Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Nerelační databáze

Bakalářská práce

Autor: Pavel Bořík
Studijní obor: Informační management

Vedoucí práce: Ing. Barbora Tesařová, Ph.D.

Hradec Králové Duben 2017

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

_____

V Hradci Králové dne 19.4.2017 Pavel Bořík

Poděkování:

Děkuji vedoucí bakalářské práce Ing. Barboře Tesařové, Ph.D. za odborné vedení, rady a čas, který mi věnovala.

Anotace

Cílem této práce je představit databázové systémy pracující na jiném než relačním principu a patřící do skupiny označované jako NoSQL. V teoretické části je nejdříve stručně vysvětlen tradiční relační přístup. Poté jsou již představeny principy, které uplatňují nerelační databáze. Zmíněna je jejich historie, základní druhy a jejich vlastnosti. Teoretická část se věnuje také rozdílné filozofii v přístupu ke škálování a konzistenci. V druhé části práce jsou představeny tři konkrétní nerelační databázové systémy – MongoDB, Cassandra a Neo4j. Pro každý z nich byl popsán případ užití, pro který je vhodné využít nerelační databáze. Následně je předvedena praktická ukázka práce s vybranými systémy.

Klíčová slova:

NoSQL, nerelační databáze, CAP, BASE, MongoDB, Cassandra, Neo4j, dokumentové databáze, sloupcové databáze, grafové databáze

Annotation

Title: Non-relational databases

The main goal of this bachelor thesis is to introduce database systems, that work in a different manner than the relational ones, and that belong to the group referred to as NoSQL databases. The first chapter of the theoretical part briefly explains the traditional relational approach. Then the principles used by the non-relation databases are introduced. Their history, basic types and properties are mentioned. The theoretical part also deals with different philosophy in the approach towards the scalability and consistency. In the second part of the thesis, three specific non-relational database systems - MongoDB, Cassandra and Neo4j - are introduced. For each of them, a specific use case suited for the usage of non-relational database is described. After that, the practical example of working with the chosen systems is shown.

Keywords:

NoSQL, non-relational databases, CAP, BASE, MongoDB, Cassandra, Neo4j, document stores, column-family stores, graph stores

Obsah

1 Úvod

Termín Big Data je v posledních letech skloňován stále častěji. Důvodem je jak počet uživatelů internetu, který se neustále zvyšuje, tak i rozvoj různých dalších IT směrů generujících data v různém množství a formě, jako například internetu věcí. Big Data se samozřejmě objevují i v komerční sféře – firmy jsou nuceny ukládat a analyzovat obrovské množství dat o svých službách a zákaznících a získat tak konkurenční výhodu.

Big Data jsou většinou popisována jako data s obrovským objemem, vysokou rychlostí nárůstu a také různou mírou variability. Ukládaná data tak mohou nabývat nejrůznějších podob – od videí a fotografií, přes záznamy ze sensorů, až po informace generované sociálními sítěmi. Právě tato různorodost a objem dat se staly příčinou nutnosti změny přístupu k systémům, které tato data spravují. Relační databáze, které byly několik desetiletí ústředním prostředkem pro ukládání dat, se pro účel práce s Big Data stávají nedostačujícími. Z tohoto důvodu se již několik let rozvíjí úložiště nerelačního typu, které reagují na limitace relačních databází, hlavně z hlediska potřeby vysoké škálovatelnosti a ukládání různě strukturovaných dat. Nerelační (nazývané také jako NoSQL) databáze ovšem vznikají jako řešení konkrétních problémů a nemají žádný standardizovaný model jak v přístupu k datovému modelu, tak k dotazovacím jazykům. Orientace v množství nerelačních databází je tak poměrně náročná.

Tato práce je určena čtenáři, který měl dosud zkušenost pouze s relačním přístupem a který nemá přehled v různých typech NoSQL databází. Cílem práce je seznámení čtenáře s NoSQL problematikou a ukázka fungování konkrétních databází na ukázkových příkladech. Ty by měly čtenáři poskytnout základní přehled o práci s vybranými databázemi a také poukázat na rozdíly mezi relačním a nerelačním přístupem při tvorbě datového modelu.

2 Teoretická část

2.1 Klasický relační přístup

Pro pochopení problematiky NoSQL je vhodné stručně připomenout problematiku relačních databází. V této kapitole bude nejdříve stručně představen datový model. Dále budou zmíněny normalizační formy, indexování a funkce relačního systému řízení báze dat. Nakonec bude poukázáno na limity týkající se škálování.

2.1.1 Relační datový model

Relační model se poprvé objevil v článku E. F. Codda v roce 1970, který ho vystavěl na matematickém konceptu relačních množin a predikátové logiky. Relace jsou v tomto přístupu chápány jako dvourozměrné **tabulky**, jejichž **řádky** představují jednotlivé záznamy. **Sloupce** potom označují atributy těchto záznamů. Průsečík řádku a sloupce se nazývá pole. Atributy nabývají pouze hodnot, které jsou definovány **doménou**, což je množina přípustných hodnot pro daný atribut [1].

Každý řádek je v tabulce nutno jednoznačně identifikovat. To se provádí pomocí **primárního klíče**, který nabývá unikátní hodnoty v rámci této tabulky buď jako hodnota jediného, nebo složením více sloupců. Obvykle se primární klíč řeší automaticky generovanou číselnou řadou – každý nový záznam dostane unikátní číslo z této řady, která byla inkrementována o jednotku. Pokud se primární klíč jedné tabulky objeví jako atribut v jiné tabulce, je označován jako **cizí klíč** a definuje určitý vztah mezi těmito dvěma tabulkami.

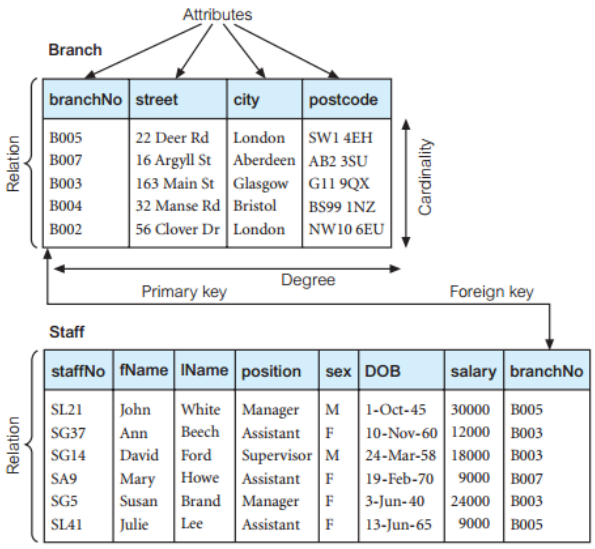
Tento vztah může také nabývat různého typu – označuje se jako **multiplicita**. Tabulky mohou být spojeny vazbou 1:1, 1:N nebo M:N. Nejběžnějším vztahem je vztah 1:N, který říká, že jedna položka tabulky A odpovídá několika položkám tabulky B, zatímco jedna položka tabulky B odpovídá pouze jedné položce v tabulce A. Jako příklad lze uvést vztah, kde jedno nakladatelství vydává mnoho knih, avšak každá kniha má pouze jedno nakladatelství.

Vztah M:N je komplikovanější - jedna položka tabulky A odpovídá několika položkám tabulky B, to samé platí analogicky i z druhé strany pro položky tabulky B. Pro správné zachycení informací je nutné vytvořit třetí tabulku spojenou dvěma

vztahy 1:N, kde je primární klíč složen z cizích klíčů tabulek A a B. Například u vztahu Autoři a Knihy by byla vytvořena tabulka Autoři_Knihy, která by dokázala zachytit jak více knih určitého autora, tak i knihy, které byly napsány více autory.

Konečně, vztah 1:1 je asi nejméně používaný, neboť lze takový vztah většinou vyjádřit v rámci jedné tabulky. Účelem spojení 1:1 tak může být například rozdělení tabulky s mnoha atributy, izolování části tabulky kvůli zabezpečení nebo ukládání dat pro krátkodobé použití. [2]

Ukázka dvou tabulek a jejich vlastností je vyobrazena na obrázku 1.



Obrázek 1 – Tabulky v relačním modelu dat, Zdroj: [1]

2.1.2 Normalizace

Při návrhu tabulek v relačním přístupu je důležité správně zachytit funkční závislosti mezi atributy tak, aby výsledné relace co nejlépe využívaly relačního přístupu a eliminovaly redundantní data. Proces, který se touto problematikou zabývá, se nazývá normalizace. Bez normalizování by se mohla relační databáze střetnout s problémy při vkládání, úpravě i mazání dat. Normalizace je definována pro několik úrovní takzvaných normálních forem. Tři stěžejní úrovně normálních forem jsou definovány následovně [1]:

- 1. NF – každé pole v tabulce obsahuje atomickou (nedělitelnou) hodnotu.

- 2. NF – tabulka splňuje 1. NF a zároveň každý neklíčový atribut je závislý na celém primárním klíči.
- 3. NF – tabulka splňuje předchozí dvě NF a všechny její neklíčové atributy jsou navzájem nezávislé.

Mimo tyto hlavní tři normální formy existuje ještě několik vyšších úrovní – Boyce-Coddova NF, 4. NF a 5. NF.

Ačkoliv je normalizace základním stavebním kamenem návrhu relačního databázového modelu, přináší tento přístup i svoje nevýhody, projevující se zejména na výkonu databáze, zejména při spojování několika tabulek pomocí operace JOIN. Pokud je spojovaných tabulek velké množství, musí být pro přečtení informace vyhledány na disku v jeho různých místech, což má následně negativní vliv na rychlost dotazu.

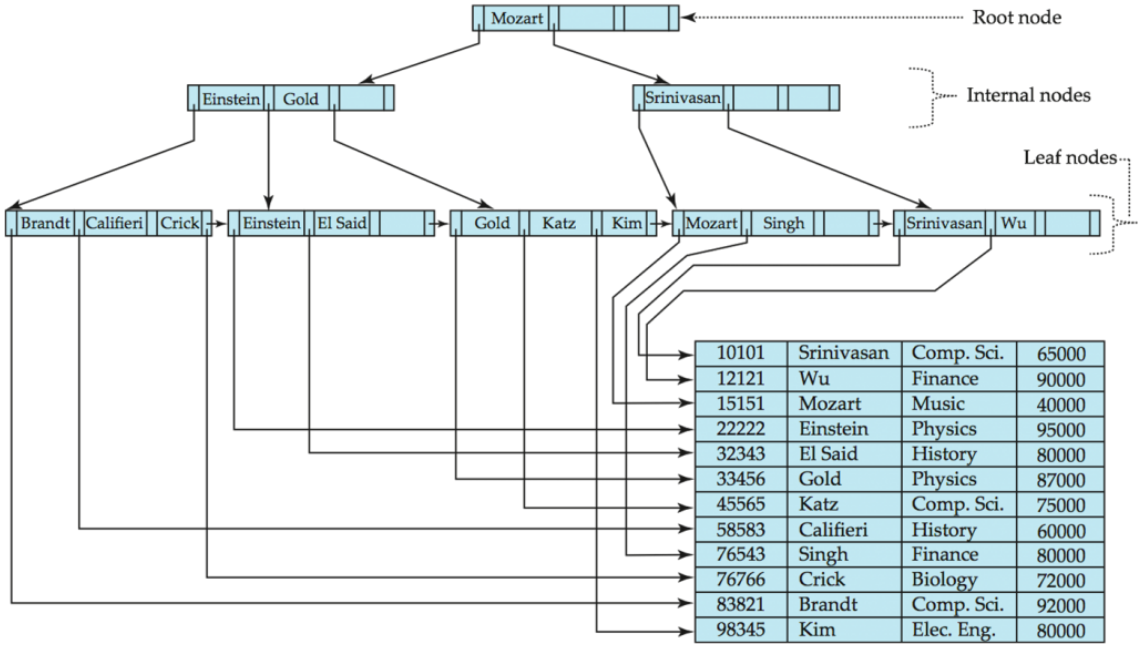
2.1.3 Indexování

Neméně důležitou součástí relačního (i nerelačního) modelu jsou indexy. Index je pomocná datová struktura obsahující hodnoty určitého sloupce tabulky a reference na umístění záznamu v paměti. Použitou datovou strukturou je nejčastěji **B-strom**, který umožňuje snadné setřídění dat a také jejich vkládání, mazání a vyhledávání s logaritmickou časovou složitostí. [3]

Indexování napomáhá zrychlení vyhledávání dotazovaného záznamu – při ukládání do databáze nejsou totiž záznamy bez indexu nijak tříděny, naopak jsou ukládány za sebou tak, jak byly vloženy. Při následném získávání dat z databáze podle určeného kritéria musí být všechny tyto záznamy sekvenčně projity, což velmi zpomaluje celý proces. To se samozřejmě stane velkým problémem, pokud jsou dotazy prováděny nad tabulkami s velkým množstvím uložených dat. [4]

Naopak u indexovaných záznamů, kde jsou hodnoty seřazené v B-stromu, je možno přistoupit přímo na požadované záznamy – například při vyhledávání jmen začínajících na „M“ databáze může přistoupit přímo na záznamy s touto vlastností místo prohledávání všech jmen od „A“ do „Z“. [3]

Princip fungování B-stromu je znázorněn na obrázku 2.



Obrázek 2 - Princip třídění v B-stromu, Zdroj: [3]

Indexy jsou vytvářeny nad jedním i více sloupci, a jedna tabulka jich může obsahovat hned několik. Je ale nutné vybrat jen ty sloupce, kde je využití indexů opodstatněné, jelikož databáze aktualizuje strukturu indexů při vkládání (úpravně, mazání) nových záznamů a je touto nutnou režií naopak zpomalována. Z toho také plyne, že indexy by neměly být vytvářeny nad tabulkami, do kterých se data převážně vkládají. [4]

2.1.4 Relační SŘBD

Relační systém řízení báze dat (angl. zkratka RDBMS) je sada nástrojů sloužící k práci s daty v relačním datovém modelu. Skládá se z několika součástí (převzato z [5]):

- **nástroje pro správu ukládání** – databáze neustále ukládá data, ať už na lokální úložiště, nebo do velkých diskových polí. Databázový systém musí mít přehled, kde jsou jaká data uložena, a případně vyžádaná data načíst. Další úlohou jsou optimalizace rozmístění souborů v úložišti nebo zálohování.

- **nástroje pro správu paměti** – tyto komponenty zodpovídají za správu dat v operační paměti. Jelikož načítání dat z paměti je rychlejší než načítání z disků, efektivní správa paměti má značný vliv na celkový výkon aplikace.
- **databázový slovník –** tato část relačního SŘBD udržuje přehled o struktuře dat v databázi. Jedná se o tabulky, názvy sloupců a jejich datové typy, indexy, integritní omezení nebo pohledy.
- **dotazovací jazyk –** dotazovací jazyk relačních databází se nazývá SQL (angl. zkratka pro Structured Query Language) a plní několik důležitých funkcí. Můžeme jím upravovat strukturu databáze (část SQL anglicky nazvaná Data Definition Language), její obsah (Data Manipulation Language) i přidávat a odebírat přístupová práva (Data Control Language).

2.1.5 Zajišťování konzistence

Konzistence je důležitou vlastností každého databázového systému. Tento pojem podle [5] znamená udržení jediného, logicky spojitého pohledu na data. Konzistentní stav dat znamená, že tato data splňují integritní omezení, tedy podmínky, které jsou na ně kladeny z hlediska dané aplikace (například známka z předmětu musí být v rozsahu A až F, jiné hodnoty nejsou přípustné). Důležitá je také referenční integrita neboli korektnost v propojení dvou tabulek pomocí cizího a primárního klíče [6].

Konzistentní stav dat je velice důležitý například ve finančním sektoru, kde je nezbytně nutné vidět informace (například stav bankovního účtu) v aktuálním a konzistentním stavu. K zajištění konzistence se při změnách dat v databázi využívá transakcí, což jsou *„sekvence logicky souvisejících operací, které převádějí data z jednoho konzistentního stavu do druhého“* [6].

Existují dvě speciální operace, se kterými se můžeme u transakcí setkat – *COMMIT* a *ROLLBACK.* Pokud je transakce úspěšně dokončena, jsou provedené změny operací commit propsány do databáze a jsou také viditelné všem ostatním transakcím. Naopak při neúspěšném proběhnutí jsou operací rollback všechny změny vráceny zpět a do databáze se vůbec nedostanou. [1]

Pro zajištění konzistence existují dva koncepty: ACID, na kterém jsou postaveny relační databáze, a BASE, který převažuje v databázích nerelačních.

2.1.5.1 ACID

ACID je zkratka anglických slov atomicity, consistency, isolation a durability (česky atomicita, konzistence, izolace, trvalost). Jedná se o vlastnosti, na kterých je postaveno transakční zpracování. Jednotlivé vlastnosti jsou popsány takto:

- Atomicity – transakce probíhá jako celek. Všechny dílčí úlohy musí být úspěšně dokončeny, jinak transakce skončí s chybou. [7]
- Consistency – transakce transformuje databázi z jednoho konzistentního stavu do druhého. Transakce buď zapíše nové hodnoty, pokud byla úspěšná, nebo data zůstanou ve stavu před jejím startem. [8]
- Isolation – probíhající transakce se nemohou navzájem ovlivňovat. Databáze ovšem kvůli zachování konkurenčního zpracování podporují různé úrovně izolace, takže je zde možnost, že dotazovaná data mohou být dočasně nesprávná. [5]
- Durability – jakmile je transakce dokončena, data jsou trvale zapsána v databázi. Neměla by se tak ztratit například při výpadku napájení nebo jiné systémové chybě. [7]

2.1.5.2 Transakce v distribuovaném prostředí

V distribuovaném prostředí rozlišujeme dva druhy transakcí – lokální a globální. Lokální transakce probíhá pouze nad daty z lokálního úložiště, v globální (distribuované) transakci participují dva a více uzlů z clusteru.

Aby byly při distribuované transakci naplněny vlastnosti ACID, musí být transakce určitým způsobem řízena. Asi nejznámějším přístupem je potvrzovací protokol nazvaný **dvoufázový commit**, což je algoritmus koordinující všechny procesy, které se na distribuované transakci podílejí. V jeho první fázi řídící uzel zvaný koordinátor zašle všem participujícím uzlům zprávu *PREPARE*. Pokud účastník transakce odpoví pozitivně (tedy jím realizovaná lokální transakce proběhla v pořádku), musí zablokovat objekty, se kterými daná transakce pracovala, a čeká na další příkaz

koordinátora. Ten realizuje druhou fázi 2PC protokolu buď operací *COMMIT* (transakce na všech uzlech proběhla v pořádku), nebo *ABORT* (na některém uzlu došlo k chybě) [6].

Jak je z výše uvedeného postupu zřejmé, zajištění ACID vlastností je v distribuovaném prostředí poměrně složité a náročné na výpočetní zdroje. Pokud tedy účel databáze dovoluje počítat s určitou mírou nekonzistence, vyplatí se uvažovat o konceptu BASE, který je běžný v nerelačních databázích.

2.1.6 Škálování relačních databází

Relační přístup svojí strukturou vyřešil limitace jiných databázových modelů, jako hierarchických nebo síťových, a stal se tak dominantním typem využívaným v databázových aplikacích. S rozvojem webu se ale začaly objevovat faktory, které tento model limitují. Velké objemy dat a obrovský počet uživatelů přistupujících k databázovým aplikacím (zejména u firem jako Google, LinkedIn nebo Amazon) začal omezovat zajištění důležitých atributů, jako udržení high availability, co nejkratší odezvy odpovědí a podpory velkého počtu read/write operací [5].

Relační databáze jsou totiž postaveny na vysoké atomizaci datových záznamů uspořádaných do jednotlivých normalizovaných tabulek. Pokud systém musí následně k sestavení odpovědi na dotaz spojit velké množství těchto tabulek dohromady, mohou nastat výkonnostní problémy.

Možným způsobem zmírnění výše zmíněných problémů je prosté zvýšení výkonu databázového serveru. Zákazník může přistoupit k přikoupení procesorů, další paměti nebo rychlejších úložných zařízení. Tomuto způsobu se říká **vertikální škálování**. Toto ovšem nefunguje donekonečna – servery mají své limity, ať ve velikosti podporované paměti, nebo počtu procesorů, jež lze do nich nainstalovat. Dalším problémem může být proprietární uzamčení, jelikož výkonný serverový hardware je poskytován pouze omezeným množstvím firem, jejichž řešení jsou většinou proprietární. Zákazník tak musí nakupovat nové komponenty u stále stejného výrobce [6].

Opačným řešením je využívat pro provoz systému více serverů – distribuované zpracování. Přidávání zařízení do distribuovaného databázového systému se nazývá **horizontální škálování**. Provozování jednoho databázového systému na více zařízeních je ale v relačním prostředí velmi komplexní operace, jelikož základem je zde transakční zpracování s vlastnostmi ACID a vysokou úrovní izolace. Právě důraz na ACID vlastnosti je jedním z důvodů, díky kterým jsou relační databáze brány jako nepříliš vhodné k horizontálnímu škálování [6].

2.2 Nerelační (NoSQL) přístupy

V této kapitole bude představena historie, základní vlastnosti a typy nerelačních databází. Dále bude popsán jejich přístup k distribuování a škálovatelnosti. Nakonec bude zmíněn framework MapReduce.

2.2.1 Historie nerelačních databází

Termín NoSQL, kterým se nerelační databáze označují, byl pravděpodobně poprvé použit v roce 1998 na konferenci v San Franciscu, kde Ital Carlo Strozzi představoval svoji odlehčenou relační databázi, ke které se nepřistupovalo přes tradiční dotazovací jazyk SQL, nýbrž přes shellové skripty [6].

Po letech vývoje a testování odlišných přístupů se začaly objevovat první nerelační databázové systémy, které známe dnes – v roce 2006 Bigtable společnosti Google a v roce 2007 aplikace Dynamo společnosti Amazon. Mezi lety 2007 a 2009 pak vznikly další: Riak, MongoDB, HBase, Redis, Cassandra nebo Neo4j [8].

Dnes pojem NoSQL zahrnuje stovky nerelačních databázových aplikací, které řeší různé nedostatky databází klasických. Neznamená to ovšem, že relační databáze vymizí, jelikož existuje velké množství aplikací, pro které je relační standardizovaný model vhodný.

2.2.2 Základní vlastnosti

Každý typ nerelační databáze je uzpůsoben ke zpracování odlišných typů dat. Některé klíčové výhody jsou ale pro většinu z nich společné. Podle [6] mezi ně patří následující:

- Flexibilní škálovatelnost – nerelační databáze jsou uzpůsobeny k bezproblémovému horizontálnímu škálování, na rozdíl od relačních databází, které škálují spíše vertikálně. Zpracování úloh mohou distribuovat v rámci clusteru. Velikost clusteru je možno dle potřeby zvětšovat nebo zmenšovat podle aktuální výkonové potřeby.

- Flexibilní datový model – určení databázového schématu většinou není vyžadováno, nebo je schéma dat volné. Jeho změna tak nepředstavuje pro databázi významnou zátěž a jeho správa je často přenášena na aplikační logiku.
- Orientace na efektivní čtení – nerelační databáze využívají takové datové struktury, které co nejlépe odpovídají často pokládaným dotazům. Ty tak mohou být zpracovány velmi rychle, a systém tak zvládne za jednotku času zpracovat jejich velké množství.
- Ekonomický aspekt – horizontální škálovatelnost umožňuje nepřikládat velkou důležitost na hardware v používaných uzlech. Mohou být využity relativně levné počítače, nikoliv drahý serverový stroj. Horizontální škálování je také výhodné z hlediska dostupnosti služeb. Pokud jeden server přestane fungovat, ostatní servery si rozdělí jeho práci a dostupnost není omezena. Dále je většina nerelačních databázových systémů zdarma a s volným přístupem ke kódu (open source). Takto se uživatelé vyhnou komplikovanému výběru licence některých výrobců relačních databázových systémů.

2.2.3 Typy nerelačních databází

Jak již bylo zmíněno, každá nerelační databáze se zaměřuje na ukládání odlišných typů dat. Existují čtyři nejběžnější typy: databáze klíč hodnota, dokumentové databáze, sloupcové databáze a grafové databáze. Nejznámější představitelé jednotlivých druhů jsou uvedeny v tabulce 1.

Tabulka 1 - Představitelé NoSQL databází, Zdroj: vlastní zpracování

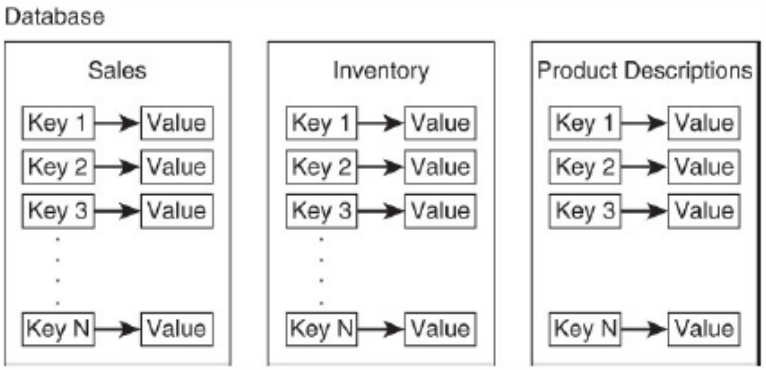
Typ nerelační databáze	Nejznámější představitelé
Databáze klíč-hodnota	Aerospike, Dynamo, Voldemort, Riak
Dokumentové databáze	CouchDB, MongoDB
Sloupcové databáze	HBase, BigTable
Grafové databáze	Neo4j, FlockDB

2.2.3.1 Databáze klíč-hodnota

Úložiště klíč-hodnota je nejjednodušším typem nerelační databáze. Je postaveno na vlastnostech asociativních polí, kde hodnoty nemusí být indexovány pouze pomocí posloupnosti celých čísel, nýbrž lze využít například i textové řetězce. Klíče mohou být umělé nebo automatický generované a slouží k identifikaci požadované hodnoty, která může nabývat různých forem, jako klasický řetězec znaků, obsah ve formátu JSON nebo třeba datový typ BLOB. Klíče musí být unikátní v celém doménovém prostoru, který se anglicky nazývá „bucket“. Takovýchto bucketů může být v databázi více. [5]

Databázové systémy typu klíč-hodnota jsou jednoduché jak strukturou dat, tak i operacemi, které může uživatel využít. Většinou neposkytují žádný nástroj k prohledávání dat podle jejich obsahu, naopak fungují na principu třech základních operací [6]:

- Vložení páru klíč-hodnota do databáze (operace PUT).
- Nalezení hodnoty určitého klíče (operace GET).
- Smazání páru z databáze (operace DELETE).



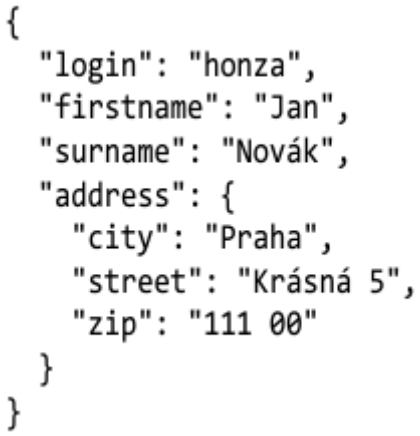
Obrázek 3 - Ukázka tří bucketů v databázi klíč hodnota, Zdroj: [5]

Díky jednoduchosti struktury dat bývají tato úložiště většinou extrémně rychlá a lze je efektivně distribuovat. [6]

Databáze klíč-hodnota mají spíše podpůrné využití – lze je použít například pro kešování dat relačních databází, sledování hodnot webových aplikací nebo ukládání konfigurací a uživatelských informací v mobilních aplikacích.

2.2.3.2 Dokumentové databáze

Tento typ nerelační databáze je podobný předchozímu přístupu klíč-hodnota, avšak jako hodnoty jsou zde ukládány semistrukturované entity, nazvané dokumenty. Tyto dokumenty jsou běžně ve formátu XML nebo JSON (viz. obrázek 4). Databáze využívá těchto formátů k ukládání metadat o jejich obsahu a poskytuje tak možnost provádět dotazy nad dokumenty, na rozdíl od databází klíč-hodnota [9].



Obrázek 4 - Ukázka struktury formátu JSON, Zdroj: [6]

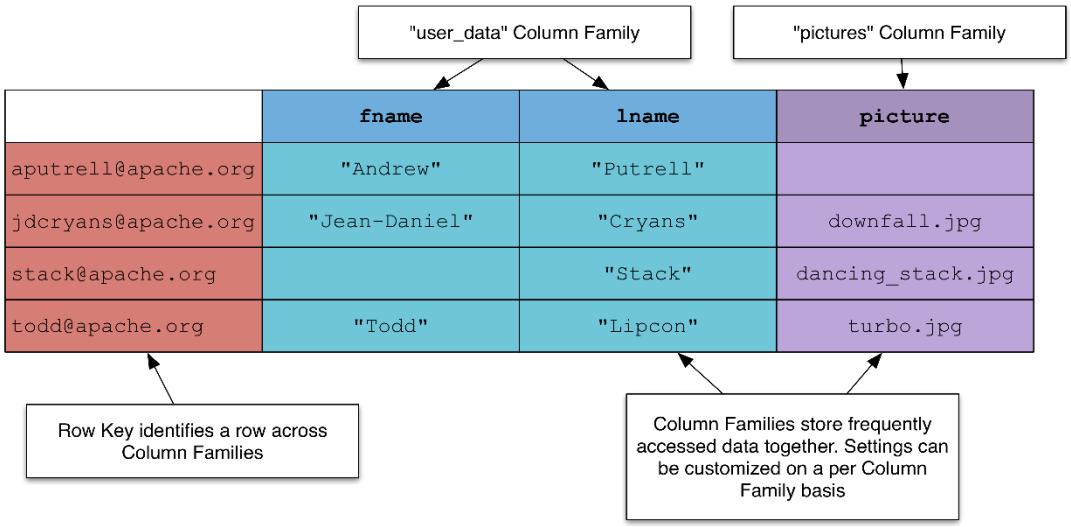
Základní výhodou je absence předem definovaného schématu. Každý uložený dokument může mít jinou strukturu atributů, jejichž obsah může být i jiný zanořený dokument. Nemusíme ukládat žádná pole, která pak zůstanou prázdná, na rozdíl od relačních databází. Dokumenty, které sdílí podobnou strukturu, jsou sdružovány do kolekcí.

Dokumentové databáze mohou být využity k ukládání uživatelsky generovaného obsahu, vytvořeným například na sociálních sítích. Dále jsou vhodné pro katalogová data, například o lidech nebo produktech. Atributy se v těchto případech mohou různě měnit, což není problém pro flexibilní přístup dokumentových databází. Z přístupu ukládání dokumentů ve formátu JSON nebo XML také samozřejmě mohou těžit aplikace, které tyto struktury používají. [10]

2.2.3.3 Sloupcové databáze

Sloupcové databáze (angl. Column-Family Databases) jsou asi nejkomplexnějším typem nerelační databáze. Jejich struktura se podobá klasickému relačnímu modelu – existují zde klasické řádky, které odpovídají záznamům, a sloupce odpovídající atributům. Avšak jsou zde důležité odlišnosti. Sloupcové databáze sdružují společně používané sloupce (například jméno a příjmení) do skupin nazývaných *column families*. Tyto sloupce nebo skupiny sloupců jsou ukládány na disk pospolu a usnadňují vyhledávání při velkém množství dat. V relačních databázích jsou položky naopak ukládány na disk po řádcích [9].

Dalším rozdílem je absence schématu. Sloupce mohou být přidávány podle potřeby. Sloupcová databáze je také uzpůsobena k podpoře řádků s mnoha sloupci (lze ukládat až miliony sloupců). Řádky jsou identifikovány unikátním klíčem řádku (angl. row key). Ukázka sloupcového modelu je na obrázku 5.



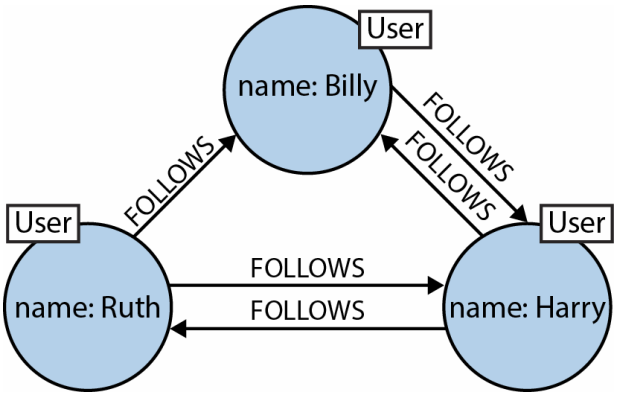
Obrázek 5 - Ukázka obsahu ve sloupcové databázi, Zdroj: [11]

Sloupcové databáze jsou užitečné v prostředí Big Data. Mají dostatečný výkon pro zpracování obrovského množství dat, jsou tak vhodné pro shromažďování dat z akciových trhů nebo sociálních sítí. Například Google pro své potřeby využívá sloupcovou databázi Big Table.

2.2.3.4 Grafové databáze

Grafové databáze jsou zřejmě nevíce specifickou oblastí NoSQL. Namísto ukládání strukturovaných či semistrukturovaných dat jsou využívány k modelování vztahů. Principy vycházejí z teorie grafů, což je jedna z oblastí diskrétní matematiky.

Základním prvkem je zde graf, což je objekt skládající se z vrcholů, které jsou pospojovány hranami. Graf tak může představovat zjednodušenou strukturu z reálného světa, znázorněnou pomocí bodů a čar. Vrcholy představují objekty – může jít například o města, lidi, organismy v ekosystému nebo autobusové zastávky. Hrany znázorňují vztah mezi vrcholy – propojení měst nebo různé vztahy mezi lidmi. Příklad grafu je vyobrazen na obrázku 6. Vrcholy jsou zde uživatelé sociální sítě, hrany představují jejich vztah ve smyslu sledování příspěvků.



Obrázek 6 - Ukázka grafu – sociální síť, Zdroj: [12]

Hlavní výhody grafových databází jsou následující (převzato z [5]):

- Rychlejší dotazování bez spojování tabulek – v relační databázi se pro hledání spojení mezi relacemi používá operace join. Ta hledá hodnotu z jedné tabulky v tabulce jiné. Tato operace je časově náročná, pokud se provádí opakovaně nad tabulkami s velkým množstvím dat. V grafových databázích se propojení zobrazí jednoduše jako hrana, a dotazování probíhá rychleji.
- Zjednodušené modelování - v relačních databázích se ve vztahu M:N musí mezi dvěma entitami vytvořit tabulka navíc, která toto spojení rozdělí na dva vztahy 1:N. V grafových databázích se M:N vztah může vyznačit hranami.

- Více vztahů mezi dvojicí entit – například různé druhy dopravy mezi dvěma městy. To lze vytvořit i v relační databázi, nicméně použití vrcholů a hran tuto problematiku zpřehlední.

Grafové databáze jsou svým přehledným vyznačováním vztahů vhodné pro použití v prostředí sociálních sítí. Dále se hodí pro doporučovací systémy nebo aplikace, které pracují s komplexní množinou propojených prvků, jako biologické, informační a technologické sítě. Možné je také využití jiných druhů nerelačních databází ve spolupráci s grafovou databází. V nich dochází k uložení potřebných dat, nad kterými lze poté provádět operace na grafech [13].

2.2.4 Distribuční modely

Ačkoliv mohou nerelační databáze samozřejmě fungovat na jediném serveru, je pro jejich účel zpracování Big Data žádoucí využít distribuovaného návrhu. Nerelační databáze jsou většinou k běhu v distribuovaném prostředí uzpůsobeny a také, jak již bylo zmíněno, dovolují snadné horizontální škálování. V distribuovaném prostředí existují dvě základní techniky distribuce dat: rozdělení neboli sharding, a replikace.

2.2.4.1 Sharding

Sharding ve své podstatě znamená ukládání různých množin dat jedné databáze na různé servery, které se v tomto kontextu nazývají *shards*. Cílem je zpracovávat požadavek pouze na jednom nebo několika málo uzlech – uživatel přistupující k datům totiž komunikuje pouze se servery, na kterých jsou požadované informace uloženy. Takový přístup by měl rozložit výkon databázového systému a zamezit výkonnostním problémům. [14]

Pro efektivní fungování shardingu je ovšem nutné správně rozložit data na jednotlivé shardy. Zde můžeme vzít v úvahu například fyzické umístění konkrétního shardu a umístit na něj data relevantní pro danou lokalitu. [14]

2.2.4.2 Replikace

Replikace na rozdíl od shardingu ukládá na různých uzlech stejná data. Tímto způsobem je zlepšena dostupnost při výpadku sítě nebo uzlu. Komplikací je nutnost

zajištění konzistence při změně dat, které jsou uloženy na více replikách. Existují dva různé přístupy k replikaci: Master-slave a Peer-to-peer.

V **Master-slave** přístupu figuruje v clusteru jeden primární uzel nazvaný *master* a několik sekundárních uzlů nazývaných *slaves*. Master uzel je jediný, který může provádět zápis, naopak čtení je obslouženo z kteréhokoliv uzlu v clusteru. Z toho vyplívá základní nevýhoda Master-slave přístupu, a to limitace výkonem primárního uzlu při možném zvýšení požadavků na zápis. Proto je také tato strategie vhodná pro data, která jsou především čtena a minimálně modifikována [6].

V přístupu **Peer-to-peer** již neexistuje žádná centralizovaná architektura, naopak jsou si všechny uzly rovny, tedy mohou provádět jak čtení, tak zápis. V tomto přístupu je ovšem nutno ošetřit problematiku nekonzistentních zápisů, kdy se o změnu dat může pokusit více uzlů najednou. [14]

Technika replikování je často kombinována s rozdělením dat. Limituje se tak nevýhoda shardingu, kde při výpadku uzlu nebo sítě může být část dat nedostupná.

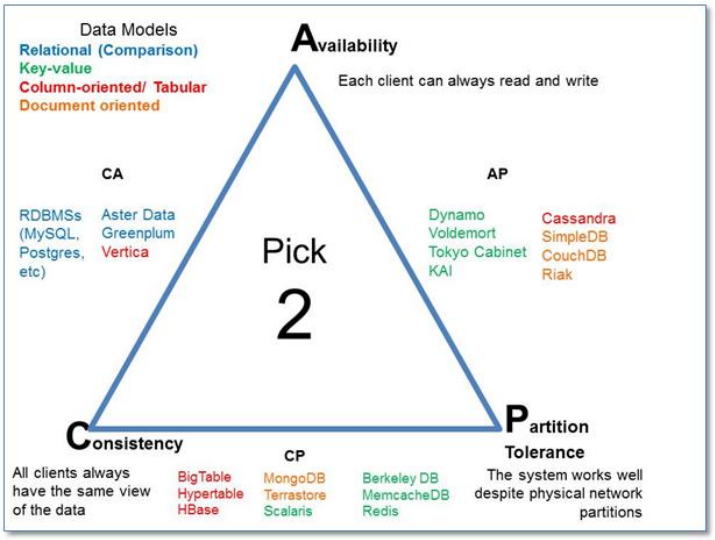
2.2.5 CAP teorém

Myšlenka CAP teorému byla publikována již v roce 2000 Ericem Brewerem [15]. Zkratka CAP označuje tři vlastnosti data managementu v databázích, a to consistency, availability a partition tolerance (česky konzistence, dostupnost a odolnost proti rozdělení). Jednotlivé pojmy v kontextu CAP teorému znamenají následující (podle [16]):

- Consistency – všichni klienti po dotazu na data dostanou stejnou, nejnovější hodnotu.
- Availability – všichni klienti mohou číst a zapisovat data. Jinými slovy, databázový server je schopný odpovědět v rozumné době.
- Partition tolerance – databázový systém nadále funguje, i po výpadku některé jeho části.

CAP teorém říká, že žádný distribuovaný systém nemůže v jednom časovém okamžiku stoprocentně podporovat všechny tři výše uvedené vlastnosti najednou. Toto tvrzení implikuje rozdělení databázových systémů na CP (Consistency +

Partition tolerance), AP (Availability + Partition tolerance) a CA (Consistency + Availability) podle toho, jaké dvě vlastnosti jsou podporovány [15]. Schéma tohoto rozdělení i s jednotlivými databázovými systémy je vyobrazeno na obrázku 7.



Obrázek 7 - Rozdělení podle CAP teorému, Zdroj: [17]

V roce 2012 ovšem samotný Brewer označil výše uvedené rozdělení jako příliš zjednodušené [18]. Databázové systémy nejsou navrhovány černobíle tak, aby splňovaly pouze dvě vlastnosti ze tří. Naopak se snaží (při detekci rozdělení) o maximalizaci konzistence a dostupnosti, která je dosahována kompromisy mezi těmito vlastnostmi. Důraz na tu či onu vlastnost je také většinou možné měnit přímo v nastavení databázového systému.

2.2.6 BASE

V prostředí nerelačních databází se mnohdy uplatňuje přístup, který upřednostňuje vlastnosti jako škálovatelnost, před striktní okamžitou konzistencí a aktuálností dat. Tato metoda uplatňování konzistence se označuje jako BASE, což je zkratka anglických termínů Basically available, Soft-state a Eventually consistent. Přístup BASE je opak k ACID přístupu. Zatímco ACID je označován jako pesimistický a vyžaduje konzistenci na konci každé operace, BASE optimisticky akceptuje neustálé proměny dat a dá se označit jako „občasně konzistentní“.

BASE tedy představuje následující charakteristiky:

- Basically available - distribuovaný systém pokračuje v chodu, ačkoliv se na některém jeho místě objevila chyba. Teoreticky tedy v případě výpadku serveru databáze stále funguje i přesto, že některé dotazy na data nemusí proběhnout úspěšně. [5]
- Soft-state - tento pojem je spjatý s následujícím termínem Eventually consistent. Značí, že data existují ve volném, přechodně nekonzistentním stavu, kvůli jejich aktualizaci mezi servery. [7]
- Eventually consistent - databáze nemusí vždy poskytovat aktuální pohled na data, nicméně do konzistentního stavu se dostane v průběhu času. Tento časový úsek nekonzistence musí být přijatelně dlouhý. Známý systém uplatňující tuto vlastnost je služba DNS. [7]

Shrnutí rozdílů mezi přístupem BASE a ACID je uvedeno v tabulce 2.

Tabulka 2 - Rozdíly mezi přístupy ACID a BASE, Zdroj: [13]

ACID	BASE
Silná konzistence	Občasná konzistence
Důraz na izolaci	Důraz na dostupnost
Využívání funkce *Commit*	Přístup best-effort
Pesimistický	Optimistický
Bezpečný	Rychlý
Vertikální škálování	Horizontální škálování
CA (podle CAP teorému)	AP nebo CP (podle CAP teorému)
Běh DB na jednom serveru	Běh DB v clusteru
Jednoduchý aplikační kód, robustní databáze	Složitý aplikační kód, jednoduchá databáze

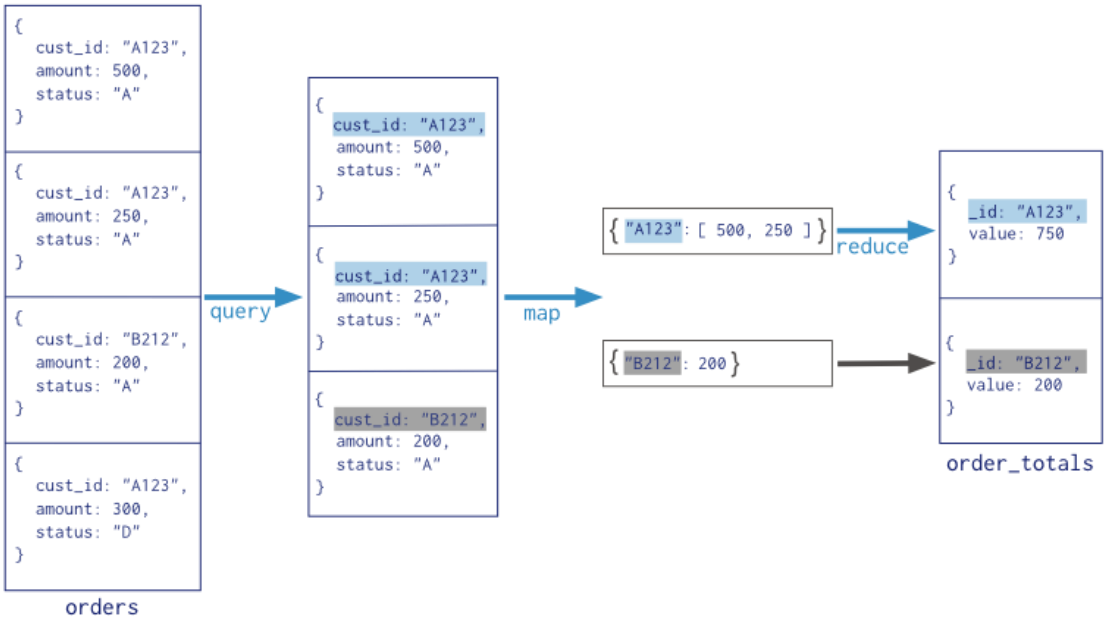
Jak již bylo zmíněno, při výběru databázového systému je nutné stanovit si požadavky na důležitost aktuálnosti a konzistence dat. Například na sociálních sítích není třeba, aby všichni uživatelé viděli změny v databázi, jako například příchozí zprávu, v řádu sekund. Je tak možné vybrat databázi implementující přístup BASE. Naopak v bankovním sektoru je důležité mít aktuální a konzistentní informace, tedy využít databázi s ACID vlastnostmi.

2.2.7 MapReduce framework

MapReduce je programovací model, který se používá v oblasti paralelního distribuovaného zpracování Big Data. Důvodem vzniku byla nutnost přizpůsobit zpracování velkého množství dat prostředí clusteru, kde je sice možné využít výkonu více strojů, avšak kde je stále nutné efektivně minimalizovat množství dat, které musí projít skrz síť, po prvotním zpracování na jednotlivých zařízeních [14].

Myšlenka MapReduce byla představena firmou Google v roce 2004 na konferenci OSDI a od té doby byla implementována do mnoha systémů. Nejznámější implementací je zřejmě systém Hadoop, nicméně princip MapReduce se objevuje i jako součást některých NoSQL databází, konkrétně Riak nebo MongoDB [6].

Základním principem MapReduce je myšlenka „rozděl a panuj“, kterou obstarávají dvě funkce – Map a Reduce. Funkce Map zpracovává objekty z množiny vstupních dat a jejím výstupem je jedna nebo více dvojic klíč hodnota pro každý zpracovaný objekt. Po vytvoření množiny všech hodnot pro každý jednotlivý klíč z mezivýsledku pak funkce Reduce sloučí jednotlivé hodnoty do celkového výsledku [6]. Ukázka práce MapReduce nad daty v databázi MongoDB je znázorněna na obrázku 8.

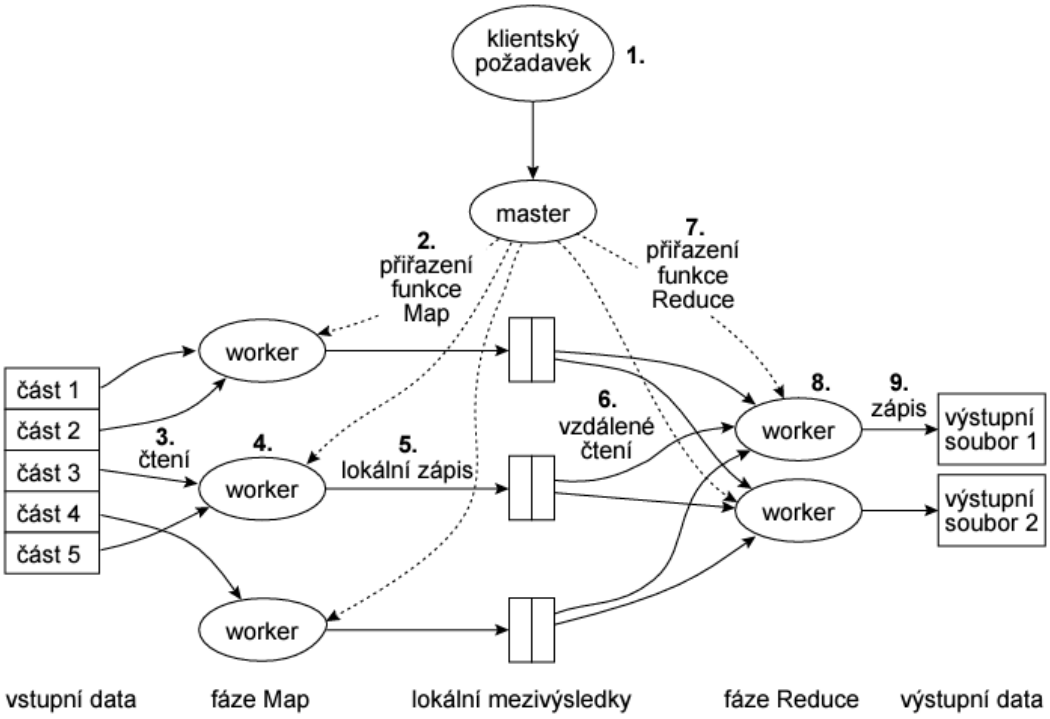


Obrázek 8 - MapReduce v MongoDB, Zdroj: [19]

Samotnou funkcionalitu v clusteru zajišťuje jeden Master uzel řídící celý výpočet a několik uzlů nazývaných Workers (dělníci). Master nejprve přijme požadavek obsahující kód funkce Map, kód funkce Reduce, vstupní data a další parametry. Následně jsou masterem rozděleny zpracovávaná data na M částí a připraven seznam Map úloh, které jsou přidělovány spolu s daty Worker uzlům v clusteru. Ty aplikují Map funkci na přidělených datech a výsledek ukládají na lokální disk. Informace o uložení jsou následně předány zpět Master uzlu [6].

Následuje druhá fáze, kde Master vytvoří R úloh Reduce, které jsou znovu delegovány na Worker uzly. Ty vzdáleně načtou data z míst, která jim byla Masterem přidělena, a začnou provádět Reduce úlohu, která aplikací funkce Reduce vytvoří příslušnou část MapReduce požadavku [6].

Funkcionalita MapReduce v clusteru je přehledně vyobrazena na obrázku 9.



Obrázek 9 - Práce frameworku MapReduce v clusteru, Zdroj: [6]

3 Praktická část

Praktická část se zabývá třemi nerelačními databázovými systémy, a to dokumentovým MongoDB, sloupcovým Cassandra a grafovým Neo4j. Tyto konkrétní databázové systémy byly vybrány zejména proto, že se jedná o nejpopulárnější zástupce z jejich jednotlivých kategorií (podle hodnocení serveru db-engines.com). Cílem praktické části je popsat konkrétní případ užití, ke kterému se hodí právě nerelační databáze. Dále bude předvedena praktická ukázka vybraných databázových systémů, ve které bude popsaný případ užití (ve zjednodušené míře) implementován. Pro práci s databázovými systémy bude využito příkazové řádky.

3.1 Dokumentová databáze MongoDB

MongoDB je dokumentová databáze, jejíž vývoj byl započat již v roce 2007 firmou 10gen (později MongoDB, Inc.) [20]. MongoDB funguje na principech společných pro nerelační databáze – nabízí flexibilní datový model ve formě dokumentů, možnost horizontálního škálování, shardingu nebo agregační funkce MapReduce.

Dokumenty jsou ukládány ve formátu BSON, což je dokument JSON zakódovaný do binárního formátu. BSON rozšiřuje klasický JSON o další datové typy jako Date nebo BinData a poskytuje rychlé kódování a dekódování dokumentů [21].

Tabulka 3 - MongoDB, základní informace, Zdroj: [22]

Vývojář	MongoDB, Inc
První vydání	2009
Implementováno v	C++
Podporované OS	Linux, OS X, Solaris, Windows
Podpora programovacích jazyků	26 jazyků
Způsob rozdělení dat	Sharding
Způsob replikace	Master-slave
Podpora MapReduce	Ano
Podpora ACID transakcí	Ne

3.1.1 Verze a licence

MongoDB Server je šířen ve dvou verzích – Community a Enterprise. Community verze spadá pod licenci AGPL v3.0 a je dostupná zdarma. Enterprise verze zdarma není – je součástí předplaceného programu MongoDB Enterprise Advanced a funguje pod komerční licencí. Placená verze nabízí různá vylepšení zabezpečení dat nebo zpracování dat v paměti. [23]

3.1.2 Srovnání terminologie SQL a MongoDB

Struktura uložení dat v MongoDB je odlišná od relačních databází a liší se také terminologie této problematiky. V tabulce 4 jsou uvedeny termíny tak, jak jsou používány v relačním přístupu, s jejich ekvivalentem v MongoDB.

Tabulka 4 - Terminologie SQL vs. MongoDB, Zdroj: [24]

SQL koncepty	MongoDB koncepty
Tabulka	Kolekce
Řádek	Dokument
Sloupec	Pole
spojování tabulek pomocí JOIN operací	Vnořené dokumenty, příkaz $lookup

Zatímco v relačních databázích jsou data uložena v tabulkách s pevným schématem, v MongoDB jsou data (dokumenty) sdružována v kolekcích bez pevně zadané struktury. Dokumenty lze přirovnat k řádkům, pole v dokumentech (mající tvar klíč: hodnota) ke sloupcům.

3.1.3 Využití MongoDB – katalog produktů

Jedním vhodným případem užití databáze MongoDB je realizace katalogu produktů elektronického obchodu. Pokud takový obchod nabízí mnoho různých druhů zboží s odlišnými atributy, může být návrh v relačním modelu složitý nebo neefektivní.

3.1.3.1 Nevýhody relačního přístupu

V relačním přístupu existuje několik řešení problému, mají však své nevýhody:

- Nová tabulka pro každou kategorii produktů – při velkém množství kategorií vzniká neúnosné množství nových tabulek se specifickými atributy.
- Jedna tabulka pro všechny produkty – je nutno přidat nové sloupce pro specifické atributy a modifikovat tak strukturu celé tabulky při každém vkládání nové kategorie produktů. Po zařazení nového produktu do tabulky také zůstává většina definovaných atributů vyplněná hodnotou NULL.
- Spojení více tabulek – je možné vytvořit obecnou tabulku produkt a dále konkretizovat jednotlivé atributy v dílčích tabulkách spojených s obecnější tabulkou cizím klíčem. Tento přístup nabízí určitou flexibilitu a šetří místo, dotazování ovšem může být zpomaleno nutností použít spojení tabulek.

3.1.3.2 Produkty v MongoDB

V MongoDB jsou produkty uloženy ve schématu, které se neřídí žádným pevným formátem. Dokumenty tak obsahují jen atributy vztahující se ke konkrétnímu produktu, jak ukazují výpisy 1 a 2. Z důvodu přehlednosti příkladu nejsou u produktů uvedeny všechny možné atributy.

Výpis 1 - Produkt grafická karta v MongoDB

```
{
    "_id" : 1,
    "nazev" : "MSI Radeon RX 480 8GB",
    "kategorie" : "Graficke_karty",
    "cena" : 7500,
    "technicke_parametry" : {
        "vyrobce" : "MSI",
        "graficky_cip" : "Radeon RX 480",
        "rychlost_grafickeho_cipu" : "1266 Mhz",
        "velikost_graficke_pameti" : "8192 MB",
        "rychlost_graficke_pameti" : "8000 Mhz"
    }
}
```

Výpis 2 - Produkt mobilní telefon v MongoDB

```
{
    "_id" : 2,
    "nazev" : "Apple iPhone 7 Plus, 256GB",
    "kategorie" : "Mobilni_telefony",
    "cena" : 30000,
    "technicke_parametry" : {
        "operacni_system" : "Apple iOS",
        "velikost_displeje" : 5.5,
        "rozliseni_displeje" : "1920 x 1080",
        "interni pamet" : "256 GB"
    }
}
```

Ke snadnému prohledávání celého katalogu obsahují produkty nejdříve pole společná pro všechny produkty, zde tedy *nazev, kategorie* a *cena*. Specifické atributy jsou řešeny vnořeným dokumentem *technicke_parametry*.

3.1.4 Ukázka

Praktická ukázka práce s databází naváže na výše zmíněný případ užití. Využita bude MongoDB verze 3.4.1 a příkazy budou zadávány v databázovém rozhraní ovládací konzole *mongo shell*.

Syntax jazyka pro ovládání databáze vychází z konvence formátu JSON. Před spuštěním shellu je také nutné spustit databázový démon *mongod*.

3.1.4.1 Vytvoření požadovaných struktur a manipulace s daty

Jako první je třeba přepnout se do požadované databáze (nazvanou *ProduktyDB*) příkazem *use*. Ten požadovanou databázi vytvoří, pokud neexistuje.

Výpis 3 - Výběr databáze v MongoDB

```
> use ProduktyDB
switched to db ProduktyDB
```

Poté už lze rovnou zadávat produkty v notaci JSON bez definice jakéhokoliv schématu příkazem *insert()*. V mongo shellu jsou příkazy zadávány ve formátu, kde je specifikován i název kolekce – v tomto případě kolekce *produkty*. U jednotlivých dokumentů lze specifikovat jejich jedinečný identifikátor *_id,* nutné to ale není – MongoDB v případě nutnosti vygeneruje vlastní jedinečný hexadecimální řetězec.

Výpis 4 - Vložení produktů v MongoDB

```
> db.produkty.insert({ "_id" : 1, "nazev" : "MSI Radeon RX 480", "kategorie" : "Graficke_karty",
"cena" : 7500, "technicke parametry" : { "vyrobce" : "MSI", "graficky_cip" : "Radeon RX 480",
"rychlost_grafickeho_cipu" : "1266 Mhz", "velikost_graficke_pameti" : "8192 MB",
"rychlost_graficke_pameti" : "8000 Mhz" } })

> db.produkty.insert({"_id":2,"nazev":"Apple iPhone 7 Plus,
256GB","kategorie":"Mobilni_telefony","cena":30000,"technicke_parametry":{"operacni_syst
em":"Apple iOS","velikost_displeje":5.5,"rozliseni_displeje":"1920 x 1080","interni
pamet":"256 GB"}})
```

Úprava vložených dokumentů je prováděna příkazem *update()*. V prvním argumentu tohoto příkazu je zadána podmínka, která identifikuje dokument (nebo více dokumentů), kde je požadována úprava hodnot. Ta je prováděna příkazem $set, kde je specifikován název pole a nová hodnota. Příkaz update() v základním tvaru změní pouze jeden dokument. Pokud je nutné změnit všechny dokumenty vyhovující podmínce, použije se třetí argument *multi,* jehož hodnota je nastavena na 1.

Výpis 5 - Úprava dokumentů v MongoDB

```
> db.produkty.update( {'nazev': 'MSI Radeon RX 480 8GB'}, { $set: {'cena': 7700}})

//úprava více dokumentů, které vyhovují podmínce v prvním argumentu
> db.produkty.update( {'nazev': 'MSI Radeon RX 480 8GB'}, { $set: {'cena': 7700}}, {multi: 1})
```

3.1.4.2 Dotazování

Pro katalog produktů jsou důležité zejména prohledávací operace. Uživatelé mohou využívat mnoho různých filtrů nebo řazení. Prohledávání v MongoDB je prováděno příkazem *find()*, ve kterém lze specifikovat parametry vyhledávání. Při použití pouze příkazu *find()* bez parametrů jsou vráceny všechny záznamy dané kolekce. V příkladu ve výpisu 6 je uveden příkaz, který nalezne všechny grafické karty od výrobců MSI nebo GIGABYTE a seřadí je vzestupně podle ceny. Protože parametr *vyrobce* je vnořen v dokumentu *technicke_parametry*, přistupuje se k němu přes tečku. Pokud by bylo požadováno řazení podle ceny sestupně, byl by ve funkci *sort()* použit parametr -1.

Výpis 6 - Vyhledávání produktů v MongoDB

```
> db.produkty.find({'kategorie': 'Graficke_karty', 'technicke_parametry.vyrobce': { $in: ["MSI",
"GIGABYTE"] } }).sort({cena: 1})
```

3.1.4.3 MapReduce

Princip MapReduce byl vysvětlen v teoretické části, je tedy vhodné ho demonstrovat i v praktické ukázce. Pro dostatek dat bude nutné vložit dva další dokumenty – mobilní telefon a grafickou kartu.

Výpis 7 - Vkládání dalších produktů v MongoDB

```
> db.produkty.insert({"nazev":"Samsung Galaxy S7 - 32GB, černá","kategorie":"Mob
ilni_telefony","cena":17790,"technicke_parametry":{"operacni_system":"Google And
roid","velikost_displeje":5.1,"rozliseni_displeje":"2560 x 1440","interni pamet"
:"32 GB"}})
> db.produkty.insert({"nazev":"GIGABYTE GeForce GTX 1060 Xtreme Gaming 6G","kate
gorie":"Graficke_karty","cena":8993,"technicke_parametry":{"vyrobce":"GIGABYTE","
graficky_cip":"GeForce GTX 1060","rychlost_grafickeho_cipu":"1620 Mhz","velikost
_graficke_pameti":"6144 MB","rychlost_graficke_pameti":" 8164 Mhz"}})
```

Samotným cílem dotazu bude zjištění průměrné ceny produktů v jednotlivých kategoriích. K dosažení cíle bude použita funkce mapReduce(), která je vyobrazena na výpisu 8.

Výpis 8 - Použití příkazu MapReduce v MongoDB

```
> db.produkty.mapReduce(
... function() {emit(this.kategorie, this.cena);},
... function(key, values) {return Array.avg(values);},
... {out: "produktyMapReduce"})
{
      "result" : "produktyMapReduce",
      "timeMillis" : 108,
      "counts" : {
            "input" : 4,
            "emit" : 4,
            "reduce" : 2,
            "output" : 2
      },
      "ok" : 1
}
```

První funkcí je zde funkce Map, spuštěná na každém dokumentu, který patří do kolekce produkty. Tato funkce vytváří dvojice klíč: hodnota, kde klíčem je kategorie produktu a hodnotou cena produktu. Druhá funkce Reduce hodnoty pro každý klíč zprůměruje a uloží do nové kolekce *produktyMapReduce.* V příkazové řádce dostaneme po spuštění příkazu i statistiky provedených operací.

Obsah výsledné tabulky můžeme zkontrolovat již představeným příkazem *find().*

Výpis 9 - Výsledek po provedení příkazu MapReduce v MongoDB

```
> db.produktyMapReduce.find()
{ "_id" : "Graficke_karty", "value" : 8346.5 }
{ "_id" : "Mobilni_telefony", "value" : 23895 }
```

3.2 Sloupcová databáze Cassandra

Cassandra je nerelační sloupcová databáze vyvíjená společností Facebook, která ji v roce 2008 vypustila do světa jako open-source aplikaci. Nyní je Cassandra vyvíjena v rámci iniciativy Apache Software Foundation. Databázový systém nabízí pro práci se strukturou dat i dotazy jazyk nazvaný Cassandra Query Language, zkráceně CQL. Syntax tohoto jazyka je velmi podobná klasické SQL notaci.

Architektura Cassandry je koncipována jako peer-to-peer kruh, tedy všechna propojená zařízení mají stejnou roli – neexistuje zde vztah master-slave. Díky vlastnostem takovéhoto řešení neexistuje bod, který by při výpadku vyřadil celý systém, a je tak snadné databázový cluster rozšiřovat nebo zmenšovat bez toho, aby byl celý systém výrazně omezen nebo dokonce nefunkční. [16]

Cassandra dovoluje uživateli nastavit různý důraz na konzistenci nebo dostupnost. Existuje několik úrovní jak pro čtení, tak i zápis.

Cassandra dále ve spojení s frameworkem Apache Hadoop nabízí vyhodnocování agregačních dotazů pomocí přístupu MapReduce [6].

Tabulka 5 - Cassandra, základní informace, Zdroj: [25]

Vývojář	Apache Software Foundation
První vydání	2008
Implementováno v	Java
Podporované OS	Linux, OS X, BSD, Windows
Podpora programovacích jazyků	13 jazyků
Způsob rozdělení dat	Sharding
Způsob replikace	Peer-to-peer
Podpora MapReduce	Ano

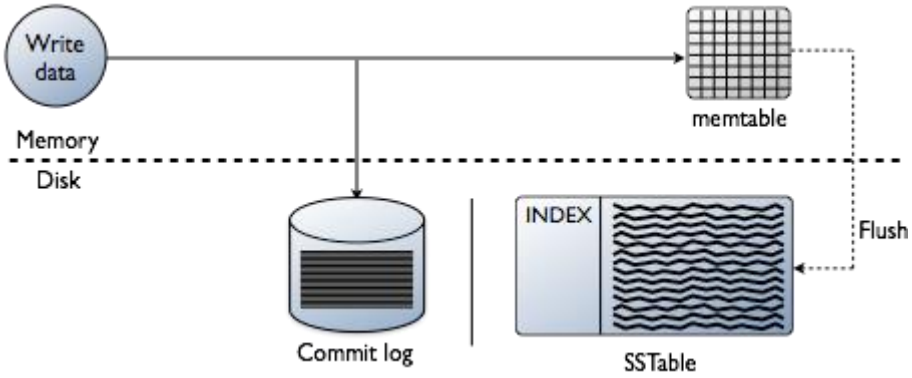
Podpora ACID transakcí	Ne

3.2.1 Verze a licence

Cassandra sama o sobě je vyvíjena jako open-source pod licencí Apache License 2.0, je ale také úzce spojena se společností DataStax. DataStax je hlavním přispěvatelem ve vývoji Cassandry a poskytuje edice s různými programovými nadstavbami pro soukromé i komerční použití.

3.2.2 Cassandra a organizace dat

Lokální uložiště se musí starat jak o persistentní uložení dat, tak i o maximalizaci propustnosti operací zápisu a čtení. V Cassandře je lokální úložiště složeno ze základních komponent nazvaných paměťová tabulka (memtable), SSTable (z anglických slov sorted string table) a zápisový log. Posloupnost akcí pro zápis dat je vyobrazena na obrázku 10.



Obrázek 10 - Zápis dat v databázi Cassandra, Zdroj: [26]

Data jsou nejprve zapsána do zápisového logu, kde se zapisuje pouze na jeho konec, proto je zápis velmi rychlý. Poté jsou data zapsány do paměťové tabulky. Když objem dat v paměťové tabulce dosáhne kritického bodu, dojde ke konsolidaci neboli zápisu na disk do struktury SSTable [16]. Při této akci se data přeskupí tak, aby bylo čtení prováděno co nejefektivněji. Při konsolidaci jsou také fyzicky odstraněny záznamy, jejichž smazání bylo promítnuto zatím pouze v paměti.

Celý tento princip fungování je popsán jako struktura nazvaná Log-Structured Merge-Tree (zkráceně LSM tree). Hlavním benefitem je rychlost zápisu, která je

vyšší než u relačních databází využívající principu B-stromu. Proto jsou databáze implementující LSM přístup vhodné pro data, která jsou zapisována často a ve velkém objemu. [27]

3.2.3 Využití Cassandry – časové řady

Datový model databáze Cassandra je vhodný pro ukládání časových řad, což jsou data generovaná v pravidelných intervalech. Data mohou pocházet například ze senzorů nebo finančních trhů a poskytovat podklad pro následnou analýzu nebo předpovědi. Je zřejmé, že použitá databáze si bude muset poradit s velkým množstvím dat, jelikož požadavek na zápis může nastat i několikrát za vteřinu. Cassandra je pro rychlé zápisy vhodné kvůli implementaci výše popsaného LSM tree přístupu.

Řešením ukládání časových řad ve sloupcové databázi bude využití „širokých řádků“. Cassandra je pro tento způsob ukládání navržena a dovoluje uložení až dvou miliard sloupců na jednom databázovém oddílu [28]. Jako příklad bude sloužit zaznamenávání hodnot akcií na burze v čase. Základní návrh schématu je zachycen v následující tabulce.

Tabulka 6 - Schéma zaznamenání časových řad, Zdroj: vlastní zpracování

$Akcie\_ID_1$	$Časové\ razítko_1$	$Časové\ razítko_2$	…	$Časové\ razítko_m$
	$Hodnota_1$	$Hodnota_2$	…	$Hodnota_m$
$Akcie\_ID_2$	$Časové\ razítko_1$	$Časové\ razítko_2$	…	$Časové\ razítko_n$
	$Hodnota_1$	$Hodnota_2$	…	$Hodnota_n$
$Akcie\_ID_x$	$Časové\ razítko_1$	$Časové\ razítko_2$	…	$Časové\ razítko_o$
	$Hodnota_1$	$Hodnota_2$	…	$Hodnota_o$

Identifikátorem řádku je zde Akcie_ID, data jsou poté dynamicky zachytávána do nových sloupců.

3.2.4 Ukázka

V praktické ukázce práce s databázovým systémem Cassandra ve verzi 3.9.0 bude předvedena implementace příkladu se zaznamenáváním hodnot akcií v čase.

3.2.4.1 Vytvoření požadovaných struktur a manipulace s daty

Nejprve je nutné vytvořit nový keyspace, který bude obsahovat požadovanou rodinu sloupců. Zde je nutné specifikovat způsob replikace. Cassandra dává na výběr buď SimpleStrategy nebo NetworkTopologyStrategy. Volba SimpleStrategy umožňuje kontrolovat replikační faktor (počet využitých replik) pouze pro cluster jako celek a je vhodná například pro testování databáze na jednom počítači. NetworkTopologyStrategy je vhodná pro produkční nasazení do distribuovaného prostředí – bere v potaz topologii sítě a umožňuje specifikovat různé replikační faktory pro různá datacentra (topologie sítě ale musí být ručně nastavena).

Výpis 10 - Vytvoření keyspace v databázi Cassandra

```
cqlsh> CREATE KEYSPACE akciekeyspace WITH replication = {'class' : 'SimpleStrategy',
'replication_factor' : 1};

//nebo

cqlsh> CREATE KEYSPACE akciekeyspace WITH replication = {'class':'NetworkTopologyStrategy',
'datacentrum1' : 2, 'datacentrum2' : 3};
```

Následně bude vytvořena rodina sloupců. Cassandra v tomto případě vyžaduje definování struktury, čehož lze dosáhnout příkazem *CREATE TABLE*. Tento příkaz lze nahradit také aliasem *CREATE COLUMNFAMILY.* Nutností definovat strukturu dat se tak Cassandra stává v tomto směru výjimkou mezi nerelačními databázemi.

Zde je důležité zmínit několik důležitých principů, týkajících se zejména definování primárního klíče. Ve výpisu 11 je vyobrazena definice schématu (představeného v podkapitole 3.2.3), které zaznamenává údaje o akciích, konkrétně identifikátor akcie (*akcie_id*), čas záznamu a její hodnotu. Primární klíč je definován jako složený primární klíč z údajů *akcie_id* a *cas*. To zajistí zaznamenávání údajů ve formě „širokých řádků“, jelikož kombinace identifikátoru akcie a času nemůže být pro dva záznamy stejná. V této deklaraci dále *akcie_id* plní roli **dělícího klíče** a *cas* roli tzv. **clustering column**. Dělící klíč zajišťuje ukládání všech dat nesoucí tento klíč na

jeden konkrétní uzel v databázi. Clustering column potom určuje sloupec, podle kterého budou data seřazena na disku.

Výpis 11 - Vytvoření rodiny sloupců v databázi Cassandra

```
cqlsh:akciekeyspace> CREATE TABLE Akcie (akcie_id text, cas timestamp, hodnota float,
PRIMARY KEY (akcie_id, cas));
```

Někdy může být ovšem množství dat v jednom řádku příliš velké pro zpracování na jednom databázovém uzlu. Ve výpisu 12 je uvedena definice rodiny sloupců *Akcie* s novým sloupcem *datum*, který bude plnit roli dělícího klíče společně se sloupcem *akcie_id*. Tím se docílí rozdělení záznamu konkrétní akcie na více řádků podle jednotlivých dnů. Sloupec *cas* funguje znovu jako clustering column.

Výpis 12 - Vytvoření rodiny sloupců s odlišnou definicí primárního klíče v databázi Cassandra

```
cqlsh:akciekeyspace> CREATE TABLE Akcie (akcie_id text, datum date, cas timestamp, hodnota
float, PRIMARY KEY ((akcie_id, datum), cas));
```

Primární klíč již nelze po vytvoření tabulky změnit (bylo by nutné vytvořit novou tabulku a do té data zkopírovat), proto je nutné tomuto aspektu věnovat při vytváření datového modelu značnou pozornost.

Do připravené struktury již lze přidávat nové záznamy. Přidávání několika záznamů je vyobrazeno ve výpisu 13. Cassandra při klasickém zadání nového záznamu s duplikátním primárním klíčem neohlásí chybu, naopak celý záznam přepíše novými hodnotami. Tomuto se lze vyhnout přidáním klíčových slov *IF NOT EXISTS* na konec celého příkazu.

Výpis 13 - Vkládání záznamů v databázi Cassandra

```
cqlsh:akciekeyspace> INSERT into akcie (akcie_id, datum, cas, hodnota) VALUES ('AMD', '2017-
01-01', '00:00:00', 14.06);
cqlsh:akciekeyspace> INSERT into akcie (akcie_id, datum, cas, hodnota) VALUES ('AMD', '2017-
01-01', '00:00:05', 14.07);

// Ochrana proti přepsání
cqlsh:akciekeyspace> INSERT into akcie (akcie_id, datum, cas, hodnota) VALUES ('AMD', '2017-
01-01', '00:00:05', 15.08) IF NOT EXISTS;

[applied]  | akcie_id | datum         | cas                          | hodnota
------------+------------+---------------+------------------------------+---------
   False   |     AMD   | 2017-01-01 | 00:00:05.000000000 |   14.07
```

Zajímavou funkcí je přiřazení hodnoty TTL k jednotlivým záznamům. Hodnota TTL se zadává v sekundách a označuje čas, po kterém je určený záznam smazán. TTL lze k záznamu přiřadit při jeho vkládání, jak je vyobrazeno na výpisu 14.

Výpis 14 - Vložení záznamu se specifikací TTL v databázi Cassandra

```
cqlsh:akciekeyspace> INSERT into akcie (akcie_id, datum, cas, hodnota) VALUES ('AMD', '2017-01-01', '00:00:15', 14.10) USING TTL 500;
```

3.2.4.2 Dotazování

Ačkoliv je dotazování pomocí CQL velmi podobné využití jazyka SQL, distribuovaná architektura Cassandry přináší určitá omezení.

Základním dotazem navazujícím na příklad s akciemi by mohl být výběr informací o akciích za určité časové období. Následující dotaz vyhledá informace o akciích, které byly zaznamenány 1. ledna 2017 od půlnoci do čtyř hodin. Tento dotaz je velmi efektivní, jelikož se jedná o sekvenční čtení záznamů, které jsou na disku seřazeny podle sloupce (clustering column) *cas.*

Výpis 15 - Ukázka příkazu select v databázi Cassandra

```
cqlsh:akciekeyspace> SELECT * FROM akcie WHERE akcie_id='AMD' AND datum = '2017-01-01' AND cas > '00:00:00' AND cas < '04:00:00';
```

Následující dotaz se pokouší získat informace a všech akciích za jeden konkrétní den, Cassandra ho ale vyhodnotila jako chybný. Důvodem je nutnost v klauzuli *WHERE* specifikovat buď celý dělící klíč, nebo žádnou jeho část. Cassandra specifikovaný dělící klíč využije k vypočtení heše, s jehož pomocí vyhledá uzly s požadovanými daty [29].

Výpis 16 - Dotaz bez specifikace celého dělícího klíče v databázi Cassandra

```
cqlsh:akciekeyspace> SELECT * FROM akcie WHERE datum='2017-01-01';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Partition key parts: akcie_id must be restricted as other parts are"

// Korektní provedení dotazu – uvedení celého dělícího klíče
cqlsh:akciekeyspace> SELECT * FROM akcie WHERE akcie_id='AMD' AND  datum='2017-01-01';
```

Výpis 17 ukazuje situaci dotazování se na data bez uvedení dělícího klíče. V příkladu je uveden dotaz k získání záznamů o akciích za předpokladu, že jejich hodnota je větší než 200.

Výpis 17 - Dotaz bez specifikace dělícího klíče v databázi Cassandra

```
cqlsh:akciekeyspace> SELECT * FROM akcie WHERE hodnota > 200;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this
query as it might involve data filtering and thus may have unpredictable performance. If you
want to execute this query despite the performance unpredictability, use ALLOW FILTERING"
```

Databázový systém v tomto případě nemůže zaručit efektivní provedení dotazu, jelikož musí z tabulky přečíst všechny záznamy a vyfiltrovat ty, které mají hodnotu větší než 200. V nejhorším případě by taková operace mohla projít celou tabulku a nenajít žádný vhodný záznam. Vynucení tohoto postupu lze dosáhnout připojením příkazu *ALLOW FILTERING* na konec dotazu [30].

Ačkoliv je možné filtrovaný dotaz provést, nelze tento přístup doporučit vzhledem k principům, na kterých Cassandra funguje. Provádění dotazů, pro který není datový model vhodně vytvořen, může negativně působit na výkon celého databázového systému, zejména pokud má poskytovat vysokou míru škálovatelnosti. Řešením je zde denormalizace a vytvoření nové tabulky tak, aby byl dotaz prováděn optimálně, jelikož zápis je v Cassandře prováděn velmi rychle. Toto je jeden z největších rozdílů sloupcové databáze oproti relačnímu přístupu.

3.3 Grafová databáze Neo4j

Neo4j je grafová databáze vyvíjená americko-švédskou společností Neo Technology. První její verze vyšla v roce 2007. Neo4j se zaměřuje na efektivní ukládání grafových struktur, tedy vrcholů a hran. Implementován je v jazyce Java, tudíž je přenositelný mezi operačními systémy. Na rozdíl od jiných nerelačních databází má Neo4j podporu ACID vlastností. [6]

K databázi lze přistupovat buď z programového kódu pomocí Java API nebo REST API, nebo využít rozhraní speciálního jazyku Cypher. [6]

Cypher je deklarativní jazyk, který slouží pro úpravu dat a dotazování. Jazyk umožňuje využití ASCII-Art symbolů k reprezentaci určitých vzorců, například při

prohledávání grafu. To umožňuje srozumitelné vyjádření dotazů a usnadňuje jejich pochopení. Důležitá je deklarativní vlastnost jazyka, která se soustředí na otázku „co" je získáno po projití grafu, ne „jak" tímto grafem projít. [31]

Tabulka 7 - Neo4j, základní informace, Zdroj: [32]

Vývojář	Neo Technology
První vydání	2007
Implementováno v	Java
Podporované OS	Linux, OS X, Solaris, Windows
Podpora programovacích jazyků	13 jazyků
Způsob rozdělení dat	Nepodporuje
Způsob replikace	Kauzální clustering[1]
Podpora MapReduce	Ne
Podpora ACID transakcí	Ano

3.3.1 Verze a licence

Neo4j je dostupný v Community a Enterprise edici. Community verze (zdarma) je šířena pod GPLv3 licencí, Enterprise verze (placená) buď pod AGPLv3 nebo komerční licencí. Placená verze nabízí pokročilý monitoring, online zálohování nebo možnost škálování, která v Community verzi není dostupná. [33]

3.3.2 Využití Neo4j – doporučovací systém

Doporučování produktů, zejména v reálném čase, je oblast, která při správném využití může přinést provozovateli značný zisk. Doporučovací algoritmy určují a hledají vztahy mezi lidmi, produkty nebo službami, podle potřeb problémové domény. Vztahy jsou určovány na základě chování uživatelů, tedy například podle položek jejich nákupů nebo hodnocení vybraných produktů. Ze zaznamenaných

[1] Dostupné pouze v placené verzi Enterprise

vzorců chování jsou následně určeny produkty, o které může mít určitý jedinec zájem. [12]

Vzhledem k tomu, že problémová doména je plná vztahů mezi věcmi a lidmi, je vhodné pro analýzu takové struktury vhodné použít grafovou databázi. Jedním z uživatelů grafové databáze Neo4j je americký obchodní řetězec Walmart, který ji využívá k doporučování produktů zákazníkům na svých webových stránkách. Vzhledem k tomu, že Walmart obslouží až 250 milionů zákazníků týdně, je generovaný objem dat tak velký, že využití relační databáze nebylo pro firmu dostatečně efektivní, a pro účely doporučování v reálném čase začala využívat právě Neo4j. [34]

3.3.3 Ukázka

Pro účel ukázky bude vytvořen jednoduchý dataset obsahující zákazníky, nabízené produkty a výrobce těchto produktů. Na těchto datech budou předvedeny dotazy, které mohou být využity ke zjišťování vztahů mezi zákazníky a produkty. V ukázce je použit databázový systém Neo4j ve verzi 3.0.4.

3.3.3.1 Vytvoření požadovaných struktur a manipulace s daty

Nejprve je nutné vytvořit entity představující zákazníky, produkty a výrobce. Toho je dosaženo příkazem *CREATE*, ve kterém se nejprve specifikuje štítek, který bude označovat typ nově vytvořené entity – vrcholu v grafu. Za štítkem je možné zadat vlastnosti nově vytvářeného vrcholu.

Výpis 18 - Vytvoření vrcholů

```
CREATE (a:Zakaznik { jmeno: 'Ales Novotny'}), (b:Zakaznik { jmeno: 'Anna Zelena'})
CREATE (n:Produkt { nazev: 'Televize', cena: 15000})
CREATE (n:Vyrobce { nazev: 'Samsung'})
```

Následně budou vytvořeny vztahy mezi vrcholy. Zde je nutné využít příkazu *MATCH*, který slouží k prohledávání databáze. Jsou specifikovány vrcholy pomocí jejich štítků, poté jsou tyto vrcholy filtrovány podle podmínky v klauzuli *WHERE.* Nakonec je vytvořen vztah mezi určenými vrcholy. Vztahy jsou v Neo4j vždy orientované a lze k nim přiřadit jak štítek (v tomto případě *zna, jeVyrobcem* a *koupil)*, tak i vlastnosti.

Výpis 19 - Vytvoření vztahů mezi vrcholy

```
//Vytvoření vztahu mezi zákazníky
MATCH (z1:Zakaznik),(z2:Zakaznik)
WHERE z1.jmeno = 'Anna Zelena' AND z2.jmeno = 'Ales Novotny'
CREATE (z1)-[r:zna]->(z2)

//Vytvoření vztahu jeVyrobcem mezi výrobcem a produktem
MATCH (p:Produkt),(v:Vyrobce)
WHERE p.nazev = 'Smartphone' AND v.nazev = 'Samsung'
CREATE (v)-[r:jeVyrobcem]->(p)

//Vytvoření vztahu koupil mezi zákazníkem a produktem
MATCH (z:Zakaznik),(p:Produkt)
WHERE z.jmeno = 'Ales Novotny' AND p.nazev = 'Smartphone'
CREATE (z)-[r:koupil]->(p)
```

Pro účel příkladu byly analogicky vytvořeny další vrcholy a vztahy mezi nimi. Vytvořené schéma je vyobrazeno na obrázku 11.

Obrázek 11 - Datové schéma příkladu s Neo4j, Zdroj: webové rozhraní databázového systému Neo4j, vlastní zpracování

3.3.3.2 Dotazování.

K dotazování je znovu využito příkazu *MATCH.* První dotaz slouží ke zjištění produktů, které si koupili dva určití zákazníci. Zákazníci jsou zde přímo specifikováni pomocí jména, vztahy lze snadno vyjádřit pomocí upřesnění jejich názvu a směru. Směr lze přehledně formulovat pomocí ASCII symbolů, například *-[:vztah]->.* Výsledkem dotazu bude produkt *lednice*.

Výpis 20 - Ukázka dotazu - vyhledání společného produktu

```
MATCH (z1:Zakaznik {jmeno: 'Petra Vranova'})-[:koupil]->(produkt)<-[:koupil]-(z2:Zakaznik
{jmeno: 'Jana Zelena'})
RETURN produkt
```

Výsledkem dotazu bude produkt *lednice*, který je společný pro specifikované zákaznice.

Další dotaz se zaměřuje na vztah mezi zákazníky samotnými. Pro prvního zákazníka specifikovaného jménem jako *Anna Zelena* jsou doporučeny produkty zakoupené zákazníky, které *Anna Zelena* zná.

Výpis 21 - Ukázka dotazu - vyhledání produktů, kteří si koupili přátelé

```
MATCH (z1:Zakaznik {jmeno: 'Anna Zelena'})-[:zna]->(z2:Zakaznik)-[:koupil]-
>(DoporucenyProdukt)
RETURN DoporucenyProdukt
```

Jelikož je Neo4j zaměřený na práci se vztahy, je snadné vyjádřit i dotaz, který vyhledá produkty z druhé úrovně přátelství mezi zákazníky. Tento dotaz ukazuje opravdovou sílu databázového systému Neo4j, jelikož podobný dotaz v relační databázi by byl značně komplikovaný a uživatelsky nepřívětivý.

Výpis 22 - Ukázka dotazu - přátelé přátel a jejich produkty

```
MATCH (z1:Zakaznik {jmeno: 'Ales Novotny'}), (z1)-[:zna]->(Znamy), (Znamy)-[:zna]-
>(ZnamyZnameho), (ZnamyZnameho)-[:koupil]->(produkt)
WHERE NOT (ZnamyZnameho.jmeno = 'Ales Novotny')
RETURN z1,Znamy,ZnamyZnameho,produkt
```

Poslední ukázkou bude dotaz, který se zaměřuje na loajalitu zákazníka k určité značce. Je totiž pravděpodobné, že po nákupu několika produktů od stejného výrobce bude mít zákazník zájem i o další. Dotaz tedy hledá produkty, které zákazník ještě nezakoupil a které jsou produkovány výrobcem, od kterého zákazník již

zakoupil více než jeden produkt.

Výpis 23 - Ukázka dotazu - loajalita značce

```
MATCH (z:Zakaznik)-[:koupil]->(p1:Produkt)<-[:jeVyrobcem]-(v:Vyrobce)-[:jeVyrobcem]-
>(DoporucenyProdukt:Produkt)
WITH z, count(p1) as pocetZakoupenych, DoporucenyProdukt
WHERE NOT (z)-[:koupil]->(DoporucenyProdukt) AND pocetZakoupenych > 1
RETURN z, DoporucenyProdukt
```

Výsledkem dotazu bude zákazník *Jan Novak*, kterému bude doporučen *tablet* od firmy *Samsung.*

4 Shrnutí výsledků

V práci byly představeny tři konkrétní nerelační databázové systémy, a to MongoDB, Cassandra a Neo4j. Každý z těchto třech aplikuje odlišný přístup k vytváření datového modelu, což znamená odlišné možnosti jejich využití. Tyto možnosti byly demonstrovány na popsaných případech užití.

MongoDB se ukázal být vhodným pro ukládání vzájemně velmi různých záznamů – v tomto případě údajích o produktech elektronického obchodu. Toho bylo docíleno možností ukládat dokumenty ve struktuře JSON bez přesně daného schématu.

Cassandra je vhodná pro data, která jsou zapisována často a ve velkém objemu, díky specifickému přístupu k jejich ukládání. Představen byl příklad ukládání časových řad na akciovém trhu.

Možnosti Neo4j byly ukázány na příkladu zákazníků nakupujících různé produkty. Jazyk Cypher, který Neo4j využívá, se ukázal být silným nástrojem k vytváření přehledných dotazů na vztahy mezi daty.

V praktických ukázkách práce s databázemi vyšla najevo jedna z nevýhod NoSQL obecně, a to nejednotnost v dotazovacích jazycích – MongoDB využívá JavaScript, Cassandra jazyk CQL, a Neo4j používá dotazovací jazyk Cypher. Speciálně jazyk CQL může být pro potencionální zájemce až zavádějící, jelikož je velmi podobný jazyku SQL, ale díky odlišné architektuře databázového systému Cassandra je nutné ho využívat odlišným způsobem.

Celkově lze konstatovat, že vybrané databázové systémy ukázaly svoji sílu při řešení určitého specifického problému.

5 Závěry a doporučení

Nerelační databáze jsou rozmanitým odvětvím informačních technologií. Ačkoliv jsou zaškatulkovány pod jednotným názvem NoSQL, existuje zde mnoho různých přístupů jak k datovým modelům, tak k možnostem škálovatelnosti a konzistence.

V této práci byl nejdříve stručně popsán klasický relační přístup. Následující část se věnovala přístupům nerelačních databází. Představena byla jejich historie, základní vlastnosti a typy. Dále se práce zaměřovala na odlišný přístup nerelačních databází ke konzistenci a škálovatelnosti. Představen byl i framework MapReduce. Praktická část se pak věnovala případům užití, ve kterých lze nerelační databáze využít. Součástí byla praktická ukázka práce s vybranými databázemi.

Bakalářská práce by mohla být přínosem pro databázové administrátory nebo vývojáře, kteří nemají zkušenosti s nerelačním přístupem, nebo se s ním setkali jen okrajově. Dále pro studenty, kteří se chtějí seznámit s novými technologiemi.

V rámci dalšího výzkumu lze představit další konkrétní databázové systémy typů, kterým se tato práce nevěnovala, například key-value (Redis, Riak), vyhledávací enginy (Elasticsearch, Solr) nebo hybridní databáze kombinující více datových modelů (OrientDB). Možná je také implementace představených databází do funkční aplikace.

6 Seznam použité literatury

[1] CONNOLLY, Thomas M., BEGG Carolyn E. *Database systems: a practical approach to design, implementation, and management*. 4. vydání. New York: Addison-Wesley, 2005. ISBN 0321210255. Dostupné z: http://www.palinfonet.com/download/software2/database%20systems.pdf

[2] Microsoft. *Types of Table Relationships (Visual Database Tools)*. In: *Microsoft,* [online]. 2016, [cit. 15. 10. 2016]. Dostupné z: https://technet.microsoft.com/en-us/library/ms190651(v=sql.105).aspx

[3] BESENYEI, Enrico. *How a database index can help performance?* [online]. [cit. 18. 11. 2016]. Dostupné z: http://www.eandbsoftware.org/how-a-database-index-can-help-performance/

[4] VRÁNA, Jakub. *Využití databázových indexů*. In: *Root.cz* [online]. 2003, cit. [15. 10. 2016]. Dostupné z: https://www.root.cz/clanky/vyuziti-databazovych-indexu/

[5] SULLIVAN, Dan. *NoSQL for mere mortals*. Hoboken, NJ: Addison-Wesley, 2015. For mere mortals series. ISBN 9780134023212

[6] HOLUBOVÁ, Irena, KOSEK, Jiří, MINAŘÍK, Karel, NOVÁK, David. *Big Data a NoSQL databáze*. 1. vydání. Praha: Grada Publishing, a.s., 2015, 288 s. ISBN 978-80-247-5938-8.

[7] CELKO, Joe. *Joe Celko's Complete guide to NoSQL: what every SQL professional needs to know about nonrelational databases*. Boston: Elsevier/Morgan Kaufmann, 2014. ISBN 9780124071926.

[8] FOWLER, Adam. *Nosql for dummies*. Indianapolis, In John Wiley and Sons, 2015. ISBN 1118905741.

[9] GIRISH, Kumar. *Exploring the different types of NoSql databases part II*. In: *3 Pillar global,* [online]. [cit. 15. 5. 2016]. Dostupné z: http://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases

[10] WONG, Han. *Common DocumentDB use cases*. In: *Microsoft Azure*, [online]. 2016, [cit. 17. 5. 2016]. Dostupné z: https://azure.microsoft.com/en-us/documentation/articles/documentdb-use-cases/

[11] ANDERSON, Jesse. *A Look At HBase, the NoSQL Database Built on Hadoop.* In: *The new stack*, [online]. 2015, [cit. 17. 5 2016]. Dostupné z: http://thenewstack.io/a-look-at-hbase/

[12] ROBINSON, Ian, WEBBER, Jim, EIFREM, Emil. *Graph Databases*. 2. vydání. O'Reilly Media, 2013. 224 s. ISBN 978-1-4493-5626-2. Dostupné z: http://graphdatabases.com/

[13] CHANDRA, Deka Ganesh. *BASE analysis of NoSQL database*. In Future Generation Computer Systems: No. 52, 2015, s. 13–21 [online]. [cit. 21. 5. 2016]. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0167739X15001788

[14] SADALAGE, Pramod J., FOWLER Martin. *NoSQL distilled a brief guide to the emerging world of polyglot persistence*. Upper Saddle River, NJ: Addison-Wesley, 2012. ISBN 9780133036138.

[15] BREWER, Eric. *Towards robust distributed systems (invited talk)*. In PODC. ACM Press, 2000. str. 7. Dostupné z: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[16] HEWITT, Eben. *Cassandra: the definitive guide*. Beijing: O'Reilly Media, 2011. ISBN 9781449390419.

[17] Flux7, *CAP Theorem: Its importance in distributed systems.* In: *E&B Software*, [online]. 2014, [cit. 22. 10. 2016]. Dostupné z: http://blog.flux7.com/blogs/nosql/cap-theorem-why-does-it-matter

[18] BREWER, Eric. *CAP Twelve Years Later: How the "Rules" Have Changed.* In: *InfoQ,* [online]. 2012, [cit. 23. 9. 2016]. Dostupné z: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

[19] MongoDB, Inc. *Map-Reduce.* In: *MongoDB* [online]. 2016, [cit. 22. 10. 2016]. Dostupné z: https://docs.mongodb.com/manual/core/map-reduce/

[20] HARRIS, Derrick. *10gen embraces what it creted, becomes MongoDB, Inc.* In: Gigaom, [online]. [cit. 28. 1. 2017]. Dostupné z: https://gigaom.com/2013/08/27/10gen-embraces-what-it-created-becomes-mongodb-inc/

[21] Creative Commonst, *BSON* [online]. [cit. 28. 1. 2017]. Dostupné z: http://bsonspec.org/faq.html

[22] DB-Engines. *MongoDB System Properties.* In: *DB-Engines.com* [online]. 2017, cit. [1. 4. 2017]. Dostupné z: http://db-engines.com/en/system/MongoDB

[23] MongoDB, Inc. *MongoDB Enterprise Advanced.* In: *MongoDB,* [online]. 2017, cit. [30. 3. 2017], Dosupné z: https://www.mongodb.com/products/mongodb-enterprise-advanced

[24] MongoDB, Inc. *SQL to MongoDB Mapping Chart.* In: *MongoDB,* [online]. 2017, cit. [16. 3. 2017]. Dostupné z: https://docs.mongodb.com/manual/reference/sql-comparison/

[25] DB-Engines. *Cassandra System Properties.* In: *DB-Engines.com* [online]. 2017, cit. [1. 4. 2017]. Dostupné z: http://db-engines.com/en/system/Cassandra

[26] DataStax, Inc. *How is data written.* In: *Datastax,* [online]. 2017, [cit. 10. 2. 2017]. Dostupné z: http://docs.datastax.com/en/archived/cassandra_win/3.0/cassandra/dml/dmlHowDataWritten.html

[27] O'NEIL, Patrick, et al. *The log-structured merge-tree (LSM-tree)*. Acta Informatica, 1996, 33(4), str. 351-385.

[28] DataStax, Inc. *CQL limits.* In: *Datastax,* [online]. 2017, [cit. 5. 2. 2017]. Dostupné z: https://docs.datastax.com/en/cql/3.1/cql/cql_reference/refLimits.html

[29] LERER, Benjamin. *A deep look at the CQL WHERE clause.* In: *Datastax,* [online]. 2015, [cit. 23. 2. 2017]. Dostupné z: http://www.datastax.com/dev/blog/a-deep-look-to-the-cql-where-clause

[30] LERER, Benjamin. *ALLOW FILTERING explained.* In: *Datastax,* [online]. 2014, [cit. 23. 2. 2017]. Dostupné z: http://www.datastax.com/dev/blog/allow-filtering-explained-2

[31] Neo4j, Inc. *Cypher.* In: Neo4j, [online]. 2017, [cit. 7. 3. 2017]. Dostupné z: http://neo4j.com/docs/developer-manual/current/cypher/

[32] DB-Engines. *Neoj4 System Properties.* In: *DB-Engines.com* [online]. 2017, cit. [1. 4. 2017]. Dostupné z: http://db-engines.com/en/system/Neo4j

[33] Neo4j, Inc. *Compare Neo4j Editions* [online]. 2017, cit. [30. 3. 2017]. Dostupné z: https://neo4j.com/editions/

[34] Neo4j, Inc. *Walmart uses Neo4j to optimize customer experience with personal recommendations.* In: Neo4j, [online]. 2015, [cit. 9. 3. 2017]. Dostupné z: http://info.neo4j.com/rs/neotechnology/images/neo4j-casestudy-walmart.pdf

7 Seznam obrázků

8 Seznam tabulek

9 Seznam výpisů

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2016/2017

Studijní program: Systémové inženýrství a informatika
Forma: Prezenční
Obor/komb.: Informační management (im3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bořík Pavel	Malecí 572, Nové Město nad Metují	I14316

TÉMA ČESKY:

Nerelační databáze

TÉMA ANGLICKY:

Non-relational databases

VEDOUCÍ PRÁCE:

Ing. Barbora Tesařová, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cílem práce je představení problematiky nerelačních (neboli NoSQL) databází v kontextu relačního přístupu, hledání možností jejich uplatnění a předvedení práce s nimi na ukázkových příkladech.
Osnova:
1. Úvod
2. Teoretická část
2.1 Běžný relační přístup
2.2 Nerelační (NoSQL) přístupy
3. Praktická část
4. Závěry a doporučení
5. Seznam použité literatury

SEZNAM DOPORUČENÉ LITERATURY:

HOLUBOVÁ, Irena, KOSEK, Jiří, MINAŘÍK, Karel, NOVÁK, David.
Big Data a NoSQL databáze. 1. vydání. Praha: Grada Publishing, a.s., 2015, 288 stran, ISBN 9788024759388,
SULLIVAN, Dan. NoSQL for mere mortals. Hoboken, NJ: Addison-Wesley, 2015. For mere mortals series. ISBN 9780134023212,
CELKO, Joe. Joe Celko's Complete guide to NoSQL: what every SQL professional needs to know about nonrelational databases. Boston: Elsevier/Morgan Kaufmann, 2014. ISBN 9780124071926.

Podpis studenta: ..... **Datum:** 17.10.2016

Podpis vedoucího práce: ..... **Datum:** 17.10.2016