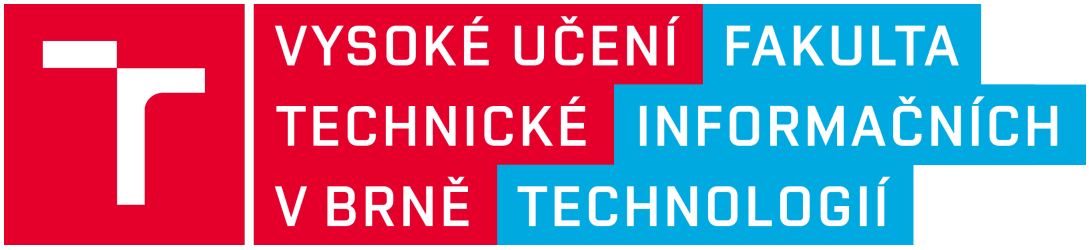


Projekt, 1. Část - Ukládání rozsáhlých dat v NoSQL databázích

Matěj Žalmánek, Jan Zimola, Eva Mičánková

28. října 2023

Obsah

1 Cíl projektu

Cílem této části projektu je analyzovat požadavky a navrhnout optimální způsob uložení rozsáhlých dat do vhodné NoSQL databáze tak, aby tato data bylo možno rychle dotazovat a aktualizovat.

1.1 Zadání

Řešení této části se skládá několika kroků:

1. Na základě přednášek a cvičení z předmětu prostudujte vlastnosti a možnosti použití NoSQL databází různých druhů, konkrétně:
   - sloupcová wide-column databáze (např. Apache Cassandra)
   - dokumentová databáze (např. MongoDB)
   - grafová databáze (např. Neo4J)
   - databáze časových řad (např. InfluxDB)
2. Prostudujte dostupné datové sady Národním katalogu otevřených dat pro statutární město Brno, zejména se soustřeďte na dílčí datové sady z daného zdroje, jejich strukturu (schéma), typy datových položek, identifikátory, možnosti propojení datových sad (společné entity) či napojení na externí data (jiné zdroje, entity reálného světa), změnu dat v čase (aktualizace), a jiné.
3. Pro každý z výše uvedených druhů NoSQL databází mezi výše odkazovanými datovými sadami najděte datovou sadu, kterou bude vhodné uložit a dotazovat v daném druhu NoSQL databáze (a ne v jiném). Celkem použijete tedy nejméně 4 různé datové sady pro 4 různé typy NoSQL databází. Konkrétně, pro každý jednotlivý případě, popište:
   - plný název zvolené datové sady vč. odkazu URL ve výše uvedeném katalogu
   - vhodnou distribuci dané datové sady (např. CSV či GeoJSON) dle nabídky v katalogu
   - zvolený druh NoSQL databáze a konkrétní databázový produkt/server (např. Apache Cassandra)
   - podrobné a logické vysvětlení, proč je nejlepší k uložení a dotazování zvolené datové sady použít právě daný druh NoSQL databáze (oproti jiným druhům NoSQL databází) - vysvětlení musí být konkrétní pro danou datovou sadu a databázi a odkazovat se na konkrétní jejich charakteristické vlastnosti, např. sloupce (struktura/schéma, datové typy, velikosti domény, atp.), řádky (počet, velikost, různorodost, existence identifikátorů a referencí, atp.), způsob a průběh vzniku dat a jejich uložení (zdroj, frekvence a perioda aktualizace, důvěryhodnost/chybovost dat, možnost zpětné změny již publikovaných dat či jen přidávání nových, možnosti komprese a agregace, retence/zapomínání dat, distribuce a škálovatelnost úložiště, redundance, atp.), způsob a průběh spotřeby dat (předdefinované a ad-hoc dotazy, frekvence a perioda čtení výsledků, distribuce a škálovatelnost zpracování dat pro dotazy, pozice konzumentů výsledků vzhledem k místu uložení a zpracování dat, možnosti urychlení dotazů pomocí cache/před-počítání, možnosti indexace, atp.)
   - syntakticky i sémanticky korektní příkazy pro definici úložiště v daném produktu/serveru NoSQL databáze pro danou datovou sadu (zápis definice schéma v CQL, příklad vložení dokumentu v JavaScript, import uzlů a hran v Cypher, atp., dle zvoleného databázového serveru)
   - algoritmický popis importu dat ze zvolené distribuce dané datové sady do připravené databáze a to jak počáteční naplnění prázdné databáze daty, tak pozdější doplnění nových či změněných dat (soustřeďte se na zvolení a použití vhodného klíče záznamů v NoSQL

tak, aby bylo podle něj možné provést UPSERT, namísto INSERTu duplicitního záznamu; smazání všech dat v databázi a jejich opětovné vložení není přípustné) - můžete odevzdat krátký skript (např. v jazyce Python) nebo uvedené popsat pseudokódem či popisem kroků zvoleného algoritmu

- alespoň jeden syntakticky i sémanticky korektní dotaz v jazyce daného databázového produktu nad v databázi uloženými daty dané datové sady, který bude demonstrovat vhodnost zvoleného druhu NoSQL databáze pro daná data vč. popisu způsobu, jakým databázový server dotaz zodpoví (jak nalezne uzly, kde jsou uložena požadovaná data; jak data z uzlů získá a dále distribuovaným způsobem zpracuje; jak výsledky doručí klientovi, který zadal dotaz, a jak je tento zkonzumuje)

4. Zkontrolujte a v týmu diskutujte vhodnost zvolené datové sady, správnost odůvodnění zvoleného druhu NoSQL databáze, použitelnost a náročnost popsaného způsobu načítání dat do databáze a provádění dotazů v databázi a korektnost všech příkazů či dotazů (zde je nezbytné si příkazy a dotazy nad datovými sadami a databázovými produkty prakticky vyzkoušet, a tím ověřit jejich korektnost).

5. Požadované výsledky sepište strukturovanou formou do dokumentu, případně doplňte skripty či ukázkami v souborech z dokumentu odkazovaných, zabalte do ZIP archivu a odevzdejte do IS VUT.

2 InfluxDB

2.1 Informace o datové sadě

- Použitá datová sada: Kvalita ovzduší
- Distribuce: GeoJSON

Data o kvalitě ovzduší ze stanic na území města Brna. Nejedná se o verifikovaná data. Stanice jsou ve správě ČHMÚ, města Brna a Státního zdravotnického ústavu Ostrava.

Aktualizace

- SO2_1h (každou hodinu/updated hourly)
- NO2_1h (každou hodinu/updated hourly)
- CO_8h (každých 8h/each 8 hours)
- PM10_1h (každou hodinu/updated hourly)
- O3_1h (každou hodinu/updated hourly)
- PM10_24h (24h průměr/daily average)
- PM2_5_1h (každou hodinu/updated hourly)

2.2 Odůvodnění vhodnosti zvoleného datasetu pro daný typ databáze

Charakteristické vlastnosti

Data ve zvolené datové sadě obsahují časový aspekt, což je vhodný typ dat pro InfluxDB. InfluxDB je specializována na práci s časově provázanými daty, což usnadňuje jejich analýzu a škálovatelnost a efektivní uložení. Dále InfluxDB obsahuje nástroje pro agregaci dat v různých časových rozmezích, což umožňuje snadné vytváření grafů a generování statistik. Důležité je také to, že si u dat můžeme nadefinovat retenci a tak uchovávat pouze jejich potřebné množství (shared group). Zásadní pro použití InfluxDB je také to, že data mají stále schéma.

Atribut	Typ
code	string
name	string
owner	string
lat	float(2)
lon	float(2)
actualized	Date
so2_1h	string
no2_1h	string
co_8h	string
pm10_1h	float(2)
o3_1h	string
pm10_24h	float(2)
pm2_5_1h	string

Tabulka 1: Kvalita Ovzduší - Schéma

Uložení dat

Zdrojem dat jsou měření jsou sdílena Českým hydrometeorologickým úřadem, městem Brno a Ostravským zdravotnickým ústavem. Data se aktualizují v intervalech závislých na měřené latce, proto jsou zřídka všechny hodnoty uvedeny v jednom záznamu. Ale můžeme zhruba říct, že se aktualizují skoro každou hodinu. Data pocházejí z 9 měřících stanic.

Způsob a průběh spotřeby dat

- **Dotazy na Data:** Konzumenti dat mohou provádět jak předdefinované dotazy[1], které jsou vytvořeny předem, tak ad-hoc dotazy, které jsou vytvářeny na základě aktuální potřeby. Frekvence a perioda čtení výsledků závisí na specifických požadavcích a potřebách aplikace, ale můžeme předpokládat, že se bude jednat hlavně o dva typy:
  - Analýza dat, která může být dělána jednou za několik měsíců
  - Hlídání hodnot v skoro realném čase, aby se mohlo reagovat na nebezpečné či podezřelé hodnoty
- **Distribuce a škálovatelnost:** Postupem času množství dat bude růst a jejich využití se může zvyšovat vzhledem ke zvýšenému zaměření na životní prostředí. Databáze tak může čelit zvýšenému počtu dotazů. Proto je důležité, že InfluxDB je horizontálně škálovatelná. Pro zmenšení latence bude vhodné zvolit datové centrum v Evropě, nejlépe v Brně, jelikož lze předpokládat, že konzumenti budou hlavně osoby či organizace v Brně.
- **Optimalizace výkonu:** Pro urychlení dotazů mohou být v InfluxDB vytvořeny indexy na často používané atributy, kterým se v InfluxDB říká **tagy**. Indexace zrychluje dotazování, řazení, agregaci dat i v kombinaci s agregováním pro časové intervaly.

2.3 Příkazy pro definici úložiště

Pro účely projektu jsme použili cloudové úložiště InfluxDB. Pro práci s InfluxDB jsme použili knihovnu `influxdb-client3`. Pro definici úložiště a import dat byl použit skript `influxdb.py`. Příklad připojení k NoSQL databázi InfluxDB:

```
from influxdb_client_3 import InfluxDBClient3, Point
# Token ziskany z InfluxDB
token = "---token---"

org = "org"
database = "jmeno databaze"

host = "host address"

# Vytvoreni klienta
client = InfluxDBClient3(host=host, token=token, org=org)

# Ukazkova data
zaznam = {
        # Neco jako tabulka v sql databazy
        'measurement': 'merici tabulka',
        'tags': {
            'nazek': 'textova hodnota',
        },
        'time': 'cas udalosti',
        'fields': {
```

[1] https://docs.influxdata.com/influxdb/v2/upgrade/v1-to-v2/migrate-cqs/#Copyright

```
                'nazev': hodnota
            },
    }
    # Zapis Dat
    client.write(database=database, record=zaznam)
```

2.4 Algoritmický popis importu dat

Pro import dat byl použit skript `influxdb.py`. Hlavní algoritmus importu dat:

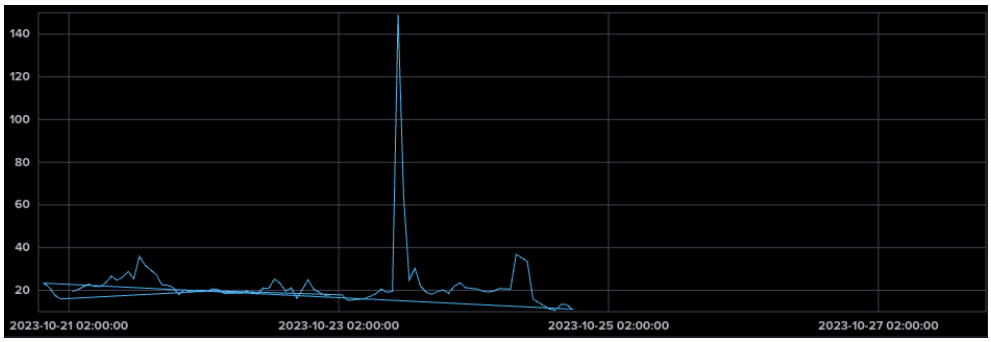
```
    for feature in features:
    properties = feature.get('properties', {})

    if properties:
        actualized_str = properties.get('actualized', '')
        try:
            no2_1h = properties.get('no2_1h')
            no2_1h_value = float(no2_1h) if no2_1h is not None else None
            actualized_date = datetime.fromisoformat(actualized_str[:-1])
            actualized_date = actualized_date + timedelta(days=40)
            if (datetime.now() - actualized_date) > timedelta(days=25):
                continue
            data_point = {
                'measurement': 'your_measurement_name',
                'tags': {
                    'code': properties.get('code', ''),
                    'name': properties.get('name', ''),
                },
                'time': actualized_date.isoformat(),
                'fields': {
                    'no2_1h': float(properties.get('no2_1h')) if properties.get('no2_1h
') is not None else None,
                    'pm10_1h': float(properties.get('pm10_1h')) if properties.get('
pm10_1h') is not None else None,
                    'pm10_24h': float(properties.get('pm10_24h')) if properties.get('
pm10_24h') is not None else None,
                    'pm2_5_1h': float(properties.get('pm2_5_1h')) if properties.get('
pm2_5_1h') is not None else None,
                },
            }
            influx_data.append(data_point)
        except ValueError:
            pass
```

2.5 Dotaz v jazyce daného databázového produktu

```
    # Jaka je prumerna hodnota no2 pro kazdy den a stanice
    query = """
        SELECT MEAN(no2_1h) as no2_mean
        FROM data_ovzdusi2
        WHERE time > now() - 30d
        GROUP BY time(1d), code
    """
    table = client.query(query=query, database=database, language='influxql')
    # Take to muzeme konvertovat do pandas framu pro nasledne zpracovani
    df = table.to_pandas()
```

```
#  Pocet hrubsich castic v case (PM10) pro Brno-Arboretum v poslednich 7 dnech
query = """
```

```
        SELECT code, pm10_1h, time,
        FROM "data_ovzdusi2"
        WHERE
        time >= now() - interval '7 days'
        AND
        ("no2_1h" IS NOT NULL)
        AND
        "code" IN ('BBMAA')

    """
    table = client.query(query=query, database=database, language='sql')
```



Obrázek 1: Zobrazeni dotazu pomocí influxDB na počet hrubších částic v čase

Popis způsobu, jakým databázový server dotaz zodpoví

InfluxDB využívá distribuovaný systém shardů, meta nodů a data nodů pro efektivní zpracování dotazů[2]. Když klient zadá dotaz, meta nody analyzují dotaz a určí, ve kterých datových nodech jsou potřebná data uložena a zasílají jim dotazy. Data nody obsahují samotná data a provádějí operace na základě dotazu. Výsledky jsou následně kombinovány v koordinačním uzlu a klient obdrží odpověď na svůj dotaz a poté ho může zkonzumovat například s pomocí pandas.

Pokud používáme několik serverů, tak jsou data rozdistribuována mezi ně na základě replikačního faktoru a shard doby. Pokud máme shard dobu 1 den, replikační faktor 2, a 4 datové uzly A, B, C a D, tak potom to může vypadat následovně. Pro každý den bude existovat shared group, která bude obsahovat 2 shardy, které odpovídají replikačnímu faktoru. Shard 1 bude uložen například na datových uzlech A a B, a shard 2 bude uložen datových na uzlech C a D. Data se sdílí mezi shardy na základě hashe[3].

[2]https://docs.influxdata.com/enterprise_influxdb/v1/concepts/clustering/
[3]https://docs.influxdata.com/enterprise_influxdb/v1/concepts/clustering/

3 Neo4J

3.1 Informace o datové sadě

- Použitá datová sada: Intenzita dopravy - Intenzita cyklistů
- Distribuce: GeoJSON

3.2 Odůvodnění vhodnosti zvoleného datasetu pro daný typ databáze

Charakteristické vlastnosti

Grafová databáze se hodí zejména pro popis a uložení dat, které lze vizualizovat grafově. Graf formálně definujeme podle (1), kde: V je množina vrcholů (nebo také uzlů) a E je množina hran, definovaná (2).

$$G = (V, E) \tag{1}$$

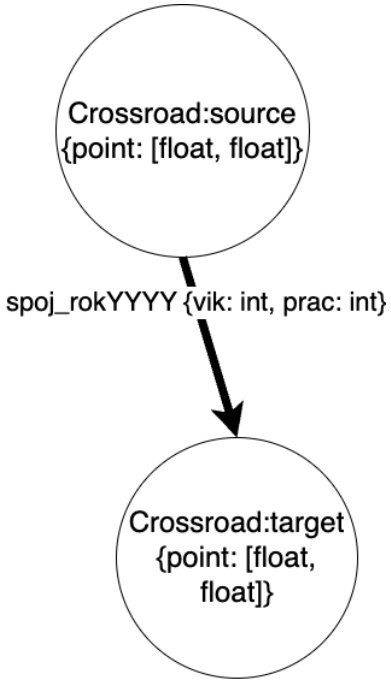
$$E \subseteq \{\{u, v\} | u, v \in V\} \tag{2}$$

V případě Neo4j se však jedná o orientovaný graf, kde oproti neorientovanému grafu definujeme hrany E jako v (3).

$$E \subseteq V^2, \forall e \in E : \exists u, v \in V : (u, v) = e \tag{3}$$

V praxi se grafy nejčastěji používají při určování nejkratších cest, nejrychlejších cest a podobně. Například se toto využívá při algoritmech, které řeší zadané úlohy (Breath first search, depth first search...), vyhledávání nejlepší cesty v routovací tabulce pro odeslání paketu, či v mapách a navigacích. Náš dataset je prakticky mapa bodů na mapě a jejich spojení skrze cyklostezku, na které se měří intenzita cyklistů za 24hodin. Našimi vrcholy grafu jsou tedy body na mapě (křižovatky cest) a hrany jsou jednotlivé cesty, pro které byla zaznamenávána intenzita cyklistů. Pro tento dataset schéma grafu vypadá jako na obrázku 2. Atribut `point` odpovídá souřadnici počátečního bodu dané trasy v souřadném systému GCS WGS84. Hrana s názvem `spoj_rokYYYY` odpovídá měření z daného roku YYYY. Atribut hrany `vik` odpovídá vytíženosti o víkendech a atribut hrany `prac` odpovídá vytíženosti v pracovní dny. [4]

[4] `vik` a `prac` je v kódu dále přejmenováno na `weekend`, respektive `week`



Obrázek 2: Schéma grafu Neo4j pro dataset intenzity cyklistů

Způsob, průběh vzniku dat a jejich uložení

Data ze zvoleného datasetu pochází z měření intenzity cyklistů v jednotlivých úsecích v Brně. Tyto data pravidelně zpracovávají Brněnské komunikace, a.s., tudíž lze tyto data považovat za důvěryhodné. Intenzita cyklistů je zpracovávána jednou za dva roky, jak plyne z dokumentace datasetu. To prakticky vylučuje možnost zpětné změny již publikovaných dat, ovšem lze předpokládat, že každé dva roky se přidá nové měření intenzity cyklistů, s čímž databáze musí počítat. S ohledem na typ dat je nutné především efektivně vyhledávat v grafu, k čemuž databáze Neo4j poskytuje řadu metod, jako například implementaci A* Shortest Path, Dijkstrův algoritmus, Depth first search, Breadth first search a další. Avšak kromě zpětné změny zapsaných dat lze očekávat, že bude potřeba přidat nové uzly.

Způsob a průběh spotřeby dat

Vzhledem k typu dat a informacím, které lze získat z datasetu lze očekávat, že se často bude vyhledávat nejméně / nejvíce vytížená cesta o víkendu / v týdnu. Také může být snaha hledat trasu z bodu A, do bodu B, což však pravděpodobně nebude tak častý případ jako první zmíněný případ. Taky by nás mohly zajímat nejvytíženější úseky, například pro odklonění dopravy alternativní cestou a podobně. Konzumenti výsledků budou z valné většiny právě z Brna, avšak nelze vyloučit, že například nějaký turista mimo Brno se nebude chtít podívat na vytížení jednotlivých cyklostezek v Brně pro účely pozdějšího výletu na kole. Pro zefektivnění dotazů databáze Neo4j disponuje několika druhy indexů: range index, lookup index, text index, point index, full-text index, což může napomoci k případně vyšším požadavkům na výkonnost.

3.3 Příkazy pro definici úložiště

Pro definici úložiště bylo třeba definovat uzly a hrany. Toto se děje v souboru `neo4j/loadFromGeoJson.py`, konkrétně funkce `insertNodes` 1, která využívá rozhraní třídy `Neo4jDB`, která sdružuje metody pro práci s databází 2. Tato třída především disponuje metodou `addNode`, která skrze příkaz `MERGE` vloží nový uzel do databáze pouze pokud uzel se souřadnicemi x, y (ty odpovídají souřadnicovému systému

GCS WGS84) se ještě v databázi nenachází. Hrany mezi uzly poté přidává metoda `softAddRelation`, která hranu mezi dva uzly vloží pouze tehdy, pokud mezi těmito uzly neexistuje stejně pojmenovaná hrana. V případě již existující pojmenované hrany mezi uzly se pouze přepíší atributy této hrany. Hrany se přidávají z obou stran (z uzlu a do uzlu b, i z uzlu b do uzlu a), jelikož z dokumentace spíše vypadá, že se počítala obousměrná intenzita cyklistů pro jeden úsek.

```
def insertNodes(featureObjects):
    for featureObject in featureObjects:
        geometry = featureObject.getGeometry()
        points = geometry.getPoints()
        startPoint = points[0]
        endPoint = points[len(points) - 1]
        startPointJson = startPoint.toJson()
        endPointJson = endPoint.toJson()
        startLabelNode = "ids{}".format(startPoint.getString())
        endLabelNode = "ide{}".format(endPoint.getString())
        db.addNode("Crossroad:{}".format(startLabelNode), "{" + "point: {}".format(
    startPointJson) + "}")
        db.addNode("Crossroad:{}".format(endLabelNode), "{" + "point: {}".format(
    endPointJson) + "}")
```

Listing 1: Funkce insertNodes ze souboru neo4j/loadFromGeoJson.py

```
from neo4j import GraphDatabase, RoutingControl

class Neo4jDB():
    def __init__(self, uri="neo4j://", host = "localhost", port = 7687, user = "",
    password = "") -> None:
        self.host = host
        self.port = port
        self.user = user
        self.password = password
        self.uri = uri

    def connect(self, database = "neo4j"):
        self.driver = GraphDatabase.driver(self.uri + self.host + ":" + str(self.port),
     auth=(self.user, self.password))
        self.database = database

    def clearDatabase(self):
        query = "MATCH (n) DETACH DELETE n"
        self._execute_query_w(query)

    def addNode(self, label, props):
        if label == "":
            query = "MERGE (node" + props + ")"
        else:
            query = "MERGE (node:" + label + props + ")"
        self._execute_query_w(query)

def softAddRelation(self, name, nodeFromLabel, nodeToLabel, props):
        query = "MATCH(a:{}), (b:{}) MERGE(a)-[rel:{}]->(b) SET rel = properties({})".
    format(nodeFromLabel, nodeToLabel, name, props)
        self._execute_query_w(query)
```

Listing 2: Třída Neo4jDB pro práci s databází

3.4 Algoritmický popis importu dat

Jelikož již máme v databázi uzly i hrany vytvořené, máme prakticky i naimporovanou většinu potřebných dat. Dále musíme pouze doimportovat dodatečné atributy hran či uzlů. Jelikož o uzlech není v datasetu žádná další informace, dodatečný import dat uzlů nebyl předpokládán. V případě hran stačí v případě vytvoření nových záznamů (periodičnost každé dva roky) pouze importovat nové hrany (pro každé dva roky se používá jedna hrana) metodou `softAddRelation`. Toto demonstruje kód 3. V tomto kódu si lze všimnout přidání hran pro rok 2016 vícekrát, což demonstruje to, že se tyto hrany přidají pouze jedenkrát (příkaz `MERGE`).

```
addRelations(featureObjects, "2016")
addRelations(featureObjects, "2016")
addRelations(featureObjects, "2018")
addRelations(featureObjects, "2020")
addRelations(featureObjects, "2022")
```

Listing 3: Import dalších hran pro nové záznamy, či aktualizace dat starších záznamů

3.5 Dotaz v jazyce daného databázového produktu

Dotaz nad Neo4j databází, který jsme implementovali, je zjištění nejméně vytížené trasy z bodu A do bodu B. Výsledkem tohoto dotazu je potom cesta grafem, obsahující vždy nejméně vytíženou trasu. V případě našeho datasetu sice nebyly k dispozici žádné křižovatky, kde by se dalo určovat jakou trasu zvolit, ovšem lze počítat, že do budoucna křížení v datasetu může nastat. Tak či tak je žádoucí, nalézt cestu z bodu A do bodu B. Neprve je potřeba vytvořit projekci grafu uloženou v katalogu grafu, která se uloží pod určitým názvem a se kterou poté bude pracovat dijskrův algoritmus pro vyhledání nejlevnější cesty. Toto je vidět v kódu 4, kde nejprve smažeme z katalogu grafů graf 'myGraph' a poté vytvoříme nový. Nakonec skrze volání Dijkstrova algoritmu nalezneme nejméně vytíženou cestu (v týdnu) z uzlu *source* do uzlu *target* 5.

```
CALL gds.graph.drop('myGraph', False);
CALL gds.graph.project(
    'myGraph',
    'Crossroad',
    'spoj_rok2016',
    {
        relationshipProperties: 'week'
    }
);
```

Listing 4: Projekce grafu do katalogu grafů pod názvem 'myGraph'

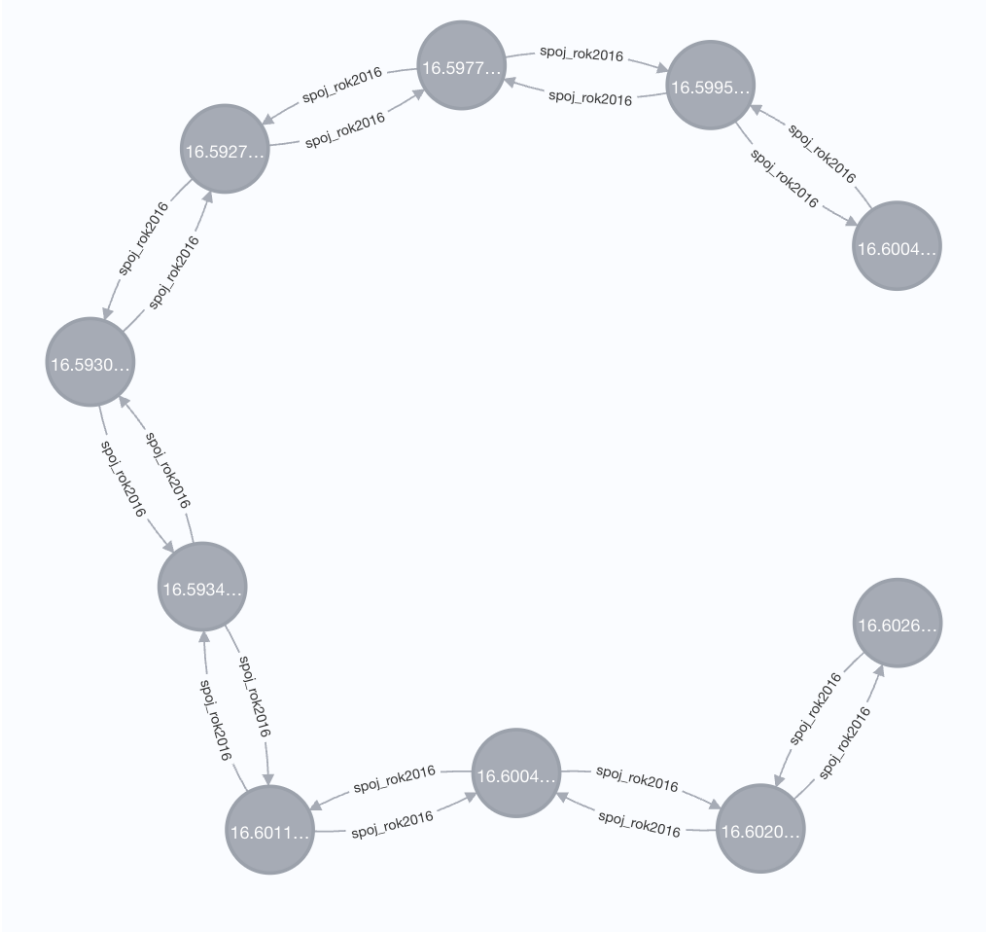
```
MATCH (source{point:[16.600407200000063,49.200695100000075]}), (target{point:
[16.602668400000027,49.200981900000045]})
CALL gds.shortestPath.dijkstra.stream('myGraph', {
    sourceNode: source,
    targetNode: target,
    relationshipWeightProperty: 'week'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
    index,
    gds.util.asNode(sourceNode).name AS sourceNodeName,
    gds.util.asNode(targetNode).name AS targetNodeName,
    totalCost,
    [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
    costs,
    nodes(path) as path
ORDER BY index
```

Listing 5: Volání Dijkstrova algoritmu pro nalezení nejméně vytížené trasy v týdnu z uzlu *source* do uzlu *target*

Popis způsobu, jakým databázový server dotaz zodpoví

Jak již bylo zmíněno výše, databázový server nejprve musí vytvořit projekci grafu do katalogu grafů. V tomto kroce z grafu vybere potřebné informace o hranách / uzlech. V našem případě probíhá projekce uzlů `Crossroad`, hran `spoj_rok2016` a atributu hran *week*. Tento atribut je dále používán v Dijkstrově algoritmu jako cena přechodu. Po vytvoření této projekce a uložení do katalogu grafů pod názvem `myGraph` může tento graf být zpracován Dijkstrovým algoritmem [5] a vrácena výsledná cesta. Cesta, kterou vrátil databázový server je vidět na obrázku 3. Tyto výsledky dále doručí klientovi skrze patřičný protokol, který je může dále interpretovat. Například je klient může vizuálně zobrazit na mapě, včetně intenzity provozu, a podobně.

[5] Dijkstra's Shortest Path Algorithm



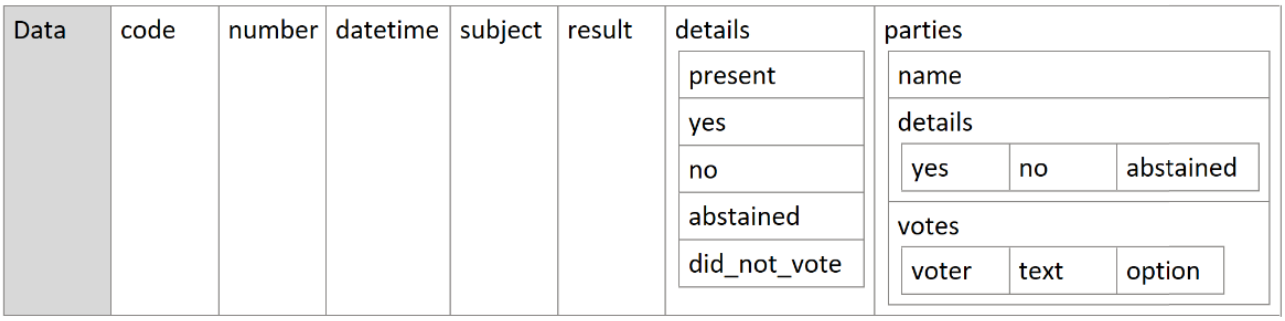
Obrázek 3: Nejméně vytížená cesta z uzlu *source* do uzlu *target*

4 MongoDB

4.1 Informace o datové sadě

- Použitá datová sada: Hlasování zastupitelstva
- Distribuce: JSON

Datová sada obsahuje záznamy, které mají strukturu zobrazenou na obr. 4. Dataset obsahuje podrobné výsledky z hlasování na schůzích městského zastupitelstva. Zdrojem dat jsou oficiální zápisy z jednotlivých zasedání, které jsou zveřejňována také na webu města Brna[6]. Data jsou poskytována ve formátu JSON a jsou aktualizována v horizontu jednoho týdne od konání schůze.



Obrázek 4: Struktura datasetu Hlasování zastupitelstva

4.2 Odůvodnění vhodnosti zvoleného datasetu pro daný typ databáze

Charakteristické vlastnosti

Data ve zvolené datové sadě jsou rozsáhlá, mají proměnlivé schéma (např. některé záznamy obsahují klíč `did_not_vote`) a obsahují vnořené objekty a pole (např. `data.parties`, `data.parties.votes`). Data jsou uložena ve formátu JSON, který MongoDB (jako dokumentová databáze) umožňuje vkládat jako jednotlivé dokumenty. Obsahují poměrně hodně textu (předmět a výsledek hlasování, názvy politických stran, jména zastupitelů, jejich volbu apod.), číselné hodnoty (např. výsledky hlasování), datumy a pole objektů (např. seznam zastupitelů jednotlivých stran), se kterými může MongoDB snadno manipulovat, provádět sumace, filtrovat data apod.

Datová sada neobsahuje přímo specifikovaný identifikátor, jako např. `_id`. Po důkladné analýze datové sady se ukázalo, že unikátním identifikátorem dat by mohl být `code` (číslo schůze zastupitelstva) v kombinaci s `number` (číslo hlasování v rámci dané schůze) např. `Z9/09_29`. Tento identifikátor je jedinečný v rámci datové sady, je krátký a má v rámci dat význam.

Způsob, průběh vzniku dat a jejich uložení

Zdrojem dat jsou konané schůze Zastupitelstva města Brna, které se konají cca každých 6 týdnů, data tedy obsahují výsledky hlasování o návrzích, které probíhá na těchto schůzích. Data jsou aktualizována v horizontu jednoho týdne od konání schůze. Můžeme předpokládat, že zdroj dat je důvěryhodný, jelikož je zastupitelstvo orgán veřejné správy a data jsou aktualizována po konání schůze. Chybovost dat by měla být minimální s ohledem na to, že data jsou zveřejňována s týdenním odstupem od hlasování, dochází tedy k revizi dat před jejich aktualizací. V kontextu výsledků hlasování zastupitelstva by mělo

[6] https://www.brno.cz/zasedani-zmb

být zřídka kdy potřeba měnit již publikovaná data. Jakmile je hlasování uzavřeno a výsledky jsou zaznamenány a publikovány, měly by tyto výsledky zůstat neměnné pro transparentnost a důvěryhodnost procesu. Proto budeme předpokládat pouze možnost přidávání nových dat.

Pro provedení složitých dotazů nad strukturovanými daty lze využít agregační framework MongoDB, který poskytuje sadu operací, které lze řetězit (viz příklad níže). Zapomínání dat je závislé na zákonech a nařízeních, týkajích se retence dat a veřejného přístupu k záznamům zastupitelstva. Záleží tedy na účelu aplikace. Pokud bychom předpokládali, že data budou vyvěšena webových stránkách zastupitelstva (jako tomu je na oficiálních webových stránkách Města Brna - Zasedání ZMB[7]), tak v takovém případě musí být výsledky dostupné k nahlédnutí veřejnosti po dobu několika let. Množství dat uložených v databázi tedy bude velký a lze předpokládat že poroste. Pro zajištění vysoké propustnosti, dostupnosti, spolehlivosti a efektivity bude potřeba využít metodu *shardingu* (horizontální škálování).

Způsob a průběh spotřeby dat

Pro předdefinované dotazy lze v MongoDB použít `aggregation pipelines`, které transformují a analyzují data, lze je optimalizovat pomocí indexů a mohou být využívány aplikací pomocí API endpointů nebo procedur. Předdefinované dotazy budou pokrývat dotazy na body hlasování konkrétní schůze, přijaté / zamítnuté návrhy schůze apod.. V MongoDB je také možné vytvářet ad-hoc dotazy. Pokud dotaz odpovídá existujícímu indexu, MongoDB automaticky využije tento index k urychlení dotazu. V aplikaci, která data pouze zobrazuje uživateli, však ad-hoc dotazy nebudou častým jevem. Pokud by se v budoucnu rozhodlo rozšířit funkčnost aplikace nebo umožnit uživatelům provádět vlastní dotazy, mohlo by to mít výkonnostní dopady, pokud nebudou data správně indexována.

Protože Zastupitelstvo zasedá přibližně jednou za 6 týdnů a data v datové sadě jsou aktualizována v horizontu týdne od konání schůze, lze očekávat zvýšený počet dotazů na databázi během tohoto týdne, kdy obyvatelé města hledají informace o programu, usneseních nebo výsledcích hlasování. Jinak bude frekvence čtení nízká. Perioda čtení by v horizontu tohoto týdne měla být častá (několikrát denně), aby byly zajištěny aktuální informace o zasedání. Mimo tento týden bude perioda čtení méně častá, např. týdně.

Pro distribuci dat pro dotazy Mongo podporuje replikační sady pro zajištění vysoké dostupnosti. Jak je zmíněno výše, je očekáváno, že objem dat bude v čase růst, proto bude potřeba využít *sharding* na základě zvoleného shard klíče. Volba vhodného shard klíče je kritická pro efektivní škálování databáze, protože udává jak budou data rozdělena mezi shardy, kdy ideální klíč by měl zajistit, že data budou rozdělena rovnoměrně mezi shardy. Vhodným shard klíčem pro tuto datovou sadu (viz obr. 5) by mohl být `code` pro kolekci Subject a `name` pro kolekci Parties, protože lze očekávat, že budou data dotazována na základě kódu zasedání (`code`), či jména politické strany (`name`). Před nastavním shard klíče je nutné pro hodnoty shard klíče nastavit indexaci (pokud jako shard key není zvolen `_id`). Lze předpokládat, že hlavními konzumenty dat budou obyvatelé města Brna, proto je vhodné vybrat místo uložení a zpracován co nejblíže Brnu, aby byla minimalizována latence při načítání dat. MongoDB má integrovanou cache pro časté dotazování, takže v případě že budou data často dotazována, mohou být uchována v paměti pro rychlejší přístup. Výsledky výpočetně náročných dotazů lze také předpočítat a uložit do databáze.

4.3 Příkazy pro definici úložiště

Pro účely projektu jsme použili cloudové úložiště MongoDB Atlas. Pro práci s MongoDB jsme použili knihovnu `pymongo`. Pro definici úložiště a import dat byl použit skript `mongo_db.py`. Příklad připojení k NoSQL databázi MongoDB:

```
from pymongo import MongoClient
```

[7] https://www.brno.cz/zasedani-zmb

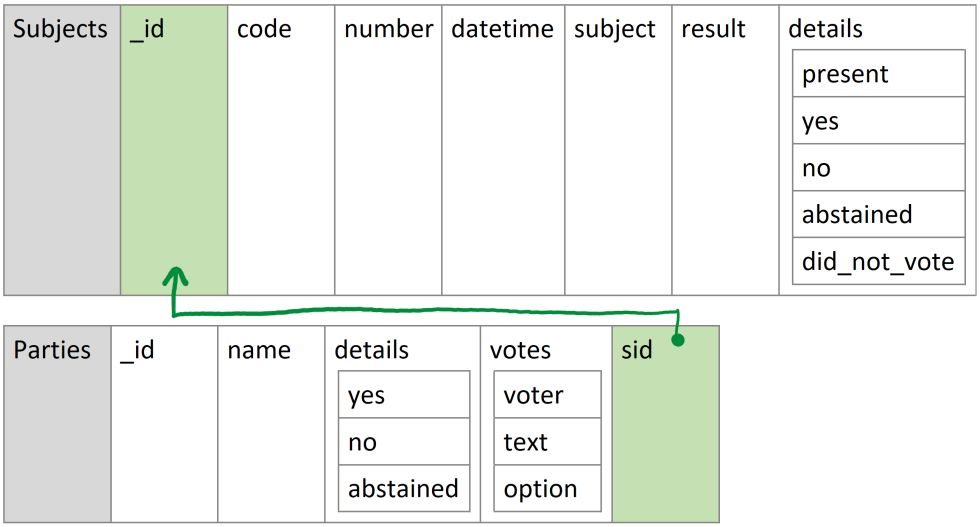
```
# Pripojovaci retezec z MongoDB Atlas
mongo_uri = "your_mongo_uri"

# Vytvoreni klienta
client = MongoClient(mongo_uri)

# Vyber databaze
db = client['your_database_name']
```

Způsobů, jak definovat kolekce nad vybranou datovou sadou existuje několik. Záleží na tom, jak budou data dotazována, jak často budou aktualizována a jestli bude aplikace orientována převážně na čtení, nebo zápis. Jestliže víme, že budeme často provádět dotazy přes různé kolekce, může být výhodné data denormalizovat a kombinovat je do jedné kolekce. Toto řešení je ideální pro aplikace, které jsou primárně zaměřeny na čtení dat. Využitím schopností MongoDB můžeme vytvářet složitě strukturovaná vnořená data, a tím značně urychlit dotazování. Naopak pokud předpokládáme časté aktualizace nebo zápis dat, je praktičtější mít pro tyto údaje separátní kolekci. Tím zajistíme, že změny v jedné části databáze neovlivní ostatní části, systém bude rychlejší a bude zajištěna konzistence dat. Denormalizace však může způsobit datovou nekonzistenci, a z toho důvodu je důležité pečlivě zvážit kompromis mezi výkonem a konzistencí při návrhu databáze a zvolit správný přístup pro konkrétní potřeby aplikace.

Předpokládejme hypotetickou aplikaci pro veřejnost, která bude pouze zobrazovat výsledky zasedání zastupitelstva. Budeme předpokládat že data, která Zastupitelstvo zveřejní budou finální a data nebudou dále aktualizována. Taková aplikace bude pro každé zasedání zobrazovat základní údaje o schvalovaných návrzích, tj. název návrhu, rozhodnutí a základní statistiku o hlasování (kolik zastupitelů bylo pro a proti, se zdrželo a bylo nepřítomno). Po výběru určitého návrhu bude zobrazovat statistiku hlasů jednotlivých stran a po rozkliknutí jednotlivých stran bude zobrazeno hlasování každého zastupitele dané strany.

Pro takovou aplikaci jsme zvolili dvě kolekce, Parties a Subjects (znázorněno na obr. 5). Kolekce Parties je denormalizována, jelikož pole `votes` obsahuje pole objektů jednotlivých hlasů každého zastupitele ve straně (nepředpokládáme dotazy na hlasy jednotlivých zastupitelů). Parties `name` a `votes.votes` nejsou normalizovány, protože nepředpokládáme častou změnu jména zastupitelů a politických stran. Díky denormalizaci je pak možné provádět efektivnější dotazy nad daty a využít tak potenciál MongoDB. Přesto jsme data rozdělili do kolekcí Subject a Parties, aby bylo možné provádět dotazy na hlasování politických stran efektivně. Poté na základě potřeb takové aplikace vytvoříme kolekce v `pymongo` následovně:

```
# Vyber kolekce
subjects = db["subjects"]
parties = db["parties"]
```



Obrázek 5: Kolekce v MongoDB

Příklad vložení dokumentu v `pymongo`:

```
# Data o navrhu pro vlozeni
subject = {
    "_id": sid,
    "code": "Z9/09",
    "number": 98,
    "datetime": "2023-09-05T10:13:23+00:00",
    "subject": "93. Navrh prodeje pozemku p. c.5497 v k. u. Bystrc",
    "result": "Prijato",
    "details": {
            "present": 51,
            "yes": 42,
            "no": 0,
            "abstained": 7,
            "did_not_vote": 2
        }
}
# Kriterium pro hledani v kolekci subjects podle klice
criteria = {"code": data["code"], "number": data["number"]}
# Aktualizace dat v databazi (pokud neexistuji, jsou vlozena)
subjects.update_one(criteria, {"$set": subject}, upsert=True)
```

4.4 Algoritmický popis importu dat

Pro import dat byl použit skript `mongo_db.py`. Hlavní algoritmus importu dat:

```
for data in data_json["data"]:
    # Vytvoreni _id kombinaci dvou hodnot
    sid = f"{data['code']}_{data['number']}"
    # Data o navrhu pro vlozeni
    subject = {
        "_id": sid,
        "code": data["code"],        # Cislo schuze zastupitelstva
        "number": data["number"],    # Cislo hlasovani v ramci dane schuze
        "datetime": datetime,        # Datum a cas hlasovani
        "subject": data["subject"], # Predmet hlasovani
        "result": data["result"],    # Vysledek hlasovani (Prijato/Neprijato)
        "details": data["details"]  # Detailni vysledky hlasovani
    }
```

```
        # Kriterium pro hledani v kolekci subjects podle klice
        criteria = {"code": data["code"], "number": data["number"]}
        # Aktualizace dat v databazi (pokud neexistuji, jsou vlozena)
        subjects.update_one(criteria, {"$set": subject}, upsert=True)

        for p in data["parties"]:
            party = {
            "_id": pid,
            "name": p["name"],      # Nazev politicke strany
            "details": p["details"],# Detailni vysledky hlasovani dane strany
            "votes": p["votes"],    # Pole udaju o hlasovani jednotlivych zastupitelu
            "sid": sid,
            }

            # Kriterium pro hledani v kolekci parties podle klice
            criteria = {"name": p["name"], "sid": sid}
            # Aktualizace dat v databazi (pokud neexistuji, jsou vlozena)
            parties.update_one(criteria, {"$set": party}, upsert=True)

            pid+=1
```

4.5 Dotaz v jazyce daného databázového produktu

```
        # Kolik navrhu ze schuze s kodem Z9/09 bylo prijato?
        code = "Z9/09"
        result = "Prijato"
        agg_result2 = collection.aggregate([
                {"$match": {"code": code, "result": result}},
                {"$group": {"_id": "$code", "accepted_total": {"$sum": 1}}}
            ])
```

Popis způsobu, jakým databázový server dotaz zodpoví

Každý dokument v kolekci, která je rozdělena mezi shardy, je umístěn na konkrétním shardu na základě hodnoty jeho shard key (klíče, podle kterého jsou data rozdělena). Pokud dotaz obsahuje shard key, může být dotaz směrován přímo na správný shard. Pokud shard key neobsahuje, je dotaz rozeslán na všechny shardy (tzv. broadcast)

Dotazy jsou zpracovávány paralelně na všech relevantních shardech, což umožňuje distribuované zpracování dat a paralelní výpočty. Pokud je dotaz agregace, každý shard zpracuje agregaci (`$match` a `$group`) na své vlastní sadě dat a vrátí mezivýsledky. Po zpracování dotazu na relevantních shardech jsou výsledky vráceny zpět do routeru dotazů v sharded prostředí MongoDB zvaného `mongos`, který kombinuje výsledky a vrátí jej klientovi.

5 Apache Cassandra

5.1 Informace o datové sadě

- Použitá datová sada: Obsazenost parkovacích domů a parkovišť
- Distribuce: CSV

Data z veřejně dostupných parkovacích ploch v Brně. Aktualizace datové sady probíhá každých 5 minut a zobrazují základní atributy o počtu volných míst, kapacitě. Bližší popis obsažených atributů je uveden v tab.2. Data s historickými záznamy rok zpětně jsou dostupná na vyžádání: data@brno.cz. Souřadnicový systém: GCS WGS84.

5.2 Odůvodnění vhodnosti zvoleného datasetu pro daný typ databáze

Charakteristické vlastnosti

Jednotlivé sloupce jsou popsány v tab.2. V Apache Cassandra je možné přidat/odstranit sloupce do/ze stávající tabulky. S vývojem aplikace by tato vlastnost mohla být v budoucnu užitečná pro přidání dalších potřebných sloupců. V datasetu zatím není mnoho záznamů, ale s narůstajícím počtem parkovišť, z nichž budou sbírána data, lze očekávat jejich nárůst. Data v datové sadě již obsahují identifikátor `objectid`.

Atribut	Popis
objectid	id záznamu
name	název parkovací plochy
capacity	celková kapacita dané plochy
free	počet volných míst
statusTime	čas platnosti záznamu
Latitude	souřadnice x
Longitude	souřadnice y
cars	počet vozidel
capacityProcent	zaplněnost (%)
datum	datum
startDate	čas a datum záznamu
spacesSubscribersVacant	volná místa předplatitelé
spacesSubscribersOcupied	obsazená místa předplatitelé
spacesAllUsersOccupied	obsazená místa všichni uživatelé
capacityForPublic	počet volných míst pro lidi bez předplatného parkovacího místa

Tabulka 2: Firmy v Brně: Schéma

Způsob, průběh vzniku dat a jejich uložení

Zdrojem dat jsou parkovací plochy v Brně, které monitorují obsazenost téměř v reálném čase. Zda jsou data schromažďována pomocí IoT zařízení, či jinou metodou z dokumentace datasetu není patrné. Aktualizace dat je prováděna v pravidelných 5-ti minutových intervalech, takže je očekáván častý zápis dat. Apache Cassandra je vhodná pro takové scénáře, protože je optimalizována pro rychlý zápis dat. V datové sadě se nachází pouze aktuální hodnoty pro každé monitorované parkoviště. Proto na základě dostupných dat nelze určit jejich důvěryhodnost a chybovost dat. Z dat které jsou dostupné není chybovost viditelná, takže předpokládáme je minimální. Vzhledem k vysoké frekvenci aktualizace dat je pravděpodobné, že zpětná změna dat bude minimální, a bude převážně docházet k přidávání nových.

Uživatelé by mohli ocenit možnost základních agregací, jako je např. nejméně aktuálně obsazená parkovací plocha. Databáze Apache Cassandra také podporuje datovou kompresi na bázi tabulek[8] pomocí pěti algoritmů, které mají různé výhody i nevýhody a lze je volit podle specifických potřeb a charakteru dat.

Retence dat bude odpovídat časovému úseku mezi jednotlivými aktualizacemi dat. Na základě datové sady předpokládámě, že aplikace bude zobrazovat pouze aktuální data pro každou monitorovanou plochu. Každá nově příchozí data budou podle příslušného `objectID` aktualizována, nebude tedy uchovávána historie dat. Na narůstající objem dat v databázi je možné reagovat horizontálním škálováním, tj. přidat další uzly do clusteru, bez nutnosti zastavení služby. Data budou pak automaticky redistribuována mezi všemi uzly. Redundance dat je zásadní pro vlastnosti Apache Cassandra.

Způsob a průběh spotřeby dat

Pro dotazování se používá Cassandra Query Language (CQL). Jelikož jsou data aktualizována každých 5 minut, měla by perioda čtení odpovídat také 5-ti minutám. Pokud by se objem dat v databázi zvýšil, lze jednoduše a bez výpadků přidat další uzly do clusteru. Jelikož dataset obsahuje záznamy pouze o Brněnských parkovištích, je jasné že konzumenti se budou převážně nacházet na území Brna. Nelze však vyloučit, že si data nebudou zobrazovat také uživatelé ve větší vzdálenosti, např. právě přijíždějící řidiči. Jednou z vlastností Cassandry je možnost specifikovat, kde budou data fyzicky uložena. V případě rozšíření aplikace do jiných měst, tato vlastnost by snížila latenci pro každého uživatele. Cassandra disponuje vlastní cache[9] *Key Cache* a *Row Cache*. Tyto cache mohou být konfigurovány tak, aby optimalizovaly výkon pro časté dotazy. Cassandra také disponuje sekundárními indexy, nevhodným použitím však může dojít ke snížení výkonu.

5.3 Příkazy pro definici úložiště

```
            # Definice uloziste pro Apache Cassandra
            # Popis importu dat pro Apache Cassandra
            import json
            import pandas as pd
            from cassandra.auth import PlainTextAuthProvider
            from cassandra.cluster import Cluster
            ASTRA_DB_APPLICATION_TOKEN = "token"
            ASTRA_TOKEN_PATH = 'token.json'

            data = pd.read_csv(
                'park.csv')
            cluster = Cluster(cloud={
                "secure_connect_bundle": "secure-connect.zip",
            },
                auth_provider=PlainTextAuthProvider(
                "token",
                ASTRA_DB_APPLICATION_TOKEN,
            ),)
            session = cluster.connect("bamboo")
            # Vytvoreni keyspacu
            # session.execute(
            #     "CREATE KEYSPACE IF NOT EXISTS bamboo WITH replication = {'class': '
    SimpleStrategy', 'replication_factor': 1}")

            session.execute("USE bamboo")
            # Vytvoreni tabulku
            session.execute("""
```

[8] https://cassandra.apache.org/doc/latest/cassandra/operating/compression.html
[9] https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/operations/opsSetCaching.html

```
    CREATE TABLE IF NOT EXISTS parking (
        ObjectId INT PRIMARY KEY,
        name TEXT,
        capacity INT,
        free INT,
        ....
    )
""")
```

5.4 Algoritmický popis importu dat

```
# Vloz data do tabulky
for index, item in data.iterrows():
    # print(item["X"])
    insert_query = """
        INSERT INTO parking (
            ObjectId, name, capacity, free, ...
        )
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    """
    session.execute(session.prepare(insert_query), [
        item["ObjectId"], item["name"], item["capacity"], item["free"],
        ...
    ])
# Close the Cassandra session and cluster connection
session.shutdown()
cluster.shutdown()
```

5.5 Dotaz v jazyce daného databázového produktu

```
# Dotaz pomoci ktereho ziskam aktualni pocet volnych mist na parkovistich
res = session.execute("Select name, free from parking")
for row in res:
    print(f"name: {row.name}, free spaces: {row.free}")
```

Popis způsobu, jakým databázový server dotaz zodpoví

Čtecí požadavek od klienta je odeslán na koordinační uzel, který zjišťuje, které repliky jsou zodpovědné za data a zda je dosaženo požadované úrovně konzistence.

Koordinátor zkontroluje, zda je zodpovědný za tato data. Pokud ano, uspokojí požadavek. Pokud ne, pošle požadavek k nejrychleji odpovídající replice. Také odešle k ostatním replikám žádost o kontrolní součet.

Porovná kontrolní součty vrácených dat. Pokud jsou všechny stejné a požadovaná úroveň konzistence je dosažena, data jsou vrácena z nejrychleji odpovídající repliky. Pokud kontrolní součty nejsou stejné, koordinátor spustí některé operace na opravu čtení. Požadovanou úroveň kozintence je možné měnit a budu nam určovat, jaký balanc mezi konzitencí a dostupností si chceme zvolit.

Na každé replice se provádí několik kroků: kontrola řádkové cache, kontrola memtable a kontrola sstables[10].

[10] https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/dml/dmlAboutReads.html