
# UPA – Ukládaní a příprava dat

## Ukládání rozsáhlých dat v NoSQL databázích

(Dokumentace k první části projektu)

**Bc. Ján Maťufka** xmatuf00@stud.fit.vutbr.cz

**Bc. Maxim Plička** xplick04@stud.fit.vutbr.cz

**Bc. Michal Pyšík** xpysik00@stud.fit.vutbr.cz

# 1 Kličové vlastnosti jednotlivých typů NoSQL databází

## 1.1 Sloupcová wide-column databáze

Wide-column NoSQL databáze[1] organizuje úložiště dat do flexibilních sloupců, které se můžou být rozloženy v rámci vícera serverů či databázových uzlů. Je využito více-dimenzionální mapování pro možnost referencování dat dle jednotlivých sloupců, řádků, či časových razítek (timestamps).

Důležitou vlastností je, že názvy a formáty jednotlivých sloupců se pro jednotlivá data můžou lišit, a to i v rámci jedné tabulky – velká flexibilita. Zatímco klasické relační databáze ukládají data po řádcích (každý záznam musí mít stejný počet hodnot – při nevyužití třeba dostadit `null` nebo nějakou výchozí hodnotu), zde jsou data vnitřně uložena ve sloupcích, tudíž pro každý záznam stačí do databáze vložit pouze opravdu existující hodnoty, a dotazy na jednotlivé samostatné hodnoty v daném sloupci jsou velice rychlé. Sloupce, které spolu smysluplně souvisí, lze také modelovat jako společnou sloupcovou rodinu, ve které jsou data ukládány klasickým způsobem po řádcích.

Hlavními benefity wide-column NoSQL databází jsou tedy rychlost dotazování, škálovatelnost, a flexibilní přístup k datovému modelování.

## 1.2 Dokumentová databáze

Data v dokumentových databázích se ukládají ve formě dokumentů. Tyto dokumenty jsou většinou ve formátu JSON nebo BSON. Vyznačují se flexibilitou a různorodostí dat, jelikož každý ukládaný dokument může mít jinou strukturu. Ze všech typů NoSQL jsou dokumentové databázy nejpopulárnejší jako alternativa k tradičním tabulkovým relačním databázím. Výhodou dokumentových databáz jsou zejména[2]:

- **Intuitivnost datového modelu** – umožňuje rychlost a jednoduchost práce pro vývojáře. To je mimo jiné umožněno faktem, že v mnohých populárních programovacích jazycích se dokumenty můžou jednoduše mapovat na objekty.
- **Flexibilita datových schémat** – ne všechny dokumenty z dané kolekce musí mít stejná pole. Datový model se může za běhu měnit/vyvíjet podle potřeby bez výrazných problémů.
- **Možnost horizontálního škálování** – pomocí „shardingu“ a replikace dat. Sharding (rozmístění logicky souvisejích dat do různých uzlů v síti) umožňuje distribuci dat a replikace zajišťuje jejich odolnost (vůči výpadkům apod.).

## 1.3 Grafová databáze

V případech, kdy se pracuje s daty, které jsou hluboce propojené komplexními vztahy, je velice výhodné využít grafové databáze. Data se v nich ukládají jako uzly a hrany, kde uzly reprezentují entity a hrany popisují vztahy mezi entitami. Grafové databáze se hodí zejména u aplikací, které vyžadují hluboké a komplexní dotazy. Využití proto najdou zejména v sociálních sítích, doporučujících enginech (YouTube apod.), nebo systémech pro detekci podvodů. Lze je využít i v jiných oblastech, např. řízení dodavatelských řetězců, správa sítí a infrastruktury a bioinformatika[3].

Mezi hlavní výhody grafových databází patří:

- Zpracování a reprezentace vztahů – vztahy mezi entitami jsou stejně důležité jako entity a často je nelze snadno reprezentovat v tradiční relační databázi.
- Flexibilita – poradí si s měníci se strukturou dat, je možno ji přizpůsobit novým případům použití bez výrazných změn schémy databáze (výhoda pro aplikace s rychle se měnícími datovými strukturami nebo složitými požadavky na data).

Grafové databáze se ale nemusí hodit vždy, zejména pro aplikace, které využívají jenom jednoduché dotazy nebo které se zabývají jen daty, které možno jednoduše reprezentovat tradiční relační databázi. Navíc práca s grafovými databázemi může vyžadovat k efektivnímu použití specializované znalosti a zkušenosti.

[1] https://www.scylladb.com/glossary/wide-column-database/
[2] https://www.mongodb.com/document-databases
[3] https://www.geeksforgeeks.org/introduction-to-graph-database-on-nosql/

### 1.4 Databáze časových řad

Databáze časových řad je specializovaný druh databáze, který je navržen k ukládání a správě velkého množství časově závislých dat. Časové řady jsou seřazené datové body, které jsou zaznamenávány v pravidelných intervalech času a mohou pocházet od různých zdrojů. Typický zdroj těchto dat může představovat například senzor, který pravidelně zasílá aktualizace snímaných hodnot. Mimo pravidelné akce mohou tyto databáze mohou zpracovávat i akce nepravidelné, které mohou být vyvolány jako reakce na určitou událost. Z těchto zmíněných faktorů vyplývá, že hlavními využitími tohoto typu databáze jsou schopnost analyzovat historická data, provádět predikce budoucího chování a reagovat na události v reálném čase. Tyto databáze také umožňují automatickou agregaci a mazání dat podle potřeby. Kromě toho jsou tyto databáze snadno rozšiřitelné. Při přidání dalšího zdroje dat, například nového senzoru, je třeba pouze přidat nový typ měření, aniž by bylo nutné provádět zásadní změny v existující databázi[4].

Mezi hlavní výhody tohoto typu databáze patří:

- Rychlý přístup k datům – Databáze časových řad jsou speciálně navrženy pro ukládání dat časových řad, což je dělá efektivnějšími v ukládání a dotazování.
- Operace nad daty – Databáze časových řad často obsahují vestavěné analytické a správcovské funkce navržené speciálně pro data časových řad.
- Komprese dat – Databáze obsahují mechanismy pro ukládání dat v komprimované podobě, což je důležité pro ukládání velkého objemu dat.

Na druhou stranu se s tímto druhem databáze pojí nevýhody. Mezi ně se určitě řadí specificita formátu dat. Tato databáze se jednoduše nehodí pro obecný formát dat a nemá tak široké spektrum využití jako některé obecnější databáze. Další nevýhodou může představovat složitost nastavení této databáze. A v neposlední řadě existuje relativně málo možností výběru mezi databázemi časových řad, tudíž může být problémové najít specifickou databázi, která vyhovuje potřebám uživatele.

## 2 Úvodní poznámky

Poznámka: Kořenová složka odevzdaného zip souboru obsahuje 4 složky pro jednotlivé datasety a typy NoSQL databází, dále pak (kromě této dokumentace) soubor `requirements.txt`, obsahující seznam potřebných Python balíčků (kolektivně pro všechny skripty – `pip install -r requirements.txt`). Podrobnější návody, případně další užitečné materiály specifické pro danou ukázku jsou poté v jednotlivých složkách. Snažili jsme se vyhnout kopírování Python kódu zde do dokumentace (jednotlivé kroky jsou zde popsány více abstraktně, nejedná-li se o příkaz v samotném dotazovacím jazyce), je tedy doporučeno při čtení nahlížet přímo do přiložených skriptů.

Při práci se čerpalo zejména z dokumentací jednotlivých databází (viz Tabulka 1) na Internetu. Všechny demonstrace byly vytvořeny a testovány s použitím Python verze 3.10 a výš.

| | |
|---|---|
| **wide-column:** | `Cassandra 4.1.3, cqlsh 6.1.0, CQL spec 3.4.6, Native protocol v5` |
| **dokumentová:** | `MongoDB 7.0.2, Mongosh 2.0.1` |
| **grafová:** | `Neo4j 5.12.0, Cypher-Shell 5.12.0` |
| **časových řad:** | `InfluxDB v1.8.10` |

Tabulka 1: Jednotlivé verze databáz a dotazovacích jazyků (veškerá práce s databázemi probíhala přes Docker kontejnery).

## 3 Cyklistické nehody

Pro sloupcovou wide-column databázi jsme vybrali dataset **Cyklistické nehody**[5] (formát `csv`). Tuto část projektu měl na starost primárně xpysik00.

[4] https://devops.com/time-series-database-basics/

[5] https://data.gov.cz/datov%C3%A1-sada?iri=https%3A%2F%2Fdata.gov.cz%2Fzdroj%2Fdatov%C3%A9-sady%2F44992785%2F791b73551a6e1e619e2f594de13ea6a4

## 3.1 Zvolený typ NoSQL databáze a datové sady

Tento dataset jsme zvolili jako vhodný pro práci s wide-column databází. Jako konkrétní produkt jsme zvolili **Apache Cassandra**[6], a to hlavně z důvodu, že je aktuálně nejpopulárnější/nejrozšířenější. Samotný skript pro demonstraci je ve formě Python skriptu `cassandra/demonstration.py` a pracuje s nejnovější distribucí Cassandry běžící uvnitř Docker containeru.

Mezi důvody výběru tohoto datasetu pro wide-column databázi patří například chybějící hodnoty u některých sloupců v (pouze) některých záznamech, různorodost/obecnost jednotlivých sloupců této datové sady obecně, a rád bych řekl že také velikost datasetu (tyto databáze jsou známé schopností efektivní práce s obrovským množstvím dat), ale z praktických důvodů jsme pro demonstraci vynechali příliš velké datasety (1346 záznamů/řádků je v praxi hodně málo, avšak pro jednoduchou demonstraci snad dostačující).

## 3.2 Popis přípravy a uložení dat

Datová sada byla stažena ve formátu `csv`, jelikož se jedná o asi nejminimalističtější formát a pro tento případ bohatě postačující (není zde například žádné vnořování kvůli složeným atributům).

Data jsou nejprve z csv souboru načtena do Pandas dataframu. Původní sloupce `datum` a `cas` jsou poté sloučeny a převedeny do jediného sloupce obsahující timestamp ve vhodném podporovaném formátu[7], v případě invalidních dat (např. čas nehody 28:47) je pro daný záznam (pouze) daný sloupec vynechán. Sloupce které jsme shledali za nepotřebné či redundantní (není třeba ukládat 'leden' když už máme timestamp) jsou poté kompletně zahozeny, vyjma případů kde jsou data odvoditelné, ale ne přímě (uchováváme informaci že se nehoda stala v neděli, i když pomocí kalendáře si každý může dohledat že dané datum vychází na neděli).

Cassandra v našem případě běží paralelně/distribuovaně v rámci 2 služeb, abychom demonstrovali její distributivní vlastnost. Dále je vytvořeno keyspace speciálně pro naše účely, především abychom naši tabulku separovali od systémového rozhraní nutném pro samotnou správnou funkčnost Cassandry (i tato systémová data mají často formy tabulek):

```
CREATE KEYSPACE IF NOT EXISTS cn
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1}
```

V daném keyspace jsme poté vytvořili tabulku s vhodným schématem včetně jednotlivých datových typů:

```
CREATE TABLE IF NOT EXISTS cn.cyklonehody (
id BIGINT, datum TIMESTAMP, den INT, srazka TEXT, ...,
PRIMARY KEY (id)
);
```

Poznámka: Jednotlivé sloupce by bylo možné také seskupit do sloupcových rodin (např. `point_x` a `point_y` seskupit do rodiny "zeměpisné souřadnice"), to však ve skriptu demonstrováno není.

Po definici schématu jsou jednotlivá data záznam po záznamu vložena do databáze:

```
INSERT INTO cn.cyklonehody ( id, datum , ... )
VALUES (60040140073, 2014/08/03 04:30:00+0100, ... ) IF NOT EXISTS;
```

Ve skriptu byly chybějící hodnoty dříve převedeny z `NaN` na `None`, a to aby bylo při vkládání jednodušší vždy ukládat pro daný záznam hodnoty jen v relevantních (nechybějících) sloupcích.

## 3.3 Dotazování nad daty

Dotazy v tomto typu NoSQL databáze jsou velmi podobné klasickému SQL. Jako vhodný dotaz byla zvolena otázka "Kolik lidí v Brně zemřelo v cyklonehodách zaviněných muži (dospělými, ne chlapci) od roku 2012?". Dotaz poté vypadá následovně:

```
SELECT SUM(usmrceno_os) as celkem_usmrceno_os
FROM cn.cyklonehody
WHERE datum > '2012-01-01 00:00:00+0100' AND pohlavi = 'muž'
ALLOW FILTERING;
```

[6] https://cassandra.apache.org/doc/latest/cassandra/cql/index.html
[7] https://cassandra.apache.org/doc/latest/cassandra/cql/types.html#timestamps

Odpověď na dotaz: `Row(celkem_usmrceno_os=4)`.

Co se zpracování dotazu Cassandrou týče, klient komunikuje s koordinačním uzlem (coordinator node), jehož úkolem je směrovat dotaz na příslušné uzly s daty (koordinační uzel má přístup k topologii clusteru i distribuci dat). Poté je extrahován tzv. partition key (filtrování pomocí `WHERE`), pro směřování dotazu na správné uzly – k tomu je navíc využit mechanismus distribuovaného hashování, koordinační uzel tedy kontaktuje vhodné uzly a ty mu vrátí dotazovaná data (tento proces se děje paralelně, ne sekvenčně). Samotná agregace v dotazu (`SUM`) je také provedena paralelně, a to tak že každý dotazovaný uzel vypočítá svou „podsumu“ ze svých vlastních dat, a až poté koordinační uzel sečte tyto „podsumy“ v sumu kterou poté vidí klient. Nakonec koordinační uzel samozřejmě zašle samotný výsledek dotazu zpět klientovi, interpretace pak už závisí spíše na klientské aplikaci.

Originálně byla místo prezentovaného dotazu zamýšlena otázka „Kolik lidí v Brně zemřelo v cyklonehodách potenciálně zapříčiněnými alkoholem od roku 2012?“, přičemž by přítomnost alkoholu byla odvozena z hledání podřetězce „alkohol“ ve sloupci `stav_ridic` obsahující slovní popis stavu řidiče. Při dosazení `AND stav_ridice LIKE '%alkohol%'` však docházelo k problémům jelikož Cassandra údajně není na hledání podřetězců dobře optimalizována, a jelikož nepomohlo ani vytvoření indexu nad tímto sloupcem, byl nápad nakonec zahozen.

# 4 Dopravní nehody

Pro dokumentovou databázi jsme vybrali dataset **Dopravní nehody**[8] ve formátu GeoJSON. Tuto část projektu vypracoval primárně xmatuf00.

## 4.1 Zvolený typ NoSQL databáze a datové sady

Tento dataset nám přišel vhodný uložit do dokumentové databáze. Použili jsme **MongoDB**[9], který je v daném ohledu nejpoužívanější a zároveň obsahuje rozsáhlou a detailní dokumentaci s mnohými příklady a způsoby využití. Mimo jiné existuje mnoho driverů (zejména `pymongo` v Pythonu, který jsme chtěli použít pro demonstraci) pro tento produkt.

Dataset obsahuje informace o dopravních nehodách, které se staly na území města Brna od roku 2010. Tyto data pocházejí od Policie České Republiky a je aktualizován jednou za rok. K říjnu 2023 se v datasetu nacházelo téměř 70 tisíc záznamů. Jelikož MongoDB se vyznačuje vysokou škálovatelností a flexibilitou uložení jednotlivých dat, zdálo se nám vhodné ho využít pro právě tento dataset.

Formát GeoJSON byl vybrán s ohledem na to, že MongoDB umožňuje vykonávat i dotazy na základě geografické pozice a to se dalo jednoduše udělat pomocí položky `"geometry"` v každém záznamu. Zároveň má tento formát blízko k struktuře dokumentů a je ve velké míře využíván pro potřeby – je to v podstatě JSON obohacen o geografické informace. Samotné záznamy o nehodách (položky v poli `"features"`) měli následující strukturu:

```
"type": "Feature",
"properties": { "key1": "string_value1", "key2": numeric_value2, ...},
"geometry": { "type": "Point", "coordinates": [ x_longitude, y_latitude ] }
```

V rámci `"properties"` každého prvku pole `"features"` se nachází 56 informací popisujících různé aspekty nehody, jako třeba:

- identifikátory (prvky `OBJECTID, id, GlobalID`),
- datum, čas a místo nehody (ulice, městská část),
- okolnosti nehody (viník, příčina, viditelnost, situování, alkohol),
- statistiky o škodách a obětech (hmotná škoda, lehké/těžká zranění, smrti),
- geografické informace (zeměpisné souřadnice, WGS84[10]),
- případně další data (okolnosti o chodcích jako účastnících nehody, informace o vozidle).

[8] https://data.gov.cz/datov%C3%A1-sada?iri=https%3A%2F%2Fdata.gov.cz%2Fzdroj%2Fdatov%C3%A9-sady%2F44992785%2Ff7604237598371dd478232df3ad93ce9

[9] https://www.mongodb.com/docs/manual/

[10] https://cs.wikipedia.org/wiki/World_Geodetic_System

## 4.2 Popis přípravy a uložení dat

Data byla stáhnutá z dané web-stránky a načtena do JSONu. Jako identifikátor dokumentu (hodnota `"_id"`) byla zvolena hodnota atributu `OBJECTID`, která značí pořadové číslo zaznamenané nehody v rámci datasetu (číslováno od 1 od roku 2010). Toto číslo nezodpovídá přesně pořadí, ve kterém se nehody staly, nicméně pro potřeby tohoto projektu je postačující, jelikož je v rámci datasetu unikátní a nikdy nechybí.

Bližší analýzou dat jsme zjistili, že data o chodcích jako účastnících nehody (jedná se o 6 datových hodnot) jsou úzce propojena. Buď jsou všechny `null` (více než 90 % případů), nebo většina z nich obsahuje. Proto byla vytvořena položka `"ucast_chodce"`, která je nastavena na `null`, jsou-li všechny relevantní atributy prázdné, nebo obsahuje JSON objekt (ve výsledku tedy vnořený dokument) s obsahem relevantních atributů (těch 6 datových hodnot), je-li nějaký z nich neprázdný.

Dále byly časová data z atributu `cas`, které byli zapsány ve vojenském formátu, vloženy do atributu `datum`, který je ve formátu ISO 8601[11], ale původně měl uložen jen datum, nikoliv čas. Tím se staly všechny ostatní atributy o datu nebo času redundantní, a proto se ve výsledném dokumentu nevyskytovali.

Potom se zmazaly všechny ostatní nadbytečné geografické a jiné informace (WGS84, duplikované zeměpisné souřadnice, atribut `"id_nehody"`, který byl vždy `null` a jiné zbytečné identifikátory), zůstal jen atribut `"location"`, který v sobě nesl JSON Point objekt z GeoJSON formátu, který se pak využil v geografických dotazech. Výsledná vyčištěna struktura (`accident`) byla vložena do databázové kolekce (`car_accidents`) příkazem:

```
db.car_accidents.updateOne(
  { "_id": accident["_id"] },
  { "$set": accident },
  { "upsert": true }
);
```

Při další inspekci dat se ukázalo, že mnohé atributy nabývají hodnoty z množin o max. 10-20 prvcích. Jelikož se jedná o dlouhé řetězcové hodnoty, které se opakují beze změny, bylo by výhodnější tyto hodnoty ukládat jako číselné a používat pomocné překladové tabulky, čím by se zamezila redundance/opakování dat.

Dalším optimalizačním aspektem by mohlo být využití vnořených dokumentů při ukládání okolností nehody. Některé nehody se totiž v databáze vyskytují vícenásobně (stejné souřadnice, stejný čas), a liší se jen v tom, z pohledu kterého účastníka se na nehodu dívá (např. jeden účastník nehody má vozidlu typu 1, druhý má vozidlo typu 2, případně byli vodiči v jiných věkových kategoriích). Pro porovnání je k říjnu 2023 v datasetu 69741 záznamů a jen 32882 z nich má jedinečnou kombinaci souřadnic a času, co pravděpodobně znamená vysokou míru duplikace dat. Řešením by bylo vytvořit vnořené dokumenty do dokumentu popisujícího 1 nehodu, které by popisovali situaci jednotlivých vodičů, případně jejich vozidel apod.

Tyto dvě optimalizace (ukládání polí nabývajících hodnot z nízkého počtu možností a vnořování dokumentů) už ale nejsou součástí demonstrace.

## 4.3 Dotazování nad daty

Níže se nachází agregační řetězec (nebo pipeline), který vrátí počty nehod, které se odehrály v okruhu 250m od náměstí Svobody (konkrétně od Brněnského orloje) v roce 2022 podle dní v týdnu, kdy nastaly:

```
[
    {$geoNear: {
      near: {type: "Point", coordinates: [16.6084, 49.1951]},
      distanceField: "distance",
      maxDistance: 250,
      spherical: true
    }},
    {$addFields: {
        year: { $year: { $dateFromString: { dateString: "$datum" } } },
        week_day: { $dayOfWeek: { $dateFromString: { dateString: "$datum" } } }
    }},
```

[11] https://en.wikipedia.org/wiki/ISO_8601

```
    {$match: {year: 2022}},
    {$group: {_id: "$week_day", total: { $sum: 1 }}},
    {$sort: {_id: 1}}
]
```

Fungování této agregační pipeline se vykonává ve vícero fázích:

1. Fáze `$geoNear` je první operací v této agregační pipeline. Provádí geoprostorový dotaz. MongoDB používá geoprostorové indexy k efektivnímu vyhledávání dokumentů v zadané geoprostorové oblasti. Tento index pracuje na základě hodnoty v `"geometry"`, který byl specifikován před spouštěním této pipeline.
   - `"near"` specifikuje referenční místo pro vyhledáváni geoprostorvých dat,
   - `"distanceField"` určuje název atributu, ve kterém se uloží spočtená vzdálenost, která se potom připojí do dokumentu s výsledky,
   - `"maxDistance"` – na základě této hodnoty se filtrují dokumenty, které jsou dostatečně blízko referenčnímu bodu (250 jednotek – v případě Point objektů GeoJSONu se jedná o metry),
   - `"spherical"` jen značí, že se pracuje s geografickými souřadnicemi na Zemi, a tedy se má zohledňovat i zakřivení Země (to by mělo význam zejména při dotazech na větší vzdálenosti apod.).

   Toto filtrování může být provedeno současně ve více uzlech nebo „shardech“ v závislosti na rozložení dat.
2. Ve fázi `$addFields` se ke každému dokumentu přidají pole `year` a `week_day`, které jsou extrahovány z hodnoty v `"datum"`, aby se pak na základě těchto hodnot mohli dál filtrovat a seskupovat dokumenty.
3. Fáze `$match` filtruje dokumenty z předchozího stupně a zachovává pouze ty, u nichž je hodnota `year` rovno 2022. Toto filtrování probíhá v každém uzlu nebo shardu.
4. Fáze `$group` seskupí filtrované dokumenty podle pole `week_day` a vypočítá celkový počet pro každou skupinu. Každý uzel nebo střep zpracovává svou část dat a počítá tato seskupení a celkové součty nezávisle.
5. Fáze `$sort` seřadí seskupené výsledky podle dní vzestupně (od pondělí do neděle). Toto třídění se provádí v každém uzlu nebo shardu pro jeho podmnožinu dat.
6. Doručení výsledků – po dokončení všech agregačních operací v různých uzlech nebo oddílech jsou výsledky sloučeny do jediné sady, seřazeny podle `"den"` a doručeny klientovi.
7. Využití dat uživatelem – na závěr jsou tyto data vložena do pandas datového rámce, číselné hodnoty dní jsou přepsány na slova a výsledek se může vypsat.

V dodaných skriptech jsou i další 2 dotazy na tuhle databázi. Jeden zjišťuje, kolik nehod se zúčastnili i chodci a seřadí do rozsahů podle materiálních škod. Poslední dotaz ukáže nehody, které se odehrály v období Vánoc (mezi 23. a 27. prosincem) a seřadí je sestupně podle datumu a času.

# 5 Základní školy - seznam a spádové oblasti

Pro grafovou databázi jsme vybrali 2 datasety: **Základní školy – seznam**[12] a **Základní školy – spádové oblasti**[13] ve formátu GeoJSON. Tuto část projektu řešil primárně xmatuf00.

[12] https://data.gov.cz/datov%C3%A1-sada?iri=https%3A%2F%2Fdata.gov.cz%2Fzdroj%2Fdatov%C3%A9-sady%2F44992785%2Fbc65d3bc29f688738fde342afd57ed49

[13] https://data.gov.cz/datov%C3%A1-sada?iri=https%3A%2F%2Fdata.gov.cz%2Fzdroj%2Fdatov%C3%A9-sady%2F44992785%2Fbcffa5f75361a584462ffc75340d970d

## 5.1 Zvolený typ NoSQL databáze a datové sady

Jako grafová databáze byla vybrána **Neo4j.** Má velice příjemnou a rozsáhlou dokumentaci[14], taky je nejvíc používaná a proto jsme s ní pracovali. Výběr vhodného datasetu pro grafovou databází byl poměrně náročný. Mnoho datasetů popisuje jenom jeden druh dat, tedy reprezentovatelný jednou entitou (nebo typem uzlů). Vzájemně byly často tyto datasety nepřepojitelné, popisovali totiž neslučitelné jevy (nebylo možné se nijak odkazovat mezi množinami popisovaných dat). Pro potřeby demonstrace grafové databáze jsme ale potřebovali využít data, kde se můžou nějakým způsobem vybudovat vztahy. Našli jsme dva datasety, které tuto podmínku splňovali a bylo tedy možné vytvořit mezi nimi nějakou vazbu. Dataset *Základní školy – seznam* obsahuje jen data o různých základních školách v Brně (celkem 64). Dataset *Základní školy – spádové oblasti* potom obsahuje různé adresy (přes 62 tisíc záznamů), které náleží do spádových oblastí škol z prvního datasetu. V každém záznamu o adrese jsou pak i pole popisující, které základné škole adresa „náleží“, zejména se jedná o pole `"kod_skoly"`, které je unikátní v rámci škol. Formát GeoJSON byl zvolen pro snadnější a pohodlnější čistění dat (taky jsme uložili data o geolokaci jednotlivých adres a škol).

Poznámka: První pokus zpracovávat data v grafové databáze se týkal datasetu o finančních tocích neziskovému sektoru ČR (různé příspěvky od ministerstev organizacím a pod.). Tyto data ale nebyli úplně konsistentní, dataset obsahoval spoustu záznamů s mnoha atributy (zdlouhavý import), které často měli prázdné hodnoty, a ne ve všech záznamech se dali identifikovat zasílatelé nebo částka příspěvku. Proto byl tento návrh nerealizován.

## 5.2 Popis přípravy a uložení dat

Podobně jako u ostatních typech, i tady se nejprv stáhne dataset, pokud se nenachází v složce se skripty. Před samotným vkládáním dat do databáze se vytvořili integritní omezení na unikátnost identifikátoru pro všechny druhy uzlů. V databázi jsme pracovali s 5 typy uzlů:

- `House` – obsahuje data o 1 adrese z datasetu o spádových oblastech.
- `School` – obsahuje data o 1 škole z datasetu o základních školách.
- `Municipality` – identifikováno polem `"cobce_kod"` (kód obce) v datech o 1 adrese z datasetu o spádových oblastech (podobně i v následujících 2 položkách).
- `TownPart` – identifikováno polem `"momc_kod"` (kód městské části).
- `Cadastre` – identifikováno polem `"katuze_kod` (kód katastrálního území).

Uzly typu `Municipality, TownPart, Cadastre` slouží podobně jako překladové tabulky pro snížení redundance dat. Mezi těmito 3 typy uzlů ale existují všude M:N vztahy, jelikož všechny tři typy členění města jsou různé a mají různé překryvy pro různé adresy a podobně. Proto jsme se dále nezabývali vytvářením síti vztahů mezi těmito územními celky, zároveň to pro účely demonstrace nebylo potřebné.

Příklad příkazu pro vytvoření integritního omezení pro unikátnost identifikátoru pro uzel typu `House` (podobně i pro ostatní typy uzlů):

```
CREATE CONSTRAINT IF NOT EXISTS FOR (h:House) REQUIRE (h.id) IS UNIQUE;
```

Pro všechny typy uzlů taky byly vytvořeny indexy na základě identifikátorů pro rychlejší vkládaní dat (i dotazování). Příkaz pro uzly typu `House`:

```
CREATE INDEX IF NOT EXISTS FOR (h:House) ON (h.id);
```

Načítaní dat z datasetu o základních školách byl poměrně přímočarý proces. Důležité je zmínit, že jako identifikátor školy se použil její kód (v rámci Brna se vždy jednalo o čtyřciferné číslo, například 2264). V datasetu o spádových oblastech se ale nacházeli nějaké nesrovnalosti. Některé domy (adresy) měli zřejmou chybu v kódu školy (měli o cifru navíc než jiná škola, která měla stejnou adresu i ostatní informace). Tyto nesrovnalosti se zjistili a při nahrávání dat do databáze se vkládali opravená a konzistentní data. Dále se odlišili adresy, které neměli jasnou spádovou oblast (kód školy byl z množiny $\{0, 1, 2, 3\}$ a chyběla adresa školy a další informace atd.). Pro tyto adresy byla vytvořena společná spádová oblast pomocí příkazu: `MERGE (s:School { id:0, name: "Undefined" });`.

[14] https://neo4j.com/docs/cypher-manual/current/introduction/

Jako první se vkládaly data z datasetu o základních školách, čím se vytvořili všechny uzly typu `School`. Je to důležité udělat jako první, zejména protože pak se při vkládání dat o jednotlivých adresách budeme snažit vytvářet vazby (hrany) mezi domem (nebo adresou) a školou, a budeme se chtít referencovat na identifikátor školy pomocí atributu `"kod_skoly"` v datasetu a adresách. Některé atributy jako název ulice, nebo orientační písmeno (u většiny adres) mohli nabývat prázdné hodnoty. To bylo vyřešeno pomocí operace `COALESCE`, která v takovém případě zvolí nějakou výchozí hodnotu (prázdný řetězec).

```
MERGE (s:School {
    id: $id,
    name: $name,
    street: COALESCE($street, ''),
    conscription_no: $conscription_no,
    orientation_no: $orientation_no,
    orientation_letter: COALESCE($orientation_letter, ''),
    postcode: $postcode,
    municipality: COALESCE($municipality, ''),
    x: $longitude,
    y: $latitude
});
```

Všechny hodnoty se znakem `$` před nimi jsou klíče do pomocného asociativního pole, které obsahuje přesné hodnoty atributů, které se vkládají do databáze. Stejný postup byl zvolen i pro načtení druhého datasetu. Spolu s dotazem se toto asociativní pole pošle jako transakce databázovému řadiči. Toto pomocné překladové pole zrychluje vykonání transakcí, než kdyby se to vkládalo přímo do řetězce dotazu (nebo Cypher příkazu na vkládaní dat).

Podobně se zpracovával dataset o spádových oblastech. Pro každý záznam byl vytvořen uzel obsahující informace o adrese. Potom se podle informace v poli `"kod_skoly"` v záznamech vytvořil vztah mezi zpracovávanou adresou a školou, kterou dané pole identifikuje. Na to slouží vztah `IN_CATCHMENT_AREA`. Následně byly na základě identifikátorů obce, městské části a katastrálního území vytvořeny i korespondující uzly. Avšak to jen za předpokladu, že daný identifikátor (nebo kód) nebyl nulový (v těch případech se jednalo o nedostatečně vyplněna data) a ještě daný uzemní celek není uložen v databáze. Pro tyto potřeby existují vztahy `IN_MUNICIPALITY`, `IN_TOWN_PART` a `IN_CADASTRE`.

Zde je Cypher příkaz, kterým byly načteny data, vytvořeny uzly `School, Municipality, TownPart, Cadastre` a vztahy mezi nimi.

```
MERGE (h:House {
    id: $id,
    code: $code,
    address: COALESCE($address, ''),
    street: COALESCE($street, ''),
    conscription_no: $conscription_no,
    orientation_no: $orientation_no,
    orientation_letter: COALESCE($orientation_letter, ''),
    postcode: $postcode,
    x: $longitude,
    y: $latitude
})

WITH h WHERE $school_id <> 0
MERGE (s:School {id: $school_id})
WITH h, s~MERGE (h)-[:IN_CATCHMENT_AREA]->(s)

WITH h WHERE $town_part_id <> 0
MERGE (t:TownPart {id: $town_part_id, name: COALESCE($town_part_name, '')})
WITH h, t MERGE (h)-[:IN_TOWN_PART]->(t)

WITH h WHERE $municipality_id <> 0
MERGE (m:Municipality {id: $municipality_id, name: COALESCE($municipality_name, '')})
```

```
WITH h, m MERGE (h)-[:IN_MUNICIPALITY]->(m)

WITH h WHERE $cadastre_id <> 0
MERGE (c:Cadastre {id: $cadastre_id, name: COALESCE($cadastre_name, '')})
MERGE (h)-[:IN_CADASTRE]->(c);
```

Uvažovali jsme i nad způsobem, jak provázat jednotlivé domy/adresy mezi sebou. Šlo by provázat domy se stejnou ulicí, nebo přibližně stejnými souřadnicemi mezi sebou. Způsob přes ulice by možná nebyl úplně funkční, jelikož se někdy mohou budovy nacházet na křižovatce a podobně, dále by nebylo možné provázat ulice mezi sebou a podobně. Pomocí souřadnic ale určitě existuje způsob (sice není v demonstraci ukázán, ale pro zajímavost ho popíšeme). Neo4j totiž poskytuje operátor `DISTANCE`, který umí počítat vzdálenost v metrech mezi 2 mezi dvěma poskytnutými souřadnicemi (které možno vytvořit provázáním pomocí funkce `POINT`).

```
distance(POINT({ x: h1.x, y: h1.y }), POINT({ x: h2.x, y: h2.y })) <= 100
```

Pomocí takové podmínky by šlo tedy (pro domy `h1` a `h2`) společně provázat adresy, kterých vzdálenost je menší než 100 metrů. Tohle ale nebylo ve skriptu demonstrováno, nicméně tato možnost je k dispozici, kdyby byla potřeba pracovat s daty nějakým komplexnějším způsobem, jako třeba identifikace hranic spádových oblastí, různé prostorové vizualizace spádových oblastí, překryvy a rozdíly mezi spádovými oblastmi a územným členěním.

## 5.3 Dotazování nad daty

Pro demonstraci byly vytvořeny 2 dotazy nad touhle databází. První dotaz zjistí 5 škol s největší spádovou oblastí (tj. je k nim přiřazeno nejvíc adres) a seřadí je podle velikosti sestupně. Druhý dotaz zjistí, které obce (případně části obce) mají nejvíc škol (vypíše jen prvních 5).

```
MATCH (s:School)
OPTIONAL MATCH (h:House)-[:IN_CATCHMENT_AREA]->(s)
WITH s, COUNT(h) AS catchmentArea
ORDER BY catchmentArea DESC
LIMIT 5 RETURN s, catchmentArea;

MATCH (h:House)-[:IN_CATCHMENT_AREA]->(s:School)
MATCH (h)-[:IN_MUNICIPALITY]->(m:Municipality)
WITH m, COLLECT(DISTINCT s) AS schools
RETURN m, SIZE(schools) AS numSchools
ORDER BY numSchools DESC
LIMIT 5;
```

Na druhém dotazu bude popsán způsob, jakým je zpracován databází Neo4j:

1. Neo4j začne vyhledávat hledat odpovídajíci uzly typu `House`, které jsou přepojeny s uzly typu `School` skrz vztah `IN_CATCHMENT_AREA` (v našem případě to jsou všechny domy/adresy z datasetu o spádových oblastech). Stejné vyhledávání proběhne i pro domy, které jsou spojené s částí obce (uzly `Municipality`) pomocí vztahu `IN_MUNICIPALITY` (to už nejsou všechny domy, jelikož v některých záznamech chyběli údaje o části obce, které náleží).

2. Dotaz může být paralelizován v závislosti od složitosti grafu skrz vícero vláken nebo serverových uzlů. Tento projekt byl vyvíjen a testován jen lokálně, takže se paralelizovalo jen na úrovni vláken.

3. z každého uzlu, který byl nalezen dotazem (tedy v podstatě jen ty domy, které mají přiřazenou část obce), se vytáhnou data. V případě tohoto dotazu se vybere seznam všech různých uzlů škol pro každý relevantní uzel části obce. Vyberou se všechny potřebná data z těchto uzlů, tedy zejména identifikátor a ostatní atributy.

4. Následuje agregační část. Všechny různé školy v rámci každé části obce se agregují do kolekce pomocí `COLLECT(DISTINCT s)`. Vypočítá se velikost této kolekce, čím se určí počet různých škol týkajících se každé městské části. Výsledky se seřadí sestupně podle této velikosti s pomocí `ORDER BY`.

5. Neo4j vytvoří výslednou množinu obsahující uzly typu `Municipality` a jejich korespondující počet různych škol (`numSchools`). Pomocí `LIMIT` je tato množina omezena jen na vrchních 5 výsledků.

6. Na závěr se výsledek zašle klientovi, který o něj žádal. Tento výsledek přijde ve strukturovaném formátu podobným JSONu, který je dál přehledněji formátován ve skriptu. Výsledek je vypsaný do standardního výstupu.

# 6 Kvalita ovzduší

Pro databázi časových řad jsme vybrali dataset obsahující data o kvalitě ovzduší [15] ve formátu GeoJSON.

## 6.1 Zvolený typ NoSQL databáze a datové sady

Pro tento dataset jsme se rozhodli použít databázi časových řad **InfluxDB**[16]. Hlavním faktorem, který nás přivedl k tomuto výběru, byla struktura jednotlivých dat, která odpovídá struktuře časové řady. Pro vložení dat do influxDB je ovšem prvně potřeba data převést do formátu line protocolu, který je koncipován tak, aby efektivně zpracovával velký objem dat v časových řadách.

Časová řada v line protocolu se skládá ze čtyř základních atributů. Prvním atributem je název měření, který označuje typ časové řady. Na základě tohoto atributu, lze v databázi rozlišovat různé měření. Název měření je typu řetězec, píše se bez uvozovek a je povinným prvkem. V našem případě jsme pro název měření zvolili "air_quality".

Druhým atributem jsou tagy, což jsou páry typu klíč-hodnota. Tagy jsou řetězce, které slouží ke kategorizaci dat měření a jsou nepovinné. Tagy a názvy měření jsou invertovaně indexované, což značně zrychluje dotazy, které se nad časovými řadami používají. Z našeho vybraného datasetu jsme z tohoto důvodu pro pole použili sloupce, které popisují odběrové stanice. Tyto sloupce nejsou tak variabilní, tudíž jsou vhodné pro identifikaci časové řady.

Třetím atributem je pole, které obsahuje samotná naměřená data. Opět se jedná o páry typu klíč-hodnota, ovšem tento atribut je povinný. Z našeho datasetu jsme pro tento atribut vybrali sloupce s naměřenými hodnotami složek ve vzduchu. Mimo jiné jsme do pole zařadili i sloupec `"globalID"`. Tento sloupec obsahuje unikátní hodnotu pro každý záznam, což by v případě zařazení mezi tagy způsobilo značné zpomalení celé databáze.

Posledním atributem je časová známka, která označuje čas vzniku dat v dané časové řadě. V případě, že při vkládání tento atribut není vyplněn, použije se aktuální čas. Každý záznam v naší datové sadě obsahuje atribut času, kdy byla daná data pořízena, tudíž tento problém nebylo potřeba nijak řešit.

Dalším faktorem při výběru tohoto datasetu bylo pravidelné aktualizování dat jednotlivými stanicemi. Stanice aktualizují hodnoty každou hodinu, každých 8 hodin nebo jednou denně. Díky tomuto faktu se v datové sadě nenachází duplicitní záznamy. Navíc Tento dataset obsahuje více než 50 000 záznamů, což představuje dostatečný objem dat pro demonstrování rychlosti, kterou InfluxDB dosahuje při provádění dotazů.

Konkrétní Python skript pro demonstraci práce s InfluxDB a tímto datasetem se nachází ve složce `influxdb/setup.py`. Tento skript pracuje s InfluxDB verzí 1.8.10 uvnitř Docker containeru. Důvodem vybrání této verze byla podobnost s verzí na referenčním virtuálním počítači.

### Popis přípravy a uložení dat

Datová sada je prvně stažena ve formátu GeoJSON, jelikož je to formát s kterým se snadně pracuje.

Následně jsou data načtena do asociativního pole. Jednotlivé záznamy se poté z tohoto pole převedou do line procotol formátu, kterým se data v databázi uloží v komprimované podobě. Aby nedocházelo ke zbytečné redundanci dat, byly z jednotlivých záznamů odstraněny duplicitní sloupce. V tomto případě se jednalo o sloupce `lat` a `lon`, které obsahovaly stejné hodnoty jako sloupce `x`, `y` (tyto duplicitní sloupce se nacházely jak v GeoJSON tak v csv formátu). Následně bylo nutné přetypovat datové typy některých sloupců, kvůli lepší konzistenci dat ve výsledné databázi. Konkrétně šlo o sloupce s nulovatelnými hodnotami složek vzduchu, které byly v původním datasetu uloženy jako řetězce, ale jejich hodnota odpovídala

[15] https://data.gov.cz/datov%C3%A1-sada?iri=https%3A%2F%2Fdata.gov.cz%2Fzdroj%2Fdatov%C3%A9-sady%2F44992785%2F4907e5c199620d99a44c1ec388703cd5

[16] https://docs.influxdata.com/influxdb/v1/query_language/

desetinnému číslu. Díky tomuto kroku bylo později možné nad všemi sloupci s hodnotami složek vzduchu dělat agregační dotazy.[17]

Data jsou po upravení jejich typů postupně vkládána do databáze v následujícím formátu dle dřívějšího popisu rozdělení původních dat do jednotlivých atributů:

```
"measurement": "air_quality",
"tags": {"key": "value"},
"time": "value",
"fields": {"key": "value"}
```

Pro vložení dat bylo využito základní přednastavené politiky, která určuje dobu po kterou jsou data uložena a také hodnotu replikačního faktoru. Hodnota replikačního faktoru značí počet replikací dat na více uzlech v klastru, což zajišťuje, že data jsou dostupná i v případě selhání jednoho nebo více uzlů. Pokud by časová řada, která se má vložit do databáze, již existovala (s totožnými hodnotami tagů a časovou známkou), byly by hodnoty polí tohoto měření přepsány.

## 6.2 Dotazování nad daty

Pro demonstraci dotazů byl zvolen následující dotaz. Jakou průměrnou hladinu no2 měly za rok 2022 jednotlivé stanice. Ukázka zápisu tohoto dotazu je následující:

```
SELECT MEAN("no2_1h") FROM "air_quality"
WHERE time >= '2022-01-01T00:00:00Z' AND time <= '2022-12-31T23:23:59Z'
GROUP BY "name"
```

Dotaz je dále zpracován následovně. Klient na začátku zašle dotaz na server pomocí HTTP protokolu. Databáze nejprve určí typ dotazu. V našem případě se jedná o dotaz s výrazem `SELECT MEAN("no2\_1h")`. Databáze dále určí časový rozsah, který se použije pro filtrování dat. Poté databáze určí, na které uzly je třeba přistoupit na základě časového rozsahu. Dále se podívá se na seznam měření a časový rámec, aby identifikovala relevantní uzly pokrývající zadaný časový rozsah (od 1. ledna 2022 do 31. prosince 2022). Poté databáze ověří, zda je dotaz sémanticky správný. Dotazový řadič dále směřuje úložný řadič k vytvoření iterátorů pro každý uzel, ke kterému je třeba přistupovat. Tyto iterátory jsou dále zodpovědné za čtení a filtrování dat z uzlů. Jelikož dotaz specifikuje agregaci `GROUP BY "name"`. Databáze data seskupí podle hodnot v značce `"name"` a pro každou skupinu vypočítá průměr pole `"no2\_1h"`. Následuje sloučení výsledků z různých uzlů, provedení agregace (v tomto případě výpočet průměru) a vytvoření výsledného souboru výsledků.[18]

[17] https://www.influxdata.com/blog/influxdb-internals-101-part-one/
[18] https://www.influxdata.com/blog/influxdb-internals-101-part-two/