Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Práce s velkými objemy dat v relačních a NoSQL databázích

Bakalářská práce

Autor: Jaroslav Schnaubert
Studijní obor: Aplikovaná informatika

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

Odborný konzultant: Ing. Karel Schejbal, Unicorn a.s.

Hradec Králové Srpen 2019

Prohlášení:

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

V Hradci Králové dne 13.8.2019 Jaroslav Schnaubert

Poděkování:

Chtěl bych poděkovat vedoucí bakalářské práce Ing. Barboře Tesařové, Ph.D. za metodické vedení práce. Rovněž bych chtěl poděkovat odbornému konzultantovi Ing. Karlu Schejbalovi za jeho cenné rady a pomoc při tvorbě práce.

Anotace

Tématem této práce jsou relační a NoSQL databáze z pohledu práce s velkými objemy dat. Práce je konkrétně zaměřena na relační databázi MySQL a dokumentově orientovanou NoSQL databázi MongoDB.

Práce se zabývá řešením problematiky škálovatelnosti, replikace, zálohování, návrhu databáze z hlediska vysoké dostupnosti databáze, návrhu indexů, manipulace s velkými objemy dat, optimalizace dotazů a monitorování databází.

Celý text předpokládá, že je čtenář dostatečně seznámen se základními principy relačních a NoSQL databází. Cílem této práce je poskytnout čtenáři vodítko při rozhodování mezi databázemi MySQL a MongoDB.

Klíčová slova:

Databáze, Relační databáze, NoSQL, MySQL, MongoDB

Annotation

Title: Work with Large Quantities of Data in Relational and NoSQL databases

The scope of this thesis is relational and NoSQL databases from the perspective of working with large quantities of data. The thesis is focused on relational database MySQL and document-oriented NoSQL database MongoDB.

This work is engaged in problematics of scalability, replication, backup, database design in terms of high database availability, index design, handling of large volumes of data, query optimization and database monitoring.

The whole text assumes that the reader is sufficiently familiar with the basic principles of relational and NoSQL databases. The aim of this work is to provide the reader with a guide in making decisions between MySQL and MongoDB.

Key words:

Database, Relational database, NoSQL, MySQL, MongoDB

Obsah

1 Úvod ... 1

1.1 Důvod výběru tématu seminární práce ... 1

1.2 Cíl bakalářské práce ... 1

1.3 Rešerše podobně zaměřených prací ... 1

2 Relační a NoSQL databáze ... 3

2.1 Relační databáze ... 3

2.1.1 MySQL ... 4

2.2 NoSQL databáze ... 5

2.2.1 Sloupcově orientované databáze ... 5

2.2.2 Grafové databáze ... 6

2.2.3 Databáze s klíčem a hodnotou ... 7

2.2.4 Dokumentově orientované databáze ... 7

3 Škálovatelnost, replikace a zálohování ... 9

3.1 Škálovatelnost ... 9

3.1.1 Horizontální škálování ... 9

3.1.2 Vertikální škálování ... 14

3.2 Replikace ... 14

3.2.1 Replikace MySQL ... 15

3.2.2 Replikace MongoDB ... 15

3.3 Zálohování ... 18

4 Jak navrhovat strukturu databází ... 20

4.1 Jak navrhovat strukturu MySQL ... 20

4.1.1 Normalizace MySQL databáze ... 21

4.2 Jak navrhovat strukturu MongoDB ... 22

4.3 Indexy ... 23

4.3.1 MySQL indexy ... 23

4.3.2 MongoDB indexy 26

5 Manipulace se strukturou a daty v živém provozu 30

5.1 MySQL – Přidání sloupce 30

5.1.1 Výběr způsobu vložení nového sloupce 30

5.1.2 Analýza 31

5.1.3 Přidání sloupce 31

5.1.4 Závěr úlohy 31

5.2 Hromadná úprava hodnot v MySQL tabulce s 500 milióny řádky 32

5.2.1 Analýza 32

5.2.2 Odstranění duplicitních verzí 33

5.2.3 Hledání duplicitních verzí 33

5.2.4 Zarovnání duplicitních verzí 35

5.2.5 Nahrazení dlouhých kódů krátkými 36

5.2.6 Závěr úlohy 37

5.3 Optimalizace MongoDB indexů 38

5.3.1 Analýza 38

5.3.2 Návrh a aplikace nových indexů 39

5.3.3 Porovnání výsledků 40

5.3.4 Závěr úlohy 41

5.4 Využití MongoDB repliky pro aktualizace verze databáze 41

5.4.1 Analýza 41

5.4.2 Přidání sekundárních serverů do MongoDB replica setu 42

5.4.3 Naplnění nových serverů daty 42

5.4.4 Aktualizace MongoDB serverů na verzi 3.4 44

5.4.5 Závěr úlohy 45

6 Optimalizace dotazů a izolačních úrovní 46

6.1 Izolační úrovně 46

6.2 Izolační úrovně v MySQL 46
6.3 Izolační úrovně v MongoDB 47
7 Monitorování databáze a operací 48
7.1 Monitorování MySQL databáze 48
7.2 Monitorování MongoDB databáze 49
7.3 Obecné monitorovací systémy 50
8 Zhodnocení výsledků 51
9 Závěr 52
10 Seznam zdrojů 53
11 Přílohy 55

Seznam obrázků

Obrázek 1 - Grafová databáze ... 6
Obrázek 2 - Ukázka Škálování [14] ... 10
Obrázek 3 - Škálování – systém s jedním serverem [14] ... 11
Obrázek 4 - Škálování – lineární škálování [14] ... 11
Obrázek 5 - Škálování – nelineární škálování [14] ... 11
Obrázek 6 - MySQL Load balancer [14] ... 12
Obrázek 7 - Škálování – příklad MongoDB shardů [15] ... 13
Obrázek 8 - Škálování – Sharderd Cluster [15] ... 13
Obrázek 9 - MongoDB replikace [15] ... 16
Obrázek 10 - Volba nového primárního MongoDB serveru bez arbitera [15] ... 17
Obrázek 11 - Volba nového primárního MongoDB serveru s arbiterem [15] ... 17
Obrázek 12 - Relace pomocí odkazu [15] ... 22
Obrázek 13 - Vnořená data [15] ... 23
Obrázek 14 - B-Tree index [16] ... 24
Obrázek 15 - MognoDB index [15] ... 27
Obrázek 16 - Single field index [15] ... 27
Obrázek 17 – Compound index [15] ... 28
Obrázek 18 – Multikey index [15] ... 29

Seznam SQL příkazů

SQL příkaz 1 – tabulky ukládající reference duplicitních hodnot ... 34
SQL příkaz 2 – vyhledávání duplicit prvního typu ... 34
SQL příkaz 3 - vyhledávání duplicit druhého typu ... 35
SQL příkaz 4 - nahrazení dlouhých kódů krátkými ... 36
SQL příkaz 5 - nahrazení dlouhých kódů krátkými pro historické hodnoty ... 37
SQL příkaz 6 - nahrazení dlouhých kódů krátkými pro aktuální hodnoty ... 37

Seznam MongoDB příkazů

MongoDB příkaz 1 – nově navržené indexy 39
MongoDB příkaz 2 – smazání starých indexů 40
MongoDB příkaz 3 – přidání serverů do replica setu 42
MongoDB příkaz 4 – snížení priority v replica setu 42
MongoDB příkaz 5 – zvětšení velikosti oplogu 43
MongoDB příkaz 6 – povolení řetězové replikace 44
MongoDB příkaz 7 – odstranění serveru z MongoDB replica setu 44
MongoDB příkaz 8 - zvýšení priority v replica setu 44

Seznam tabulek

Tabulka 1 – porovnání délky načítání dat před a po změně indexů 40

1 Úvod

1.1 *Důvod výběru tématu seminární práce*

Hlavním důvodem vybrání tématu je autorova záliba v relačních a nerelačních databázových systémech, které již v několika projektech využil.

Dalším důvodem je autorův zájem ponořit se hlouběji do problematiky NoSQL databází a jejich rozdílů oproti databázím relačním.

1.2 *Cíl bakalářské práce*

Hlavním cílem této práce je porovnat a zhodnotit výhody a nevýhody relačních a NoSQL databází pro práci s velkými objemy dat, a to konkrétně pro relační databázi MySQL[1] a dokumentově orientovanou databázi MongoDB[2]. Případným čtenářům by práce mohla být užitečná při výběru databáze pro jejich budoucí aplikace.

V teoretické části práce budou nejprve zmíněny práce, které se zabývají problematikou relačních a NoSQL databází. Dále budou připomenuty základy relačních databází, NoSQL databází a také základní kategorie NoSQL databází. Následně bude popsána replikace, zálohování a škálovatelnost pro databáze MySQL a MongoDB. Dále bude v teoretické části popsáno, jak efektivně tvořit strukturu databáze.

V praktické části práce bude testována manipulace s daty v živém provozu, bude popsána optimalizace dotazů pro co nejefektivnější práci s oběma databázemi a budou znázorněny možnosti monitorování databáze a operací.

1.3 *Rešerše podobně zaměřených prací*

Před vypracováním této práce byly prostudovány následující práce, které jsou zaměřené na příbuzná téma:

Srovnání řešení správy dat v MySQL a MongoDB při použití Doctrine 2 ve frameworku Symfony 2 (Vysoká škola ekonomická v Praze, 2016) – V této bakalářské práci její autor Dominik Firla popisuje základy relačních a NoSQL

---

[1] Oficiální stránky MySQL - https://www.mysql.com/

[2] Oficiální stránky MongoDB - https://www.mongodb.com/

databází s konkrétním zaměřením na MySQL a MongoDB. Následně se věnuje implementaci obou databázových systému ve frameworku Symfony 2, při které testuje náročnost implementace obou databází. Nakonec autor vygeneruje testovací data pro provedení výkonových testů. Během těchto testů není jednoznačně prokázáno, že by jeden nebo druhý databázový systém byl jednoznačně rychlejší za všech okolností. [3]

NoSQL databáze (Jihočeská univerzita v Českých Budějovicích, 2013) – Tomáš Panyko se v této práci nejprve věnuje části teoretické, ve které popisuje historii databází, následně se zabývá popsáním základů relačních databází a poté se věnuje popisu NoSQL databází. Následně v části praktické porovnává zástupce relačních databází MySQL se zástupcem NoSQL databází Redis a ukazuje, jak tyto produkty nainstalovat, nakonfigurovat a jak s nimi pracovat. [2]

NoSQL databáze (Vysoká škola ekonomická v Praze, 2014) – Jakub Mrozek se v této bakalářské práci zabývá NoSQL databázemi. Autor ve své práci předpokládá, že čtenář je již dostatečně seznámen se základními principy NoSQL databází. Autor v teoretické části práce popisuje 3 kategorie NoSQL databází a nejpoužívanějšího zástupce od každé kategorie. První typem NoSQL databází, kterou autor ve své práci popisuje je dokumentově orientovaná databáze s konkrétním zaměřením na databázi MongoDB, následně autor popisuje grafové databáze se zaměřením na databázi Neo4j a posledním popisovaným typem NoSQL databáze je databáze hodnota – klíč, kde je autor zaměřen na databázi Redis. V praktické části autor srovnává u každého zmíněného typu NoSQL databáze 3 zástupce daného typu podle několika parametrů. [1]

2 Relační a NoSQL databáze

2.1 *Relační databáze*

Relační databáze organizují data do tabulek, které se skládají ze sloupců a řádků. Sloupce se nazývají atributy a jejich název musí být unikátní, řádky se nazývají záznamy. Jednotlivé tabulky mají mezi sebou předem definované vztahy.

Každý sloupec tabulky má určen datový typ, který definuje, jaká data bude daný sloupec obsahovat. Datových typů je velké množství a jejich výčet je závislý na konkrétním databázovém systému. Datové typy mohou být například Integer (číslo), Varchar (text) či Date (datum).

Data v tabulce jsou drženy v řádcích, což je řez tabulkou přes jednotlivé sloupce. Tabulka by neměla mít řádek, který by obsahoval úplně stejné hodnoty v jednotlivých sloupcích, jako řádek jiný. Z toho důvodu obsahuje většina tabulek sloupec, kterému se říká primární klíč. Primární klíč je jednoznačný identifikátor, který slouží k rozlišení jednotlivých záznamů z důvodu zamezení duplicit. Klasickým příkladem primárního klíče je například rodné číslo, které má každá osoba unikátní.

Hlavním užitím primárního klíče tabulky je jeho použití v tabulce jiné. Pokud je primární klíč určité tabulky použit v tabulce jiné, je nazýván cizím klíčem. Cizí klíče jsou využívány k zamezení duplicitních dat. Příkladem může být například adresa, jelikož na jedné adrese může bydlet více osob.

Hlavní charakteristikou relačních databází je pevná struktura dat. Díky pevné struktuře dat je předem známá struktura dat v databázi, což se hodí při práci se stálou strukturou dat. Avšak pokud je potřebné mít možnost flexibilně měnit strukturu databáze, není tento typ databáze vhodnou volbou. Změna schématu tabulky s velkým počtem záznamů je komplikovaná a není to operace na pár vteřin. Navíc jsou změny struktury velmi těžko realizovatelné bez odstávek.

K manipulaci s daty jsou v relačních databázích využívány transakce. Transakce je skupina příkazů, které je nutné vykonat k dokončení dané operace. K úspěšnému provedení transakce musí být úspěšně provedeny všechny příkazy v dané transakci.

Typickým příkladem transakce je bankovní převod, který se skládá z následujících kroků:

1. Kontrola, zdali odesílatel má na svém účtu danou částku.
2. Odečtení dané částky na z účtu odesílatele.
3. Přičtené dané částky na účtu příjemce.

Transakce v relační databázi musí splňovat skupinu 4 vlastností, známé pod zkratkou ACID. Jedná se o následující vlastnosti:

- Atomicita
    - Atomicita zajišťuje, že transakce proběhne jako celek, tzn. buď se provedou všechny kroky transakce anebo žádný.
- Konzistence
    - Databáze se vždy po průběhu transakce přesune z jednoho konzistentního stavu do dalšího.
- Izolace
    - Výsledky jednotlivých příkazů v transakci jsou skryté ostatním transakcím, dokud daná transakce není kompletně provedená. Izolace zaručuje, že se změny v databázi projeví až po dokončení dané transakce.
- Trvanlivost
    - Jakmile je transakce dokončená, její změny jsou permanentní [5].

2.1.1 MySQL

MySQL je databázový systém, který je vlastněný společností Oracle a je distribuovaný jak pod bezplatnou licencí, tak i pod komerční licencí.

MySQL patří mezi nejznámější a nejpoužívanější zástupce nejen relačních databázových systémů, ale databázových systému celkově. Dalšími z nejznámějších

a nejpoužívanějších relačních databázových systémů jsou například mimo jiné také Oracle Database[3] či MS SQL[4] [4].

Mezi hlavní výhody MySQL patří vysoký výkon, škálovatelnost a jednoduchost implementace [6].

MySQL podporuje několik úložných formátů, které umožňují přizpůsobit jednotlivé tabulky pro konkrétní použití. Nejpoužívanějšími formáty jsou formáty MyISAM a InnoDB [7].

MyISAM je velmi výhodné používat pro databáze, ve kterých jsou tabulky, ze kterých se data převážně jen čtou a málokdy upravují [6].

InnoDB přináší velkou výhodu v podpoře transakčního zpracování, což je nezbytné mají-li být v databázi splněné tzv. ACID vlastnosti. InnoDB také oproti MyISAM podporuje omezení ve formě cizích klíč, čímž je možné zaručit konzistenci dat a není možné, aby nastala situace, ve které by cizí klíč odkazoval na neexistující záznam jiné tabulky [6].

2.2 *NoSQL databáze*

NoSQL neboli nerelační databáze jsou databáze, které se nedrží relační cesty řízení dat. Podle názvu lze předpokládat, že se jedná o protějšek relačních databází, kvůli spojení slov No a SQL. Tato zkratka může být matoucí a neexistuje jednotný názor, co zkratka vlastně znamená. Nejvíce rozšířený názor udává význam zkratky jako „Not only SQL“, jelikož některé NoSQL databáze již mohou podporovat dotazovací jazyky podobné SQL [2].

Termín NoSQL zaštiťuje především sloupcově orientované databáze, databáze s klíčem a hodnotou, grafové databáze a dokumentově orientované databáze. [2]

2.2.1 Sloupcově orientované databáze

Sloupcově orientované databáze jsou podobné těm relačním, jelikož ukládají data do tabulek. Avšak sloupcově orientované databáze ukládají data do sloupců, na rozdíl od relačních databází, která ukládají data do řádků [8].

---

[3] Oficiální stránky Oracle Database - http://www.oracle.com/technetwork/database/

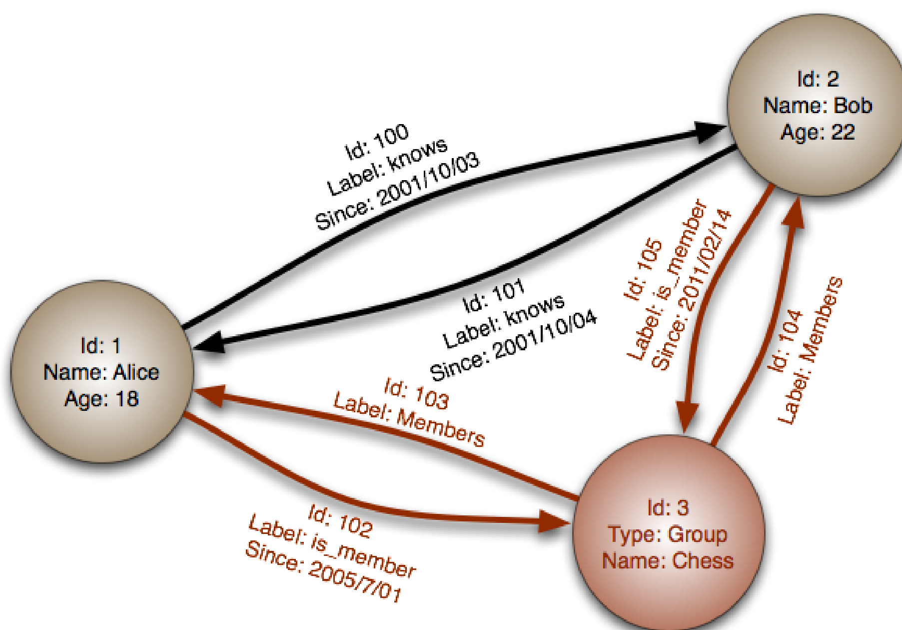
[4] Oficiální stránky MS SQL - https://www.microsoft.com/cs-cz/sql-server

Mezi nejznámější zástupce patří databáze HBase[5][4].

2.2.2 Grafové databáze

Grafové databáze jsou používány pro případy, ve kterých může existovat mnoho vazeb mezi entitami. Takové případy je velmi těžké, ne-li nemožné, vyřešit v relačních databázích. Použití grafové databáze je vhodné například pro sociální sítě [1].

V grafové databázi jsou entity prezentovány jako uzly a vztahy mezi entitami jsou reprezentovány jako hrany mezi uzly. Díky tomu je tyto databáze vhodné použít zejména pro hledání nejkratší cesty v grafu, což může být využito například pro navigační systémy [9].



Obrázek 1 - Grafová databáze

Mezi zástupce grafických databází patří OrientDb[6] [4].

[5] Oficiální stránky HBase - https://hbase.apache.org/
[6] Oficiální stránky OrientDB - http://orientdb.com/orientdb/

2.2.3 Databáze s klíčem a hodnotou

Databáze s klíčem a hodnotu je velmi jednoduchý datový model, ve kterém jsou uložené hodnoty pod unikátními klíči, ve formátu klíč – hodnota. Hlavní výhodou těchto databází je jejich rychlost. V těchto databázích neexistuje žádné schéma, a proto také tyto databáze potřebují oproti ostatním databázím mnohem méně místa na disku [10].

Mezi hlavní zástupce patří například Redis[7], který ukládá data v operační paměti, nebo také Cassandra[8] [4].

2.2.4 Dokumentově orientované databáze

Dokumentové databáze pracují s daty jako s dokumenty. Tyto databáze je možné považovat za komplexnější variantu databází s klíčem a hodnotou, jelikož každý dokument je tvořen skupinou párů klíč – hodnota [3].

Na rozdíl od relačních databází, kde je předem definovaná struktura tabulek a dat v nich, data v dokumentových databázích mají semi-strukturovaná data. To znamená, že každý dokument obsahuje kromě samotných dat také popis jejich struktury.

Z toho vyplývá, že každý dokument může mít odlišnou strukturu, oproti ostatním dokumentům ve stejné kolekci.

Bez pevně daného schématu je přidávání či odebírání jednotlivých atributů v dokumentech mnohem jednodušší. [13].

Dokumenty v sobě navíc mohou obsahovat vnořené dokumenty, což je další rozdíl oproti relačním databázím, kde je nutné hierarchii dat rozložit mezi více záznamů a propojit je pomocí cizích klíčů [13].

Mezi nejpopulárnější dokumentové databáze patří MongoDB, CouchDB[9] nebo RavenDB[10] [4].

---

[7] Oficiální stránky Redis - https://redis.io/
[8] Oficiální stránky Cassandra - http://cassandra.apache.org/
[9] Oficiální stránky CouchDB - http://couchdb.apache.org/
[10] Oficiální stránky RavenDB - https://ravendb.net/

2.2.4.1 MongoDB

MongoDB je dokumentová databáze, ve které jsou dokumenty drženy v kolekcích. Kolekci je možné přirovnat k tabulce v relační databázi, kde dokumenty odpovídají řádkům v tabulce. V těchto kolekcích mohou být drženy libovolné dokumenty. Je ale doporučené, aby měli podobnou strukturu, pro efektivní indexování [12].

Hlavní výhodou MongoDB je, že i bez použití schématu umí s dokumenty zacházet podobně, jako zacházejí relační databáze s entitami. Díky tomu je možné jednoduché vytváření dotazů [12].

Každý dokument musí mít v rámci své kolekce jedinečný identifikátor, který je možné zadat při vkládání dokumentu anebo ho MongoDB přiřadí sám v podobě hexadecimálního čísla o délce 24 znaků. Dokumenty jsou ukládány ve formátu BSON, což je binárně kódovaná reprezentace JSON formátu, ve které jsou vnořené sady klíčů a hodnot. Výhodou formátu BSON je, že podporuje regulární výrazy, binární data a datumy [12].

Další charakteristikou MongoDB je rychlost a snadná škálovatelnost, jelikož nepodporuje některé vlastnosti relačních databází, například transakce. Z toho důvodu není MongoDB vhodnou volbou pro aplikace, pro které je podpora transakcí na více objektech důležitá. [13].

MongoDB byla k datu 25.01.2018 nejpoužívanější NoSQL databází [4].

3 Škálovatelnost, replikace a zálohování

3.1 *Škálovatelnost*

Škálovatelnost je schopnost databáze měnit výkonost v závislosti na aktuální potřebě. Špatně škálovatelné systémy dosahují nižší výkonosti. Škálovatelnost je často označována jako synonymum termínů výkonost a kapacita, avšak tyto termíny se od navzájem sebe liší. Kapacita je definována jako množství operací, které je databáze schopná provádět ve stejném čase a výkonost lze z databázového pohledu definovat jako čas odpovědi (čím větší výkonost, tím rychleji dokáže databáze odpovědět na daný dotaz). [14].

Kapacita a škálování jsou nezávislé na výkonosti, což lze vysvětlit pomocí analogie s dopravou na dálnici:

- Výkonost znamená, jaké rychlosti jsou jednotlivé vozy schopné dosáhnout.
- Kapacita je maximální bezpečná rychlost jízdy.
- Škálování je míra, do které je možné přidat více aut bez zpomalení dopravy.

Rozlišují se dva druhy škálování – horizontální a vertikální [14].

3.1.1 Horizontální škálování

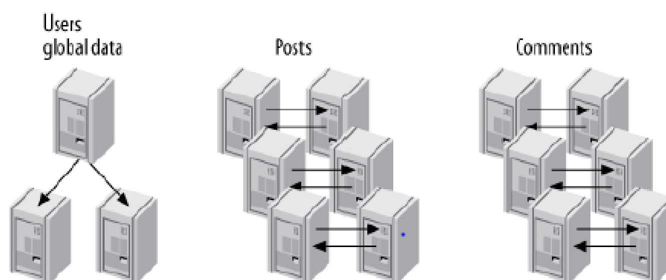
Horizontální škálování znamená zvyšování výkonosti databáze přidáváním dalších serverů. Výhodou tohoto způsobu je, že pořízení nových serverů může být finančně méně náročné, jelikož k tomuto účelu postačí levný hardware. Nevýhodou však je obtíženější spravování databáze, která je rozložená přes více serverů [14].

Nejjednodušší a nejběžnější způsob škálování je použití replikace, kde jsou slave servery použity k čtení dat. Tento způsob může fungovat velmi dobře pro aplikace s vysokým počtem čtecích dotazů.

MySQL i MongoDB podporují horizontální škálování metodou sharding, pomocí které jsou data rozdělena na menší části, uložených na více serverech. Data jsou rozdělována podle klíče, který je definován při vytváření shardingu.

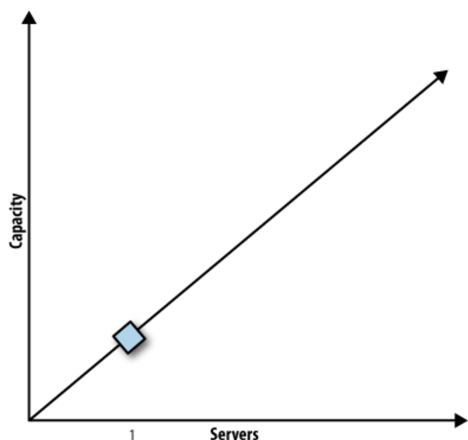
Většina velkých databázových systémů aplikuje sharding pouze na tabulky či kolekce, které obsahují větší objem dat. Menší tabulky či kolekce pak bývají uložené na samostatných serverech [14].

Příkladem může být databázový systém sociální sítě. Tato sociální síť může mít tisíce uživatelů, kteří mohou každý den vytvářet nové příspěvky či psát komentáře pod jednotlivé příspěvky. V takovémto případě je vhodné logicky rozvrhnout databázi. Neexistuje však žádné řešení, které by se dalo označit jako jediné správné. Databázi lze například rozvrhnout oddělením uživatelských informací, příspěvků uživatelů a komentářů. Tabulky s uživatelskými daty nemusí být rozděleny pomocí metody sharding, pokud se neočekává vysoký počet uživatelů v řádech sta milionů. Je však vhodné použít master-slave replikaci a přesměrovat čtecí dotazy na slave servery za účelem snížení zátěže master serveru. Jelikož uživatel může vytvořit desítky příspěvků denně, je nutné rozdělit příspěvky metodou sharding, Vhodným rozdělovacím identifikátorem může být identifikátor uživatele. Jednotlivé příspěvky můžou mít tisíce komentářů, a proto je též vhodné aplikovat sharding, v tomto případě za pomocí identifikátoru příspěvku, pod který patří.

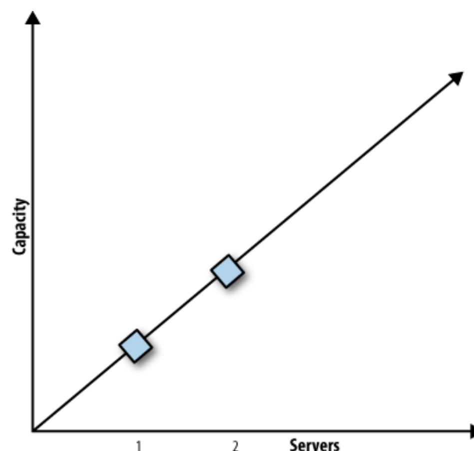


Obrázek 2 - Ukázka Škálování [14]

Na následujícím příkladu je možné vidět, jak se mění škálovatelnost při přidávání serverů. Přidáním jednoho serveru se zvýší dostupná kapacita databáze. Škálování na příkladu níže je nazýváno lineárním, jelikož byl dvojnásobně zvětšen, jak počet serverů, tak i kapacita.

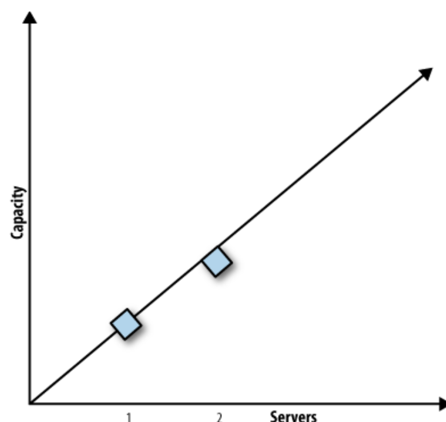


Obrázek 3 - Škálování – systém s jedním serverem [14]



Obrázek 4 - Škálování – lineární škálování [14]

Většina systémů však není lineárně škálovatelná, jelikož přidáním serveru nemusí být dosaženo dané kapacity a tím vzniká lineární odchylka. Dokonce se databáze může dostat do stavu, kdy se již nevyplatí přidávat další zdroje [14].



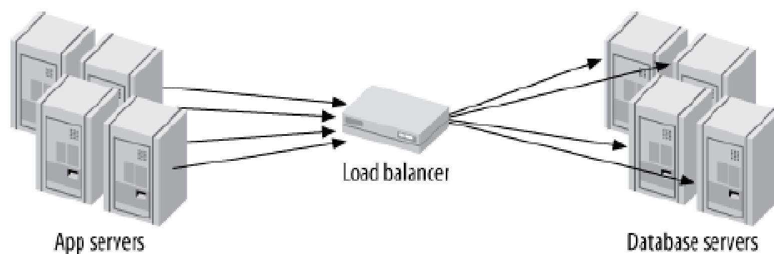
Obrázek 5 - Škálování – nelineární škálování [14]

3.1.1.1 Horizontální škálování MySQL

Horizontální škálování MySQL je provedeno metodou sharding, která pomocí klíče rozdělí danou tabulku na partitions a jednotlivé partitions umístí na jednotlivé servery. To znamená, že jednotlivé servery mají samostatné databáze, které zpravují

svoji vlastní část dat. Toto oddělení dat umožňuje aplikaci distribuovat dotazy na více serverů současně a vytvářet paralelní dotazy.

Jako rozhraní mezi databázovými servery a aplikací slouží tzv. load balancer [14].



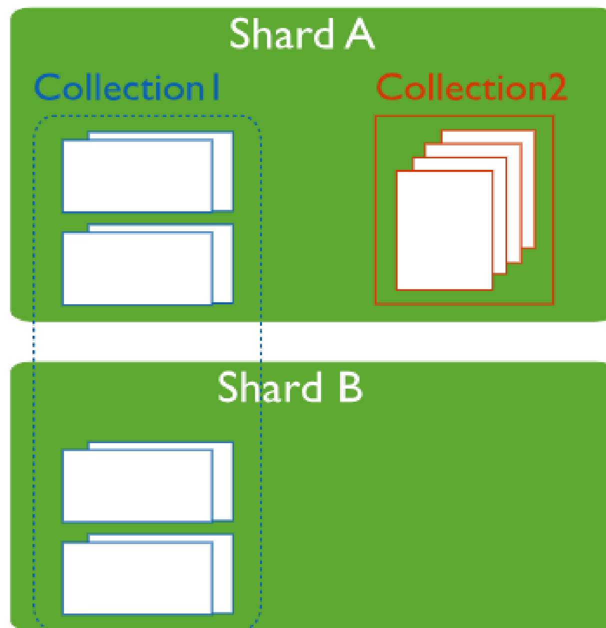
Obrázek 6 - MySQL Load balancer [14]

3.1.1.2 Horizontální škálování MongoDB

Stejně jako MySQL, i MongoDB podporuje horizontální škálování pomocí metody sharding, která vytvoří tzv. sharded cluster. Sharding rozděluje kolekce automaticky na menší části neboli chunks, které se pak samostatně spravují.

Sharder cluster se skládá z několika komponent:

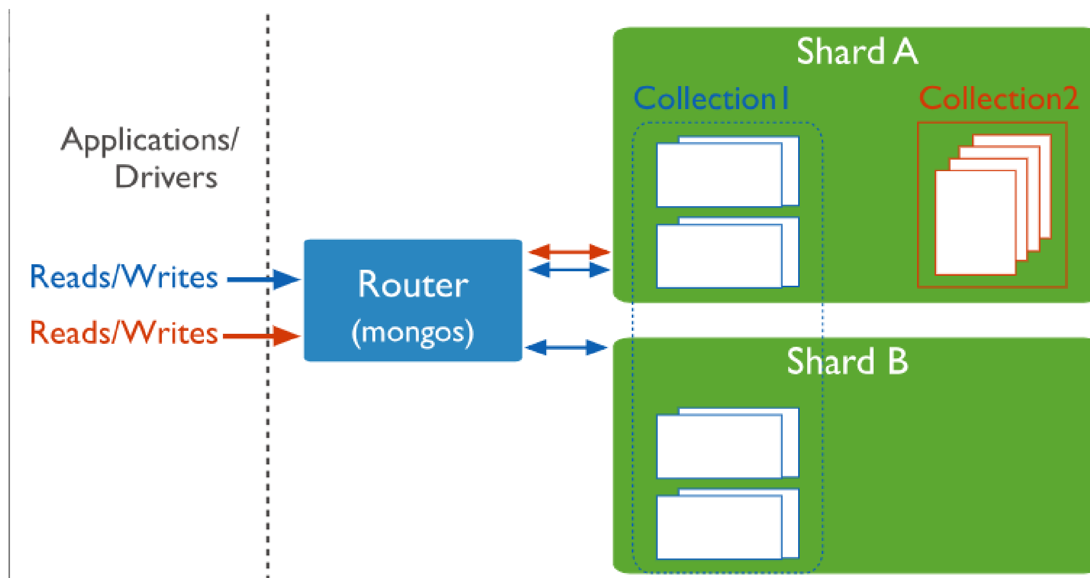
- Shard – Jedná se o samostatné servery, které obsahují jednotlivé části dat (chunks).



Obrázek 7 - Škálování – příklad MongoDB shardů [15]

- Mongos – Proces, který poskytuje rozhraní mezi aplikací a sharderd cluster.
- Konfigurační server – Servery, které ukládají informace o tom, jaká data jednotlivé shardy drží.

Na následujícím obrázku je možné vidět, jak funguje interakce jednotlivých komponent v rámci sharded clusteru.



Obrázek 8 - Škálování – Sharderd Cluster [15]

3.1.2 Vertikální škálování

Vertikální škálování znamená navyšování výkonu fyzického serveru, na kterém daná databáze běží. Výhodou vertikálního škálování je fakt, že správa jediného serveru je mnohem jednodušší než správa několika serverů. Například provedení záloh a obnov tzv. „single server“ databázového systému je mnohem jednodušší. Nevýhodou však může být cena hardware potřebného pro navýšení výkonu [14].

3.2 *Replikace*

Replikace představuje způsob udržování několika identických kopií databáze a je doporučené ji používat na všech produkčních databázích.

Existuje několik způsobů replikace, mezi nejpoužívanější patří principy master-slave a master-master.

Master-slave je způsob replikace, kde jeden databázový server je označený jako master (hlavní) a ostatní jako slave (podřazené). Veškeré změny, které jsou provedeny na master databázi jsou v rámci replikace propsány na všechny databáze označené jako slave. Slave databáze bývá často označován jako replika.

Díky replikaci je možné přesměrovat některé čtecí dotazy do repliky, což umožňuje výrazně snížit zátěž master databáze. Zároveň je to také skvělý způsob zálohování dat pro případ nedostupnosti master databáze [14].

Master-master neboli multi-master replikace umožňuje zapisovat data skrze libovolný databázový server a zajišťuje konzistentnost dat skrze všechny servery. Každý server je označen jako master (hlavní) pro určitou část dat a jako jediný může tyto data modifikovat. Ostatní servery mohou tyto data číst, ale modifikační příkazy přeposílají na master server.

Pří porovnání master-master replikace s master-slave replikací je hlavní výhodou fakt, že pokud jeden ze serverů přestane odpovídat, veškeré dotazy mohou být přesměrovány na jiný server. Dále také master-master replikace disponuje vyšší dostupností dat a rychlejší odezvou serverů. Hlavní nevýhodou však je, že multi-master replikační systémy s větším množstvím serverů mohou být nekonzistentní,

což je v rozporu s vlastností ACID. Další nevýhodou je složitost databázového systému, jelikož s přibývajícím množstvím serverů může být zvýšena latence.

3.2.1 Replikace MySQL

MySQL podporuje dva způsoby replikace – statement-based replication a row-based replication. Oba tyto způsoby zapisují veškeré změny provedené na master databázi do tzv. master binary logu. Tyto změny jsou poté zkopírovány replikou do tzv. relay logu, odkud jsou změny aplikovány přímo do repliky. Zápis do master binary logu a relay logu probíhá asynchronně, což znamená, že replika nemusí být v daný moment stoprocentně synchronizovaná s master databází. Neexistuje také žádná záruka, jak velké může být zpoždění repliky, jelikož náročnější dotazy mohou replikaci zpozdit v rámci sekund, minut nebo dokonce i hodin [14].

Statement-based replication je způsob replikace, který ukládá do master binary logu příkazy, které byly v master databázi provedeny. Tyto příkazy jsou poté ve stejném pořadí provedeny i nad replikou. Výhodou tohoto způsobu je, že velikost binary logu neroste tak rychle, jak u druhého způsobu. Také je možné, aby tabulka v replice neměla úplně identické schéma jako v master databázi, pouze musí obsahovat všechny sloupce, která jsou v tabulce v master databázi. Nevýhodou tohoto způsobu však je, že výsledky jednotlivých příkazů můžou být odlišné na master databázi a na slave databázi, obzvlášť pokud je použit v příkazu limit bez pořadí [14].

Row-based replication ukládá do master binary logu místo příkazů přímo změněná data, která jsou následovně aktualizována v replice. Tento způsob zajišťuje, že jsou data v master i slave databázích skutečně konzistentní a je tím pádem bezpečnější. Dále pak při provádění příkazů z relay logu není nutné zamykat tolik řádků – zamykají se pouze řádky, které jsou skutečně aktualizovány. Nevýhodou však je, že schémata tabulek v master databázi i replice musí být totožná [14].

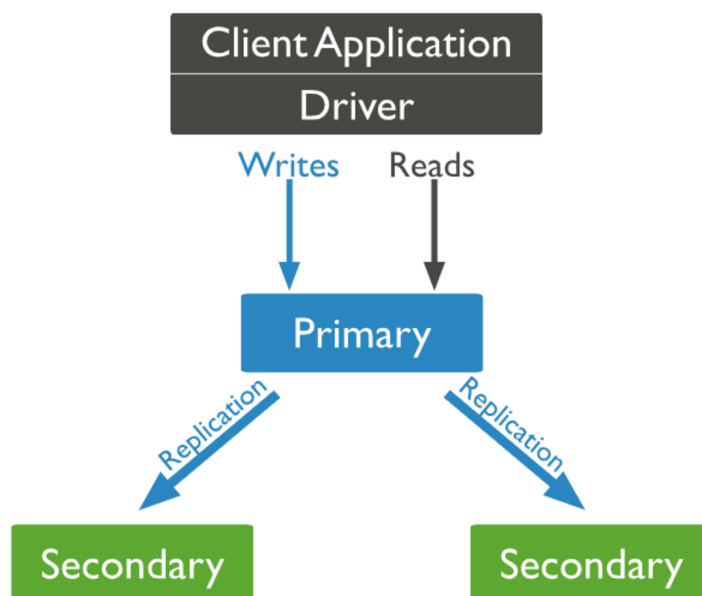
3.2.2 Replikace MongoDB

MongoDB používá za účelem replikace tzv. replica set, což je skupina databázových serverů, kde je jeden označen jako primární (master) a ostatní jako sekundární (slave). Replica set může být také rozšířen o další server, kterému se přezdívá arbiter. Arbiter je databázový server, jehož účelem není držení dat, ale výběr nového

primárního serveru v případě, že aktuální primární server není dostupný či neodpovídá.

V replica setu je možné zapisovat pouze do primárního databázového serveru. Ze sekundárních databázových serverů je možné pouze číst [13].

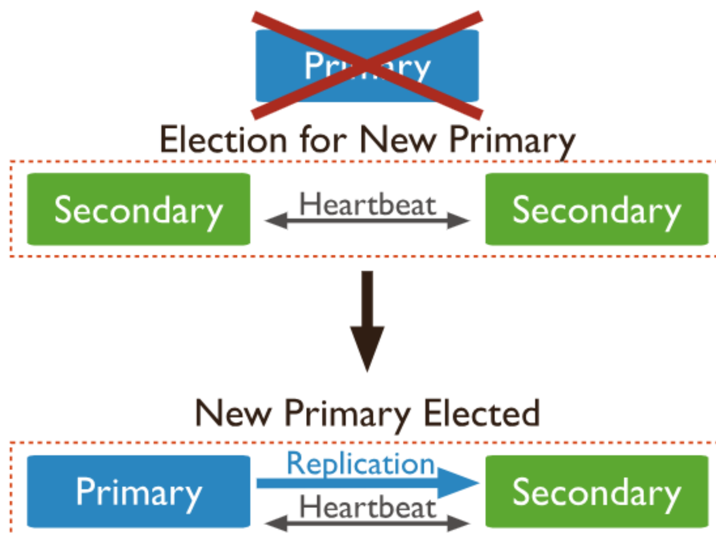
Replikace probíhá tak, že primární server provádí operace, které mění data v databázi a veškeré změny zapisuje do oplogu (operační log), což je speciální kolekce, kde jsou drženy záznamy o všech provedených operacích, které afektovaly data. Sekundární servery si stahují změny zapsané v oplogu a aplikují je na svá data tak, aby byly konzistentní s daty v primárním serveru. Každá operace v oplogu je idempotentní, což znamená, že daná operace bude mít stejný výsledek ve všech databázových serverech.



Obrázek 9 - MongoDB replikace [15]

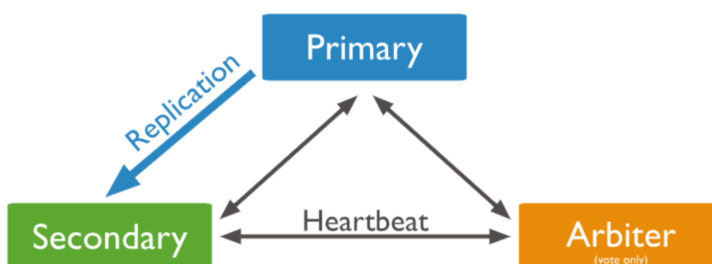
Oplog je automaticky vytvořen při startu databáze a má výchozí velikost 5 % disku, na kterém se databáze nachází, což je ve většině případů dostačující velikost. To znamená, že pokud se oplog dokáže zaplnit během 24 hodin, sekundární servery mají 24 hodin na stažení dat z oplogu než se stanou příliš zastaralými, aby mohli pokračovat v replikaci. Velikost oplogu je konfigurovatelná při startu databáze.

Replica set poskytuje výhodu v tom, že sám dokáže nahradit neodpovídající primární server jiným. Pokud primární server z jakéhokoliv důvodu přestane odpovídat, je v replica setu mezi sekundárními servery zvolen server, který se stane novým primárním serverem. Sekundární servery si v rámci replica setu posílají tzv. heartbeat neboli ping a na základě toho si mezi sebou zvolí nový primární server [15].



Obrázek 10 - Volba nového primárního MongoDB serveru bez arbitera [15]

V případě, že replica set obsahuje arbitera, je nový server zvolen arbiterem, který kontroluje dostupné servery a vybere z nich právě jeden nový primární. Použití arbitera pro volbu nového serveru je efektivnější, jelikož arbiter neobsahuje žádná data v databázi, a tudíž nemá tak vysoké hardwarové nároky [15].



Obrázek 11 - Volba nového primárního MongoDB serveru s arbiterem [15]

3.3 *Zálohování*

Zálohování je ochranou před selháním databáze a je dobré vytvářet je z následujících důvodů:

- Obnovení databáze po havárii
    - V případě selhání hardwaru, poškození dat či poškození serveru je možné obnovit databázi z předem vytvořené zálohy.
- Audit
    - Často je z důvodů zpětného zkoumání výsledků operace potřeba zjistit, v jakém stavu se databáze nacházela před provedením dané operace.
- Testování
    - Testování aplikace nad zálohou produkční databáze je nejjednodušší cesta, jak otestovat např. opravu v kódu aplikace či odhalit další chyby v aplikaci.

Zálohování se rozděluje na dva hlavní typy:

- Logická záloha
    - Logická záloha je záloha vytvořená pomocí nástroje dané databáze. Data jsou většinou uložena ve formě příkazů, které jsou v případě obnovení databáze spouštěny.
    - Výhody:
        - Jejich obnova pomocí nástrojů dané databáze bývá jednoduchá.
        - Vytvořit zálohu či obnovit zálohu je možné z jiného serveru v síti.
        - Vytvoření logické zálohy je flexibilní – nástroje databází umožnují vytvořit zálohu pouze dané tabulky (či kolekce) nebo dat v ní.
        - Obnova dat nevyžaduje restart databáze.
    - Nevýhody:
        - Vytváří je databázový server, což může ovlivnit výkon CPU.

- Hrubá záloha
    - Za hrubou zálohu je považováno zálohování složek a souborů, které databáze využívá.
    - Výhody:
        - Hrubá záloha představuje pouze provedení kopie vybraných souborů a složek na jiné místo, což pro databázový server nepředstavuje žádnou práci.
        - Obnovení hrubé zálohy bývá rychlejší, jelikož databáze neprovádí žádné příkazy.
    - Nevýhody:
        - Obnovení hrubé zálohy vyžaduje odstávku databáze.
        - Hrubá záloha často zabere více místa na disku než logická záloha.
        - Jelikož není možné zkopírovat všechny soubory v jeden daný moment, je potřeba zajistit, aby se data po dobu zálohy nezměnila.

Nejčastějším způsobem zálohování MySQL dat je použití mysqsldump, což je klientské rozhraní pro logické zálohování. Při záloze MySQL tabulky vytvoří mysqldump soubor, který obsahuje strukturu tabulky a zároveň data ve formátu validních SQL příkazů.
Zálohování MongoDB dat je nejčastěji prováděno za použití mongodump, což je klientské rozhraní pro logické zálohování. Mongodump vytvoří složku „dump“, ve které jsou zálohovaná data rozdělena dle databáze, do které patří. Data samotná jsou zálohována ve formátu bson.

4 Jak navrhovat strukturu databází

4.1 *Jak navrhovat strukturu MySQL*

Základem výkonné MySQL databáze je dobrý logický a fyzický návrh databáze.

MySQL podporuje velké množství datových typů a pro dosažení vysoké výkonosti databáze je volba nejvhodnějšího datového typu zásadní [14].

Prvním krokem při volbě datového typu je obecná definice ukládaní dat, tzn. budou-li v daném sloupci ukládána čísla, text, časové údaje apod [14].

Dalším krokem je výběr konkrétního datového typu. Mnoho datových typů může ukládat stejný druh dat, ale liší se navzájem v rozsahu hodnot, které mohou ukládat, přesnosti, kterou dovolují nebo v náročnosti na fyzické místo disku. Některé datové typy také můžou mít speciální chování či vlastnosti [14].

Příkladem jsou datové typy DateTime a TimeStamp, které ukládají ten samý druh dat – datum a čas s přesností na sekundy. TimeStamp oproti DateTime využívá pouze poloviční množství místa na disku a drží informaci o časové zóně.

Na druhou stranu má však mnohem menší rozsah povolených hodnot.

Důležitou volbou je volba datového typu primárního klíče tabulky. Je totiž velmi pravděpodobné, že primární klíč dané tabulky bude použit v jiné tabulce jako cizí klíč. Je důležité zvážit nejen způsob ukládání dat, ale také způsob, jakým MySQL provádí výpočty a porovnávání daného datového typu. Například datové typy Enum a Set jsou ukládány interně jako celá čísla, které jsou převáděny do textových hodnot při používání [14].

Je vhodné používat datový typ s co nejmenším rozsahem. Například pro tabulku se státy Evropské Unie je vhodnější pro číselný primární klíč upřednostnit datový typ TinyInt před Int, jelikož předem víme, že tabulka nebude obsahovat tisíce záznamů a TinyInt je velikostně o 3 bity menší, což může být velký rozdíl, pokud bude hodnota použita v jiné tabulce jako cizí klíč [14].

4.1.1 Normalizace MySQL databáze

Normalizace je proces rozkládání relací za účelem jednoduší manipulace s daty a zamezení redundance dat. V normalizované databázi je každý záznam přítomen pouze jednou, naopak denormalizovaná databáze obsahuje duplikované záznamy. Normalizace není deterministická, což znamená, že existuje více než jedno řešení, jak normalizovat relaci [14].

Hlavní výhodou normalizace je rychlejší provedení aktualizací v tabulce. Pokud je tabulka řádně normalizovaná, tak obsahuje minimum duplicitních dat či žádná duplicitní data, což v případě změny znamená minimální množství aktualizací v databázi. Normalizované tabulky jsou obvykle menší, a tudíž se také rychleji načítají do paměti. Nižší počet redundantních dat také může snížit potřebu složitějších dotazů, jako jsou DISTINCT či GROUP BY.

Nevýhodou normalizovaného schématu však může být netriviální vyhledávání dat skrze více tabulek [14].

Při normalizaci se používají normalizační formy, což jsou pravidla, která data v relaci musí splňovat. Čím větší je číslo normalizační formy, tím jednodušší by měla být práce s daty. Normalizačních forem se používá celá řada, avšak nejpoužívanější jsou tyto tři normalizační formy:

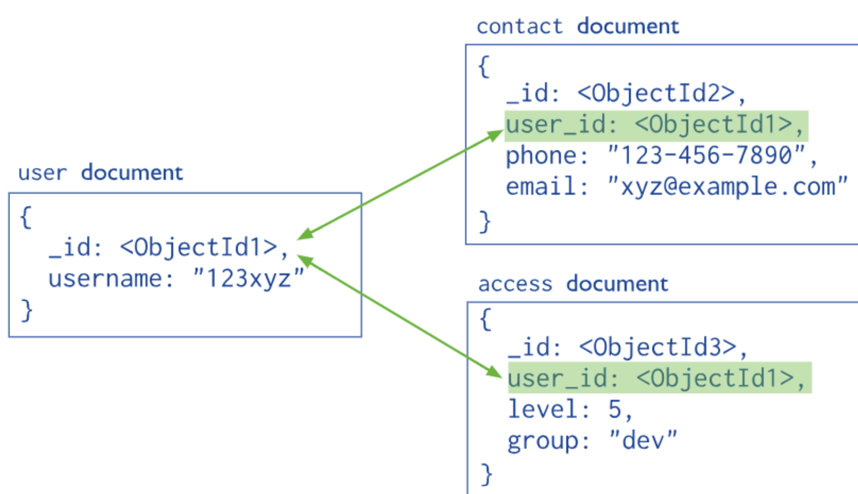
- První normalizační forma
  - První normalizační formy je dosaženo v případě, že je každý atribut v relaci atomický.
- Druhá normalizační forma
  - Pokud je relace v první normalizační formě a každý neklíčový atribut je úplně funkčně závislý na primárním klíči, jedná se o druhou normalizační formu.
- Třetí normalizační forma
  - Třetí normalizační formy je dosaženo v případě, že relace je v druhé normalizační formě a zároveň neobsahuje tranzitivní závislosti, což znamená, že neklíčové atributy jsou vzájemně nezávislé.

4.2 *Jak navrhovat strukturu MongoDB*

Na rozdíl od MySQL dokumenty v MongoDB kolekci nemusí mít stejnou strukturu.

Klíčové rozhodnutím při navrhování MongoDB datových modelů se točí kolem způsobu, jakým aplikace představuje vztahy mezi daty. MongoDB umožnuje následující možnosti ukládání relací:

- Odkazy do jiných kolekcí
  - Odkazy ukládají vztahy mezi daty pomocí odkazů na dokumenty z jiných kolekcí, což umožňuje návrh normalizovaného datového modelu.



Obrázek 12 - Relace pomocí odkazu [15]

- Vnořená data
  - Vnořené dokumenty zachycují vztahy mezi daty ukládáním souvisejících dat do jednoho dokumentu. V tomto případě se jedná o denormalizovaný datový model.

Obrázek 13 - Vnořená data [15]

4.3 *Indexy*

Indexy jsou kritickou součástí databází s vysokou dostupností dat a jejich důležitost roste s přibývající velikostí databáze. Význam indexů může být popsán pomocí analogie s rejstříkem knihy. Pokud čtenář hledá v knize pojem, podívá se do rejstříku, ve kterém tento pojem najde mnohem rychleji, než kdyby pročítal celou knížku řádek po řádku. Na stejném principu fungují indexy.

Index je datová struktura, kterou databáze využívá pro rychlejší hledání dat, avšak na úkor rychlosti vkládání, mazání či úpravy dat, jelikož jakákoliv manipulace s daty zahrnuje i manipulaci s indexem. To znamená, že s přibývajícím počtem indexů může být zásadně ovlivněna výkonost databáze. Z pohledu relačních databází mají indexy přiřazený jeden či více sloupců v dané tabulce, z pohledu dokumentových databází mají indexy přiřazený jeden či více parametrů dokumentů v dané kolekci. Indexy jsou oddělené fyziky i logicky od dat tabulky či kolekce, a proto je možné indexy vytvářet či indexy bez jakékoliv změny dat v databázi.

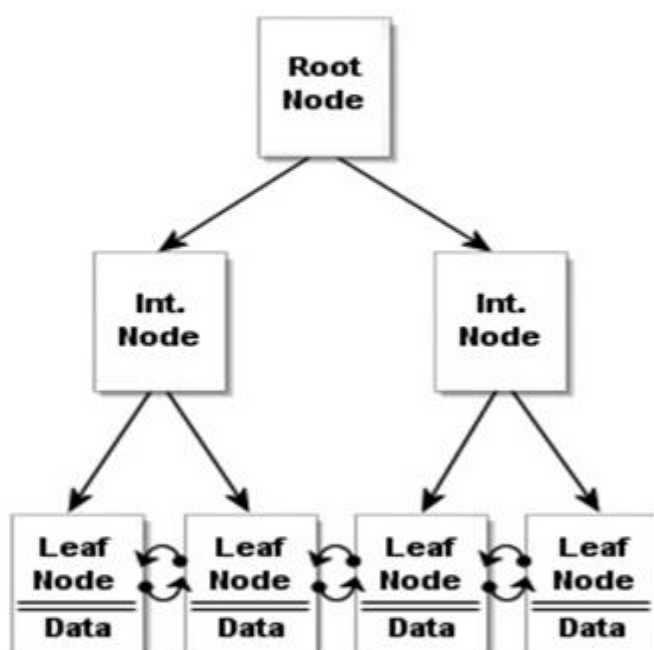
4.3.1 MySQL indexy

4.3.1.1 B-Tree index

B-Tree index je nejčastěji používaným typem indexů. Tento typ indexu je vhodné použít pro tabulky s velkým počtem dat, velkou kardinalitou dat a nízkou selektivitou dat.

Pojem kardinalita dat označuje různorodost dat v tabulce. Příkladem nízké kardinality může být pohlaví, kde se rozlišují pouze dvě hodnoty (muž a žena) a příkladem vysoké kardinality může být primární klíč tabulky. Selektivita značí, kolik procent dat je vráceno pro jednu konkrétní hodnotu s ohledem na celkový počet dat v tabulce. Z toho vyplývá, že čím větší je kardinalita dat, tím menší je selektivita dat. Pro dosažení co největší efektivity B-Tree indexu by neměla selektivita daného sloupce být větší než 10 %.

B-Tree index je založen na principu b-stromu, což je struktura skládající se ze tří částí – kořen (root), uzel (node) a list (leaf).



Obrázek 14 - B-Tree index [16]

Kořen stromu i uzly mohou obsahovat větší počet hodnot, které jsou seřazené. Ve stromové struktuře platí, že pokud má kořen (root) n hodnot, pak má n+1 vazeb s uzly pod sebou. To samé platí i v případě uzlů a jejich vazeb na podřízené listy. Hodnoty v uzlech a listech jsou seřazené od nejmenší hodnoty po největší. Listy pak také navíc odkazují odkazy na primární klíče řádků, ke kterým daná hodnota patří.

Při hledání konkrétní hodnoty je strom procházen od kořene (root) a porovnávají se jednotlivé hodnoty s hodnotou hledanou. Při dosažení nejnižší část stromu (list) je hledaná hodnota nalezena (pokud ovšem existuje).

B-Tree index může být vytvořen nad jedním či více sloupci. Při tvorbě indexu záleží na tom, v jakém pořadí jsou sloupce v indexu definovány, a to z důvodu řazení dat v indexu. Příkladem může být tabulka osob, nad kterou bude vytvořen index nad sloupci, ve kterých jsou uložená jména a příjmení osob.

CREATE INDEX osobaIndex ON osoba (prijmeni, jmeno);

V případě spuštění příkazu níže je index úspěšně použit.

*SELECT * FROM osoba WHERE prijmeni = 'Novak' and jmeno = 'Jan';*

Index bude použit i v případě následujícího příkazu, a to i přes to, že dotaz obsahuje pouze specifikované příjmení. Důvodem je pořadí sloupců v indexu. Jelikož je příjmení uvedeno jako první v indexu, data jsou v indexu seřazena nejprve podle příjmení a až poté podle jména, a tudíž je možné hledat data ve stromu jen pomocí příjmení.

*SELECT * FROM osoba WHERE prijmeni = 'Novak';*

Avšak pokud by byl spuštěn následující příkaz, kde je specifikované pouze jméno, index nebude použit, a to z toho důvodu, že data v indexu jsou seřazená nejprve podle příjmení a až pak podle jména.

*SELECT * FROM osoba WHERE jmeno = 'Jan';*

4.3.1.2 **Hash index**

Hash index je určen pro dotazy, které filtrují data pomocí všech sloupců indexu. Pro filtrování dat nevyužívá strom, nýbrž hash tabulku. Pro každý řádek tabulky, nad kterou je index založen, je vytvořen hash kód, který je uložen v hash tabulce. V hash tabulce je také uložen ukazatel na primární klíč řádku, na který daný hash kód ukazuje. V případě, že se hash kód několika řádků shoduje, index uloží všechny ukazatele k jednomu hash kódu.

Při spuštění dotazu, který má v klauzuli WHERE specifikované hodnoty všech sloupců obsažených v hash indexu, je vytvořen hash kód. Tento hash kód se skládá pouze ze sloupců, které jsou obsažené v indexu, a to takovém pořadí, v jakém jsou tyto sloupce definované v indexu. Následně je tento hash kód vyhledán v hash tabulce. Pokud je hash kód v hash tabulce nalezen, jsou navráceny všechny řádky,

na které daný hash kód ukazuje. Po navrácení řádků jsou nakonec všechny hodnoty zkontrolovány bez použití hash kódu, aby bylo jisté, že dotaz vrací hledaná data.

Hash indexy v MySQL mají následující limitace:

- Řazení dat pomocí hash indexu není podporováno.
- Hash index není možné použít při spuštění dotazu, ve kterém nejsou specifikované hodnoty všech sloupců, ze kterých se daný index skládá.
- Hash index podporuje pouze dotazy, ve kterých jsou přesně definované hodnoty všech sloupců indexu pomocí operátorů '=', '<=>' nebo pomocí funkce IN().

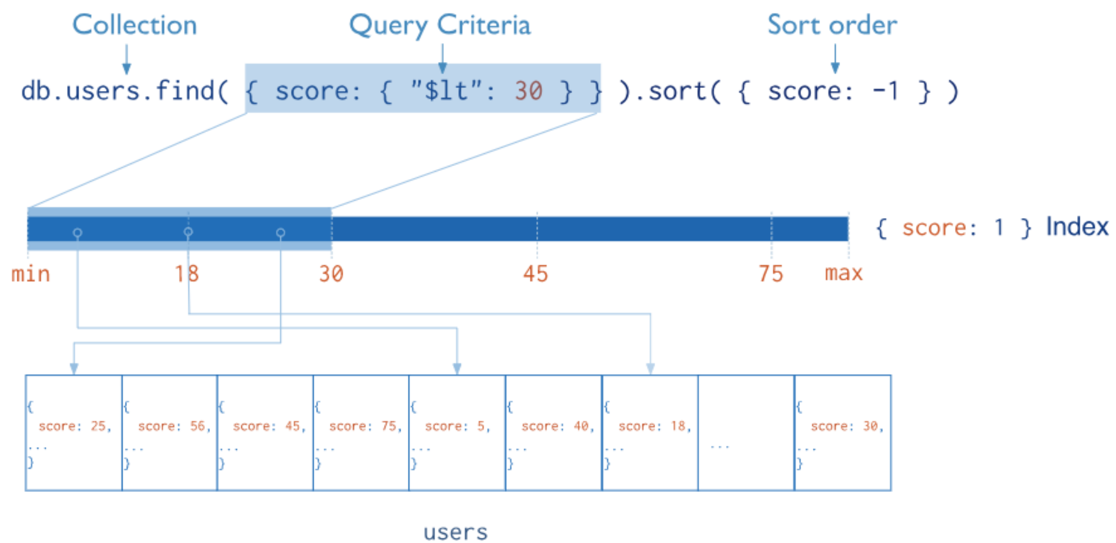
4.3.2 MongoDB indexy

Indexy v MongoDB ukládají hodnoty určitého atributu nebo sady atributů, které mohou být seřazené od největšího po nejmenší, či naopak.

Index lze vytvořit pomocí příkazu níže. Parametr *atribut1* definuje, nad jakým atributem je index vytvořen. U toho atributu je dále použito číslo 1 nebo -1, což definuje, zdali mají být data indexu seřazena od nejmenšího po největší (1), či od největšího po nejmenší (-1).

db.kolekce.createIndex({ atribut1: 1 })

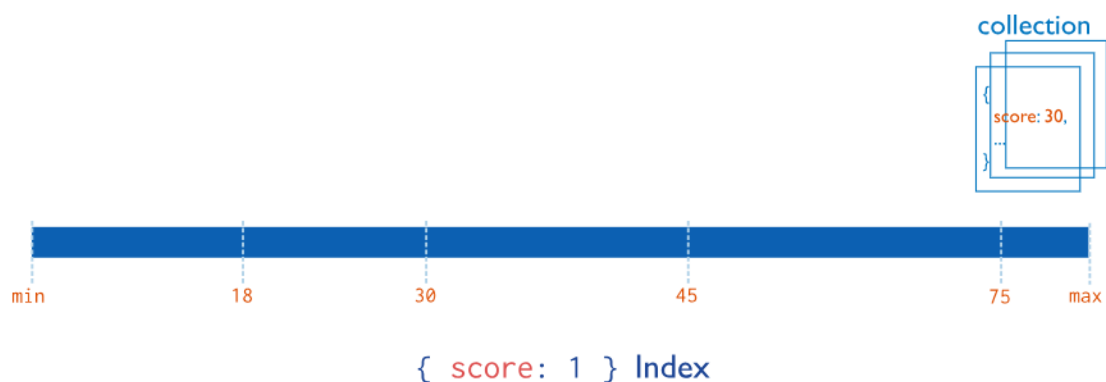
MongoDB indexy fungují obdobně jako indexy v MySQL. Při založení indexu je vytvořena datová struktura, ve které jsou drženy indexované hodnoty a odkazy na konkrétní dokumenty v dané kolekci.



Obrázek 15 - MognoDB index [15]

4.3.2.1 Single Field Index

MongoDB poskytuje podporu pro indexování pro jakýkoliv atribut dokumentu v dané kolekci. Příkladem Single Field indexu je unikátní index nad atributem _id, který je na základě výchozího nastavení MongoDB ve všech kolekcích. Tímto unikátním indexem je zajištěno, že se v dané kolekci nebudou vyskytovat dokumenty s duplicitním identifikátorem. [15]



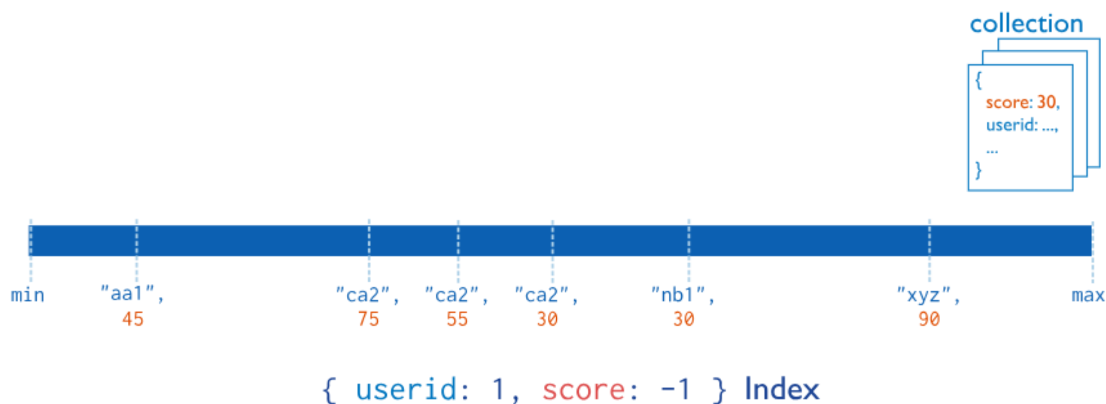
Obrázek 16 - Single field index [15]

4.3.2.2 Compound index

Compound neboli složený index se skládá z několika atributů, které společně odkazují na jednotlivé dokumenty v kolekci.

Tento index je možné vytvořit pomocí příkazu níže. Je nutné zdůraznit, že záleží na pořadí, v jakém jsou atributy definovány. Vytvořený index bude totiž obsahovat reference na dokumenty, které budou seřazeny nejprve podle prvního atributu a až poté podle druhého atributu. Z tohoto důvodu nebude možné index použít, pokud budou data filtrována pomocí druhého atributu. [15]

db.kolekce.createIndex({ "atribut1": 1, " atribut2": 1 })

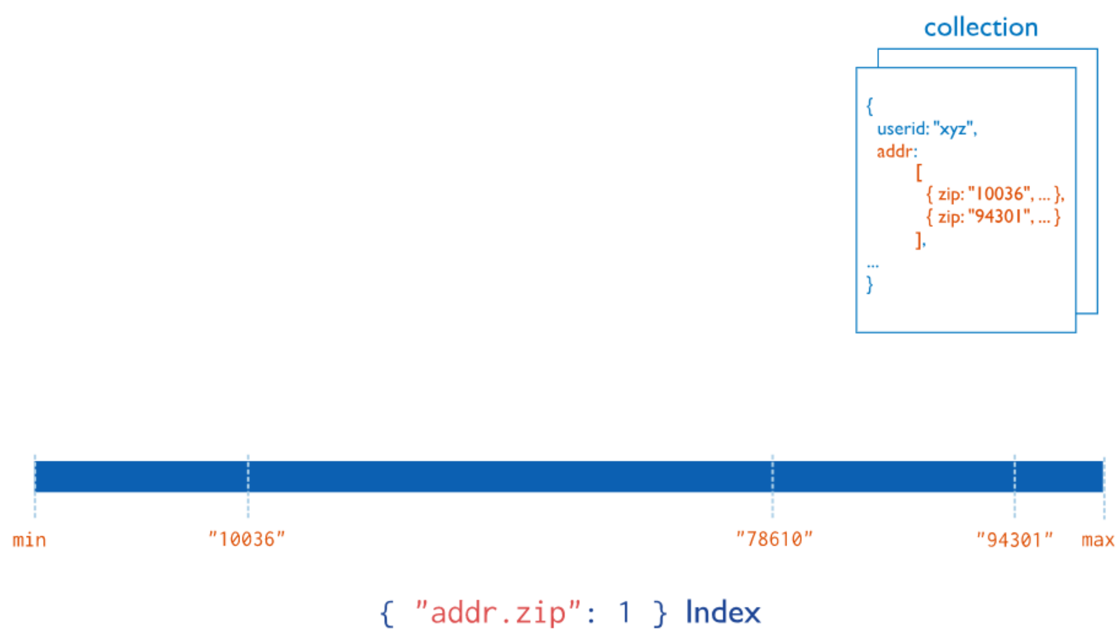


Obrázek 17 – Compound index [15]

4.3.2.3 Multikey index

Multikey index je používaný nad v MongoDB nad polem, kde MongoDB vytváří indexový klíč pro každý prvek v poli. Tento typ indexu je používán pro vyhledávání konkrétních elementů pole.

Tento typ indexu je oproti ostatním výrazně pomalejší, jelikož mnoho elementů pole může ukazovat na jeden dokument, a tudíž musí MongoDB odmazávat duplicitní dokumenty před vrácením dat.



Obrázek 18 – Multikey index [15]

5 Manipulace se strukturou a daty v živém provozu

Manipulace se strukturou nebo daty v živém provozu obnáší modifikaci dat či struktury tabulky v databázi, nad kterou běží nějaká aplikace. Jedná se tudíž o operace, které vyžadují obezřetnost, jelikož například aplikace nemusí být připravená na smazání sloupce v MySQL databázi apod.

5.1 *MySQL – Přidání sloupce*

První řešenou úlohou bylo přidání sloupce do již existující tabulky v živé produkční databázi. Aplikaci, která s touto databází pracuje, nebylo možné odstavit na více než hodinu. Cílová tabulka měla přes 300 000 000 řádků a byly do ní zapisovány stovky řádků za minutu. Databáze, ve které se cílová tabulka nachází byla replikována.

5.1.1 Výběr způsobu vložení nového sloupce

Navzdory tomu, že MySQL podporuje změnu schématu tabulky bez zamčení databáze, byl k vyřešení úlohy použit nástroj Percona Toolkit[11]. Důvodem je, že Percona poskytuje dodatečné funkčnosti a možnosti.

MySQL i Percona Toolkit přidávají sloupeček do tabulky stejným způsobem. Vytvoří se kopie tabulky, která je zmodifikována a poté se do ní postupně vkládají data, která byla vložena do původní tabulky od začátku přidávání sloupce. Po vložení všech dat jsou obě tabulky zamčeny a vyměněny.

Jelikož z důvodu zamykání tabulek byla nutná odstávka aplikace, tak představuje nástroj Percona Toolkit ideálnější variantu, jelikož oproti MySQL vypisuje procentuálně postup modifikace tabulky a časový odhad, jak dlouho bude tabulka ještě modifikována. Dalším důvodem výběru nástroje Percona Toolkit byl fakt, že oproti MySQL Percona Toolkit původní tabulku automaticky nemaže, ale umožňuje ji při použití správných parametrů zachovat, což je z důvodu zálohování ideální varianta.

---

[11] Oficiální stránky nástroje Percona Toolkit - https://www.percona.com/software/database-tools/percona-toolkit

5.1.2 Analýza

Jelikož nástroj Percona Toolkit vytváří kopii modifikované tabulky, bylo nejprve nutné provést analýzu, zdali bude místo na disku s primární databází dostatečné. A protože je primární databáze replikována do sekundární databáze, bylo také nutné zajistit, aby byl dostatek volného místa i na disku se sekundární databází.

Při analýze bylo zjištěno, že celková velikost tabulky včetně indexů byla 300 GB. Jak již bylo zmíněno, Percona Toolkit vytváří kopii tabulky, a tudíž bylo nutné zajistit, aby bylo na disku primární i sekundární databáze dalších 300 GB volného místa na kopii tabulky. Na disku primární databáze se nacházelo 200 GB volného místa a na disku sekundární databáze se nacházelo 350 GB místa.

5.1.3 Přidání sloupce

Jelikož na obou discích nebylo alokováno veškeré místo, byla velikost obou disků rozšířena tak, aby zbývalo 400 GB volného místa, což pokrylo velikost kopírované tabulky a zároveň představovalo dostatečnou rezervu pro nově vkládaná data do databáze.

Následně byl spuštěn nástroj Percona Toolkit, který začal modifikovat cílovou tabulku. Během modifikace tabulky byl monitorován výstup nástroje Percona Toolkit, který každou minutu vypisoval odhady, jak dlouho bude zhruba tabulka ještě upravována. Po 19 hodinách a 40 minutách, když nástroj Percona Toolkit odhadoval posledních 2 minuty modifikace, byla odstavena aplikace, která pracuje s databází, ve které se nachází modifikovaná tabulka.

Po dokončení modifikace tabulky prohodil nástroj Percona Toolkit původní tabulku s novou tabulkou a bylo možné znovu zapnout aplikaci.

5.1.4 Závěr úlohy

Přidávání sloupce trvalo 19 hodin a 42 minut. Odstávka aplikace trvala 5 minut.

5.2 *Hromadná úprava hodnot v MySQL tabulce s 500 milióny řádky*

V této řešené úloze bylo nutné na produkčním prostředí upravit záznamy v MySQL tabulce, jenž obsahuje záznamy o výrobě elektřiny v jednotlivých výrobních blocích elektráren. Tato tabulka se skládá z několika sloupců a obsahuje přes 500 milionů záznamů. Sloupce klíčové pro tuto úlohu jsou:

- DateTimePeriodID – Tento sloupec označuje čas, ke kterému daný řádek patří, ve formátu YYYYMMDDHHmm
- AreaCode – Označení oblasti, ve které se výrobní jednotka nachází.
- GenerationUnitCode – Označení výrobní jednotky.
- ActualGenerationOutput – Hodnota definující, kolik elektřiny bylo vyrobeno.
- VersionNumber – Verze dané hodnoty.

První verze aplikace vkládala do této tabulky data, kde výrobní bloky byly označeny tzv. krátkými kódy, kde krátký kód představoval EIC[12] výrobní jednotky. Poté byla nasazena další verze aplikace, která vkládala do tabulky data, kde výrobní bloky byly označeny tzv. dlouhými kódy, které se skládají z EIC elektráren a EIC jejich výrobních bloků, ve formátu KOD_ELEKTRARNY|KOD_BLOKU.

Cílem této úlohy je při odstávce upravit kód výrobního bloku všech řádků, které obsahují dlouhý kód tak, aby řádky obsahovaly pouze krátký kód a zároveň bylo dodrženo originální verzování dle sloupce VersionNumber. Odstávek může být využito více, avšak žádná odstávka nesmí být delší než hodinu.

5.2.1 Analýza

Nejprve bylo nutné zjistit, zdali je bezpečné zaměnit všechny dlouhé kódy za krátké kódy. Při analýze bylo zjištěno, že to bezpečné není, jelikož tabulka obsahuje dva druhy duplicit.

---

[12] Energy Identification Code – 16 místní kód využívaný v energetice. Seznam všech schválených EIC je veřejně dostupný na https://www.entsoe.eu/data/energy-identification-codes-eic/eic-approved-codes/

Prvním druhem duplicity bylo nesprávné verzování ve sloupci VersionNumber u řádků s dlouhým kódem. Ve druhé verzi aplikace se vyskytl bug, kvůli kterému se občas nenalezla předchozí verze dané hodnoty, a tudíž byl založen duplicitní řádek s verzí 1.

Dalším druhem duplicity bylo také porušené verzování, a to v případech, kde byla data vložena starší verzí aplikace. Tato data obsahovala krátký kód ve sloupci GenerationUnit. Následně novější verze aplikace vložila do této tabulky data pro stejný výrobní blok a stejný čas, ale s dlouhým kódem. Jelikož aplikace hledala předchozí verze pouze s dlouhým kódem, nebyla předchozí verze nalezena, a tudíž aplikace v těchto případech započala verzování znovu od čísla 1.

Při analýze byl úkol rozdělen na 2 fáze – zarovnání verzí u duplicitních řádků a nahrazení dlouhých kódů krátkými kódy.

5.2.2 Odstranění duplicitních verzí

Jelikož hledání duplicitních záznamů obnášelo spouštění příkazů s klauzulí Group By, která představovala potenciální hrozbu pro výkonost produkční databáze, byla pro nalezení všech duplicitních řádků vytvořena záloha produkční tabulky. Tato záloha byla vložena na vývojové prostředí, na kterém byly vyhledány všechny duplicity.

5.2.3 Hledání duplicitních verzí

K nalezení duplicit byly vytvořeny 2 tabulky, každá pro jeden typ dané duplicity. Tyto tabulky byly určené k tomu, aby v sobě ukládali reference na duplicitní data.

```
CREATE TABLE IF NOT EXISTS actualgenerationoutputperunit_script1(
    DateTimePeriodID BIGINT(20) NOT NULL,
    AreaCode VARCHAR(30) NOT NULL,
    GenerationUnitCode VARCHAR(40) NOT NULL
    PRIMARY KEY (DateTimePeriodID, AreaCode, GenerationUnitCode)
  );

CREATE TABLE IF NOT EXISTS actualgenerationoutputperunit_script2(
    ID BigInt(20) AUTO_INCREMENT PRIMARY KEY,
    DateTimePeriodID BIGINT(20) NOT NULL,
    AreaCode VARCHAR(30) NOT NULL,
    GenerationUnitLongCode VARCHAR(40) NOT NULL,
    GenerationUnitShortCode VARCHAR(40) NOT NULL
  );
```

SQL příkaz 1 – tabulky ukládající reference duplicitních hodnot

Pro nalezení duplicit prvního typu (duplicitní verze u řádků se stejnou kombinací datumu, oblasti a výrobní jednotky) byl použit následující příkaz, který vložil reference na existující duplicity do tabulky *actualgenerationoutputperunit_script1*.

```
INSERT INTO actualgenerationoutputperunit_script1
SELECT DateTimePeriodID, AreaCode, GenerationUnitCode
   FROM actualgenerationoutputperunit
   GROUP BY DateTimePeriodID, AreaCode, GenerationUnitCode,
VersionNumber
   HAVING COUNT(*) > 1
   ORDER BY DateTimePeriodID,GenerationUnitCode;
```

SQL příkaz 2 – vyhledávání duplicit prvního typu

Duplicity druhého typu byly odhaleny pomocí následujícího příkazu, který vložil reference na existující duplicity do tabulky *actualgenerationoutputperunit_script2*. Tento příkaz nejprve nalezl všechny řádky v tabulce. Mezi těmito řádky nalezl duplicitní verze pro kombinaci datumu, oblasti a výrobní jednotky. Jelikož některé výrobní jednotky měly více duplicitních verzí, tak příkaz vyfiltroval pouze unikátní hodnoty datumu, oblasti a výrobní jednotky.

```sql
INSERT INTO actualgenerationoutputperunit_script2 (DateTimePeriodID, AreaCode,
GenerationUnitLongCode, GenerationUnitShortCode)
SELECT DateTimePeriodID, AreaCode,
 IF (LOCATE('|', maxFilter) > 0, maxFilter, minFilter) AS GenerationUnitLongCode,
 IF (LOCATE('|', maxFilter) = 0, maxFilter, minFilter) AS GenerationUnitShortCode
 FROM (
   SELECT DateTimePeriodID, AreaCode, MAX(GenerationUnitCode) AS maxFilter,
MIN(GenerationUnitCode) AS minFilter
      FROM (
        SELECT DateTimePeriodID, AreaCode, GenerationUnitCode, VersionNumber
        FROM actualgenerationoutputperunit
        ORDER BY GenerationID DESC
      ) AS allValues
    GROUP BY DateTimePeriodID, AreaCode,
        IF (LOCATE('|', GenerationUnitCode) > 0, SUBSTRING(GenerationUnitCode,
LOCATE('|', GenerationUnitCode) + 1), GenerationUnitCode),
        VersionNumber
    HAVING COUNT(*) > 1
    ORDER BY DateTimePeriodID, GenerationUnitCode
   ) AS firstFilter
 GROUP BY DateTimePeriodID, AreaCode,
 IF (LOCATE('|', GenerationUnitCode) > 0, SUBSTRING(GenerationUnitCode, LOCATE('|',
GenerationUnitCode) + 1), GenerationUnitCode)
 HAVING COUNT(*) > 1;
```

SQL příkaz 3 - vyhledávání duplicit druhého typu

5.2.4 Zarovnání duplicitních verzí

Před zarovnáním duplicitních dat byly na produkční prostředí nahrány tabulky *actualgenerationoutputperunit_script1* a *actualgenerationoutputperunit_script2* připravené na vývojovém prostředí.

Vzhledem k tomu, že referenční tabulky obsahovaly dohromady více než 650 000 referencí, bylo zarovnávání duplicitních dat rozděleno do dvou odstávek. Během každé odstávky byl zarovnán jeden typ duplicit za pomocí procedury, která byla pro daný typ duplicity připravena.

Obě procedury fungovaly tak, že si načetly všechny reference z tabulky určené pro daný typ duplicity. Následně procházely jednu referenci po druhé a postupně si za

pomocí reference nalezly všechny relevantní záznamy v tabulce *actualgenerationoutputperunit*. Následně byly všechny nalezené záznamy seřazeny dle primárního klíče, což je číslo vytvořené pomocí vlastnosti Auto increment. Seřazením dle primárního klíče bylo zaručeno, že záznamy byly seřazené v takovém pořadí, v jakém byly vloženy do tabulky. Takto seřazeným záznamům byla postupně nastavována verze. Prvnímu záznamu byla přiřazena verze 1 a u každého dalšího záznamu se verze zvyšovala o 1.

Procedury se od sebe lišily pouze způsobem, jakým vyhledávaly data v tabulce *actualgenerationoutputperunit.* První procedura filtrovala data ve sloupci GenerationUnit pouze za pomocí jednoho kódu, kdežto druhá procedura filtrovala data dle dlouhého i krátkého kódu.

5.2.5 Nahrazení dlouhých kódů krátkými

Jelikož dlouhý kód v sobě obsahuje i krátký kód, nabízel se pro zkrácení dlouhých kódů SQL příkaz níže.

```
UPDATE actualgenerationoutputperunit
    SET GenerationUnitCode = SUBSTRING(GenerationUnitCode, LOCATE('|',
GenerationUnitCode) + 1);
```

SQL příkaz 4 - nahrazení dlouhých kódů krátkými

Tento příkaz však nebylo možné spustit na produkčním prostředí nad takto objemnou tabulkou. Při spuštění tohoto příkazu na vývojovém prostředí byl příkaz proveden za 122 minut. Jelikož byla maximální doba odstávky aplikace 60 minut, bylo zkracování dlouhých kódů rozděleno na 2 fáze. Nejprve byly před odstávkou zkráceny kódy historických hodnot (hodnoty pro rok 2018 a starší) a poté byly zkráceny kódy zbylých hodnot (hodnoty pro rok 2019).

Pro co nejlepší výkonost příkazu bylo nalezeno ID nejstarší hodnoty v roce 2019 a následně byl spuštěn upravený příkaz, který zkracoval dlouhé kódy hodnot pro rok 2018 a starší.

```
UPDATE actualgenerationoutputperunit
    SET GenerationUnitCode = SUBSTRING(GenerationUnitCode, LOCATE('|',
GenerationUnitCode) + 1)
WHERE GenerationID < 405409912;
```

SQL příkaz 5 - nahrazení dlouhých kódů krátkými pro historické hodnoty

Tento příkaz byl na produkčním serveru proveden za 108 minut.

Následně bylo vyčkáno na odstávku aplikace. Během odstávky byl spuštěn poslední příkaz, který nahradil dlouhé kódy krátkými pro hodnoty z roku 2019.

```
UPDATE actualgenerationoutputperunit
    SET GenerationUnitCode = SUBSTRING(GenerationUnitCode, LOCATE('|',
GenerationUnitCode) + 1)
WHERE GenerationID >= 405409912;
```

SQL příkaz 6 - nahrazení dlouhých kódů krátkými pro aktuální hodnoty

Doba provádění tohoto příkazu byla 14 minut.

5.2.6 Závěr úlohy

Úprava dat v tabulce byla rozdělena na několik kroků. Nejprve byly úspěšně zarovnány verze jednotlivých hodnot a poté byly dlouhé kódy zaměněny za krátké.

Celá procedura vyžadovala 3 odstávky aplikace, žádná odstávka nepřesáhla hodinu.

5.3 *Optimalizace MongoDB indexů*

Cílem této úlohy bylo zrychlit načítání dat z kolekcí v produkční MongoDB databázi. Jednalo se o kolekce:

- ActualGenerationOutputPerUnit
- ActualTotalLoad
- GenerationUnitOutage
- ProductionUnitOutage
- ScheduledCommercialExchanges
- TransmissionGridOutage

5.3.1 Analýza

Při analýze byly prozkoumány již existující indexy v daných kolekcích. Každá ze zmíněných kolekcí obsahovala právě jeden index.

V kolekcích ActualGenerationOutputPerUnit, ScheduledCommercialExchanges a ActualTotalLoad byly všechny atributy, které využívá aplikace při hledání dat v databázi, ve vnořeném dokumentu metadata. Indexy všech tří kolekcí byly navrženy stejně. Indexy těchto kolekcí se skládaly z atributů metadata.startDate (datum označující počátek platnosti dat v dokumentu), metadata.endDate (datum označující konec platnosti dat v dokumentu) a metada.Dimensions (pole textů obsahující kódy byznysových dimenzí).

Ve zbývajících 3 kolekcích se nenacházel vnořený dokument metadata. Kolekce GenerationUnitOutage a ProductionUnitOutage měli taktéž stejně navržené indexy. Oba tyto indexy se skládaly z atributů start (datum označující počátek platnosti dat v dokumentu), end (datum označující konec platnosti dat v dokumentu), area (textové označení byznysové dimenze), outageType (textové označení typu dat) a status (textové označení typu dat).

Index poslední kolekce TransmissionGridOutage měl stejné složení jako indexy předchozích dvou kolekcí, pouze měl namísto atributu area atributy inArea a outArea. Stejně jako atribut area u předchozích kolekcí, i tyto dva atributy označovaly byznysové dimenze.

Při analýze nakonec byly prozkoumány dotazy, kterými se aplikace dotazuje na data v MongoDB databázi a bylo zjištěno, že indexy všech kolekcí již obsahují všechny atributy použité při filtrování dat.

5.3.2 Návrh a aplikace nových indexů

Při analýze bylo zjištěno, že ve všech kolekcí je index vytvořen stejným způsobem – data jsou nejprve filtrována pomocí datumů a až poté pomocí dalších textových atributů. Na základě tohoto zjištění se jako řešení nabízela možnost prohodit v indexech textové atributy a datumy.

Na produkční prostředí byly tudíž aplikovány nové indexy níže. Jelikož MongoDB umožnuje použití atributu background při tvorbě nového indexu, bylo možné vytvořit tyto indexy bez odstávky aplikace, jelikož nedošlo k zamčení dat v kolekcích.

```
db.ActualTotalLoad.ensureIndex({"metaData.dimensions": 1, "metaData.startDate"
: 1, "metaData.endDate" : 1}, {"name": "IXActualTotalLoad", "background": true})

db.ActualGenerationOutputPerUnit.ensureIndex({"metaData.dimensions": 1,
"metaData.startDate" : 1, "metaData.endDate" : 1}, {"name":
"IXActualGenerationOutputPerUnit", "background": true})

db.ScheduledCommercialExchanges.ensureIndex({"metaData.dimensions": 1,
"metaData.startDate" : 1, "metaData.endDate" : 1}, {"name":
"IXScheduledCommercialExchanges", "background": true})

db.GenerationUnitOutage.ensureIndex({"area" : 1, "outageType" : 1,"status" : 1,
"start" : 1, "end" : 1}, {"name": "IXGenerationUnitOutage", "background": true})

db.ProductionUnitOutage.ensureIndex({"area" : 1, "outageType" : 1, "status" : 1,
"start" : 1, "end" : 1}, {"name": "IXProductionUnitOutage", "background": true})

db.TransmissionGridOutage.ensureIndex({"inArea" : 1, "outArea" : 1,
"outageType" : 1, "status" : 1, "start" : 1, "end" : 1}, {"name":
"IXTransmissionGridOutage", "background": true})
```

MongoDB příkaz 1 – nově navržené indexy

Po přidání nových indexů byly z kolekcí původní indexy smazány.

```
db.ActualTotalLoad.dropIndex("IX1ActualTotalLoad")

db.ActualGenerationOutputPerUnit.dropIndex("IX1ActualGenerationOutputPerUni
t")

db.GenerationUnitOutage.dropIndex("IX1GenerationUnitOutage")

db.ProductionUnitOutage.dropIndex("IX1ProductionUnitOutage")

db.ScheduledCommercialExchanges.dropIndex("IX1ScheduledCommercialExchan
ges")

db.TransmissionGridOutage.dropIndex("IX1TransmissionGridOutage")
```

MongoDB příkaz 2 – smazání starých indexů

5.3.3 Porovnání výsledků

Pro porovnání výsledků byly připraveny odkazy na GUI aplikace. Stejné stránky na GUI aplikace byly zobrazeny před aplikací nových indexů i po aplikaci nových indexů.

Kolekce	Doba načítání dat před aplikací indexů	Doba načítání dat po aplikaci indexů
ActualGenerationOutputPerUnit	4,3 sec	2,3 sec
ActualTotalLoad	1,2 sec	0.9 sec
GenerationUnitOutage	92,2 sec	7,1 sec
ProductionUnitOutage	92,2 sec	7,1 sec
ScheduledCommercialExchanges	1,4 sec	1,4 sec
TransmissionGridOutage	32,9 sec	4,2 sec

Tabulka 1 – porovnání délky načítání dat před a po změně indexů

5.3.4 **Závěr úlohy**

Prohozením textových atributů a datumů v indexech došlo k výraznému zrychlení načítání dat z 3 MongoDB kolekcí. U dalších 2 kolekcí bylo zrychlení nepatrné a u poslední kolekce bylo změnou indexu dosaženo stejného výsledku jako před aplikací původních indexů. Doba načítání dat ze všech kolekcí se po aplikaci nových indexů pohybuje v řádu jednotek sekund, což je pro použití na produkčním prostředí odpovídající.

5.4 *Využití MongoDB repliky pro aktualizace verze databáze*

Poslední úloha obnášela aktualizaci verze MongoDB databáze a operačního systému serverů, na kterých databázové servery běžely. Databázové servery bylo nutné aktualizovat z verze 2.6 na verzi 3.4 a operační systém bylo nutné aktualizovat z CentOS6 na CentOS7. Celé řešení komplikoval fakt, že není možné povýšit v jednom kroku MongoDB z verze 2.6 na 3.4. Aktualizace databáze je dle oficiální dokumentace možná z verze 2.6 na 3.0, následně z 3.0 na 3.2 a teprve potom z 3.2 na 3.4. Na produkčním prostředí se v MongoDB replica setu nacházely 3 MongoDB servery. První server zastával funkci arbitera. Na tomto serveru již byl nainstalovaný operační systém CentOS7. Další dva servery byly servery datové, z nichž jeden byl server primární a druhý server sekundární. Aktualizaci MongoDB databáze bylo možné provést s několika odstávkami aplikace, která pracuje s tímto MongoDB replica setem, avšak žádná nesměla trvat déle než hodinu.

5.4.1 **Analýza**

Jako vhodné řešení této úlohy bylo navrženo vytvoření dvou nových serverů, na které se nainstaloval CentOS7 a MongoDB servery s verzí 2.6. Tyto nové MongoDB servery byly následně připojeny do MongoDB replica setu a po naplnění daty aktualizovány na verzi 3.4.

Hlavním důvodem výběru tohoto řešení byl fakt, že budou aktualizovány nově přidané servery a ty staré bude stále možné použít v případě jakéhokoliv problému s novými databázovými servery.

Úloha byla během analýzy rozdělena na 2 fáze. První fází bylo přidání sekundárních serverů do replica setu a druhou fází aktualizace těchto nově přidaných serverů. Obě tyto fáze vyžadovaly odstávku aplikace, která s MongoDB replica setem pracovala.

5.4.2 Přidání sekundárních serverů do MongoDB replica setu

Nejprve byly vytvořeny nové servery s operačním systémem CentOS7, na které bylo nainstalováno Mongo s verzí 2.6. Následně bylo vyčkáno na odstávku aplikace. Při odstávce aplikace byly do MongoDB replica setu přidány tyto 2 nové servery, a to spuštěním příkazů níže na primárním MongoDB serveru.

```
rs.add("emfipsmgdb03:27017")
rs.add("emfipsmgdb04:27017")
```

MongoDB příkaz 3 – přidání serverů do replica setu

Následně však bylo nutné manuálně zakázat možnost, aby byl jeden z těchto 2 nových serverů zvolen primárním serverem, jelikož ani jeden z těchto 2 serverů v sobě neobsahoval data. Toho bylo docíleno snížením priority těchto nových serverů v replica setu. Za tímto účelem byla spuštěna následující sekvence příkazů na primárním serveru.

```
cfg = rs.conf()
cfg.members[3].priority = 0
cfg.members[4].priority = 0
rs.reconfig(cfg)
```

MongoDB příkaz 4 – snížení priority v replica setu

Po provedení této sekvence příkazů byla ukončena odstávka aplikace, která trvala 45 minut, a bylo vyčkáno, než se nové sekundární servery naplní daty z primárního serveru.

5.4.3 Naplnění nových serverů daty

Během naplňování nových serverů daty bylo na základě rychlosti naplňování odhadnuto, že nové databázové servery budou obsahovat všechna data během 6-10 hodin.

Po 4 hodinách se však nové databázové servery přestaly naplňovat daty, což bylo způsobeno nedostatečnou velikostní oplogu na primárním MongoDB serveru. Velikost oplogu byla dle výchozích parametrů nastavená na 12 GB. Aplikace pracující s tímto MongoDB replica setem prováděla v databázi stovky změn každou minutu. Současně běžící replikace celé databáze způsobila, že se replikace nových dat hromadila a oplog byl zaplněn během 4 hodin. Jelikož nové databázové servery neměly doreplikovaná všechna data, stal se pro ně oplog neaktuální a z toho důvodu skončila replikace těchto serverů s chybou.

Bylo tedy nutné zvětšit velikost oplogu na všech databázových serverech, kompletně smazat nová data na nových sekundárních serverech a začít replikaci znovu. Za tímto účelem bylo nutné naplánovat další odstávku aplikace.

Během odstávky aplikace byl smazán oplog a byl vytvořen nový oplog s velikostí 40 GB, což mělo dle odhadu pokrýt všechny změny v databázi provedené za posledních 11 hodin. Pro smazání a vytvoření nového oplogu byla spuštěna sekvence příkazů níže.

```
db.oplog.rs.drop()
db.runCommand( { create: "oplog.rs", capped: true, size: (40 * 1024 * 1024 * 1024) } );
```

MongoDB příkaz 5 – zvětšení velikosti oplogu

Tato sekvence příkazů bylo nejprve spuštěna na sekundárním serveru s operačním systémem CentOS6. Následně byl tento sekundární server manuálně zvolen primárním serverem a tato sekvence příkazů byla spuštěna i na ostatních serverech.

Za účelem snížení vytížení primárního serveru byla posléze v MongoDB replica setu povolena řetězová replikace, což znamenalo, že se nové sekundární servery nereplikovaly z primárního serveru, ale ze sekundárního serveru, který byl plně synchronizovaný s primárním serverem.

```
cfg = rs.config()
cfg.settings.chainingAllowed = true
rs.reconfig(cfg)
```

MongoDB příkaz 6 – povolení řetězové replikace

Následně byly smazány všechna data z nově přidaných serverů, do kterých se poté začaly data replikovat znovu. Nakonec byla ukončena odstávka aplikace, která ihned po odstávce začala znovu pracovat s databází. Odstávka aplikace trvala 60 minut.

Celý proces naplňování dat byl i po odstávce dále monitorován. Data na nových sekundárních serverech byla úspěšně synchronizována po 16 hodinách.

5.4.4 **Aktualizace MongoDB serverů na verzi 3.4**

Poté, co byly všechny sekundární servery synchronizované s primárním serverem, byla naplánována finální odstávka aplikace, během které došlo k aktualizaci MongoDB serverů.

Před začátkem aktualizace byl z replica setu odpojen sekundární server s operačním systémem CentOS6 jako záloha pro případ výskytu jakékoliv komplikace při aktualizaci databáze.

```
rs.remove("emfipsmgdb02:27017")
```

MongoDB příkaz 7 – odstranění serveru z MongoDB replica setu

Po odpojení záložního databázového serveru byla novým serverům přidělena možnost stát se primárním serverem nastavením priority na hodnotu 1.

```
cfg = rs.conf();
cfg.members[3].priority = 1;
cfg.members[4].priority = 1;
rs.reconfig(cfg);
```

MongoDB příkaz 8 - zvýšení priority v replica setu

Následně byly v replica serveru všechny databáze postupně aktualizovány dle doporučeného postupu v oficiální MongoDB dokumentaci. Nejprve byl na verzi 3.0

aktualizován arbiter replica setu, poté byly aktualizovány sekundární servery a poté byl aktualizován i primární server. Po této aktualizaci byly aktualizovány všechny databázové servery na verzi 3.2 – opět byl aktualizován nejprve arbiter replica setu, poté sekundární servery a posléze i server primární. Nakonec byly stejným postupem databáze aktualizovány na verzi 3.4.

Po aktualizaci MongoDB serverů byl z replica setu odebrán i druhý server s operačním systémem CentOS6.

5.4.5 Závěr úlohy

Úlohu se podařilo navrženým postupem úspěšně vyřešit. K vyřešení bylo nutné uskutečnit 3 odstávky aplikace. Žádná z těchto odstávek netrvala déle než 60 minut.

6 Optimalizace dotazů a izolačních úrovní

6.1 *Izolační úrovně*

Izolační úroveň definuje, jak je integrita transakcí viditelná pro ostatní uživatele a systémy.

6.2 *Izolační úrovně v MySQL*

MySQL podporuje izolační úrovně Read Committed, Read Uncommitted, Repeatable Read a Serializable.

Izolační úroveň Read Commited plně podporuje vlastnost izolace, která již byla popsána v kapitole 2.1 Relační databáze. V případě, že v databází probíhá transakce, tak tato transakce vidí pouze změny z transakcí, které skončily se stavem Commited. Tato izolační úroveň podporuje tzv. nonrepeatable read, což znamená, že pokud je spuštěn v dané transakci ten samý příkaz dvakrát, databáze může vrátit v každém případě jiná data.

Read Uncommited je izolační úroveň, která rozporuje vlastnosti izolace. V této izolační úrovni totiž běžící transakce vidí změny ostatních probíhajících transakcí, což může způsobit spoustu problémů, pokud tomu není aplikace nad databází uzpůsobená. Čtení dat, které vytvořily neukončené transakce, se nazývá dirty read, neboli špinavé čtení. Tato izolační úroveň bývá využívána nejméně, jelikož z hlediska výkonosti není tato izolační úroveň nijak výrazně rychlejší než ostatní, které navíc poskytují spoustu výhod.

Izolační úroveň Repeatable Read je výchozí izolační úrovní MySQL. Garantuje, že dotaz provedený v dané transakci dvakrát, dostane pokaždé stejný výsledek, jelikož za pomocí zámků je nemožné data měnit. Zámky jsou drženy po celou dobu transakce. V této úrovni se však může vyskytnout tzv. fantom, což je řádek, který je nově vložen do tabulky.

Poslední izolační úrovní je úroveň Serializable. Tato izolační úroveň je považována za nejbezpečnější, jelikož odstraňuje problémy všech ostatních úrovní. Tato úroveň zamyká všechny všechna čtená data, ale i data, která by mohl jiný proces vložit do čteného rozsahu řádků.

6.3 *Izolační úrovně v MongoDB*

MongoDB nepodporuje transakce ani použití izolačních úrovní na úrovni kolekcí, avšak umožňuje použití izolačních úrovní na úrovni jednotlivých dokumentů. MongoDB podporuje izolační úroveň Read Uncommited. V této izolační úrovni vidí dotazy výsledky všech operací, které mění data, ještě před tím, než se tyto změny zapíšou do binárního transakčního logu.

MongoDB operace, která mění více dokumentů (např *updateMany()*) je atomická pro každý dokument, nicméně operace jako celek je neatomická. Dokument, který je upravován, je zamčen a čtecí operace tento dokument nezvládne přečíst, dokud nebudou všechny měněné atributy daného dokumenty změněny.

7 Monitorování databáze a operací

Správné monitorování databáze vyžaduje především dva druhy monitorování – monitorování stavu databáze, pro detekování a upozornění v případě selhání databáze, a monitorování metrik pro sledování trendů databáze, diagnózy atd.

7.1 *Monitorování MySQL databáze*

MySQL nabízí spoustu možností, jak monitorovat MySQL databázi.

Pro základní monitorování MySQL databáze nabízí MySQL příkaz *show processlist*. Tento příkaz je užitečný zejména v případech, kdy při spuštění dotazů odpoví MySQL server chybou „Too many connections“. Příkaz *show processlist* vypíše všechny běžící procesy v MySQL serveru. Pro každý běžící proces je vypsán identifikátor procesu, uživatel, který proces spustil, IP, ze které je proces spuštěn, databáze, ve které je proces prováděn, čas, jak dlouho je již proces prováděn, status, definující, v jakém stavu proces je a samozřejmě také příkaz, který je v rámci daného procesu spuštěn v MySQL databázi. Pokud by bylo nutné ukončit nějaký běžící proces v databázi, je možné použít příkaz *kill*. Příkaz k*ill 27* ukončí v databázi proces, jehož identifikátor v processlistu je 27. Příkaz *kill* bývá často používán pro ukončení tzv. zombie vláken, což jsou vlákna, která již neprovádí žádné příkazy, nicméně v databázi stále existují.

Dalším užitečným příkazem pro základní monitorování MySQL je příkaz *SHOW ENGINE*. Tento příkaz umožnuje zobrazení informací o zvoleném úložném formátu. Příkazem *SHOW ENGINE INNODB STATUS* jsou zobrazeny informace ze standartního monitoringu z úložného formátu InnoDB. Nejdůležitější informací, která je v tomto monitoringu obsažená je informace o všech běžících transakcích v databázi. Další důležitou informací z tohoto monitoringu je informace o posledním detekovaném deadlocku. Deadlock je nežádoucí situace, kdy se blokují 2 nebo více transakcí, jelikož si navzájem zamykají řádky či tabulky potřebné pro jejich dokončení.

Užitečným nástrojem, který MySQL nabízí je tzv. slow query log. Jedná se o soubor, ve kterém jsou zaznamenány všechny provedené příkazy, jejichž délka provedení byla delší, než počet nakonfigurovaných sekund v parametru *long_query_time*.

Každý příkaz zaznamenaný ve slow query logu obsahuje informace o tom, jak dlouho byl prováděn, jak dlouho mu trvalo zamknout řádky, které afektoval, kolik řádků afektoval a kolik řádků prozkoumal. Slow query log se dá použít pro analýzu a optimalizaci dlouho běžících příkazů.

7.2 *Monitorování MongoDB databáze*

MongoDB poskytuje řadu nástrojů či příkazů pro sběr dat o stavu MongoDB serveru.

Nejpoužívanějším nástrojem je nástroj mongostat. Mongostat poskytuje rychlý přehled o stavu MongoDB serveru. Tento nástroj zobrazuje počty operací databáze podle typu (insert, update, delete, atd.).

Dalším často používaným nástrojem je nástroj mongotop, který monitoruje aktuální trendy v čtení a zápisech na MongoDB serveru pro jednotlivé kolekce.

Mezi nejpoužívanější monitorovací příkazy patří *db.serverStatus*, *db.dbStats*, *db.collection.stats, db.currentOp, rs.Status* a *rs.printReplicationInfo*.

Příkaz *db.serverStatus* vrací obecný přehled o stavu databáze, využití disku a paměti.

Další příkaz *db.dbStats* lze použít k zobrazení informací o tom, kolik zabírá databáze celkově místa na disku a kolik zabírají místa na disku data, indexy či jednotlivé kolekce.

K monitorování statistik jednotlivých kolekcí lze použít příkaz *db.collection.stats* kde collection představuje název dané kolekce. Tyto statistiky jsou stejné jako u příkazu *db.dbStats*, ale jsou spjaté přímo jen s danou kolekcí.

Pro zobrazení všech aktuálně běžících procesů v MongoDB serveru je používán příkaz *db.currentOp*. Pro každý proces je v tomto přehledu vypsáno několik informací. Mezi nejdůležitější informace patří příkaz, který daný proces provádí, index, který daný příkaz používá, doba trvání daného procesu či zámky, na které daný proces čeká.

Pro monitorování replica setu je používán příkaz *rs.Status*, který vrací přehled o daném replica setu. Tento přehled zobrazuje informace o všech serverech v replica setu a jejich stavu. U každého serveru je v tomto přehledu napsané, zdali je daný server v daném replica setu primární, sekundární či arbiter. Tento příkaz byl

používán v úloze 5.4 pro kontrolu stavu replica setu při manuálním zvolení nového primárního serveru.

Poslední z těchto příkazů *rs.printReplicationInfo* zobrazuje výpis informací o oplogu daného serveru. V tomto výpisu se nachází informace o nastavené velikosti oplogu, o časech, ke kterým patří první a poslední příkaz v oplogu a také kolik hodin příkazů tento oplog pokrývá. Tento příkaz byl také používán v úloze 5.4.

7.3 *Obecné monitorovací systémy*

Mezi nejpoužívanější veřejně dostupný monitorovací software patří Nagios[13] a Zabbix[14]. Tyto obecné monitorovací systémy jsou hojně využívány také pro monitorování stavu a metrik databáze. Běžnou praxí je kombinace monitoringu stavu DB systému (běžící procesy) a vybraných provozních metrik (počty operací, zombie procesy, dlouho trvající operace atp.).

Nagios periodicky kontroluje, jestli nadefinované procesy běží, provádí nadefinované kontroly a porovnává výsledky kontrol s předpokládanými výsledky. Pokud se předpokládané výsledky kontrol liší se skutečnými výsledky, systém odešle notifikace nadefinovaným komunikačním kanálem.

Zabbix využívá flexibilní způsob notifikačního mechanismu, který umožňuje uživatelům konfigurovat odesílané notifikace pro jakoukoliv událost. Díky tomu mohou uživatelé rychle reagovat na případné problémy s monitorovanými servery. Oproti Nagiosu neukládá Zabbix všechny své konfigurace v konfiguračních souborech, nýbrž v databázi. Zabbix také podporuje větší množství typů dat a díky tomu má širší možnosti monitorování dat.

---

[13] Oficiální stránky Nagios - https://www.nagios.org/
[14] Oficiální stránky Zabbix - http://www.zabbix.com/

8 Zhodnocení výsledků

Hlavním cílem této práce bylo porovnat a zhodnotit výhody a nevýhody databází MySQL a MongoDB z hlediska práce s velkými objemy dat.

V práci byly nejprve představeny principy relačních a NoSQL databází. Speciálně byly představeny databáze MySQL a MongoDB. Pro tyto dvě databáze byly posléze popsány možnosti škálování, replikace, zálohování a navrhování dat a indexů. Popis navrhování indexů byl také doplněn jednoduchou ukázkou použití. Dále byly popsány praktické úlohy zabývající se manipulací s daty a strukturou databáze v živém provozu. Nakonec byly popsány optimalizační úrovně a způsoby monitorování obou databází.

Z práce vyplývá, že obě databáze MySQL a MongoDB slouží k určitému specifickému významu. Pokud není pevně daná struktura dat, které má databáze uchovávat, je vhodné použít databázi MongoDB, která neomezuje záznamy v kolekci pevným schématem. V případě, že je aplikace pracující s databází citlivá na transakční zpracování dotazů, je databáze MySQL jednoznačně lepší volbou, jelikož databáze MongoDB transakce nepodporuje.

9 Závěr

Práce zhodnocuje a porovnává výhody a nevýhody databází MySQL a MongoDB z hlediska práce s velkými objemy dat.

Výsledky této práce mohou být přínosné osobám zabývající se problematikou softwarové architektury, a to při volbě vhodné databázové technologie pro konkrétní aplikaci. Tato práce je také velmi přínosná pro autora práce, jelikož mu pomohla ponořit se hlouběji do problematiky rozdílů relačních a NoSQL databázových systémů.

Mezi náměty pro rozšíření práce se nabízí přidání dalších typů relačních a NoSQL databází a jejich porovnání s databázemi MySQL a MongoDB.

10 Seznam zdrojů

[1] MROZEK, Jakub. NoSQL Databáze [Bakalářská práce], Vysoká škola ekonomická v Praze 2013 [cit. 15.01.2018]

[2] PANYKO, Tomáš. NoSQL Databáze [Bakalářská práce] Jihočeská univerzita v Českých Budějovicích 2013. [cit. 15.01.2018]

[3] FIRLA, Dominik. Srovnání řešení správy dat v MySQL a MongoDB při použití Doctrine 2 ve frameworku Symfony 2 [Bakalářská práce], Vysoká škola ekonomická v Praze, 2016. [cit. 15.01.2018]

[4] Db engines. Ranking. [online] [cit. 25. 01. 2018] Dostupný z https://db-engines.com/en/ranking

[5] SILBERSCHATZ, Abraham, Henry F KORTH a S SUDARSHAN. Database system concepts. 6th ed. Boston: McGraw-Hill, c2011. ISBN 978-0-07-352332-3

[6] DYER, Russell J. MySQL in a nutshell. 2nd ed. Sebastopol, CA: O'Reilly, c2008. ISBN 978-0-596-51433-4.

[7] MySQL 5.7 Reference Manual [Online] [cit. 24. 01. 2018] Dostupné z: https://dev.mysql.com/doc/refman/5.7/en/

[8] Column-Oriented Database Systems [Prezentace] [Online] [cit. 25. 01. 2018] Dostupné z: http://www.cs.yale.edu/homes/dna/talks/Column_Store_Tutorial_VLDB09.pdf

[9] ROBINSON, Ian a kol. Graph Databases. O'Reilly, 2013. ISBN 978-1449356262.

[10] STRAUCH, Christoff. NoSQL Databases.[Online] [cit. 26. 01. 2018] Dostupné z: http://www.christof-strauch.de/nosqldbs.pdf

[11] ABITEBOUL, Serge. Querying Semi-Structured Data. [online] [cit: 27. 01. 2018] Dostupný z http://www.dtic.mil/dtic/tr/fulltext/u2/a428473.pdf

[12] BANKER, Kyle. MongoDB in action. Shelter Island, NY: Manning, c2012. ISBN 1935182870.

[13] CHODOROW, Kristina. MongoDB: The Definitive Guide. Second edition. Sebastopol: O'Reilly, 2013. ISBN 978-1-449-34468-9.

[14] SCHWARTZ, Baron, ZAITSEY, Peter, TKACHENKO Vadim. High Performance MySQL: Optimization, Backups, and Replication. O'Reilly, 2012. ISBN 9781449314286.

[15] The MongoDB 3.6 manual [Online] [cit 25. 11. 2018] Dostupné z https://docs.mongodb.com/v3.6/

[16] SQL Index – Clustered Index and Non-Clustered Index [Online] [cit 14. 03. 2019] Dostupné z http://jobinjohn.blogspot.com/2011/04/sql-index-clustered-index-and-non.html

11 Přílohy

1) Zadání bakalářské práce

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Jaroslav Schnaubert**
Studium:	I1500427
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Práce s velkými objemy dat v relačních a NoSQL databázích**
Název bakalářské práce AJ:	Work with Large Quantities of Data in Relational and NoSQL databases

Cíl, metody, literatura, předpoklady:

Výhody relačních a nerelačních databází a jejich použití pro různé typy úloh v praxi. Optimalizace práce s velkým objemem dat, omezení a úskalí jednotlivých technologií a často řešené problémy v reálném provozu. 1. Úvod 2. Porovnání SQL a NoSQL a jejich typické případy použití 3. Replikace, zálohování a škálovatelnost 4. Jak navrhovat strukturu databází 5. Manipulace se strukturou a daty v živém provozu 6. Optimalizace dotazů a izolačních úrovní 7. Monitorování databáze a operací 8. Zhodnocení výsledků 9. Závěr

Luke Welling, Laura Thomson - MySQL Membrey Peter - MongoDB Basics

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Barbora Tesařová, Ph.D.

Datum zadání závěrečné práce: 14.1.2015