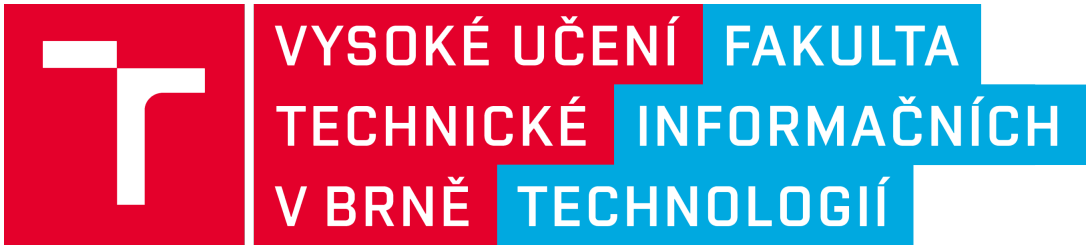


UPA – projekt, 1. část

Ukládání rozsáhlých dat v NoSQL databázích

Lucie Svobodová, Jakub Kuzník, Adam Kaňkovský

xsvobo1x, xkuzni04, xkanko00

29. října 2023

Tým 0903

Obsah

1 Dokumentová databáze – MongoDB

Typ NoSQL databáze	Databáze dokumentů
Zvolená databáze	MongoDB
Zvolená datová sada	Pítka
Formát dat	GeoJSON

1.1 Důvod výběru MongoDB

Hlavním důvodem pro výběr MongoDB namísto ostatních NoSql databází, je možnost efektivně pracovat s geoprostorovými daty. MongoDB totiž podporuje ukládání dat ve formátech jako jsou například GeoJSON Point, LineString, Polygon, nebo MultiPolygon, nad kterými můžeme efektivně provádět goeprostorové dotazy, jako je například, hledání bodů v polygonu, nebo hledání blízkých bodů. V praxi tak můžeme díky tomuto formátu uložení najít například restauraci v okolí, nebo všechny parky v určité části města. Tuto výhodu si vyzkoušíme i s naší sadou a pomocnou sadou Hranice městských částí pomocí které budeme filtrovat pítka podle městských částí. Tohle jsou však výhody vztahující se pouze na geoprostorová data abychom mohli popsat ostatní výhody MongoDB, musíme si prvně popsat co vlastně MongoDB je. V rámci CAP teoremu se MongoDB řadí do podmnožiny CP, protože je navržená primárně pro vysokou dostupnost a odolnost vůči části sítě, s konfigurovatelnou úrovní konzistence.

1.2 Co je MongoDB

MongoDB je open source NoSql databáze dokumentů postavená na horizontální škálované architektuře a používá flexibilní schéma pro ukládání dat. To znamená, že namísto ukládání dat do tabulek řádků a sloupců, jako jsou ukládány SQL databáze, má MongoDB každý záznam v databázi popsaný dokumentem ve formátu BSON (binární reprezentaci dat). Samotné načítání dat je však možné z více formátů, jako jsou například: JSON, GeoJSON, BSON, CSV, nebo TSV.[1]

1.3 Výhody MongoDB

1. **Flexibilita Schématu:** Jako NoSQL databáze poskytuje MongoDB flexibilitu v tom jak jsou data strukturována. To znamená, že když je potřeba přidat další informace k pítkům, jako je například PH místní vody, není potřeba provádět náročné migrace databáze.
2. **Výkon:** MongoDB je optimalizováno pro vysoký výkon, zejména při manipulaci s velkými objemy dat, což zajišťuje rychlé a efektivní odpovědi na dotazy.
3. **Vysoká dostupnost:** MongoDB nabízí automatické převzetí služeb při selhání a replikaci, což zajišťuje, že data jsou vždy dostupná a chráněná proti selhání hardwaru.
4. **Snadné Škálování:** Databáze MongoDB lze snadno škálovat horizontálně přidáním dalších serverů, což umožňuje efektivní zpracování rostoucího objemu geoprostorových dat.

1.4 Struktura MongoDB

Jelikož je MongoDB databázový systém založený na dokumentech, je jeho struktura značně odlišná od relačních databází. Zde jsou základní stavební prvky MongoDB:

1. **Databáze:** Je organizovaný kontejner pro ukládání dat, který spojuje související datové sady do jednoho logického celku, každá má své vlastní soubory na disku a obsahuje jednu nebo více kolekcí. Databáze můžeme v MongoDB organizovat, zabezpečovat, replikovat nebo škálovat, což nám vede ke

[1] Dokumentace MongoDB: `https://www.mongodb.com/docs/`

zvýšení výkonu, nebo dává možnost ukládat větší množství dat. Pro naše řešení jsme si vytvořili databázi Brno_data, která má v sobě dvě kolekce: Pítka která obsahuje náš primárně zpracovávaný dataset a Hranice městských částí kterou využijeme při dotazu nad naším primárně zpracovaným datasetem.

2. **Kolekce:** Jedná se o skupinu MongoDB dokumentů a jedná se tak o ekvivalent tabulky v relační databázi. Na rozdíl od ní však nemá pevně definované schéma tzn. že jednotlivé dokumenty v kolekci mohou mít různé struktury. Kolekce jsou uvnitř jednotlivých databází.

   Naše kolekce Pítka s námi zpracovaný datasetem obsahuje tyto sloupce s následujícími datovými typy:

   Tabulka 1: Struktura JSON dokumentu

Atribut	Datový Typ
OBJECTID	Object ID
TITLE	String
DESCRIPTION	String
IMAGE_URL	String
GlobalID	String
PARO	String
location	Object
type	String
coordinates	Array

   Velikost většiny domén je omezena stejně jako je omezená velikost jednoho BSON dokumentu a to 16MB. Z námi uvedených datových typů má jiné omezení pouze Object ID a to 12 bajtů.

3. **Dokument:** V MongoDB svými vlastnostmi nahrazuje řádek v porovnání s relační databází. Každý dokument je popsán ve formátu BSON. Dokument obsahuje jedno nebo více polí, kde každé pole má dvojici název(klíč) a hodnotu. Hodnota může nabývat různých datových typů, jako jsou například: Řetězce, čísla, booleovské hodnoty, polí, vnořených dokumentů a dalších. Každý dokument obsahuje také speciální pole _id, které je využíváno jako unikátní identifikátor pro daný dokument v kolekci. Hodnota _id tak musí být v rámci jedné kolekce unikátní.

   Dokument může obsahovat také vnořené dokumenty, které v MongoDB umožňují ukládání hierarchických nebo vztahově vázaných dat.

   V našem řešení jsou tak jednotlivé dokumenty/řádky, záznamy o konkretních pítkách, doplněné o geolokace. Jak už bylo dříve definováno velikost každého BSON dokumentu je omezena 16MB.

   Náš dataset obsahuje dva identifikátory a to ObjectID, které jsme při importu použili i jako '_id' kolekce. Dalším identifikátorem je GlobalID který je ve formátu GUID a je generován tak aby byl unikátní napříč časem a prostorem.

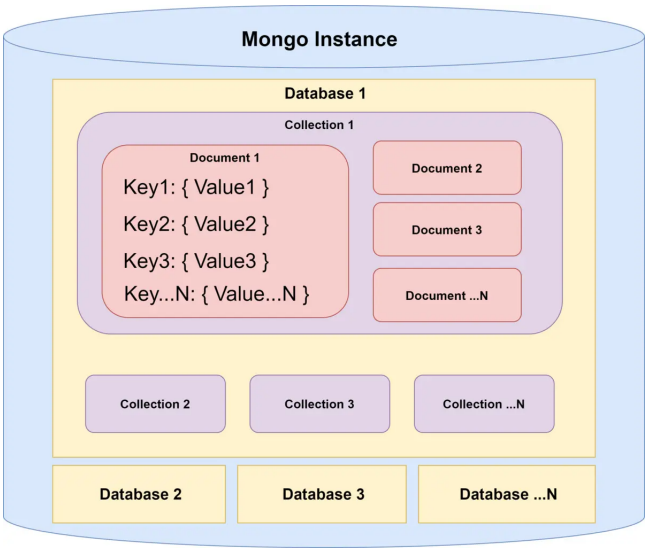
   Reference jsou je možné dělat v MongoDB dvěma různými způsoby. Námi vybraná datová sada sice žádnou takovou referenci neobsahuje, ale s využitím Datové sady Hranice městských částí si později obě zkusíme vytvořit. Způsoby referencí jsou:

   Reference na úrovni Aplikace (Manual References) - Do dokumentu si vložíme id z jiného dokumentu jako referenci. Potom přes tento identifikátor můžeme daný dokument načíst.

   DBRef (Database Reference) - Do dokumentu si jako objekt vložíme celý dokument, nebo pole dokumentu se kterými má daný dokument referenci a k těmto datům potom můžeme přistupovat přímo.

4. **Index:** Indexy, stejně jako v relačních databázích, podporují efektivní vykonávání dotazů nad kolekcí. Bez nich musí MongoDB provádět tzv. "full collection scan", takže prohledává celou kolekci, což může

být velmi časově náročné. Automaticky se v rámci dokumentu vytvářejí indexy podle pole '_id'.



Obrázek 1: Grafická ukázka struktury MongoDB databáze[2]

1.5 Vlastnosti dat

Uložení dat: Data jsou uložena ve formátu GCS WGS84 na stránce `https://data.brno.cz/datasets/mestobrno::p%C3%ADtka-drinking-fountains/about` kde je možné si o datové sadě přečíst i základní informace. Data jsou aktualizovány v nepravidelných intervalech, zatím však ročně, vždy jsou však aktualizovány jen z části tzn. že je přidáno nebo ubráno pouze pár pítek je tak nepravděpodobné u tohoto druhu dat, že by frekvence aktualizací byla nějak veliká.

Možnost zpětné změny: Data jsou možné do datové sady jakkoliv přidávat pokud je však dodrženy už zavedené pravidla identifikátorů jednotlivých řádků. To stejné platí i při mazání dat.

Komprese a Agregace: MongoDB nabízí mnoho kompresních algoritmů ze kterých je možné si vybrat, jako jsou například snappy, zstd a wiredTige (výchozí). Agregace je pak možná pomocí Aggregation Frameworku, který umožňuje provádět složité operace zpracování a transformace dat přímo v databázi.

Distribuce a škálovatelnost úložiště: Obě tyto akce jsou v MongoDB možné a jsou také hojně používány. Jak už bylo dříve řečeno distribuce je v MongoDB možné dosáhnout pomocí shardingu, kdy se data horizontálně rozdělí do více databází shardu, které potom mohou být nezávisle uloženy na více serverech. Škálovatelnost je v MongoDB také možná, protože v každém dokumentu je možné si strukturu definovat trochu jiným způsobem (škálovat jí), ale na ostatní data to nemá žádný vliv.

Redundance: MongoDB poskytuje vysokou odolnost vůči selhání pomocí replikacích sad, které umožňují udržovat několik kopií dat napříč různými servery nebo datovými centry. V případě selhání jednoho uzlu mohou ostatní uzly převzít jeho funkci a zaručit tak nepřetržitý přístup k datům.

1.6 Definice úložiště

MongoDB ve svých dokumentech nemá pevně dané schéma jako relační databáze. To znamená, že není nutné předem definovat strukturu dat, která budou uložena. Místo toho MongoDB ukládá data ve formátu BSON (binární JSON), a můžete do ní vkládat dokumenty s libovolnou strukturou.

[2]Zdroj: `https://www.infoworld.com/article/3623357/what-is-mongodb-a-quick-guide-for-developers.html`

I když MongoDB nemá pevně dané schéma, můžete stále definovat tzv. ”validation rules”(pravidla pro validaci) pro konkrétní kolekce, aby byla data ukládána konzistentním způsobem.

Zde jsou některé základní příkazy pro práci s úložištěm v MongoDB:

```
use nazevDatabaze
```

Listing 1: Vytvoření a výběr databáze

```
db.createCollection("pitka", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["title", "location"],
      properties: {
        title: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        description: {
          bsonType: "string",
          description: "must be a string"
        },
        imageUrl: {
          bsonType: "string",
          description: "must be a string"
        },
        globalId: {
          bsonType: "string",
          description: "must be a string"
        },
        paro: {
          bsonType: "string",
          description: "must be a boolean"
        },
        location: {
          bsonType: "object",
          required: ["type", "coordinates"],
          properties: {
            type: {
              enum: ["Point"],
              description: "must be a Point"
            },
            coordinates: {
              bsonType: "array",
              minItems: 2,
              maxItems: 2,
              items: {
                bsonType: "double"
              },
              description: "must be an array of two numbers"
            }
          }
        }
      }
    }
  }
```

```
49 })
```

Listing 2: Vytvoření kolekce s validačním schématem pro datovou sadu Pítek

```
db.pitecka.insert({
    _id: 1,
    OBJECTID: 1,
    TITLE: 'Ulice husitská',
    DESCRIPTION: 'Pítko je situované na rušné Husitské ulici a to v pruhu
    mezi komunikacemi, přímo na trase přechodů v těsné blízkosti Palackého
    třídy.',
    IMAGE_URL: 'https://lh3.googleusercontent.com/pw/ACtC-3c2NC_',
    GlobalID: '{E4351F09-AD6D-44F3-BA58-7C445918C6F1}',
    PARO: null,
    location: {
        type: 'Point',
        coordinates: [
            16.594488412000032,
            49.22402152700005
        ]
    }
})
```

Listing 3: Vložení dokumentu do kolekce

1.7 Algoritmický popis importu dat

Algoritmický popis importu dat je rozšířený o části kódu, který je psaný v jazyce python s využitím knihovny pymongo[3].

V prvním kroku bylo nutné stanovit připojení s MongoDB serverem, vytvořit databázi a kolekci pro Pítka.

```
client = MongoClient("mongodb://login:password@localhost:27017")
db = client.get_database("Brno_data")
db["pitka"].drop()
pitka_collection = db.create_collection("pitka")
```

Listing 4: Stanovení připojení, vytvoření databáze a kolekce

Dále bylo nutné data z datové sady trochu reformátovat pro jednodušší práci s nimi. Bylo potřeba definovat si '_id' pro jednoznačnou identifikaci hodnot a bylo nutné přesunout geoprostorová data do stejné struktury. Poté už jsme data pouze vložili do kolekce. K tomu byl použitý příkaz insert_many, který dokáže jakýkoliv slovník vložit do mongoDB.

```
def reformat_data_id(data, id_name):
    inserted_data = []
    for feature in data['features']:
        feature_properties = feature['properties']
        feature_properties['location'] = feature['geometry']
        _id = feature_properties.get(id_name)
        if _id:
            feature_properties['_id'] = _id
            inserted_data.append(feature_properties)

```

[3]Skript s kroky popsanými v tomto popisu je možné nalézt v `scripts/mongodb_init.py`

```
    return inserted_data

data_to_insert = reformat_data_id(pitka_dict, 'OBJECTID')

if data_to_insert:
    hranice_collection.insert_many(data_to_insert)
```

Listing 5: Reformátování dat a vložení do kolekce

Upsert je možný v mongoDB prověs dvěma způsoby. První možností je příkaz update_one a druhý je repace_one.

```
for one_value in data_to_upsert:
    collection.replace_one(filter={"_id": one_value["_id"]},
    replacement=one_value, upsert=True)
```

Listing 6: Upsert hodnot v kolekci pomocí replace_one

1.8 Dotaz v jazyce daného databázového produktu.

Pro využití plného potenciálu, které mongoDB nabízí jsme si pro vytvoření tohoto dotazu přidali do naší databáze už dříve zmiňovanou kolekci Hranice městských částí, která obsahuje multipolygony geograficky popisující tyto části. My tak budeme moci vytvořit dotaz, který nám vrátí pouze pítka z některé konkrétní městské části.

Pro zlepšení vyhledávání v obou kolekcích je nejdříve nutné si vytvořit index nad těmito geoprostorovými daty:

```
db.pitka.createIndex({ "location": "2dsphere" })
db.hranice.createIndex({ "location": "2dsphere" })
```

Listing 7: Vytvoření indexu nad geoprostorovými daty kolekce pitka a hranice.

A zde už je samotný dotaz, který hledá v kolekci pitka všechny dokumenty, jejichž pole TITLE obsahuje slovo ”Park”bez ohledu na velikost písmen, a pro každé nalezené pitko kontroluje, zda se nachází v městské části ”Brno-střed”dle kolekce hranice.

```
db.pitka_collection.find({
  "TITLE": {
    "$regex": ".*Park.*",
    "$options": "i"
  }
}).forEach(function(pitka) {
  var point = pitka.location;
  if (point) {
    var cast = db.hranice_collection.findOne({
      "location": {
        "$geoIntersects": {
          "$geometry": point
        }
      },
      "nazev": "Brno-střed"
    });

    if (cast) {
```

```
19          printjson(pitka);
20        }
21      }
22  });
```

Listing 8: Dotaz na vyhledání všech pítek v Brno-střed obsahující v názvu slovo Park

1.8.1 Odpověď severu:

MongoDB používá B-stromy (nebo jejich varianty) pro indexování dat. V případě textových dotazů (jako je regulární výraz v tomto dotazu) MongoDB používá textový index, pokud je vytvořen. V případě geoprostorových dotazů MongoDB používá geoprostorový index. Pokud nejsou tyto indexy dostupné, server musí provést kompletní prohledávání kolekce, což je mnohem méně efektivní. Výsledky jsou poté načteny z uzlů, kde jsou uloženy. Klient dostane BSON soubory výsledků v případě dotazu přímo na MongoDB serveru. V případě využití knihovny pymongo je výsledek doručen jako kurzor a je možné tímto kurzorem iterovat.

2 Databáze časových řad – InfluxDB

2.1 Zvolená datová sada

Pro řešení problému ukládání dat ve formátu časových řad jsme zvolili následující databázi a datovou sadu.

Typ NoSQL databáze	Databáze časových řad
Zvolená databáze	InfluxDB
Zvolená datová sada	Obsazenost parkovacích domů a parkovišť
Formát dat	CSV

Hlavním důvodem proč jsme pro následující datovou sadu zvolili InfluxDB je fakt, že datová sada obsahuje data, jenž jsou zaznamenávána v periodických intervalech pěti minut. Datová sada monitoruje počet volných parkovacích míst, pro jednotlivá parkovací místa v Brně.

Databáze bude obsahovat mnoho podobných záznamů z devíti parkovišť v Brně s různými časovými značkami. Toto uspořádání je ideální pro databázi InfluxDB, neboť se jedná o data ve formátu pravidelných časových řad, kde časové značky budou v pravidelných intervalech. To nám umožní využít delta kompresi a u po sobě jdoucích záznamů můžeme timestamp v nejlepším případě uložit až na jednom bitu. Zároveň hodnoty, jako počet volných míst, jsou v rozumných rozsazích, které se budou opakovat, například obsazenost parkoviště nikdy nepřesáhne 100%, což InfluxDB umí také efektivně zakódovat.

Hodnoty, které se opakují, například název parkoviště nebo jeho souřadnice, budou definovány jako tzv. `tag`. Tímto způsobem databázi efektivně rozčleníme na skupiny dat s identickými `tag values`, což nazýváme `tag set`, to nám umožní optimalizovat následné dotazování. Vhodným zvolením `tagů` rozdělíme data do skupin dat se společnými vlastnostmi a zároveň je připravíme pro budoucí dotazy. Jinými slovy, provádíme indexaci dat na základě opakujících se parametrů, což výrazně zvyšuje efektivitu vyhledávání v databázi. Ostatní parametry, které se tolik neopakují, budou ukládány jako tzv. `fields`. Datové typy, které InfluxDB nabízí, jsou pro tuto datovou sadu dostatečné a data budeme ukládat v rámci jednoho.

Databáze InfluxDB je optimalizována pro práci s časovými řadami, jejich vyhledávání a ukládání. Toto je zvláště důležité, když zvažujeme, že budeme ukládat přibližně 105 120 záznamů o volných parkovacích místech ročně. Navíc data nebudou v čase editována, což je další faktor, který potvrzuje vhodnost volby této databáze. [4]

InfluxDB poskytuje dotazovací jazyk, který je speciálně navržen pro práci s časovými řadami. Tímto způsobem můžete snadno provádět analýzu dat, vytvářet grafy a získávat informace o typickém chování jednotlivých parkovišť.

Dalším užitečná funkcionalita, kterou nám databáze poskytuje, je podpora zpracování událostí v reálném čase. InfluxDB nabízí možnost provádět zpracování událostí v reálném čase. To znamená, že můžete okamžitě reagovat na nová data a provádět různé akce na základě těchto dat. Tato schopnost je například využitelná, pokud chceme zobrazit aktuální dostupná parkovací místa na webovém portálu nebo informační ceduli.

InfluxDB představuje kombinaci přístupů CP (consistency and partition tolerance) a AP (availability and partition tolerance). CP se využívá pro metadata clusteru, zatímco AP přístup je využíván pro operace zápisu a čtení dat. Pro nás je pozitivní, že tato databáze vždy umožní zapsat a přečíst data. Vzácné nekonzistence se zde mohou vyskytnout, ale s ohledem na charakter našich dat obvykle nepředstavují zásadní problém. InfluxDB je také vysoce odolná a škálovatelná, což zajišťuje její spolehlivost. [5]

Ostatní typy databází se příliš nehodí na data ve formátů časových řad, ale vzhledem k tomu, že dat není extrémně mnoho a budou nás neustále zajímat stejné informace, tak bychom alternativně mohli použít databázi Apache Cassandra, která ale není určena přímo pro ukládání časových řad.

[4] Demonstrační cvičení z předmětu UPA, InfluxDB: `https://rychly-edu.gitlab.io/dbs/nosql/influxdb-lab/`
[5] Dokumentace InfluxDB: `https://docs.influxdata.com/influxdb/v1/`

2.2 Definice úložiště v InfluxDB

V následujících příkazech jsme vytvořili úložiště pro naši datovou sadu. Začali jsme tím, že jsme vytvořili databázi pomocí příkazu `CREATE DATABASE`. Dále jsme definovali strukturu tabulky prostřednictvím příkazu `INSERT`, který určuje, jak data budou organizována. Tímto příkazem jsme určili, které parametry budou použity jako `tags` a které budou pouze `fields`. Pro lepší přehlednost jsme také zkrátili některé hodnoty.

Jinými slovy, vytvořili jsme časový záznam, který obsahuje informace o počtu volných parkovacích míst pro různé typy uživatelů, včetně odběratelů s vyhrazenými místy a běžných uživatelů parkoviště. Tento záznam obsahuje také informace o adrese parkoviště, jeho kapacitě, souřadnicích a rozměrech.

```
1 CREATE DATABASE PARKOVACI_MISTA_V_BRNE;
2
3 INSERT volne_mista,name=husova,capacity=367,X=49.1,Y=16.6,Latitude=49.1,
4 Longitude=16.6,SSV=76,SAUV=160 SSU=23,SSO=108,cars=207,free=160
```

Listing 9: Vytvoření databáze PARKOVACI_MISTA_V_BRNE a vložení záznamu do databáze.

Jednotlivé zkratky z naší datové sady jsou vysvětleny v následující tabulce.

Zkratka	Pojem	Vysvětlení
SSV	Spaces Subscribers Vacant	Volné parkovací místo pro odběratele
SSO	Spaces Subscribers Occupied	Obsazené parkovací místo pro odběratele
SAUV	Spaces All Users Vacant	Volné parkovací místo pro všechny uživatele
SAUO	Spaces All Users Occupied	Obsazené parkovací místo pro všechny uživatele

Tabulka 2: Přehled zkratek použitých ve výpise 9.

2.3 Algoritmický popis importu dat

V prvním kroku algoritmu provedeme odstranění pole ObjectID z CSV souboru, protože v našem kontextu není potřebné pro identifikaci záznamu. Místo toho budeme používat časovou značku ve spojení s tagy pro identifikaci. Následně přejmenujeme některé parametry v CSV souboru, abychom získali kompaktnější verzi dat. Provádíme konverzi časových údajů do vhodného formátu, abychom zajistili, že data budou správně interpretována v databázi. Nakonec provedeme samotný import dat do databáze InfluxDB. Tímto způsobem budou data importována s přesností na sekundy, což umožní úsporu místa pomocí delta komprese.

```
1. Vybereme sloupce, které budeme používat
2. Přejmenujeme parametry na námi vybrané kompaktnější varianty
3. Konvertujeme čas do vhodného formátu
4. importujeme pomocí příkazu:
influx -import -path=data.txt -precision=s -database=PARKOVACI_MISTA_V_BRNE
```

2.4 Dotazy nad databází InfluxDB

První dotaz zjistí, kolik volných míst je průměrně denně na parkovišti P+R Líšeň u Zetoru.

```
SELECT MEAN("free") FROM "volne_mista"
WHERE "name"='p+r-lisen-u-zetoru'
GROUP BY time(24h);
```

Listing 10: Příklad dotazu v databázi InfluxDB, který zjistí, kolik volných míst je průměrně denně na parkovišti P+R Líšeň u Zetoru.

Druhý dotaz najde nejvíce volných míst na určitém parkovišti v určitém časovém období.

```
SELECT MAX(free) AS max_free, "name" FROM "volne_mista"
WHERE "name"='parkovaci-dum-domini-park-brno-husova-712-14a'
    AND time >= '2019-08-18T00:00:00Z'
    AND time <= '2019-09-08T00:30:00Z';
```

Listing 11: Dotaz v InfluxDB, který najde nejvíce volných míst na parkovišti Parkovací dům Domini Park Brno, Husova 712/14a v časovém období od 18. 8. 2019 do 8. 9. 2019.

Oba tyto dotazy využívají vyhledávání na základě tagů. InfluxDB efektivně seskupuje hodnoty se stejnými tagy. To znamená, že InfluxDB nejprve identifikuje všechna data s tagem name, který je definován v našem dotazu. Poté jsou samotné časové řady indexovány, což nám umožňuje efektivně lokalizovat data odpovídající odpovídající určitému časovém rozmezí.

3 Sloupcová databáze – Apache Cassandra

3.1 Zvolená datová sada

Pro demonstraci práce s databází Apache Cassandra byla zvolena následující datová sada:

Typ NoSQL databáze	Sloupcová wide-column
Zvolená databáze	Apache Cassandra
Zvolená datová sada	Cyklistické nehody
Formát dat	CSV

Při volbě datové sady jsme brali v úvahu především typické dotazy, které budeme v databázi provádět. Apache Cassandra nám umožňuje efektivně zpracovávat rychlé zápisy a čtení velkého objemu dat. To je v souladu s našimi požadavky, zejména pokud vezmeme v úvahu, že ne vždy potřebujeme veškerá data v odpovědi na dotaz. Například, pokud nás zajímá, kolik dopravních nehod bylo způsobeno alkoholem na určitém místě, nepotřebujeme načítat informace o hmotné škodě. Díky tomu, že můžeme data vhodně rozdělit do více sloupců, minimalizujeme načítání zbytečných informací a dosahujeme efektivního výkonu.

Ve výchozím stavu je Apache Cassandra databáze typu AP (Availability, Partition tolerance), ale můžeme docílit resp. nakonfigurovat konzistenci založenou na dotazech, které budou kladeny. Tím dosáhneme toho, aby databáze splňovala vlastnosti CA (Consistency, Availability). Vzhledem k tomu, že jsme schéma databáze navrhovali především podle dotazů, můžeme zajistit lepší konzistenci, což je obzvláště důležité pro tak citlivá data jako jsou informace o dopravních nehodách.

Nakonec, Apache Cassandra nám poskytuje flexibilitu v úpravě dat a struktury databáze. Pokud by v budoucnu byla potřeba přidat novou informaci, tedy změnit schéma databáze, můžeme jednoduše přidat nový sloupec do databáze, což je velmi užitečné pro naše potřeby, jelikož se může stát, že se v budoucnu začnou sledovat i další aspekty dopravních nehod, než pouze ty, které jsou v současné datové sadě zaznamenány.

Celkově jsme se rozhodli pro Apache Cassandra na základě těchto klíčových faktorů, které nám umožňují efektivně spravovat data, minimalizovat načítání zbytečných informací a zajišťovat vysoký výkon aplikace, zejména s ohledem na očekávané typické dotazy.[6]

Použití Neo4J v naší datové sadě nepředstavuje optimální volbu, neboť v našich datech nejsou zachycovány vztahy mezi daty. Ukládání těchto dat do grafového modelu by tedy bylo zbytečné a neproduktivní. Použití InfluxDB je opět nevhodné, jelikož InfluxDB je primárně určena pro zpracování pravidelných časových řad. Naše data, ačkoli mohou být interpretována jako časové události, nejsou pravidelné, a tím pádem by ztratila jednu z hlavních výhod InfluxDB - kompresi časových událostí. Dalším faktorem, který činí InfluxDB nevhodnou volbou, je obtížnost rozšiřování dat o nové sloupce a nutnost načítat celý řádek pro přístup k datům. MongoDB by také mohla být vhodnou databází pro tuto datovou sadu, ale v případě Apache Cassandra není vždy nutné načítat celý řádek pro zodpovězení dotazu. Načítání celého řádku by bylo pro mnoho dotazů týkající se zvolené datové sady neefektivní.

3.2 Definice úložiště v Apache Cassandra

Pro definici vlastní databáze v Apache Cassandra je nejdříve potřeba vytvořit tzv. prostor klíčů (keyspace). Následujícími příkazy vytvoříme v CQL Shell for Apache Cassandra (cqlsh) prostor klíčů nazvaný nehody, přičemž můžeme specifikovat replikační faktor, tedy počet kopií jednotlivých „partitions“. Tyto kopie jsou pak uloženy v jednotlivých uzlech („nodes“) klastru. Pomocí příkazu `USE` zajistíme, že budeme následně nově vytvořený prostor klíčů nehody používat.[7]

[6] Dokumentace Apache Cassandra: `https://cassandra.apache.org/_/index.html`

[7] Demonstrační cvičení z UPA, Apache Cassandra: `https://rychly-edu.gitlab.io/dbs/nosql/cassandra-lab/`

```
CREATE KEYSPACE IF NOT EXISTS nehody WITH replication = {'class':
    'SimpleStrategy', 'replication_factor' : 3};
USE nehody;
```

Listing 12: Vytvoření prostoru klíčů v Apache Cassandra.

Dalším krokem je definice struktury tabulky, která se provádí podobně jako v klasické relační databázi příkazem `CREATE TABLE`. Pro jednotlivé sloupce definujeme jejich datové typy a následně určíme primární klíč. Primární klíč je v Apache Cassandra tvořen dvěma částmi: „partition“ a „clustering“ klíč. „Partition key“ rozděluje data v tabulce mezi uzly a „clustering key“ data v každé části tabulky následně uspořádává. Tyto klíče je potřeba zvolit tak, aby byla data vhodně rozdistribuována po celém klastru. Zároveň se při dotazech snažíme číst z co nejmenšího množství uzlů, takže je vhodné volit klíče na základě budoucích dotazů, které předpokládáme, že se budou týkat především druhů srážek a lokací, ve kterých nastaly (městské části Brna). Proto byla za „partition“ klíč zvolena dvojice (srazka, nazev), kde srážka je typ srážky a nazev je název městské části. Za „clustering“ klíč bylo zvoleno uuid. Původně bylo uvažováno nad volbou sloupce datum jako „clustering“ klíče, protože se předpokládalo, že uspořádávání dat podle data by mohlo být pro dotazy užitečné. Nicméně po prohlédnutí záznamů v datové sadě jsme zjistili, že z data se ve zvolené datové sadě vždy ukládá pouze den, měsíc a rok, a ve stejné datum mohlo ve stejné čtvrti nastat více nehod se stejným typem srážky. Proto bylo místo datumu zvoleno unikátní UUID za poslední část primárního klíče, která zajistí, že nebudou již existující záznamy přepisovány (pokud to samozřejmě není záměr při aktualizaci dat). Pomocí následujícího příkazu bude vytvořena tabulka s názvem: cyklisticke_nehody se zvoleným primárním klíčem: ((srazka, nazev), uuid).

```
CREATE TABLE IF NOT EXISTS cyklisticke_nehody (
    uuid UUID,
    datum TIMESTAMP,
    srazka TEXT,
    pricina TEXT,
    alkohol TEXT,
    zavineni TEXT,
    nasledky TEXT,
    stav_vozovky TEXT,
    povetrnostni_podm TEXT,
    viditelnost TEXT,
    rozhled TEXT,
    misto_nehody TEXT,
    druh_komun TEXT,
    druh_vozidla TEXT,
    stav_ridic TEXT,
    ovlivneni_ridice TEXT,
    osoba TEXT,
    ozn_osoba TEXT,
    pohlavi TEXT,
    nasledek TEXT,
    usmrceno_os INT,
    tezce_zran_os INT,
    lehce_zran_os INT,
    hmotna_skoda INT,
    cas INT,
    hodina INT,
    vek_skupina TEXT,
    nazev TEXT,
    PRIMARY KEY (
      (srazka, nazev),
      uuid
    )
```

```
34 );
```

Listing 13: Definice schématu tabulky `cyklisticke_nehody` v Apache Cassandra.

3.3 Algoritmický popis importu dat

Po připravení databáze pro data (tedy vytvoření prostoru klíčů a schéma databáze, v našem případě tabulky s názvem cyklisticke_nehody) nastává fáze importu dat. Data je potřeba před samotným importem upravit do požadované podoby, a to ideálně pomocí vhodného skriptu[8] Vzhledem k tomu, že z původní tabulky nutně nepotřebujeme všechny sloupce původní tabulky (především sloupce irelevantní k cyklistickým nehodám, které jsou obsaženy v každé datové sadě stáhnuté z webových stránek Statutárního města Brna, a to konkrétně sloupce: X, Y, objectid, join_count, target_fid, id, d, e, point_x, point_y, globalid, dále pak sloupce, které by byly duplicitní nebo nepotřebné: den, rok, mesic, mesic_t, den_v_tydnu), je potřeba tyto sloupce z datové sady odstranit. Následně je potřeba vytvořit sloupec uuid s unikátním identifikátoru jednotlivých záznamů. Dále je také potřeba převést původní formát sloupce datum z formátu "%Y/%m/%d %H:%M:%S+%f"na formát podporovaný Apache Cassandra, tedy "%Y-%m-%d"(informaci o přesném času si můžeme dovolit ze záznamů odstranit, protože v původní datové sadě je čas vždy nastaven na hodnotu 00:00:00+00). Po transformaci původních dat na vyhovující formát již můžeme upravená data, uložená například v souboru data.csv, importovat do databáze. Vkládání dat do databáze je možné buď pomocí příkazu `INSERT INTO`, podobně jako v SQL, ale vzhledem k tomu, že chceme data importovat z CSV souboru, zvolili jsme druhou variantu importu dat, a to příkaz `COPY`. Tento příkaz nám zajistí nejen import nových dat do databáze, ale zároveň je pomocí něj možné data i aktualizovat (záznamy pro aktualizaci jsou vybrány pomocí primárního klíče, pokud již v databázi existuje). Při počátečním naplňování databáze daty budou tedy všechna data z CSV souboru vložena do databáze jako data nová. Po aktualizaci tabulky data.csv a provedení stejného příkazu by byla i data v databázi aktualizována, případně přidána nová.

Popis přípravy dat a jejich importu

1. Stažení datové sady ve formátu CSV.
2. Odstranění nepotřebných sloupců v datové sadě (sloupce: X, Y, objectid, join_count, target_fid, id, d, e, den, rok, mesic, mesic_t, den_v_tydnu, point_x, point_y, globalid)
3. Vytvoření sloupce uuid pro zajištění unikátnosti záznamů (např. s použitím python balíčku uuid a funkce uuid4()).
4. Transformace sloupce datum do formátu YY-MM-DD (např. pomocí python balíčku datetime a funkcí strptime() a strftime()).
5. Import dat do databáze z upraveného souboru data.csv do databáze pomocí příkazu `COPY`.

```
1 COPY cyklisticke_nehody (uuid, datum, srazka, pricina, alkohol, zavineni,
      nasledky, stav_vozovky, povetrnostni_podm, viditelnost, rozhled,
      misto_nehody, druh_komun, druh_vozidla, stav_ridic, ovlivneni_ridice,
      osoba, ozn_osoba, pohlavi, nasledek, usmrceno_os, tezce_zran_os,
      lehce_zran_os, hmotna_skoda, cas, hodina, vek_skupina, nazev)
2 FROM 'data.csv'
3 WITH HEADER = true AND NULL = 'unknown';
```

Listing 14: Iport dat do připravené databáze s použitím příkazu `COPY`.

[8] Ukázka skriptu, který je možné použít pro úpravu dat před importem do databáze, je přiložena v souboru `scripts/apache-cassandra_prepare.py`.

3.4 Dotazy nad databází Apache Cassandra

Dotazování se v Apache Cassandra provádí podobně jako v SQL pomocí příkazu `SELECT`. Následující dotaz vypíše počet lehce zraněných, těžce zraněných a usmrcených osob u nehod typu havárie, ke kterým došlo v městské části Brno-střed.

```
SELECT  nazev,
        SUM(lehce_zran_os) AS "lehce zranene osoby",
        SUM(tezce_zran_os) AS "tezce zranenene osoby",
        SUM(usmrceno_os) AS "usmrcene osoby"
FROM cyklisticke_nehody
WHERE srazka = 'havárie'
    AND nazev = 'Brno-střed';
```

Listing 15: Dotaz v jazyce Cassandra Query Language (CQL), který vypíše počet lehce zraněných, těžce zraněných a usmrcených osob u nehod typu havárie, ke kterým došlo v městské části Brno-střed.

Při zpracování výše uvedeného dotazu bude databázový server postupovat následovně. Vzhledem k tomu, že dvojice (srazka, nazev) je zvolena jako „partitioning“ klíč, záznamy se stejnou dvojicí jsou uloženy na stejném „partition“. Díky tomu, že v příkazu `WHERE` specifikujeme konkrétní hodnotu této dvojice, není potřeba procházet více „partitions“ a tedy filtrovat data, což by bylo při velkém objemu dat náročné. Poté Apache Cassandra provede agregace nad zbylými sloupci vybraných záznamů, a tedy provede součet hodnot ve sloupcích lehce_zran_os, tezce_zran_os a usmrceno_os a výsledky sum vrátí jako sloupce výsledné odpovědi na dotaz, spolu s názvem městské části Brno-střed, jak je zobrazeno níže.

```
 nazev      | lehce zranene osoby | tezce zranenene osoby | usmrcene osoby
------------+---------------------+-----------------------+----------------
 Brno-střed |                 101 |                     9 |              0
```

Listing 16: Odpověď databázového serveru na dotaz z výpisu 15.

Vzhledem k tomu, že data do databáze importujeme z CSV souborů, předpokládáme, že export tabulek a výsledků dotazů by také mohlo být vhodné ukádat do CSV souborů. Pro uložení celé tabulky cyklisticke_nehody do CSV souboru data_export.csv nebo na standardní výstup je možné provést opět s použitím příkazu COPY, a to následujícím způsobem:

```
-- export do souboru data_export.csv
COPY cyklisticke_nehody TO 'data_export.csv' WITH HEADER = TRUE;
```

Listing 17: Export tabulky cyklisticke_nehody do souboru data_export.csv.

Pokud chceme do CSV souboru exportovat výsledky dotazů, je to možné například s použitím příkazu `CAPTURE` s upřesněním souboru, do kterého chceme dané výsledky dotazů ukládat, v našem případě je to soubor data_capture.csv. Výsledky dotazů se do tohoto souboru budou vkládat vždy nakonec, takže při provedení více dotazů je možné tyto výsledky uložit do jednoho CSV souboru. Pokud bychom chtěli výsledek každého dotazu vložit do jiného souboru, lze jednoduše opět pomocí příkazu `CAPTURE` specifikovat jiný výstupní soubor a až poté provést další dotazy. Klient tedy může výsledky dotazů, případně celé tabulky následně zpracovávat z těchto uložených CSV souborů, případně ze standardního výstupu, pokud použijeme příkaz `CAPTURE` bez dalšího parametru.

```
CAPTURE 'data_capture.csv'
-- následně je možné provést jakýkoliv dotaz, jehož výsledek bude uložen do
   souboru data_capture.csv
```

Listing 18: Zaznamenání výsledku dotazu do souboru data_capture.csv pomocí příkazu `CAPTURE`.

4 Grafová databáze – Neo4J

Grafová databáze Neo4J byla vybrána pro uložení datové sady BRNO BRZO: Stavební projekty a záměry. Tato datová sada obsahuje záznamy o stavebních projektech na území města Brna. Obsahuje informace o jednotlivých projektech, jejich investorech, architektech, umístění a dalších detailech.

Typ NoSQL databáze	Grafová databáze
Zvolená databáze	Neo4J
Zvolená datová sada	BRNO BRZO: Stavební projekty a záměry
Formát dat	CSV

Datová sada obsahuje strukturovaná data a především vztahy mezi nimi, na které je vhodné použít grafový typ databáze, protože ten umožňuje snadný způsob modelování a dotazování na vztahy mezi jednotlivými záznamy. Jednotlivé projekty, architekty, investory a městské části je vhodné ukládat jako uzly a pomocí vztahů mezi uzly modelovat vztahy mezi těmito entitami, například vztah, že projekt byl sponzorován určitým investorem nebo že daný projekt je lokalizován v určité městské části. Podrobné informace o jednotlivých uzlech je vhodné modelovat jako atributy těchto uzlů. Neo4J umožňuje flexibilitu při úpravě schématu databáze a přidávání nových atributů k uzlům v grafu, což může být užitečné, pokud by v budoucnu vznikla potřeba zaznamenávat i další informace o projektech, architektech nebo investorech.

Jednou z hlavních nevýhod Neo4J je horní omezení ohledně velikosti grafu, který je možné v databázi uchovat. Databáze ale dokáže uchovávat desítky miliard uzlů, atributů a vztahů v jednom grafu a vzhledem k tomu, že vybraná datové sada obsahuje pouze o informace týkající se stavebních projektů v Brně, architektů a investorů, neočekáváme, že by tato datová sada obsahovala množství dat přesahující kapacitu databáze a toto omezení by tedy nemělo v tomto případě představovat problém. Další nevýhodou Neo4J je, že na datové úrovní nejsou samotná data šifrována, což by ale nemělo v našem případě vadit, protože datová sada obsahuje veřejně dostupné informace.

Co se týče CAP teorému, Neo4J je databáze typu CP. Upřednostňuje tedy konsistenci a „partition tolerance“ před dostupností, což znamená, že v případě výpadku sítě obětuje dostupnost kvůli zachování konzistence dat. Vzhledem k tomu, že vybraná datová sada neobsahuje kritická data, je případná nedostupnost přijatelnější než porušení konzistence dat. Neo4J ve výchozím režimu běží na jediném serveru, nicméně je možné vytvořit i cluster, díky čemuž může být databáze distribuována na více serverech. V tomto režimu jsou data rozdělena mezi uzly (nodes), což umožňuje horizontální škálování a zvyšuje dostupnost dat. Každý uzel v clusteru obsahuje část grafu a zpracovává část dotazů. V cluster režimu může být definována i replikace dat mezi uzly, což opět přináší větší dostupnost a odolnost proti výpadkům, kdy při výpadku jednoho uzlu může systém stále fungovat na základě replikovaných dat.[9]

Další databázové systémy, jako je Apache Cassandra, MongoDB a InfluxDB, nejsou pro uložení zvolené datové sady tak vhodné, jako Neo4J. Apache Cassandra i MongoDB jsou sice určeny pro distribuované ukládání dat, ale nejsou vhodné pro modelování vztahů mezi daty. Použití InfluxDB pro uchování dat z vybrané datové sady také není vhodné, protože je to časová databáze určená pro časové řady dat. Časové informace se v této datové sadě sice vyskytují, nicméně jsou to pouze informace o poslední editaci záznamů, které zaprvé nejsou časovou řadou, a zadruhé ani není potřeba tuto informaci pro zvolenou datovou sadu uchovávat.

4.1 Definice úložiště v Neo4J

Uzly v Neo4J reprezentují entity, vztahy určují spojení mezi jednotlivými entitami a atributy obsahují informace o těchto entitách.

[9] Dokumentace Neo4J: `https://neo4j.com/docs/`

Pro definici schématu databáze v jazyce Cypher Query Language (CQL) je možné použít klauzule `CREATE`, které definují typy jednotlivých uzlů a vztahů. Definice schématu databáze pro zvolenou datovou sadu by pak mohla být následující:

```
// Uzly
CREATE (p:Project)
CREATE (i:Investor)
CREATE (a:Architect)
CREATE (d:District)
// Vztahy
CREATE (:Project)-[:FINANCED_BY]->(:Investor)
CREATE (:Project)-[:DESIGNED_BY]->(:Architect)
CREATE (:Project)-[:LOCATED_IN]->(:District)
;
```

Listing 19: Definice uzlů `Project`, `Investor`, `Architect` a `District` a vztahů `FINANCED_BY` mezi projektem a investorem, `DESIGNED_BY` mezi projektem a architektem a `LOCATED_IN` mezi městskou částí a projektem v databázi Neo4J.

Definici samotného schématu databáze není nutné provádět předem a je možné ji spojit s vlastním importem prvních dat, jak bude popsáno v následující podkapitole.

4.2 Import dat do Neo4J

Pro import datové sady, která je k dispozici ve formátu CSV, je možné využít příkazu `LOAD CSV`. V případě, že nechceme z CSV souboru ukládat do databáze všechny sloupce, není potřeba jej předem upravovat skriptem. Sloupce z CSV souboru spárujeme s jednotlivými atributy a uzly databáze tak, že jeden záznam v CSV souboru je při importu nejdříve vždy načten do proměnné `row`, jejíž atributy (sloupce CSV souboru) následně specifikujeme při ukládání do potřebného atributu nebo uzlu. K vytvoření jednotlivých uzlů a vztahů mezi nimi je použit příkaz `MERGE`, který zajistí, že pokud daný uzel nebo vztah v databázi již existuje, nebude přidán záznam duplicitní. Data z datové sady budou do databáze importována následujícím příkazem.

```
LOAD CSV WITH HEADERS
FROM "https://data.brno.cz/datasets/mestobrno::brno-brzo-stavebn%C3%AD-
    projekty-a-z%C3%A1m%C4%9Bry-brno-brzo-development-projects-and-plans.csv"
AS row
MERGE (p:Project {name: COALESCE(row.nazev_projektu, "unknown_name")})
ON CREATE SET
  p.address = row.adresa,
  p.web = row.web_projektu,
  p.status = row.stav,
  p.investment_type = row.typ_investice,
  p.notes = row.poznamky,
  p.project_type = row.typ_projektu,
  p.act = row.aktuality,
  p.desc = row.popis
MERGE (i:Investor {name: COALESCE(row.developer_investor,
    "unknown_investor")})
MERGE (d:District {name: row.mestska_cast})
MERGE (a:Architect {name: COALESCE(row.architekt, "unknown_architect")})

MERGE (p)-[:FINANCED_BY]->(i)
MERGE (p)-[:DESIGNED_BY]->(a)
```

```
21 MERGE (p)-[:LOCATED_IN]->(d);
```

Listing 20: Příkaz použitý k importu dat z CSV souboru specifikovaného URL adresou do Neo4J databáze.

Při použití příkazu `MERGE` je možné definovat, jaká hodnota má být do daného atributu nastavena v případě přidávání nového uzlu nebo vztahu, a jaká hodnota má být nastavena v případě, že daný uzel nebo vztah již existuje. Například, pokud bychom chtěli vytvořit/aktualizovat uzel projektu s názvem "Parkovací dům Purkyňova", můžeme pomocí `ON CREATE` klauzule specifikovat hodnotu atributu status, která bude při vytvoření nového uzlu nastavena na "plánovaný", ale pokud by již uzel s daným jménem existoval, chceme jeho status aktualizovat na hodnotu "probíhající"pomocí klauzule `ON MATCH`. To zajistíme následujícím příkazem v CQL:

```
1 MERGE (p:Project {name: "Parkovací dům Purkyňova"})
2 ON CREATE SET p.status = "plánovaný"
3 ON MATCH SET p.status = "probíhající"
4 ;
```

Listing 21: CQL příkaz, který při přidání nového projektu s názvem „Parkovací dům Purkyňova“ nastaví jeho status na hodnotu „plánovaný“, ale pokud již projekt s daným názvem existuje, jeho status bude aktualizován na hodnotu „probíhající“.

Přidání dalších dat do databáze z aktualizovaného CSV souboru je možné opět s použitím příkazu uvedeném ve výpisu 20. Pokud by bylo potřeba přidat například pouze jeden záznam, je možné využít následujícího příkazu 22. Ten přidá nový projekt, investora, městskou část a architekta, pokud již neexistují, a vytvoří všechny potřebné vztahy.

```
1  // Vytvoření projektu
2  MERGE (p:Project {
3    name: "Nová budova FIT",
4    address: "Božetěchova",
5    web: "www.jeste-lepsi.fit.com",
6    status: "planovany",
7    investment_type: "verejna",
8    notes: "V areálu fakulty FIT VUT bude vystavěna nová budova.",
9    project_type: "školství",
10   act: "",
11   desc: "V areálu fakulty FIT VUT bude vystavěna nová budova, do které bude
       vést most z budovy D přes silnici. V této budově bude přednášková
       místnost s kapacitou až 800 osob."
12 })
13 // Vytvoření investora
14 MERGE (i:Investor {name: "Statutární město Brno"})
15 // Vytvoření městské části
16 MERGE (d:District {name: "brno_kralovo_pole"})
17 // Vytvoření architekta
18 MERGE (a:Architect {name: "Atelier 99 s.r.o."})
19 // Vytvoření vztahů
20 CREATE (p)-[:FINANCED_BY]->(i)
21 CREATE (p)-[:DESIGNED_BY]->(a)
22 CREATE (p)-[:LOCATED_IN]->(d)
```

Listing 22: Příkaz zajišťující vytvoření nového projektu, architekta, investora a městské části, pokud tyto již neexistují, a vytvoření potřebných vztahů mezi nimi.

4.3 Dotazy nad databází Neo4J

Dotazy se v Neo4J opět provádějí pomocí jazyka Cypher Query Language (CQL). Klauzule `MATCH` se používá pro definici vzoru grafu, který chceme vyhledat, je obdobou příkazu `SELECT` z jazyka SQL a slouží k identifikaci uzlů, vztahů a jejich vzájemných vazeb. Následující dotaz vypíše počet a seznam plánovaných projektů sponzorovaných statutárním městem Brnem v jednotlivých brněnských městských částech.

```
1 MATCH (i:Investor)<-[:FINANCED_BY]-(p:Project)-[:LOCATED_IN]->(d:District)
2 WHERE i.name = "Statutární město Brno"
3 AND   p.status = "planovany"
4 RETURN
5   d.name AS `Městská část`,
6   COUNT(p) AS `Počet plánovaných projektů`,
7   COLLECT(p.name) AS `Seznam plánovaných projektů`
8 ORDER BY COUNT(p) DESC;
```

Listing 23: Příklad dotazu v Neo4J databázi. Tento dotaz vypíše počet a seznam plánovaných projektů sponzorovaných statutárním městem Brnem v jednotlivých brněnských městských částech.

Neo4J po získání dotazu vytvoří exekuční plán, který je možné vypsat s použitím příkazu `EXPLAIN`. Výsledek tohoto příkazu a tedy detail toho, jakým způsobem je dotaz zpracován v databázi, je zobrazen ve výpisu 24. Tento plán se čte směrem zespoda nahoru a začíná tedy operátorem NodeByLabelScan. Tento krok provádí vyhledávání investorů na základě jejich labelu (typu) Investor. Následně se pomocí operátoru Filter, který odpovídá klauzuli `WHERE` v dotazu, vyberou pouze ti investoři, jejichž název odpovídá názvu "Statutární mětso Brno". Operátor Execute zahrnuje procházení vztahů v grafu. Graf prochází všechny vztahy FINANCED_BY mezi vybranými investory a projekty. Dále jsou opět vykonávány operátory Filter a Execute, které vyberou projekty se stavem "plánovaný"a umístěné v jednotlivých městských částech. EagerAggregation zahrnuje agregační funkci pro spočítání počtu plánovaných projektů v jednotlivých částech a operátor Sort zajístí setřízení výsledků podle počtu plánovaných projektů sestupně. Tento operátor odpovídá klauzuli `ORDER BY COUNT(p) DESC` v původním dotazu. Provedení operátoru ProduceResults je konečným krokem exekučního plánu, ve kterém je výsledek dotazu vrácen klientovi.

Výsledky dotazů mohou být vráceny klientovi různými způsoby. Při práci přímo v Cypher Shell jsou výstupy vypisovány zpět přímo v konzoli, nicméně Neo4J poskytuje také HTTP REST API, které umožňuje klientovi jak posílat dotazy a importovat data do databáze HTTP požadavkem, tak poskytuje výsledky dotazů ve formě JSON odpovědi, které může klient dále zpracovávat podle potřeby dané aplikace.

```
+-----------------+----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| Operator        | Estimated Rows | Identifiers                                                     | Ordered by                    | Other                             |
+-----------------+----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +ProduceResults |              2 | Městská část, Počet plánovaných projektů, Seznam plánovaných projektů | Počet plánovaných projektů DESC |                                   |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +Sort           |              2 | Městská část, Počet plánovaných projektů, Seznam plánovaných projektů | Počet plánovaných projektů DESC | Počet plánovaných projektů        |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +EagerAggregation |            2 | Městská část, Počet plánovaných projektů, Seznam plánovaných projektů |                               | Městská část                      |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +Filter         |              4 | i, anon[19], anon[47], p, d                                     |                               | d:District                        |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +Expand(All)    |              4 | i, anon[19], anon[47], p, d                                     |                               | (p)-[anon[47]:LOCATED_IN]->(d)    |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +Filter         |              4 | anon[19], i, p                                                  |                               | p.status = $`  AUTOSTRING1`       |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +Expand(All)    |             36 | anon[19], i, p                                                  |                               | (i)<-[anon[19]:FINANCED_BY]-(p)   |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +Filter         |             18 | i                                                               |                               | i.name = $`  AUTOSTRING0`         |
| |               +----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
| +NodeByLabelScan |           183 | i                                                               |                               | :Investor                         |
+-----------------+----------------+-----------------------------------------------------------------+-------------------------------+-----------------------------------+
```

Listing 24: Výsledek příkazu `EXPLAIN` pro dotaz z výpisu 23 v `cypher-shell`.