelnost a výkon, což právě významně komplikuje, ne-li znemožňuje využití příliš komplexního dotazovacího jazyka. V počátcích tak NoSQL databáze většinou žádné možnosti dotazování ne-nabízely. V předchozím textu jsme ale viděli, že některé systémy v omezené míře dotazování podporují, nezřídka pomocí jazyků, které jsou právě jazykem SQL inspirovány. V této kapitole se proto podíváme na to, jaké přístupy a jazyky pro dotazování ve světě NoSQL existují.

11.1 Přímý přístup pomocí programového rozhraní

V kapitole 6 jsme se seznámili s nejjednoduššími zástupci NoSQL databází, databázemi typu klíč-hodnota. Jak již víme, tyto databáze umí efektivně uložit a načíst hodnotu s daným klíčem a v mnoha případech je hodnota pro databázi zcela transparentní. V takovém případě nemá ani smysl nasazení dotazovacího jazyka. Pokud potrebujeme nad takto uloženými daty provést nějaký dotaz, nezbývá nám, než aby aplikace všechny potřebné hodnoty načetla do paměti a tam s nimi provedla požadované operace. Pro složitější dotazy to samozřejmě znamená ruční psaní velkého množství kódu, který manipuluje s daty z databáze. Pokud programátor dobrě nechápe princip a funkce použité databáze, může snadno napsat aplikaci, která z databáze načítá příliš mnoho dat a je neefektivní.

NoSQL databáze využívají mnoho různých protokolů pro přenos a serializaci dotazů i dat. Některé databáze, jako například Redis nebo MongoDB, využívají proprietární protokol i datový formát, mnoho dalších, jako například Riak nebo Elasticsearch, využívá protokol HTTP a formát JSON.

Jak jsme viděli v sekci 6.1.1, načtení hodnoty pro klíč **David** můžeme v systému Riak provést pomocí požadavku HTTP:

```
curl -X GET http://localhost:8098/buckets/autori/keys/David
```

Přesto poskytuje Riak, podobně jako téměř všechny ostatní databáze, klientské knihovny pro nejpoužívanější programovací jazyky. Tyto knihovny obalují volání databáze rozhraním, jež je přirozené a obvyklé pro daný programovací jazyk.

Následující příklad ukazuje, jak provést požadavek z předchozího příkladu pomocí Java API:

```
// jmenný prostor klíčů, který používáme  
Namespace namespace = new Namespace("autori");
```

```
// identifikace hodnoty, kterou chceme načíst (jmenný prostor a v něm klíč "David")
Location location = new Location(namespace, "David");

// pomocná třída pro načítání hodnot výsledku
FetchValue fv = new FetchValue.Builder(location).build();

// odeslání požadavku na server a jeho provedení
FetchValue.Response response = client.execute(fv);

// načtení objektu výsledku
RiakObject obj = response.getValue(RiakObject.class);
```

Podobně fungují například i knihovny pro dotazování nad dokumentovou databází MongoDB. Dotazovací jazyk použitý v MongoDB je silně inspirován formátem JSON (viz sekce 7.2). Například vypsání uživatelů starších 33 let z kolekce `users` můžeme pomocí dotazovacího jazyka MongoDB zapsat jako:

```
db.users.find( { age: { $gt: 33 } } )
```

Pokud bychom chtěli stejný dotaz provést přímo z aplikace v jazce Java, použijeme následující relativně dlouhý kód:

```
// připojení k MongoDB serveru
MongoClient mongoClient = new MongoClient();

// připojení ke konkrétní databázi "db"
DB db = mongoClient.getDB("db");

// otevření požadované kolekce "users"
DBCollection coll = db.getCollection("users");

// dotaz, který vrátí uživatele starší 33 let
BasicDBObject query = new BasicDBObject("age", new BasicDBObject("$gt", 33));

// kurzor umožňující průchod výsledkem
DBCursor cursor = coll.find(query);

// zpracování všech vrácených dokumentů
try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
} finally {
    cursor.close();
}
```

Z výše uvedených příkladů by se mohlo zdát, že klientská knihovna práci spíše komplikuje než usnadňuje – v obou příkladech je programový kód mnohem delší než „surový“ požadavek. Klientské knihovny však jednak poskytují mnohem více než pouhý „obal“ pro požadovanou operaci: například využití více serverů v clusteru (v případě systémů Riak či Elasticsearch). Především však serializují a deserializují do nativních entit daného programovacího jazyka, což je důležité zejména u silně typovaného jazyka, jako je např. Java, a umožňují data předávat dalším komponentám aplikace. „Upovídánost“ uvedeného příkladu přitom pramení

ze zákonitostí syntaxe jazyka Java – v programovacím jazyce Ruby bychom například poslední příklad zapsali takto:

```
client = Mongo::Client.new([ '127.0.0.1:27017' ], database: 'db')
users = client[:users].find "age" => { "$gt" => 33 }
users.each { |user| puts user }
```

11.2 MapReduce

Problémy se škálováním dotazů nad velkými objemy dat řeší MapReduce (podrobnejší viz kapitola 4). Dotaz nad daty implementujeme jako dvě funkce Map a Reduce, které se pak spustí distribuovaně. Oproti předchozímu příkladu tak samotný výpočet dotazu probíhá v databázi, nikoliv na klientovi.

Z hlediska dotazování nad daty má ovšem přístup MapReduce dva klíčové nedostatky. Funkce většinou musíme implementovat v jednom konkrétním programovacím jazyce, kterému rozumí databáze. A větším problémem je samotný princip MapReduce, jelikož korektní implementace složitějších dotazů jako funkce Map a Reduce vyžaduje od programátora poměrně velké intelektuální úsilí. MapReduce se tak postupně stává jen běhovým prostředím pro vyhodnocení dotazů, které se zapisují pohodlnějším způsobem ve specializovaném dotazovacím jazyce. Na tomto principu funguje například dotazovací jazyk Pig Latin (viz sekce 4.3.3.1), který se překládá na MapReduce úlohy. Ještě více vstříc požadavkům uživatelů vychází například dotazovací jazyk HiveQL (viz sekce 4.3.3.3), který se opět překládá na MapReduce úlohy, nicméně jeho syntaxe v podstatě odpovídá jednoduché podmnožině jazyka SQL. A podobně například firma Oracle nabízí nástroj pro překlad XQuery dotazů na MapReduce úlohy pro Hadoop.¹ Z MapReduce se tak postupně stává černá skříňka bežící na pozadí uživatelsky příjemnějších způsobů pro zápis dotazu.

11.3 Specifické dotazovací jazyky

Tím, že relační databáze vycházejí ze stejného datového modelu a mají za sebou dlouhý vývoj, bylo dost času, aby se jazyk SQL etabloval jako standardní dotazovací jazyk, který podporují všechny relační databáze. Pro NoSQL databáze zatím žádný jednotný dotazovací jazyk neexistuje. Důvodů je několik. Jednak je pole NoSQL poměrně heterogenní a různé databáze používají různé datové modely. Dotazovací jazyk, jím podporované operátory a datové typy přitom musí odpovídat použitému datovému modelu. Dalším problémem je, že navrhnut správně dotazovací jazyk typu SQL není úplně triviální úloha. Kvůli nutnosti rychlého řešení tak mnoho NoSQL databází nabízí dotazovací jazyky, které nepřicházejí s vlastní dobře srozumitelnou syntaxí, ale používají již existující strojově čitelný zápis pro strukturovaný zápis dotazu. Do této skupiny patří například dotazovací jazyk systému MongoDB nebo Elasticsearch se svým Query DSL [92].

¹ http://docs.oracle.com/cd/B51174_01/doc.24/e51161/oxb.htm#BDCUG526

11.3.1 Elasticsearch Query DSL

Dotazy pro Elasticsearch se v Query DSL zapisují jako objekty JSON. Struktura objektu pak popisuje, co přesně má dotaz provést. Nebudeme zde suplovat referenční příručku tohoto dotazovacího jazyka, pouze na několika příkladech ukážeme, jako tento dotazovací jazyk vypadá (viz také sekce 10.3.1).

Dotaz se zapisuje jako objekt JSON, který má vlastnost `query`. Tato vlastnost pak reprezentuje objekt dotazu, ve kterém lze jako vlastnosti používat další klauzule. Např. klauzule `match` vyhledá ty dokumenty, kde se v zadané položce vyskytuje hledaná hodnota. Následující dotaz by tak nalezl dokumenty, které v položce `name` obsahují hodnotu `NoSQL`:

```
{
  "query": {
    "match": {
      "name": "NoSQL"
    }
  }
}
```

Pomocí klauzule `bool` pak můžeme konstruovat složitější dotaz. Pokud bychom chtěli najít dokumenty, které mají v položce `name` hodnotu `NoSQL`, ale zároveň v ní není text `BigData`, použili bychom následující dotaz:

```
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "name": "NoSQL"
        }
      },
      "must_not": {
        "match": {
          "name": "BigData"
        }
      }
    }
  }
}
```

Uvedená klauzule `match` realizuje fulltextové vyhledávání. Pokud nám jde o přesnou shodu hodnoty, musíme použít klauzuli `term`. Například nalezení dokumentů, které v položce `currency` mají hodnotu `EUR`, provedeme pomocí dotazu:

```
{
  "query": {
    "term": {
      "currency": "EUR"
    }
  }
}
```

Chceme-li najít dokumenty s hodnotou v určitém rozsahu, použijeme klauzuli `range`. Například nalezení dokumentů, kde je v poli `price` hodnota mezi 100 a 200, zajistí následující dotaz:

```
{
  "query": {
    "range": {
      "price": {
        "gte": 100,
        "lte": 200
      }
    }
  }
}
```

Potřebujeme-li výsledek dotazu filtrovat, je efektivnější používat filtry místo prostého logického spojování podmínek pomocí klauzule `bool`. Pro nalezení dokumentů, které mají v názvu `NoSQL` a jejich cena je mezi 100 a 200, tak můžeme použít oba dva následující dotazy:

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "name": "NoSQL" } },
        { "range": { "price": { "gte": 100, "lte": 200 } } }
      ]
    }
  }
}

{
  "query": {
    "filtered": {
      "query": {
        { "match": { "name": "NoSQL" } }
      },
      "filter": {
        { "range": { "price": { "gte": 100, "lte": 200 } } }
      }
    }
  }
}
```

Jak je vidět, filtr se vytváří tak, že objekt s dotazem obalíme do dalšího objektu uloženého ve vlastnosti `filtered`. Za samotný dotaz se pak ještě doplní filtr do vlastnosti `filter`.

Chceme-li výsledek dotazu seřadit, přidáme do dotazu vlastnost `sort`, která obsahuje objekt popisující, podle jakých položek a jakým způsobem má řazení proběhnout. Pokud bychom tedy výsledky předchozího dotazu chtěli seřadit podle ceny, stačí dotaz pouze mírně doplnit:

```
{
  "query": {
```

```

"filtered": {
  "query": {
    { "match": { "name": "NoSQL" } }
  },
  "filter": {
    { "range": { "price": { "gte": 100, "lte": 200 } } }
  }
},
"sort": { "price" }
}

```

Dotazovací jazyk Elasticsearch nabízí mnoho dalších možností, jako například agregace nebo konkrétní výskyty hledaného textu (*highlight*). Již z výše uvedených ukázek je však patrné, že zápis složitějších dotazů pomocí komplexního objektu JSON přestává být přehledný, a proto existují nadstavbové knihovny pro .NET, Python nebo Ruby, které umožňují jejich definici prostřednictvím výrazů daného programovacího jazyka.

11.4 Univerzální dotazovací jazyky

Zápisy dotazů jako objekt JSON jsou sice funkční, ale spíše odpovídají vnitřní strojové reprezentaci dotazu než zápisu, který by byl snadno srozumitelný pro člověka. Ve světě dotazovacích jazyků tak kralují jazyky, které se snaží svojí syntaxí uživatelům usnadnit zápis nejběžnějších dotazů. Většina z těchto jazyků navíc bývá standardizována, takže jakmile se je jednou naučíte, lze je používat v prostředí různých databázových systémů.

11.4.1 Deriváty SQL

Jak jsme již několikrát zmínili, jazyk SQL je nejznámější dotazovací jazyk, a proto se i mnoho NoSQL databází snaží nabídnout možnost používat alespoň podmnožinu tohoto jazyka nebo jazyk inspirovaný syntaxí a přístupem SQL. V knize jsme takových příkladů viděli několik – například HiveQL nadstavbu nad systémem Hadoop (viz sekce 4.3.3.3) a CQL v systému Cassandra (viz sekce 8.4.1). Využití syntaxe SQL je možné zejména v případech, kdy je možné data, s nimiž pracujeme, mapovat na relační datový model. Pro jiné datové modely není jazyk SQL navržen a není tudíž příliš vhodný.

11. Dotazování nad NoSQL
databázemi

11.4.2 Rozšíření SQL

Jazyk SQL lze nicméně použít i v případech, kdy chceme pracovat s nerelačními daty. Obvyklým řešením je doplnit do jazyka SQL nové funkce, které pracují s daty v jiném formátu a nabízejí prostředky, jak je převádět do relačního modelu a zpět. Na tomto principu pracuje například SQL/XML [22] – rozšíření jazyka SQL o podporu práce s daty ve formátu XML. Do jazyka SQL byl přidán nový datový typ XML a řada funkcí, které umějí relační data vracet v podobě tohoto nového

datového typu a naopak umí data z datového typu XML extrahovat a používat je dále v dotazu.

Tuto techniku ukážeme na konkrétním příkladu. Předpokládejme, že máme v relační databázi tabulku **Conferences**, kde je v každém záznamu ve sloupci **Detail** v datovém typu XML uložena informace o jedné konferenci následujícím způsobem:

```
<conference>
    <name>XML Prague 2015</name>
    <start>2015-02-13</start>
    <end>2015-02-15</end>
    <web>http://xmlprague.cz/</web>
    <price currency="EUR">120</price>
    <topic>XML</topic>
    <topic>XSLT</topic>
    <topic>XQuery</topic>
    <topic>Big Data</topic>
    <venue>
        <name>VŠE Praha</name>
        <location lat="50.084291" lon="14.441185"/>
    </venue>
</conference>
```

Nyní budeme chtít v SQL napsat dotaz, který nám vypíše názvy, začátek a cenu konferencí, které mají jako jedno z témat „Big Data“. Využijeme k tomu SQL/XML funkci **XMLELEMENT()**, jež provede XQuery dotaz a jeho výsledky namapuje na relační tabulku:

```
SELECT result.name, result.from, result.price
FROM
    Conferences conf,
    XMLTABLE('for $conf in /conference
              where $conf/topic = "Big Data"
              return
              <result>
                  { $conf/name }
                  { $conf/start }
                  { $conf/price }
              </result>' PASSING conf.Detail
    COLUMNS
        "name" VARCHAR(30) PATH '/result/name',
        "from" DATE PATH '/result/start',
        "price" DECIMAL PATH '/result/price',
    ) AS result;
```

Funkce **XMLELEMENT()** jako první argument obdrží vnořený dotaz v XQuery. Klauzule **PASSING** říká, jaká data se dotazu předají – v našem případě obsah sloupce **Detail**. Klauzule **COLUMNS** pak definuje, jakými hodnotami naplnit sloupce výsledku – určujeme přitom datový typ a pomocí cesty v XPath vybíráme hodnotu k předání z výsledku dotazu XQuery. (Možnosti propojení světa XML a SQL jsou v SQL/XML podstatně bohatší, viz například [127].)

Jak jsme v knize několikrát viděli, nejpopulárnějším dokumentovým formátem je dnes JSON. Na to reagují i výrobci databází a do svých produktů přidávají jeho podporu. V roce 2014 dokonce začaly práce na standardizaci podpory a integrace formátu JSON do jazyka SQL.² Nicméně vytvoření standardu nějakou dobu trvá, takže na fungující podporu SQL/JSON budeme muset ještě několik let počkat. Do té doby můžeme používat to, co nabízí relační databáze již dnes. Například v sekci 14.1.1.1 se podrobněji podíváme na podporu formátu JSON v databázi PostgreSQL. JSON lze používat jako nový datový typ a sloupec s tímto typem tak funguje jako dokumentová databáze. PostgreSQL pak samozřejmě také rozšiřuje svůj dialekt SQL o práci s uloženými JSON objekty.

S podporou formátu JSON nezůstávají pozadu ani komerční databáze jako Oracle nebo MS SQL Server.³ Například Oracle jako jeden ze spolutvůrců SQL/JSON podporuje syntaxi, která odpovídá pracovním verzím připravovaného standardu. Pokud bychom například měli informace o konferencích uložené v položce `Detail` typu JSON jako:

```
{
  "name": "XML Prague 2015",
  "start": "2015-02-13",
  "end": "2015-02-15",
  "web": "http://xmlprague.cz/",
  "price": 120,
  "currency": "EUR",
  "topics": ["XML", "XSLT", "XQuery", "Big Data"],
  "venue": {
    "name": "VŠE Praha",
    "location": {
      "lat": 50.084291,
      "lon": 14.441185
    }
  }
}
```

mohli bychom informace o názvu, začátku a ceně všech konferencí zjistit pomocí následujícího dotazu:

```
SELECT json_value(Detail, '$.name'),
       json_value(Detail, '$.start'),
       json_value(Detail, '$.price')
FROM Conferences
```

Pro výběr částí JSON objektu se přitom uvnitř funkce `json_value()` používá syntaxe jazyka JsonPath [93], který byl inspirován dotazovacím jazykem XPath pro XML.

² Bohužel je přístup k pracovním dokumentům většiny pracovních skupin v rámci standardizační organizace ISO/IEC možný pouze pro její členy. Nicméně základní informace o přípravě normy SQL/JSON lze získat z prezentace dostupné na adrese http://jtc1bigdatasg.nist.gov/_workshop/08_SQL_Support_for_JSON_abstract.pdf. Pro opravdové zájemce o další informace pak existuje jediná cesta – domluvit se s Úřadem pro technickou normalizaci, metrologii a státní zkušebnictví (<http://www.unmz.cz>), aby vás nominoval jako expertsa do pracovní skupiny ISO/IEC JTC1 SC32 WG3.

³ <http://www.microsoft.com/sqlserver/>

11.4.3 XQuery

XQuery [139] je standardizovaný dotazovací jazyk pro dokumenty XML. Je podporován všemi XML databázemi a ve světě XML tak plní podobnou roli jako SQL ve světě relačním. Samotné XQuery rozšiřuje dotazovací XPath o další konstrukce, které dovolují zapisovat složitější dotazy. Základem jsou FLWOR výrazy – zkratka přitom vychází z počátečních písmen názvů nejdůležitějších klíčových slov jazyka: **FOR-LET-WHERE-ORDER BY-RETURN**.

Předpokládejme, že máme v databázi uloženy informace o konferencích ve výše uvedené struktuře. Vypsání konferencí o NoSQL, jejichž cena je mezi 100 a 200, a jejich seřazení podle ceny můžeme snadno provést pomocí následujícího dotazu:

```
for $conf in /conference
let $price := $conf/price
where contains($conf/name, 'NoSQL') and ($price ge 100) and ($price le 200)
order by $price
return $conf
```

Klauzule **for** přitom vybírá položky ke zpracování, pomocí klauzule **let** můžeme vytvářet proměnné, které zjednoduší další zápis, a klauzule **where** funguje jako klasický filtr. O řazení hodnot se postará klauzule **order by** a konečně klauzule **return** definuje, co má být výsledkem dotazu. V našem případě kompletní kopie celého dokumentu jedné konference, která vyhovuje zadaným podmínkám.

Nicméně pomocí klauzule **return** můžeme konstruovat libovolný výstup, třeba nový element, který bude obsahovat cenu a název konference v attributech:

```
for $conf in /conference
let $price := $conf/price
where contains($conf/name, 'NoSQL') and ($price ge 100) and ($price le 200)
order by $price
return
<konference název="{$conf/name}" cena="{$conf/price}"/>
```

Uvedený příklad hledá u řetězce **NoSQL** přesnou shodu. Pokud bychom chtěli využít fulltextové vyhledávání, které umí hledat různé gramatické tvary slov apod., můžeme použít rozšíření XQuery and XPath Full Text [56]. Získáme tak chování podobné klauzuli **matches** v Elasticsearch Query DSL. Například následující dotaz vrátí všechny konference o databázích, i kdyby toto slovo bylo v názvu uvedeno v jiném než prvním pádu:

```
for $conf in /conference
where ($conf/name contains text ('databáze' using stemming))
return $conf
```

XQuery bylo původně navrženo jen pro práci s XML. Nicméně stejně tak jako se SQL snaží integrovat další datové modely a obohacuje se postupně o podporu XML (v podobě SQL/XML) a JSON (v podobě SQL/JSON), snaží se i XQuery integrovat data z dalších zdrojů. Jednotlivé XML databáze často nabízejí vlastní rozšiřující funkce, které z prostředí XQuery dotazů zpřístupňují relační i JSON data. Většina implementací přitom data mapuje na strukturu XML, se kterou umí XQuery přirozeně pracovat.

Pro ještě přesnější mapování přinese nová verze XQuery 3.1 dva nové datové typy – pole a asociativní pole. Díky tomu bude možné JSON mapovat na datový model XQuery přímočaře a otevřá se tak prostor pro těsnější integraci obou datových modelů v jednom dotazovacím jazyce. Stejná funkcionality pak bude tradičně dostupná i v samotném dotazovacím jazyce XPath a v transformačním jazyce XSLT [55].

11.4.4 JSONiq

Velice zajímavým počinem je dotazovací jazyk JSONiq [140]. Jedná se o dotazovací jazyk speciálně určený pro JSON, silně inspirován jazykem XQuery. Ostatně mnoho z autorů JSONiq se podílelo i na vývoji jazyka XQuery.

Podobnost s XQuery ilustruje následující příklad, který vrátí pole objektů s údaji o názvu a ceně NoSQL konferencí:

```
for $conf in collection('conferences')
let $price := $conf.price
where contains($conf.name, 'NoSQL') and ($price ge 100) and ($price le 200)
order by $price
return
{ "název": $conf.name,
  "cena": $conf.price }
```

Jak vidíme, pro přístup k vlastnostem objektu se používá tečková notace. V klauzuli `return` můžeme přímo generovat nové struktury JSON.

Kromě verze zaměřené čistě na JSON existuje i verze JSONiq, která je integrovaná s XQuery a dovoluje pokládání dotazů nad XML a JSON zároveň.

JSONiq by byl jistě pěkný kandidát na standardizovaný dotazovací jazyk pro NoSQL úložiště. Jestli se tak stane, ukáže čas. V současné době existuje jeho open source implementace Zorba⁴ a společnost 28msec⁵ nabízí JSONiq konektory pro různá NoSQL úložiště jako MongoDB, CouchBase,⁶ Elasticsearch nebo Amazon S3.

11.4.5 SPARQL

Velmi specifickou oblastí dotazování jsou dotazovací jazyky pro grafová data. Podrobněji se jimi budeme zabývat v sekci 13.4.3. Širším nasazením a větším počtem implementací se z těchto jazyků může chlubit jen jazyk SPARQL určený pro dotazy nad RDF grafy.

Pro porovnání SPARQL s ostatními jazyky ukážeme dotaz, který by opět našel název a cenu konferencí věnovaných NoSQL s cenou mezi 100 a 200. Předpokládáme přitom, že RDF graf bude obsahovat informace o konferencích ve stejné podobě jako v příkladu 2.7 na straně 41.

⁴ <http://www.zorba.io>

⁵ <http://www.28.io>

⁶ <http://www.couchbase.com>

PREFIX schema: <http://schema.org/>

```
SELECT ?name ?price
WHERE {
    ?conf a schema:Event ;
          schema:name ?name ;
          schema:description ?desc ;
          schema:offers ?offer .
    ?offer schema:price ?price .
  FILTER (
    ?price > 100 &&
    ?price < 200 &&
    REGEX(?desc, 'Big Data')
  )
}
ORDER BY ?price
```

11.5 Závěr

Oblast dotazování pro NoSQL databáze se teprve vyvíjí, jelikož samotné NoSQL databáze jsou relativně nová oblast. Dá se tedy očekávat, že postupem času budou uživatelé vyžadovat pohodlnější nástroje pro tvorbu dotazů a stále více se budou používat dotazovací jazyky odvozené od SQL nebo XQuery, které uživatele odstíní od nutnosti psát dotazy ručně jako procedurální kód nebo funkce pro MapReduce framework.

12.

Transakce v distribuovaném prostředí

V kapitole 3 jsme vysvětlili několik základních principů transakčního zpracování v distribuovaných systémech tak, abychom mohli vzápětí představit jednotlivé typy NoSQL databází. Problematika transakcí v distribuovaném prostředí je ale mnohem složitější a existuje několik přístupů, které je možné využít [94] [66] [134] a které blíže představíme v této kapitole.

12.1 Vlastnosti CAP podrobněji

Podívejme se nejprve podrobněji na základní vlastnosti, které od distribuovaného systému očekáváme. Jak již víme, CAP teorém hovoří o konzistenci (*consistency*), dostupnosti (*availability*) a odolnosti vůči rozpadu sítě (*partition tolerance*), přičemž všechny tyto tři vlastnosti nejsme schopni v distribuovaném systému zajistit na 100 % (viz sekce 3.2.2). Hovoříme-li o distribuci, pak nás samozřejmě zajímá především odolnost vůči rozpadu sítě, tedy situace, kdy chceme, aby systém fungoval, i když dojde k přerušení spojení mezi některými uzly v clusteru. Mezi

zbylými dvěma vlastnostmi, tedy dostupností a konzistencí, pak musíme najít vhodný kompromis.

Jak jsme již uvedli v sekci 3.2, v oblasti NoSQL databází často netrváme na silné konzistenci dat, ale stačí nám konzistence občasná (*eventual consistency*). Jedná se o situaci, kdy jsou data jednou za čas nebo v případě, že je to nutné, uvedena do konzistentního stavu, ale permanentně v něm být nemusí. Jedním z hlavních způsobů, jak zajistit konzistenci dat, jsou právě transakce. Různé transakční modely, tedy modely řízení transakcí, představíme v této kapitole. Poznamenejme ještě, že typů konzistence dat je mnohem více. V různých systémech můžeme nalézt pojmy jako sekvenční, kauzální, pomalá, obecná, lokální nebo slabá konzistence.

Dostupnost systému se obvykle vyjadřuje jako procentuální podíl času, kdy má systém přijatelnou odezvu, k celkovému nabízenému času. Jak je vidět v tabulce 12.1, z hlediska dostupnosti rozlišujeme několik tříd, které odpovídají různorodým nárokům konkrétních systémů [95].

Tabulka 12.1: Třídy dostupnosti

Třída	Dostupnost
1 = unmanaged	90 %
2 = managed	99 %
3 = well managed	99,9 %
4 = fault-tolerant	99,99 %
5 = high availability	99,999 % (např. jaderný reaktor)
6 = very high availability	99,9999 % (např. telefonní ústředna)
7 = ultra availability	99,99999 % (např. systémy v letadlech)

U každého, tedy nejen distribuovaného systému nás pak ještě zajímá především jeho *odolnost vůči chybám* (*fault tolerance*). V konkrétním systému může nastat množství chyb způsobených chybou v kódu programátora nebo selháním některého z modulů systému. *Latentní* je chyba, která se ještě neprojevila, zatímco *efektivní* chyba je taková, která se již projevila. Chyby můžeme také dělit na *slabé* (*soft*), z nichž se systém dokáže zotavit, a *silné*, pokud ne. Dostupnost systému je pak možné definovat vzorcem $MTTF / (MTTF + MTTR)$, kde *MTTF* je průměrný čas do příští chyby (*mean time to failure*) a *MTTR* je průměrný čas na zotavení z ní (*mean time to repair*).

12.2 Základní transakční modely

Abychom se mohli zaměřit na problematiku transakcí v distribuovaném prostředí, nejprve pro úplnost a lepší zasazení do kontextu krátce připomeneme základní transakční modely v nedistribuovaných systémech [94]. V distribuovaném prostředí jsou pak mnohé z nich využívány, nicméně je nutné je přizpůsobit dalším situacím, které mohou v tomto prostředí nastat.

12.2.1 Ploché transakce

Pravděpodobně nejpopulárnějším transakčním modelem jsou *ploché transakce* (*flat transactions*). Kromě příkazů provádějících samotnou transakci obsahují operace **BEGIN** (začátek transakce), **COMMIT** (úspěšné dokončení transakce, tzv. *potvrzení transakce*) a **ABORT**, respektive **ROLLBACK** (neúspěšné dokončení transakce, a tedy odvolání dosud provedených změn, tzv. *zrušení transakce*). Ploché transakce mají základní vlastnosti ACID (atomicita, konzistence, izolovanost a trvalost) a jsou velmi populární především v klasických relačních databázových systémech. Tento model má ovšem několik zásadních nevýhod:

- Je-li **ABORT** vyvolán po změnách velkého množství dat, znamená to typicky opět velké množství operací odvolání provedených změn.
- Ploché transakce nemohou spolupracovat s jinými transakcemi.
- Nejsou tolerovány dílčí neúspěchy, a to ani v případech, kdy by neměly vliv na ostatní operace.

První z uvedených problémů se často řeší pomocí *bodů návratu* (*save points*), které umožňují odvolání změn pouze do určitého místa aktuální transakce. Body návratu ovšem neřeší situaci, kdy dojde k výpadku celého systému a tudíž celá nedokončená transakce musí být zopakována. Body návratu totiž nejsou persistentní. Jinak bychom porušili podmínku atomicity ploché transakce, tedy podmínku „všechno nebo nic“.

12.2.2 Zřetězené transakce

Dalším typem transakcí jsou *zřetězené transakce* (*chained transactions*), které zavádějí novou operaci **CHAIN WORK**. Úkolem této operace je (atomicky) provést dvojici operací **COMMIT WORK** (úspěšné ukončení) aktuální transakce a **BEGIN WORK** (zahájení) transakce nové. Mezi těmito dvěma transakcemi není nutné uvolnit používané objekty, např. uzavřít databázové kurzory, a současně je možné provést **ABORT** pouze aktuálně poslední transakce v řetězci.

Speciálním případem zřetězených transakcí jsou tzv. *ságy*. Sága je vlastně sekvence transakcí, z nichž každá smí provést přímo klasický **COMMIT**. Pokud by došlo k situaci, že v určité fázi ságy je ale třeba provést **ABORT** celé ságy vzhledem k nutnosti zrušení jedné z jejích transakcí, je třeba zrušit také již potvrzené transakce ságy. K tomu se využívá princip *kompenzační transakce*, tedy transakce, která dokáže vrátit zpět efekty jiné transakce a navíc musí vždy skončit úspěšně. Jinými slovy, u ságy předpokládáme, že každá její transakce má také svou kompenzační transakci.

12.2.3 Hnizděně transakce

Ještě složitější, i když méně často používanou strukturu pak nabízejí *hnizděně* neboli *vnořené transakce* (*nested transactions*). Místo toho, aby se dívaly na transakce jen jako na sekvenci operací (resp. transakcí), považují je za libovolnou

hierarchii operací (resp. transakcí). Hnízděná transakce je tedy strom transakcí, jejíž podstromy odpovídají hnízděným transakcím (tvořícím vnitřní uzly stromu) nebo plochým transakcím (tvořícím listy stromu). Jinými slovy, v rámci aktuální transakce zavoláme další operaci `BEGIN`, která zahájí novou transakci hnízděnou do transakce aktuální.

Transakce nejvyšší úrovně (top level transaction) je transakce, která není hnízděná do žádné jiné transakce. Oproti tomu podtransakce (*subtransaction*) má vždy právě jednu rodičovskou transakci, jejíž je součástí.

Podtransakce může provést `COMMIT` nebo `ABORT`, nicméně dokud není proveden `COMMIT` transakce nejvyšší úrovně, neprojeví se změny podtransakcí v systému – jsou viditelné pouze pro rodičovskou transakci. Naopak provede-li některá transakce `ABORT`, musí ho provést i všechny její podtransakce v hierarchii. Podmínky na `COMMIT` transakce se liší. Obvykle vyžadujeme, aby každá transakce mohla provést `COMMIT` pouze v případě, že provedou `COMMIT` všechny její podtransakce. V některých modelech jsou ale povoleny částečné akce `ABORT` v podtransakcích.

Speciálním případem hnízděných transakcí jsou pak *transakce víceúrovňové (multi-level transactions)*. V tomto typu transakcí je povolen pre-`COMMIT`, tedy možnost uvolnění objektů, s nimiž transakce pracuje. Abychom ale mohli i takovou transakci případně zrušit, musí i v tomto případě existovat odpovídající kompenzační transakce.

12.3 Transakce v distribuovaném prostředí

V distribuovaném prostředí je transakční zpracování komplikované tím, že různé části transakce probíhají na různých uzlech v clusteru, na nichž jsou uložena příslušná data. Rozlišujeme tedy *globální* (neboli *distribuované*) *transakce* a *lokální transakce*. Lokální transakce jsou klasické transakce, které pracují pouze s daty uloženými v lokálním úložišti. Distribuovaná transakce pracuje s daty umístěnými na dvou nebo více uzlech v clusteru, které distribuované transakci nabízejí k dispozici své zdroje, typicky prostřednictvím tzv. *manažera zdrojů* (*resource manager*). Jinými slovy, globální transakci si lze představit jako klasickou transakci, která ale probíhá na více uzlech v clusteru.

Stejně jako v lokálním úložišti, i koordinaci distribuovaných transakcí zajišťuje *manažer transakcí* (*transaction manager*). Má na starost spouštění, ukončování i restartování transakcí. Jeho práce je ale v distribuovaném prostředí složitější právě díky distribuci dat a s tím souvisejícími známými problémy.

Pro řízení a koordinaci distribuovaných transakcí se používá několik typů strategií. Ty nejpopulárnější představíme v následujících sekcích. Jejich hlavním cílem je, stejně jako v nedistribuovaném prostředí, zajistit *uspořadatelnost* a případně i *zotavitelnost*, tedy schopnost systému korektně prokládat operace v transakcích a schopnost návratu do posledního korektního stavu při nastalé chybě.

12.3.1 2PC protokol

Pravděpodobně nejznámějším přístupem pro řízení transakcí v distribuovaném prostředí je *dvoifázový COMMIT protokol* (*two-phase commit protocol*, 2PC protokol), někdy také *dvoifázový potvrzovací protokol*. Jedná se o algoritmus, který koordinuje všechny procesy podléhající se na distribuované transakci a zajišťuje, že dojde ke korektnímu provedení operace **COMMIT** nebo **ABORT** distribuované transakce vzhledem k (ne)úspěšnosti provedení jejích jednotlivých částí. Předpokládáme, že distribuovanou transakci řídí uzel zvaný *koordinátor* a jednotlivé lokální transakce probíhají na uzlech nazývaných *účastníci* transakce, na nichž jsou uloženy příslušné fragmenty dat.

Jak už název napovídá, po provedení poslední operace distribuované transakce proběhnou dle 2PC protokolu dvě fáze:

- 1. Hlasovací fáze:** Koordinátor distribuované transakce zašle všem účastníkům zprávu **PREPARE**, tedy se zeptá, zda jimi realizovaná lokální transakce úspěšně proběhla a může proběhnout **COMMIT**. Vzhledem k tomu, že celá distribuovaná transakce může úspěšně skončit pouze v případě, že proběhnou všechny její lokální části, je třeba pozitivní odpověď od všech. V opačném případě není možné distribuovanou transakci úspěšně dokončit a je třeba ji zrušit. Pokud účastník odpoví pozitivně, změní svůj stav na *čekající*, tj. zablokuje všechny objekty, s nimiž daná transakce pracuje, a čeká na další pokyn. Pokud odešle negativní odpověď, provede operaci **ABORT**, která lokální transakci zruší a příslušné objekty uvolní.
- 2. Fáze COMMIT:** Na základě výsledků z předchozí fáze zašle koordinátor všem účastníkům pokyn k provedení operace **COMMIT** nebo **ABORT**. Účastníci odpovídající operaci provedou.

Výše uvedený postup má pochopitelně své nevýhody. Jelikož se nacházíme v distribuovaném prostředí, může docházet k přerušení spojení, ale obecně také k selhání jednotlivých uzlů v clusteru. V první řadě po odeslání zprávy **PREPARE** koordinátor očekává, že všichni účastníci odpoví, což se nemusí stát a dojde tak *zablokování koordinátora*. Naopak poté, co účastník odešle informaci o úspěchu aktuální transakce, čeká na pokyn k provedení operace **COMMIT** nebo **ABORT**. Pokud v tomto okamžiku dojde k výpadku koordinátora, dojde k *zablokování (některých) účastníků* transakce.

Jedním z možných řešení je využít časového limitu, do něhož musí uzel odpovědět, jinak je jeho odpověď považována za negativní. To ale řeší pouze problém zablokování koordinátora, který tím pádem nebude čekat na odpovědi všech účastníků nekonečně dlouho. Zablokování účastníka ale umí řešit až rozšíření protokolu na *třífázový COMMIT protokol* (viz sekce 12.3.2).

Poznamenejme ještě, že dalšími rozšířeními 2PC protokolu může být odhadování výsledku distribuované transakce na základě chování jednotlivých procesů, díky němuž je možné ušetřit velké množství zpráv. A existuje také hierarchická varianta tohoto protokolu, tedy vlastně jakási obdoba hnízděných transakcí, kdy může účastník místo lokální transakce vyvolat další distribuovanou transakci řízenou 2PC protokolem.

12.3.2 3PC protokol

Jak bylo uvedeno, hlavním problémem 2PC protokolu je právě selhání koordinátora v okamžiku, kdy účastník zablokuje objekty své lokální transakce a čeká na pokyn koordinátora, jakým způsobem ji má ukončit. *Třífázový COMMIT protokol* (3PC protokol) [147] proto 2PC protokol dále rozšiřuje:

1. *Hlasovací fáze* probíhá podobně jako v předchozím případě, tj. koordinátor distribuované transakce zašle všem účastníkům zprávu **PREPARE**, kterou se ptá, zda jimi realizovaná lokální transakce úspěšně proběhla a může proběhnout **COMMIT**.
2. *Fáze pre-COMMIT*: Pokud dojde k selhání (např. k výpadku komunikace), vyprší časový limit nebo některý z účastníků vrátí negativní odpověď, koordinátor vyšle všem účastníkům pokyn k provedení **ABORT**. (Ten je proveden také v případě, že koordinátor v časovém limitu žádný pokyn nevyšle.) Jinak, tedy v případě, že od všech účastníků dostane včas pozitivní odpověď, pošle všem pokyn **pre-COMMIT** a čeká na jeho potvrzení. Transakce bude úspěšně dokončena pouze v případě, že všichni účastníci pokyn potvrdí do časového limitu.
3. *Fáze COMMIT*: Na základě výsledků z předchozí fáze zašle koordinátor všem účastníkům pokyn k provedení operace **COMMIT** nebo **ABORT**. Pokud koordinátor selže, tedy někteří účastníci nedostanou finální pokyn k operaci **COMMIT** do časového limitu, provedou jej automaticky.

Přidáním fáze, kdy účastníci potvrdí, že jsou připraveni provést operaci **COMMIT** (tedy fáze **pre-COMMIT**), se protokol vyhne situaci, že by některý z účastníků provedl **COMMIT** bez toho, aby se všichni dozvěděli, že k němu má dojít. Přidáním časových limitů a automatického provedení **COMMIT** pak protokol řeší problém selhání jednotlivých uzlů v clusteru.

12.4 Optimistické a pesimistické off-line zámky

Stejně jako v lokálních úložištích, i v distribuovaných databázích existují metody, které využívají zámky. V klasickém pojetí transakcí je **zámek** chápán jako atribut, který zajišťuje exkluzivní nebo řízeně sdílený přístup k danému objektu v úložišti. (Ve složitějších systémech se může jednat i o celé množiny nebo hierarchie objektů.) *Exkluzivní zámek* zajišťuje, že s daným objektem může pracovat (může jej číst nebo ho modifikovat) pouze jediný proces. *Sdílený zámek* zajišťuje, že daný objekt může číst více procesů, ale žádný jej nesmí modifikovat. Pro získání zámku přibývá do transakce operace **LOCK**, pro jeho uvolnění operace **UNLOCK**. Dále předpokládáme, že vždy, když chce transakce s nějakým objektem pracovat, tak ho zamkne, a po skončení práce (nejpozději při skončení transakce) ho opět odemkne.

Princip zámků se v NoSQL databázích využívá také [144], např. v případě tzv. *business transakcí*, tedy transakcí, které sestávají z několika *systémových transakcí*. Systémová transakce odpovídá jedné sekvenci operací nad vybranou množinou dat, která obvykle končí ukončením interakce s uživatelem. Business transakce se pak skládá z několika systémových transakcí. Např. jedna business transakce

může odpovídat situaci, kdy si uživatel prohlíží katalog výrobků, jeden z nich si vybere, provede objednávku, zadá údaje z kreditní karty a objednávku uhradí. V průběhu této business transakce probíhají různé požadavky na různá data uložená na různých místech, které ovšem nevyžadují, aby byla po celou dobu veškerá dotčená data uzamčena. Rozdelením této transakce na několik systémových transakcí tento problém vyřešíme – dotčená data budou uzamčena vždy jen po nezbytně nutné době. Nicméně při takovém rozdelení může nastat situace, kdy budeme číst nebo modifikovat nepotvrzená data (viz konfliktní operace popsané v sekci 3.2.1), tedy určité výpočty nebo rozhodnutí můžeme provádět na základě neplatných dat. Např. se v průběhu business transakce mohou změnit ceny výrobků, může proběhnout změna adresy zákazníka apod. Pro řešení této situace se používají dva přístupy – tzv. optimistické nebo pesimistické off-line zamykání (*optimistic resp. pessimistic off-line lock*).

12.4.1 Optimistický přístup

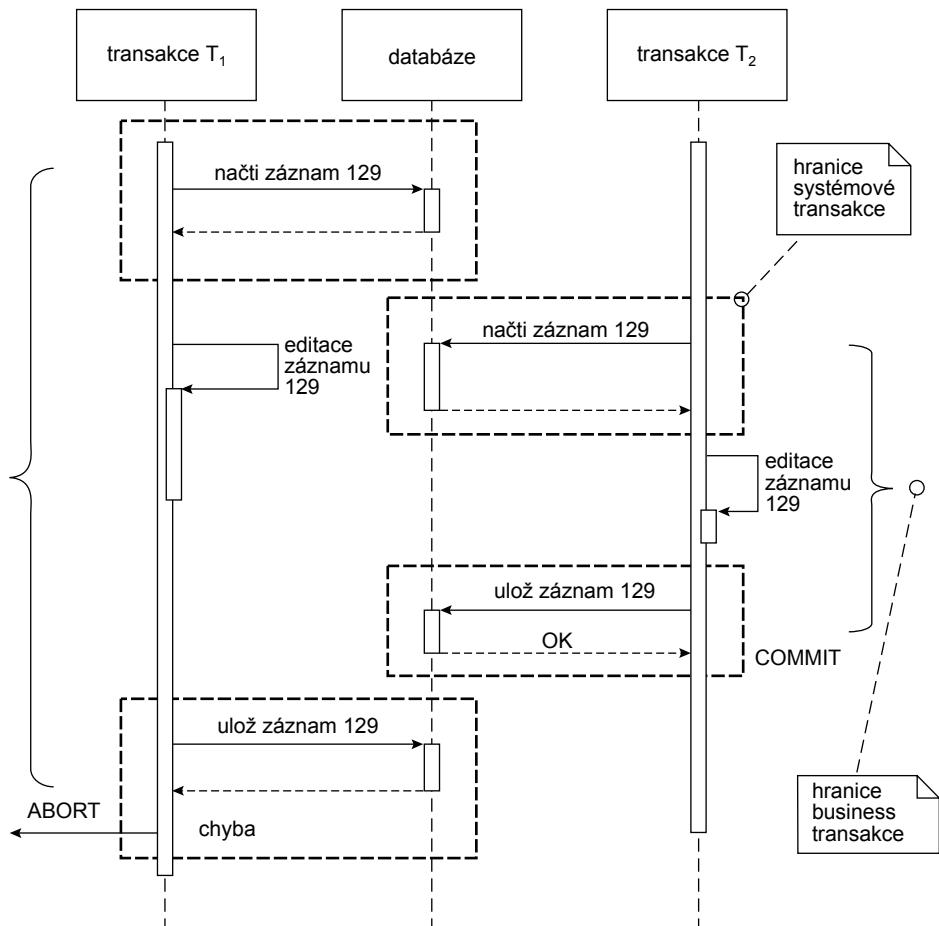
Optimistické off-line zamykání předpokládá, že ke konfliktní situaci příliš často nedochází. Uvažme situaci na obrázku 12.1 na následující straně, kde se snaží se stejným objektem v databázi pracovat dvě různé transakce. Transakce T_1 nejprve načte hodnotu záznamu. Během toho, co ji lokálně modifikuje, tuto hodnotu načte také transakce T_2 a navíc ji modifikuje a tuto změnu uloží do databáze. Transakce T_1 chce nyní novou hodnotu také uložit do databáze, což ale pochopitelně není žádoucí. Optimistické off-line zamykání proto přidává do operace zápisu zamčení ukládaného objektu, tedy kontroly, zda se data v mezičase nezměnila, a tudíž bude možné **COMMIT** provést. Někdy se tato fáze nazývá lokální pre-COMMIT. Pokud kontrola neprojde, příslušná transakce je zrušena.

Optimistický je tento přístup proto, že se problému nesnažíme předcházet, což může být náročné na ošetření a v určitých situacích zbytečné. Až když je to nezbytně nutné, zjišťujeme, zda k problému nedošlo. Pokud ano, vyřešíme ho, byť zrušením celé transakce. Optimisticky předpokládáme, že toto nenastává příliš často.

12.4.2 Pesimistický přístup

Pesimistické off-line zamykání přistupuje k této situaci zcela opačně. Předpokládá, že k problematickým situacím dochází často a je tudíž naopak lepší jím předcházet, než aby docházelo k neustálému rušení transakcí, které mohou být poměrně rozsáhlé. Situace je demonstrována na obrázku 12.2 na straně 213, kde vidíme, že jakmile jednou transakce T_1 záznam zamkne exkluzivním zámkem, dostane T_2 při pokusu o jeho čtení chybové hlášení. O problému se tedy dozví okamžitě.

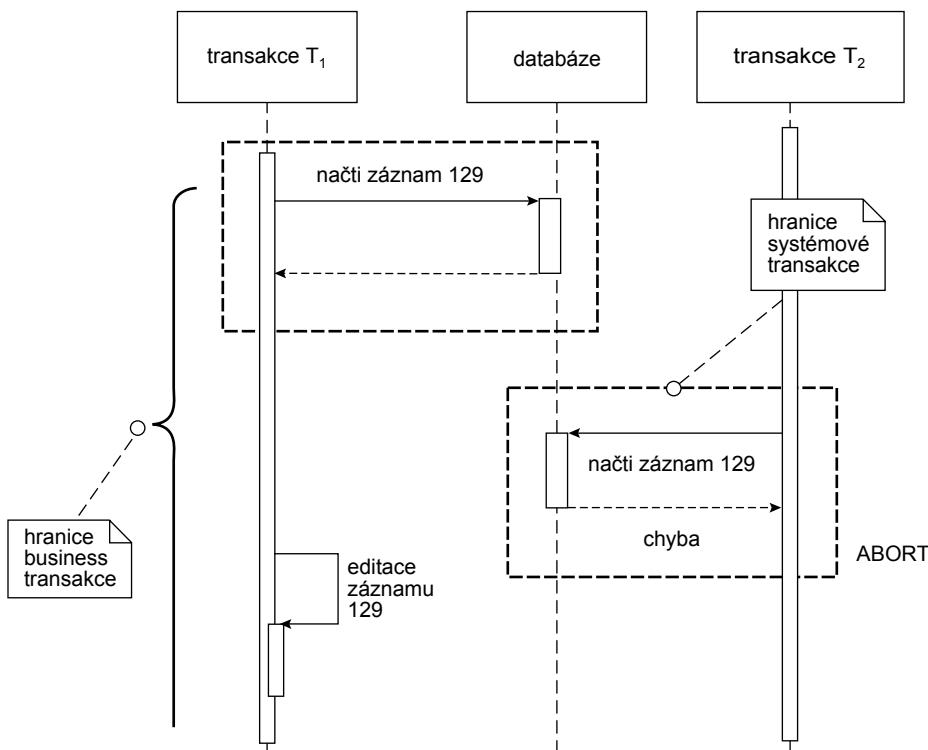
Jednou z nevýhod pesimistického přístupu k zamykání je ovšem klasický problém zamykacích protokolů: *uváznutí* (*deadlock*), tedy situace, kdy dvě nebo více transakcí vzájemně čekají na uvolnění zámku od jiné transakce a žádná z nich nemůže provádět další operace. Tuto situaci je třeba nejprve detekovat a následně jednu ze vzájemně čekajících transakcí ukončit (a spustit později, až budou příslušné objekty k dispozici).



Obrázek 12.1: Optimistické off-line zamykání

Detekce uváznutí se provádí vybudováním *grafu čekajících transakcí* (*wait-for graph*), jehož uzly tvoří jednotlivé transakce a orientovaná hrana vede z uzlu T_1 do uzlu T_2 , pokud transakce T_1 čeká na uvolnění zámku od transakce T_2 . Cyklus v tomto grafu odpovídá právě situaci uváznutí transakcí.

Obvyklým způsobem řešení uváznutí je jeho předcházení s využitím principu *časových razítok*. Každá transakce má přiřazeno časové razítko svého začátku. Pokud se transakce T_2 pokusí získat zámek, který má aktuálně transakce T_1 , nejprve dojde ke kontrole časových razítok. Pokud je časové razítko T_2 menší než razítko T_1 , může T_2 čekat, v opačném případě je ukončena. Tato strategie se v literatuře nazývá *wait-die*. Existuje také strategie *wound-wait*, v níž je podmínka na časová razítka opačná. Obě situace nicméně každopádně zajistí, že nedojde k vzájemnému cyklickému čekání na uvolnění zámků, jelikož vždy může čekat jedině starší, resp. jedině mladší transakce.



Obrázek 12.2: Pesimistické off-line zamykání

12.5 Uspořádání časových razítek

Využití časových razítek je možné nalézt také v metodách *optimistického*, resp. *pesimistického uspořádání časových razítek* (*optimistic*, resp. *pessimistic timestamp ordering*), které zajišťují, že nečekáme na uvolnění zámku a tedy nemůže dojít k uváznutí. Naproti tomu ale musíme zajistit, že máme v celém systému totální uspořádání časových razítek, tj. pro libovolná dvě časová razítka v systému umíme určit jejich vzájemné pořadí v čase. Tuto podmínuje je možné zařídit např. pomocí *Lamportových hodin* [121]. Ty sice neudržují přesný čas, ale ten nepotřebujeme. V distribuovaném systému potřebujeme dodržovat pouze pořadí operací. U Lamportových hodin hovoříme o *logickém čase*, tedy informaci o tom, která z operací nastala dříve.

12.5.1 Pesimistické uspořádání

V pesimistickém přístupu předpokládáme, že má každá transakce T své časové razítko $T.start$ začátku práce. Každý objekt O v databázi má své časové razítko zápisu $O.wts$ a časové razítko čtení $O.rts$ udržující informaci o časovém razítku té transakce, která daný objekt naposledy modifikovala nebo četla. Operace v transakci pak probíhají takto:

- Pokud dojde v transakci T ke čtení z objektu O , jsou nejprve porovnána časová razítka. Pokud $T.start < O.wts$, tedy transakce T se snaží číst data, která byla

modifikována novější transakcí, je T *restartována* (tedy zrušena a znova spuštěna s novým časovým razítkem). V opačném případě je čtení provedeno a hodnota $O.rts$ je nastavena na hodnotu $\max(O.rts, T.start)$.

- Pokud dojde v transakci T k zápisu do objektu O , jsou také nejprve porovnána časová razítka. Pokud $T.start < O.rts$ nebo $T.start < O.wts$, tedy transakce T se snaží modifikovat data, která byla čtena nebo modifikována novější transakcí, je T restartována. V opačném případě je zápis proveden a $O.wts = T.start$.

Tento přístup je pesimistický v tom smyslu, že pokud se transakce T pokusí přistoupit na objekt, s nímž pracovala novější transakce, okamžitě je T restartována. Např. v případě využití zámků by došlo pouze k čekání na uvolnění zámku (které by sice teoreticky mohlo skončit uváznutím, ale obvykle ne ve všech případech). Na druhou stranu v pesimistickém uspořádání časových razítek máme jistotu, že nikdy uváznutí nenastane.

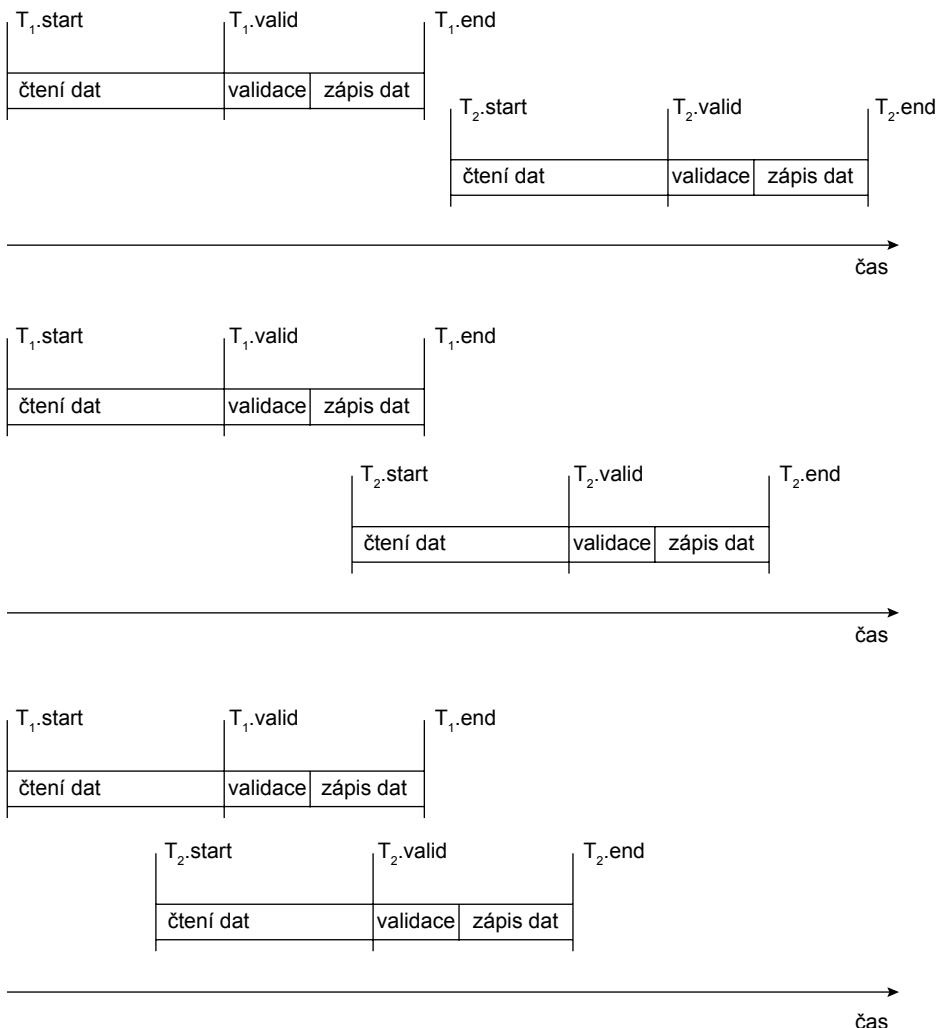
12.5.2 Optimistické uspořádání

V optimistickém přístupu předpokládáme, že ke konfliktním situacím nedochází příliš často a jejich řešení můžeme odložit na nezbytně nutnou dobu. Každá transakce T s časovým razítkem počátku $T.start$ načte požadovaná data z úložiště, ale jejich modifikace provádí ve svém lokálním pracovním prostoru. Pokud má dojít ke zrušení transakce T , znamená to, že T pouze uvolní svůj lokální pracovní prostor. Má-li dojít k potvrzení transakce T , dojde k *validaci transakce*, tedy kontrole konfliktů vzhledem ke všem transakcím, které se s T časově překrývají. Teprve pokud validace proběhne v pořádku, jsou změny promítnuty z lokálního pracovního prostoru do dat v úložišti.

Přesněji řečeno předpokládáme, že má každá transakce T tři časy: čas začátku $T.start$, čas ukončení $T.end$ a čas počátku validace $T.valid$. Mějme dvě transakce T_1 a T_2 takové, že $T_1.valid < T_2.valid$, tedy nejprve zahájila validaci transakce T_1 . Pak může nastat jedna z následujících situací (jejichž grafické zobrazení je uvedeno na obrázku 12.3 na následující straně):

- $T_1.end < T_2.start$, tedy transakce T_1 skončila dříve než T_2 začala. Pak k žádnému konfliktu dojít nemohlo.
- $T_1.end > T_2.start$ a zároveň $T_1.end < T_2.valid$, tedy transakce T_1 skončila až po startu T_2 , ale dříve, než T_2 zahájila validaci. Pokud T_2 nečež žádná data modifikovaná v T_1 , pak ke konfliktu dojít nemohlo.
- $T_2.valid < T_1.end$, tedy transakce T_1 skončí až po zahájení validace T_2 . Pokud T_2 nečež ani nemodifikuje žádná data modifikovaná v T_1 , pak ke konfliktu dojít nemohlo.

Při validaci jsou zkонтrolovány všechny tři výše uvedené situace. Pokud k žádnému potenciálnímu konfliktu nedošlo, příslušná transakce může data zapsat a provést *COMMIT*. V opačném případě je restartována. Poznamenejme ještě, že tento přístup má stejně jako většina optimistických přístupů výhodu pouze v případě, že ke konfliktním situacím nedochází příliš často. V opačném případě může být podstatně



Obrázek 12.3: Optimistické uspořádání časových razítek

méně efektivní než přístup pesimistický, neboť dochází ke zbytečně velkému množství rušení transakcí.

12.6 MVCC

Poslední metodou, kterou ještě krátce zmíníme, je *multiversion concurrency control* (MVCC), tedy řízení souběžného přístupu k datům prostřednictvím více verzí. Tato metoda vychází z požadavku, aby operace čtení nemusela čekat na operaci zápisu než bude moci data přečíst. Místo toho je při modifikaci určitého objektu vytvořena jeho nová verze s časovým razítkem modifikace. Starší verze objektů nejsou mazány, ale jsou určeny právě pro odpovídající operace čtení.

Předpokládáme tedy, že má každá transakce T opět časové razítko $T.start$ začátku. Každý objekt O v databázi může mít několik verzí, např. $O_{ver1}, O_{ver2}, \dots$, kde $ver1, ver2, \dots$

ver2, ... odpovídají časovým razítkům transakcí, které tyto verze vytvořily. Navíc má každý objekt časové razítko posledního čtení z objektu *O.rts*.

Uvažujme transakci T_i :

- Operace čtení v T_i z objektu O načte právě tu verzi objektu O , která má nejnovější časové razítko předcházející časovému razítku $T_i.start$, tedy nejnovější verzi dat zapsanou dříve, než začala transakce T_i .
- Při operaci zápisu v T_i do objektu O dojde ke kontrole, zda nedošlo k situaci, že by nějaká transakce T_j četla z verze objektu O_k , kde $k = T_k.start$ nějaké transakce T_k , přičemž by platilo, že $T_k.start < T_i.start < T_j.start$. Tedy nedoje k situaci, že by mladší transakce T_j četla hodnotu, kterou zapsala starší transakce T_k , přičemž T_i se pokouší o vytvoření novější verze dat, kterou by měla T_j číst. Pokud ano, je transakce T_i restartována. Jinak je vytvořena nová verze objektu O_m , kde $m = T_i.start$, tj. data jsou zapsána do nové verze objektu.

U tohoto přístupu je ovšem zjevnou hlavní nevýhodou nutnost ukládat více verzí objektů. Navíc pochopitelně nechceme ukládat všechny verze objektů navždy. Existuje tedy několik metod, jak určit, které z verzí již mohou být smazány. Některé systémy tuto funkcionality zajišťují automaticky, jiné i na explicitní vyžádání.

12.7 Závěr

Metod řízení transakčního zpracování existuje obecně i v oblasti distribuovaného zpracování dat velké množství. Jak jsme navíc uvedli ve druhé části této knihy, téměř každý systém zvolenou strategii dále upravuje dle svých potřeb a možností. Obecně lze říci, že hlavní rozhodnutí, které je třeba v distribuovaných systémech udělat, spočívá v kompromisu mezi efektivitou a mírou konzistence dat. Ta se ale může u různých aplikací výrazně lišit. Tudíž není možné jednoznačně říci, která ze strategií je univerzálně optimální.

13.

Pokročilé aspekty grafových databází

V kapitole 9 jsme představili základní principy používané v grafových databázích především prostřednictvím systému Neo4j. Jak jsme již ale naznačili, grafové databáze musí pro zajištění efektivní implementace požadované funkcionality vyřešit poměrně specifické problémy. V této kapitole se podíváme na hlavní specifika grafových databází a na vybrané vhodné metody pro jejich řešení. V současné době boomu NoSQL databází se i v této oblasti ovšem objevují stále nové efektivnější přístupy. Uvedený přehled tedy jistě není ani úplný, ani konečný.

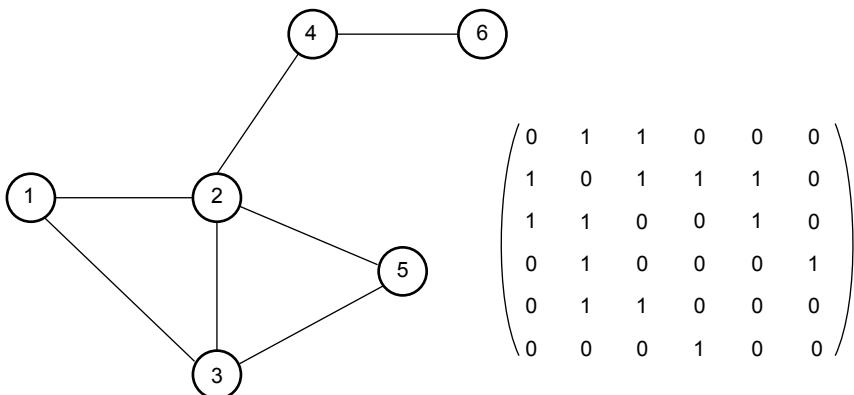
13.1 Reprezentace grafů

Abychom mohli vysvětlit, jaké problémy je třeba řešit v grafových databázích, podívejme se nejprve podrobněji na různé způsoby, jak je možné graf reprezentovat. Typicky je graf definován jako dvojice $G = (V, E)$, kde V je množina uzlů a $E = V \times V$ je množina hran, tedy dvojic $(v_1, v_2) \in E$, kde $v_1, v_2 \in V$. Dále obvykle označujeme počet uzlů jako n a počet hran m .

13.1.1 Matice sousednosti

Pro uložení grafu můžeme využít několik datových struktur. Asi nejznámější je *matice sousednosti*, což je dvourozměrné pole A , které obsahuje $n \times n$ Booleovských hodnot (tedy hodnot 0 a 1). Hodnota 1 na pozici A_{ij} říká, že mezi uzly i a j vede hrana. Hodnota 0 znamená, že nevede. Jsou-li hrany grafu neorientované, pak stačí pouze odpovídající trojúhelníková matici, neboť poloviny matice pod a nad diagonálou jsou totožné. Můžeme mít ale i *vážený graf*, tj. graf, na jehož hranách zaznamenáváme určité váhy (např. vzdálenosti uzelů). Místo hodnoty 1, pak v matici uvádíme právě tuto váhu.

Příklad grafu a jemu odpovídající matice sousednosti je uveden na obrázku 13.1. Jak je z příkladu patrné, hlavní výhodou této reprezentace je především snadné přidávání/odebírání hran, kdy pouze změníme příslušnou hodnotu na 1/0, a kontrola, zda jsou dva uzly spojeny hranou. Nevýhodou je ovšem kvadratický prostor, který potřebujeme k uložení grafu, vysoká cena za přidávání uzelů (tedy nutnost zvětšování matice) a také získání všech sousedů daného uzlu, které je lineární vzhledem k n .



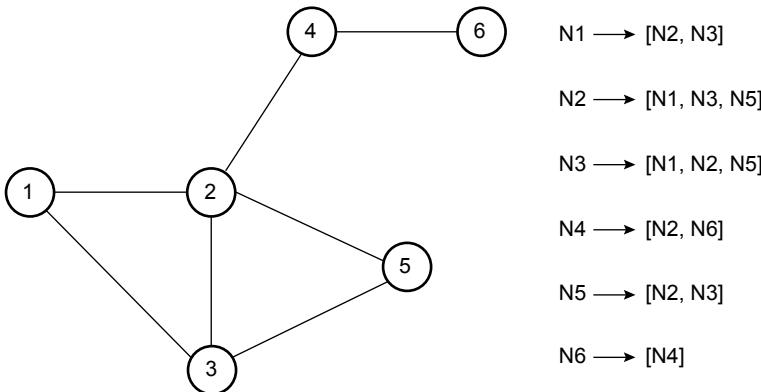
Obrázek 13.1: Reprezentace grafu pomocí matice sousednosti

13.1.2 Seznam sousedů

Dalším způsobem reprezentace grafu je *seznam sousedů*. Jedná se o datovou strukturu, která odpovídá vektoru n ukazatelů na spojový seznam sousedních uzelů každého uzlu v grafu. Máme-li tedy hranu z uzlu i do uzlu j , pak seznam sousedů uzlu i obsahuje uzel j . Velmi často jsou seznamy zkomprimovány, jelikož můžeme využít pravidelnosti grafu, opakování struktur, rozdílů podobných částí apod.

Příklad grafu a odpovídajícího seznamu sousedů je uveden na obrázku 13.2 na následující straně. Výhody i nevýhody jsou opět zjevné. Všechny sousedy uzel umíme získat efektivně (máme přímý přístup k jejich seznamu), stejně jako umíme efektivně přidávat nové uzly i hrany grafu prostým prodloužením vektoru uzelů nebo vybraného spojového seznamu sousedů. Reprezentace je také prostorově mnohem úspornější než v předchozím případě, jelikož informaci ukládáme pouze v případě, že hrana existuje. Nevýhodou je ovšem zjišťování existence hrany mezi

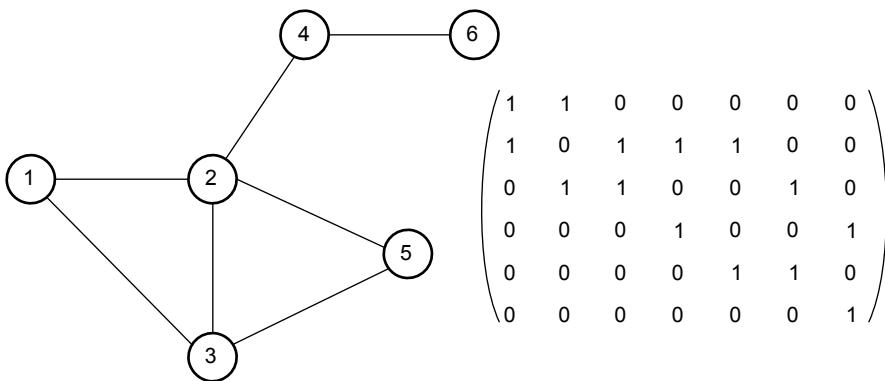
uzly, kdy musíme projít celý seznam sousedů. Tento problém je možné optimalizovat jejich seřazením, címž získáme logaritmický čas vyhledávání, ale za cenu logaritmického času přidávání.



Obrázek 13.2: Reprezentace grafu pomocí seznamu sousedů

13.1.3 Matice incidence

Dalším typem reprezentace je *matice incidence*, což je dvourozměrné pole Booleovských hodnot o n řádcích a m sloupcích. Sloupec reprezentuje hranu a hodnoty 1 označují uzly, které hranu tvoří. Řádek reprezentuje uzel a hodnoty 1 označují všechny hrany, které k tomuto uzlu náleží. Jak vidíme na obrázku 13.3, v každém sloupci jsou dvě hodnoty 1, nicméně reprezentace by umožňovala i *hypergraf*, tedy graf, jehož hrany spojují více než dva uzly. Nevýhodou této reprezentace je ale opět množství prostoru, který potřebuje $(n \times m)$, přičemž pro mnoho grafů navíc platí, že m je výrazně větší než n , tj. hran je mnohem více než uzlů.

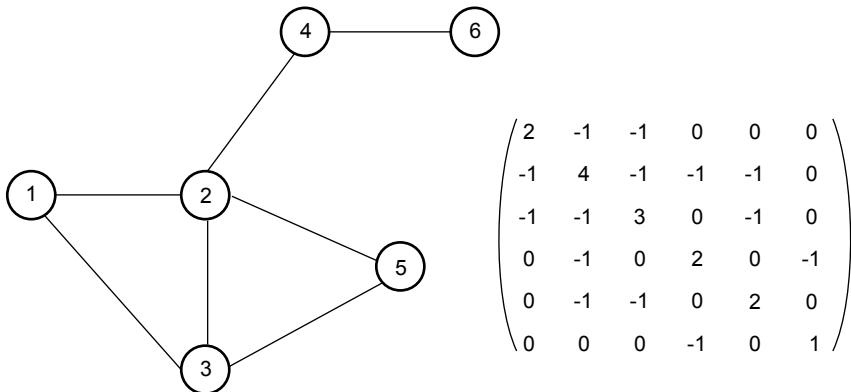


Obrázek 13.3: Reprezentace grafu pomocí matice incidence

13.1.4 Laplaceova matice

Posledním typem reprezentace grafu, který zde představíme, je *Laplaceova matice*. Opět se jedná o dvourozměrné pole o velikosti $n \times n$, tedy o čtvercovou matici.

Na diagonále matice je uveden stupeň příslušného uzlu, tj. počet hran, jejichž je prvkem. Hodnota -1 na pozici A_{ij} říká, že mezi i a j vede hrana. Hodnota 0 znamená, že nevede. Na obrázku 13.4 je uveden příklad grafu a jeho Laplaceovy matice. Tato reprezentace má stejné výhody (ale i nevýhody) jako maticy sousednosti. Navíc ale umožňuje provádění složitějších matematických operací, např. prostřednictvím výpočtu determinantu a vlastních čísel je možné analyzovat strukturu grafu.



Obrázek 13.4: Reprezentace grafu pomocí Laplaceovy matice

13.2 Lokalita dat

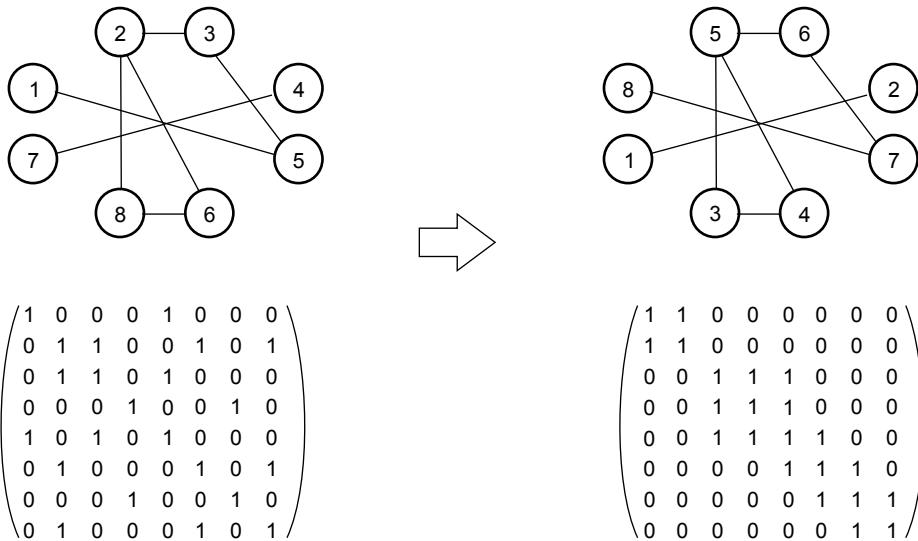
Se zvoleným způsobem reprezentace grafu úzce souvisí problematika *lokality dat*, tedy jejich optimálního uložení vzhledem k fyzické architektuře zvoleného systému. Cílem je uložit data tak, aby jejich fyzické umístění odpovídalo jejich logické vzdálenosti a při průchodu grafem byla délka této fyzické kolokaci data, která budeme potřebovat v následujícím kroku, načtena co nejrychleji.

Ukážeme jeden z mnoha přístupů určený pro optimalizaci lokality u matice sousednosti. Pro tyto účely nejprve zavedeme pojem *šířka pásma matice*, což je maximální vzdálenost mezi nenulovými prvky řádků matice sousednosti, přičemž jeden je nad a druhý pod diagonálou. Matice s nízkou šířkou pásma mají všechny nenulové prvky nahromaděny okolo diagonály, tedy „blízko sebe“. Uvažujeme-li klasickou architekturu, kdy jsou data z disku načítána po blocích, toto uložení umožní načítat sousedy daného uzlu najednou, resp. po velkých skupinách.

Podívejme se na obrázek 13.5 na následující straně. Z něho je patrné, že abychom snížili šířku pásma matice, stačí vhodně přečíslovat uzly grafu. Naneštěstí je tento problém, tzv. *problém minimalizace šířky pásma matice*, NP-těžký a pro jeho řešení se tedy využívají nejrůznější heuristiky poskytující alespoň suboptimální řešení.

Jednou z nejznámějších technik je metoda autorů Cuthilla a McKeeho, která pochází již z roku 1969 [73]. Tento algoritmus přečíslouje uzly grafu pomocí následující poměrně jednoduché, ale relativně efektivní heuristiky takto:

1. Nejnižší identifikátor dostane uzel s nejnižším stupněm, tedy nejnižším počtem sousedů.



Obrázek 13.5: Šířka pásmo matice původní a po přečíslování uzlů

2. Ostatní uzly jsou číslovány sekvenčně tak, jak jsou nalézány při průchodu grafu do šířky začínajícím v uzlu s identifikátorem 1. Navíc jsou i sousedé zpracováváni v pořadí od uzlu s nejnižším stupněm.

13.3 Distribuce grafu

Stejně jako u jakýchkoli jiných dat, i u těch grafových existují aplikace, u kterých velikost dat o několik řádů přesahuje možnosti jediného byt vysoce výkoného serveru. A pak i zde vyvstává otázka, jak grafová data distribuovat. V databázích, kde měla data formát (klíč, hodnota) s různě komplexní hodnotovou částí, předpokládáme, že tato dvojice nemá extrémní velikost a tudíž vlastně řešíme, jak vhodně rozdělit množinu dvojic na podmnožiny a tyto uložit na jednotlivé uzly v clusteru. Uvažujeme-li ale případ jednoho velkého grafu, např. popisujícího sociální síť nebo strukturu internetu, musíme řešit, jak tento graf vhodně rozdělit na podgrafy a jak uložit hrany mezi nimi. Tento problém je obecně velmi těžký, tudíž, jak jsme viděli v kapitole 9, současné grafové databáze distribuci grafu často neumožňují.

Na druhou stranu existují metody, které se efektivní distribucí grafu zabývají. Tyický dokáží optimalizovat distribuci dat vzhledem k určité operaci nebo malé množině operací, jako je např. hledání nejkratší cesty, hledání kostry grafu, průchod do šířky apod. Ukážeme nyní jednu z nich, konkrétně metodu, která se zabývá problémem, jak efektivně uložit velký graf matice sousednosti na více serverů pro účely procházení grafem do šířky [71]. Autoři nejprve navrhují naivní přístup, tzv. *1D dělení matice*, které je naznačeno na obrázku 13.6 na straně 223. Jak je vidět, mezi N serverů v clusteru (v našem příkladu $N = 4$) jsou rovnoměrně rozděleny řádky matice, což odpovídá rozdělení n uzlů (v našem příkladu $n = 12$) mezi N serverů. Každý server pak zná všechny hrany vedoucí do/z těch uzel grafu, které jsou na něm uloženy. V našem příkladu má tedy každý server informace o třech

uzlech grafu a souvisejících hranách, jak je naznačeno prostřednictvím různých typů ohraničení části matice, kde každý typ čáry odpovídá jednomu serveru.

Předpokládejme, že chceme realizovat průchod grafem do šířky. Tj. na vstupu máme počáteční uzel s na úrovni 0 a na výstupu bychom chtěli označit všechny uzly v grafu jejich úrovní označující vzdálenost od uzlu s při průchodu do šířky. Algoritmus by pracoval takto:

1. Každý server udržuje množinu tzv. *hraničních uzlů*, která na počátku obsahuje pouze uzel s pro server, na němž je uložen, zatímco pro ostatní servery je prázdná.
2. V každém kroku každý server zpracuje svoji množinu hraničních uzlů tak, že naleze jejich sousední uzly na příslušném řádku matice. (Některé jsou fyzicky uloženy na aktuálním serveru, zbylé na ostatních serverech.)
3. Sousední uzly uložené na aktuálním serveru se stávají novou množinou hraničních uzlů s novou úrovní. Pro zbylé jsou všem ostatním serverům (servery totiž nevědí, jak jsou data distribuována) zaslány zprávy, aby tyto uzly, jsou-li na nich uloženy, přidaly do svých množin hraničních uzlů, také s novou úrovní.
4. Pro uzly, které již byly zpracovány v předechozích iteracích, jsou zprávy ignorovány, tj. nejsou znova zpracovány.

Jak je tedy patrné, každý server musí komunikovat se všemi ostatními $N - 1$ servery, i když na nich požadovaná data nemusí být uložena, což představuje zbytečnou režii.

Na obrázku 13.7 na straně 224 je znázorněna optimalizace naivního přístupu nazývaná *2D dělení matice sousednosti*. Stále máme N serverů (v našem případě stále $N = 4$), mezi něž chceme rozdělit informace ve stejné matici. Tentokrát ale pro tyto účely vytvoříme mřížku o velikosti $R \times S$, kde S je počet *blokových sloupců* a R je počet *blokových řádků* (v našem případě $S = 2$ a $R = 2$). Mřížkou, v níž každá buňka reprezentuje jeden server, pak pokryjeme matici. Jak je naznačeno na obrázku, v uvedeném příkladu je mřížka zopakována dvakrát. Každý server pak tedy vlastní stejně množství dat z matice, ale v jiných částech než v 1D dělení. Na obrázku je tato situace opět naznačena různými typy ohraničení části matice, kdy každý typ čáry odpovídá jednomu serveru.

Algoritmus procházení do šířky bude fungovat obdobně (ostatně, pokud bychom nastavili $S = 1$ nebo $R = 1$, dostáváme právě 1D dělení). Rozdíl bude v tom, že v kroku 2 nemáme všechny sousedy zpracovávaného uzlu uloženy na aktuálním serveru a je tedy nutné poslat zprávy všem ostatním serverům vlastnícím informace v příslušném řádku matice (těch je $R - 1$). Naopak v kroku 3 nemusíme kontaktovat všechny servery, ale pouze ty, které vlastní informace v příslušném sloupci matice (těch je $S - 1$). (Poznamenejme, že servery nemusí znát distribuci celého grafu, ale pouze velikost mřížky odpovídající přidělení jejich částí k jednotlivým serverům.) 2D dělení matice tedy v každém kroku nevyžaduje komunikaci se všemi ostatními $N - 1$ servery, ale pouze s $(R - 1) + (S - 1)$. Tedy v našem příkladu místo se třemi pouze se dvěma. Při větším clusteru by tento rozdíl byl pochopitelně výrazně zajímavější.

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	1	1	0	
2	0	0	1	0	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	1	0
7	0	0	1	0	0	1	0	1	0	1	1	0
8	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1
10	1	0	0	0	0	0	1	0	0	0	1	0
11	1	0	0	0	0	1	1	0	1	1	0	0
12	0	0	0	1	1	0	0	0	1	0	0	0

Obrázek 13.6: 1D dělení matice sousednosti

13.4 Dotazování nad grafy

Již v úvodu jsme naznačili, že aplikace pracující s grafy a grafovými databázemi mají různé charakteristiky a můžeme je klasifikovat z hlediska různých kritérií. Obecně grafová databáze představuje množinu grafů. Pak můžeme rozlišovat tzv. *transakční* a *netransakční* grafové databáze. V obecné transakční databázi předpokládáme vhodnou normalizaci dat, dotazovací jazyky a především zajištění transakčního zpracování dat. Z hlediska grafových dat pak typicky ukládáme velké množství malých grafů, jako jsou např. chemické sloučeniny¹ nebo větné rozbory reprezentované v podobě vzájemně provázaných stromů.² U netransakčních databází naopak předpokládáme spíše analytické dotazy nenormalizovaných dat a transakční zpracování zajištěné příslušnou aplikací namísto samotnou databází. Typicky do nich ukládáme malé množství velkých grafů (nebo jeden nesouvislý graf, tj. graf, který má několik nepropojených komponent). Příkladem je třeba sociální síť.

¹ <http://pubchem.ncbi.nlm.nih.gov/>

² <http://www.cis.upenn.edu/~treebank/>

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	1	1	0
2	0	0	1	0	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	1	0
7	0	0	1	0	0	0	0	1	0	1	1	0
8	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1
10	1	0	0	0	0	0	1	0	0	0	1	0
11	1	0	0	0	0	0	1	0	1	1	0	0
12	0	0	0	1	1	0	0	0	1	0	0	0

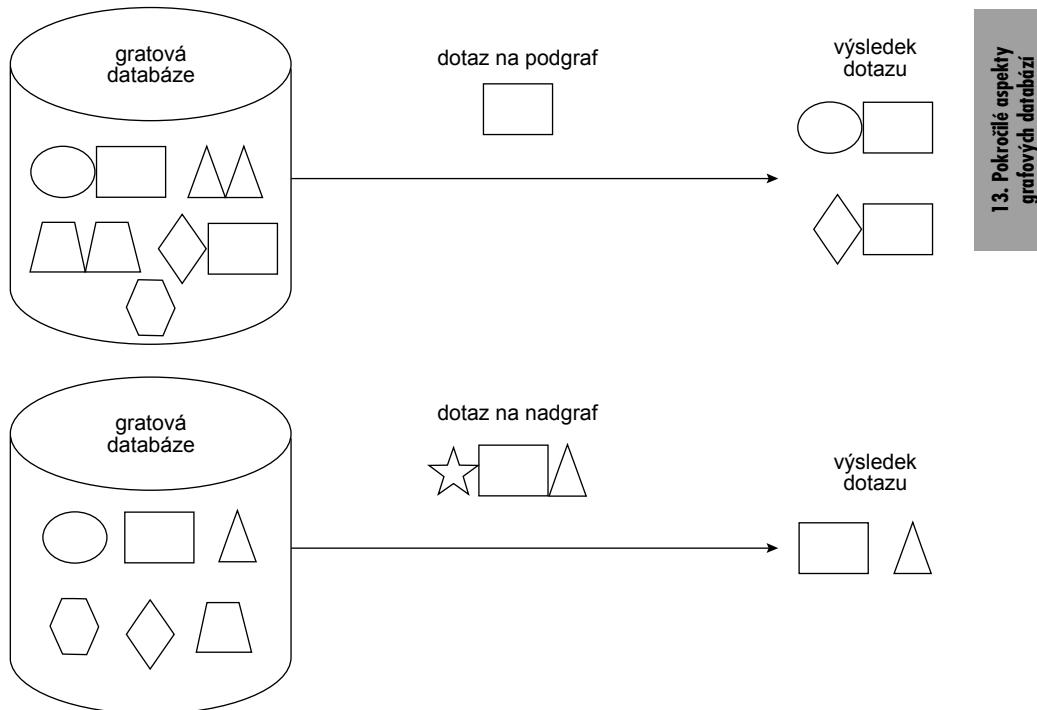
Obrázek 13.7: 2D dělení matic sousednosti

13.4.1 Typy dotazů

V netransakčních databázích typicky využíváme klasické grafové algoritmy, jako je hledání cesty mezi dvěma uzly, kde nás může zajímat existence jakékoli cesty, nalezení nejkratší cesty, všech cest nebo cest určitých vlastností (např. vedoucí přes určité typy hran nebo uzlů) apod. Např. v sociální síti podobné grafu na obrázku 9.1 na straně 144 v kapitole 9 bychom mohli zjišťovat, zda „David napsal některou ze svých knih s některým ze svých kolegů“. Nejedná se tedy o klasické dotazování nad daty.

U transakčních databází dotazy typicky odpovídají *dotazu na podgraf*, *dotazu na nadgraf* nebo *dotazu na podobný graf*. Rozdíl mezi dotazem na podgraf a nadgraf je naznačený na obrázku 13.8 na následující straně. V obou případech je vstupem, tedy dotazem, graf. V případě dotazu na podgraf hledáme v databázi všechny *datové grafy*, které zadaný *dotazový graf* obsahují. V případě dotazu na nadgraf naopak hledáme v databázi všechny datové grafy, které jsou v dotazovém grafu obsaženy. Například v databázi chemických sloučenin bychom takto mohli hledat všechny sloučeniny, které obsahují tu námi zadanou, nebo takové, které jsou v ní obsaženy, nebo ty, které jsou jí nějakým způsobem podobné.

Graf reprezentující dotaz může mít také různé charakteristiky. Jak ukážeme v sekci 13.4.3, velmi často dotazový graf obsahuje uzly nebo hrany, u nichž nespe-



Obrázek 13.8: Dotaz na podgraf vs. dotaz na nadgraf

cifikujeme bližší vlastnosti. V grafu dotazu také můžeme určit množinu přípustných typů uzlů/hran, které se v hledaných grafech na daných místech mohou vyskytovat. Existují také dotazovací jazyky, které naopak umožňují např. specifikovat, že se mezi danými uzel nesmí vyskytovat hrana nebo cesta. U podobnostního vyhledávání zase řešíme otázku, jak podobnost grafiů měřit. Obvykle nechceme hledat grafy, které jsou izomorfní zadanému grafu,³ ale povolujeme určitý stupeň odlišnosti, který by měla podobnostní funkce vystihovat (což může např. odpovídat obdobnému chování chemických sloučenin).

Je také velmi časté, že konkrétní databázový systém kombinuje oba typy přístupů k datům, tj. transakční i netransakční. Např. v grafu reprezentujícím sociální síť můžeme hledat části, které jsou podobné zadanému dotazu: hledáme podobné vztahové podstruktury.

13.4.2 Vyhodnocování dotazů a indexace grafových dat

Obecný princip vyhodnocování dotazů nad grafy funguje následujícím způsobem: Předpokládejme, že databáze má obsahovat množinu datových grafiů $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$. Každý takový graf je při svém uložení zaindexován s použitím vybrané indexační techniky, která extrahuje a indexuje vybrané charakteristiky grafiu s cílem uložit maximum zajímavých informací co nejúsporněji a pokud možno ještě tak, aby byl index při změnách v databázi snadno aktualizovatelný.

³ O dvou grafech řekneme, že jsou *izomorfní*, pokud mají stejnou strukturu (až na označení uzelů, resp. hran).

Vyhodnocení dotazu reprezentovaného dotazovým grafem G_{dotaz} pak probíhá ve třech krocích:

1. *Extrakce*: Graf G_{dotaz} je zpracován stejným způsobem jako datové grafy, tj. jsou z něj vyextrahovány indexované charakteristiky.
2. *Filtrace*: Porovnáním charakteristik jsou z datových grafů $\mathbf{G} = \{G_1, G_2, \dots, G_n\}$ vybrány ty, které odpovídají charakteristikám G_{dotaz} . Vznikne tak *množina kandidátů* na výsledek $\mathbf{G}_{kand} \subset \mathbf{G}$, kde pokud možno platí, že $|\mathbf{G}_{kand}| < |\mathbf{G}|$, a to významně, tj. chceme, aby indexační technika vracela co nejméně *falešně pozitivních kandidátů* a měla tedy vysokou *filtrační schopnost*.
3. *Verifikace*: Množina \mathbf{G}_{kand} je porovnána s dotazovým grafem G_{dotaz} ještě jednou, tentokrát ale ne pouze vzhledem k vybraným charakteristikám, ale přesněji, např. řešíme isomorfismus grafu G_{dotaz} a grafu v \mathbf{G}_{kand} . Tím vyloučíme případné zbylé falešně pozitivní kandidáty na výsledek a dostaváme množinu skutečných výsledků $\mathbf{G}_{vysl} \subseteq \mathbf{G}_{kand}$.

V některých případech je možné krok 3 vynechat, obecně ale u indexačních metod řešíme především otázku dobré filtrační schopnosti, aby verifikační fáze nebyla příliš dlouhá.

Pro indexaci dotazů na podgraf rozlišujeme dva typy indexačních přístupů – *mining-based* a *non-mining-based metody*. Pro dotazy na nadgraf a podobnostní dotazy existuje poměrně málo přístupů, ale často také využívají princip indexu. Pro každou z těchto skupin představíme několik zástupců v následujících sekcích. Poznamenejme, že se u většiny metod obvykle jedná o návrh z odborného článku, k němuž existuje experimentální implementace, ale který se často stane součástí některého z komerčních nástrojů až později.

13.4.2.1 Mining-based indexační metody pro dotazy na podgraf

Metody založené na analýze (resp. dolování) dat (*mining-based*) jsou založené na jednoduchém pozorování, že pokud určitá množina vlastností dotazového grafu G_{dotaz} není obsažena v některém z datových grafů G_1, G_2, \dots, G_n , nebude takový datový graf obsahovat ani celý dotazový graf. Jinými slovy, nepotřebujeme zkoumat celou strukturu dotazového grafu G_{dotaz} , stačí nám vybrat vhodné podstruktury a zaměřit se při indexaci a vyhodnocování dotazů pouze na ně. Zajímavými podstrukturami mohou být např. časté podstromy nebo podgrafy. Pro ně je vytvořen *invertovaný index*, tedy seznam indexovaných podstruktur a pro každou z nich pak seznam datových grafů, ve kterých je obsažena. Při vyhodnocování dotazu se nejprve v grafu G_{dotaz} identifikují příslušné podstruktury a jejich seznam je porovnán se seznamem podstruktur každého datového grafu G_1, G_2, \dots, G_n , čímž dostaneme množinu kandidátů na výsledek \mathbf{G}_{kand} .

Samozřejmě hlavní nevýhodou tohoto přístupu je ale právě závislost efektivity vyhodnocování dotazů na dobrém výběru indexovaných podstruktur, stejně jako možná degradace efektivity v závislosti na změnách dat (po nichž už nemusí být daná podstruktura zajímavá, např. určitý podstrom častý), a tím pádem nutná re-identifikace častých podstruktur a *re-indexace*. V následujícím textu krátce

představíme dva zástupce těchto metod – metodu GIndex, která se zaměřuje právě na časté podgrafy, a metodu TreePI, která indexuje časté podstromy.

GIndex

Metoda GIndex [157] z roku 2004 indexuje tzv. *frekventované diskriminativní podgrafy*. Požadovanou míru frekventovanosti, tedy jak často se daný podgraf vyskytuje, určuje práh P , který je modifikován vzhledem k tomu, jak graf narůstá. *Diskriminativní graf* je pak takový graf, který není možné rozpoznat na základě jeho diskriminativních podgrafů. Vysvětleteme tuto myšlenku na příkladu tří datových grafů G_1 , G_2 , G_3 a jednoho dotazového grafu G_{dotaz} na obrázku 13.9 na následující straně. Ta vychází z metody indexace textu založené na pozorování, že máme-li pouze malý slovník, bude lepší namísto jednotlivých slov indexovat celé fráze. Uvážíme-li tedy uvedené tři datové grafy, určitě nebude dobrou strategií indexovat pro ně např. uhlíkové řetězce až do délky tří (tj. C, C-C, C-C-C, C-C-C-C). Ty se totiž vyskytují ve všech třech grafech a tudíž by filtrační schopnost indexu byla nízká – do množiny \mathbf{G}_{kand} by se dostaly i první dva grafy, i když by se jednalo o falešně pozitivní kandidáty z hlediska uvedeného G_{dotaz} . Indexované podgrafy je tedy potřeba vybírat obezřetně.

Při vyhodnocování dotazu pak metoda GIndex pracuje takto:

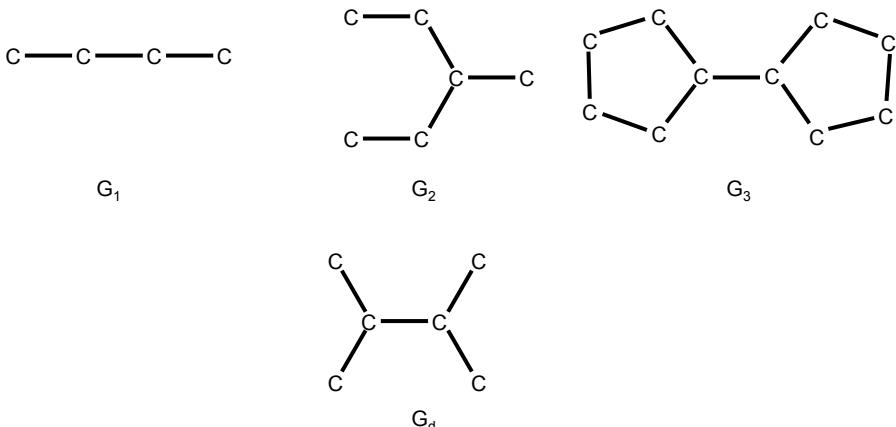
1. Pokud je G_{dotaz} přímo jedním z frekventovaných diskriminativních podgrafů, můžeme datové grafy, které ho obsahují, získat přímo, jelikož jsou uvedeny v indexu.
2. Jinak G_{dotaz} obsahuje frekventovaný diskriminativní podgraf, jehož zastoupení v datových grafech ovšem nebude vysoké, jelikož celý dotazový graf frekventovaný není a předpokládáme, že práh P také není příliš vysoký. Tudíž tento podgraf nalezneme pouze v malém množství grafů, které bude potřeba následně verifikovat.

TreePI

Druhá metoda, kterou představíme, je z roku 2007 a jmenuje se TreePI [159]. Na rozdíl od metody GIndex se zaměřuje na indexaci častých podstromů. Oba články se shodují na tom, že indexovat pouze cesty nevede k dobrému výsledku, nicméně autoři metody TreePI navíc dodávají:

1. Stromy jsou schopny uchovat téměř ekvivalentní množství strukturální informace jako obecné grafy.
2. Vyhledávání a indexace častých podstromů je jednodušší než u častých podgrafů, kde jsou odpovídající algoritmy obvykle NP-těžké.

V obecném postupu se pak obě metody výrazně neliší, ať už se jedná o princip hledání diskriminativních častých podstruktur nebo vlastní dotazování. Hlavní rozdíl je tedy především v tom, že TreePI pracuje se stromy.



Obrázek 13.9: GIndex – motivační příklad

13.4.2.2 Non-mining-based indexační metody pro dotazy na podgraf

Metody, které nejsou založeny na analýze (dolování) dat (*non-mining-based*), naopak indexují každý graf jako celek, nezkoumají jeho strukturu nebo jeho časté vlastnosti (podstruktury). Výhodou jsou efektivní modifikace grafu, které nevyžadují přebudování celého indexu, a nezávislost na dobré volbě indexovaných struktur. Nevýhodou ovšem může být nízká filtrační schopnost indexu a tedy rozsáhlá verifikační fáze.

GraphGrep

Metoda GraphGrep [90] indexuje všechny cesty v grafu až do určité délky. Index si pak lze představit jako tabulku, jejíž řádky odpovídají nalezeným cestám a sloupce datovým grafům v databázi. V buňkách tabulky jsou pak počty výskytů jednotlivých cest v datových grafech. Indexace je tedy velmi jednoduchá a rychlá, ale na druhou stranu je také patrné, že index může velmi rychle narůst do nepříjemně velkých rozměrů.

Při vyhodnocování dotazu jsou z dotazového grafu G_{dotaz} extrahovány všechny cesty, prostřednictvím indexu jsou odfiltrovány jen ty grafy, které obsahují stejné nebo větší množství těchto cest, a ve verifikační fázi probíhá kontrola izomorfismu mezi dotazovým grafem G_{dotaz} a grafy z množiny G_{kand} . Jak je z tohoto postupu vidět, filtrační schopnost indexu (který, jak už víme, může zabírat velké množství místa) nemusí být velká a tudíž verifikační fáze může být velmi zdlouhavá.

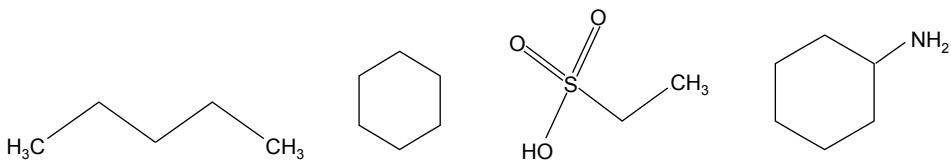
GString

Metoda GString [110] se zaměřuje na indexaci grafových struktur z organické chemie a díky této specializaci může využívat sémantiku indexovaných dat. Pro tyto účely definuje speciální podstruktury grafu zobrazené na obrázku 13.10 na následující straně:

- cesta – posloupnost uzlů spojených hranami,
- cyklus – posloupnost uzlů spojených hranami, které tvoří uzavřenou smyčku,

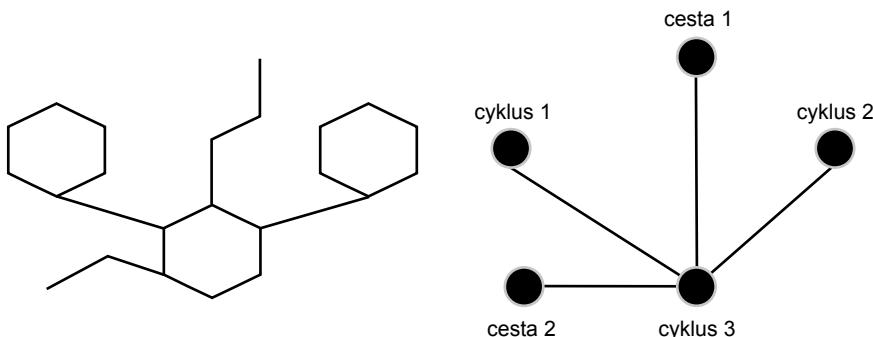
- hvězda – centrální uzel, z něhož vycházejí hrany do několika sousedních uzlů,
- větev – hrana, která je připojena k jiné struktuře pouze jedním uzlem.

Kromě nich máme k dispozici ještě klasické podstruktury uzel a hrana.



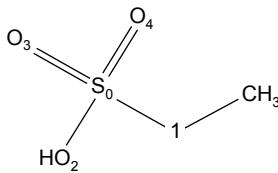
Obrázek 13.10: GString – cesta, cyklus, hvězda a větev

V indexační fázi jsou nejprve podstruktury vyhledávány v pořadí cykly – hvězdy – cesty – větve a následně kontrahovány (sloučeny) do speciálních uzlů, jak je naznačeno na obrázku 13.11.



Obrázek 13.11: GString – identifikace podstruktur

Kontrahovaný graf je pak reprezentován jako řetězec popisující vzájemné vztahy těchto struktur. Např. graf uvedený na obrázku 13.12 bychom mohli reprezentovat řetězcem: hvězda 4 {uzel 0 S} {uzel 2 0} {uzel 3 0} {uzel 4 0} {hrana 0 3 d} {hrana 0 4 d} {větev 1 C}. Zápis říká přibližně to, že graf je typu hvězda se čtyřmi částmi, z jejíhož středu s identifikátorem 0 vycházejí hrany do tří vrcholů s identifikátory 2, 3 a 4 a jedné větve s počátkem ve vrcholu s identifikátorem 1. Dále pak obsahuje dvě hrany – mezi vrcholy s identifikátory 0 a 3, resp. 0 a 4. Další písmena nesou informace o typech uzlů a jejich vlastnostech, jako např. chemických prvcích uložených v uzlu.



Obrázek 13.12: GString – příklad řetězce

Pro index je pak zvolena některá z datových struktur pro indexaci řetězců. Podobně je i dotazový graf G_{dotaz} reprezentován tímto způsobem (důležité pochopitelně je, aby bylo dodrženo výše uvedené pořadí vyhledávání struktur a jejich kontrakce),

tím pádem se celá fáze filtrace transformuje na problém porovnávání řetězců, které je podstatně jednodušší.

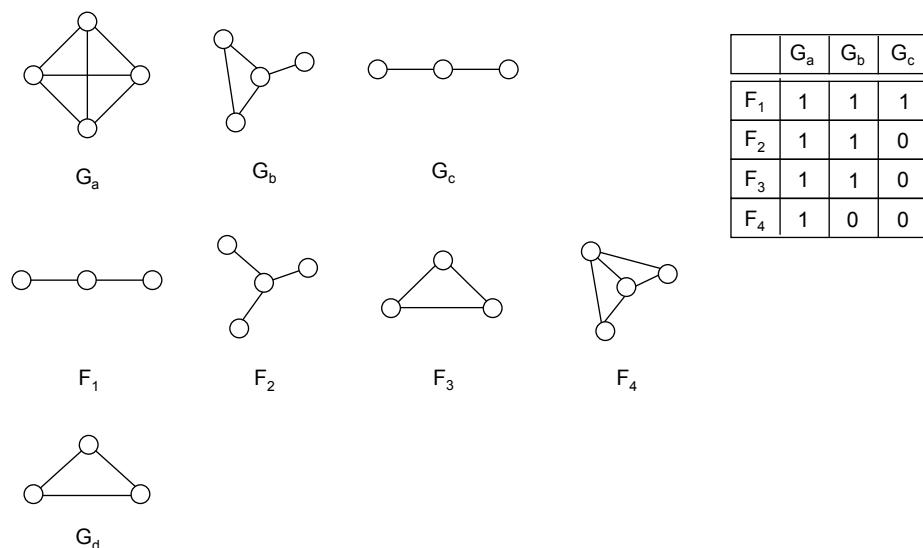
13.4.2.3 Indexy pro dotazy na nadgraf

Indexačních metod pro dotazy na nadgraf je méně, ale i v této oblasti naleznete zajímavé přístupy. Opět představíme dva zajímavé zástupce.

cIndex

Metoda cIndex [105] z roku 2007 je založena na myšlence, že pokud určitá vlastnost (podgraf) G_v není obsažena v dotazovém grafu G_{dotaz} , pak všechny datové grafy, které vlastnost G_v mají, můžeme z dalšího zpracování vynechat. Ideální stav by tedy byl, kdyby byl podgraf G_v obsažen ve velkém množství grafů, tj. kdybychom ve filtrační fázi odfiltrovali co nejvíce grafů, které nebudou součástí G_{vysl} . Hledáme tedy tzv. *kontrastní podgrafy*, tedy takové podgrafy, které jsou obsaženy ve velkém množství datových grafů, ale je velmi nepravděpodobné, že by byly obsaženy v dotazovém grafu G_{dotaz} . Tuto informaci pochopitelně nezjistíme pouze z dat, ale potřebujeme mít k dispozici např. logy starších dotazů, které byly nad databází pokládány. Tento přístup má tedy tu nevýhodu, že pokud se dotazy změní, je třeba re-indexovat, což může být poměrně často. Nicméně máme-li index aktuální, je typicky velmi malý a vyhodnocování dotazů efektivní.

Metoda cIndex vytváří pro potřeby indexace jednoduchou tabulkou, jejíž řádky tvoří indexované podgrafy a sloupce datové grafy. Příklad je zobrazen na obrázku 13.13. Konkrétně máme v databázi tři datové grafy G_a , G_b a G_c a čtyři indexované podgrafy F_1 , F_2 , F_3 , F_4 . V tabulce vpravo pak vidíme informace o tom, který indexovaný podgraf nalezneme ve kterém datovém grafu.



Obrázek 13.13: cIndex – příklad

Máme-li takto připravenou databázi, tak bychom uvedený dotazový graf G_{dotaz} mohli efektivně vyhodnotit na základě pozorování, že grafy G_a a G_b lze odfiltrovat

vzhledem k podgrafu F_2 , který není obsažen v G_{dotaz} . Na druhou stranu je např. vidět, že podgraf F_1 je obsažen ve všech grafech v databázi i v našem dotazu, ten nám tedy při filtraci nepomůže.

GPTree

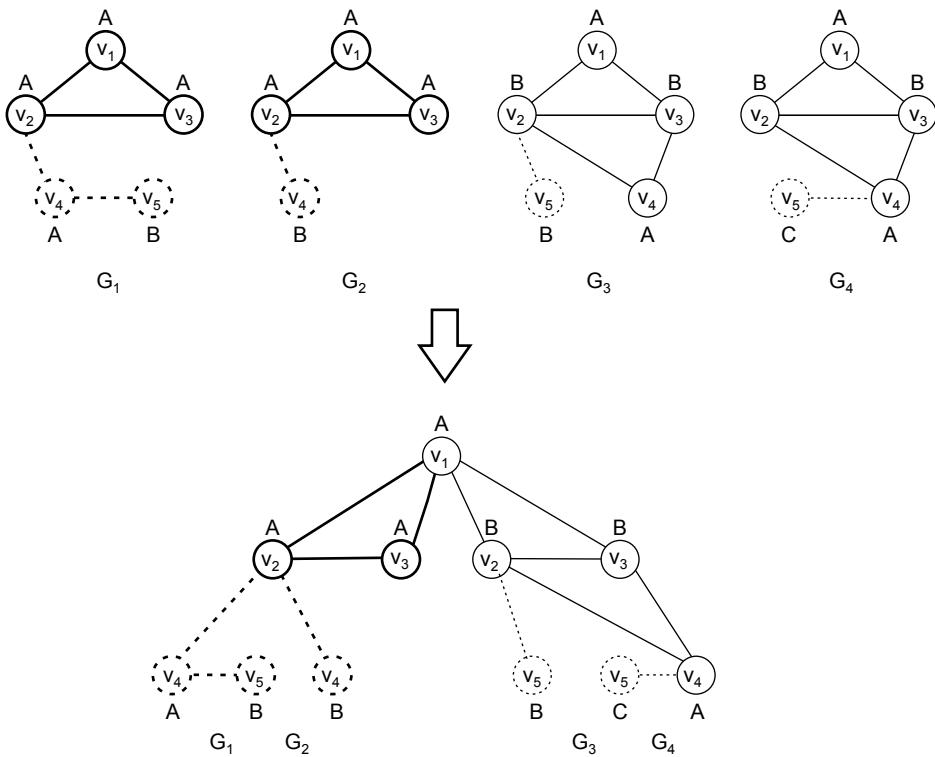
Novější metoda GPTree [160] funguje na jiném principu a snaží se vyhnout potřebě mít k dispozici logy dotazů a při změně způsobu dotazování re-indexovat. V první řadě autoři navrhují kompaktní reprezentaci databáze, která navíc umožňuje snadno identifikovat společné podgrafen díky tomu, že se všechny grafy uloží do jednoho společného grafu, v němž se společné části „překrývají“. Tato hierarchická reprezentace se nazývá GPTree a její příklad je zobrazen na obrázku 13.14 na následující straně.

Jak je vidět, chceme uložit čtyři grafy G_1 , G_2 , G_3 , G_4 , přičemž jejich struktura je velmi podobná (mají společné podgrafen). Abychom ušetřili místo a současně tuto informaci o společných podgrafech uložili a později využili, ukládá GPTree společné části pouze jednou. Jak je naznačeno na obrázku, tak pro grafy G_1 a G_2 stačí uložit společný trojúhelníkový podgraf pouze jednou, stejně tak můžeme ušetřit místo při ukládání grafů G_3 a G_4 . Pro všechny čtyři grafy máme ale společný pouze jediný uzel v_1 s názvem A. Poznamenejme ještě, že problém konstrukce optimální struktury GPTree je NP-úplný a autoři tedy navrhují aproximační algoritmus se sub-optimálním řešením.

Máme-li zkonstruovaný GPTree, máme nejen efektivně uložené všechny datové grafy, ale navíc máme přesnou informaci o častých podgrafech, jelikož víme, které části se překrývají a kolika grafů se tento překryv týká. Myšlenka pro optimalizaci dotazování je potom podobná jako v předchozí metodě – pokud se nějaký podgraf nevyskytuje v dotazovém grafu G_{dotaz} , nebude se vyskytovat ani ve výsledných grafech z G_{vysl} . Jelikož autoři nepředpokládají log dřívějších dotazů, snaží se určit podgrafen, které se pravděpodobně nebudou vyskytovat v dotazech, jiným způsobem. Z častých podgrafen se snaží vybírat ty, které mají co největší velikost, nejsou podgrafen jiného již vybraného grafu a jejich zastoupení převyšuje daný práh.

13.4.2.4 Indexy pro podobnostní dotazy

Podobnostní dotazy mají za cíl nalézt grafy, které jsou dotazovému grafu G_{dotaz} podobné. Klíčovou otázkou samozřejmě je, jak tuto podobnost měřit. Typickým využitím mohou být situace, kdy jsou v databázi chyby, nebo není úplná, a přibližně výsledky jsou nejlepším možným řešením, které můžeme dostat. V tomto případě povolíme např. chybějící či přebyvající uzly nebo hrany, rozdíly ve vlastnostech, resp. typech uzlů nebo hran, popř. i výraznější strukturální odchylky. Typicky pak máme zadanou prahovou hodnotu, která říká, nakolik se mohou grafy lišit. Pro podobnostní dotazy ale můžeme nalézt velmi specifické využití např. v biologických a chemických strukturách, kde podobností vyjadřujeme podobné chování, podobné interakce apod.



Obrázek 13.14: GPTree – příklad

Představíme dvě metody, které se zabývají právě první variantou, tj. odchylkami ve struktuře grafu, i když by je bylo možné relativně snadno rozšířit pro specifické typy podobnostního vyhodnocování.

Grafil

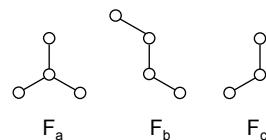
Metoda Grafil [158] z roku 2005 pro potřeby efektivního vyhledávání podobných grafů buduje dvě zajímavé datové struktury – jednu pro datové grafy a druhou pro dotazové grafy. První strukturu je *matice grafů a jejich vlastností* (*feature-graph matrix*), která obsahuje informace o počtu vnoření příslušné vlastnosti do jednotlivých datových grafů. Vlastnosti se v tomto případě rozumí vhodný části podgraf. Ukázku matice vidíme na obrázku 13.15 na následující straně pro datové grafy G_1 , G_2 , G_3 , G_4 (jejichž konkrétní struktura není pro další vysvětlení podstatná a tudíž není uvedena) a vlastnosti (podgrafy) F_a , F_b a F_c .

Matice grafů a jejich vlastností slouží k efektivnímu vyhodnocování dotazů, jelikož umožňuje spočítat rozdíl mezi počtem vlastností dotazového grafu G_{dotaz} a jednotlivých datových grafů. Případy, u nichž je rozdíl větší než stanovený práh, jsou z dalšího zpracování vyřazený, tj. nedostanou se do množiny G_{kand} . Ve verifikaci fázi se pak pro zbylé kandidáty spočítá požadovaná přesnější míra podobnosti.

matici grafů a vlastností:

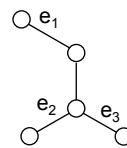
	G_1	G_2	G_3	G_4
F_a	0	1	0	0
F_b	0	0	1	0
F_c	2	3	4	4

vlastnosti:



matici hran a vlastností:

	F_a	F_b^1	F_b^2	F_c^1	F_c^2	F_c^3	F_c^4
e_1	0	1	1	1	0	0	0
e_2	1	1	0	0	1	0	1
e_3	1	0	1	0	0	1	1

graf dotazu G_d :

Obrázek 13.15: Graf

Druhou datovou strukturou, kterou metoda Grafil využívá, je *matica hran a vlastností* (*edge-feature matrix*), jež nese informaci o vztahu mezi hranami dotažového grafu a jednotlivými vlastnostmi. Podívejme se opět na obrázek 13.15, kde jsou v dotazovém grafu G_{dotaz} označeny tři hrany e_1 , e_2 a e_3 . Jim odpovídající matici hran a vlastností nás pak informuje o tom, kolik různých vnoření jednotlivých vlastností můžeme v grafu G_{dotaz} nalézt a které hrany G_{dotaz} tato vnoření pokrývají. Pro každé takové vnoření vlastnosti je v matici právě jeden sloupec.

Nyní technika Grafil využije matici datových grafů a jejich vlastností pro nalezení grafů s příslušnými počty vlastností. Pokud bychom graf, který obsahuje celkem sedm různých vnoření uvedených vlastností (jedno pro F_a , dvě pro F_b a čtyři pro F_c), v databázi nenašli, můžeme zkousit z dotazového grafu vynechat jednu z hran e_1 , e_2 nebo e_3 . Z matice hran a vlastností pak vidíme, že po vynechání jedné z těchto hran nám stále zůstanou alespoň tři vnoření indexovaných vlastností. Tudíž se do množiny **G_{kand}** nedostanou ty datové grafy, které právě alespoň tři vnoření nemají.

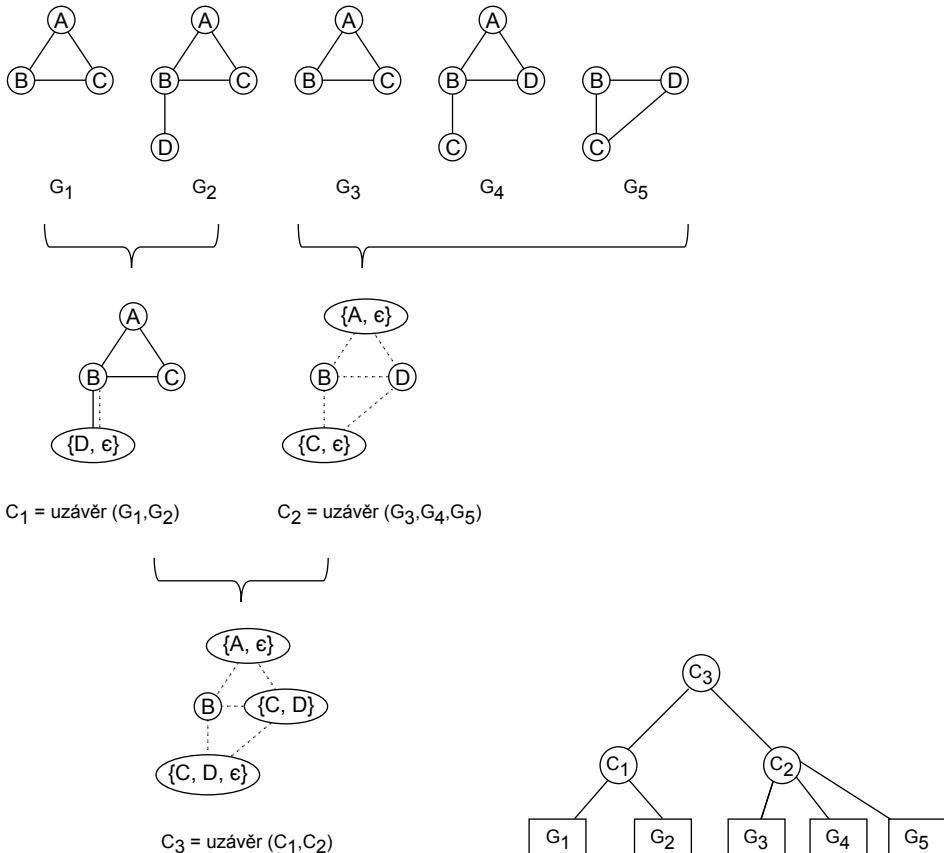
cTree

Metoda cTree [98] z roku 2006 slouží nejen k podobnostnímu dotazování, ale také k dotazům na podgrafy – záleží na tom, jak přesnou metodu zvolíme ve verifikaci fázi. Hlavní myšlenkou metody je reprezentace grafu pomocí datové struktury cTree (*closure tree*), tedy uzávěrový strom, který připomíná klasický R-strom uzpůsobený pro grafy. Každý uzel stromu obsahuje informaci o potomcích, jež umožňuje filtrovační fázi, tj. hledání kandidátů na výsledek.

Pro účely konstrukce struktury cTree je třeba zavést několik pojmu: *Uzávěr množiny uzlů V* je zobecněný uzel, jehož atributem je sjednocení atributů uzlů množiny V . *Uzávěr množiny hran E* je zobecněná hrana, jejímž atributem je sjednocení atributů hran množiny E . Uzávěr dvou grafů $G_1 = (V_1, E_1)$ a $G_2 = (V_2, E_2)$ je zobecněný graf $G = (V_G, E_G)$, kde V_G je množina uzávěrů uzlů z množin V_1 a V_2 a E_G je množina uzávěrů hran z množin E_1 a E_2 , přičemž máme dáno vhodné přiřazení vrcholů z V_1 a V_2 a hran z E_1 a E_2 .

Příklad struktury cTree pro pět grafů G_1, G_2, \dots, G_5 je zobrazen na obrázku 13.16 na následující straně, v němž mají uzly pouze jediný atribut (svůj název), zatímco

hrany nemají žádný. Nejprve je zkonstruován uzávěr grafů G_1 a G_2 označený jako C_1 a uzávěr grafů G_3 , G_4 a G_5 označený jako C_2 . Uzávěr grafů C_1 a C_2 , tedy uzávěrový graf C_3 , pak pokrývá informace všech pěti původních grafů. Jak je vidět z obrázku, vrcholy a hrany pro potřeby uzávěru grafů přiřazujeme tak, aby se co nejvíce „překrývaly“. (Speciální znak ϵ naznačuje, že v daném místě k uzávěru nedošlo.)



Obrázek 13.16: cTree

Abychom zajistili efektivní prohledávání, je cTree, stejně jako jemu podobné datové struktury (tj. B-strom, R-strom apod.), vyvážený. Při dotazování je nejprve pomocí grafového pseudo-isomorfismu efektivně nalezena množina listů, tedy kandidátů na výsledek \mathbf{G}_{vysl} . Ve verifikacní fázi je pak tato množina dále upravena vzhledem k požadavkům na výsledek, tj. opět jsou odstraněny falešně pozitivní výsledky vzhledem k tomu, jak přesný výsledek chceme.

13.4.3 Dotazovací jazyky pro grafy

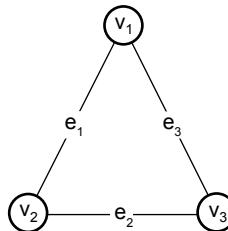
Stejně jako pro jakékoli jiné datové formáty (jako jsou data relační, XML, RDF, objektová atd.), tak i pro grafová data velice brzy vyvstala potřeba vytvoření vhodného dotazovacího jazyku. Ideálním cílem bylo vytvořit standard podobný

jazykům SQL, XQuery, RDF, OQL apod., tedy jazyk, který bude implementovat většina databázových systémů.

Existující dotazovací jazyky nad grafy se dají klasifikovat na *obecné* a *speciální*. Obecné dotazovací jazyky pracují s libovolným grafem, zatímco speciální jsou určeny pro grafy specifické pro určitou oblast, např. biologická data. Příkladem obecných dotazovacích jazyků pro grafy je např. jazyk GraphQL, Gremlin nebo Cypher. GraphQL krátce představíme v sekci 13.4.3.1, zbylé dva jazyky jsme již představili v sekcích 9.3.2 a 9.3.3, jelikož jsou používány v grafové databázi Neo4j. Zástupečných speciálních jazyků je celá řada. Např. jazyk PQL (viz sekce 13.4.3.2) je určený právě pro dotazování nad biologickými daty, zatímco jazyk BPMN-Q (viz sekce 13.4.3.3) slouží pro dotazování nad modely business procesů.

13.4.3.1 GraphQL

Jazyk GraphQL [99] byl navržen v roce 2008 s cílem vytvořit nástroj, který umožní nejen specifikovat strukturu grafu, ale také další podmínky, jež musí specifikovaný graf splňovat. Ukažme nejprve na několika příkladech, jak lze definovat strukturu grafu. Na obrázku 13.17 je ukázka jednoduchého grafu G_1 , který má tři uzly a tři hrany.



Obrázek 13.17: GraphQL – příklad grafu

Jeho reprezentace by v jazyce GraphQL vypadala takto:

```

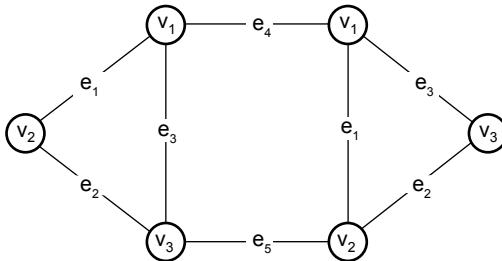
graph G1 {
    node v1, v2, v3;
    edge e1 (v1, v2);
    edge e2 (v2, v3);
    edge e3 (v3, v1);
}
  
```

Obrázek 13.18 na následující straně pak ukazuje, jak je možné z již existujícího grafu – v našem případě G_1 – vytvořit nový graf G_2 propojením vybraných uzlů novými hranami.

Jeho reprezentace v jazyce GraphQL by vypadala takto:

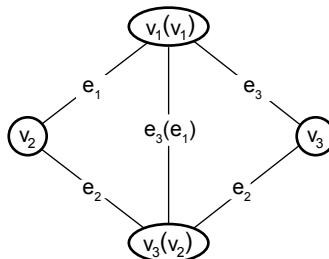
```

graph G2 {
    graph G1 as X;
    graph G1 as Y;
    edge e4 (X.v1, Y.v1);
    edge e5 (X.v3, Y.v2);
}
  
```



Obrázek 13.18: GraphQL – vytvoření grafu využitím již existujícího

Podobně obrázek 13.19 demonstruje způsob, jak vytvořit graf G_3 ze dvou grafů G_1 prostřednictvím sloučení (unifikace) vybraných uzlů a jím příslušejících hran.



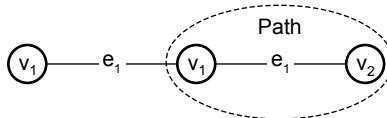
Obrázek 13.19: GraphQL – příklad sloučení uzlů a bran

V jazyce GraphQL je možné tuto situaci vyjádřit takto:

```
graph G3 {
    graph G1 as X;
    graph G1 as Y;
    unify X.v1, Y.v1;
    unify X.v3, Y.v2;
}
```

Zajímavým způsobem bychom mohli definovat cestu v grafu, tedy posloupnost hran sdílejících společný uzel. V následujícím příkladu, jehož grafické znázornění je uvedeno na obrázku 13.20 na následující straně, je pro tyto účely využit výběr z možností. Bud se jedná o graf sestávající ze dvou uzlů a hrany mezi nimi, nebo se jedná o složitější strukturu. Ta využívá rekurzivně sebe sama (tedy definici cesty) a prodlužuje ji o jednu novou hranu:

```
graph Path {
    graph Path;
    node v1;
    edge e1 (v1, Path.v1);
    export Path.v2 as v2;
} | {
    node v1, v2;
    edge e1 (v1, v2);
}
```



Obrázek 13.20: GraphQL – definice cesty

Definice cest pak umožňuje velmi snadnou definici cyklu, uvážíme-li, že cyklus je vlastně cesta, která je navíc spojena hranou mezi počátečním a koncovým uzlem:

```
graph Cycle {
    graph Path;
    edge e1 (Path.v1, Path.v2);
}
```

GraphQL umožňuje specifikovat také další podmínky, které musí graf splňovat. V následujícím příkladu je znázorněna nejen struktura grafu, ale i odpovídající podmínky na vlastnosti asociované s jeho uzly.

```
graph G4 {
    node v1;
    node v2;
    edge e1 (v1, v2);
} where v1.name = "Big Data a NoSQL databáze" and v2.copies > 10000;
```

V poslední ukázce můžeme vidět složitější příklad, který ze souboru `dblp.txt` obsahujícího na každém rádku dvojici spoluautorů postupně načte jednotlivé záznamy v podobě jednoduchých grafů P o dvou uzlech. Průchodem přes všechny tyto dvojice pak GraphQL vybuduje nový graf C tak, že pro každou načtenou dvojici (tedy každý graf P) provede následující kroky: K původnímu grafu C přidá oba uzly grafu P a spojí je novou hranou e_1 . Tyto nové uzly se pokusí unifikovat s již existujícími stejnojmennými uzly grafu C . Unifikace způsobí, že dojde k jednomu z následujících výsledků:

1. Pokud v grafu C již existují uzly se stejným jménem (atributem `name`), jsou tyto sloučeny (a případně je sloučena i již existující hrana mezi nimi, jinak je doplněna nová).
2. Nebo je sloučen alespoň jeden z uzlů, pro něž již v grafu C existuje stejnojmenný uzel, a druhý je k němu připojen novou hranou.
3. Nebo dojde ke vzniku nové komponenty grafu (tj. hrana s oběma uzly přibude do grafu, ale nebude s jeho uzly/hranami sloučena ani propojena).

Výsledný graf tedy bude reprezentovat spoluautorství – uzly budou odpovídat autorům (každý bude reprezentován právě jednou), hrany vztahu „být spoluautorem“.

```
graph P {
    node v1;
    node v2;
}
C = graph{};
for P exhaustive in doc("dblp.txt")
let C := graph {
```

```

graph C;
node P.v1, P.v2;
edge e1(P.v1, P.v2);
unify P.v1, C.v1 where P.v1.name = C.v1.name;
unify P.v2, C.v2 where P.v2.name = C.v2.name;
}

```

13.4.3.2 PQL

Jazyk PQL [123] je naopak jazykem určeným pro speciální účely, konkrétně pro dotazování nad biologickými daty – tzv. *biologickými sítěmi*, které mají formát grafu. Jedná se o deklarativní jazyk, pro jehož konstrukty se autoři inspirovali v jazyce SQL. Najdeme zde tedy např. známé klauzule **SELECT–FROM–WHERE**. Cílem dotazu je popsat požadovaný graf pomocí vlastností uzlů a cest mezi nimi. Příklad biologické sítě je uveden na obrázku 13.21 na následující straně. Uzly grafu odpovídají molekulám nebo interakcím a jsou propojeny orientovanými hranami. Graf navíc nemusí být souvislý, tj. může mít několik komponent, které nejsou propojeny.

Pro popis části grafu, která nás zajímá, máme v PQL konstrukty jako konstanty, operátory (např. pro porovnávání), funkce, logické spojky a mnoho dalších. Ukažme si základní principy pomocí několika příkladů. V tom následujícím hledáme takovou část grafu, v níž existují dva uzly A a B s uvedenými názvy **3-Isopropylmalate** a **EC1.1.1.85**.

```

SELECT *
FROM A, B
WHERE A.name = "3-Isopropylmalate" AND
      B.name = "EC1.1.1.85"

```

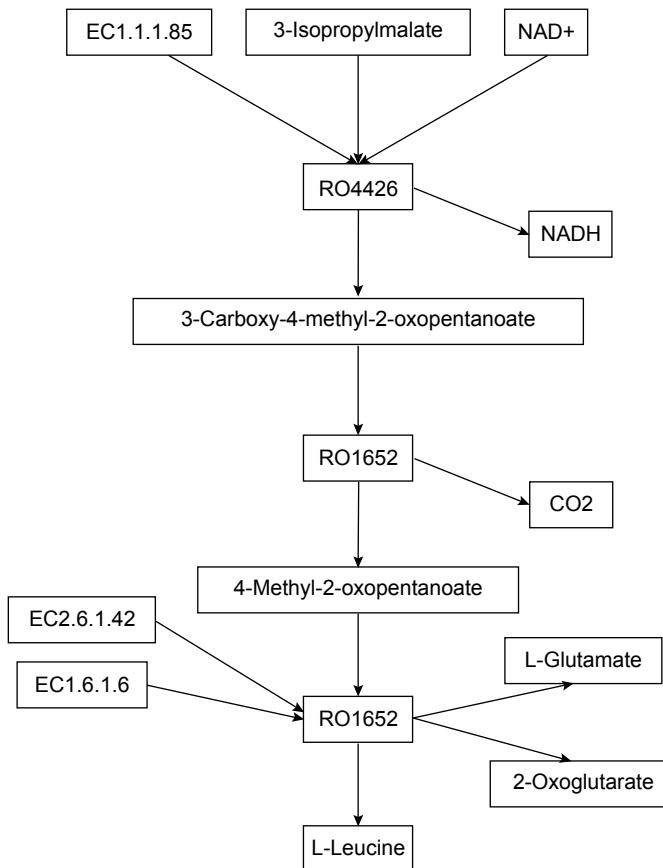
V následujícím dotazu hledáme podgraf obsahující tři uzly B, C a D, přičemž chceme, aby platilo, že mezi B a C i C a D vede cesta, D má název **L-Lactaldehyde**, C má název **Lactaldehyde** a B je typu enzym.

```

SELECT *
FROM B, C, D
WHERE B[-*]C[-*]D AND
      D.name = "L-Lactaldehyde" AND
      C.name = "Lactaldehyde" AND
      B ISA "Enzyme"

```

Obecně vyhodnocování dotazu funguje tak, že jsou proměnné (v našem případě A, B, C atd.) navázány na uzly tak, aby byly splněny všechny podmínky. Pro tyto účely jsou vyzkoušeny všechny možné varianty (proměnné jsou přiřazovány jak molekulám, tak interakcím). Výsledek dotazu je tvořen proměnnými navázanými na uzly a podgraferem specifikovaným v klauzuli **SELECT** (což je v našem případě celý odpovídající podgraf). Jinými slovy, tak jako SQL pracuje nad tabulkami a vrací tabulkou, tak PQL pracuje nad grafy a vrací graf.



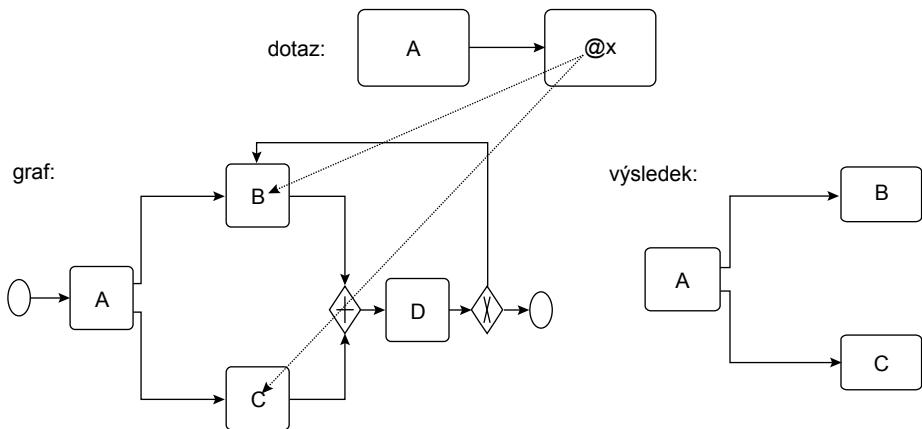
Obrázek 13.21: PQL – příklad dat

13.4.3.3 BPMN-Q

Ukažme ještě jeden příklad jazyka určeného pro speciální účely s poměrně neobvyklými konstrukty. Jedná se o jazyk BPMN-Q [60], který je určený pro dotazování nad modely business procesů vyjádřenými v BPMN (Business Process Model and Notation) [5]. BPMN slouží pro grafické znázorňování podnikových procesů pomocí procesních diagramů, které jsou podobné diagramům aktivit z UML. V BPMN diagramu (grafu) můžeme nalézt prvky jako např. události (počáteční, průběžné nebo koncové), aktivity (podprocesy nebo úlohy), brány, spojovací objekty (toky zpráv, asociace nebo sekvenční toky) a mnohé další.

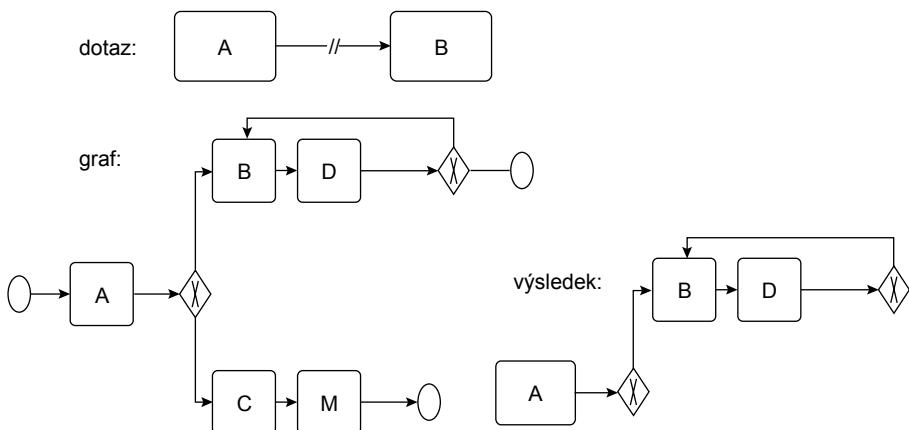
Jazyk BPMN-Q umožňuje klást nad diagramy v BPMN dotazy, které vyhledávají požadované podgrafy. Zajímavé je, že dotazový graf má stejně jako diagram grafickou podobu, která je rozšířena o nové speciální prvky. Uživatelé tedy pracují s obdobou jazyka, který už dobře znají. Podívejme se opět na několik příkladů.

Na obrázku 13.22 na následující straně je znázorněn příklad diagramu business procesů a dotaz, který obsahuje dva uzly – uzel A a proměnnou @X – mezi nimiž vede hrana. Jinými slovy hledáme všechny uzly, do nichž vede hrana z uzlu A. Výsledkem dotazu je odpovídající část grafu, jak je naznačeno ve třetí části obrázku.



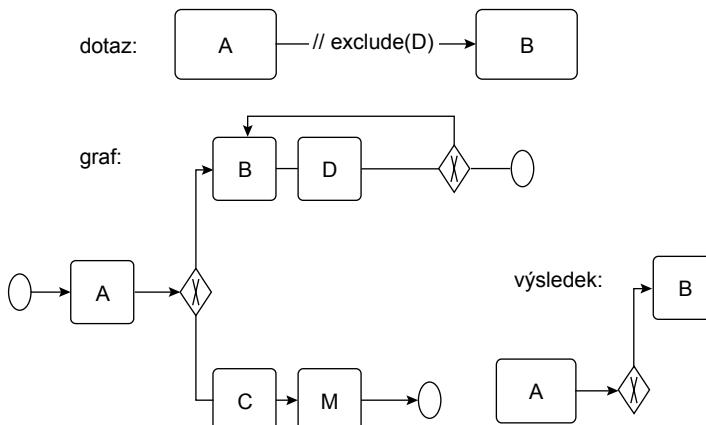
Obrázek 13.22: BPMN-Q – příklad 1

Druhý příklad, na obrázku 13.23, pokládá nad týmž diagramem business procesů dotaz pro nalezení všech cest z uzlu A do B. Vzhledem k tomu, že diagram obsahuje cykly, také do výsledku dostáváme cesty vedoucí přes odpovídající cykly.



Obrázek 13.23: BPMN-Q – příklad 2

Poslední příklad, na obrázku 13.24 na následující straně, využívá tzv. *negativní cestu*. Konkrétně hledá takové cesty z A do B, které nevedou přes uzel D.



Obrázek 13.24: BPMN-Q – příklad 3

13.4.3.4 Jazyk SPARQL

Podobně jako jsme nemohli nezmínit RDF databáze (viz sekce 9.6), jakožto specifický typ grafových databází, nemůžeme z přehledu dotazovacích jazyků nad grafy vyněchat jazyk SPARQL (SPARQL Protocol and RDF Query Language) [41] určený pro dotazování nad daty ve formátu RDF (viz sekce 2.4). Jak řekl Tim Berners-Lee,⁴ „používat sémantický web bez jazyka SPARQL je jako používat relační databáze bez jazyka SQL“ [109]. Existuje také jeho speciální rozšíření, jazyk SPARUL (SPARQL Update Language) [42], umožňující modifikaci RDF dat.

Jak jsme již viděli u mnoha jiných jazyků, i v případě SPARQL jsou hlavní prvky jazyka přímo inspirovány jazykem SQL. Nalezneme zde např. klasické klauzule **SELECT-FROM-WHERE**, které jsou přizpůsobeny datovému modelu RDF. Klauzule **FROM** specifikuje, nad kterými RDF zdroji dat identifikovanými prostřednictvím URI (což mohou být např. SPARQL end points) se dotazujeme. Klauzule **WHERE** specifikuje podmínky na data, která nás zajímají, tzv. *grafový vzor* (*graph pattern*). A klauzule **SELECT** specifikuje, jakou část výsledku splňujícího podmínky chceme na výstupu, samozřejmě včetně možnosti získat vše (prostřednictvím operátoru *****, podobně jako v SQL). Zajímavým aspektem je, že výsledkem SPARQL dotazu je množina trojic.

Z dalších klauzulí známých z SQL, které ve SPARQL nalezneme (a které jsou opět inspirovány jazykem SQL), jmenujme např. sjednocení (**UNION**), neboli disjunkci více dotazů, agregaci (**GROUP BY**), uspořádání výsledku (**ORDER BY**), omezení počtu výsledků (**LIMIT**), složitější regulární výrazy a mnohé další. Ve SPARQL ovšem nalezneme také zcela nové klauzule, vhodné právě pro práci s RDF daty a jejich specifickými vlastnostmi. Např. klauzule **OPTIONAL** umožňuje specifikovat nepovinnou část dat, která bude zahrnuta do výsledku pouze v případě, že je v datech obsažena, nebo klauzule **FILTER** umožňuje filtrovat výsledek dodatečnou podmínkou. Kromě dotazu typu **SELECT**, který vrací požadovanou podmnožinu dat ve

⁴ Sir Timothy John Berners-Lee je považován za autora myšlenky WWW (World Wide Web), kterou navrhl a jejíž základ – komunikaci přes internet mezi HTTP klientem a serverem – implementoval v roce 1989. V současné době je ředitelem konsorcia W3C, které dále dohliží na vývoj a standardizaci nejrůznějších WWW technologií.

formě množiny trojic, nabízí SPARQL ještě tři další typy dotazů. Dotaz typu **ASK** pouze testuje, zda má daný dotaz výsledek. Dotaz typu **DESCRIBE** vrací data spolu se všemi informacemi o zdroji, který vyhověl zadanému dotazu. A dotaz typu **CONSTRUCT** umožňuje z výsledků dotazu sestrojit nový graf odpovídající zadané šabloně. Dalo by se tedy říci, že poslední typ dotazu odpovídá XSLT transformacím XML dat, jelikož umožňuje transformovat RDF data na jiná RDF data.⁵

Jak je tedy vidět, jazyk SPARQL, stejně jako mnoho jiných jazyků, také využívá osvědčené postupy, které pouze vhodně rozšiřuje. Tento trend má své opodstatnění, jelikož si uživatelé nemusejí osvojovat zcela nové principy a seznámení s daným jazykem je pro ně velmi rychlé.

13.5 Závěr

Jak jsme již zmínili, grafové databáze tvoří specifický typ NoSQL databází, které mají osobitý datový model, tudíž vyžadují složitější algoritmy a metody pro ukládání i zpracování grafových dat. Naštěstí tuto složitou situaci významně zjednoduší propracovaná oblast teorie grafů, v níž stávající systémy nalézají mnoho užitečných algoritmů. Na druhou stranu dostaneme-li se do oblasti Big Data a tudíž potřebujeme graf distribuovat, je to zcela nová výzva i pro všechny tyto osvědčené a efektivní postupy. Tento úkol tedy nadále zůstává hlavním cílem grafových databází.

⁵ Je ale třeba dodat, že jazyk XSLT, v současné době ve verzi 3.0 z roku 2014, která se nachází ve stavu těsně před finální standardizací, umožňuje zpracovávat nejen XML data, ale také např. data ve formátu CSV (viz sekce 2.5). Podobně výstupním formátem nemusí být přímo formát XML, ale obecně jakýkoli textový formát.

14.

Další databáze pro Big Data

Jak jsme zmínili v úvodu knihy, pojem NoSQL databáze nemá přesnou definici ani jasné hranice. Označuje se tak zejména nová generace databázových systémů pro správu velkého množství dat, které stojí na jiných principech než tradiční relační databázové systémy. Také již víme, že existuje pojem *nepravé NoSQL databáze*, mezi něž řadíme např. nativní XML databáze nebo objektové databáze a které nemusejí být primárně škálovatelné pro velmi velká data. V této kapitole zmíníme ještě další typy databázových systémů, jež splňují naši vágní definici alespoň částečně – bud jsou založeny na jiných principech než relační databáze, nebo jsou škálovatelné horizontálně a tedy vhodné pro Big Data.

14.1 Hybridní databáze

Hybridními databázemi rozumíme takové systémy, které v sobě kombinují různé, dříve často oddělené přístupy. Za hybridní databázi můžeme například označit systém, který umožňuje zároveň pracovat s daty v relačním i dokumentovém datovém modelu. Nebo databázi, která dovoluje používat různé dotazovací prostředky nad jedním úložištěm dat. V následujícím textu stručně představíme některé zajímavé „hybridní“ rysy jedné open source databáze (systému PostgreSQL) a jedné komerční databáze (systému MarkLogic).

14.1.1 PostgreSQL

PostgreSQL¹ (též Postgres) je klasická relační databáze, jejíž vývoj začal již v 80. letech 20. století a je přímým dědicem databáze Ingres, jejímž hlavním tvůrcem byl Michael Stonebraker. Její implementace respektuje ISO/ANSI standard pro jazyk SQL [3]. Klade zásadní důraz na trvalost změn (durability) a integritu uložených dat, poskytuje pokročilou a efektivní master-slave replikaci (viz sekce 3.3.2) a umožňuje vytvářet indexy a další datové struktury pro zefektivnění dotazů.

Přesto můžeme ve vývoji PostgreSQL sledovat využití funkcí a technik, s jejichž obdobou se dnes v setkáváme v NoSQL světě. Jako příklad můžeme uvést *materializovaný pohled* (*materialized view*), který uloží výsledek určitého SQL dotazu do separátní tabulky a vytváří tedy kopii určité části dat ve zcela odlišné struktuře. Předpokládáme, že takový dotaz je zpravidla výpočetně náročný (obsahuje více operací JOIN či agregační funkce) a často se opakuje. Pro získání výsledku tudíž neopakujeme původní dotaz, ale získáme jej prostým čtením tabulky – analogicky k mnoha technikám ve světě NoSQL, které často duplikují data a optimalizují datové schéma jen proto, aby dosáhly rychlejšího čtení dat.

Kompromisy obvyklé ve světě NoSQL databází nalezneme též v možnostech replikace, které PostgreSQL nabízí. Bud můžeme využít *synchronní replikaci*, kdy operace provedená na primárním serveru čeká na své provedení na serveru sekundárním, a zaručuje tak, že při selhání primárního serveru nedojde ke ztrátě dat a že všechny servery budou v každém okamžiku vracet konzistentní data (jinými slovy zaručuje silnou konzistenci – viz sekce 3.2.3), nebo *replikaci asynchronní*, která bude rychlejší, ale nezajistí silnou konzistenci dat ani plnou ochranu proti selhání primárního serveru.

V několika posledních letech se však PostgreSQL stal jednou z nejvýkonnějších dokumentových databází [48], a to díky možnosti ukládat a dotazovat strukturovaná data ve formátu JSON. (I když je třeba připomenout, že se nejedná o klasickou NoSQL distribuovanou databázi určenou pro ukládání Big Data.) Tato vlastnost přitom o několik let předchází většinu v současnosti dominantních NoSQL systémů. Poprvé ji nalezneme ve verzi 8.2 z roku 2006,² která přinesla možnost ukládat dvojice (klíč, hodnota) pomocí datového typu **hstore**. Podpora pro formát JSON, reprezentovaná datovým typem **json** a související funkcionalitou, byla do PostgreSQL přidána ve verzi 9.2 v roce 2012.³ O dva roky později byl ve verzi 9.4 přidán datový typ **jsonb** (binární JSON).

14.1.1.1 PostgreSQL jako dokumentová databáze

S dokumenty v PostgreSQL pracujeme stejně jako s ostatními daty, tedy pomocí nastavení příslušného datového typu pro určitý sloupec v tabulce. V následujícím příkladu se jedná o sloupec s názvem **data** ve formátu **jsonb**:

```
CREATE TABLE people (
    id SERIAL PRIMARY KEY,
```

¹ <http://www.postgresql.org>

² První veřejná vydání databází Redis a MongoDB jsou až z roku 2009.

³ Podpora pro JSON přitom navazuje na existující podporu pro XML od verze 8.3.

```
data JSONB
);
```

Dokumenty pak vložíme do databáze jako text ve formátu JSON, stejně jako v případě jakékoliv nativní dokumentové databáze:

```
INSERT INTO people (data) VALUES ('  
{"first_name": "Jan",  
"last_name": "Novák",  
"birthday": "1970-01-01",  
"email": "jan@novak.name",  
"address": {  
"city": "Praha",  
"street": "Krásná 1"  
}  
}  
'');
```

```
INSERT INTO people (data) VALUES ('  
{"first_name": "Ivo",  
"last_name": "Novák",  
"birthday": "1975-02-01",  
"address": {  
"city": "Praha",  
"street": "Rovná 2"  
}  
}  
'');
```

```
INSERT INTO people (data) VALUES ('  
{"first_name": "Eva",  
"last_name": "Nováková",  
"birthday": "1973-03-01",  
"address": {  
"city": "Brno",  
"street": "Slunná 3"  
}  
}  
'');
```

A uložený dokument získáme standardním SQL SELECT příkazem:

```
SELECT *  
FROM people  
WHERE id = 1;  
  
id / data  
-----  
1 / {"first_name": "Jan", "last_name": "Novák", ...}
```

Navíc díky tomu, že PostgreSQL rozumí struktuře uložených JSON dokumentů, můžeme kupříkladu získat seznam jejich atributů:

```
SELECT DISTINCT json_object_keys(data::json)  
FROM people;
```

```
json_object_keys
```

```
-----  
first_name  
birthday  
last_name  
email  
address
```

Pomocí SQL notace, rozšířené o podporu práce s daty ve formátu JSON, můžeme získat pouze určité atributy, např. příjmení:

```
SELECT id, data->>'last_name' AS last_name  
FROM people;
```

```
id | last_name
```

```
-----+  
1 | Novák  
2 | Novák  
3 | Nováková
```

Všimněme si, že operátor `->>` (resp. `->`) pro přístup k jednotlivým atributům ve formě textu (resp. v původním datovém typu) odpovídá tečkové notaci používané v MongoDB a jiných dokumentových databázích (viz kapitola 7).

Podobně můžeme provádět dotazy s podmínkou v klauzuli `WHERE`, která zahrnuje položky v JSON dokumentech:

```
SELECT id, data->'address'->>'city' AS city  
FROM people  
WHERE data->>'last_name' = 'Novák';
```

```
id | city
```

```
-----+  
1 | Praha  
2 | Praha
```

Pro zefektivnění dotazu tohoto typu můžeme přitom i pro JSON struktury využít standardní funkce PostgreSQL – `index [32]`. A stejně jako jakýkoliv jiný sloupec tabulky, i vnořené atributy JSON dokumentu můžeme využít v komplexnějších operacích, jako je např. agregace pomocí `GROUP BY`:

```
SELECT data->'address'->>'city' AS city, COUNT(*) AS count  
FROM people  
GROUP BY city  
ORDER BY count DESC;
```

```
city | count
```

```
-----+  
Praha | 2  
Brno | 1
```

Pomocí konverzních funkcí PostgreSQL můžeme atributy dokumentu využít i pro časové kalkulace, např. pro získání aktuálního věku každé osoby:

```
SELECT id, EXTRACT(year FROM age(date(data->>'birthday'))) AS age
FROM people;
```

<i>id</i>	<i>age</i>
1	45
2	40
3	42

A hodí se i pro agregaci vracející průměrný věk všech osob v databázi:

```
SELECT avg(age(date(data->>'birthday'))) AS avg_age
FROM people;
```

<i>avg_age</i>
42 years 8 mons 14 days

Je přitom důležité, že JSON dokumenty ukládáme do jednoho konkrétního sloupce tabulky, která může samozřejmě obsahovat i další sloupce s běžnými datovými typy jako `integer`, `date` nebo `character`. Můžeme tedy volně kombinovat standardní, rigidní datové typy s bezschématovými daty ve sloupcích typu `json`, resp. `jsonb`. A samozřejmě můžeme kombinovat dotazy nad sloupci typu `json`, resp. `jsonb` s dotazy nad ostatními tabulkami, včetně podpory pro operace `JOIN`. V následujícím příkladu vytvoříme tabulku `salaries`, která bude udržovat informace o platech pro osoby z tabulky `people`. Tabulky budou přitom propojeny pomocí cizího klíče mezi dvěma sloupci datového typu `SERIAL`, se všemi zárukami, které tímto PostgreSQL jako relační databáze poskytuje.

```
CREATE TABLE salaries (
    id SERIAL PRIMARY KEY,
    person_id INTEGER UNIQUE REFERENCES people(id),
    salary BIGINT
);

INSERT INTO salaries (person_id, salary) VALUES (1, 110000);
INSERT INTO salaries (person_id, salary) VALUES (2, 120000);
INSERT INTO salaries (person_id, salary) VALUES (3, 130000);
```

Nyní můžeme vypsat jméno osoby z tabulky `people` a zároveň z tabulky `salaries` vypsat její plat:

```
SELECT people.id,
       concat(data->>'first_name', ' ', data->>'last_name') AS name,
       salaries.salary
  FROM people JOIN salaries ON (salaries.person_id = people.id);
```

<i>id</i>	<i>name</i>	<i>salary</i>
1	Jan Novák	110000
2	Ivo Novák	120000
3	Eva Nováková	130000

Operaci JOIN můžeme provést také přímo na hodnotách z dat ve formátu JSON. V následujícím příkladu vytvoříme tabulkou poštovních směrovacích čísel a příslušných měst a naplníme ji dvěma hodnotami.

```
CREATE TABLE postcodes (
    postcode TEXT PRIMARY KEY,
    city TEXT
);

INSERT INTO postcodes (postcode, city) VALUES ('CZ-11000', 'Praha');
INSERT INTO postcodes (postcode, city) VALUES ('CZ-60200', 'Brno');
```

Nyní můžeme k jednotlivým osobám z tabulky `people` vypsat města, kde žijí, společně s PSČ z tabulky `postcodes`.

```
SELECT concat(data->>'first_name', ' ', data->>'last_name') AS name,
       city,
       postcode
  FROM people JOIN postcodes
    ON (people.data->'address'->>'city' = postcodes.city);

  name      | city   | postcode
-----+-----+-----
Eva Nováková | Brno  | CZ-60200
Jan Novák   | Praha | CZ-11000
Ivo Novák   | Praha | CZ-11000
```

PostgreSQL poskytuje pro práci s JSON daty mnoho funkcí a operátorů [33], které se neustále vyvíjejí. Jsou to jednak operátory umožňující navigaci v rámci JSON dokumentu (např. výše zmíněný operátor `->>`) a predikáty pro vyhodnocování jednoduchých podmínek nad JSON daty: např. výraz `data @> '{"first_name" : "Jan"}'` otestuje, zda pole `data` obsahuje daný JSON objekt. Dále to jsou funkce pro vytváření JSON dokumentu z textových nebo relačních dat, např. funkce `row_to_json`, která je dobře použitelná pro formátování výstupu SQL dotazu. A v neposlední řadě se jedná o funkce pro zpracování JSON dat, např. funkce `json_to_record` vytvoří z JSON objektu relační záznam s tím, že specifikujeme vybrané atributy a jejich typy v dočasně vytvořené relaci. V následujícím příkazu se tato relace jmenuje `x`:

```
SELECT people.id, x.*
  FROM people, jsonb_to_record(people.data)
    AS x(first_name text, last_name text, birthday text, email text);

  id | first_name | last_name | birthday |      email
-----+-----+-----+-----+-----
  1 | Jan        | Novák    | 1970-01-01 | jan@novak.name
  2 | Ivo        | Novák    | 1975-02-01 |
  3 | Eva        | Nováková | 1973-03-01 |
```

S takto vytvořenými relačními daty bychom dále mohli pracovat pomocí SQL tak, jak jsme zvyklí, například je propojit s jinou tabulkou pomocí operace `JOIN`.

Jak vidíme, možnosti, které PostgreSQL pro práci s dokumenty ve formátu JSON nabízí, se v mnoha ohledech vyrovnají nativním dokumentovým databázím, jako

je např. MongoDB (viz kapitola 7). Poněkud překvapivé může být, že co se efektivity a rychlosti týče, dokonce je předčí. Při srovnávacím měření konzultační společnosti EnterpriseDB⁴ byla databáze PostgreSQL více než dvakrát rychlejší než databáze MongoDB při zápisu i čtení, a to ve výchozí instalaci, tedy se všemi zárukami pro trvalost a konzistenci dat, které nabízí [48].⁵

Lze přitom očekávat, že uvedením podpory pro dokumenty ve formátu JSON sbližování PostgreSQL a nerelačních databází nekončí. Nativní podpora pro jazyk Javascript umožňuje například interní validaci JSON dokumentů pomocí operace `ADD CONSTRAINT` [124]. Další metou bude pravděpodobně *multi-master replikace* [30] a podpora pro automatické rozdělení dat mezi několik běžících instancí.⁶

14.1.2 MarkLogic

Zatímco většina hybridních databází má kořeny v relačních databázích, prošla databáze MarkLogic⁷ zcela jiným vývojem. Původně se jednalo o klasickou nativní XML databázi – zařadili bychom ji tedy mezi dokumentové databáze. Se vzrůstající oblibou formátu JSON byla do MarkLogic přidána i jeho nativní podpora. Indexování a vyhledávání pak funguje pro XML i JSON data velice podobným způsobem.

Kromě toho lze do databáze ukládat i prostorová (*geospatial*) data a RDF data – ta jsou ukládána v dokumentovém úložišti, ale pro efektivní práci s nimi se používají speciální indexy. Do databáze lze ukládat i libovolná další binární data (obrázky, videa apod.), ale zde jsou již možnosti indexování a vyhledávání velice omezené.

Jedinečnou vlastností MarkLogic je možnost používat několik dotazovacích jazyků, podle toho, co se pro daný úkol nejvíce hodí. Pro data uložená v XML tak nejčastěji využijeme XQuery, pro JSON pak Javascript a pro RDF jazyk SPARQL. Pokud jsme skalní příznivci jazyka SQL, můžeme vytvořit pohled, který JSON nebo XML data mapuje na plochou strukturu, a nad tou pak dotazy klást pomocí jazyka SQL.

Pro psaní dotazů nad opravdu velkými daty lze využít princip MapReduce. Do databáze MarkLogic je totiž integrován systém Hadoop. Místo klasického HDFS úložiště (viz sekce 4.3.1) je pak možné jak pro zdrojová data, tak i pro výsledky používat přímo databázi MarkLogic.

MarkLogic je komerční produkt, ale pro vývojáře a pro potřeby seznámení s produktem je k dispozici licence zdarma.

⁴ <http://www.enterprisedb.com>

⁵ Toto měření přitom pracuje pouze s jedinou instancí MongoDB, neboť PostgreSQL nativně nepodporuje distribuci dat do několika oddílů (*shards*).

⁶ Již nyní existuje externí projekt pg_shard (https://github.com/citusdata/pg_shard), který poskytuje plně integrovanou podporu pro sharding a replikaci dat.

⁷ <http://www.marklogic.com>

14.2 Databáze ve webovém prohlížeči

Na první pohled by se mohlo zdát, že webové prohlížeče nemají s databázemi, natož těmi NoSQL nic společného. Opak je však pravdou. Každý moderní prohlížeč dnes v sobě obsahuje několik NoSQL databází. Abychom pochopili proč, je třeba stručně připomenout historii vývoje webu. V počátcích byl prohlížeč poměrně primitivní software, který jen zobrazoval statické dokumenty zapsané pomocí kombinace jazyků HTML a CSS. Později bylo možné do HTML stránky vložit formulář, který uživatel vyplnil a prohlížeč vyplněná data odeskal na webový server k dalšímu zpracování. Prohlížeč toho ale pořád nemusel moc umět.

S tím, jak prohlížeče začaly podporovat vkládání programů v Javascriptu přímo do kódu HTML, začaly se vytvářet interaktivní webové aplikace, kde se čím dál více aplikační logiky přesouvalo právě do prohlížeče v podobě kódu v Javascriptu. S tím, jak schopnosti prohlížečů a celé webové platformy rostly, začaly být zajímavou alternativou pro nativní aplikace. Jediným problémem zůstalo zajištění funkčnosti webové aplikace i bez připojení k internetu.

Představme si, že bychom chtěli, aby s webovou e-mailovou službou jako GMail⁸ šlo pracovat i v době, kdy nebudeme mít dostupné připojení k internetu. Kód aplikace v Javascriptu může do prohlížeče stáhnout nepřečtené e-maily a uložit je přímo v prohlížeči. Při následném výpadku připojení budeme moci číst alespoň e-maily uložené v prohlížeči, odpovídат na ně a psát nové. Neodeslané e-maily se rovněž uloží v prohlížeči a odešlou se až v okamžiku, kdy se obnoví připojení k internetu. Zmíněné e-maily bychom mohli uložit přímo do datových struktur běžícího javascriptového kódu. O tato data bychom však přišli v okamžiku zavření okna prohlížeče. Podobné druhy aplikace proto pro úspěšnou práci potřebují nějaké trvalejší úložiště dat, které bude dostupné v prohlížeči. Takových úložišť historicky vzniklo několik. Již dvacet let je možné využívat *cookies*,⁹ ale jejich kapacita je velice omezená, takže se nehodí pro aplikace, kde je potřeba ukládat více než jen tucet údajů. Cookies proto v moderních webových aplikacích nahradilo Web Storage.

14.2.1 Web Storage

Web Storage [53] (neboli webové úložiště) je jednoduché úložiště typu klíč-hodnota, které může využívat každá webová aplikace. Jako klíč a hodnotu lze používat pouze textové řetězce a k dispozici jsou operace pro uložení, čtení a mazání hodnot pro daný klíč. Navíc je možné zjistit počet uložených dvojic (klíč, hodnota) a získat existující klíče. Objem uložených dat je pro každou doménu, ze které se

⁸ <https://mail.google.com>

⁹ Cookies vznikly jako rozšíření protokolu HTTP. Protokol HTTP byl navržen jako bezstavový, aby byl co nejjednodušší, což v začátcích webu pro přenos statických webových stránek stačilo. S rozvojem webových aplikací však bylo potřeba udržovat pro jednotlivé uživatele pracující s webovou aplikací stavovou informaci – např. pro internetový obchod je typickou stavovou informací obsah nákupního košíku. Cookie je krátká informace, kterou webový server odešle prohlížeči, a ten ji při všech následujících požadavcích na stejný server automaticky vrací zpět. Do cookie tak můžeme uložit přímo krátkou stavovou informaci nebo jednoznačný identifikátor uživatelské relace (*session*), pod kterým je stavová informace uložena na serveru.

spouští nějaká aplikace, omezen. V závislosti na použitém prohlížeči se jedná typicky o 5 až 25 MB dat.

Web Storage tvůrcům aplikací nabízí dvě varianty úložiště – `sessionStorage` a `localStorage` – jež se liší ve svém chování. První z nich je oddělené pro každou záložku prohlížeče, ve které aplikace běží, a data se nesdílejí. Druhé je sdílené přes všechna okna a záložky prohlížeče, ve kterých aplikace běží. Pro ilustraci rozdílu si můžete otevřít ukázku z příkladu 14.1 najednou v několika záložkách prohlížeče, přepínat se mezi `sessionStorage` a `localStorage` a pozorovat rozdíly v chování.

Příklad 14.1: Použití Web Storage API

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Web Storage Demo</title>
  </head>
  <!-- Po načtení stránky zobrazíme obsah úložiště -->
  <body onload="displayStorage();">
    <script>
      // na začátku pracujeme se sessionStorage
      var storage = sessionStorage;

      // funkce pro vypsání obsahu zvoleného úložiště
      function displayStorage()
      {
        var container = document.getElementById("result");
        container.innerHTML = "Používám: " + (storage == localStorage ? "localStorage" : "sessionStorage");
        var table = document.createElement("table");
        container.appendChild(table);

        // projdeme všechny prvky v úložišti
        for (var i=0; i<storage.length; i++)
        {
          // načtení hodnoty klíče s daným pořadím
          var key = storage.key(i);
          // načtení hodnoty uložené pod klíčem
          var value = storage.getItem(key);

          // vypsání klíče a hodnoty do podoby jednoduché tabulky
          var row = document.createElement("tr");
          var keyCell = document.createElement("td");
          keyCell.innerHTML = key;
          var valueCell = document.createElement("td");
          valueCell.innerHTML = value;
          var deleteCell = document.createElement("td");

          // přidání tlačítka pro odstranění hodnoty
          deleteCell.innerHTML = "<button onclick='storage.removeItem(\"" + key + "\");displayStorage();'>Smaz</button>";
          row.appendChild(keyCell);
          row.appendChild(valueCell);
          row.appendChild(deleteCell);
        }
      }
    </script>
  </body>
</html>
```

```

        table.appendChild(row);
    }
}
</script>

<!-- vstupní pole pro přidání nové položky do úložiště -->
<div>
    <input name="key"> = <input name="value"> <button ▶
    onclick="storage.setItem(document.getElementsByName('key').item(0).value, ▶
    document.getElementsByName('value').item(0).value); displayStorage();">Přidej</button>
</div>

<!-- tlačítka pro provádění běžných operací -->
<button onclick="displayStorage();">Vypiš vše</button>
<button onclick="storage.clear();displayStorage();">Smaž vše</button>
<button onclick="storage=localStorage;displayStorage();">Přepni na localStorage</button>
<button onclick="storage=sessionStorage;displayStorage();">Přepni na sessionStorage</button>

<!-- místo pro zobrazení výsledků -->
<div id="result"></div>

</body>
</html>
```

Rozhraní objektů úložiště je velmi jednoduché. Metoda `setItem(klíč, hodnota)` dovoluje přidat novou položku. Pomocí `getItem(klíč)` můžeme hodnotu zase přečíst. Kromě tohoto zápisu lze klíč použít přímo jako jméno vlastnosti objektu. Následující dva zápisy čtení dat jsou tak totožné:

```

localStorage.setItem("user", "Pepa");

localStorage.getItem("user");      // vrátí "Pepa"
localStorage.user;                // rovněž vrátí "Pepa"
```

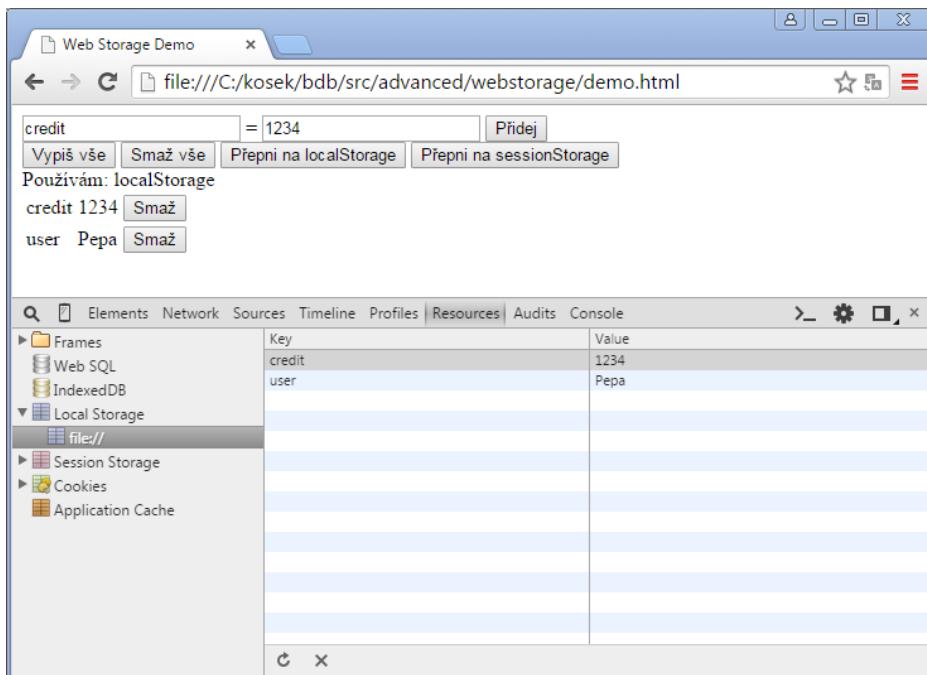
Položku můžeme odstranit pomocí `removeItem(klíč)`. Pod jedním klíčem může být uložena jen jedna hodnota a při opakovaném zápisu se stejným klíčem se tak hodnoty přepsují. Pomocí metody `clear()` lze celé úložiště smazat.

V příkladu je také vidět použití vlastnosti `length`, která vrací počet uložených položek, a metodu `key()`, jež dovoluje zjistit hodnoty jednotlivých klíčů.

Většina moderních prohlížečů obsahuje nástroje, které dovolují snadné prohlížení aktuálního obsahu Web Storage. Obrázek 14.1 na následující straně ukazuje zobrazení obsahu úložiště pomocí vývojářských nástrojů.

14.2.2 Indexed Database

Indexed Database (známá pod názvem IndexedDB) [19] je pokročilejší databází typu klíč-hodnota, která je v prohlížeči dostupná. Na rozdíl od Web Storage nabízí přídavné vlastnosti, jež dovolují její využití i pro práci s většími objemy dat (z pohledu webového prohlížeče, ne z pohledu Big Data). Maximální velikost dat,



Obrázek 14.1: Zobrazení obsahu Web Storage pomocí vývojářských nástrojů prohlížeče Chrome

kterou můžeme uložit to IndexedDB, je pro každý prohlížeč jiná a většinou je i závislá na kapacitě pevného disku. Typicky se pohybuje od stovek MB do desítek GB.

Největší rozdíly oproti Web Storage jsou:

- Jako hodnotu lze ukládat javascriptové objekty, ne jen prosté řetězce.
- Pod stejným klíčem je možné mít uloženo několik objektů a přistupovat ke všem z nich.
- Klíč může být určen ručně, přidělen automaticky nebo přečten přímo z ukládaného objektu.
- Operace se provádějí v transakcích.
- Operace se typicky provádějí asynchronně – tak jak je to zvykem v klientském Javascriptu.
- Pro databázi můžeme definovat indexy, pomocí nichž pak můžeme data rychle prohledávat.

V prohlížečích bývá implementace IndexedDB rozhraní poměrně efektivní a ve spojení s indexy se tak hodí i pro práci s většími objemy dat (opět z pohledu prohlížeče). Opět zde platí omezení na maximální velikost dat uložených pro jednu doménu. Limity jsou však obvykle podstatně větší než u Web Storage.

SQL databáze přímo v prohlížeči

Jednu dobu dokonce konsorcium W3C uvažovalo o tom, že by standardní součástí prohlížeče byla přímo databáze podporující jazyk SQL, tzv. *Web SQL Database* [52]. Nakonec se však tato myšlenka zavrhlala, protože návrh Web SQL Database byl málo obecný a kopíroval rozhraní databáze SQLite,¹⁰ kterou používala implementace v prohlížeči Chrome. Nakonec dostalo přednost IndexedDB, které je obecnější a jednodušší (např. nenabízí dotažovací jazyk), ale pro typické aplikace poskytuje dostatečné prostředky.

14.3 NewSQL databáze

Poměrně novým pojmem v oblasti databázového zpracování Big Data jsou tzv. *NewSQL databáze*. Zjednodušeně řečeno by se dalo říct, že se jedná o systémy, které nabízejí jak škálovatelné úložiště, tak veškerou funkcionality, na kterou jsme zvyklí u klasických relačních databází typu klient-server. Funkcionality se nemyslí pouze jazyk SQL, jak naznačuje název této třídy databází, ale především osvědčený relační model dat a vlastnosti ACID.

Pojem NewSQL se pravděpodobně poprvé objevil v článku [59] z roku 2011. Autor, analytik z 451 Group,¹¹ v něm rozlišuje dva přístupy, které mají stejný cíl. V prvním případě se jedná o distribuované systémy, které přidávají výhody relačního modelu a ACID vlastnosti. Obvykle jsou to systémy, jež vznikly pro tyto potřeby zcela nově. Mezi ně jsou obvykle zahrnovány např. systémy Clustrix,¹² ScaleArc,¹³ MemSQL,¹⁴ VoltDB¹⁵ a další. Ve druhém případě se jedná o původně relační databáze, které byly rozšířeny o techniky umožňující horizontální škálovatelnost. Tento typ databází reprezentuje např. systém TokuDB¹⁶ nebo JustOne DB.¹⁷ V prostředí cloudů později vznikl nový související pojem *NewSQL as a Service*, který reprezentuje myšlenku poskytovat jako cloudovou službu právě NewSQL databázi, tedy horizontálně škálovatelný relační databázový systém. Příkladem takového systému je Amazon Relational Database Service¹⁸ nebo Azure SQL Database¹⁹ firmy Microsoft. Přestože mají tyto systémy velmi odlišnou architekturu, jejich společným rysem je podpora relačního modelu a jazyka SQL.

Jak je tedy vidět, tato myšlenka se stala velmi rychle populární a systémů existuje poměrně hodně. Základní otázkou ovšem je, k čemu vlastně potřebujeme tento nový typ databází? Mezi hlavní důvody patří především tyto dva:

¹⁰ <https://www.sqlite.org>

¹¹ <https://451research.com>

¹² <http://www.clustrix.com>

¹³ <http://scalearc.com>

¹⁴ <http://www.memsql.com>

¹⁵ <http://voltedb.com>

¹⁶ <http://www.tokutek.com/tokudb-for-mysql/>

¹⁷ <http://www.justonedb.com>

¹⁸ <http://aws.amazon.com/rds/>

¹⁹ <https://msdn.microsoft.com/en-us/library/azure/ee336279.aspx>

1. Existuje mnoho aplikací pracujících s relačním modelem, které potřebují řešit náhlý nárůst objemu dat. Přechod na některý z modelů v NoSQL databázích a s ním související jiný způsob dotazování a jiné vlastnosti úložiště by znamenal jejich časově a finančně velmi náročné modifikace.
2. I pokud by přechod na jiný datový model nebyl až takovým problémem, mnoho aplikací si ze své podstaty nemůže dovolit vzdát se požadavků na silnou konzistenci dat, kterou NoSQL databáze běžně nenabízejí.

V obou případech tedy potřebujeme zachovat tradiční vlastnosti relačních databází, ale současně horizontálně škálovat. Jak je tedy vidět, stejně jako NoSQL databáze neznamenají konec klasických relačních databází, není příchod NewSQL databází hrozbo pro NoSQL systémy. Na tento typ systémů je opět třeba nahlížet jako na další alternativu, která řeší nově vzniklé potřeby určitého typu současných aplikací. Ostatně již v roce 2007 uvádí Michael Stonebraker a kol. v článku [150] nutnost vzniku různých typů databázových systémů pro různorodé účely. Dnes je tato myšlenka označována jako polyglotní persistence (viz *Polyglot persistence* na straně 125).

V následující sekci krátce představíme jeden z nejznámějších NewSQL databázových systémů, systém VoltDB, na němž ukážeme možné základní vlastnosti systému tohoto typu.

14.3.1 VoltDB

Databázový systém VoltDB byl vybudován na základě experimentálního akademického systému H-System,²⁰ který byl vytvořen ve spolupráci databázových expertů z předních amerických univerzit²¹ a firmy Intel.²² Jedná se o škálovatelný systém, který garantuje ACID vlastnosti a je založen na relačním modelu dat.

Z uživatelského hlediska je díky jazyku SQL a relačnímu modelu práce s daty velmi snadná a významně se neliší od klasických databázových systémů. SQL nám umožňuje vytvářet tabulky pomocí příkazu `CREATE TABLE`, naplnit je např. příkazem `INSERT INTO` (popř. načtením ze souboru nebo přesměrováním výstupu dotazu), upravit pomocí `ALTER TABLE` a smazat pomocí `DROP TABLE`. Můžeme také využívat základní integritní omezení, včetně primárních klíčů (nicméně omezení `CHECK` a cizí klíče zatím podporovány nejsou) a pro optimalizaci dotazů vytvářet indexy. Pro dotazování máme k dispozici příkaz `SELECT`, který zahrnuje složitější konstrukty, jako jsou agregace (prostřednictvím klauzule `GROUP BY`), množinové operace, vnořené poddotazy nebo dokonce uložené příkazy SQL a procedury implementované v jazyce Java či jejich kombinaci. Můžeme jej také využít pro vytváření databázových pohledů.

Jelikož předpokládáme ukládání Big Data, mohou tabulky obsahovat velké množství záznamů, které potřebujeme distribuovat. VoltDB zajišťuje distribuci automaticky, nicméně uživatel může systému „poradit“, podle jakého sloupce je vhodné distribuci dělat. Např. zákazníky můžeme distribuovat podle měst, zemí

²⁰ <http://bstore.cs.brown.edu>

²¹ Není překvapivé, že se na něm podílel i Michael Stonebraker.

²² <http://istc-bigdata.org>

nebo typu. Podobně můžeme distribuovat i uložené procedury. Předpokládejme, že máme tabulkou zákazníků distribuovanou podle zemí. Distribuovanou proceduru, která s tabulkou pracuje, pak můžeme spustit pouze nad určitou částí tabulky, tj. nad daty určité země, jež jsou uložena na jediném uzlu.

Architektura VoltDB je tedy distribuovaná, konkrétně tzv. *shared-nothing*, což znamená, že uzly v clusteru mezi sebou nesdílí ani paměť, ani disk. Jedná se tedy o autonomní jednotky, které mezi sebou komunikují prostřednictvím zpráv. Data jsou v systému primárně zpracovávána v paměti (*in-memory database*), čímž je zajištěna maximální efektivita.

Aby byl systém opravdu efektivní, předpokládá distribuci tabulek vhodnou pro danou konkrétní aplikaci, která navíc odpovídá příslušným distribuovaným uloženým procedurám. Ty pak umožňují jednotlivé části dat zpracovat lokálně. Každá uložená procedura odpovídá jedné transakci. VoltDB transakce v rámci jednoho uzlu serializuje, tudíž nemusí řešit zámky, logy apod., nicméně díky distribuci dat umožňuje paralelní zpracování více transakcí najednou. Pokud některá procedura potřebuje data z více uzlů, jeden z uzlů se stane koordinátorem, spustí příslušné lokální požadavky, sloučí jejich výsledky a proceduru dokončí. Tento typ úlohy je ovšem pomalejší než nezávislé distribuované procedury.

Pro další zefektivnění nabízí systém replikaci dat, jež umožňuje škálovat jejich čtení. Tento přístup se hodí především pro data, která nejsou často modifikována.

14.4 Array databases

Posledním typem databází určených pro Big Data, který představíme, jsou tzv. *array databases*, tedy česky bychom mohli říct databáze polí, ale tento pojem se nepoužívá. Jedná se o databázové systémy určené speciálně pro data, která jsou reprezentována jako jedno nebo vícerozměrná pole. Taková data se objevují v případech, kdy potřebujeme ukládané hodnoty reprezentovat v prostoru a/nebo v čase. Obvykle pocházejí z různých oblastí biologie, chemie, fyziky, geologie a jiných přírodních věd a typicky se využívají pro specifické komplexní vědecké analýzy přírodních jevů. Příkladem dat mohou být různá astronomická měření, klimatické změny, satelitní snímky Země, oceánografická data, lidský genom apod.

Uvažme např. satelitní snímky. Každý snímek můžeme reprezentovat jako dvourozměrné pole, kde jedním rozměrem je zeměpisná šířka a druhým zeměpisná délka, přičemž v buňce takového pole máme uloženou příslušnou hodnotu (nebo kombinaci hodnot) odpovídající nasnímané informaci v daném místě. Třetím rozměrem může být např. čas, kdy byl snímek pořízen, dalšími rozměry mohou být např. charakteristiky snímacího zařízení, které se také mohou lišit.

Jak je z uvedených příkladů dat zřejmé, obvykle se jedná o Big Data, navíc s velmi specifickými a náročnými požadavky na reprezentaci i analýzu. I kdyby nešlo o Big Data, tak tato data nejsou vhodná pro ukládání do plochých relací klasických relačních databázových systémů. V nich sice (po porušení první normální formy) nalezneme datový typ pole, ale často s omezenou dimenzí a s minimem souvisejících analytických operací. Jak již víme, datové modely používané v NoSQL data-

bázích také nejsou určeny pro vícerozměrná pole. Uložit by je sice uměly, ale efektivita operací by byla značně limitována. Vzhledem k tomu vzniklo v nedávné době několik specifických databázových systémů, jejichž logický model odpovídá požadavkům vícerozměrných polí, jako je např. databáze SciDB, kterou vzhledem k pojednání o významu vícerozměrných polí, nebo systém Oracle Spatial and Graph.²⁴

14.4.1 SciDB

Systém SciDB firmy Paradigm²⁵, kterou spoluzaložil Michael Stonebraker je pravděpodobně jedním z nejpopulárnějších systémů tohoto typu, a to nejen kvůli svému spoluautorovi, ale především vzhledem k pojednávání o významu vícerozměrných polí.

Datový model, s nímž SciDB pracuje, odpovídá vícerozměrnému uspořádanému poli, přičemž předpokládá, že data v databázi nejsou přepisována. Pokaždé je tedy vytvořena nová verze dat, která umožňuje analyzovat změny a opravy v čase. SciDB umí efektivně ukládat jak hustá, tak řídká pole libovolně velkých dimenzí.

Pro práci s daty nabízí SciDB dva nástroje:

- deklarativní jazyk AQL (Array Query Language),
- funkcionální jazyk AFL (Array Functional Language).

Jazyk AQL je inspirován jazykem SQL, nicméně namísto tabulek pracuje s více-rozměrnými poli, přičemž pro jejich zpracování a analýzu nabízí podstatně širší množinu operací. Také se skládá z částí DDL (Data Definition Language) pro vytváření polí a jejich naplnění daty a DML (Data Manipulation Language) pro dotazování a operace s daty.

Pro představu, jak se s daty v systému pracuje, nejprve v následujícím příkladu vytvoříme pole s názvem A a následně jej naplníme daty z uvedeného CSV souboru.

```
CREATE ARRAY A <x: double, err: double> [i=0:99,10,0, j=0:99,10,0];
```

```
LOAD A FROM '../examples/A.scidb';
```

Každé pole musí mít minimálně jeden *atribut*, tedy hodnotu určitého datového typu, která je ukládána do jednotlivých *buněk* pole. Pole A bude mít dva atributy x a err, oba s datovým typem double. Dále musí mít každé pole alespoň jednu *dimenzi*. Pole A bude mít dvě dimenze s názvem i a j. Každá dimenze pak má definované *souřadnice* (které nemusejí být omezeny), velikost datového úseku (*chunk*) a případný překryv úseků. Doporučená velikost datového úseku pro efektivní práci databáze je 10-20 MB (příliš malé nebo příliš velké úseky komplikují např. distribuci dat). SciDB totiž data v polích distribuuje a zpracovává právě po specifikovaných úsecích, jejichž velikost určíme v závislosti na datových typech atributů. Překryvy datových úseků využívají nemusíme, ale hodí se např. v případě, že se potřebujeme efektivně dotazovat na nejbližší sousedy, které bychom jinak pravděpodobně měli uložené na jiném uzlu v clusteru. Obě dimenze pole A mají

²³ <http://www.rasdaman.com>

²⁴ <http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html>

²⁵ <http://www.paradigm4.com>

stejné charakteristiky – souřadnice od 0 do 99 s úsekem o velikosti 10 buněk a nulovým překryvem.

Pro dotazování nad poli nabízí jazyk AQL podobnou syntaxi jako SQL. V následujícím příkladu nejprve vytvoříme dvě jednorozměrná pole, pak na nich ukážeme základy dotazování.

```
CREATE ARRAY A <val_a:double>[i=0:9,10,0];
LOAD A FROM '../examples/exA.scidb';
CREATE ARRAY B <val_b:double>[j=0:9,10,0];
LOAD B FROM '../examples/exB.scidb';

// vypsání hodnot souřadnice i z pole A
SELECT i FROM A;
[(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)]

// vypsání hodnot atributu val_a z pole A a val_b z pole B
SELECT val_a FROM A;
[(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)]
SELECT val_b FROM B;
[(101),(102),(103),(104),(105),(106),(107),(108),(109),(110)]

// využití klauzule WHERE a funkce sqrt (tj. výpočet druhé odmocniny)
SELECT sqrt(val_b) FROM B WHERE j>3 AND j<7;
[(),(),(),(),(10.247),(10.2956),(10.3441),(),(),()]
```

Příkaz SELECT je pochopitelně zajímavý především v okamžiku, kdy spojujeme více polí. Základní operací je *vnitřní spojení*, které spojí hodnoty v buňkách dvou polí. Spojovaná pole musí být *kompatibilní*, tj. musí mít stejně počáteční souřadnice dimenzí, velikosti datových úseků a překryvy. Množství a typy atributů se mohou lišit – ty jsou prostě sloučeny dle zadанé operace, resp. podmínky, jak je vidět v následujícím příkladu.

```
// spojení hodnot polí A a B a uložení výsledku do pole C
SELECT * INTO C FROM A, B;
[(1,101),(2,102),(3,103),(4,104),(5,105),(6,106),(7,107),(8,108),(9,109),(10,110)]

// spojení polí C a B a uložení výsledku do pole D
SELECT * INTO D FROM C,B;
[(1,101,101),(2,102,102),(3,103,103),(4,104,104),(5,105,105),(6,106,106),(7,107,107),
(8,108,108),(9,109,109),(10,110,110)]

// výpis informací o poli D (všimněme si zpracování atributů se stejným názvem)
SELECT * FROM show(D);
[("D<val_a:double,val_b:double,val_b_2:double> [i=0:9,10,0]")]

// spojení hodnot součtem
SELECT C.val_b + D.val_b FROM C,D;
[(202),(204),(206),(208),(210),(212),(214),(216),(218),(220)]

// spojení pole A sama se sebou
SELECT a1.val_a,a2.val_a+2 FROM A AS a1,A AS a2;
[(1,3),(2,4),(3,5),(4,6),(5,7),(6,8),(7,9),(8,10),(9,11),(10,12)]
```

Další typy spojení umožňují operace jako **MERGE** (slévání hodnot dvou polí se stejnými charakteristikami do jednoho výsledného pole, při němž jsou vybírány nenulové hodnoty primárně z prvního, případně druhého pole), **CROSS** (kartézský součin hodnot dvou polí, tedy vytvoření pole všech kombinací hodnot spojovaných polí), **CROSS_JOIN** (kartézský součin hodnot dvou polí, navíc s podmínkou rovnosti na souřadnice odpovídajících dimenzí), **JOIN ON** (spojení přes zadanou podmínku) a další. A stejně jako SQL, i AQL umožňuje v dotazech využívat vnořené poddotazy, agregace (klauzuli **GROUP BY**), řazení a další konstrukty.

Pro analýzy hodnot polí ovšem nabízí jazyk AQL podstatně širší škálu konstruktů než SQL. Pole je možné rozdělit podle zadané mřížky na podčásti, zpracovávat pomocí specifikovaného posuvného okna apod. Na data v polích je možné aplikovat nejrůznější statistické operátory a funkce, jako např. kvantily (z nichž nejznámější je pravděpodobně medián), odchylky, faktorizaci, normalizaci a mnoho dalších. Hodnoty z polí je možné vybírat náhodně, v daném rozsahu, po aplikaci daného filtru, po odstranění vybraných dimenzí apod. A pokud nám funkcionalita SciDB nestačí, můžeme prostřednictvím C++ doplňovat vlastní uživatelsky definované typy (tj. objekty), funkce, operátory a agregace.

Závěr

Zajímavou a přirozenou otázkou na závěr je budoucnost NoSQL databází. Jak již víme, NoSQL systémy nemají ani nemohou mít za cíl nahradit tradiční relační ani jiné typy databázových systémů. Jedná se o nový přístup určený pro nové typy aplikací, které se objevily především s obrovským rozmachem internetu a mobilních zařízení. Lze očekávat, že aplikací pro Big Data bude přibývat spolu s novými technologiemi, uživateli a zařízeními, které spolu komunikují. I nároky a požadavky na distribuované zpracování velkých dat a jejich ukládání se budou zvyšovat.

Na druhou stranu, podobně jako tomu bylo v případě XML databází, kdy postupně došlo ke sbližení tradičních relačních databází a specifických nativních indexačních metod pro hierarchická, resp. semi-strukturovaná data, můžeme podobný trend sledovat i v případě NoSQL systémů. Jak jsme ukázali, například Hadoop Map-Reduce framework nabízí v nadstavbě Hive jazyk HiveQL, který ne náhodou připomíná jazyk SQL. A tento přístup jsme mohli sledovat i u mnoha dalších systémů různých typů, jako je Apache Pig pro obecné analýzy Big Data, Apache Kylin pro OLAP zpracování Big Data, jazyk Cypher nebo Extended SQL pro dotazování nad grafovými daty a mnohé další. Z tohoto pohledu mají k tradičním relačním databázím zřejmě nejblíže sloupcové NoSQL systémy, kde také můžeme vidět snahu o sbližení s relačními databázemi a jazykem SQL. Systém Cassandra například přešel od pojmu rodina sloupců k pojmu tabulka, vlastnosti sloupcových databází obaluje do tradičních datových typů množina a asociativní pole a pro práci s daty opět nabízí obdobu jazyka SQL – jazyk CQL.

Podíváme-li se na vývoj v této oblasti z opačného konce, v tradičním relačním databázovém systému PostgreSQL nyní můžeme ukládat data ve formátu JSON bez specifikace schématu, což je typická NoSQL vlastnost, a pracovat s nimi přitom prostřednictvím rozšíření jazyka SQL. Již osvědčenou obdobu tohoto přístupu můžeme nalézt třeba v případě relačních databází s XML rozšířením a souvisejícím rozšířením jazyka SQL o část SQL/XML, díky nimž získáme možnost efektivního ukládání a dotazování relačních i XML dat v jediném systému a možnost kombinace jazyka SQL s jazykem XQuery. Podobnou budoucnost tedy můžeme zřejmě očekávat také u různých typů NoSQL databází.

Přestože se zdá, že je v oblasti NoSQL databází nepřeberné množství řešení, pochopitelně stále existují také otevřené problémy, resp. další logické kroky, které budou muset NoSQL databáze ujít. Mezi ty hlavní patří standardizace, optimalizace a rozšíření nabízené funkcionality. Z hlediska technických aspektů se pak jedná především o transakční zpracování, silnější garance zachování konzistence dat a jejich bezpečnost. Společným cílem těchto rozšíření bude nabídnout podobně

zralá a robustní NoSQL řešení, jaká v současné době automaticky využíváme ve světě tradičních relačních databází.

Od začátku textu jsme nijak nezastírali fakt, že pojem NoSQL databáze není úplně vhodný, jelikož jeho význam je poněkud nepřesný. Závěrem knihy si dovolíme dokonce predikovat, že tento pojem časem vymizí. V době vzniku těchto systémů, jež se odvážily jít proti dlouho uznávaným axiomům tradičních databází, byl jistě potřeba pojem, který by tento trend popisoval a jednotlivé snahy stmelil. Jak jsme ale ukázali, NoSQL systémy jsou velmi heterogenní a bylo by logické, aby časem převládla terminologie, která bude významově přesnější – tedy například přímo pojmy dokumentové nebo grafové databáze.

Na existenci nových alternativ si odborníci v oblasti IT pomalu zvykají a učí se využívat jejich předností. Postupem času bude také nejspíš čím dál více živá myšlenka polyglotní persistence, tedy masivního využívání různých typů databázových systémů pro různé typy dat a aplikačních požadavků, a to i v rámci jedné aplikace. Většina systémů a technologií zmíněných v této knize vznikla za přímé účasti nebo podpory velkých internetových hráčů dnešní doby, kteří je také masivně využívají. Pro ty, kteří ženou dnešní vývoj v oblasti IT kupředu, tedy doba polyglotní persistence již dávno nastala. A my doufáme, že po přečtení této knihy bude čtenář lépe připravený a více otevřený použití těch správných technologií na správném místě.

Použitá literatura

- [1] *About PostgreSQL*. The PostgreSQL Global Development Group. <http://www.postgresql.org/about/>
- [2] *Apache HBase™ Reference Guide*. Version 2.0.0-SNAPSHOT. Apache HBase Team, 31. 8. 2015. <http://hbase.apache.org/book.html>
- [3] *Appendix D. SQL Conformance*. PostgreSQL 9.4.4 Documentation. The PostgreSQL Global Development Group. <http://www.postgresql.org/docs/current/static/features.html>
- [4] *BSON Specification Version 1.0*. Creative Commons. <http://bsonspec.org/spec.html>
- [5] *Business Process Model and Notation*. V2.0. Object Management Group. <http://www.bpmn.org>
- [6] *Comparing Database Models*. The Neo4j Manual v2.2.5. Neo Technology, Inc. <http://neo4j.com/docs/stable/tutorial-comparing-models.html>
- [7] *CQL for Cassandra 2.0 & 2.1*. DataStax, Inc. Documentation, 2. 6. 2015 <http://docs.datastax.com/en/cql/3.1/>
- [8] *CQL for Cassandra 2.0 & 2.1 – CREATE INDEX*. DataStax, Inc. Documentation, 25. 8. 2015. http://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_index_r.html
- [9] *Cypher Query Language*. The Neo4j Manual v2.2.3. Neo Technology, Inc. <http://docs.Neo4j.org/chunked/stable/cypher-query-lang.html>
- [10] *DSE Analytics with Hadoop*. 2015. DataStax Enterprise 4.0. http://docs.datastax.com/en/datastax_enterprise/4.0/datastax_enterprise/ana/anaTOC.html
- [11] *ECMA-404: The JSON Data Interchange Format*. ECMA, 2013. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>
- [12] *Efficient XML Interchange (EXI) Format 1.0 (Second Edition)*. W3C, 2014. <http://www.w3.org/TR/exi/>
- [13] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C, 2008. <http://www.w3.org/TR/xml/>
- [14] *Gremlin*. TinkerPop. <https://github.com/tinkerpop/gremlin/wiki>
- [15] *HDFS Architecture Guide*. Hadoop 1.2.1 Documentation, 8. 4. 2013. Apache, Inc. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

- [16] *HTML Living Standard – 5. Microdata*. The Web Hypertext Application Technology Working Group (WHATWG), 5. 5. 2015.
<https://html.spec.whatwg.org/multipage/microdata.html>
- [17] *HTML5*. W3C, 2014. <http://www.w3.org/TR/html5/>
- [18] *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Network Working Group.
<http://tools.ietf.org/html/rfc2616>
- [19] *Indexed Database API*. W3C, 2015. <http://www.w3.org/TR/IndexedDB/>
- [20] *Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*. ISO/IEC 8824-1:2008. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=54012
- [21] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework)*. ISO/IEC 9075-1:2008. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=38641
- [22] *Information technology – Database Languages – SQL – Part 14: XML-Related Specifications (SQL/XML)*. ISO/IEC 9075-14:2011. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=53686
- [23] *Information Technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based Validation – RELAX NG*. ISO/IEC 19757-2:2008. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=52348
- [24] *Information Technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG – Amendment 1: Compact Syntax*. ISO/IEC 19757-2:2003/Amd.1:2006. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40774
- [25] *Information Technology – Generic applications of ASN.1: Fast Infoset*. ISO/IEC 24824-1:2007. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=41327
- [26] *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*. ISO 8879:1986. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=16387
- [27] *JSON-LD 1.0: A JSON-based Serialization for Linked Data*. W3C, 2014.
<http://www.w3.org/TR/json-ld/>
- [28] *MapReduce Tutorial*. Hadoop 1.2.1 Documentation, 8. 4. 2013.
http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
- [29] *Netflix Personalizes Viewing for Over 50 Million Customers with DataStax*. DataStax, 2014. <http://www.datastax.com/wp-content/uploads/2011/09/CS-Netflix.pdf>
- [30] *PGCluster*. FusionForge, 28. 10. 2009. <http://pgfoundry.org/projects/pgcluster>
- [31] *PGGraph*. FusionForge, 12. 9. 2005. <http://pgfoundry.org/projects/pggraph>
- [32] *PostgreSQL 9.4.4 Documentation – 8.14.4. jsonb Indexing*. The PostgreSQL Global Development Group. <http://www.postgresql.org/docs/9.4/static/datatype-json.html#JSON-INDEXING>

- [33] *PostgreSQL 9.4.4 Documentation – 9.15. JSON Functions and Operators*. The PostgreSQL Global Development Group. <http://www.postgresql.org/docs/9.4/static/functions-json.html>
- [34] *Protocol Buffers*. Google Developers, 27. 5. 2015. <https://developers.google.com/protocol-buffers/>
- [35] *RDF 1.1 Concepts and Abstract Syntax*. W3C. 2014. <http://www.w3.org/TR/rdf11-concepts/>
- [36] *RDF 1.1 N-Triples – A line-based Syntax for an RDF Graph*. W3C, 2015. <http://www.w3.org/TR/n-triples/>
- [37] *RDF 1.1 Turtle – Terse RDF Triple Language*. W3C, 2014. <http://www.w3.org/TR/turtle/>
- [38] *RDFa Core 1.1 – Third Edition: Syntax and Processing Rules for Embedding RDF Through Attributes*. W3C, 2015. <http://www.w3.org/TR/rdfa-syntax/>
- [39] *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C, 2007. <http://www.w3.org/TR/soap12/>
- [40] *Spark Streaming Programming Guide*. Spark, v. 1.4.0. <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
- [41] *SPARQL Query Language for RDF*. W3C, 2008. <http://www.w3.org/TR/rdf-sparql-query/>
- [42] *SPARQL Update: A language for updating RDF graphs*. W3C, 2008. <http://www.w3.org/Submission/SPARQL-Update/>
- [43] *SparqlEndpoints*. W3C Wiki, 5. 5. 2015. <http://www.w3.org/wiki/SparqlEndpoints>
- [44] *The Four V's of Big Data*. IBM Big Data & Analytics Hub. 2013. <http://www.ibmbigdatahub.com/infographic/four-vs-big-data>
- [45] *The Power of Cloud – Driving Business Model Innovation*. IBM Global Business Services. Executive Report. IBM Global Business Services, 2012. <https://www.ibm.com/cloud-computing/us/en/assets/power-of-cloud-for-bus-model-innovation.pdf>
- [46] *The Top 20 Valuable Facebook Statistics – Updated February 2015*. Zephoria Inc., 1. 3. 2015. <https://zephoria.com/top-15-valuable-facebook-statistics/>
- [47] *Unified Modeling Language™ (UML®) Resource Page*. Object Management Group, 22. 5. 2015. <http://www.uml.org>
- [48] *Using the NoSQL Capabilities in Postgres*. An EnterpriseDB White Paper. EnterpriseDB Corporation, 2015. <http://enterprisedb.com/wp-using-nosql-features-postgres>
- [49] *Use Cases: Github*. Elasticsearch, 2015. <https://www.elastic.co/use-cases/github>
- [50] *W3C DOM4*. W3C Last Call Working Draft, 18. 6. 2015. <http://www.w3.org/TR/domcore/>
- [51] *Web Services Description Language (WSDL) 1.1*. W3C, 2001. <http://www.w3.org/TR/wsdl>

- [52] *Web SQL Database*. W3C Working Group Note, 2010. <http://www.w3.org/TR/webdatabase/>
- [53] *Web Storage (Second Edition)*. W3C, 2015. <http://www.w3.org/TR/webstorage/>
- [54] *XML Schema Part 2: Datatypes Second Edition*. W3C, 2004. <http://www.w3.org/TR/xmlschema-2/>
- [55] *XSL Transformations (XSLT) Version 3.0*. W3C Last Call Working Draft, 2014. <http://www.w3.org/TR/xslt-30/>
- [56] *XQuery and XPath Full Text 1.0*. W3C, 2011. <http://www.w3.org/TR/xpath-full-text-10/>
- [57] *XQuery Update Facility 1.0*. W3C, 2011. <http://www.w3.org/TR/xquery-update-10/>
- [58] Anderson, J. C. – Lehnardt, J. – Slater, N.: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010.
- [59] Aslett, M.: *What We Talk about When We Talk about NewSQL*. 452 Group, 6. 4. 2011. http://blogs.the451group.com/information_management/2011/04/06/what-we-talk-about-when-we-talk-about-newsql/
- [60] Awad, A. – Sakr, S.: *Querying Graph-Based Repositories of Business Process Models*. Proceedings of the 15th International Conference on Database Systems for Advanced Applications, str. 33-44. Springer-Verlag, 2010.
- [61] Bailey, N. J. T.: *The Mathematical Theory of Epidemics*. Griffen Press, 1957.
- [62] Banon, S.: *Elasticsearch: Big Data, Search and Analytics*. 2012. <https://vimeo.com/44716955> (video, 24:30)
- [63] Bartunov, O.: *Jsonb has Committed!*. LiveJournal, 24. 3. 2014. <http://obartunov.livejournal.com/177247.html>
- [64] Ben-Kiki, O. – Evans, C. – döt Net, I.: *YAML Ain't Markup Language (YAML™) Version 1.2*. 2009. <http://www.yaml.org/spec/1.2/spec.html>
- [65] Bennet, C. – Tseitlin, A.: *Chaos Monkey Released Into The Wild*. The Netflix Tech Blog, 30. 7. 2012. <http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>
- [66] Bernstein, P. A. – Newcomer, V.: *Principles of Transaction Processing*, II. vydání. Morgan Kaufmann Publishers, 2009.
- [67] Beyer, M. A. – Laney, D.: *The Importance of 'Big Data': A Definition*. Gartner, Inc., 2012. <https://www.gartner.com/doc/2057415/importance-big-data-definition>
- [68] Bray, T.: *RFC 7159 – The JavaScript Object Notation (JSON) Data Interchange Format*. IETF, 2014. <http://tools.ietf.org/html/rfc7159>
- [69] Brewer E.: *CAP Twelve Years Later: How the "Rules" Have Changed*. InfoQ & IEEE Computer Society, 30. 5. 2014. <http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>
- [70] Brewer E. A.: *Towards Robust Distributed Systems* (invited talk). Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, str. 7. ACM Press, 2000.

- [71] Buluc, A. – Madduri, K.: *Graph Partitioning for Scalable Distributed Graph Computations*. Proceedings of the 10th DIMACS Implementation Challenge, str. 83-101. American Mathematical Society, 2013.
- [72] Crockford, D.: *The JSON Saga*. Yahoo! HQ, Sunnyvale, CA, USA. 28. 8. 2009. <https://www.youtube.com/watch?v=-C-JoyNuQJs>
- [73] Cuthill, E. – McKee, J.: *Reducing the Bandwidth of Sparse Symmetric Matrices*. Proceedings of the 1969 24th ACM National Conference, str. 157-172. ACM Press, 1969.
- [74] Date, C. J.: *An Introduction to Database Systems*, VIII. vydání. Addison-Wesley Longman, 1999.
- [75] Dean, J. – Ghemawat, S.: *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM, 51(1), str. 107-113. ACM Press, 2008.
- [76] DeCandia, G. – Hastorun, D. – Jampani, M. – Kakulapati, G. – Lakshman, A. – Pilchin, A. – Vogels, W.: *Dynamo: Amazon's Highly Available Key-value Store*. ACM SIGOPS Operating Systems Review, 41(6), str. 205-220. ACM Press, 2007.
- [77] DeMichillie, G.: *Cloud Platform at Google I/O – New Big Data, Mobile and Monitoring Products*. Google Cloud Platform Team, 25. 6. 2014. <http://googledevelopers.blogspot.fr/2014/06/cloud-platform-at-google-io-new-big.html>
- [78] Wilson, G.: *The Eight Fallacies of Distributed Computing*. Fog Creek, 6. 2. 2015. <http://blog.fogcreek.com/eight-fallacies-of-distributed-computing-tech-talk/>
- [79] DeWitt, D. – Stonebraker, M.: *MapReduce: A Major Step Backwards*. A Multi-author Blog on Database Technology and Innovation, 17. 1. 2008.
- [80] Edberg, J. – Tseitlin, A.: *Post-mortem of October 22, 2012 AWS Degradation*. The Netflix Tech Blog, 29. 10. 2012. <http://techblog.netflix.com/2012/10/post-mortem-of-october-222012-aws.html>
- [81] Edlich, S.: *List of NoSQL Databases*. 2015. <http://nosql-database.org>
- [82] Evans, E.: *NoSQL: A Relational Database Management System*. Eric Evans's Weblog, 12. 5. 2009. http://blog.sym-link.com/2009/05/12/nosql_2009.html
- [83] Fidge, C. J.: *Timestamps in Message-passing Systems That Preserve the Partial Ordering*, Proceedings of the 11th Australian Computer Science Conference, str. 56-66. Australian Computer Society, Inc., 1988.
- [84] Fowler, M.: *Event Sourcing*. 12. 12. 2005. <http://martinfowler.com/eaaDev/EventSourcing.html>
- [85] Fowler, M.: *Polyglot Persistence*. 16. 11. 2011. <http://martinfowler.com/bliki/PolyglotPersistence.html>
- [86] Fruchterman, T. M. J. – Reingold, E. M.: *Graph Drawing by Force-directed Placement*. Softw. Pract. Exper., 21(11), str. 1129-1164. John Wiley & Sons, Inc., 1991.
- [87] Galiegue, F. – Zyp, K.: *JSON Schema: Core Definitions and Terminology*. IETF Internet Draft, 2013. <http://tools.ietf.org/html/draft-zyp-json-schema-04>

- [88] Galiegue, F. – Zyp, K.: *JSON Schema: Interactive and Non Interactive Validation*. IETF Internet Draft, 2013. <http://tools.ietf.org/html/draft-fge-json-schema-validation-00>
- [89] Garrett, J. J.: *Ajax: A New Approach to Web Applications*. Adaptive Path, 18. 2. 2005. <http://adaptivepath.org/ideas/ajax-new-approach-web-applications/>
- [90] Giugno, R. – Shasha, D.: *GraphGrep: A Fast and Universal Method for Querying Graphs*. Proceedings of the 16th International Conference on Pattern Recognition 2, str. 112-115. IEEE Computer Society Press, 2002.
- [91] Gormley, C. – Tong, Z.: *Elasticsearch: The Definitive Guide*. Elasticsearch, 2015. <https://www.elastic.co/guide/en/elasticsearch/guide/current/index.html>
- [92] Gormley, C. – Tong, Z.: *Elasticsearch: The Definitive Guide – Query DSL*. Elasticsearch, 2015. <https://www.elastic.co/guide/en/elasticsearch/guide/current/query-dsl-intro.html>
- [93] Gössner, S.: *JSONPath – XPath for JSON*. 21. 2. 2007. <http://goessner.net/articles/JsonPath/>
- [94] Gray, J. – Reuter, A.: *Transaction Processing, Concepts And Techniques*. Morgan Kaufmann Publishers, 1994.
- [95] Gray, J. – Siewiorek, D. P.: *High Availability Computer Systems*. Computer, 24(90), str. 39-48. IEEE Computer Society Press, 1991.
- [96] Harel, D. – Koren, Y.: *A Fast Multi-Scale Method for Drawing Large Graphs*. Journal of Graph Algorithms and Applications, 6(3), str. 179-202, Sprenger-Verlag, 2000.
- [97] Harrison, G.: *10 Things You Should Know about NoSQL Databases*. TechRepublic, 26. 8. 2010. <http://www.techrepublic.com/blog/10-things/10-things-you-should-know-about-nosql-databases/>
- [98] He, H. – Singh, A. K.: *Closure-Tree: An Index Structure for Graph Queries*. Proceedings of the 22nd International Conference on Data Engineering, str. 38-49. IEEE Computer Society Press, 2006.
- [99] He, H. – Singh, A. K.: *Graphs-at-a-time: Query Language and Access Methods for Graph Databases*. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, str. 405-418. ACM Press, 2008.
- [100] Hejmalíček, A.: *Hadoop as an Extension of the Enterprise Data Warehouse*. Diplomová práce, Fakulta informatiky, Masarykova univerzita, 2015.
- [101] Hewitt, E.: *Cassandra: The Definitive Guide*. O'Reilly Media, 2010.
- [102] Chan, E.: *OLAP with Spark and Cassandra*. 2014. <http://velvia.github.io/presentations/cassandra-spark-olap-2014/index.html#/>
- [103] Chang, F. a kol.: *Bigtable: A Distributed Storage System for Structured Data*. ACM Transactions on Computer Systems, 26(2), str. 1-26. ACM Press, 2008.
- [104] Chen, C.: *Information Visualization: Beyond the Horizon*. Springer-Verlag, 2006.
- [105] Chen, C. – Yan, X. – Yu, P. S. – Han, J. – Zhang, D. Q. – Gu, X.: *Towards Graph Containment Search and Indexing*. Proceedings of the 33rd

International Conference on Very Large Data Bases, str. 926-937. VLDB Endowment, 2007.

- [106] Inmon, W. H.: *Building the Data Warehouse*, IV. vydání. Wiley, 2005.
- [107] Izrailevsky, Y. – Tseitlin, A.: *The Netflix Simian Army*. The Netflix Tech Blog, 19. 7. 2011. <http://techblog.netflix.com/2011/07/netflix-simian-army.html>
- [108] Jacob, A.: *Infrastructure as Code*. In: *Web Operations: Keeping the Data On Time*. O'Reilly Media, 2010.
- [109] Jacobs, I. – Forgue, M. C. – Hirakawa, Y.: *W3C Opens Data on the Web with SPARQL*. W3C Press Release, 2008. <http://www.w3.org/2007/12/sparql-pressrelease>
- [110] Jiang, H. – Wang, H. – Yu, P. S. – Zhou, S.: *GString: A Novel Approach for Efficient Search in Graph Databases*. Proceedings of the 23rd International Conference on Data Engineering, str. 566-575. IEEE Computer Society Press, 2007.
- [111] Kalantzis, Ch.: *Revisiting 1 Million Writes per Second*. The Netflix Tech Blog, 25. 7. 2014. <http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html>
- [112] Karger, D. – Lehman, E. – Leighton, T. – Panigrahy, R. – Levine, M. – Lewin, D.: *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. Proceedings of the 29th Annual ACM Symposium on Theory of Computing, str. 654-663. ACM Press, 1997.
- [113] Kimball, R. – Caserta, J.: *The Data Warehouse ETL Toolkit*. Wiley, 2004.
- [114] Kimball, R. – Ross, M.: *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, III. vydání. Wiley, 2013.
- [115] Kimball, R. – Ross, M. – Thorntwaite, W. – Mundy, J. – Becker, B.: *The Data Warehouse Lifecycle Toolkit*, II. vydání. Wiley, 2010.
- [116] Kobourov, S. G.: *Spring Embedders and Force Directed Graph Drawing Algorithms*. Handbook of Graph Drawing and Visualization, str. 383-408. CRC Press, 2013.
- [117] Kolomičenko, V. – Svoboda, M. – Holubová, I.: *Experimental Comparison of Graph Databases*. Proceedings of the International Conference on Information Integration and Web-based Applications & Services, str. 115-124. ACM Press, 2013.
- [118] Kosak, C. – Marks, J. – Shieber, S.: *Automating the Layout of Network Diagrams with Specified Visual Organization*. Transactions on Systems, Man and Cybernetics, 24(3), str. 440-454, IEEE Computer Society Press, 1994.
- [119] Kosek, J.: *PHP a XML*. Grada Publishing, 2009.
- [120] Kosek, J.: *XML schémata*. 10. 11. 2014. <http://www.kosek.cz/xml/schema/>
- [121] Lamport, L.: *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM, 21(7), str. 558-565. ACM Press, 1978.
- [122] Lamport, L.: *The Part-Time Parliament*. ACM Transactions on Computer Systems, 16(2), str. 133-169. ACM Press, 1998.

- [123] Leser, U.: *Query Language for Biological Networks*. Bioinformatics, 2(2), str. ii33–ii39. Oxford Journals, 2005.
- [124] Leinweber, W.: *Embracing the Web with JSON and PLV8*. 2012. <http://plv8-pgopen.herokuapp.com>
- [125] Lima, M.: *Visual Complexity: Mapping Patterns of Information*. Princeton Architectural Press, přetisk, 2006.
- [126] Marz, N.: *Big Data. Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Company, 2015.
- [127] Melton, J. – Buxton, S.: *Querying XML: XQuery, XPath, and SQL/XML in Context*. Morgan Kaufmann, 2006.
- [128] Merkle, R. C.: *A Digital Signature Based on a Conventional Encryption Function*. Advances in Cryptology. Lecture Notes in Computer Science 293, str. 369–378. Springer-Verlag, 1988.
- [129] Minařík, K.: *Kód, v němž se cestičky rozvětvují*. VŠE – přednáška, 10. 3. 2011. <http://www.vse.cz/zprava/12002>
- [130] Mlýnková, I. – Nečaský, M. – Pokorný, J. – Richta, K. – Toman, K. – Toman, V.: *XML technologie: Principy a aplikace v praxi*. Grada, 2008.
- [131] Melton, J. – Van Cappellen, M.: *XQuery API for Java™ (XQJ) 1.0 Specification*. Oracle Corporation, 2009. <http://download.oracle.com/otndocs/jcp/xqj-1.0-fr-oth-JSpec/>
- [132] Normandea, K.: *Beyond Volume, Variety and Velocity is the Issue of Big Data Veracity*. insideBIGDATA, 2013. <http://insidebigdata.com/2013/09/12/beyond-volume-variety-velocity-issue-big-data-veracity/>
- [133] O’Neil, P. E. – Cheng, E. – Gawlick, D. – O’Neil, E.: *The Log-structured Merge-tree (LSM-tree)*. Acta Informatica, 33(4), str. 351–385. Springer-Verlag, 1996.
- [134] Özsu, M. T. – Valduriez, P.: *Principles of Distributed Database Systems*, III. vydání. Springer-Verlag, 2011.
- [135] Parker, D. S., Jr. – Popek, G. J. – Rudisin, G. – Stoughton, A. – Walker, B. J. – Walton, E. – Chow, J. M. – Edwards, D. – Kiser, S. – Kline, C.: *Detection of Mutual Inconsistency in Distributed Systems*. IEEE Transactions on Software Engineering, 9(3), str. 240–247. IEEE Computer Society Press, 1983.
- [136] Philips, M.: *How the Robots Lost: High-Frequency Trading’s Rise and Fall*. Bloomberg, 6. 6. 2013. <http://www.bloomberg.com/bw/articles/2013-06-06/how-the-robots-lost-high-frequency-tradings-rise-and-fall>
- [137] Preguiça, N. – Bauquero, C. – Almeida, P. S. – Fonte, V. – Gonçalves, R.: *Brief Announcement: Efficient Causality Tracking in Distributed Storage Systems with Dotted Version Vectors*. Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing, str. 335–336. ACM Press, 2012.
- [138] Richardson, L. – Amundsen, M. – Ruby S.: *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

- [139] Robie, J. – Chamberlin, D. – Dyck, M. – Snelson, J.: *XQuery 3.0: An XML Query Language*. W3C Recommendation, 2014. <http://www.w3.org/TR/xquery-30/>
- [140] Robie, J. – Fourny, G. – Brantner, M. – Florescu, D. – Westmann, T. – Zaharioudakis, M.: *JSONiq*. 2013. <http://www.jsoniq.org/docs/JSONiq/html-single/index.html>
- [141] Robie, J. – Fourny, G. – Brantner, M. – Florescu, D. – Westmann, T. – Zaharioudakis, M.: *JSONiq Extension to XQuery 1.0*. 2013. <http://www.jsoniq.org/docs/JSONiqExtensionToXQuery/html-single/index.html>
- [142] Rodenbeck, E.: *Communicating Science to the Public*. Data Visualization from Data to Discovery: Art Center + Caltech + JPL symposium, Beckman Auditorium, Caltech, Pasadena, CA, USA, 23. 5. 2013. <https://www.youtube.com/watch?v=qToGZNA4NTQ&t=18m11s> (video)
- [143] Rogers, S.: *Big Data is Scaling BI and Analytics-Data Growth Is about to Accelerate Exponentially – Get Ready*. Information Management, 21(5). Brookfield, 2011.
- [144] Sadalage, P. J. – Fowler, M.: *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.
- [145] Saporito, P.: *2 More Big Data V's – Value And Veracity*. The Digitalist Magazine, 2014. <http://blogs.sap.com/innovation/big-data/2-more-big-data-vs-value-and-veracity-01242817>
- [146] Sheldon, R.: *SQL Server Data Warehouse Cribsheet*. Red Gate Software Ltd, 3. 11. 2008. <https://www.simple-talk.com/sql/learn-sql-server/sql-server-data-warehouse-cribsheet/>
- [147] Skeen, D. – Stonebraker, M.: *A Formal Model of Crash Recovery in a Distributed System*. IEEE Trans. Softw. Eng., 9(3), str. 219-228. IEEE Computer Society Press, 1983.
- [148] Sosinsky, B.: *Cloud Computing Bible*. Wiley, 2011.
- [149] Stonebraker, M. a kol.: *C-Store: A Column-oriented DBMS*. Proceedings of the 31st International Conference on Very Large Data Bases, str. 553-564. VLDB Endowment, 2005.
- [150] Stonebraker, M. – Madden, S. – Abadi, D. J. – Harizopoulos, S. – Hachem, N. – Helland, P.: *The End of an Architectural Era: (It's Time for a Complete Rewrite)*. Proceedings of the 33rd International Conference on Very Large Data Bases, str. 1150-1160. VLDB Endowment, 2007.
- [151] Strickland, R. – Kew, M. – Edelson, H.: *Getting Started with Apache Spark and Cassandra*. Planet Cassandra, 2014. <http://www.planetcassandra.org/getting-started-with-apache-spark-and-cassandra/>
- [152] Strozzi, C.: *NoSQL: A Relational Database Management System*. 1998. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page
- [153] Vogels, W.: *Availability & Consistency*. 7. 8. 2007. <http://www.infoq.com/presentations/availability-consistency> (video, 12:00)
- [154] Walsh, N.: *DocBook 5: The Definitive Guide*. O'Reilly Media, 2010. <http://www.docbook.org/tdg5/>

- [155] White, T.: *Hadoop: The Definitive Guide – Storage and Analysis at Internet Scale*, III. vydání. O'Reilly Media / Yahoo Press, 2012.
- [156] Wood, P. T.: *Query Languages for Graph Databases*. ACM SIGMOD Record, 41(1), str. 50-60. ACM Press, 2012.
- [157] Yan, X. – Yu, P. S. – Han, J.: *Graph Indexing Based on Discriminative Frequent Structure Analysis*. ACM Transactions on Database Systems, 30(4), str. 960-993. ACM Press, 2014.
- [158] Yan, X. – Zhu, F. – Yu, P. S. – Han, J.: *Feature-based Similarity Search in Graph Structures*. ACM Trans. Database Syst., 31(4), str. 1418-1453. ACM Press, 2009.
- [159] Zhang, S. – Hu, M. – Yang, J.: *TreePi: A Novel Graph Indexing Method*. Proceedings of the 23rd International Conference on Data Engineering, str. 966-975. IEEE Computer Society Press, 2007.
- [160] Zhang, S. – Li, J. – Gao, H. – Zou, Z.: *GString: A Novel Approach for Efficient Search in Graph Databases*. Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, str. 204-215. ACM Press, 2009.

Rejstřík

2PC protokol, *viz* protokol, 2PC
3NF, *viz* třetí normální forma
3PC protokol, *viz* protokol, 3PC
3 V, 19
4store, 165

A

ABORT, 207
Accumulo, 127
aCID, 163
ACID, 49, 88, 123, 207
Active Anti-Entropy, 104
adjacency property, 144
AFL, 257
aggregate, *viz* agregace
agregace, 90
AJAX, 33
AllegroGraph, 165
Amazon Dynamo, 98, 113
Amazon EC2, 64
Amazon Relational Database Service, 254
Amazon S3, 64, 176
Amazon Web Services, 189
Ansible, 191
anti-entropy, 104
Apache Cascading, *viz* Cascading
Apache Geode, *viz* Geode
Apache Giraffe, *viz* Giraffe
Apache Hadoop, *viz* Hadoop
Apache Hive, *viz* Hive
Apache Ignite, *viz* Ignite
Apache Kylin, *viz* Kylin
Apache Lucene, *viz* Lucene
Apache Pig, *viz* Pig
Apache Software Foundation, 132
Apache Solr, *viz* Solr
Apache Spark, *viz* Spark
Apache Storm, *viz* Storm

Apache Thrift, *viz* Thrift
AQL, 257
arbitr, 121
architektura shared-nothing, 256
array database, *viz* database, array
Array Functional Language, *viz* AFL
Array Query Language, *viz* AQL
ASN.1, 46
asociativní pole, 95, 106, 129, 134
seřazené, 130
asynchronous repair, *viz* oprava,
asynchronní
atomicita, 49, 124, 141
atomicity, *viz* atomicita
Azure, 189
Azure DocumentDB, 113
Azure SQL Database, 254

B

B+-strom, 118, 141
balancing, 122
BASE model, 54
BaseX, 114
basically available, 54
Berkeley DB, 98–99, 105, 164
Berners-Lee, Tim, 241
Big Data, 19
Bigtable, 105, 128
blind write, 51
blokový řádek, 222
blokový sloupec, 222
Blueprints, 149, 163–164
bod návratu, 207
BPMN, 239
BPMN-Q, 239
Brewer, Eric, 53
Brewerův teorém, *viz* CAP teorém
BSON, 45, 109, 111
bucket, 97

business intelligence, 176, 178
Business Process Model and Notation,
viz BPMN

C

Calliope, 178
CAP teorém, 52, 120
Cascading, 78
Cassandra, 127, 132, 138, 164
Cassandra Query Language, *viz* CQL
CDH, 65
celek, *viz* agregace
cIndex, *viz* indexační metoda, cIndex
Circos, 181
cizí klíč, *viz* klíč, cizí
closure tree, *viz* cTree
cloud
 komunitní, 188
 privátní, 188
 veřejný, 188
cloud computing, 83, 187
clustering columns, 137
Clustrix, 254
column, *viz* sloupec
columnar stores, *viz* databáze,
 sloupcové
column family, *viz* rodina sloupců
column family stores, *viz* databáze,
 sloupcové
column oriented RDBMS, 127
commit, 124
COMMIT, 49, 207
compaction, *viz* konsolidace
conformed dimensions, *viz* sjednocené
 dimenze
consistency, *viz* konzistence
consistent hashing, *viz* konzistentní
 hašování
cookies, 250
CouchDB, 64, 113
CQL, 133, 138
CRUD, 122
CSV, 43
cTree, *viz* indexační metoda, cTree
curl, 96
Cypher, 152

č

časové razítko, 97, 103, 128, 212
Lamportovo, 104

časově vázaná data, 186
čítač, 103

D

D3, 181
dashboards, 176
database
 array, 256
 in-memory, 256
databáze
 dokumentové, 93, 109, 244
 grafové, 93
 netransakční, 223
 transakční, 223
 hybridní, 243
 NewSQL, 254
 NoSQL, 93
 RDF, 164
 relační, 88, 244
 sloupcové, 93, 127, 174
 typu klíč-hodnota, 93, 95
 ve webovém prohlížeči, 250
 XML, 110, 114
databázové schéma, 89
 denormalizované, 92, 113, 173
 normalizované, 89, 112, 173
Data Definition Language, *viz* DDL
Data Manipulation Language, *viz* DML
data mining, *viz* dolování dat
DataNode, 73
DataStax Enterprise, 178
data warehouse, *viz* datový sklad
datový sklad, 80, 172
DBpedia, 165
DDL, 80, 88, 257
deadlock, 158, 211
dělení matice sousednosti
 1D, 221
 2D, 222
dělicí klíč, *viz* klíč, dělicí
dendrogram, 178
dice, 177
dimenze, 172
dimenzionální modelování, 172
dirty read, 50
DISCO, 64
distribuce grafu, 221
DML, 80, 257
DocBook, 35

dokumentové databáze, *viz* databáze,
dokumentové
dolování dat, 176
dostupnost, 53, 120
dotaz
 na nadgraf, 224
 na podgraf, 224
 rozsahový, 118
dotted version vectors, *viz* version
 vectors, dotted
drill-down, 177
driver, *viz* ovladač
durability, *viz* trvalost změn
dvoufázový potvrzovací protokol, *viz*
 protokol, 2PC
Dynamo DB, *viz* Amazon Dynamo DB

E

edge-feature matrix, *viz* matice hran a
 vlastnosti
Ehcache, 99
Elastic MapReduce, 64
Elasticsearch, 58, 65, 100, 113, 121, 125,
 164, 183
Elasticsearch, Logstash, Kibana, *viz* ELK
ELK, 186
ELT, 175–176
EnterpriseDB, 249
Enterprise Resource Planning, *viz* ERP
entitně-vztahový diagram, 89
E-R diagram, *viz* entitně-vztahový
 diagram
ERP, 175
ETL, 82, 175
event sourcing, 50, 54
eventual consistency, *viz* konzistence,
 občasná
EXI, 45
eXistdb, 114
extract, load, transfer, *viz* ELT
extract, transform, load, *viz* ETL
extrakce, 226

F

Facebook, 105, 132
fact table, *viz* faktová tabulka
faktová tabulka, 172
falešně pozitivní kandidát, 226
fantom, 52
FastInfoSet, 45

fault tolerance, *viz* tolerance k výpadku
fáze
 čištění, 175
 COMMIT, 209–210
 doručení, 175
 extrakce, 175
 hlasovací, 209–210
 pre-COMMIT, 210
 sjednocování, 175
feature-graph matrix, *viz* matice grafů
 a jejich vlastnosti
filtrace, 226
filtrační schopnost, 226
force-directed placement, *viz* silou
 řízené rozmístění
formát
 ASN.1, 46
 BSON, *viz* BSON
 CSV, 43
 EXI, 45
 JSON, *viz* JSON
 JSON-LD, 42
 výběr, 46
 XML, *viz* XML
 YAML, 39
Foursquare, 191
funkce
 Combine, 70
 Compare, 70
 Map, 66, 117
 Reduce, 66, 117

G

Gartner, 19
Geode, 82
Gephi, 181
GIndex, *viz* indexační metoda, GIndex
Giraffe, 65
GitHub, 186
Google App Engine, 189
Google Cloud Datastore, 113
Google Chrome, 105
gossip protokol, 100
GPTree, *viz* indexační metoda, GPTree
graf
 atributový, 146
 čekajících transakcí, 212
 datový, 224
 diskriminativní, 227
 dotazový, 224

- izomorfní, 225
 - jednovztahouvý, 145
 - koláčový, 178
 - kontrahovaný, 229
 - kontrastní, 230
 - neorientovaný, 145
 - obloukový, 180
 - ohodnocený, *viz* graf, vícevztahouvý
 - orientovaný, 145
 - radiální, 180
 - sloupcový, 178
 - vážený, 218
 - vícevztahouvý, 145
 - Grafil**, *viz* indexační metoda, Grafil
 - grafové databáze, *viz* databáze, grafové
 - grafový vzor, 241
 - graph**
 - arc, *viz* graf, obloukový
 - labeled, *viz* graf, ohodnocený
 - property, *viz* graf, atributový
 - wait-for, *viz* graf čekajících transakcí
 - GraphGrep**, *viz* indexační metoda, GraphGrep
 - GraphQL**, 235
 - Gremlin**, 149, 163–164
 - GridFS**, 111
 - GString**, *viz* indexační metoda, GString
- ## H
- Hadoop**, 72, 133, 183, 249
 - Hadoop Common**, 72
 - Hadoop MapReduce**, 74
 - Hadoop Streaming**, 74
 - Hadoop YARN**, 72
 - HAProxy**, 121
 - hašovací funkce, 97, 99, 136
 - hašovací prostor, 97
 - hašovací tabulka, 95
 - hašovaná hodnota, 103
 - hašování konzistentní, *viz* konzistentní hašování
 - Hazelcast**, 99
 - HBase**, 65, 127, 132, 137–139, 164, 178
 - HDFS**, 72, 137, 175, 249
 - heartbeat, 74
 - Heroku**, 191
 - HFile**, 137
 - hinted handoffs, 103
 - hit, 184
 - Hive**, 80, 139, 174–175, 177
 - HiveQL, 80
 - HOLAP**, 177
 - horizontal scaling, *viz* škálování, horizontální
 - H-System**, 255
 - HTTP proxy**, 121
 - HTTP REST API**, 96, 106, 114, 121
 - Hunk**, 181
 - hybridní databáze, *viz* databáze, hybridní
 - hybridní OLAP, *viz* HOLAP
 - hypergraf, 145, 219
 - Hypertable**, 127
- ## C
- Chaos Gorilla**, 191
 - Chaos Monkey**, 191
 - check point, *viz* kontrolní bod
 - Chef**, 190
 - chyba
 - efektivní, 206
 - latentní, 206
 - silná, 206
 - slabá, 206
- ## I
- IaaS**, 188
 - Ignite**, 82
 - immutable, 137
 - inconsistency window, *viz* nekonzistentní okno
 - independence, *viz* izolovanost
 - index**
 - binární, 106
 - celočíselný, 106
 - geografický, 119
 - hašovací, 119
 - sekundární, 106, 118, 140
 - textový, 106, 119
 - indexační metoda**
 - cIndex, 230
 - cTree, 233
 - GIndex, 227
 - GPTree, 231
 - Grafil, 232
 - GraphGrep, 228
 - GString, 228
 - pro dotazy na nadgraf, 230
 - pro dotazy na podgraf
 - mining-based, 226

non-mining-based, 228
 pro podobnostní dotazy, 231
 TreePI, 227
 Indexed Database, *viz* IndexedDB
 IndexedDB, 105, 252
 index-free adjacency, *viz* adjacency property
 Infinispan, 99, 105
 InfiniteGraph, 163
 Infrastructure as a Service, *viz* IaaS
 Ingres, 244
 in-memory database, *viz* database, in-memory
 integritní omezení, 49
 interface problem, 180
 intervalové dělení, 136
 inverted index, *viz* invertovaný index
 invertovaný index, 182
 isolation, *viz* izolovanost
 izolovanost, 49

J

jazyk pro definici dat, *viz* DDL
 jazyk pro manipulaci s daty, *viz* DML
 JDBC, 114
 Jena TDB, 165
 jmenný prostor klíčů, 97
 JobClient, 74
 JobConf, 74
 JobTracker, 74
 journal, *viz* operační log
 JSON, 30, 109, 244
 kódování, 31
 objekt, 30, 110
 schéma, 33
 JSON-LD, 42
 JSON Schema, 33
 JustOne DB, 254

K

key-value store, *viz* databáze, typu klíč-hodnota
 Kibana, 65, 186
 klíč
 cizí, 89
 dělící, 136
 primární, 89, 110, 128, 135, 140
 řádku, 128
 složený, 135
 kolekce, 92, 110, 134, 140

konflikt
 read-write, 51, 59, 123
 write-read, 50
 write-write, 51, 59, 103–104, 123
 konsolidace, 138
 kontrastní podgraf, 230
 kontrolní bod, 74
 konzistence, 49, 52, 120
 občasná, 55
 silná, 54
 konzistentní hašování, 100
 koordinátor transakce, 209
 kurzor, 115
 kvórum, 123, 137
 čtení, 60, 102
 zápisu, 60, 102
 Kylin, 178

L

Lamportovy hodiny, 213
 Lamport timestamps, *viz* časové razítko, Lamportovo
 Latency Monkey, 191
 lemmatizace, 183
 LevelDB, 99, 105, 138
 lightweight transactions, *viz* transakce, odlehčené
 Linked Data, 41
 Linked Open Data cloud, 165
 localStorage, 251
 log, 119
 operační, *viz* operační log
 Loggly, 187
 Logstash, 186
 Log-Structured Merge-Tree, 138
 lokalita grafových dat, 220
 lost update, 51
 LSM Tree, *viz* Log-Structured Merge-Tree
 Lucene, 64, 183

M

Mahout, 64
 manažer zdrojů, 208
 Many Eyes, 181
 map, *viz* asociativní pole
 MapDB, 99, 105
 MapReduce, 66, 117, 133, 176
 MapReduce framework, 68
 MarkLogic, 114, 249

master, 58, 68, 123, *viz* uzel, primární
 materialized view, *viz* materializovaný
 pohled
 materializovaný pohled, 244
 matice
 grafů a jejich vlastností, 232
 hran a vlastnosti, 233
 incidence, 219
 Laplaceova, 219
 sousednosti, 218
 mean time to failure, *viz* MTTF
 mean time to repair, *viz* MTTR
 Memcached, 98–99
 MemSQL, 254
 memtable, 137
 Merkle tree, 105
 množina kandidátů, 226
 množina replik, 119, 121
 modulo, 99
 MOLAP, 177
 MonetDB, 127
 MongoDB, 45, 64, 109, 125, 129, 244,
 249
 mongo konzole, 115–116
 MTTF, 206
 MTTR, 206
 multidimenzionální kostka, 176
 multidimenzionální OLAP, *viz* MOLAP
 multigraf, 145, 163
 multi-scale algoritmus, 180
 multiversion concurrency control, *viz*
 MVCC
 MVCC, 164, 215
 MySQL, 125

N

Nagios, 186
 NameNode, 73
 namespace, *viz* jmenný prostor klíčů
 nativní XML databáze, *viz* databáze,
 XML
 nekonzistentní okno, 55
 Neo4j, 146
 Neo4j HA, 156
 Neo4j High Availability, *viz* Neo4j HA
 Netflix, 138, 191
 NewSQL as a Service, 254
 NewSQL databáze, *viz* databáze,
 NewSQL
 Nginx, 121, 187

non-repeatable read, 51
 NoSQL, 23
 NoSQL databáze, *viz* databáze, NoSQL
 Nutch, 64

O

Objectivity/DB, 163
 objektově dokumentové mapování,
 110, 118
 objektově relační mapování, 110
 ODBC, 149
 odkaz, 111
 ODM, *viz* objektově dokumentové
 mapování
 ODM knihovna, 118
 odolnost vůči chybám, *viz* tolerance
 k výpadku
 odolnost vůči rozpadu sítě, 53, 120
 OLAP, 176
 OLAP cube, *viz* multidimenzionální
 kostka
 OLTP, 172
 online analytical processing, *viz* OLAP
 online transaction processing, *viz* OLTP
 Open Database Connectivity, *viz* ODBC
 operační log, 121, 124, 137
 oprava
 asynchronní, 104
 při čtení, 104
 při zápisu, 104
 Oracle, 105
 Oracle Spatial and Graph, 257
 OrientDB, 113, 163
 ORM, *viz* objektově relační mapování
 ovladač, 120–121

P

PaaS, 188
 paměťová cache, 99
 Papertrail, 187
 partition, 136
 partition key, *viz* klíč, dělící
 partition tolerance, *viz* odolnost vůči
 rozpadu sítě
 Paxos, 141
 pg_shard, 249
 Phoenix, 64
 Pig, 77, 139, 175
 Pig Latin, 77
 Platform as a Service, *viz* PaaS

podtransakce, 208
 polyglotní persistence, 26, 125, 255
 PostgreSQL, 113, 125, 132, 244
 postings list, 182
 potvrzení transakce, *viz* COMMIT
 PQL, 238
 prediktivní modelování, 176
 primární klíč, *viz* klíč, primární projekce, 115, 119, 139
 proprietární uzamčení, 48
 propustnost operací, 107
 Protocol Buffers, 44, 95
 protokol
 2PC, 124, 141, 209
 3PC, 210
 proud dat, 21
 Puppet, 191
 pydoop, 65

R

radial chart, *viz* graf, radiaální
 Raphael, 181
 Rasdaman, 257
 RavenDB, 113
 RDBMS, 20
 RDF, 41, 164
 RDF/XML, 41
 RDF databáze, *viz* databáze, RDF
 read committed, 52, 158
 read quorum, *viz* kvórum čtení
 read repair, *viz* oprava, při čtení
 read uncommitted, 52, 124
 read-write konflikt, *viz* konflikt,
 read-write
 realtime stream processing, 82
 Reddit, 191
 Red Hat OpenShift, 189
 Redis, 99, 106, 125, 132, 244
 referenční integrita, 49
 region, 136
 re-indexace, 226
 relační datová kostka, 174
 relační OLAP, *viz* ROLAP
 relační vrstva, 172
 relational database management system,
 viz RDBMS
 relational datacube, *viz* relační datová kostka
 RELAX NG, 37
 repeatable read, 52

replica set, *viz* množina replik
 replikace, 101, 136
 asynchronní, 244
 master-slave, 58, 119, 137
 multi-master, 249
 peer-to-peer, 59, 102, 137
 synchronní, 244
 replikační faktor, 59
 reportování, 176
 Riak, 64, 95–97, 99, 101, 104–105
 RocksDB, 99, 105, 138
 rodina sloupců, 128
 rodina supersloupců, 129
 ROLAP, 177
 rollback, 124
 ROLLBACK, 49, 207, *viz* ABORT
 roll-up, 177
 round-robin, 121
 row, *viz* řádek
 row key, *viz* klíč řádku
 rozdělení dat, *viz* sharding

Ř

řádek, 128

S

SaaS, 187
 sága, 207
 SAP, 127
 SAP IQ, 127
 save point, *viz* bod návratu
 ScaleArc, 254
 scaling out, *viz* škálování, horizontální
 scaling up, *viz* škálování, vertikální
 SciDB, 257
 sekundární index, 106
 selekce, 139
 serializable, 52
 Sesame, 165
 sessionStorage, 251
 seznam sousedů, 218
 sharding, 57, 100, 122, 136
 sharding key, 122
 showflake schema, *viz* schéma typu
 sněhová vločka
 schéma, *viz* databázové schéma
 typu hvězda, 173
 typu sněhová vločka, 174
 silou řízené rozmístění, 179
 síťový diagram, 178

sjednocené dimenze, 175
 skupina sloupců, *viz* rodina sloupců
 slave, *viz* uzel, sekundární
 slice, 177
 sliding window, 82
 sloupcové databáze, *viz* databáze,
 sloupcové
 sloupec, 128
 směrovač, 122
 soft state, 54
 Software as a Service, *viz* SaaS
 Solr, 183
 souběh transakcí, 50
 Spark, 64–65, 82, 178
 Sparksee, 163
 SPARQL, 164, 241
 SPARQL Protocol and RDF Query
 Language, *viz* SPARQL
 SPARQL Update Language, *viz* SPARUL
 SPARUL, 241
 Splunk, 186
 spring embedder, *viz* silou řízené
 rozmístění
 SQL, 138, 178
 SQLite, 254
 SSTable, 137
 Stack Overflow, 113
 stale data, *viz* zastaralá data
 star schema, *viz* schéma, typu hvězda
 stemming, 183
 Stonebraker, Michael, 82, 244, 255, 257
 stopwords, 183
 Storm, 82
 straggler, 71
 stream, *viz* proud dat
 strukturální modelování dat, 179
 super column, *viz* supersloupec
 supersloupec, 129
 Sybase IQ, *viz* SAP IQ

Š

šířka pásmá matice, 220
 škálování
 horizontální, 48
 vertikální, 48
 škálovatelnost, 48

T

Tableau, 181
 tablet, 136

tag cloud, 178
 TaskTracker, 74
 teplotní mapa, 178
 test and set, 141
 three-phase commit protocol, *viz*
 protokol, 3PC
 Thrift, 45, 95
 Thrift API, 133
 throughput, *viz* propustnost operací
 time-based data, *viz* časově vázaná data
 timestamp, *viz* časové razítko
 timestamp ordering
 optimistic, *viz* uspořádání časových
 razítka, optimistické
 pessimistic, *viz* uspořádání časových
 razítka, pesimistické
 TinkerPop, 149
 Titan, 164
 token, 182
 TokuDB, 254
 tolerance k výpadku, 101
 transaction
 chained, *viz* transakce, zřetězená
 multi level, *viz* transakce,
 víceúrovňová
 nested, *viz* transakce, hnízděná
 top level, *viz* transakce, nejvyšší
 úrovně
 transakce, 49, 124
 business, 210
 globální, 208
 hnízděná, 207
 kompenzační, 207
 lokální, 208
 nejvyšší úrovně, 208
 odlehčená, 141
 plochá, 207
 systémová, 210
 víceúrovňová, 208
 vnořená, *viz* transakce, hnízděná
 zřetězená, 207

treemap, 178
 TreePI, *viz* indexační metoda, TreePI
 trvalost změn, 49, 124, 244
 třetí normální forma, 89, 92
 trifázový potvrzovací protokol, *viz*
 protokol, 3PC
 two-phase commit protocol, *viz*
 protokol, 2PC

U

- účastník transakce, 209
- úrovňě izolace transakcí, 52, 124
- uspořádání časových razítek
 - optimistické, 214
 - pesimistické, 213
- uspořadatelnost, 208
- uváznutí, *viz* deadlock
- uzamčení poskytovatelem, *viz*
 - proprietární uzamčení
- uzel
 - primární, 58
 - sekundární, 58

V

- validace transakce, 214
- vector clock, *viz* vektorové hodiny
- vector stamps, *viz* vektorová razítka
- vektorová razítka, 103
- vektorové hodiny, 104
- vendor lock-in, *viz* proprietární
 - uzamčení
- verifikace, 226
- version vectors, 104
 - dotted, 104
- Vertica, 127
- vertical scaling, *viz* škálování, vertikální
- vestavěné diskové úložiště, 99, 105
- virtuální uzly, 101
- Virtuoso, 165
- vnořený dokument, 111, 115
- VoltDB, 254–255
- vyhledávání
 - fulltextové, 65, 106, 114, 182

W

- W3C XML Schema, 38
- wait-die, 212
- WAL, *viz* operační log
- Watson Analytics, 181
- Web SQL Database, 254
- Web Storage, 250
- wide column stores, *viz* databáze,
 - sloupcové
- WordNet, 165
- worker, 68
- wound-wait, 212
- write-ahead log, *viz* operační log
- write availability, 103

- write intensive, 102
- write-once/read-many, 55, 72
- write quorum, *viz* kvórum zápisu
- write-read konflikt, *viz* konflikt, write-read
- write repair, *viz* oprava, při zápisu
- write-write konflikt, *viz* konflikt, write-write

X

- XML, 35, 109
 - kódování, 36
 - schéma, 37
- XML databáze, *viz* databáze, XML
- XQJ, 114
- XQuery, 114
- XQuery Full Text, 114

Y

- YAML, 39

Z

- zablokování
 - koordinátora, 209
 - účastníka, 209
- zámek
 - exkluzivní, 210
 - sdílený, 210
- zamykání
 - off-line optimistické, 211
 - off-line pesimistické, 211
- zápisový log, *viz* operační log
- zastarálá data, 123
- ZooKeeper, 64
- zotavitelnost, 208
- zpozděná kopie, 121
- zpráva
 - PREPARE, 210
- zrušení transakce, *viz* ABORT

noSQěLe. Dočetli jste až sem.

A jestli si navíc

- troufáte na opravdu velká data,
- umíte z nich vydolovat nemožné,
- programujete v Pythonu nebo Javě
- a baví vás paralelní systémy a výpočty,

pište na **kariera@firma.seznam.cz**.

Možná pro vás máme volnou židli.

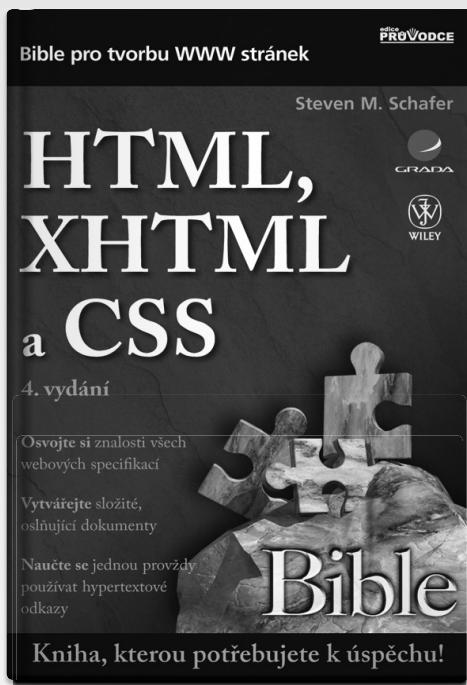
SEZNAM.CZ

Nabídka
publikací

nakladatelství

GRADA

Grada Publishing



HTML, XHTML a CSS *Bible pro tvorbu WWW stránek*

STEVEN M. SCHAFER

Kompletní průvodce světem HTML (HyperText Markup Language) a CSS (Cascading Style Sheets) vás naučí mluvit jazykem webu. Zjistíte, že umění vytvářet sofistikované interaktivní stránky a robustní aplikace je překvapivě nenáročné, navíc se dozvítě spoustu dalších možností, jak udělat vaše stránky interaktivnější a uživatelsky příjemnější. Najdete spoustu užitečných nástrojů, tipů a technik, naučíte se programovat i pro mobilní zařízení s browserem. Kniha obsahuje vše, co potřebujete, aby byly vaše stránky úspěšné.

ISBN 978-80-247-2850-6

648 stran

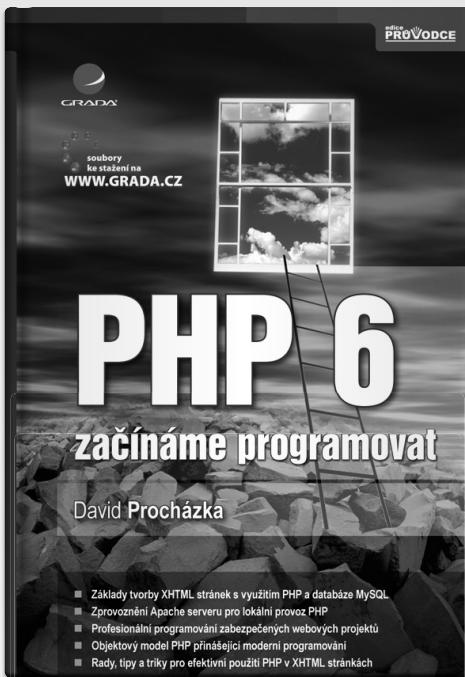
749 Kč

Nabídka
publikací

nakladatelství

GRADA

Grada Publishing



PHP 6 *začínáme programovat*

DAVID PROCHÁZKA

Chcete-li se naučit vytvářet profesionální dynamické webové stránky, pak držte tu správnou knihu – upravené vydání úspěšné učebnice programovacího jazyka PHP, rozšířené o novinky verze 6. Učtenáře se nepředpokládá předchozí znalost PHP, vhodná je ovšem orientace v hypertextovém prostředí a znalost jazyka HTML. Výklad je doplněn komentovanými příklady, na nichž snadno poznáte, jak se v PHP programuje.

ISBN 978-80-247-3899-4

192 stran

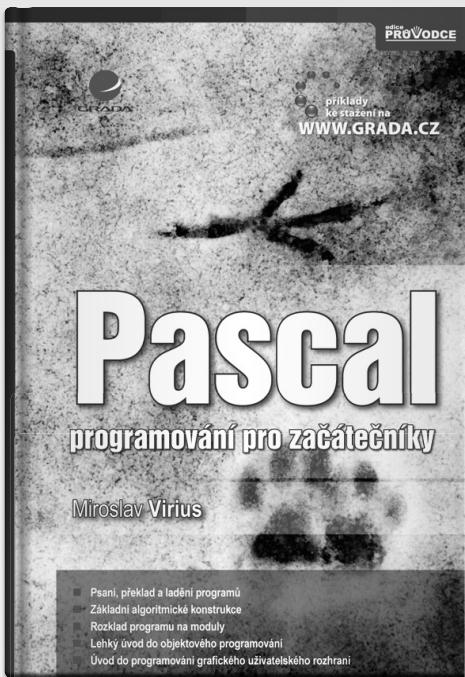
229 Kč

Nabídka
publikací

nakladatelství

GRADA

Grada Publishing



Pascal programování pro začátečníky

MIROSLAV VIRIUS

Chcete-li začít s programováním, máte v ruce správnou knihu. Je určena naprostým začátečníkům, kteří sice umějí pracovat s počítačem, ale nikdy neprogramovali. Najdete v ní kurs programovacího jazyka Pascal, který navrhl prof. N. Wirth speciálně pro výuku programování, ale v němž byla napsána řada běžně používaných aplikací. Na začátku si vysvětlíme několik nezbytných pojmu a pak už napišeme první program. Naučíme se, jak ho přeložit, jak ho spustit a jak ho ladit – tedy jak v něm hledat a odstraňovat chyby. Postupně se seznámíme se všemi základními konstrukcemi jazyka Pascal, jako jsou příkazy, datové typy, podprogramy apod.

ISBN 978-80-247-4116-1

256 stran

149 Kč

Nabídka
publikací

nakladatelství

GRADA

Grada Publishing



Algoritmy v jazyku C a C++

3., aktualizované a rozšířené vydání

JIŘÍ PROKOP

Získáte znalost algoritmů z následujících oblastí: trácení a vyhledávání údajů, teorie a využití grafů, numerické matematiky, dynamického programování a prohledávání textu. Mimo jiné se dozvítě odpovědi na otázky: jaké výhody vám přinese C++, jak vyvážit AVL strom, jak prolomit zašifrovaný text, kdy použít rekurzi a kdy ne. Na stránkách Grady naleznete programy ke stažení, uspořádané podle kapitol knihy. Publikaci mohou využít středoškolští i vysokoškolští studenti technických, ekonomických i přírodovědných směrů, ale i profesionální programátoři.

ISBN 978-80-247-5467-3

200 stran

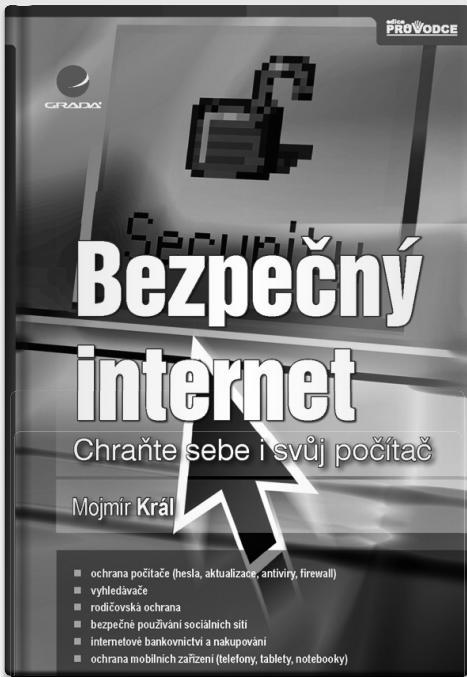
259 Kč

Nabídka
publikací

nakladatelství

GRADA

Grada Publishing



Bezpečný internet

Chraňte sebe i svůj počítač

KRÁL MOJMÍR

Autor podává vyvážený, přehledný a dostatečný návod pro výběr vhodného software, rady pro volbu správné ochrany, a také návod co běžný uživatel potřebuje, čím se má zabývat a věnovat zvýšenou pozornost, příp. jaký typ ochrany je možné pominout. Komplexně upozorňuje na nejrůznější nebezpečí a v postupných krocích radí, jak optimálně využívat ochrany pro různé používání PC. Doporučuje jednotlivé programy pro zabezpečení počítače (jak freeware, tak i nákladnější), jak programy vybírat i jak hodnotit jejich účinnost. Věnuje se především standardní ochraně počítače (hesla, aktualizace, firewall, antiviry), rodičovské ochraně a bezpečnému používání sociálních sítí.

ISBN 978-80-247-5453-6

184 stran

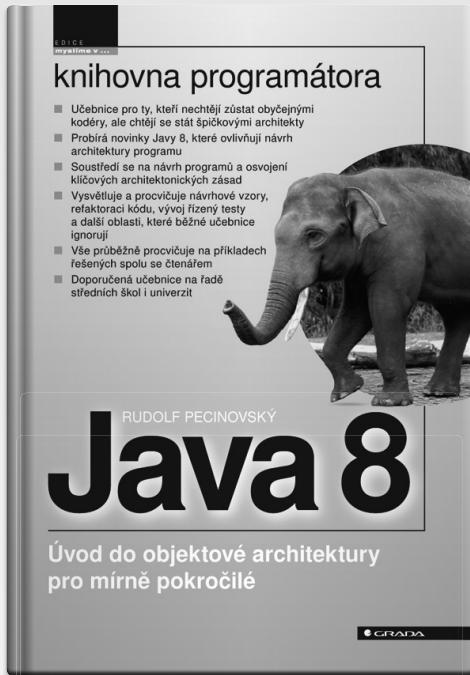
199 Kč

Nabídka
publikací

nakladatelství

GRADA

Grada Publishing



Java 8

Úvod do objektové architektury pro mírně pokročilé

RUDOLF PECINOVSKÝ

Ke studiu publikace nejsou potřebné žádné velké předběžné zkušenosti. Stačí znalosti na úrovni prvního dílu či jiných začátečnických učebnic. Avšak ohlasy na předchozí vydání ukázaly, že i zkušení programátoři s údivem zjišťují, kolik je v ní důležitých informací, které se ve svých dosavadních kurzech nedozvěděli. Výklad je postaven na příkladech, které autor spolu s čtenářem postupně řeší a přitom čtenáře učí nejenom základním programátorským návykům a dovednostem, ale předvede mu i nejrůznější užitečné triky, z nichž mnohé nikde jinde vysvětlené nenajdete. Současně upozorňuje na nejčastější začátečnické chyby, které před svými čtenáři ostatní učebnice většinou tají.

ISBN 978-80-247-4638-8

656 stran

629 Kč



Kniha je určena pro softwarové inženýry a analytiky, databázové programátory a databázové administrátory, kteří se potřebují rychle a příjemně seznámit s oblastí Big Data a NoSQL databází a zjistit, jestli je toto správná cesta pro realizaci jejich aplikací, kterou NoSQL databázi zvolit pro daný problém, nebo naopak kdy rozhodně NoSQL databáze vhodné nejsou. Na českém trhu v současnosti podobná publikace neexistuje.

Big Data se stala v posledních letech jednou z hlavních výzev v oblasti informačních technologií. Data, která vzhledem k rozsahu, různorodosti a rychlosti nárůstu nelze zpracovávat dosud osvědčenými technologiemi, přináší zcela nové problémy a způsoby jejich řešení. Vznikají tak databázové systémy nového typu, které nabízejí odpovědi na otázky, jak tato data efektivně ukládat a dotazovat. Tyto systémy jsou souhrnně označovány jako NoSQL databáze. Publikace seznamuje čtenáře s širokou

škálou pojmu a technologií souvisejících s Big Data a současně poskytuje hlubší výhled do jednotlivých typů NoSQL databázových systémů. U každého typu NoSQL databází jsou uvedeny jak základní principy a používané techniky, tak praktické příklady. Autorský tým zahrnuje odborníky z akademického i komerčního prostředí s vědeckými i praktickými zkušenostmi, jejichž různorodé zkušenosti zaručují široký záběr a objektivní pohled.



Grada Publishing, a.s.
U Průhonu 22, 170 00 Praha 7
tel.: +420 234 264 401
fax: +420 234 264 400
e-mail: obchod@grada.cz
www.grada.cz

