České vysoké učení technické v Praze

Fakulta dopravní

Ústav aplikované informatiky v dopravě

Návrh distribuovaného prostředí pro práci s Big Daty

DIPLOMOVÁ PRÁCE

Vypracoval: **Bc. Alena Chernetskaya**

Vedoucí práce: **Ing. Jan Krčál, Ph.D., Ing. Jana Kaliková, Ph.D.**

Rok: **2021**

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
děkan
Konviktská 20, 110 00 Praha 1

K614.. **Ústav aplikované informatiky v dopravě**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Bc. Alena Chernetskaya

Kód studijního programu a studijní obor studenta:

N 3710 – IS – Inteligentní dopravní systémy

Název tématu (česky): **Návrh distribuovaného prostředí pro práci s Big Daty**

Název tématu (anglicky): Design of a Distributed Environment for Working with Big Data

Zásady pro vypracování

Při zpracování diplomové práce se řiďte následujícími pokyny:

- Úvod do problematiky NoSQL
- Analýza současných NoSQL DB podle typu
- Analýza a následný výběr konkrétního DB nástroje s ohledem na požadavky k použití
- Realizace konkrétní DB za použití vybrané DB nástroje
- Zhodnocení vytvořené DB
- Implementace konkrétního DB

Rozsah grafických prací:	stanoví vedoucí diplomové práce
Rozsah průvodní zprávy:	minimálně 55 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)
Seznam odborné literatury:	Irena Holubová, Jiří Kosek, Karel Minařík, David Novák: Big Data a NoSQL databáze. Praha: Grada, 2015. David Loshin: Big data analytics. Amsterdam: Morgan Kaufmann/Elsevier, 2013

Vedoucí diplomové práce: **Ing. Jana Kaliková, Ph.D.**

Ing. Jan Krčál, Ph.D.

Datum zadání diplomové práce: **30. června 2020**

(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)

Datum odevzdání diplomové práce: **17. května 2021**

a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia

b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia

L.S.

doc. Ing. Vít Fábera, Ph.D.
vedoucí
Ústavu aplikované informatiky v dopravě

doc. Ing. Pavel Hrubeš, Ph.D.
děkan fakulty

Potvrzuji převzetí zadání diplomové práce.

Bc. Alena Chernetskaya
jméno a podpis studenta

V Praze dne..30. června 2020

Čestné prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracovala samostatně a použila jsem k tomu pouze zdroje uvedené na konci práce, a to v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Nemám závažný důvod proti užívání tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském a o právech souvisejících s právem autorským.

V Praze dne: ……………………

…………………………

Bc. Alena Chernetskaya

Poděkování

Ráda bych poděkovala svým vedoucím diplomové práce panu Ing. Janu Krčálovi, Ph.D. a paní Ing. Janě Kalikové, Ph.D. za odborné vedení, cenné rady, trpělivost a vstřícnost při konzultacích, a také za věcné připomínky a podnětné návrhy k práci. Dále bych ráda poděkovala svým rodičům a svému příteli za podporu při studiu.

……………………………………

Bc. Alena Chernetskaya

Název práce: Návrh distribuovaného prostředí pro práci s Big Data

Autor: Bc. Alena Chernetskaya

Obor: Inteligentní dopravní systémy

Druh práce: Diplomová práce

Vedoucí práce: Ing. Jan Krčál, Ph.D., Ing. Jana Kaliková, Ph.D.

Ústav aplikované informatiky v dopravě, Fakulta dopravní, České vysoké učení technické v Praze

Konzultant: —

Abstrakt: Tato diplomová práce se zabývá studováním problematiky NoSQL (not only SQL) databáze, teoretickým popisem jejich funkcionality a jejich porovnáním s relačními databázemi. Následně práce pojednává o požadavcích na budoucí distribuované prostředí z hlediska existujících dat.

Dalším z cílů bylo seznámit se s různými typy databází NoSQL a učinit mezi nimi výběr pro následné nasazení konkrétní databáze. Hlavním výstupem praktické části práce je realizace databáze dopravních dat ve vybraném prostředí a následné její otestování.

Klíčová slova: SQL, NoSQL, prostředí, databáze, model, data, tabulka, dotaz, Cassandra

Title: Design of a Distributed Environment for Working with Big Data

Author: Bc. Alena Chernetskaya

Abstract: This diploma thesis is analysis of NoSQL (not only SQL) database issues, a theoretical description of their functionality and their comparison with relational databases. Subsequently, the work discusses the requirements for the future distributed environment in terms of existing data.

Another goal was to get acquainted with different types of NoSQL databases and make a choice between them for the subsequent deployment of a particular database.

The main output of the practical part of the work is the implementation of a database of traffic data in a selected environment and its subsequent testing.

Key words: SQL, NoSQL, environment, database, model, data, table, query, Cassandra

Obsah

Obsah 8
Seznam zkratek 10
Úvod 12
1 Charakteristika SQL databáze 14
1.1 Funkcionalita SQL 15
1.2 Výhody jazyka SQL 15
1.2.1 Standardy SQL 15
1.2.2 Základem je relační model 16
1.2.3 Struktura je podobná přirozenému jazyku 16
1.2.4 Architektura „klient/server“ 16
1.2.5 Podpora „open source“ 16
2 Funkcionalita NoSQL databáze 17
2.1 Jak funguje NoSQL databáze 17
2.2 Hlavní kritéria NoSQL databáze 19
2.3 Výhody NoSQL databáze 19
2.4 Nevýhody NoSQL databáze 20
2.5 Výsledné porovnání SQL vs. NoSQL 21
3 Typy NoSQL databází 24
3.1 Databáze klíč-hodnota 24
3.2 Dokumentové databáze 25
3.3 Sloupcové databáze 27
3.4 Grafové databáze 28
3.5 Specifikace dat pro následný výběr databázového nástroje 29
3.6 Výsledný výběr databázového nástroje pro vytvoření distribuovaného prostředí 29
3.6.1 Dostupnost dat 30
3.6.2 Škálovatelnost 31
3.6.3 Dotazovací jazyk 31
3.6.4 Výsledný výběr 31
4 Praktická část 33
4.1 Věta CAP 33
4.1.1 Replikační faktor 33
4.1.2 Úroveň konzistence 34
4.2 Apache Cassandra: první kroky 35

4.2.1 Instalace a napojení uzlů v prostředí Apache Cassandra 35
4.2.2 Proces vytvoření jednoduché tabulky v prostředí Cassandra 37
4.2.3 Základy tvoření datového modelu a principy modelování 38
4.3 Realizace databáze dopravních dat 39
4.3.1 Základní nastavení v prostředí Cassandra pro DB dopravních dat 40
4.3.2 Tvorba konceptuálního modelu 40
4.3.3 Tvorba logického modelu 43
4.3.4 Tvorba fyzického modelu 45
4.4 Provedení stress testů 55
4.4.1 Write test 56
4.4.2 Read test 59
Závěr 62
Bibliografie 63
Seznam obrázků 64
Seznam tabulek 66
Přílohy 67
Skript 1 67
Skript 2 69
Skript 3 72

Seznam zkratek

ACID	Atomicity, Consistency, Isolation, Durability	Atomicita, Konzistence, Izolovanost, Trvalost
ANSI	American National Standards Institute	Americký národní standardizační institut
API	Application Programming Interface	Rozhraní pro programování aplikací
ASCII	American Standard Code for Information Interchange	Americký standardní kód pro výměnu informací
CAP	Consistency, Availability, Partition tolerance	Konzistence, dostupnost, odolnost k přerušení
CQL	Cassandra Query Language	Dotazovací jazyk Cassandra
DB	Database	Databáze
DBMS	Database Management System	Systém pro správu databází
ERD	Entity-relationship model	Entitně vztahový model
FIPS	Federal Information Processing Standards	Federální standard pro zpracování informací
GPS	Global Positioning System	Globální polohový systém
IoT	Internet of Things	Internet věcí
ISO	International Organization for Standardization	Mezinárodní organizace pro normalizaci
JSON	JavaScript Object Notation	JavaScriptový objektový zápis
NoSQL	Not only SQL	Nejen SQL

OLAP	Online Analytical Processing	Online analytické zpracování
OLTP	Online Transaction Processing	Online zpracování transakcí
RAM	Random Access Memory	Operační paměť
RDBMS	Relational Database Management System	Systém pro správu relačních databází
SDC	System Development Corporation	—
SSD	Solid-State Drive	Polovodičový disk
SQL	Structured Query Language	Strukturovaný dotazovací jazyk

Úvod

Samotný termín databáze se objevil na počátku šedesátých let a byl představen na sympoziích organizovaných SDC (System Development Corporation) v letech 1964 a 1965, ačkoli byl zpočátku chápán v poměrně úzkém smyslu, v kontextu systémové umělé inteligence. Termín se rozšířil v moderním smyslu až v 70. letech. [1]

Databázový systém je počítačový systém pro ukládání záznamů, tj. počítačový systém, jehož hlavním účelem je ukládat informace tím, že poskytuje uživatelům prostředky k jejich načítání a úpravám. Informace mohou zahrnovat cokoli, co si zaslouží pozornost jednotlivého uživatele nebo organizace používající systém, jinými slovy vše, co je nezbytné pro aktuální práci daného uživatele nebo podniku. Samotnou databázi pak lze chápat jako úložiště nebo kontejner pro sadu datových souborů zadaných do počítače.

Uživatelé tohoto systému pak dostávají příležitost přidávat nové prázdné soubory do databáze a odstraňovat existující soubory z databáze; vkládat nová data do existujících souborů a také mazat a měnit data ve stávajících souborech.

Existuje mnoho typů databází, které se liší podle různých kritérií; je určeno více než 50 typů databází (např. podle datového modelu, obsahu, typu trvalé pamětí, stupně distribuce apod.)

Databáze lze také rozdělit na relační, tj. databáze, které používají tabulkové schéma řádků a sloupců a nerelační, tj. databáze, které používají model úložiště, který je optimalizován pro konkrétní požadavky typu uložených dat. V takovém případě data mohou být například uložená jako jednoduché páry klíč-hodnota, jako dokumenty JSON, jako graf skládající se z okrajů a vrcholů, anebo jako určitý počet sloupců. Všechna tato úložiště dat jsou pak společná, protože nepoužívají relační model.

Pro práci s relačními databázemi se používá jazyk SQL, který má v sobě soubor operátorů, instrukcí a počítaných funkcí. Název „NoSQL“ se ale stal obecným pojmem pro různé databáze a úložiště a neodkazuje na žádnou konkrétní technologii nebo produkt.

NoSQL databáze jsou optimalizovány pro aplikace, které musí rychle zpracovávat velké objemy dat s různými strukturami a s nízkou latencí. Nerelační úložiště jsou tedy přímo zaměřena na velká data. Myšlenka databází tohoto typu však vznikla mnohem dříve než pojem „velká data“, již v 80. letech minulého století, v dobách prvních počítačů a byla používána pro hierarchické adresářové služby. Moderní chápání NoSQL DBMS se objevilo počátkem roku 2000 jako součást vytváření paralelně distribuovaných systémů pro vysoce škálovatelné internetové aplikace, jako jsou online vyhledávače. Obecně platí, že termín NoSQL znamená „nejen SQL“ (Not only SQL), což charakterizuje odstup od tradičního přístupu k návrhu databáze. [2]

Zpočátku to byl název databáze s otevřeným zdrojovým kódem vytvořené Carlem Strozzi, která ukládala všechna data jako soubory ASCII a místo dotazů na přístup k datům SQL používala skripty Shell. Na počátku 21. století Google vybudoval svůj vyhledávač a aplikace (GMail, Google Maps, Google Earth), které řešily problémy se škálovatelností a paralelním zpracováním velkého množství dat. Takto byly vytvořeny distribuovaný soubor a koordinační systémy, stejně jako úložiště rodiny sloupců, založené na výpočetním modelu MapReduce. Poté, co Google zveřejnil popis těchto technologií, staly se velmi oblíbenými u vývojářů s otevřeným zdrojovým kódem. Ve výsledku byl vytvořen „Apache Hadoop“, což je sbírka softwarových nástrojů s otevřeným zdrojovým kódem, která usnadňuje používání sítě mnoha počítačů k řešení problémů zahrnujících obrovské množství dat a výpočtů. Následně se k této technologií NoSQL pro správu velkých dat připojilo mnoho společností: Facebook, Netflix, eBay, Yahoo! a další IT společnosti se svými otevřenými řešeními. [3]

Tato práce je rozdělena do čtyř kapitol, v nichž je popsána problematika SQL a NoSQL databází, jejich koncepty, hlavní výhody a nevýhody a také jejich výsledné porovnání. Následně je popsána analýza a následný výběr vhodného nástroje pro vytvoření databáze s ohledem na požadavky a výsledná implementace databáze.

První kapitola je věnována charakteristice relačních (SQL) databází, jejich funkcionalitě a také vyjmenování jednotlivých výhod uložišť tohoto typu.

Druhá kapitola je věnována problematice NoSQL databází, principu jejich fungování, jejich hlavním kritériím a vyjmenování jednotlivých silných a slabých stran. Jsou následně vyjmenovány hlavní rozdíly relačních (SQL) a nerelačních (NoSQL) databází.

Kapitola tři popisuje existující typy NoSQL databází a jejich rozdíly a také analýzu a výsledný výběr databázového nástroje pro vytvoření distribuovaného prostředí.

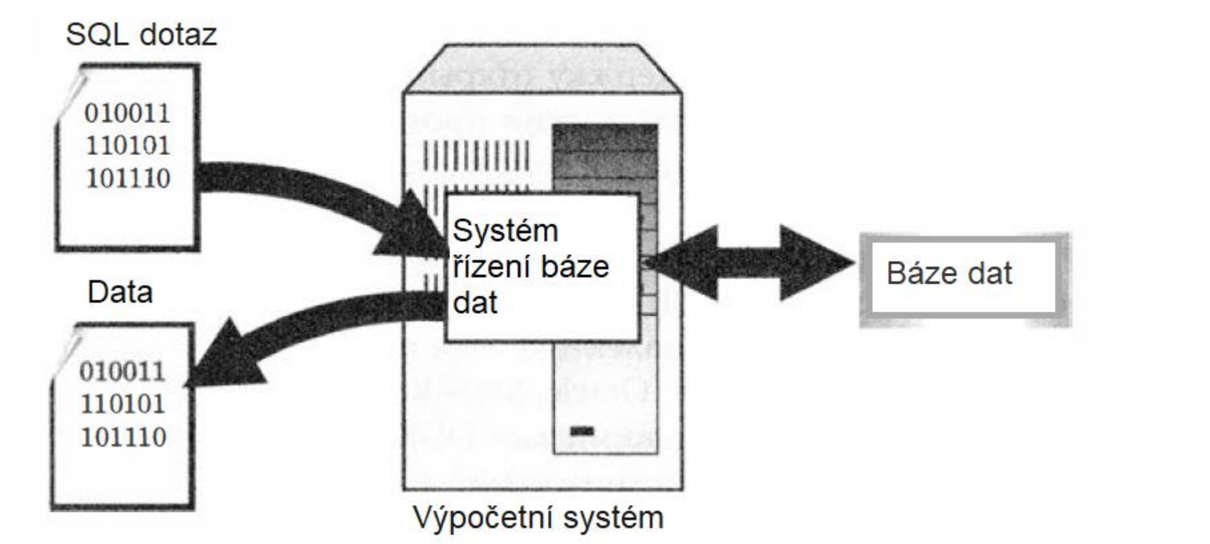
Kapitola čtyři se zaměřuje na realizaci konkrétní databáze za použití vybraného databázového nástroje. Je především popsán proces samotné instalace vybraného nástroje a poté prakticky vytvořeno prostředí pro práci s daty z Fakulty dopravní.

1 Charakteristika SQL databáze

Jako první krok je potřeba podrobně popsat metodu SQL a funkcionalitu databází tohoto typu, aby následně bylo možné posoudit jejich rozdíly s databázemi typu NoSQL, a to jak ve funkcionalitě, tak v příčinách fungování.

Oficiální mezinárodní standard SQL byl několikrát přijat a rozšířen. SQL je jádrem databází společností Microsoft, Oracle a IBM, tří největších softwarových společností na světě. Kromě toho je SQL jádrem open source RDBMS (Relational Database Management System), jako jsou MySQL a Postgres, které podporují šíření Linuxu a open source. Zpočátku byl SQL skromným výzkumným projektem IBM, ale postupem času se stal široce uznávanou důležitou výpočetní technologií. [4]

Na obr. 1 je ukázáno schéma fungování SQL databáze. Podle tohoto schématu má počítačový systém databázi, která ukládá důležité informace. Pokud výpočetní systém patří do oblasti podnikání, může databáze obsahovat například informace o hmotných aktivech, vyrobených produktech, tržbách a platech. Databáze v osobním počítači může obsahovat informace o telefonních číslech a adresách nebo informace získané z většího výpočetního systému. Počítačový program, který spravuje databázi, se nazývá systém pro správu databází neboli DBMS.



Obrázek 1: Přístup k databázi pomocí SQL, zdroj: [4]

Aby bylo možné získat informace z databáze, je potřeba poslat dotaz do DBMS pomocí SQL. Systém DBMS požadavek zpracuje, vyhledá požadovaná data a vrátí je. Proces dotazování na data z databáze a získání výsledku se nazývá dotaz do databáze.

1.1 Funkcionalita SQL

Mezi hlavními funkcemi jazyka SQL z hlediska práce s daty především patří:

• *Definice dat.* SQL umožňuje uživateli definovat strukturu a organizaci uložených dat a vztah mezi položkami uložených dat.

• *Vzorkování dat.* SQL umožňuje uživateli nebo aplikaci načítat a používat data, která obsahuje, z databáze.

• *Zpracování dat.* SQL umožňuje uživateli nebo aplikaci upravit databázi, tj. přidávat do ní nová data a také odstraňovat nebo aktualizovat data, která se již v ní nachází.

• *Řízení přístupu.* Pomocí SQL lze omezit schopnost uživatele vybírat, přidávat a upravovat data a chránit je před neoprávněným přístupem.

• *Sdílení dat.* SQL se používá ke koordinaci sdílení dat souběžnými uživateli, takže změny provedené stejným uživatelem nevedou k neúmyslnému zničení změn provedených přibližně ve stejnou dobu jiným uživatelem.

• *Integrita dat.* SQL pomáhá zajistit integritu databáze ochranou před zničením v důsledku nekonzistentních změn nebo selhání systému.

SQL je tedy dostatečně silný jazyk pro správu a interakci s DBMS a je standardním jazykem pro práci s relačními databázemi. [4]

1.2 Výhody jazyka SQL

Hlavní vlastnosti jazyka SQL je spolehlivost a neměnnost dat a také nízké riziko ztráty informací. Když jsou data aktualizována, je zaručena jejich integrita, protože taková data budou nahrazena vždy ve stejné tabulce. Relační databáze jsou ideální pro práci se strukturovanými daty, která nepodléhají častým změnám.

Hlavními výhodami jazyka SQL jsou jeho standardy, základ jazyka ve formě relačního modelu a jeho struktura, připomínající přirozený jazyk a také architektura jazyku typu „klient/server“ a podpora „open source“. [4]

1.2.1 Standardy SQL

Oficiální jazykový standard SQL byl publikován Americkým národním normalizačním institutem (ANSI) a Mezinárodní normalizační organizací (ISO) v roce 1998, poté byl rozšířen v roce 1989 a poté – v letech 1992, 1999, 2003 a 2006. Kromě toho je SQL americkým federálním standardem pro zpracování informací (FIPS), a proto je dodržování předpisů jedním ze základních požadavků obsažených ve velkých vládních kontraktech pro počítačový průmysl. [4]

1.2.2 Základem je relační model

SQL je jazykem pro relační databáze, a proto se stal populárním ve chvíli, kdy se rozšířil model prezentace relačních dat. Tabulková struktura relační databáze s řádky a sloupci je pro uživatele intuitivní, takže jazyk SQL je jednoduchý a snadno se učí.

Relační model má pevný teoretický základ, který sloužil jako základ pro vývoj a implementaci relačních databází. V návaznosti na popularitu relačního modelu se SQL ve skutečnosti stal jediným jazykem pro relační databáze.

1.2.3 Struktura je podobná přirozenému jazyku

Příkazy SQL jsou podobné běžným větám v angličtině, takže je snadné se je naučit a porozumět jim. To je často způsobeno skutečností, že příkaz SQL popisuje data, která mají být získána, a nikoli způsob, jak je vyhledat. Tabulky a sloupce v relační databázi mohou mít dlouhé popisné názvy.

Výsledkem je, že většina příkazů SQL znamená přesně to, co odpovídá jejich jménům, takže je lze číst jako jednoduché přirozené věty.

1.2.4 Architektura „klient/server“

SQL je prostředek pro implementaci aplikací, které používají distribuovanou architekturu klient / server. V této roli funguje SQL jako spojení mezi klientským systémem, který interaguje s uživatelem, a serverovým systémem, který spravuje databázi, a umožňuje každému soustředit se na provádění svých funkcí. Kromě toho SQL umožňuje osobním počítačům fungovat jako klienti ve vztahu k síťovým serverům nebo větším databázím; to umožňuje přístup k podnikovým datům z aplikací spuštěných na osobních počítačích.

1.2.5 Podpora „open source“

Jedním z posledních důležitých vývojových trendů v počítačovém průmyslu bylo použití přístupu „open source“ k vytváření komplexních softwarových systémů.

S tímto přístupem je zdrojový kód otevřený a je k dispozici zdarma, takže k němu může přispět mnoho programátorů: přidávat do něj nové funkce, opravovat chyby, zlepšovat funkčnost atd. Taková komunita programátorů, kteří mohou pracovat v různých organizacích po celém světě, se s určitou koordinací stává silným motorem nových technologií. Software s otevřeným zdrojovým kódem je obvykle k dispozici za nízkou cenu (nebo dokonce zdarma), což jen zvyšuje jeho přitažlivost.

2 Funkcionalita NoSQL databáze

Databáze NoSQL byly speciálně vytvořeny pro určité datové modely a na rozdíl od SQL mají flexibilní schémata, která umožňují vyvíjet moderní aplikace. Databáze NoSQL jsou rozšířeny díky snadnosti vývoje, funkčnosti a výkonu v libovolném měřítku.

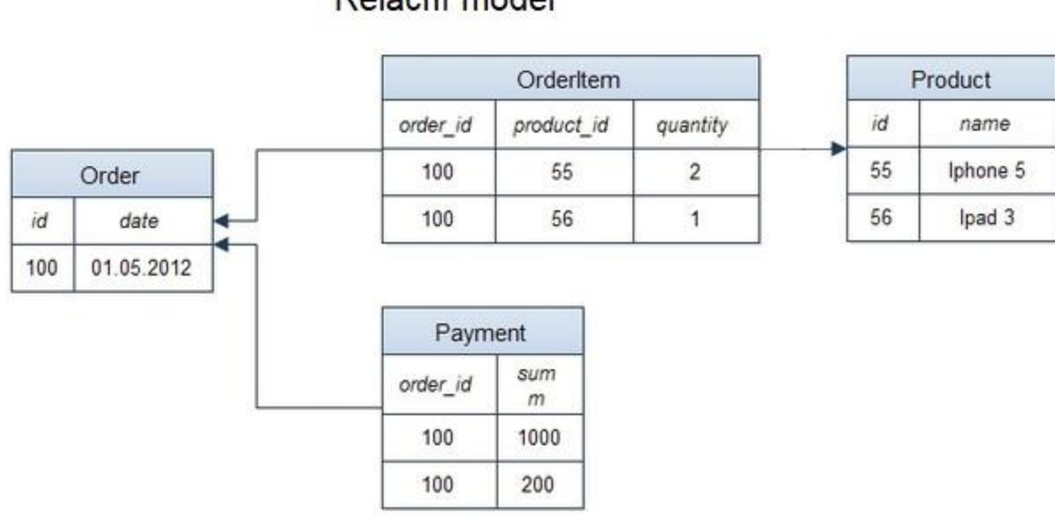
Tento typ databáze používá různé datové modely pro přístup a správu dat a je optimalizovaný pro aplikace, které pracují s velkým množstvím dat, vyžadují nízkou latenci a flexibilní datové modely. To vše je dosaženo zmírněním přísných požadavků na konzistenci dat charakteristických pro jiné typy databází. [4]

Databáze NoSQL se hodí hlavně pro moderní aplikace, ve kterých jsou vyžadovány flexibilní škálovatelné databáze s vysokým výkonem a širokou funkčností, a které mohou poskytnout maximální použitelnost. Jsou to především mobilní, herní a internetové aplikace.

2.1 Jak funguje NoSQL databáze

Pro znázornění konkrétních rozdílů mezi SQL DB a NoSQL DB za příklad byla vzata jednoduchá databáze knih.

V relační databázi (SQL) je položka „OBJEDNÁVKA“ často rozdělena do několika částí a uložena do samostatných tabulek, jejichž vztahy jsou určeny omezeními primárního a cizího klíče. (obr.2)



Obrázek 2: Příklad relačního modelu dat, zdroj: [5]

V daném příkladu jsou čtyři tabulky:

1) Tabulka „OBJEDNÁVKA“ se sloupci „ID“ (primární klíč) a „DATUM“
2) Tabulka „VÝROBEK“ se sloupci „ID“ (primární klíč) a „NÁZEV“
3) Tabulka „POLOŽKA“ se sloupci „ID OBJEDNÁVKY“, „ID VÝROBKU“ (oba jsou cizí klíče) a „MNOŽSTVÍ,
4) Tabulka „PLATBA“ se sloupci „ID OBJEDNÁVKY“ (cizí klíč) a „SOUČET“.

Relační model je navržen tak, aby zajistil integritu referenčních dat mezi tabulkami v databázi. Data jsou normalizována, aby se snížila redundance, a obecně optimalizovaná pro ukládání.

V databázi NoSQL lze položky „OBJEDNÁVKA“ a „PLATBA“ uložit jako dokumenty JSON. (obr.3) Pro každou objednávku nebo platbu budou ostatní hodnoty uloženy jako atributy v jednom dokumentu. V tomto modelu jsou data optimalizována pro intuitivní design a horizontální škálovatelnost.

Nerelační model

```
// Order document
{
     100,
     1000,
     01.05.2012,
     {
          {
          55,
          "Iphone5",
          2
          },
          {
          56,
          "Ipad3",
          1
          },
     },
     {
          {
          1000,
          03.05.2012
          }
     }
}
// Product document
```

Obrázek 3: Příklad nerelačního modelu dat (dokument JSON), zdroj: [5]

2.2 Hlavní kritéria NoSQL databáze

Flexibilita. Databáze NoSQL zpravidla nabízejí flexibilní schémata, která umožňují rychlejší vývoj a postupnou implementaci. Díky použití flexibilních datových modelů jsou NoSQL databáze vhodné pro částečně strukturovaná a nestrukturovaná data.

Škálovatelnost. Databáze NoSQL jsou navrženy tak, aby škálovaly pomocí distribuovaných hardwarových clusterů, spíše než pomocí upgradu komponent serveru.

Vysoký výkon. Databáze NoSQL jsou optimalizovány pro konkrétní datové modely a přístupové vzorce, což má za následek vyšší výkon než u relačních databází.

Rozsáhlé funkce. Databáze NoSQL poskytují rozhraní API (Application Programming Interface) a datové typy s rozsáhlými funkcemi, které jsou speciálně navrženy pro jejich příslušné datové modely.

Nestrukturovanost (schemaless). V databázích NoSQL, na rozdíl od relačních databází, není datová struktura regulovaná – do samostatného řádku nebo dokumentu lze přidat libovolné pole bez předběžných deklarativních změn v struktuře celé tabulky. Pokud je tedy nutné změnit datový model, pak jedinou dostatečnou akcí je odrazit změnu v kódu aplikace.

Důsledkem toho, že žádné schéma neexistuje je efektivní práce s „řídkými" (sparse) daty. Pokud má jeden dokument nějaké pole, ale druhý nikoli, nebude pro druhé pole vytvořeno žádné prázdné pole. [6]

2.3 Výhody NoSQL databáze

- Schopnost ukládat velké množství nestrukturovaných informací. NoSQL nemá žádná omezení týkající se typů uložených dat a v případě potřeby lze přidat nové datové typy.

- NoSQL databáze jsou více škálovatelné. Ačkoli je škálování podporováno v databázích SQL, vyžaduje mnohem více lidských a hardwarových prostředků.

- Horizontální škálování, tj. situace, když několik nezávislých počítačů je propojeno dohromady a každý z nich zpracovává svou vlastní část požadavků, umožňuje zvýšit kapacitu klastru přidáním nového serveru.

- Replikace, tj. kopírování aktualizovaných dat na jiné servery, což umožňuje větší odolnost vůči chybám a škálovatelnost systému.
  Existují dva typy takového procesu:

1) Typ replikace master-slave: existuje jeden hlavní server a několik podřízených serverů. Zápis lze provádět pouze na hlavní server, který přenáší změny na podřízené stroje. Tento typ replikace poskytuje dobrou škálovatelnost pro čtení, ale ne zápisy, protože zápisy jdou pouze na hlavní server. Tento typ replikace má své nevýhody – v případě selhání hlavního serveru je nutné vybrat nový hlavní server (automaticky nebo ručně). [7]

2) Typ replikace peer-to-peer – všechny uzly jsou stejné ve schopnosti obsluhovat požadavky na čtení a zápis. Informace o aktualizaci dat se přenáší ze serveru na server. [7]

- Sdílení, tj. rozdělení informací mezi různými síťovými uzly. Každý uzel odpovídá pouze za konkrétní sadu dat a zpracovává požadavky související pouze s touto sadou dat. NoSQL předpokládá, že sdílení je implementováno samotnou databází a bude probíhat automaticky.

- Využití cloudových výpočtů a úložiště. Slouží k testování a vývoji místního zařízení a následnému přesunu systému do cloudu.

- Rychlý vývoj. Databáze NoSQL nevyžadují mnoho přípravných prací, které jsou vyžadovány pro relační databáze.

- Mnoho řešení NoSQL má omezenou funkčnost, protože řeší určité problémy. Proto práce s takovými databázemi nevyžaduje hluboké znalosti dotazů SQL.

- Jednodušší technologie dotazů v NoSQL mají za důsledek menší počet chyb.

- Proprietární dotazovací jazyky moderních úložišť NoSQL jsou mnohem vhodnější pro provádění jednoduchých databázových manipulací.

2.4 Nevýhody NoSQL databáze

- Aplikace je silně svázána s konkrétním DBMS. Na rozdíl od toho, jazyk SQL je univerzální pro všechna relační úložiště, a proto, pokud bude změněn DBMS, nebude potřeba přepisovat celý kód.

- Omezená kapacita vloženého dotazovacího jazyka. SQL je velmi výkonným a propracovaným nástrojem pro manipulaci s daty. Téměř všechny dotazovací jazyky a

metody NoSQL storage API byly postaveny na vrcholu nějaké funkce SQL, ale mají při tom méně funkcí.

- Proces vytváření relačního úložiště zahrnuje fázi návrhu datového modelu. V této fázi lze navrhnout skutečně spolehlivý a pohodlný systém. Řešení NoSQL nevyžadují, aby před zahájením práce bylo definováno schéma databáze, takže během procesu vývoje existuje možnost narazit na nepředvídané potíže, které mohou vést k opuštění tohoto konkrétního řešení NoSQL.

- Problémy s rychlým přechodem z jedné nerelační databáze do druhé.

- Častá potřeba vyvinout vlastní nástroje pro práci s databází.

- Je mnohem snazší najít specialisty s dobrou znalostí jazyka SQL, zatímco jen velmi málo lidí se zajímá o specifika API některých řešení NoSQL na vysoké úrovni.

- Většina společností prostě nemá takové objemy dat a jiné konkrétní provozní podmínky, ve kterých by byla NoSQL řešení prospěšná jako hlavní databáze. Úložiště NoSQL fungují dobře v symbióze s relačními databázemi. Například v systémech, kde většinu informací ukládá SQL a za „mezipamět" odpovídá NoSQL.

- Velmi omezená podpora transakcí

- Nerelačním systémům stále chybí univerzálnost, spolehlivost, konzistence a předvídatelnost.

2.5 Výsledné porovnání SQL vs. NoSQL

V tabulce 1 jsou shrnuty hlavní rozdíly mezi databázemi NoSQL a SQL.

Tabulka 1 : Porovnání relačních a nerelačních databází, zdroj: autor

Kritérium / Typ databáze	Relační databáze (SQL)	Nerelační databáze (NoSQL)
Vhodné pracovní vytížení	Relační databáze jsou navrženy pro transakční a vysoce konzistentní aplikace pro zpracování transakcí v reálném čase (OLTP) a jsou	Databáze NoSQL jsou navrženy pro práci s řadou šablon pro přístup k datům, včetně aplikací s nízkou latencí. Prohledávání

	vhodné pro analytické zpracování v reálném čase (OLAP).	databází NoSQL je určeno pro analýzu částečně strukturovaných dat.
Datový model	Relační model normalizuje data a transformuje je do tabulek s řádky a sloupci. Schéma pevně definuje tabulky, řádky, sloupce, indexy, vztahy mezi tabulkami a další databázové prvky. Taková databáze zajišťuje integritu referenčních dat ve vztazích mezi tabulkami.	Databáze NoSQL poskytují celou řadu datových modelů, jako jsou páry klíč-hodnota, dokumentové, sloupcové a grafové databáze, které jsou optimalizovány pro vysoký výkon a škálovatelnost.
ACID vlastnosti	Relační databáze poskytují sadu vlastností ACID: atomicita, konzistence, izolace a spolehlivost. Atomicita vyžaduje přísné dokončení transakce. Konzistence znamená, že jakmile transakce skončí, data musí odpovídat schématu databáze. Izolace vyžaduje, aby paralelní transakce probíhaly odděleně od sebe navzájem. Spolehlivost označuje schopnost zotavit se do posledního uloženého stavu po neočekávaném selhání systému nebo výpadku napájení.	Databáze NoSQL často nabízejí kompromis uvolněním přísných požadavků na vlastnosti ACID ve prospěch flexibilnějšího datového modelu, který lze škálovat. Díky tomu je NoSQL skvělou volbou pro případy použití s vysokou propustností a nízkou latencí, které je třeba škálovat nad rámec jedné instance.
Výkon	Výkon závisí hlavně na diskovém subsystému.	Výkon obvykle závisí na velikosti základního

	Optimalizace dotazů, indexů a struktury tabulky je často nutná k dosažení nejlepšího výkonu.	hardwarového klastru, latenci sítě a volající aplikaci.
Škálování	Relační databáze se obvykle zvětšují při zvýšení výpočetního výkonu hardwaru nebo přidáním samostatných kopií pro pracovní zatížení.	NoSQL databáze jsou obecně vysoce sdílené kvůli škálovatelným vzorům přístupu založeným na distribuované architektuře. To zvyšuje propustnost a poskytuje trvalý výkon v téměř neomezeném měřítku.
API (Application Programming Interface)	Požadavky na vstup a načítání dat jsou psány v SQL. Tyto dotazy jsou analyzovány a prováděny relační databází.	Objektově orientovaná rozhraní API umožňují vývojářům aplikací snadno zapisovat a načítat datové struktury. Pomocí použití klíčů sekcí mohou aplikace prohledávat páry klíč-hodnota, sady sloupců nebo polostrukturované dokumenty obsahující sériové objekty a atributy aplikace.

Z výše uvedené tabulky lze udělat závěr, že mezi relačními a nerelačními databázemi neexistuje žádná opozice, navíc se často používají společně k řešení různých problémů:

- **Relační databáze (SQL)** jsou vhodné pro ukládání strukturovaných dat, zejména v případech, kdy je jejich integrita nesmírně důležitá. Je také lepší zvolit tento model, pokud projekt potřebuje technologii založenou na standardech, při jejímž použití je třeba počítat s velkým počtem doplňků a spoustou zkušeností vývojářů.

- **Nerelační databáze (NoSQL)** jsou potřebné, pokud jsou požadavky na data nejasné, vágní a mohou se měnit s růstem a vývojem projektu. A také v případech, kdy hlavními požadavky na databáze jsou vysoká rychlost a obrovský počet dat.

3 Typy NoSQL databází

Díky popularizaci Big Data je potřeba ukládat stále více dat, přičemž k datům v databázi je třeba přistupovat co nejrychleji, ale relační databáze má omezení rychlosti kvůli operaci spojení. Mnoho podniků již používá databáze NoSQL, která splňuje požadavek rychlého přístupu k datům. Je však důležité vždy vybrat vhodnou databázi NoSQL podle konkrétních požadavků, protože toto rozhodnutí následně ovlivní výkonnost podnikových operací.

Celkově existují čtyři hlavní typy úložišť NoSQL. Liší se v datovém modelu, přístupu k distribuci a replikaci, díky čemuž mohou být v různé míře vhodné pro určité typy konkrétních úkolů. [6]

3.1 Databáze klíč-hodnota

Databáze využívající páry klíč-hodnota udržují vysokou oddělitelnost a poskytují horizontální škálování, což není možné při použití jiných typů databází. Dobrým případem použití pro databáze s klíč-hodnotou jsou hry, reklama a aplikace IoT (Internet of Things). [6]

Úložiště klíč-hodnota je nejjednodušším typem databáze. Ve skutečnosti jde o asociativní pole, kde každá hodnota je spojena s vlastním jedinečným klíčem. Jednoduchost tohoto typu úložiště otevírá obrovskou škálovatelnost, protože nejsou vyžadována žádná schémata konstrukce databáze, neexistuje žádné spojení mezi hodnotami, ve skutečnosti je počet prvků asociativního pole omezen pouze výpočetním výkonem. Proto je tento typ úložiště zajímavý především pro společnosti poskytující cloudové hostingové služby.

Na druhou stranu díky jednoduchosti úložišť pro databází typu klíč-hodnota je velmi obtížné pracovat s většinou obvyklých operací s hodnotami tohoto uložiště: pokud lze klíče „obsluhovat" jakýmkoli způsobem, pak může být pokus o hledání podle hodnot trvat mnohem déle než v relační databázi. A spolu s omezenou sadou manipulací nad hodnotami paměťových buněk existuje skutečný problém s rychlou analýzou informace dostupné v databázi a také se shromažďováním statistiky.

Proto se taková úložiště používají v případech, kdy konkrétní obsah samostatné buňky není pro operátora databáze zajímavý – jinými slovy, neexistují žádná spojení mezi jednotlivými buňkami úložiště. Databáze typu „klíč-hodnota" se nehodí jako úplná náhrada za relační databáze, ale našly svou aplikaci jako mezipaměť pro objekty, protože mezi objekty v mezipaměti různých uživatelů neexistují žádná spojení, důležitá je pouze rychlost přístupu do mezipaměti a schopnost rychle změnit rozsah systému.

Klíčem v takové databázi obvykle slouží jedinečné ID, které odkazuje na data, se kterými je spojeno. Mezi možné operace, které lze použít s hodnotou klíče, patří získání hodnoty klíče (get), uvedení nebo přiřazení hodnoty klíče (put) a odstranění klíče (delete). Hodnotou může být jakýkoli

objekt, jako je řetězec, číslo, datum, pole, JSON atd. Celý systém je bez schémat. Aplikace je zodpovědná za pochopení typu objektu a jeho odpovídající analýzu.

Nejznámějšími příklady tohoto typu DBMS jsou Amazon DynamoDB, Berkeley DB, MemcacheDB, Redis a Riak.

Příklad úložiště klíč-hodnota je uveden na obr.4. Klíč adresy je tady přidružen k hodnotě JSON, která obsahuje tři atributy: ulice, město a stát. V relační databázi by takové uložiště bylo neplatné.

Key	Value
ID	"a78d1238dfedhz"
Building	"COOR Hall"
Address	{ street: "976 S Forest Mall", city: "Tempe", state: "Arizona" }

Obrázek 4: Příklad úložiště klíč-hodnota, zdroj: [8]

3.2 Dokumentové databáze

V aplikačním kódu jsou data často reprezentována jako objekt nebo dokument ve formátu podobném JSON, protože pro vývojáře je to efektivní a intuitivní datový model. Dokumentové databáze umožňují vývojářům ukládat a dotazovat data v databázi pomocí stejného modelu dokumentu, jaký používají v kódu aplikace. Flexibilní, polostrukturovaná, hierarchická povaha dokumentů a databází dokumentů jim umožňuje rozvíjet se v souladu s potřebami aplikací. Model dokumentu funguje dobře v adresářích, uživatelských profilech a systémech správy obsahu, kde je každý dokument jedinečný a časem se mění.

Dokumentově orientovaná databáze je systém pro ukládání hierarchických datových struktur (dokumentů) se stromovou nebo lesní strukturou. Stromová struktura začíná v kořenovém uzlu a může mít více vnitřních a listových uzlů. Listové uzly obsahují listová data, která se po přidání zadají do základních indexů, díky nimž lze provádět rychlé vyhledávání i při poměrně složité celkové struktuře úložiště. Ve skutečnosti jsou dokumentové databáze složitější verzí úložišť typu klíč-hodnota a stále nejsou příliš dobré pro systémy, které implikují mnoho vztahů mezi prvky, ale umožňují načítání na vyžádání bez úplného načtení jednotlivých dokumentů do paměti RAM.

Vyhledávače následně umožňují najít jak celé dokumenty, tak jejich části a stromová struktura umožňuje uspořádat samostatné sbírky dokumentů podle stejného typu nebo podobného tématu. [6]

Například při vytváření hudebního úložiště lze vytvořit sbírku hudby z 80. let, vytvořit v ní samostatné sbírky podle roku a uvnitř nich oddělit dokumenty s jednotlivými skladbami z alb vydaných v tomto roce. Pokud ale uživatel chce vidět hodnocení nejpopulárnějších skladeb určitého desetiletí, bude tento požadavek trvat dlouho, protože bude potřeba prohlédnout každý dokument celé databáze. Lze tedy dojít k závěru, že dokumentově orientované databáze naleznou své uplatnění v úkolech, kde je vyžadováno uspořádané ukládání informací, ale mezi daty není mnoho spojení a není třeba o nich neustále shromažďovat statistiky. Dokumenty nevyžadují definici schématu – to znamená, že každý jednotlivý dokument může obsahovat libovolný počet jedinečných polí – na rozdíl od relačních databází, ve kterých se při pokusu o uložení heterogenních dat nevyhnutelně objeví prázdná pole.

Tento typ databáze vyžaduje, aby jeho uložená hodnota (nazývaná dokument) byla strukturována a kódována metadaty. Mezi běžné kódování patří XML, JSON, BSON a pro prostorová data je GeoJSON dobře akceptován mnoha databázemi NoSQL pro ukládání dokumentů.

Příklady tohoto typu DBMS: CouchDB, Couchbase, MarkLogic, MongoDB, eXist.

Na obr.5 je ukázána událost zemětřesení uložena v MongoDB jako dokument.

Dokument je ve formátu GeoJSON s vlastnostmi zemětřesení a jeho geometrií.


```
{
    "_id": {
        "$oid": "59d5495476bbdae070419021"
    },
    "type": "Feature",
    "properties": {
        "mag": 3.2,
        "place": "151km SSW of Chirikof Island, Alaska",
        "time": 1507149758758,
        "updated": 1507150112184,
        "tz": -600,
        "url": "https://earthquake.usgs.gov/earthquakes/eventpage/ak16989132",
        "type": "earthquake",
        "title": "M 3.2 - 151km SSW of Chirikof Island, Alaska"
    },
    "geometry": {
        "type": "Point",
        "coordinates": [
            -156.904,
            54.6732,
            0
        ]
    }
}
```


Obrázek 5: Příklad dokumentové databáze, zdroj: [8]

3.3 Sloupcové databáze

Sloupcové databáze obsahují data uspořádaná v řídké matici, jejíž řádky a sloupce se používají jako klíče. Tato úložiště mají mnoho společného s dokumentovanými databázemi – oni také obsahují systémy pro správu obsahu, registrace událostí, blogy. Přestože tento typ databáze vypadá podobně jako „sloupcové úložiště“ (což je v podstatě relační databáze se samostatným úložištěm sloupců), nejsou tyto databáze ve skutečnosti stejné.

Hlavní rozdíl mezi sloupcovými a jinými typy databází je způsob, jakým ukládají data na disk. Relační databáze vytváří soubor pro každou tabulku a postupně ukládají hodnoty pro všechny řádky. Na rozdíl od nich, sloupcové databáze vytváří soubor pro každý sloupec tabulek. [6]

Tato struktura umožňuje agregovat data a provádět určité dotazy efektivněji, ale je nutné se ujistit, že data splňují všechna omezení takových databází.

Obvykle se tato úložiště používají pro indexování webu a další úkoly, které zahrnují obrovské množství dat. Dotazování na řádky je náročné na paměť a vyžaduje přístup na disk, zejména když každý řádek obsahuje mnoho sloupců. V takovém uložišti jsou sloupce seskupeny do „rodin sloupců“ a každá taková „rodina“ může mít neomezený počet sloupců uvnitř. Tímto způsobem je mnohem snazší dotazovat se na celou rodinu sloupců pro všechny řádky.

Příklady tohoto typu DBMS jsou: HBase, Cassandra, Hypertable, SimpleDB.

V uvedeném příkladu jsou zobrazeny různé struktury úložiště založeného na řádcích (vlevo) a úložiště založeném na sloupcích (vpravo) (obr.6).

Name	GPA	Year	Program
Tom	3.4	2	GEOG

Mary	4.0	1	CS

David	3.7	3	ENV

Name
Tom
Mary
David

GPA
3.4
4.0
3.7

Year
2
1
3

Program
GEOG
CS
ENV

Obrázek 6: Úložiště založené na řádcích versus úložiště založené na sloupcích, zdroj: [8]

3.4 Grafové databáze

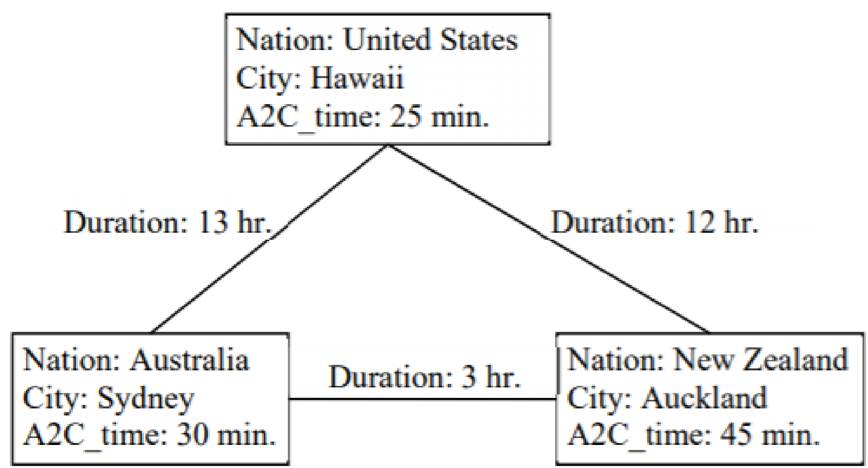
Grafové databáze usnadňují vývoj a spouštění aplikací, které pracují se složitými datovými sadami. Typickými příklady použití grafových databází jsou sociální sítě, poradenské služby a grafy znalostí. Takový model je zevšeobecněním síťového datového modelu a vyznačuje se silným spojením mezi uzly.

Grafové databáze jsou nejvhodnější pro projekty zahrnující přirozenou datovou strukturu grafů. V takových úkolech tento typ je daleko před relačními databázemi z hlediska výkonu, snadnosti provádění změn a viditelnosti prezentovaných informací. Některé databáze mají speciální optimalizační mechanismy pro práci s jednotkami SSD. Pro práci s dostatečně velkými grafy se používají algoritmy, které naznačují částečné umístění grafu do paměti RAM. [6]

Nejznámější grafy DBMS jsou ArangoDB, FlockDB, Giraph, HyperGraphDB, Neo4j, OrientDB.

Jako příklad je uvedena letecká společnost z Oceánie.

Letecká společnost musí mezi některými městy ukládat letové hodiny. Data mohou být uložena v databázi grafů, jak je znázorněno na obr.7. V této databázi grafů každý vrchol obsahuje některá data, například národnost, město a A2C_time (čas z letiště do centra města) a každá hrana představuje trvání letu mezi dvěma městy.



Obrázek 7: Příklad dokumentové databáze, zdroj: [6]

3.5 Specifikace dat pro následný výběr databázového nástroje

Vzhledem k existenci několika typu NoSQL databází bylo potřeba nejdřív definovat pro jaká data a za jakým účelem bude databáze tvořena.

Pro danou databázi byla využita data poskytnuta Fakultou dopravní z dopravních senzorů, které jsou umístěny v úsecích dopravních komunikací po celé Praze. Data byla sbírána v roce 2018, vždy od 9. října do 18. listopadu ve formátu .txt. Tato data byla rozdělena do jednotlivých profilů, což jsou konkrétní body umístění jednoho nebo skupiny detektorů ve dvou směrech. Jeden soubor .txt vždy obsahoval data sbírána během jednoho dne na každém z profilu. Takovým způsobem z jednoho profilu vznikalo 41 souborů (tj. jeden soubor na každý den z 9. října po 18. listopadu). Data byla sbírána každých 5 minut na každém z detektorů, kterých v rámci práce bylo dohromady 60.

Každý z těchto souborů obsahoval následující data:

- Datum a čas kdy byla data načtena do systému ve formátu YYYY-MM-DD hh:mm:ss+02:00
- Odesílatel
- ID detektoru
- Typ detektoru
- Počet aut, včetně rozdělení na osobní, nákladní a jejich součet
- Rychlost aut, včetně rozdělení na osobní, nákladní a jejich součet
- Obsazenost detektoru, včetně rozdělení na osobní, nákladní a jejich součet

Pro tato konkrétní dopravní data na ústavu K620 Fakulty dopravní již byla navržena relační databáze. [10]

Nicméně vzhledem k tomu, že v budoucnu se objem takových dat může postupně zvětšovat, bylo rozhodnuto, že databáze typu NoSQL bude vhodná pro použití.

3.6 Výsledný výběr databázového nástroje pro vytvoření distribuovaného prostředí

Pro následnou realizaci bylo potřeba vybrat jeden ze čtyř existujících typů NoSQL databáze s ohledem na požadavky:

- Databáze klíč-hodnota
- Dokumentová databáze
- Sloupcová databáze

- Grafová databáze

Jako první byly vyloučeny grafové databáze z důvodu jejich specifických datových struktur, které jsou představeny většinou samotnými grafy. Bylo rozhodnuto, že v případě práce s dopravními daty takový typ databáze nebude vhodný.

Jako druhé byly vyloučeny databáze typu klíč-hodnota kvůli obtížnosti práce s většinou obvyklých operací s hodnotami uložiště: pokus o hledání podle hodnot v nich trvá dlouho a z toho plyne problém s rychlou analýzou informace dostupné v databázi a také se shromažďováním statistiky, což může být docela důležité při práci s dopravními daty. DynamoDB má dokonce omezenou podporu pro různé datové typy. Například podporuje pouze jeden číselný typ a vůbec nepodporuje typ „datum“, což může být také problémem při práci s dopravními daty.

Výběr mezi dokumentovými a sloupcovými databázemi se prováděl v odstavci 3.6.1 již na základě porovnání konkrétních zástupců databázových nástrojů každého typu, protože, jak již bylo zmíněno, tyto dva typy databáze mají mezi sebou hodně podobného: obě jsou schopny pracovat s nestrukturovanými daty v reálném čase pomocí horizontálního škálování. Obě jsou open-source, tj. databáze s otevřeným zdrojovým kódem, což znamená že existuje možnost stáhnout databázové balíčky, nastavit je a nakonfigurovat bez jakýchkoli nákladů.

Mají ale i své určité rozdíly, které jsou ukázány během porovnání dvou konkrétních nástrojů:

- MongoDB (jako dokumentová databáze)
- Apache Cassandra (jako sloupcová databáze)

3.6.1 Dostupnost dat

Jedním z nejvýznamnějších rozdílů mezi MongoDB a Cassandra je jejich strategie týkající se dostupnosti dat. Tato funkce závisí na počtu hlavních podřízených (master slaves) v clusteru.

MongoDB má jeden hlavní a několik podřízených uzlů. Pokud hlavní uzel spadne, převezme jeho roli jeden z podřízených uzlů. Ačkoli strategie automatického převzetí služeb při selhání zajišťuje zotavení, může trvat až minutu, než se podřízený uzel stane hlavním. Během této doby databáze není schopna reagovat na požadavky.

Cassandra na rozdíl od MongoDB používá jiný model. Místo toho, aby v ni existoval hlavní uzel, využívá uvnitř clusteru více uzlů, které jsou stejně důležité, díky čemuž pak nejsou přítomné žádné „prostoje“. Redundantní model zajišťuje vždy vysokou dostupnost.

Tím pádem, pokud aplikace vyžaduje vysokou dostupnost a závisí na okamžitých odpovědích na požadavky, je vhodnější volbou Cassandra.

3.6.2 Škálovatelnost

Škálovatelnost je funkce přímo spojená s modelem clusteru. Cassandra a MongoDB mají v tomto významné rozdíly.

V *MongoDB* pouze hlavní uzel může psát a přijímat vstup. Mezitím se podřízené uzly používají pouze ke čtení. Kvůli existenci jediného hlavního uzlu, je *MongoDB* omezena z hlediska škálovatelnosti zápisu.

U *Cassandra* existence více rovných uzlů zvyšuje možnosti psaní a tím pádem umožňuje této databázi koordinovat zápisy současně. Čím více je uzlů je v clusteru, tím vyšší je rychlost zápisu (škálovatelnost).

Tím pádem, pokud se rychlost a škálovatelnost psaní považují za prioritu, Cassandra je lepší možnosti.

3.6.3 Dotazovací jazyk

Dalším rozlišujícím faktorem je, zda je potřeba databáze podporující dotazovací jazyk.

MongoDB používá dotazy strukturované do fragmentů JSON a dosud nemá podporu žádného jazyka. Správa je ale dost snadná.

Na rozdíl od MongoDB má *Cassandra* svůj vlastní dotazovací jazyk s názvem CQL (Cassandra Query Language). Jeho syntaxe je podobná SQL, ale stále má určitá omezení. Databáze má v zásadě jiný způsob ukládání a obnovy dat, protože není relační.

Tím pádem, pokud je nutná podpora dotazovacího jazyka, práce s Cassandra bude pohodlnější.

3.6.4 Výsledný výběr

Cassandra je jediná databáze, která zapisuje data rychleji, než čte. To je způsobeno skutečností, že zápis do něj se úspěšně dokončí (nejrychlejším způsobem) ihned po zápisu na disku. Čtení však vyžaduje kontroly, několik čtení z disku a výběr nejnovějšího záznamu. Cassandra je robustní a poměrně rychlý škálovatelný datový archiv.

Do *MongoDB* se píše pomalu, ale relativně rychle se čte. Pokud se data (pracovní sada) nevejdou do paměti, proces se výrazně zpomalí.

Vzhledem k tomu, že při práci s konkrétními dopravními daty, obsahující záznamy o rychlostech a intenzitách každých 5 vteřin z 60 detektorů je očekáváné celkem velké zatížení a je potřeba rychlý zápis, Cassandra s několika rovnými uzly nejspíš poskytne lepší výsledky. Kromě toho, dotazovací jazyk Cassandra (CQL) je o hodně jednodušší v případě, že existovala nějaká praxe s práci v SQL.

Tím pádem, po zvážení několika faktorů byl vybrán pro následnou realizaci konkrétní databáze a vytvoření prostředí pro práci s dopravními daty nástroj Apache Cassandra.

4 Praktická část

Tato část práce je věnována především distribuovanému systému pro správu databází Apache Cassandra, jeho instalaci a výsledné realizaci konkrétní databáze za použití vybraného databázového nástroje, a také teoretické základně tohoto systému, především CAP teorému.

4.1 Věta CAP

Věta CAP byla navržena Ericem Brewerem v roce 2000 a uvádí, že distribuovaný systém nemůže současně poskytovat všechny žádoucí vlastnosti C (consistency = konzistence), A (availability = dostupnost) a P (partition tolerance = tolerance oddílů, tj. odolnost k přerušení). Podle věty CAP se konzistence týká schopnosti všech uzlů vidět stejná data ve stejnou dobu. Dostupnost odkazuje na záruku, že všechna čtení a zápisy budou vždy úspěšné a odolnost k přerušení odkazuje na záruku, že vlastnosti CAP systému budou zachovány i v případě sítě, která brání některým strojům nebo počítačům vzájemně komunikovat. [9]

Věta CAP je z tohoto důvodu potřeba vždy, pokud počet uzlů je větší, než 1. V případě Cassandra je systém obvykle klasifikován jako AP, což znamená, že poskytuje dostupnost a odolnost k přerušení na úkor konzistence. Protože Cassandra nemá hlavní uzel, musí být všechny uzly k dispozici nepřetržitě. Cassandra však poskytuje případnou konzistenci tím, že umožňuje klientům kdykoli zapisovat do libovolných uzlů a co nejrychleji sladit nesrovnalosti (když je oddíl vyřešen, databáze AP obvykle znovu synchronizují uzly, aby opravily všechny nekonzistence v systému).

Dostupnost uvádí, že každý požadavek dostane odpověď na úspěch / selhání. Dosažení dostupnosti v distribuovaném systému vyžaduje, aby systém zůstal funkční 100% času. Každý klient dostane odpověď, bez ohledu na stav kteréhokoli jednotlivého uzlu v systému.

Odolnost k přerušení uvádí, že systém pokračuje v provozu i přes počet zpráv zpožděných sítí mezi uzly. Systém, který je odolný vůči přerušení, může vydržet jakékoli množství menších selhání, které ale následně nevedou k selhání celé sítě. Datové záznamy jsou dostatečně replikovány napříč kombinacemi uzlů a sítí, aby byl systém udržen v provozu bez ohledu na občasné výpadky.

Přestože Apache Cassandra spadá pod AP systém, lze ho dále vyladit pomocí replikačního faktoru (počet kopií dat) a úrovně konzistence (čtení a zápis).

4.1.1 Replikační faktor

Cassandra ukládá repliky dat na více uzlech, aby zajistila spolehlivost a odolnost proti chybám. Strategie replikace pro každý klíčový prostor určuje uzly, na kterých jsou umístěny repliky. [11]

Celkový počet replik prostoru klíčů napříč clusterem Cassandra se označuje jako replikační faktor prostoru klíčů. Replikační faktor = 1: znamená, že v clusteru Cassandra existuje pouze jedna kopie každého řádku. Replikační faktor = 2: znamená, že existují dvě kopie každého řádku, přičemž je každá kopie umístěna v jiném uzlu. Všechny repliky jsou stejně důležité; neexistuje žádná primární ani hlavní replika.

V produkčním systému se třemi nebo více uzly Cassandra v každém datovém centru je výchozí faktor replikace pro klíčový prostor = 3. Obecně platí, že faktor replikace by neměl překročit počet uzlů Cassandra v clusteru.

Pokud do clusteru budou přidány další uzly Cassandra, výchozí faktor replikace to neovlivní.

Například pokud počet uzlů Cassandra bude zvýšen na šest, ale bude ponechán faktor replikace tři, nebude zajištěno, že všechny uzly Cassandra budou mít kopii všech dat. Pokud jeden z uzlů spadne, vyšší faktor replikace bude znamenat vyšší pravděpodobnost toho, že data v clusteru existují na jednom ze zbývajících uzlů. Nevýhodou vyššího faktoru replikace je pak zvýšená latence při zápisu dat. [11]

4.1.2 Úroveň konzistence

Úroveň konzistence Cassandra je definována jako minimální počet uzlů Cassandra, který musí potvrdit operaci čtení nebo zápisu, před tím, než bude možné operaci považovat za úspěšnou. [11]

Při každém čtení a zápisu může klient určit požadovanou úroveň konzistence: ANY, ONE, TWO, THREE, QUORUM, SERIAL, ALL a další. Například výchozí úroveň ONE říká, že požadavek by měl dosáhnout alespoň jednoho uzlu, který bude odpovědný za uložení řádku. Úroveň konzistence QUORUM pak znamená, že požadavek by měl být přijat většinou uzlů, například 2 ze 3. To umožňuje vybírat mezi rychlostí dotazu a spolehlivostí.

Výpočet hodnoty LOCAL_QUORUM pro datové centrum je:

LOCAL_QUORUM = (replikační_faktor / 2) + 1

Pokud výchozí faktor replikace se třemi uzly Cassandra je tři, je výchozí hodnota LOCAL_QUORUM = (3/2) +1 = 2 (hodnota se zaokrouhlí na celé číslo dolů).

S LOCAL_QUORUM = 2 musí alespoň dva ze tří uzlů Cassandra v datovém centru reagovat na operaci čtení / zápis, aby operace proběhla úspěšně. U clusteru Cassandra se třemi uzly by cluster mohl tolerovat výpadek jednoho uzlu v datovém centru.

Zadáním úrovně konzistence jako LOCAL_QUORUM se lze vyhnout latenci požadované ověřováním operací napříč více datovými centry. Pokud by klíčový prostor používal jako úroveň

konzistence hodnotu Cassandra QUORUM, musely by být operace čtení / zápisu ověřovány ve všech datových centrech. [11]

4.2 Apache Cassandra: první kroky

Apache Cassandra je distribuovaný a decentralizovaný úložný systém (databáze) s otevřeným zdrojovým kódem, určený ke správě velmi velkého množství strukturovaných dat. Poskytuje vysoce dostupnou službu bez jediného bodu selhání a distribuci dat mezi uzly clusteru.

4.2.1 Instalace a napojení uzlů v prostředí Apache Cassandra

Prvním krokem bylo potřeba nainstalovat pomocné programy ke spuštění Cassandra. Vzhledem k tomu, že pro práci s Cassandra je vhodnější prostředí Linux, nejprve bylo potřeba nainstalovat prostředí VirtualBox (Oracle VM) pro virtuální zobrazení Linux ve Windows. Následně bylo potřeba nainstalovat samotný soubor Debian 10 (64bit), který obsahoval systémovou verzi Cassandra a 3 funkční nody. Vzhledem k tomu, že v rámci práce nebyly k dispozici fyzické stroje pro vytvoření clusteru se 3 uzly, byl použit virtuální stroj, ve kterém byly tyto uzly nasimulovány pomocí testovacího prostředí. V případě různých serverů při budoucím nasazení by bylo potřeba na každý z nich instalovat Cassandra z balíčku zvlášť.

Přestože systémová Cassandra automaticky běží po startu, všechny 3 nody jsou jinými instancemi Cassandra a nespouští se defaultně. Velmi důležité je také to, že aby nody fungovaly z jednoho počítače v rámci tohoto testovacího prostředí je potřeba aby systémová Cassandra byla vypnuta z důvodu duplikace.

Na obr.8 je ukázán dotaz na Cassandra, pomocí kterého jde zjistit, jestli je aktivní v danou chvílí.

```
user@bigdata:~$ systemctl status cassandra
● cassandra.service - LSB: distributed storage system for structured data
   Loaded: loaded (/etc/init.d/cassandra; generated)
   Active: active (running) since Thu 2021-03-25 11:13:00 CET; 1h 35min ago
     Docs: man:systemd-sysv-generator(8)
  Process: 564 ExecStart=/etc/init.d/cassandra start (code=exited, status=0/SUCCESS)
    Tasks: 66 (limit: 4689)
   Memory: 1.6G
   CGroup: /system.slice/cassandra.service
           └─712 java -ea -da:net.openhft... -XX:+UseThreadPriorities -XX:+HeapDumpOnOutOfM
lines 1-9/9 (END)
```

Obrázek 8: Dotaz na Cassandra (je aktivní), zdroj: autor

Aby se Cassandra zastavila byl použit následující příkaz z obr.9.

```
user@bigdata:~$ sudo systemctl stop cassandra.service
user@bigdata:~$
```

Obrázek 9: Zastavení Cassandra, zdroj: autor

A po zjištění statusu z obr.10 lze vidět, že Cassandra je úspěšně pozastavená.

```
user@bigdata:~$ systemctl status cassandra.service
● cassandra.service - LSB: distributed storage system for structured data
   Loaded: loaded (/etc/init.d/cassandra; generated)
   Active: inactive (dead) since Thu 2021-03-25 13:23:42 CET; 2min 18s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 564 ExecStart=/etc/init.d/cassandra start (code=exited, status=0/SUCCESS)
  Process: 11864 ExecStop=/etc/init.d/cassandra stop (code=exited, status=0/SUCCESS)
```

Obrázek 10: Dotaz na Cassandra (je neaktivní), zdroj: autor

Pokud bude potřeba zase zpustit Cassandra, lze použit podobný příkaz jako na obr.9, s tím rozdílem že místo slova „stop“ je potřeba použit „start“. Kromě toho jde použit i slovo „restart“, například pokud něco bude změněno v konfiguračním souboru.

Následně lze zjistit i status uzlů. Na to existuje příkaz „nodetool status“ (obr.11).

```
user@bigdata:~$ nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load        Tokens  Owns (effective)  Host ID                               Rack
UN  127.0.0.1  209.71 KiB  16      ?                 b23ff83f-2da0-4fed-8836-1c8e7b79e8dc  rack1
```

Obrázek 11: Dotaz na 1.uzel, zdroj: autor

Pomocí tohoto příkazu jde zjistit, jestli uzel běží a jaké má příznaky. (jeho IP adresa, kolik má tokenů apod.) UN tady znamená UP NODE, tj. že uzel v tuto chvílí skutečně běží. Momentálně byl zpuštěn uzel č.1.

Stejným způsobem byly spuštěny i ostatní dva uzly, a z obr.12 lze vidět, jak se uzly postupně připojovaly. UJ u třetího uzlu tady znamená UP JOINING, což je v podstatě proces připojení.

```
root@bigdata:~# /root/node3/bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load        Tokens  Owns (effective)  Host ID                               Rack
UJ  127.0.0.3  90.52 KiB   128     ?                 8960c1b7-92ee-4b64-8d7e-170a6ccdb05e  rack1
UN  127.0.0.2  100.33 KiB  128     ?                 84292779-4457-41e6-a9b9-06bc817c506d  rack1
UN  127.0.0.1  76.1 KiB    128     ?                 a0c2d2b6-3e19-4ac5-836e-6f51c112c467  rack1
```

Obrázek 12: Připojení třetího uzlů, zdroj: autor

Nakonec z obr.13 je vidět, že všechny uzly jsou v režimu UN (UP NODE) a tím pádem všechny tři uzly v tuto chvílí jsou funkční.

```
root@bigdata:~# /root/node3/bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns (effective)  Host ID                               Rack
UN  127.0.0.3  76.02 KiB  128     ?                 8960c1b7-92ee-4b64-8d7e-170a6ccdb05e  rack1
UN  127.0.0.2  76.05 KiB  128     ?                 84292779-4457-41e6-a9b9-06bc817c506d  rack1
UN  127.0.0.1  95.8 KiB   128     ?                 a0c2d2b6-3e19-4ac5-836e-6f51c112c467  rack1

root@bigdata:~#
```

Obrázek 13: Funkční uzly, zdroj: autor

4.2.2 Proces vytvoření jednoduché tabulky v prostředí Cassandra

Hlavním cílem použití prostředí Cassandra je vytvoření databáze, tj. tabulek spojených mezi sebou. Pro tento cíl je potřeba aktivace speciálního jazyka Cassandra, tzv. Cassandra Query Language (CQL).

Aktivace probíhá pomocí příkazu *cqlsh.* (obr.14)

```
root@bigdata:/home/user# cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 4.0-beta4 | CQL spec 3.4.5 | Native protocol v4]
Use HELP for help.
```

Obrázek 14: Spuštění jazyka Cassandra, zdroj: autor

Následně je potřeba zadat důležitá základní nastavení: název klíčového prostoru (KEYSPACE) a nastavení replikace, která se skládá ze dvou vlastností: kategorie replikace (class) a replikační faktor pro každý uzel.

Existuje dvě nejrozšířenější kategorie replikace: 'SimpleStrategy' a 'NetworkTopologyStrategy'. Kategorie 'SimpleStrategy' není schopná dělat nic jiného než jenom postupně zapisovat zadaná data. Kategorie 'NetworkTopologyStrategy' již je schopná porozumět topologii konkrétní sítě, tj. tomu, že v ní existují jeden nebo více uzlů a že uzly mají své serverové stojany.

Dále se ukazuje replikační faktor, který odpovídá za počet kopií.

Na obr.15 lze vidět příklad KEYSPACE s názvem *users*, s kategorii replikace 'NetworkTopologyStrategy' a replikačním faktorem = 3.

```
cqlsh> CREATE KEYSPACE users
   ... WITH replication = {
   ... 'class': 'NetworkTopologyStrategy',
   ... 'datacenter1' : 3
   ... };
```

Obrázek 15: Příklad klíčového prostoru, zdroj: autor

Následně lze vytvořit již konkrétní tabulku.

Na obr.16 lze vidět příklad takové tabulky s názvem users_by_city, do které následně bude možné zapsat název města a také unikátní id, příjmení, jméno, adresu a email konkrétního člověka (lze pozorovat podobnost jazyka CQL a SQL).

```
cqlsh> CREATE TABLE users.users_by_city (
   ... city text,
   ... user_id UUID,
   ... last_name text,
   ... first_name text,
   ... address text,
   ... email text,
   ... PRIMARY KEY ((city), user_id));
```

Obrázek 16: Příklad tabulky, zdroj: autor

Primární klíč se skládá ze dvou částí: Klíč oddílu (Partition Key) a Seskupení sloupců (Clustering Columns). Funkce Partition Key odpovídá za definování sekcí, Clustering Columns definují pořadí řazení a také ujišťují se, že primární klíč je skutečně jedinečný a unikátní.

4.2.3 Základy tvoření datového modelu a principy modelování

Existují 4 principy modelování dat v Cassandra, které odrážejí klíčové kroky [12]:

1) Znalost dat
2) Znalost dotazů
3) Vnoření dat
4) Duplicita dat

Znalost dat, tj. jejich pochopení je základem pro úspěšný návrh, protože data jsou pak zachycená konceptuálním datovým modelem. Proto je potřeba definovat co je uloženo v databázi a zachovat vlastnosti, aby byla data správně uspořádána. Součásti konceptuálního datového modelu zahrnují:

- Subjekty (entities)
- Vztahy (relationships)
- Atributy (attributes)
- Klíče (keys)
- Omezení mohutnosti (cardinality constraints)

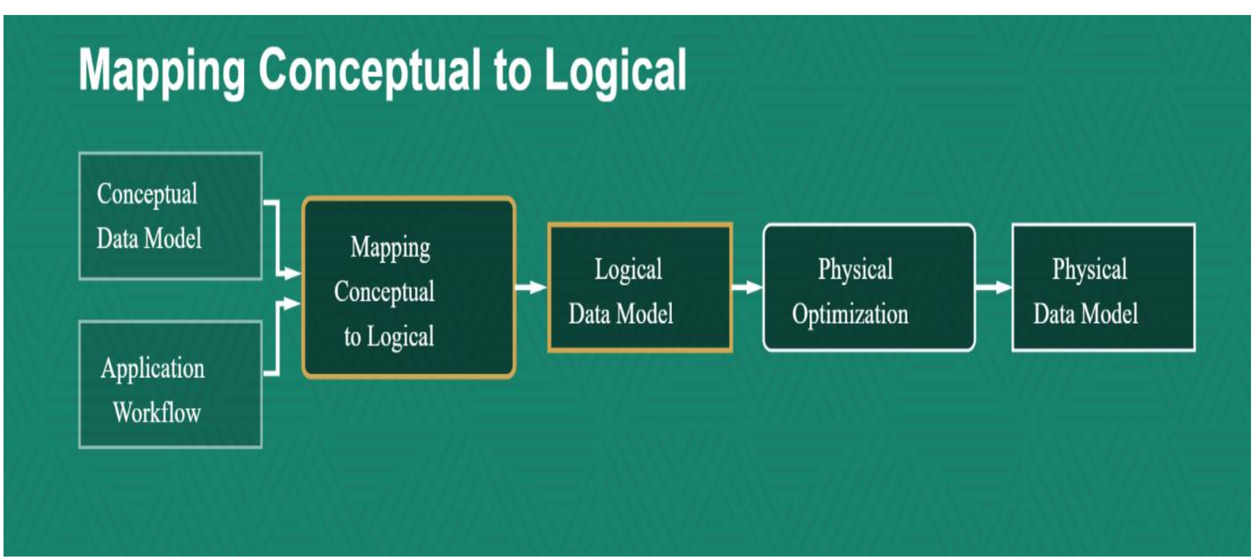
Klíčová omezení ovlivňují návrh schématu. Klíče entit a vztahů ovlivňují primární klíče tabulky. Primární klíč jednoznačně identifikuje řádek / entitu / vztah. Omezení mohutnosti ovlivňují klíč pro vztahy. Vztah 1:1 může použít klíč od kterékoli entity. Více:více používá klíč od obou entit.

Znalost dotazů přímo ovlivňuje návrh schématu, protože dotazy jsou zachyceny modelem pracovního postupu aplikace (application workflow). Při změně dotazů se návrh schématu tabulky změní. Návrh schématu pak organizuje data pro efektivní spouštění dotazů.

Vnoření dat je hlavní technikou modelování dat: organizuje více entit do jednoho oddílu a podporuje oddíl na přístup k datům dotazu.

Duplikace dat je posledním z principu modelování dat a je důležitým z toho důvodu, že vždy je lepší duplikovat data, než je spojovat, přičemž duplikovat lze napříč tabulkami, oddíly a / nebo řádky, což je vyžadováno k efektivní podpoře různých dotazů na stejná data. Ve světě Cassandra je kompromis mezi prostorovou účinností a časovou účinností téměř vždy ve prospěch druhé. Normalizace zde není prioritou. Výsledky dotazu jsou předem vypočítány a zhmotněny.

Pro vytvoření konečného datového modelu je potřeba se řídit následujícím schématem z obr.17.



Obrázek 17: Základní fáze vytvoření datového modelu, zdroj: [12]

Z obr.17 je také vidět, že prvním krokem je vytvoření konceptuálního modelu a také tzv. „application workflow“, tj. pracovní postup aplikace a dotazy. Následovat pak bude převedení tohoto modelu do logického datového modelu a po fyzické optimalizaci následuje konečný fyzický datový model.

4.3 Realizace databáze dopravních dat

Jako první krok zpracování dat bylo potřeba všechny soubory obsahující data z jednoho profilu a jednoho směru sloučit, tj. nově v jednom souboru se nacházely data jenom z konkrétního detektoru v konkrétním směru.

Následně byla data převedena do formátu .xml. Bylo rozhodnuto pro tento krok zpracování vytvořit speciální skript v jazyce Python. Tento skript je uveden na konci práce v přílohách jako Skript 1.

4.3.1 Základní nastavení v prostředí Cassandra pro DB dopravních dat

Jako první krok byl přejmenován cluster pro budoucí databází dopravních dat jako ‚FD Cluster‘. (obr.18)

```
Connected to FD Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 4.0 | CQL spec 3.4.5 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Obrázek 18: Přejmenování clusteru, zdroj: autor

Následně bylo potřeba zvolit vhodný replikační faktor a úroveň konzistence. Vzhledem k tomu, že v této práci byl zvolen počet uzlů = 3, podle odstavce 4.1.1 byl zvolen replikační faktor = 3, což znamená, že v systému by měly existovat tři kopie každého řádku, přičemž každá kopie by se měla nacházet na jiném uzlu, které jsou ale v rámci práce simulovány skutečně na jednom serveru.

Co se tyká úrovně konzistence, podle odstavce 4.1.2 byla zvolena úroveň LOCAL_QUORUM, což pro tento konkrétní případ bude znamenat úroveň = (3/2) +1 = 2. Tato úroveň dovolí, aby cluster mohl tolerovat výpadek jednoho uzlu v datovém centru, tj. pokud by jedem z uzlů spadl, informace by byla zachována.

4.3.2 Tvorba konceptuálního modelu

S ohledem na existující soubory s daty pro konceptuální model bylo potřeba nejprve vydefinovat entity, jejich atributy a vztahy.

Pro potřeby dané databáze dopravních dat byly definovány následující entity a jejich atributy:

- Entita AREA s atributy: *bucket, number, zone, lat, lon, n_detectors.*

Tato entita představuje určité místo, ve kterém je umístěna skupina detektorů (nejčastěji v obou směrech).

Bucket v dané databáze značí břeh Vltavy, na kterém se detektor nachází (1 je pravý, 2 je levý). V rámci práce všechny detektory, ze kterých byla data poskytnuta byly umístěny na levém břehu a mají tak bucket 2. Nicméně tato entita je tvořena za předpokladu, že se databáze bude rozšiřovat.

Obecně bucketing je jednou z nejdůležitějších technik při práci s daty časových řad v Cassandra. Pomocí tohoto přístupu lze podporovat skenování rozsahu, přičemž databáze bude chráněna před neočekávanými provozními nárazy. Je to strategie, která umožňuje řídit, kolik dat je uloženo v každém oddílu, a také distribuovat záznamy po celém clusteru. Za předpokladu že se objem dat v takové databáze bude postupně zvětšovat bylo rozhodnuto tuto strategie použit již na dané etapě modelování.

Number je unikátní číslo místa, kde se nachází skupina detektorů.

Zone je textový popis místa, kde se nachází detektor. V rámci práce je vždy uvedeno tzv. „číslo Prahy“, např. Praha_3.

Lat a *lon* jsou přesné souřadnice místa, kde se skupina detektorů nachází.

N_detectors značí počet detektorů v konkrétní skupině (v případě dané databáze muže být od 1 do 4).

- Entita DETECTOR s atributy: *id, type, direction.*

Tato entita poskytuje všechny informace o konkrétním detektoru ve skupině.

ID je unikátní číslo detektoru.

Type je typ konkrétního detektoru (v rámci práce byl u všech detektorů stejný).

Direction je směr, ve kterém je daný detektor umístěn, a to buď jih (J), sever (S), západ (Z) anebo výhod (V).

- Entita DATA s atributy: *parameter, value.*

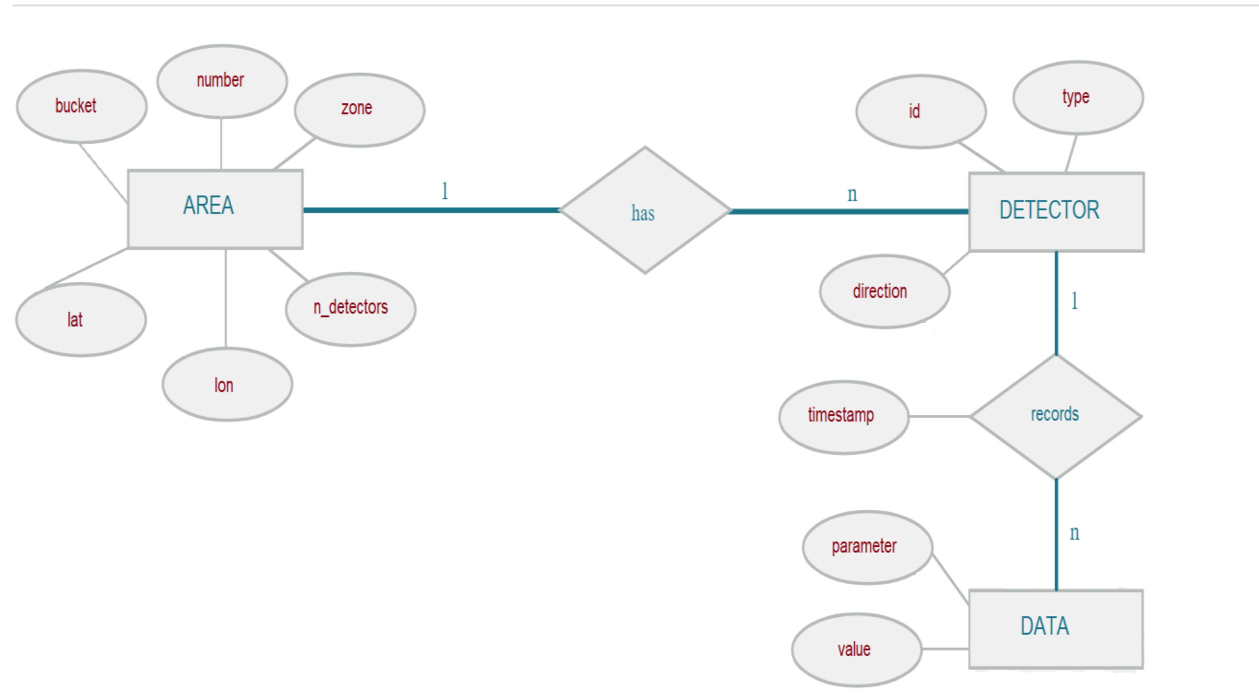
Tato entita již poskytuje data z konkrétního detektoru.

Parameter je v rámci práce buď intenzita anebo rychlost vozidla,

Value je skutečná hodnota, odpovídající intenzitě nebo rychlosti vozidla.

Vztahy mezi všemi třemi entitami je vždy typu 1:více, tj. 1 area obsahuje více detektorů, 1 detektor pak obsahuje více dat.

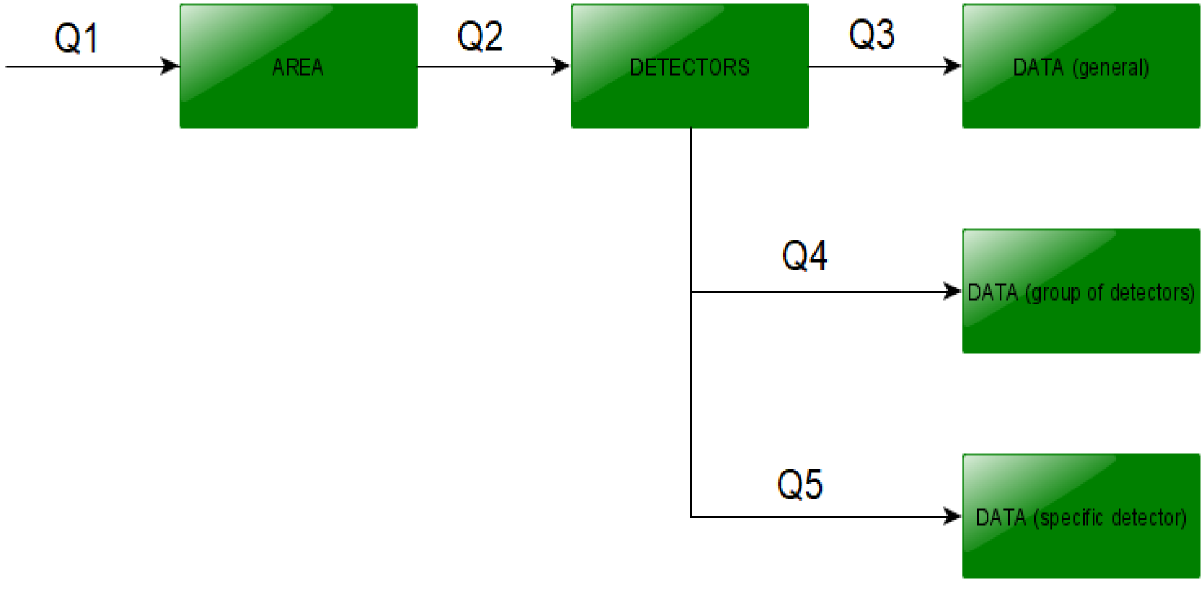
Podle těchto předpokladů byl vytvořen následující konceptuální model (obr.19)



Obrázek 19: Konceptuální model, zdroj: autor

Následně pro tvorbu logického modelu bylo potřeba vytvořit application workflow, tj. postup aplikace a také dotazy, které by se obracely na data a představovaly „řez" touto databází. Application workflow vychází z požadavků konkrétní aplikace na data, se kterými tato aplikace bude pracovat, a podle toho je pak tvořena celá databázová struktura.

Zvolený workflow je vidět na obr.20. Obsahuje dohromady 5 dotazů: dotaz na konkrétní areu, dotaz na vybrané detektory v arei a dotazy na data, přičemž jeden z nich je na data ze všech detektorů, druhý se specifikuje na skupinu detektorů a poslední se ptá na jeden konkrétní detektor.



Obrázek 20: Application workflow, zdroj: autor

Podle tohoto workflow byly vytvořeny následující dotazy:

- Najít všechny areí buď s pravého anebo levého břehu Vltavy.

Tento dotaz by měl otevřít všechny skupiny detektorů, které jsou umístěny na zadaném břehu.

Dotaz v systému by pak vypadal následujícím způsobem:

```
SELECT * FROM areas WHERE bucket = ?;
```

- Najít všechny detektory z konkrétní arei.

Tento dotaz ukáže informaci o všech detektorech, které se nachází v zadané skupině.

Dotaz v systému by pak vypadal následujícím způsobem:

```
SELECT * FROM detectors_by_area WHERE area_number = ?;
```

- Najít všechna data ze všech detektorů podle vybraného času.

Dotaz v systému by pak vypadal následujícím způsobem:

```
SELECT * FROM data_by_area WHERE date_hour >= ? AND date_hour <= ?;
```

- Najít všechna data z konkrétní skupiny detektorů podle vybraného času.

Dotaz v systému by pak vypadal následujícím způsobem:

```
SELECT * FROM data_by_area WHERE area_number = ? AND date_hour >= ? AND
date_hour <= ?;
```

- Najit všechny data z konkrétního detektoru podle vybraného času.

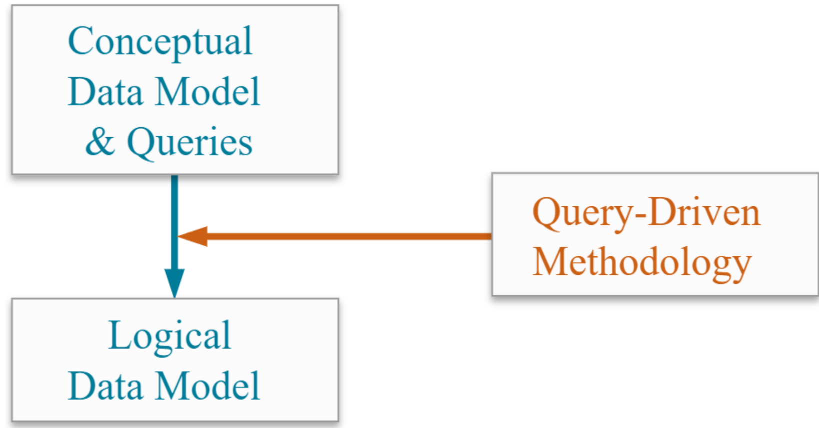
Dotaz v systému by pak vypadal následujícím způsobem:

```
SELECT * FROM data_by_area WHERE area_number = ? AND date_hour >= ? AND
date_hour <= ? AND id_detector = ?;
```

Vzhledem k tomu, že konceptuální model a application workflow jsou vytvořeny, lze z toho již tvořit logický model.

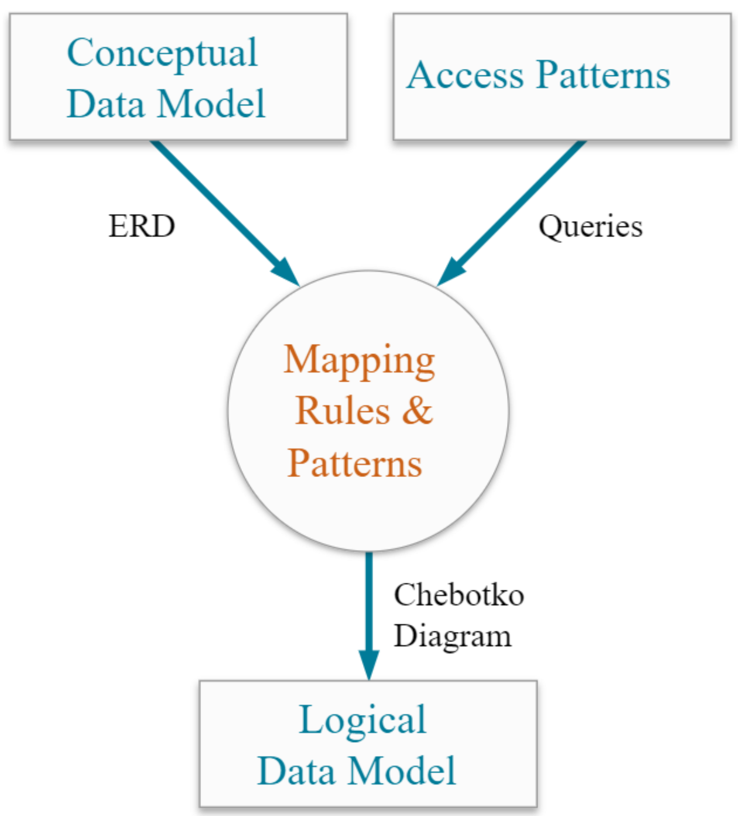
4.3.3 Tvorba logického modelu

Proces návrhu logického modelu používá přístup shora dolů a lze ho definovat algoritmicky. (obr.21)



Obrázek 21: Schéma tvorby logického modelu, zdroj: [12]

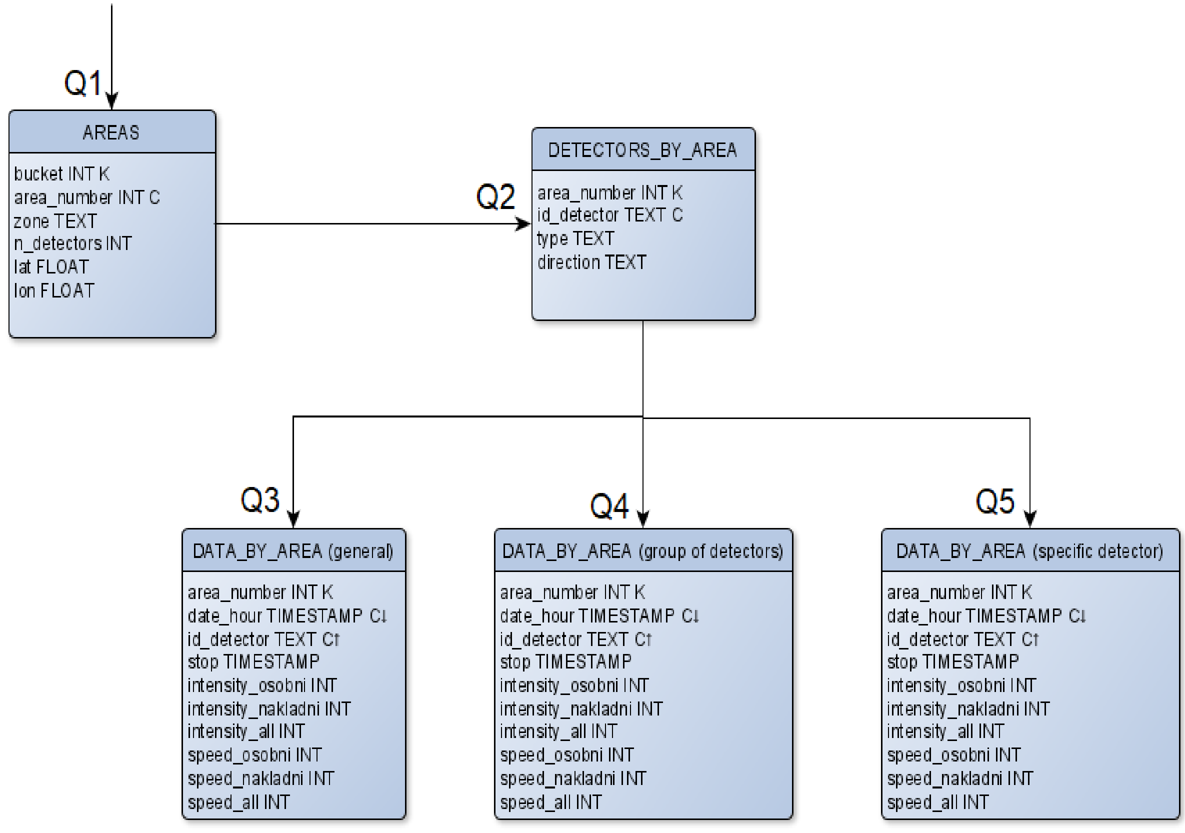
Z obr.22 pak také vyplývá celý proces tvorby logického modelu.



Obrázek 22: Postup tvorby logického modelu, zdroj: [12]

Chebotko Diagram je vizuálním diagramem pro tabulky Cassandra a přístupové vzory. Je to grafické znázornění návrhu databázového schématu Cassandra, který dokumentuje logický a fyzický datový model.

Pro danou databází byl vytvořen následující Chebotko diagram (obr.23).



Obrázek 23: Chebotko diagram, zdroj: autor

V Chebotko diagramu jsou uvedeny všechny datové formáty a také označení K (primární klíč) a C (clustering column). Po písmenu C lze následně uvést buď šípku nahoru (seřazeno vzestupně) nebo šípku dolu (seřazeno sestupně). Tabulky pro dotazy Q3, Q4 a Q5 jsou úplně identické, protože do databáze bude zadána ve skutečnosti jenom jedna tabulka z důvodu praktičnosti.

Dalším krokem je tvorba fyzického modelu.

4.3.4 Tvorba fyzického modelu

Pro tvorbu fyzického modelu je potřeba již přejít do prostředí Cassandra a pomocí jazyku CQL vytvořit nejprve tzv. KEYSPACE, tj. klíčový prostor. Je to místo, kam následně budou uloženy všechny jednotlivé tabulky, a poté i data do každé z tabulek. Pro tvorbu je potřeba zadat příkaz *cqlsh*, který převádí uživatele do prostředí příkazového řádku pro interakci s Cassandra prostřednictvím CQL.

Tvorba takového KEYSPACE lze vidět na obr.24. KEYSPACE byl nazván data_from_detectors, v tomto kroku bylo potřeba uvést také class a replication faktor.

```
cqlsh> CREATE KEYSPACE data_from_detectors
   ... WITH replication = {'class':'SimpleStrategy',
   ... "replication_factor':'3'};
```

Obrázek 24: Tvorba keyspace data_from_detectors, zdroj: autor

Následně bylo potřeba do KEYSPACE vložit všechny tři tabulky s atributy s následujícími názvy:

- Area (primární klíč se skládá z *bucket* jako partition key a *area_number* jako clustering column)
- Detectors_by_area (DETECTORS) (primární klíč se skládá z *area_number* jako partition key a *id_detector* jako clustering column)
- Data_by_area (DATA) (primární klíč se skládá z *area_number* jako partition key a *date_hour* a *id_detector* jako clustering columns)

Jak již bylo zmíněno v odstavci 4.2.2 partition key v primárním klíči odpovídá za definování oddílů, clustering columns následně pomáhají s definováním pořadí řazení a zároveň se ujišťují, že primární klíč je skutečně unikátní.

Vložení lze vidět na obr.25, 26 a 27.

```
cqlsh> CREATE TABLE data_from_detectors.areas
   ... (bucket INT,
   ... area_number INT,
   ... zone TEXT,
   ... n_detectors INT,
   ... lat FLOAT,
   ... lon FLOAT,
   ... PRIMARY KEY ((bucket), area_number));
cqlsh>
```

Obrázek 25: Tvorba tabulky AREA, zdroj: autor

```
cqlsh:data_from_detectors> CREATE TABLE detectors_by_area
                       ... (area_number UUID,
                       ... id_detector TEXT,
                       ... type TEXT,
                       ... direction TEXT,
                       ... PRIMARY KEY ((area_number), id_detector));
cqlsh:data_from_detectors>
```

Obrázek 26: Tvorba tabulky DETECTORS, zdroj: autor

```
cqlsh> USE data_from_detectors;
cqlsh:data_from_detectors> CREATE TABLE data_by_area
                       ... (area_number INT,
                       ... date_hour TIMESTAMP,
                       ... id_detector TEXT,
                       ... intensity_all INT,
                       ... intensity_nakladni INT,
                       ... intensity_osobni INT,
                       ... speed_all INT,
                       ... speed_nakladni INT,
                       ... speed_osobni INT,
                       ... stop TIMESTAMP,
                       ... PRIMARY_KEY ((area_number),
                       ... date_hour, id_detector))
                       ... WITH CLUSTERING ORDER BY
                       ... (date_hour DESC, id_detector ASC);
```

Obrázek 27: Tvorba tabulky DATA, zdroj: autor

Z obrázků je vidět, že se k datům z KEYSPACE lze přistoupit dvěma způsoby: buď pomocí spojení konkrétní tabulky a KEYSPACE pomocí tečky (obr.25) anebo je potřeba nejdřív „změnit" KEYSPACE pomocí příkazu USE, a až pak zadat název tabulky (obr.27)

Dalším krokem je načtení dat do jednotlivých tabulek.

Pro tabulku AREA data byla tvořena ručně v závislosti na několika faktorech. Prvním z nich byla geografická poloha každé oblasti (neboli arei): odsud se vyvíjelo číslo bucketu, číslo samotné arei a také číslo Prahy, ve které je daný detektor umístěn. Dalším faktorem byl počet detektorů umístěných v každé arei. A jako poslední byla data doplněna o GPS souřadnice, které také byly ručně vyhledány pro každou areu zvlášť. Poté byla všechna tato data vložena pomocí jednotlivých insertů a způsob vložení je vidět na obr.28.

```
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES ( 2, 50106,'Praha_13', 2, 50.04891416966162, 14.286291498691615 );
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,50150, 'Praha_5',  2, 50.06807551124339, 14.34458386204517);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,50157, 'Praha_5',  2, 50.070791434094076, 14.332382092725249);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,50663, 'Praha_13',  2, 50.04061647642163, 14.331928534532352);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,50699, 'Praha_13',  2, 50.063694427813296, 14.336187046903934);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,51031, 'Praha_13',  1, 50.059538361425204, 14.336518823982821);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,51211, 'Praha_5',  2, 50.04062300909111, 14.401086577790245);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,52030, 'Praha_5',  2, 50.06655066558754, 14.411054792606691);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,52033, 'Praha_5',  2, 50.061634411938236, 14.410816846817813);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,60069, 'Praha_6',  2, 50.0896196958065, 14.372518902388231);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,60671, 'Praha_6',  2, 50.09819217758463, 14.310637734336465);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,61495, 'Praha_17',  2, 50.07404677631216, 14.311050615604039);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,61710, 'Praha_6',  2, 50.103151181792526, 14.384536325655336);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,61761, 'Praha_6',  2, 50.10184416949858, 14.356555193848457);
cqlsh> INSERT INTO data_from_detectors.areas (bucket,area_number,zone,n_detectors,lat,lon) VALUES (2,70138, 'Praha_7',  2, 50.108471452316984, 14.44396659409058);
cqlsh>
```

Obrázek 28: Naplnění tabulky AREAS, zdroj: autor

Následně je vždy potřeba ověřit, zda se všechna data skutečně uložila do tabulky v Cassandra. Na to byl použit příkaz SELECT *, který má ukázat všechna data uložena do dané tabulky. Výsledek je vidět na obr.29.

```
cqlsh> SELECT * FROM data_from_detectors.areas;

 bucket | area_number | lat      | lon      | n_detectors | zone
--------+-------------+----------+----------+-------------+----------
      2 |       50106 | 50.04892 | 14.28629 |           2 | Praha_13
      2 |       50150 | 50.06808 | 14.34458 |           2 |  Praha_5
      2 |       50157 | 50.07079 | 14.33238 |           2 |  Praha_5
      2 |       50663 | 50.04062 | 14.33193 |           2 | Praha_13
      2 |       50699 | 50.06369 | 14.33619 |           2 | Praha_13
      2 |       51031 | 50.05954 | 14.33652 |           1 | Praha_13
      2 |       51211 | 50.04062 | 14.40109 |           2 |  Praha_5
      2 |       52030 | 50.06655 | 14.41105 |           2 |  Praha_5
      2 |       52033 | 50.06163 | 14.41082 |           2 |  Praha_5
      2 |       60069 | 50.08962 | 14.37252 |           2 |  Praha_6
      2 |       60671 | 50.09819 | 14.31064 |           2 |  Praha_6
      2 |       61495 | 50.07405 | 14.31105 |           2 | Praha_17
      2 |       61710 | 50.10315 | 14.38454 |           2 |  Praha_6
      2 |       61761 | 50.10184 | 14.35655 |           2 |  Praha_6
      2 |       70138 | 50.10847 | 14.44397 |           2 |  Praha_7

(15 rows)
cqlsh>
```

Obrázek 29: Ověření načtení dat do tabulky AREAS, zdroj: autor

A posledním krokem při tvorbě fyzického modelu je dotazování na tabulku. Dotaz vypadal následovně:

```
SELECT * FROM areas WHERE bucket = ?;
```

Výsledkem tohoto dotazu by měla být tabulka areí, které se nachází na levém břehu Vltavy.

Výsledek dotazu je na obr.30.

```
cqlsh> SELECT * FROM data_from_detectors.areas WHERE bucket = 2;

 bucket | area_number | lat      | lon      | n_detectors | zone
--------+-------------+----------+----------+-------------+----------
      2 |       50106 | 50.04892 | 14.28629 |           2 | Praha_13
      2 |       50150 | 50.06808 | 14.34458 |           2 |  Praha_5
      2 |       50157 | 50.07079 | 14.33238 |           2 |  Praha_5
      2 |       50663 | 50.04062 | 14.33193 |           2 | Praha_13
      2 |       50699 | 50.06369 | 14.33619 |           2 | Praha_13
      2 |       51031 | 50.05954 | 14.33652 |           1 | Praha_13
      2 |       51211 | 50.04062 | 14.40109 |           2 |  Praha_5
      2 |       52030 | 50.06655 | 14.41105 |           2 |  Praha_5
      2 |       52033 | 50.06163 | 14.41082 |           2 |  Praha_5
      2 |       60069 | 50.08962 | 14.37252 |           2 |  Praha_6
      2 |       60671 | 50.09819 | 14.31064 |           2 |  Praha_6
      2 |       61495 | 50.07405 | 14.31105 |           2 | Praha_17
      2 |       61710 | 50.10315 | 14.38454 |           2 |  Praha_6
      2 |       61761 | 50.10184 | 14.35655 |           2 |  Praha_6
      2 |       70138 | 50.10847 | 14.44397 |           2 |  Praha_7

(15 rows)
cqlsh>
```

Obrázek 30: Výsledek dotazu Q1, zdroj: autor

Z obr.30 lze vidět, že dotaz proběhl správně, protože počet vybraných areí je stejný jako jejich celkový počet. Vyplívá to z toho faktu, že v datech existují senzory jenom z levého břehu Vltavy, což odpovídá bucketu č. 2.

S tabulkami DETECTORS a DATA postup vložení dat byl odlišný. Do nich se data měla vložit ze zpracovaných souborů, které byly vygenerovány pomocí předem vytvořených skript v jazyce Python. Tyto dva skripty filtrují potřebná data a převádí je do formátu .csv, který je vhodný pro následné použití v prostředí Cassandra. Tyto skripty jsou taktéž uvedeny na konci práce v přílohách jako Skript 2 (pro tabulku DETECTORS) a Skript 3 (pro tabulku DATA).

Vzhledem k tomu, že oba soubory (jak pro tabulku DETECTORS, tak pro tabulku DATA) byly moc velké, při jejich zpracování v Pythonu byla pokaždé vygenerována tabulka ve formátu .csv, která ale obsahovala jenom polovinu ze všech záznamů. Proto bylo potřeba oba soubory ve formátu .xml (jak pro tabulku DETECTORS, tak pro tabulku DATA) rozdělit a nahrávat do Pythonu zvlášť nadvakrát, a následně vkládat i do Cassandra dva vygenerované soubory .csv.

Z obr.31 je vidět, jak první část dat byla vložena do tabulky DETECTORS.

```
cqlsh> COPY data_from_detectors.detectors_by_area FROM 'output_detectors_by_area1.csv' WITH DELIMITER=';' AND HEADER=TRUE;
Using 3 child processes

Starting copy of data_from_detectors.detectors_by_area with columns [area_number, id_detector, direction, type].
Processed: 393543 rows; Rate:    6191 rows/s; Avg. rate:   19474 rows/s
393543 rows imported from 1 files in 20.209 seconds (0 skipped).
```

Obrázek 31: Naplnění tabulky DETECTORS, zdroj: autor

Stejným způsobem pak byla vložená i druhá část dat. Následně bylo potřeba se znovu ujistit, že se data do Cassandra uložila správně. Výsledek je vidět na obr.32.

```
cqlsh> select * from data_from_detectors.detectors_by_area;

 area_number | id_detector | direction | type
-------------+-------------+-----------+----------
       52030 |  R520302-S1 |         S | CollectR
       52030 |  R520302-S2 |         S | CollectR
       52030 |  R520303-J1 |         J | CollectR
       52030 |  R520303-J2 |         J | CollectR
       52033 |  R520338-S1 |         S | CollectR
       52033 |  R520338-S2 |         S | CollectR
       52033 |  R520339-J1 |         J | CollectR
       52033 |  R520339-J2 |         J | CollectR
       50150 |  R501502-Z1 |         Z | CollectR
       50150 |  R501502-Z2 |         Z | CollectR
       50150 |  R501503-V1 |         V | CollectR
       50150 |  R501503-V2 |         V | CollectR
       50157 |  R501575-V1 |         V | CollectR
       50157 |  R501575-V2 |         V | CollectR
       50157 |  R501576-Z1 |         Z | CollectR
       50157 |  R501576-Z2 |         Z | CollectR
       50699 |  R506998-J1 |         J | CollectR
       50699 |  R506998-J2 |         J | CollectR
       50699 |  R507001-S1 |         S | CollectR
       50699 |  R507001-S2 |         S | CollectR
       51211 |  R512118-S1 |         S | CollectR
       51211 |  R512118-S2 |         S | CollectR
       51211 |  R512142-J1 |         J | CollectR
       51211 |  R512142-J2 |         J | CollectR
       50663 |  R506639-Z1 |         Z | CollectR
       50663 |  R506639-Z2 |         Z | CollectR
       50663 |  R506640-V1 |         V | CollectR
       60069 |  R600694-Z2 |         Z | CollectR
       50699 |  R506998-J1 |         J | CollectR
       50699 |  R506998-J2 |         J | CollectR
       50699 |  R507001-S1 |         S | CollectR
       50699 |  R507001-S2 |         S | CollectR
       51211 |  R512118-S1 |         S | CollectR
       51211 |  R512118-S2 |         S | CollectR
       51211 |  R512142-J1 |         J | CollectR
       51211 |  R512142-J2 |         J | CollectR
       50663 |  R506639-Z1 |         Z | CollectR
       50663 |  R506639-Z2 |         Z | CollectR
       50663 |  R506640-V1 |         V | CollectR
       50663 |  R506640-V2 |         V | CollectR
       50106 |  R501061-Z1 |         Z | CollectR
       50106 |  R501061-Z2 |         Z | CollectR
       50106 |  R501062-V1 |         V | CollectR
       50106 |  R501062-V2 |         V | CollectR
       60671 |  R606713-V1 |         V | CollectR
       60671 |  R606713-V2 |         V | CollectR
       60671 |  R606716-Z1 |         Z | CollectR
       60671 |  R606716-Z2 |         Z | CollectR
       51031 |  R510311-V1 |       Z+V | CollectR
       51031 |  R510311-V2 |       Z+V | CollectR
       51031 |  R510311-Z1 |       Z+V | CollectR
       51031 |  R510311-Z2 |       Z+V | CollectR
       61710 |  R617105-V1 |         V | CollectR
       61710 |  R617105-V2 |         V | CollectR
       61710 |  R617106-Z1 |         Z | CollectR
       61710 |  R617106-Z2 |         Z | CollectR

(60 rows)
```

Obrázek 32: Ověření uložení dat do tabulky DETECTORS, zdroj: autor

Načetlo se dohromady 60 řádků, což odpovídá počtu detektorů. Z tohoto obrázku je také vidět, že se na areách může vyskytnout opravdu různý počet detektorů.

Následným krokem bylo dotazování na tabulku. V tomto případě dotaz byl následující:

```
SELECT * FROM detectors_by_area WHERE area_number = ?;
```

Výsledkem tohoto dotazu by měla být tabulka detektorů, které se nachází ve vybrané arei.

Výsledek dotazu je vidět na obr.33.

```
cqlsh> select * from data_from_detectors.detectors_by_area WHERE area_number = 51211;

 area_number | id_detector | direction | type
-------------+-------------+-----------+---------
       51211 |  R512118-S1 |         S | CollectR
       51211 |  R512118-S2 |         S | CollectR
       51211 |  R512142-J1 |         J | CollectR
       51211 |  R512142-J2 |         J | CollectR

(4 rows)
cqlsh>
```

Obrázek 33: Výsledek dotazu Q2, zdroj: autor

Z tohoto obrázku lze vidět, že tento dotaz také proběhl správně: ukázaly se všechny detektory, patřící zadané arei.

Dalším krokem bylo naplnění tabulky DATA. Postup byl zcela stejný jako v případě předchozí tabulky. Výsledek je na obr.34.

```
cqlsh:data_from_detectors> COPY data_by_area FROM 'output_detectors-data1.csv' WITH DELIMITER=';' AND HEADER=TRUE;
Using 3 child processes

Starting copy of data_from_detectors.data_by_area with columns [area_number, date_hour, id_detector, intensity_all, intensity_nakladni, intensity_
osobni, speed_all, speed_nakladni, speed_osobni, stop].
Processed: 393543 rows; Rate:    6216 rows/s; Avg. rate:    9507 rows/s
393543 rows imported from 1 files in 41.394 seconds (0 skipped).
```

Obrázek 34: Naplnění tabulky DATA, zdroj: autor

A následně bylo zase provedeno ověření, že se data skutečně uložila do databáze. Těchto dat tady bylo víc, proto je na obr.35 ukázána jenom jejich část.

```
cqlsh> SELECT * FROM data_from_detectors.data_by_area;
```

area_number	date_hour	id_detector	intensity_all	intensity_nakladni	intensity_osobni	speed_all	speed_nakladni	speed_osobni	stop
52030	2018-11-19 00:25:27.000000+0000	R520302-S1	6	0	6	66	0	66	2018-11-19 00:25:27.000000+0000
52030	2018-11-19 00:25:27.000000+0000	R520302-S2	3	0	3	73	0	73	2018-11-19 00:25:27.000000+0000
52030	2018-11-19 00:25:27.000000+0000	R520303-J1	10	0	10	42	0	42	2018-11-19 00:25:27.000000+0000
52030	2018-11-19 00:25:27.000000+0000	R520303-J2	7	0	7	59	0	59	2018-11-19 00:25:27.000000+0000
52030	2018-11-19 00:20:27.000000+0000	R520302-S1	6	0	6	63	0	63	2018-11-19 00:20:27.000000+0000
52030	2018-11-19 00:20:27.000000+0000	R520302-S2	3	0	3	71	0	71	2018-11-19 00:20:27.000000+0000
52030	2018-11-19 00:20:27.000000+0000	R520303-J1	15	1	14	42	36	43	2018-11-19 00:20:27.000000+0000
52030	2018-11-19 00:20:27.000000+0000	R520303-J2	13	1	12	61	65	61	2018-11-19 00:20:27.000000+0000
52030	2018-11-19 00:15:27.000000+0000	R520302-S1	8	0	8	66	0	66	2018-11-19 00:15:27.000000+0000
52030	2018-11-19 00:15:27.000000+0000	R520302-S2	5	0	5	67	0	67	2018-11-19 00:15:27.000000+0000
52030	2018-11-19 00:15:27.000000+0000	R520303-J1	10	0	10	46	0	46	2018-11-19 00:15:27.000000+0000
52030	2018-11-19 00:15:27.000000+0000	R520303-J2	9	0	9	63	0	63	2018-11-19 00:15:27.000000+0000
52030	2018-11-19 00:10:27.000000+0000	R520302-S1	10	0	10	64	0	64	2018-11-19 00:10:27.000000+0000
52030	2018-11-19 00:10:27.000000+0000	R520302-S2	4	0	4	72	0	72	2018-11-19 00:10:27.000000+0000

Obrázek 35: Ověření načtení dat do tabulky DATA, zdroj: autor

Následným krokem bylo dotazování na tabulku DATA a měly by se použit postupně tři dotazy podle zadaného času: dotaz přes všechna data, přes vybranou skupinu detektorů a na jeden konkrétní detektor.

Během provedení těchto dotazů byla objevena jedna ze základních vlastností Cassandra, která se týká ukládání dat a která byla zmíněna v teoretické části práce v odstavci 3.3.

První dotaz na tabulku vypadal následovně:

```
SELECT * FROM data_by_area WHERE date_hour >= ? AND date_hour <= ?;
```

Tento dotaz by měl ukázat všechna data, která se načetla během zadaného okamžiku. Nicméně, Cassandra na takový dotaz odpovídá chybou, která je vidět na obr. 36.

```
cqlsh> SELECT * FROM data_from_detectors.data_by_area WHERE  date_hour >= '2018-10-09 02
:27:06+02:00' AND date_hour <= '2018-10-09 03:44:06+02:00' ;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute thi
s 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 FI
LTERING"
```

Obrázek 36: Chyba dotazu Q3 nad všemi daty v tabulce DATA, zdroj: autor

Přestože dotaz se v tuto chvíli neuskutečnil, Cassandra nabízí speciální funkcí ALLOW FILTERING. Po její využití dotaz se stává funkčním, jak je ukázáno na obr.37.

```
cqlsh> SELECT * FROM data_from_detectors.data_by_area WHERE  date_hour >= '2018-10-09 02:27:06+02:00' AND date_hour <= '2018-10-09 03:44:06+02:00'
ALLOW FILTERING ;

 area_number | date_hour                       | id_detector | intensity_all | intensity_nakladni | intensity_osobni | speed_all | speed_nakladni
| speed_osobni | stop
-------------+---------------------------------+-------------+---------------+--------------------+------------------+-----------+---------------
+--------------+---------------------------------
       52030 | 2018-10-09 01:40:27.000000+0000 |  R520302-S1 |             7 |                  0 |                7 |        60 |              0
|         60 | 2018-10-09 01:40:27.000000+0000
       52030 | 2018-10-09 01:40:27.000000+0000 |  R520302-S2 |             2 |                  0 |                2 |        84 |              0
|         84 | 2018-10-09 01:40:27.000000+0000
       52030 | 2018-10-09 01:40:27.000000+0000 |  R520303-J1 |            12 |                  0 |               12 |        43 |              0
|         43 | 2018-10-09 01:40:27.000000+0000
       52030 | 2018-10-09 01:40:27.000000+0000 |  R520303-J2 |             6 |                  0 |                6 |        71 |              0
|         71 | 2018-10-09 01:40:27.000000+0000
       52030 | 2018-10-09 01:35:27.000000+0000 |  R520302-S1 |            10 |                  0 |               10 |        59 |              0
|         59 | 2018-10-09 01:35:27.000000+0000
       52030 | 2018-10-09 01:35:27.000000+0000 |  R520302-S2 |             2 |                  0 |                2 |        63 |              0
|         63 | 2018-10-09 01:35:27.000000+0000
       52030 | 2018-10-09 01:35:27.000000+0000 |  R520303-J1 |             6 |                  0 |                6 |        43 |              0
|         43 | 2018-10-09 01:35:27.000000+0000
       52030 | 2018-10-09 01:35:27.000000+0000 |  R520303-J2 |             0 |                  0 |                0 |         0 |              0
|          0 | 2018-10-09 01:35:27.000000+0000
       52030 | 2018-10-09 01:30:27.000000+0000 |  R520302-S1 |             8 |                  0 |                8 |        60 |              0
|         60 | 2018-10-09 01:30:27.000000+0000
       52030 | 2018-10-09 01:30:27.000000+0000 |  R520302-S2 |             2 |                  0 |                2 |        56 |              0
|         56 | 2018-10-09 01:30:27.000000+0000
```

Obrázek 37: Funkční dotaz Q3 s použitím ALLOW FILTERING, zdroj: autor

Přestože dotaz funguje bylo potřeba se zamyslet nad tím, co funkce ALLOW FILTERING doopravdy děla a proč při některých dotazech vzniká chybová hláška.

Ve skutečnosti, Cassandra nabízí tuto funkcí, protože ví, že nemůže efektivně provést dotaz. Proto následně vzniká chybová hláška, která varuje: „Buďte opatrní. Provedení tohoto dotazu nemusí být dobrý nápad, protože může využívat spoustu výpočetních zdrojů.“

Jediným způsobem, jak může Cassandra provést tento dotaz je načtení všech řádků z tabulky a následné odfiltrování těch, které nemají požadovanou hodnotu pro sloupec date_hour.

Pokud by tabulka obsahovala 1 milion řádků a 95 % z nich by mělo požadovanou hodnotu pro sloupec date_hour, bude dotaz stále relativně efektivní a může se použit funkce ALLOW FILTERING.

Jiná situace se nastane, pokud tabulka bude obsahovat 1 milion řádků a pouze 2 řádky budou obsahovat požadovanou hodnotu pro sloupec date_hour. Takový dotaz se stává extrémně neefektivním: Cassandra načte 999998 řádků zbytečně. Pokud by se takový dotaz používal často, lepší volbou by bylo přidat index do sloupce date_hour.

Cassandra nemá žádný způsob, jak rozlišovat mezi dvěma výše uvedenými případy, protože všechno závisí na distribuci dat v tabulce. Cassandra proto varuje a spoléhá na uživatele, že udělá správnou volbu.

V tomto konkrétním případě bude dotaz efektivní, protože počet dat odpovídajících vybranému časovému okamžiku je skutečně dostatečně velký, takže použití funkce ALLOW FILTERING je zcela možné.

Druhý dotaz nad tabulkou DATA se prováděl již nad skupinou detektorů. Vypadal následovně:

```
SELECT * FROM data_by_area WHERE area_number = ? AND date_hour >= ? AND
date_hour <= ?;
```

Po provedení dotazu by se měla ukázat data, která se načetla během zadaného okamžiku ale již z konkrétní skupiny detektorů.

Takový dotaz již splňuje všechny požadavky Cassandra, a proto proběhl v pořádku bez použití funkcí ALLOW FILTERING. (obr.38)

```
cqlsh> SELECT * FROM data_from_detectors.data_by_area WHERE area_number = 60069 AND date_hour >= '2018-10-09 02:27:06+02:00' AND date_hour <= '2018-10-09 03:44:06+02:00';

 area_number | date_hour                       | id_detector | intensity_all | intensity_nakladni | intensity_osobni | speed_all | speed_nakladni | speed_osobni | stop
-------------+---------------------------------+-------------+---------------+--------------------+------------------+-----------+----------------+--------------+----------
---------------------
       60069 | 2018-10-09 01:42:06.000000+0000 |  R600693-V1 |             2 |                  0 |                2 |        68 |              0 |           68 | 2018-10-09
 01:42:06.000000+0000
       60069 | 2018-10-09 01:42:06.000000+0000 |  R600693-V2 |             5 |                  0 |                5 |        53 |              0 |           53 | 2018-10-09
 01:42:06.000000+0000
       60069 | 2018-10-09 01:42:06.000000+0000 |  R600694-Z1 |            11 |                  0 |               11 |        45 |              0 |           45 | 2018-10-09
 01:42:06.000000+0000
       60069 | 2018-10-09 01:42:06.000000+0000 |  R600694-Z2 |             6 |                  0 |                6 |        56 |              0 |           56 | 2018-10-09
 01:42:06.000000+0000
       60069 | 2018-10-09 01:37:06.000000+0000 |  R600693-V1 |             6 |                  1 |                5 |        54 |             44 |           57 | 2018-10-09
 01:37:06.000000+0000
       60069 | 2018-10-09 01:37:06.000000+0000 |  R600693-V2 |             8 |                  0 |                8 |        53 |              0 |           53 | 2018-10-09
 01:37:06.000000+0000
       60069 | 2018-10-09 01:37:06.000000+0000 |  R600694-Z1 |            19 |                  0 |               19 |        48 |              0 |           48 | 2018-10-09
 01:37:06.000000+0000
       60069 | 2018-10-09 01:37:06.000000+0000 |  R600694-Z2 |            14 |                  0 |               14 |        55 |              0 |           55 | 2018-10-09
 01:37:06.000000+0000
       60069 | 2018-10-09 01:32:06.000000+0000 |  R600693-V1 |             9 |                  0 |                9 |        56 |              0 |           56 | 2018-10-09
 01:32:06.000000+0000
       60069 | 2018-10-09 01:32:06.000000+0000 |  R600693-V2 |             4 |                  0 |                4 |        62 |              0 |           62 | 2018-10-09
 01:32:06.000000+0000
       60069 | 2018-10-09 01:32:06.000000+0000 |  R600694-Z1 |            16 |                  1 |               15 |        44 |             51 |           44 | 2018-10-09
 01:32:06.000000+0000
       60069 | 2018-10-09 01:32:06.000000+0000 |  R600694-Z2 |            10 |                  0 |               10 |        51 |              0 |           51 | 2018-10-09
 01:32:06.000000+0000

(12 rows)
cqlsh>
```

Obrázek 38: Funkční dotaz Q4, zdroj: autor

S posledním dotazem, který by měl ukázat data ve vybraném časovém okamžiku na konkrétním detektoru znovu vznikla komplikace, podobná komplikaci s dotazem Q3.

Dotaz vypadal následovně:

SELECT * FROM data_by_area WHERE area_number = ? AND date_hour >= ? AND date_hour <= ? AND id_detector = ?;

Jeho výsledek je na obr.39.

```
cqlsh> SELECT * FROM data_from_detectors.data_by_area WHERE area_number = 60069 AND  da
te_hour >= '2018-10-09 02:27:06+02:00' AND date_hour <= '2018-10-09 03:44:06+02:00' AND
 id_detector = 'R600694-Z2' ;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Clustering column
 "id_detector" cannot be restricted (preceding column "date_hour" is restricted by a no
n-EQ relation)"
```

Obrázek 39: Chyba dotazu Q5, zdroj: autor

Přestože se chyba liší od chyby s dotazem Q3, problém spočívá zase v ukládání dat.

Řádky v Cassandra jsou rozděleny podle segmentů a poté jsou seřazeny na disku podle date_hour. Protože v době spuštění již je proveden dotaz na rozsah (non-EQ relace na date_hour), nelze omezit následující klíč (id_detector). Důvodem je to, že takové omezení může diskvalifikovat některé řádky v daném rozsahu. To následně vede k neefektivnímu, nespojitému

čtení. Cassandra nenabízí velkou míru flexibility dotazů a je navržena tak, aby chránila před psaním podobných dotazů, které mohou mít nepředvídatelný výkon.

Po použití funkce ALLOW FILTERING problém byl odstraněn. (obr.40)

```
cqlsh> SELECT * FROM data_from_detectors.data_by_area WHERE area_number = 60069 AND  date_hour >= '2018-10-09 02:27:06+02:00' AND date_hour <= '20
18-10-09 03:44:06+02:00' AND id_detector = 'R600694-Z2' ALLOW FILTERING;

 area_number | date_hour                       | id_detector | intensity_all | intensity_nakladni | intensity_osobni | speed_all | speed_nakladni
| speed_osobni | stop
-------------+---------------------------------+-------------+---------------+--------------------+------------------+-----------+---------------
+--------------+---------------------------------
       60069 | 2018-10-09 01:42:06.000000+0000 |  R600694-Z2 |             6 |                  0 |                6 |        56 |              0
|         56 | 2018-10-09 01:42:06.000000+0000
       60069 | 2018-10-09 01:37:06.000000+0000 |  R600694-Z2 |            14 |                  0 |               14 |        55 |              0
|         55 | 2018-10-09 01:37:06.000000+0000
       60069 | 2018-10-09 01:32:06.000000+0000 |  R600694-Z2 |            10 |                  0 |               10 |        51 |              0
|         51 | 2018-10-09 01:32:06.000000+0000

(3 rows)
```

Obrázek 40: Funkční dotaz Q5 s použitím ALLOW FILTERING, zdroj: autor

Nicméně jak již bylo zmíněno, přestože dotaz je nyní funkčním, použití funkce ALLOW FILTERING není v tomto případě optimální volbou. Je důležité zmínit, že na rozdíl od dotazu Q3, chyba tohoto dotazu nezávisí na struktuře databáze: na tuto chybu mají vliv určitá pravidla omezení, a hlavně míra flexibility dotazů, které nemohou být změněny ani v případě jiné struktury databáze.

Z příkladu tohoto prostředí lze vidět, jak se skutečně krok za krokem tvoří fyzický datový model od začátku až do chvíli, kdy se již pracuje s konkrétními dotazy na vytvořenou databázi. Nejprve se tak musí vytvořit model konceptuální: z něho plyne, jaké jsou v modelu entity, atributy a vztahy, a také aplikační workflow, který definuje budoucí dotazy na databázi a podle kterého je tvořena databázová struktura. Následuje tvoření logického modelu, který již je grafickým znázorněním návrhu databázového schématu. Fyzický datový model je konečným výstupem návrhu prostředí. Obsahuje klíčové prostory, tabulky a všechna data, nad kterými se následně provádějí dotazy. Jeho vytvoření je vytvořením konečného databázového modelu, což bylo cílem dané části práce a její konečným výsledkem.

4.4 Provedení stress testů

Možnosti modelování dat mohou výrazně ovlivnit výkon aplikace. Nejlepší metodou pro zjišťování problémů s konkrétním datovým modelem je testování významného zatížení během několika pokusů. Nástroj cassandra-stress je efektivní nástroj pro naplnění clusteru a zátěžové testování tabulek a dotazů CQL. Testy se v rámci práce prováděly ve virtuálním prostředí na systémovém nodu Cassandra, a proto mohou být trochu zkresleny oproti použití na fyzickém serveru.

Nástroj cassandra-stress je nástroj založený na prostředí Java pro zátěžové testování clusteru Cassandra.

4.4.1 Write test

Pro začátek takového testu, který se obvykle skládá z write a read testů, je potřeba nejprve naplnit existující tabulky daty, a proto jako první se provádí tzv. write test. Spustí se pomocí speciálního příkazu:

```
cassandra-stress write
```

Tento příkaz provede více než milion zápisů, a to několikrát po sobě. Jak tento proces začíná po uvedení daného příkazu je vidět z obr. 41.

```
root@bigdata:/# cassandra-stress write
******************* Stress Settings *******************
Command:
  Type: write
  Count: -1
  No Warmup: false
  Consistency Level: LOCAL_ONE
  Target Uncertainty: 0.020
  Minimum Uncertainty Measurements: 30
  Maximum Uncertainty Measurements: 200
  Key Size (bytes): 10
  Counter Increment Distibution: add=fixed(1)
Rate:
  Auto: true
  Min Threads: 4
  Max Threads: 1000
Population:
  Sequence: 1..1000000
  Order: ARBITRARY
  Wrap: true
Insert:
```

Obrázek 41: Začátek procesu zápisu, zdroj: autor

Následně se napojí na potřebný cluster (což je v tomto případě FD Cluster) a začne fyzicky do něho zapisovat data. Tento proces je ukázán na obr.42.

```
Connected to cluster: FD Cluster, max pending requests per connection 128, max connections per host 8
Datacenter: datacenter1; Host: localhost/127.0.0.1:9042; Rack: rack1
Created keyspaces. Sleeping 1s for propagation.
Sleeping 2s...
Warming up WRITE with 50000 iterations...
Failed to connect over JMX; not collecting these stats
Thread count was not specified
WARNING: uncertainty mode (err<) results in uneven workload between thread runs, so should be used for high level analysis only

Running with 4 threadCount
Running WRITE with 4 threads until stderr of mean < 0.02
Failed to connect over JMX; not collecting these stats
type                                          total ops,    op/s,    pk/s,   row/s,    mean,     med,     .95,     .99,    .999,     max,   t
ime,   stderr, errors,  gc: #,  max ms,  sum ms,  sdv ms,      mb
total,                                              638,     638,     638,     638,     1.9,     1.4,     4.3,     9.0,    15.4,    16.2,
1.0,  0.00000,      0,      0,       0,       0,       0,      0
total,                                             3211,    2573,    2573,    2573,     1.5,     1.3,     2.8,     5.7,    10.2,    15.3,
2.0,  0.42592,      0,      0,       0,       0,       0,      0
total,                                             6351,    3140,    3140,    3140,     1.2,     1.1,     2.2,     4.6,     7.2,    21.9,
3.0,  0.29290,      0,      0,       0,       0,       0,      0
total,                                             7392,    1041,    1041,    1041,     3.8,     1.3,     5.4,     9.8,   506.2,   520.1,
4.0,  0.21937,      0,      0,       0,       0,       0,      0
total,                                            10207,    2815,    2815,    2815,     1.4,     1.1,     2.7,     6.2,    14.8,    15.5,
5.0,  0.17357,      0,      0,       0,       0,       0,      0
```

Obrázek 42: Proces zápisu dat, zdroj: autor

Po nějaké době se proces skončí a Cassandra zobrazí výsledky. (obr.43). Všechny údaje je potřeba důkladně interpretovat.

```
Results:
Op rate                   :    5,853 op/s  [WRITE: 5,853 op/s]
Partition rate            :    5,853 pk/s  [WRITE: 5,853 pk/s]
Row rate                  :    5,853 row/s [WRITE: 5,853 row/s]
Latency mean              :    0.7 ms [WRITE: 0.7 ms]
Latency median            :    0.5 ms [WRITE: 0.5 ms]
Latency 95th percentile   :    1.1 ms [WRITE: 1.1 ms]
Latency 99th percentile   :    2.4 ms [WRITE: 2.4 ms]
Latency 99.9th percentile :   10.4 ms [WRITE: 10.4 ms]
Latency max               :  511.7 ms [WRITE: 511.7 ms]
Total partitions          :  1,172,305 [WRITE: 1,172,305]
Total errors              :          0 [WRITE: 0]
Total GC count            : 0
Total GC memory           : 0.000 KiB
Total GC time             :    0.0 seconds
Avg GC time               :    NaN ms
StdDev GC time            :    0.0 ms
Total operation time      : 00:03:20

Sleeping for 15s
```

Obrázek 43: Výsledky write testu, zdroj: autor

Pro přehlednost byly výsledky zobrazeny do tabulky 2 včetně jednotlivých popisů.

Tabulka 2: Výsledky stress testu na zápis

Data	Hodnoty	Popis
`total ops`	1172305	Celkový počet spuštěných operací
`op/s`	13	Počet operací za sekundu
`pk/s`	13	Počet operací oddílu za sekundu
`row/s`	13	Počet operací řádků za sekundu
`mean`	486.9	Průměrná latence v milisekundách pro každou operaci
`med`	501.5	Střední latence v milisekundách pro každou operaci
`.95`	503.1	95% času byla latence menší než číslo zobrazené ve sloupci.
`.99`	503.1	V 99% případů byla latence menší než číslo zobrazené ve sloupci.
`.999`	503.1	99,9% času byla latence menší než číslo zobrazené ve sloupci.
`max`	200.3	Maximální latence v milisekundách.
`stderr`	0	Standardní chybový výstup. Čím menší je hodnota, tím přesnější je míra výkonu clusteru.
`gc: #`	0	Garbage collection (způsob automatické správy paměti: vyhledává a uvolňuje úseky paměti, které již program nebo proces nepoužívá)
`max ms`	0	Nejdelší sběr garbage collection za milisekundy.
`sum ms`	0	Celková práce garbage collection v milisekundách.
`sdv ms`	0	Směrodatná odchylka v milisekundách.
`mb`	0	Velikost garbage collection v megabajtech.

Z tabulky lze vidět, že více než jeden milion operací zápisu se provedly během 3 minut a 20 vteřin. Čím je ten čas kratší, tím lépe je navrženo schéma databáze z pohledu funkčnosti. Nicméně o rychlostech se dá soudit a nějak je porovnávat až ve chvíli, kdy se přidávají nové uzly, protože přidáváním uzlů se zvyšuje práce konána clusterem. V rámci testovacího prostředí hlavním kritériem správného návrhu je uskutečnění testu od začátku až do konce a také počet nalezených chyb během testování.

Z tabulky 2 je vidět, že počet chyb během celého procesu zůstal na 0, což je základem pro správné fungování databáze.

Standardní chybový výstup také zůstal na 0, což říká o vysoké přesnosti míry výkonu clusteru.

4.4.2 Read test

V tuto chvíli jsou tabulky naplněny daty a jak již bylo zmíněno je vhodné provést tzv. read test, který ukáže, jak rychlý je proces čtení dat ve vytvořené databází.

Test se spustí pomocí speciálního příkazu:

```
cassandra-stress read
```

Tento příkaz zase provede více než milion operací (tentokrát čtení), a to několikrát po sobě. Jak tento proces začíná po uvedení daného příkazu je vidět z obr. 44.

```
root@bigdata:/# cassandra-stress read
******************* Stress Settings *******************
Command:
  Type: read
  Count: -1
  No Warmup: false
  Consistency Level: LOCAL_ONE
  Target Uncertainty: 0.020
  Minimum Uncertainty Measurements: 30
  Maximum Uncertainty Measurements: 200
  Key Size (bytes): 10
  Counter Increment Distibution: add=fixed(1)
Rate:
  Auto: true
  Min Threads: 4
  Max Threads: 1000
Population:
  Distribution: Gaussian:  min=1,max=1000000,mean=500000.500000,stdev=166666.500000
  Order: ARBITRARY
  Wrap: false
Insert:
  Revisits: Uniform:  min=1,max=1000000
  Visits: Fixed:  key=1
  Row Population Ratio: Ratio: divisor=1.000000;delegate=Fixed:  key=1
  Batch Type: not batching
Columns:
  Max Columns Per Key: 5
  Column Names: [C0, C1, C2, C3, C4]
  Comparator: AsciiType
  Timestamp: null
```

Obrázek 44: Začátek procesu čtení, zdroj: autor

Následně se zase napojí na vytvořený FD Cluster (stejně jako v případě write testu) a začne se proces čtení, který je ukázán na obr.45.

```
Sleeping 2s...
Warming up READ with 50000 iterations...
Connected to cluster: FD Cluster, max pending requests per connection 128, max connections per host 8
Datacenter: datacenter1; Host: localhost/127.0.0.1:9042; Rack: rack1
Failed to connect over JMX; not collecting these stats
^[[A^[[AThread count was not specified
WARNING: uncertainty mode (err<) results in uneven workload between thread runs, so should be used for high level analysis only

Running with 4 threadCount
Running READ with 4 threads until stderr of mean < 0.02
Failed to connect over JMX; not collecting these stats
type                                                 total ops,    op/s,    pk/s,   row/s,    mean,     med,     .95,     .99,    .999,     max,
ime,   stderr, errors,  gc: #,  max ms,  sum ms,  sdv ms,      mb
total,                                                    1190,    1190,    1190,    1190,     1.5,     1.2,     3.4,     7.3,    10.1,    10.9,
1.0,  0.00000,      0,      0,       0,       0,       0,      0
total,                                                    4628,    3438,    3438,    3438,     1.1,     0.9,     2.2,     4.5,    14.5,    16.2,
2.0,  0.34491,      0,      0,       0,       0,       0,      0
total,                                                    8611,    3983,    3983,    3983,     1.0,     0.8,     1.8,     3.3,     6.0,    11.6,
3.0,  0.24330,      0,      0,       0,       0,       0,      0
total,                                                   11539,    2928,    2928,    2928,     1.3,     0.9,     2.0,     4.1,   184.2,   198.8,
4.0,  0.17924,      0,      0,       0,       0,       0,      0
total,                                                   15327,    3788,    3788,    3788,     1.0,     0.8,     1.8,     4.3,    15.5,    36.3,
5.0,  0.14355,      0,      0,       0,       0,       0,      0
total,                                                   19745,    4418,    4418,    4418,     0.9,     0.8,     1.6,     2.9,     7.2,    10.4,
6.0,  0.12447,      0,      0,       0,       0,       0,      0
total,                                                   25335,    5590,    5590,    5590,     0.7,     0.6,     1.2,     2.1,     8.4,    17.1,
7.0,  0.12492,      0,      0,       0,       0,       0,      0
total,                                                   31210,    5875,    5875,    5875,     0.7,     0.6,     1.2,     2.5,     7.5,    13.4,
8.0,  0.11972,      0,      0,       0,       0,       0,      0
total,                                                   37601,    6391,    6391,    6391,     0.6,     0.5,     1.0,     2.0,     4.5,     6.6,
```

Obrázek 45: Proces čtení dat, zdroj: autor

A po nějaké době to Cassandra zase vyhodnotí. (obr.46)

```
Results:
Op rate                   :    5,709 op/s  [READ: 5,709 op/s]
Partition rate            :    5,709 pk/s  [READ: 5,709 pk/s]
Row rate                  :    5,709 row/s [READ: 5,709 row/s]
Latency mean              :    0.7 ms [READ: 0.7 ms]
Latency median            :    0.5 ms [READ: 0.5 ms]
Latency 95th percentile   :    0.9 ms [READ: 0.9 ms]
Latency 99th percentile   :    1.8 ms [READ: 1.8 ms]
Latency 99.9th percentile :    8.8 ms [READ: 8.8 ms]
Latency max               :  508.0 ms [READ: 508.0 ms]
Total partitions          :    822,059 [READ: 822,059]
Total errors              :          0 [READ: 0]
Total GC count            : 0
Total GC memory           : 0.000 KiB
Total GC time             :    0.0 seconds
Avg GC time               :    NaN ms
StdDev GC time            :    0.0 ms
Total operation time      : 00:02:24
```

Obrázek 46: Výsledky read testu, zdroj: autor

Z toho obrázku již lze vidět, že proces čtení probíhal velmi podobným způsobem, jako v případě testu zápisu. Cassandra nezaznamenala žádnou chybu během operací, což je pro celý proces základem. Celkový čas provedení testu je ale skoro o minutu kratší, protože proces čtení je pro Cassandra obecně rychlejší a jednodušší.

Po provedení obou testu lze učinit závěr, že struktura databáze v testovacím prostředí byla vytvořena vhodným způsobem: oba testy se provedly od začátku až do konce, a ani v jednom z testů se nevyskytla chyba. Z těchto vyhodnocení lze následně definovat navrženou strukturu jako vhodnou pro budoucí rozšiřování databáze dopravních dat.

Závěr

Cílem této práce bylo studovat a analyzovat problematiku NoSQL databází, jejich výhody, vlastnosti a specifika použití. V teoretické části jsou popsány SQL a NoSQL databáze, uvedeny jejich rozdíly a funkcionality a následuje bližší seznámení s NoSQL, včetně výsledného výběru konkrétního nástroje pro vytvoření prostředí pro práci s dopravními daty.

První kapitola se zaměřuje na SQL databáze, jejich charakteristiku, včetně funkcionality, výhod a standardů tohoto typu databáze.

Ve druhé kapitole je popsána funkcionalita databází typu NoSQL. Důraz je kladen na principy fungování takových systémů, hlavní kritéria a silné a slabé stránky, které z toho vyplívají. Následuje výsledné porovnání databází typu SQL a NoSQL.

Třetí kapitola se zaměřuje na seznámení s typy NoSQL databází včetně uvedení konkrétních příkladů nástrojů. Typy jsou podrobně popsány a poté i porovnány mezi sebou, což udává informaci pro konečný výběr typu a nástroje pro budoucí návrh prostředí.

Praktická část diplomové práce byla stanovena na samotný návrh prostředí pro práci s konkrétními dopravními daty. Nejprve čtenáře seznámí s větou CAP, která je nejdůležitější pro následné pochopení fungování vybraného prostředí Cassandra. Následuje popis prostředí včetně napojení, vytváření tabulek a vkládání dat.

Hlavní části je pak samotná realizace databáze dopravních dat, která byla popsána od úplného začátku a představuje praktický návod na vytvoření. Nejprve popíše základní nastavení a následuje tvorba konceptuálního a logického modelu, a nakonec fyzická implementace.

Na konci práce se prováděly stress testy databáze, pomocí kterých lze následně ověřit, jestli je celá databáze navržena správně. Po provedení dvou testů byl udělán závěr, že návrh je plně funkční.

Je zřejmé, že vytvořený návrh databáze se může v budoucnu rozšiřovat o další klíčové prostory, tabulky a naplňovat se větším množstvím dat. Nicméně stress test ukázal, že systém je navržen takovým způsobem, že je na to připraven.

Při budoucím rozšiřování je nezbytně nutné klast důraz na budoucí potřeby. Databáze takového typu jsou opravdu připraveny na zápis a čtení obrovského množství dat, nicméně je potřeba tato data ukládat správně podle všech pravidel Cassandra.

Výsledky práce přinášejí závěr, který definuje NoSQL databáze jako nástroj budoucna. Objem informací na světě se každoročně zvyšuje o 30 % a je potřeba zavádět nové způsoby pro práci s nimi. Počet zdrojů informace pro tento problém není dodnes dostatečně velký, ale je opravdu nutné šířit informaci o dané problematice.

Bibliografie

[1] DATE, C. J. *An Introduction to Database Systems*. 2. Ed. World Stud. Ser. 1. print. Amsterdam: Addison-Wesley Publishing Company, 1979.

[2] HOLUBOVÁ, Irena, Jiří KOSEK, Karel MINAŘÍK a David NOVÁK. *Big Data a NoSQL databáze*. Praha: Grada, 2015. Profesionál. ISBN 978-80-247-5466-6.

[3] TIWARI, Shashank: Professional NoSQL. 1. vydáni, Wrox, 2011. ISBN 978-047A942246.

[4] GROFF, James; WEINBERG, Paul; and OPPEL, Andy. SQL The Complete Reference, 3rd Edition. US: McGraw-Hill Osborne Media, 2009.

[5] Prezentace na téma: Relační DBMS - systém správy relační databáze. Archiv souborů pro studenty. Copyright © [cit. 25.05.2021]. Dostupné z: https://studfile.net/preview/4428260/page:2/

[6] CHEN, J.-K.; LEE, W.-Z. An Introduction of NoSQL Databases Based on Their Categories and Application Industries. *Algorithms* 2019, *12*, 106.

[7] SACHA, J., DOWLING, J. (2005). A Gradient Topology for Master-Slave Replication in Peer-to-Peer Environments. 4125. 86-97. 10.1007/978-3-540-71661-7_8.

[8] LI, Z. (2018). NoSQL Databases. The Geographic Information Science & Technology Body of Knowledge (2nd Quarter 2018 Edition), John P. Wilson (Ed).

[9] AJAYI, R. (2015). Acid Properties, CAP Theorem & Mobile Databases. 10.13140/RG.2.1.1941.0084.

[10] ČVUT DSpace [online]. Copyright © [cit. 29.06.2021]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/85980/F6-DP-2019-Krejci-Jakub-F6-DP-2019-krejci-jakub-diplomova-prace.pdf?sequence=-1&isAllowed=y

[11] About Cassandra Replication Factor and Consistency Level. Apigee Edge | Apigee Docs [online]. Copyright © [cit. 29.06.2021]. Dostupné z: https://docs.apigee.com/private-cloud/v4.17.09/about-cassandra-replication-factor-and-consistency-level

[12] What is Cassandra? | DataStax. DataStax | NoSQL Database Built on Apache Cassandra [online]. Copyright © [cit. 29.06.2021]. Dostupné z: https://www.datastax.com/cassandra

Seznam obrázků

OBRÁZEK 1: PŘÍSTUP K DATABÁZI POMOCÍ SQL 14
OBRÁZEK 2: PŘÍKLAD RELAČNÍHO MODELU DAT 17
OBRÁZEK 3: PŘÍKLAD NERELAČNÍHO MODELU DAT (DOKUMENT JSON) 18
OBRÁZEK 4: PŘÍKLAD ÚLOŽIŠTĚ KLÍČ-HODNOTA 25
OBRÁZEK 5: PŘÍKLAD DOKUMENTOVÉ DATABÁZE 26
OBRÁZEK 6: ÚLOŽIŠTĚ ZALOŽENÉ NA ŘÁDCÍCH VERSUS ÚLOŽIŠTĚ ZALOŽENÉ NA SLOUPCÍCH 27
OBRÁZEK 7: PŘÍKLAD DOKUMENTOVÉ DATABÁZE 28
OBRÁZEK 8: DOTAZ NA CASSANDRA (JE AKTIVNÍ) 35
OBRÁZEK 9: ZASTAVENÍ CASSANDRA, ZDROJ: AUTOR 35
OBRÁZEK 10: DOTAZ NA CASSANDRA (JE NEAKTIVNÍ) 36
OBRÁZEK 11: DOTAZ NA 1.UZEL 36
OBRÁZEK 12: PŘIPOJENÍ TŘETÍHO UZLŮ 36
OBRÁZEK 13: FUNKČNÍ UZLY 37
OBRÁZEK 14: SPUŠTĚNÍ JAZYKA CASSANDRA 37
OBRÁZEK 15: PŘÍKLAD KLÍČOVÉHO PROSTORU 37
OBRÁZEK 16: PŘÍKLAD TABULKY 38
OBRÁZEK 17: ZÁKLADNÍ FÁZE VYTVOŘENÍ DATOVÉHO MODELU 39
OBRÁZEK 18: PŘEJMENOVÁNÍ CLUSTERU 40
OBRÁZEK 19: KONCEPTUÁLNÍ MODEL 42
OBRÁZEK 20: APPLICATION WORKFLOW 42
OBRÁZEK 21: SCHÉMA TVORBY LOGICKÉHO MODELU 44
OBRÁZEK 22: POSTUP TVORBY LOGICKÉHO MODELU 44
OBRÁZEK 23: CHEBOTKO DIAGRAM 45
OBRÁZEK 24: TVORBA KEYSPACE DATA_FROM_DETECTORS 46
OBRÁZEK 25: TVORBA TABULKY AREA 46
OBRÁZEK 26: TVORBA TABULKY DETECTORS 46
OBRÁZEK 27: TVORBA TABULKY DATA 47
OBRÁZEK 28: NAPLNĚNÍ TABULKY AREAS 48
OBRÁZEK 29: OVĚŘENÍ NAČTENÍ DAT DO TABULKY AREAS 48
OBRÁZEK 30: VÝSLEDEK DOTAZU Q1 49
OBRÁZEK 31: NAPLNĚNÍ TABULKY DETECTORS 50
OBRÁZEK 32: OVĚŘENÍ NAČTENÍ DAT DO TABULKY DETECTORS 50
OBRÁZEK 33: VÝSLEDEK DOTAZU Q2 51
OBRÁZEK 34: NAPLNĚNÍ TABULKY DATA 51
OBRÁZEK 35: OVĚŘENÍ NAČTENÍ DAT DO TABULKY DATA 52
OBRÁZEK 36: CHYBA DOTAZU Q3 NAD VŠEMI DATY V TABULCE DATA 52
OBRÁZEK 37: FUNKČNÍ DOTAZ Q3 S POUŽITÍM ALLOW FILTERING 53

OBRÁZEK 38: FUNKČNÍ DOTAZ Q4 54
OBRÁZEK 39: CHYBA DOTAZU Q5 54
OBRÁZEK 40: FUNKČNÍ DOTAZ Q5 S POUŽITÍM ALLOW FILTERING 55
OBRÁZEK 41: ZAČÁTEK PROCESU ZÁPISU 56
OBRÁZEK 42: PROCES ZÁPISU DAT 57
OBRÁZEK 43: VÝSLEDKY WRITE TESTU 57
OBRÁZEK 44: ZAČÁTEK PROCESU ČTENÍ 60
OBRÁZEK 45: PROCES ČTENÍ DAT 60
OBRÁZEK 46: VÝSLEDKY READ TESTU 61

Seznam tabulek

TABULKA 1: POROVNÁNÍ RELAČNÍCH A NERELAČNÍCH DATABÁZÍ 21
TABULKA 2: VÝSLEDKY STRESS TESTU NA ZÁPIS 58

Přílohy

Skript 1

```
import glob
import re
import os

dir_zdroj = './zdroj'
dir_xml = './xml'

#kod programu na opravu souboru, rozdeleni a tvorbu xml
def zpracuj_soubor(soubory):
    for filename in soubory:
        # Open file to read
        with open(dir_zdroj+'/'+filename, "r") as r:
            n=1
            for i, line in enumerate(r):
                z = re.match('(^</dete.*)(<\?xml .*)', line)
                if z:
                    with open(dir_xml+'/'+"{}-{}.xml".format(filename, n), "a") as f:
                        f.write(z.groups()[0])
                    n+=1
                    # Write lines to file
                    with open(dir_xml+'/'+"{}-{}.xml".format(filename, n), "a") as f:
                        f.write(z.groups()[1])
                else:
                    # Write lines to file
                    with open(dir_xml+'/'+"{}-{}.xml".format(filename, n), "a") as f:
                        z = re.match('(^.*>)', line)
                        if z:
                            f.write(z.groups()[0])
                        else:
                            f.write(line)

#funkce, ktera smaze soubory "profil*.xml" a vytvori seznam profile*.txt souboru
#uvnitr aktualniho adresare
def priprav_soubory():
```

```
    #smaz soubory profile*.xml
    os.chdir(dir_xml)
    for file in glob.glob("profil*.xml"):
        os.remove(file)

    os.chdir("..")
    #vytvor pole souboru profile*.txt s daty
    zdroj_souboru = []
    os.chdir(dir_zdroj)
    for file in glob.glob("profil*.txt"):
        zdroj_souboru.append(file)
    os.chdir("..")
    return zdroj_souboru


# kod programu
#soubor = 'profil_02_2018-10-13.txt'
zdroj_dat = priprav_soubory()
print(zdroj_dat)
#soubor = zdroj_dat[0]
zpracuj_soubor(zdroj_dat)
```

Skript 2

```
import glob
import os
import lxml.etree as ET
import os.path
from os import path

#zdroj XML souboru
dir_xml = './xml'
file_csv = 'output_detectors_by_area2.csv'

detector = {}



detector['R501061'] = {"area_number": '50106', "direction": 'Z'}
detector['R501062'] = {"area_number": '50106', "direction": 'V'}

detector['R501502'] = {"area_number": '50150', "direction": 'Z'}
detector['R501503'] = {"area_number": '50150', "direction": 'V'}

detector['R501576'] = {"area_number": '50157', "direction": 'Z'}
detector['R501575'] = {"area_number": '50157', "direction": 'V'}

detector['R506639'] = {"area_number": '50663', "direction": 'Z'}
detector['R506640'] = {"area_number": '50663', "direction": 'V'}

detector['R506998'] = {"area_number": '50699', "direction": 'J'}
detector['R507001'] = {"area_number": '50699', "direction": 'S'}

detector['R510311'] = {"area_number": '51031', "direction": 'Z+V'}

detector['R512142'] = {"area_number": '51211', "direction": 'J'}
detector['R512118'] = {"area_number": '51211', "direction": 'S'}

detector['R520303'] = {"area_number": '52030', "direction": 'J'}
detector['R520302'] = {"area_number": '52030', "direction": 'S'}

detector['R520339'] = {"area_number": '52033', "direction": 'J'}
```

```
detector['R520338'] = {"area_number": '52033', "direction": 'S'}

detector['R600694'] = {"area_number": '60069', "direction": 'Z'}
detector['R600693'] = {"area_number": '60069', "direction": 'V'}

detector['R606716'] = {"area_number": '60671', "direction": 'Z'}
detector['R606713'] = {"area_number": '60671', "direction": 'V'}

detector['R614950'] = {"area_number": '61495', "direction": 'J'}
detector['R614951'] = {"area_number": '61495', "direction": 'S'}

detector['R617106'] = {"area_number": '61710', "direction": 'Z'}
detector['R617105'] = {"area_number": '61710', "direction": 'V'}

detector['R617610'] = {"area_number": '61761', "direction": 'Z'}
detector['R619462'] = {"area_number": '61761', "direction": 'V'}

detector['R701387'] = {"area_number": '70138', "direction": 'J'}
detector['R701386'] = {"area_number": '70138', "direction": 'S'}


def vytvor_csv(soubor):
    with open(file_csv, "a") as f:
        if (os.stat(file_csv).st_size == 0):
            #hlavicka
            hlavicka = '"detector_id";"area_number";"type";"direction"'
            f.write(hlavicka)

        #naplnim daty
        xml = ET.parse(dir_xml+'/'+soubor)
        koren_xml = xml.getroot()
        #print(len(koren_xml.getchildren()))
        for node in koren_xml.iter():
            if node.tag == "detector":
                radek_dat = ''


                #detector_id
                radek_dat = '\n"'+node.get('id')+'"'
```

```
            #detector_area_number
            radek_dat+= ';"'+nalezni_v_poli(detector,node.get('id'))['area_number']+'"'


            #dector_type
            radek_dat+= ';"'+node.get('type')+'"'


            #detector_direction
            radek_dat+= ';"'+nalezni_v_poli(detector,node.get('id'))['direction']+'"'

            f.write(radek_dat)

#hledani v asociativni poli
def nalezni_v_poli(kde_hledam, co_hledam):
    hodnota = ''
    for key in kde_hledam:
        if (co_hledam.find(key) != -1):
            hodnota = kde_hledam[key]
    return hodnota



#vlastni program
#1) smazu vystupni soubor - pokud existuje
if (path.exists(file_csv)):
    os.remove(file_csv)

#2) proskenuju adresar s XML soubory a zapisu si je do pole
pole_souboru = []
os.chdir(dir_xml)
for file in glob.glob("profil*.xml"):
  pole_souboru.append(file)
os.chdir("..")

#3) projdu kazdy XML soubor a data zapisu do vystupniho souboru
for jeden_soubor in pole_souboru:
    vytvor_csv(jeden_soubor)
```

Skript 3

```
import glob
import os
import lxml.etree as ET
from os import path

#zdroj XML souboru
dir_xml = './xml'
file_csv = 'output_detectors-data.csv'

detector = {}



detector['R501061'] = {"area_number": '50106', "direction": 'Z'}
detector['R501062'] = {"area_number": '50106', "direction": 'V'}

detector['R501502'] = {"area_number": '50150', "direction": 'Z'}
detector['R501503'] = {"area_number": '50150', "direction": 'V'}

detector['R501576'] = {"area_number": '50157', "direction": 'Z'}
detector['R501575'] = {"area_number": '50157', "direction": 'V'}

detector['R506639'] = {"area_number": '50663', "direction": 'Z'}
detector['R506640'] = {"area_number": '50663', "direction": 'V'}

detector['R506998'] = {"area_number": '50699', "direction": 'J'}
detector['R507001'] = {"area_number": '50699', "direction": 'S'}

detector['R510311'] = {"area_number": '51031', "direction": 'Z+V'}

detector['R512142'] = {"area_number": '51211', "direction": 'J'}
detector['R512118'] = {"area_number": '51211', "direction": 'S'}

detector['R520303'] = {"area_number": '52030', "direction": 'J'}
detector['R520302'] = {"area_number": '52030', "direction": 'S'}

detector['R520339'] = {"area_number": '52033', "direction": 'J'}
detector['R520338'] = {"area_number": '52033', "direction": 'S'}
```

```
detector['R600694'] = {"area_number": '60069', "direction": 'Z'}
detector['R600693'] = {"area_number": '60069', "direction": 'V'}

detector['R606716'] = {"area_number": '60671', "direction": 'Z'}
detector['R606713'] = {"area_number": '60671', "direction": 'V'}

detector['R614950'] = {"area_number": '61495', "direction": 'J'}
detector['R614951'] = {"area_number": '61495', "direction": 'S'}

detector['R617106'] = {"area_number": '61710', "direction": 'Z'}
detector['R617105'] = {"area_number": '61710', "direction": 'V'}

detector['R617610'] = {"area_number": '61761', "direction": 'Z'}
detector['R619462'] = {"area_number": '61761', "direction": 'V'}

detector['R701387'] = {"area_number": '70138', "direction": 'J'}
detector['R701386'] = {"area_number": '70138', "direction": 'S'}


def vytvor_csv(soubor):
    with open(file_csv, "a") as f:
        if (os.stat(file_csv).st_size == 0):
            #hlavicka
            hlavicka = '"id_detector";"area_number";"start_date";"stop_date";"intensity_all";"intensity_osobni";"intensity_nakladni";"speed_all";"speed_osobni";"speed_nakladni";"occupancy_all"'
            f.write(hlavicka)

        #naplnim daty
        xml = ET.parse(dir_xml+'/'+soubor)
        koren_xml = xml.getroot()
        #print(len(koren_xml.getchildren()))
        for node in koren_xml.iter():
            if node.tag == "detector":
                radek_dat = ''
                #detector_id
                radek_dat = '\n"'+node.get('id')+'"'
```

```
            #detector_area_number
            radek_dat+= ';"'+nalezni_v_poli(detector,node.get('id'))['area_number']+'"'

            #start_date
            radek_dat+= ';"'+node[2].text+'"'
            #stop_date
            radek_dat+= ';"'+node[2].text+'"'
            #intensity_all
            radek_dat+= ';"'+node[3][0].text+'"'
            #intensity_osobni
            radek_dat+= ';"'+node[3][1].text+'"'
            #intensity_nakladni
            radek_dat+= ';"'+node[3][2].text+'"'
            #speed_all
            radek_dat+= ';"'+node[4][0].text+'"'
            #speed_osobni
            radek_dat+= ';"'+node[4][1].text+'"'
            #speed_nakladni
            radek_dat+= ';"'+node[4][2].text+'"'
            #ocupancy_all
            radek_dat+= ';"'+node[5][0].text+'"'

            f.write(radek_dat)

#hledani v asociativni poli
def nalezni_v_poli(kde_hledam, co_hledam):
   hodnota = ''
   for key in kde_hledam:
      if (co_hledam.find(key) != -1):
         hodnota = kde_hledam[key]
   return hodnota


#vlastni program
#1) smazu vystupni soubor - pokud existuje
if (path.exists(file_csv)):
   os.remove(file_csv)

#2) proskenuju adresar s XML soubory a zapisu si je do pole
```

```
pole_souboru = []
os.chdir(dir_xml)
for file in glob.glob("profil*.xml"):
    pole_souboru.append(file)
os.chdir("..")

#3) projdu kazdy XML soubor a data zapisu do vystupniho souboru
for jeden_soubor in pole_souboru:
    vytvor_csv(jeden_soubor)
```