VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

ANALÝZA DATABÁZÍ VHODNÝCH DO PROSTŘEDÍ IOT

THE ANALYSIS OF DATABASES SUITABLE FOR IOT ENVIRONMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PETR KOHOUT**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Ing. JAN KOŘENEK, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kohout Petr**
Program: Informační technologie
Název: **Analýza databází vhodných do prostředí IoT**
The Analysis of Databases Suitable for IoT Environment
Kategorie: Databáze

Zadání:

1. Vyberte a nastudujte databáze, které podporují ukládaní velikého množství malých dat v čase tak, jak tomu je v IoT systémech. Zaměřte se na open-source systémy.
2. Proveďte analýzu jednotlivých řešení s ohledem na výkon a požadované systémové prostředky.
3. Vybrané řešení implementujte jako rozšíření do systému používanému na projektu IoTCloud.
4. Otestujte parametry vašeho řešení s ohledem na původní stav před použitím vybrané databáze.
5. Diskutujte dosažené výsledky a možnosti pokračování práce.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Kořenek Jan, doc. Ing., Ph.D.**
Konzultant: Korček Pavol, Ing., Ph.D., UPSY FIT VUT
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 26. října 2018

Abstrakt

V posledních letech výrazně narůstá objem ukládaných dat v aplikacích všech typů. Se vzrůstajícím objemem dat rostou i požadavky na databáze. Klasické relačních databáze přestávají vyhovovat, a proto se přechází na nová, nerelační úložiště (tzv. NoSQL). V rámci této práce byla provedena analýza NoSQL databází zaměřujících se na zpracování velkého objemu dat, porovnání jejich výkonností a systémových nároků. Dále proběhla specifikace požadavků na nové úložiště senzorických dat pro projekt IoTCloud, implementace ovladače pro databázi Cassandra a integrace do projektu IoTCloud.

Abstract

In recent years, the volume of data stored in applications of all types has increased. As the volume of data grows, the demands on databases increase. The classic relational databases are no longer suitable. Therefore, it is switching to new, non–relational storage (called NoSQL). During this work, NoSQL databases focusing on processing large data volumes were analyzed and compared for their performance and system requirements. In addition, requirements for new sensor data storage for IoTCloud project were specified. Driver for database Cassandra was implemented and integrated into the IoTCLoud project.

Klíčová slova

Databáze, IotCloud, BeeeOn, PostgreSQL, Cassandra, MongoDB, Redis, Neo4j, NoSQL, SQL

Keywords

Database, IotCloud, BeeeOn, PostgreSQL, Cassandra, MongoDB, Redis, Neo4j, NoSQL, SQL

Citace

KOHOUT, Petr. *Analýza databází vhodných do prostředí IoT*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jan Kořenek, Ph.D.

Analýza databází vhodných do prostředí IoT

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Jana Kořenka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Petr Kohout
14. května 2019

Poděkování

Za konzultace a odborné vedení v praktické části bych rád poděkoval Ing. Janu Viktorinovi. Za odborné vedení a konzultace v průběhu psaní teoretické části bych rád poděkoval Doc. Ing. Janu Kořenkovi, Ph.D. a Ing. Pavlovi Korčekovi, Ph.D.

Obsah

1 Úvod **2**

2 Databáze SQL **4**
- 2.1 Relační databáze . . . 4
  - 2.1.1 Integritní omezení . . . 5
  - 2.1.2 Transakce . . . 6
  - 2.1.3 Databázové soubory . . . 7
  - 2.1.4 Indexování . . . 10
  - 2.1.5 Škálovatelnost . . . 10
  - 2.1.6 PostgreSQL . . . 11
  - 2.1.7 Shrnutí vlastností relačních databází . . . 11

3 NoSQL **13**
- 3.1 CAP teorém . . . 13
- 3.2 Občasná konzistence . . . 14
- 3.3 Distribuce . . . 16
- 3.4 Členění NoSQL databází . . . 17
  - 3.4.1 Databáze typu klíč–hodnota . . . 18
  - 3.4.2 Dokumentové databáze . . . 20
  - 3.4.3 Sloupcové databáze . . . 22
  - 3.4.4 Grafové databáze . . . 25
- 3.5 Shrnutí vlastností NoSQL databází . . . 26

4 Projekt IoTCloud **27**
- 4.1 Architektura projektu . . . 27

5 Analýza možností urychlení práce s daty **29**

6 Porovnání databází **30**

7 Ovladač pro databázi Cassandra **36**

8 Integrace databáze Cassandra **38**

9 Závěr **41**

Literatura **42**

Kapitola 1

Úvod

S rozvojem lidské společnosti vznikla potřeba uchovávat záznamy o jejím stavu. Z počátku se uchovávaly informace, které byly nezbytně nutné pro fungování státní správy. Jednalo se například o záznamy výběrů daní, evidenci stavu obyvatelstva, či událostí, které se za dané období staly (kroniky). S postupným růstem společnosti a její specializací se začalo uchovávat čím dál víc specializovaných dat, jako jsou stavy vodních hladin, či počasí. S příchodem novověku se jich hlavně ve státní správě začínají uchovávat velká množství a objevují se první problémy s jejich skladováním a tříděním. Po nástupu informačních technologií se přechází k digitalizaci dat a jejich uchovávání v datových skladech. Systémy pro skladování dat se nazývají databáze.

Předchůdcem dnešních počítačových databází byly kartotéky, které v současnosti můžeme pozorovat v některých lékařských ordinacích. Jedná se o soubory dat skladovaných v tištěné formě. Data jsou řazena a později vyhledána podle specifikovaných kritérií, jako je například příjmení pacienta. Podobný princip řazení a vyhledávání používají i dnešní databázové systémy.

Přelomovým obdobím je polovina 20. století, kdy začal rozvoj počítačů. Pro nízkou efektivitu strojového kódu procesorů pro operace prováděných nad databází vznikl požadavek pro vývoj vyššího jazyka, který by tato data zpracoval. Prvním jazykem byl COBOL, který se rychle rozšířil a stal se nejpoužívanějším jazykem pro hromadné zpracování dat. Od poloviny šedesátých let začínají vznikat systémy, které provádějí operace nad daty. Jedná se o tzv. systém řízení báze dat (SŘBD).

Roku 1971 byla popsána celá architektura sítového databázového systému a byly definovány pojmy jako je *schéma databáze*, *jazyk pro definici schématu*, *subschéma* a další. První relační databáze byly zveřejněny roku 1970 a o čtyři roky byla vydána první verze dotazovacího jazyka SQL. Devadesátá léta minulého století jsou spjaty s vznikem a vývojem objektových a objektově-relačních databází [19].

V druhém desetiletí 21. století se rozvijí tzv. NoSQL databáze, které slouží k zpracování velkého objemu dat.

Databáze jsou dnes všeobecně známý pojem. Používají se k uchování milionů záznamů. V současné době se výrazně zvyšuje objem sbíraných dat v kratším intervalu, čímž rostou nároky na rychlost jejich zpracování. Tato data jsou produkována velkým množstvím zařízení v krátkých intervalech a setkáváme se u nich s vysokou redundancí. Příkladem může být sběr dat ze senzoru, který měří teplotu v místnosti. Při každém zápisu nepotřebujeme ukládat, z kterého senzoru byla data získána. Tento přístup by zvolily tradiční relační systémy pro správu databáze, které byly navrženy za účelem zachytit vztahy mezi objekty reálného světa a umožnit nad nimi jednoduché a velice různorodé dotazování. Práce nad

těmito daty je velice náročná na systémové prostředky a na čas, protože každá vstupní data podléhají procesu kontroly integritních omezení. Kontroluje se správná struktura dat, unikátnost primárního klíče a existence případných cizích klíčů. Při sběru dat v IoT (Internet of Things) se začínáme setkávat s jevem, kdy relační databáze nestíhají zpracovávat množství záznamů v intervalech, ve kterých jsou produkována. Tato data jsou často nestrukturovaná a při vzniku aplikace se velice obtížně definuje jejich podoba. Z tohoto důvodu vznikly nové databázové systémy, které se zaměřují na zpracování velkého objemu dat, ale neumožňují nad nimi provádět tolik operací. NoSQL databáze neprovádí složité vstupní kontroly a umožňují ukládat data v tzv. surovém (raw) formátu, jehož struktura je předem neznámá. Další výhodou je možnost vytváření tzv. clusterů. Cluster je množina strojů v síti, které společně spolupracují a umožňují tak distribuovat zátěž v systému.

V práci se zaměřím na databáze SQL a NoSQL. Obě skupiny popíši a uvedu jejich základní principy, na kterých pracují. Jejich vlastnosti budu popisovat s cílem využití v rámci IoT. Provedu měření výkonnosti a paměťových nároků jednotlivých databázových systémů, okomentuji výsledky a vyberu vhodného zástupce pro oblast ukládání dat v projektu IoTCloud. Dále specifikuji systém, pro který bylo rozšíření vytvořeno a integraci do toho systému.

Kapitola 2

Databáze SQL

Tento termín označuje skupinu databází, které pro dotazování používají jazyk SQL. V této práci se budu věnovat pouze relačním SQL databázím, které jsou v současnosti nejrozšířenější.

2.1 Relační databáze

Tento druh databáze pracuje s relacemi. Data jsou uložena ve struktuře typu tabulka, která sice neodpovídá přesnému matematickému vyjadřování, ale v rámci databáze pojem tabulka reprezentuje implementaci pojmu relace. Zachycení vztahů mezi daty v databázi je realizováno opět pomocí relace. Celá kapitola o relačních databázích čerpá ze studijní opory pro předmět Databázové systémy (IDS) [20].

Definition 1 *"Nechť $D_1, D_2, ..., D_n$ jsou množiny atomických hodnot označované jako domény. Relace (databázová) na doménách $D_1, D_2, ..., D_n$ je dvojice $R = (R, R\star)$, kde $R = R(A_1 : D_1, A_2 : D_2, ...A_n : D_n)$ je schéma relace, kde $A_i(A_i \neq A_j$ pro $i \neq j)$ značí jméno atributu definovaného na doméně Di, a $R* \subseteq D_1 \times D_1 \times ... \times D_n$ je tělo relace. Počet atributů n relace se označuje stupeň (řád) relace, kardinalita těla relace $m =\mid R\star \mid$ se označuje kardinalita relace."([20] strana 10)*

```
gateways := (id, name)
devices := (id, name, gateway)
```

Příklad zápisu relace [20].

Relace `gateways` je definována jako dvojice, která zachycuje vztah dvou prvků a `devices` je relací definovanou jako trojice, která zachycuje vztah tří prvků, přičemž `gateway` je prvek z relace `gateways`.

Pro zachycení pojmu relace se data v relačních databázích ukládají prostřednictvím tabulek. Tabulku (např. 2.1) můžeme rozdělit na dvě části – záhlaví a tělo tabulky. Záhlaví tabulky se označuje jako *schéma relace* a tělo tabulky odpovídá matematickému pojmu relace.

Každá relace je určena pomocí schématu, které se skládá z domén atributů. Relační model klade na obsah domén přísné omezení. Domény (dnes označovány jako *datový typ*) mohou obsahovat pouze atomické (skalární) hodnoty, což znamená, že hodnota atributu musí z hlediska významu zastupovat nedělitelný celek. Příklad si můžeme uvést na obrázku 2.1, kde tabulka obsahuje dva atributy – `prijmeni` a `jmeno`. Pokud bychom tyto

id_zamestnance	prijmeni	jmeno	plat	datum_nastupu
1	Novák	Adam	32000	2013-05-02
2	Nová	Jana	45000	2013-06-11

Obrázek 2.1: Relační databáze [9].

atributy sloučili do jednoho atributu (např. `cele_jmeno`), řekli bychom, že celé jméno nerozlišujeme na základě příjmení, či křestního jména. Pokud bychom celé jméno rozlišovali na základě příjmení, či jména, jedná se o atribut *složený*. Dále atribut definuje jakého datového typu bude jeho hodnota nabývat a přes jaký identifikátor se k nim bude přistupovat. Jako příklad si můžeme uvést atribut `plat`, který je určen celočíselným datovým typem.

2.1.1 Integritní omezení

Data jsou uložena v jednotlivých tabulkách. Protože obvykle zachycují stavy objektů reálného světa (jeden objekt odpovídá jednomu záznamu v tabulce – např. student Jan Novák, tabulka 2.1), je nutné také zachytit vztahy, které mezi nimi existují (žák chodí do třídy). U systémů, které předcházely relačním databázím, se tento problém řešil pomocí ukazatelů. Značnou nevýhodou byla závislost na cestě určené ukazatelem. V relačním modelu se k problematice vztahů mezi záznamy v tabulce přistupuje jiným způsobem, který navrhl E. F. Codd. Pro zachycení vztahu *žák chodí do třídy* je nutné tato data uložit do databáze. V předrelačních databázích vznikne ukazatel, který definuje fyzickou vazbu mezi záznamy. V relačním modelu se tato vazba vytváří jako logická, kdy je požadována shoda hodnot záznamů z odkazované a odkazující tabulky. Pro zachycení vztahu byl v tabulce *žák* vytvořen nový atribut `Třída`, který obsahuje hodnotu atributu `Třída` ze stejnojmenného atributu tabulky *třída* [20].

Jméno	Příjmení	Datum_narození	Třída
Petr	Homolka	2002-06-07	4.F
Jan	Novák	2001-05-11	4.F

Tabulka 2.1: Tabulka: žák

Třída	Patro	Třídní_učitel
4.F	4	Lasička Pavel
5.G	2	Beneš Tomáš

Tabulka 2.2: Tabulka: třída

Pro možnost vytvoření logické vazby mezi záznamy je zapotřebí, aby byly jednotlivé záznamy jednoznačně identifikovatelné. Za tímto účelem je nutné v každé tabulce definovat tzv. *primární klíč*, který je vybrán z množiny *kandidátních klíčů*, a v odkazující se tabulce je nutné vytvořit *cizí klíč*, který slouží k odkazování na záznam v odkazované tabulce. Pro kandidátní klíče musí platit dvě časově nezávislé vlastnosti [20].

- Kandidátní klíč je unikátní. Tedy v relaci se nevyskytuje shodná dvojice kandidátních klíčů [20].
- Kandidátní klíč je minimální (neredukovatelný) a pokud je složen z více atributů, tak nelze vynechat ani jednu jeho složku, protože jinak by přestal být unikátní [20].

Primární klíč je vybraný kandidátní klíč a slouží v rámci databáze k jednoznačné identifikaci záznamu [20]. Z tabulky *třída* 2.2 primárním klíčem označíme sloupec `Třída`, zatímco v tabulce 2.1 za primární klíč označíme trojici – `Jméno`, `Příjmení` a `Datum_narození` za předpokladu, že školu nenavštěvují dva žáci se stejným křestním jménem, příjmením a datem narození.

Atribut relace **R2** je cizím klíčem, právě tehdy když splňuje časově nezávislé podmínky.

- *"Každá hodnota cizího klíče je plně zadaná nebo plně nezadaná"* ([20] strana 17).
- *"Existuje relace **R1** s kandidátním klíčem takovým, že každá zadaná hodnota cizího klíče je identická s hodnotou kandidátního klíče nějaké n-tice relace **R1**"* ([20] strana 17).

V tabulce *žáci* jako cizí klíč označíme sloupec `Třída`, který obsahuje primární klíč z tabulky *třída* 2.2.

2.1.2 Transakce

Za databázovou transakci označujeme jednotku operací, kterou provádíme při práci v databázi. Abychom zajistili integritu dat v databázi, musíme zajistit vlastnosti operací nad daty, které při detekci chyby jedné dílčí operace zaručí, že databáze bude vrácena do původního (konzistentního) stavu. SŘBD musí disponovat dvěma důležitými vlastnostmi – zotavení po poruše a řízení souběžného přístupu.

Relační databáze pracuje s daty na principu **ACID**. Slovo ACID vzniklo spojením prvních písmen slov **A**tomicity, **C**onsistency, **I**solation a **D**urability, která popisují vlastnosti relační databáze [20].

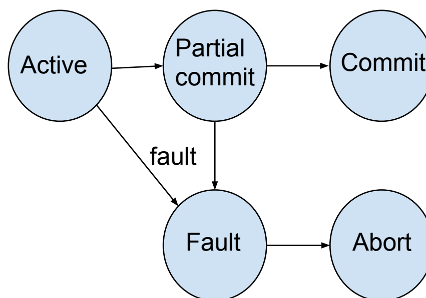
Atomicity (atomičnost) je vlastnost, která popisuje práci s operacemi jako s dále nedělitelnými. Pokud jakákoli část operace selže, operace se přestane provádět a databáze se vrátí do původního (konzistentního) stavu před provedením operace. Příkladem může být transakce, která provádí převod částky z jednoho účtu na druhý. Atomičnost zaručuje, že při vzniku chyby během provádění transakce (výpadek napájení atd.) ji SŘBD během procesu zotavení po poruše detekuje a následně opraví. V tomto případě transakce musí platit, že součet stavu účtů před a po provedení transakce bude shodný. Pokud se tyto hodnoty liší je detekována chyba a stav je vrácen do původního stavu před transakcí.

Consistency (konzistence) je vlastnost, která zaručuje, že databáze se v jakýkoli čas nachází v konzistentním neboli validním stavu. Za zajištění této vlastnosti zodpovídá programátor aplikace, který určuje chování aplikace.

Isolation (izolace) je vlastnost popisující izolovanost jednotlivých operací. Pokud v jeden čas na jeden řádek (entitu) míří více dotazů na zápis, jsou tyto operace z pohledu systému řízení báze dat vykonány postupně (v pořadí jejich příchodu) jako ve frontě.

Durability (trvalost) zaručuje okamžité zapsání dat na pevný disk, aby byla chráněna před ztrátou, např. při přerušení napájení. Je zaručeno uchování stavu databáze těsně před výpadkem, tedy v posledním konzistentním (validním) stavu [20].

Existují dva způsoby, kterými může databázová transakce skončit. Transakce může být úspěšně dokončena nebo zrušena. Rušení transakce se provádí ve dvou situacích. První situací je porušení podmínky, kterou stanovil programátor, takže k zrušení transakce dojde v ošetření nějaké chyby, které vznikla za běhu transakce. Druhou situací je zrušení transakce ze strany SŘBD, kdy chybu vzniklou při běhu transakce (výpadek napětí) ošetřuje systém a ne aplikace. Transakce prochází v průběhu zpracování některými stavy, které můžeme vidět na obrázku 2.2.



Obrázek 2.2: Stavy transakce [20].

Počáteční stav je reprezentovaný stavem *active* (aktivní) a odpovídá začátku zpracování transakce. Poté se transakce stává *partial commit* (částečně potvrzenou), což znamená, že byly uskutečněny všechny transakční operace, ale změny způsobené provedením transakce ještě nebyly spolehlivě uloženy na pevný disk, aby byla zaručena vlastnost trvalosti. Po dokončení těchto operací je transakce převedena do stavu *commit* (potvrzení).

Při detekci chyby ze strany aplikace, či SŘBD je transakce ze stavu *active*, či *partial commit* převedena do stavu *fault* (chybový stav), kdy je provedeno zrušení transakce pomocí činnosti označované jako *rollback* [20].

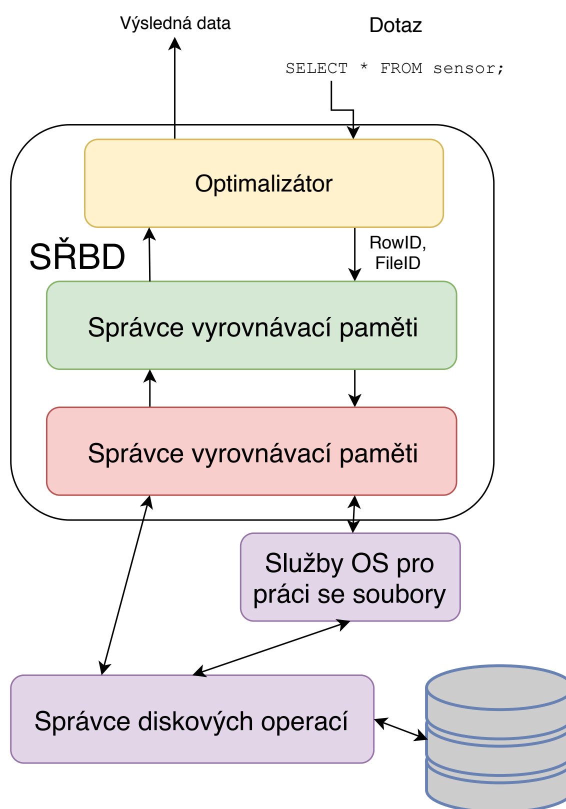
2.1.3 Databázové soubory

Na logické úrovni jsou data uspořádána do tabulek, které se fyzicky zapisují na disk do tzv. *databázových souborů*. Databázové soubory jsou umístěny na nevolatilních úložištích, které v současnosti tvoří především magnetický disk. Operace čtení a zápisu z tohoto typu paměti je časově náročné, a proto databázové systémy využívají vyrovnávací (volatilní) paměť, která zrychluje čtecí a zapisovací operace, ale umožňuje uchovávat pouze omezené množství dat (v rámci MB až GB).

Při dotazování databáze prochází námi zformulovaný dotaz zpracováním na několika úrovních. Při přijetí dotazu SŘBD nejdříve provede syntaktickou a sémantickou kontrolu příkazu. Poté je příkaz předán *optimalizátoru*, který provede analýzu dotazu a určí nejvýhodnější posloupnost operací pro dosažení výsledků. Příkladem může být sekvenční prohledávání záznamů, které patří k nejjednodušším, ale i k nejméně efektivním. Dnes často používanými metodami pro vyhledávání záznamů jsou *indexování* a *hašování* (viz 2.1.4). Výstupem tohoto prohledávání je *identifikátor řádku* (RowID) a případně i identifikátor souboru (FileID).

Identifikátor řádku a souboru je předán k zpracování další vrstvě, která se označuje jako *správce vyrovnávací paměti*. Správce vyrovnávací paměti pracuje se stránkami ve volatilní paměti, kde jedna stránka odpovídá jednomu diskovému bloku. Nejprve se provede ověření, zda potřebná data jsou dostupná ve vyrovnávací paměti. Při úspěchu jsou tato data vrácena vyšším vrstvám SŘBD. Pokud data nejsou na této úrovni dostupná, požadavek se deleguje na *správce diskového prostoru*.

Správce diskového prostoru může pro přístup k datům využít služeb operačního systému pro práci se soubory nebo nízkoúrovňových služeb pro práci s diskem. Tento prvek SŘBD zodpovídá za výběr správného diskového bloku na základě RowID a FileID. Po nalezení příslušného bloku je tento blok předán správci vyrovnávací paměti, který si jej uloží ve formě stránky. Dále jsou data zpřístupněna SŘBD pro zpracování na logické úrovni (relační operace nad daty) [20].



Obrázek 2.3: Postup SŘBD při reakci na dotaz [20].

Při práci s jednotlivými bloky disku je žádoucí, abychom při načtení jednoho bloku získali co největší objem požadovaných dat. Data mohou být pevné, či proměnné délky, takže je zapotřebí zvolit nejvhodnější způsob, jak data reprezentovat v databázových souborech. Existují dva způsoby – v rámci jednoho souboru ukládat záznamy stejné délky nebo umožnit ukládání záznamů proměnné délky s nutností rozšířit soubor a části obsahující informace o jednotlivých záznamech.

V rámci práce se záznamy pevné délky se jednotlivé záznamy ukládají postupně za sebe a je nutné se zabývat dvěma problémy. Prvním problémem je naložení s místem, které vznikne po jejich smazání. Jednou z možností je jejich reorganizace tak, aby vzniklá mezera

byla zaplněna. Tento přístup se v současnosti příliš nepoužívá, protože přesuny dat vyžadují, z pohledu SŘBD, drahé operace v podobě přístupů na disk a změně hodnot odkazů na paměťový prostor, který bývá v rámci databáze uložen na více místech (např. v indexech). Z tohoto důvodu se upřednostňuje ponechání volného místa s možností zaplnění při zápisu nového záznamu, kdy se pro uchování informací o volném prostoru používá datová struktura seznam. Druhým problémem je rozložení jednoho záznamu do dvou diskových bloků.

Při práci se záznamy různé délky můžeme ukládat datové typy s proměnlivou délkou (VARCHAR, BLOB, CLOB v jazyce SQL) nebo v rámci jednoho souboru ukládat záznamy různých typů, které mají mezi sebou nějaký vztah a je tak pravděpodobné, že se s nimi bude pracovat současně. Jedná se o tzv. *shlukování* (*clustering*).

Prvním způsobem jak ukládat tyto záznamy je pomocí *řetězce bytů*. Záznamy se k sobě shlukují a ukládají se za sebe s následujícím ukončovacím řetězcem, který označuje konec jejich sekvence. Problémy, se kterými se zavedením tohoto způsobu zpracování musí pracovat, jsou opětovné použití uvolněného místa po smazání a nulový prostor pro expanzi záznamu. Při nárůstu paměťových nároků u záznamů je zapotřebí alokovat nový a větší prostor. Do tohoto prostoru přesunout původní záznam rozšířený o nová data a následně původní paměťový prostor uvolnit. Proto se v současnosti více využívá modifikace, která používá záhlaví diskového bloku pro uložení informací o datech v něm obsažených. Obsahuje ukazatel na začátek a konec záznamů, počet záznamů v bloku a pole obsahující prvky s ukazatelem na záznam a s jeho délkou. Výhodou této modifikace je práce v rámci jednoho diskového bloku, takže jednotlivé přístupy na disk nevyžadují tolik režie. Odkaz na fyzický adresový prostor je zprostředkován pomocí prvku v poli, který se nachází v záhlaví diskového bloku a není nutné měnit ukazatele v rámci celého SŘBD.

Dalším způsobem je převedení záznamů proměnlivé délky na záznamy pevné délky a následné využití seznamu. Při přetečení dat v prostoru vymezeném pro záznam se do seznamu, ve kterém se prvek vyskytuje, vytvoří nový záznam, který bude obsahovat pouze nová data. Nevýhodou tohoto přístupu je nevyužití volného prostoru v rámci prvků v seznamu. Pro odstranění této nevýhody lze použít přístup, kdy se do jednotlivých bloků ukládají záznamy stejného typu. Bloky se označují jako *základní* a *přetokové*. V základním bloku jsou obsaženy záznamy pevné délky, které se při přetečení odkazují na seznam prvků v přetokovém bloku. Tento přístup umožňuje stanovit pevnou délku záznamu v rámci bloku, ale různou délku záznamu v rámci databázového souboru a využít tak maximální množství paměťového prostoru.

Z pohledu na databázový soubor jako celek rozlišujeme databázové soubory podle toho, jak jsou v nich jednotlivé záznamy rozloženy. Jsme schopni klasifikovat tři základní typy souborů.

Prvním druhem je tzv. *neuspořádaný soubor (heap file)*, ve kterém jsou záznamy libovolně rozprostřeny. Většinou je tento druh souboru použit pro ukládání záznamů z jedné tabulky.

Dalším typem souboru je *sekvenční soubor (sequential file)*. Záznamy jsou uloženy v uspořádaném seznamu podle hodnot tzv. *vyhledávacího klíče*. Při vkládání nových, či rozšiřování stávajících prvků se uplatňuje postup při zaplňování bloků jako je popsáno výše (přetokové bloky 2.1.3). Při velkém množství záznamů v přetokových blocích je zapotřebí provést reorganizaci souboru a uspořádat fyzicky záznamy za sebe podle vyhledávacího klíče, protože přístupy do přetokových bloků snižují efektivnost.

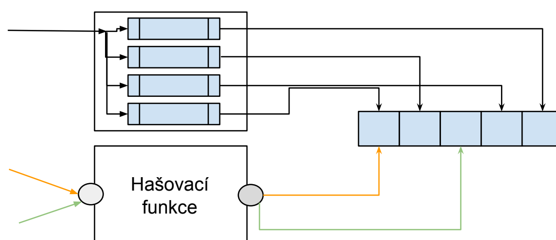
Posledním typem souboru je *hašovací soubor (hashed file)*. V rámci tohoto souboru je každý záznam umístěn do bloku, který je určen hodnotou hašovací funkce pro *hašovací klíč* [20].

2.1.4 Indexování

Při práci s daty v databázi je jejich modifikace, přidávání, či mazání pouze menší zastoupením všech operací, které SŘBD provádí. Z předchozích kapitol vyplývá 2.1.3, že kromě samotných dat, která do databáze ukládáme, jsou tato data obalena tzv. metadaty, které slouží k určení sémantiky těchto dat, či následnému řazení a vyhledávání.

Data jsou uložena v souborech. První možností, která by určitě fungovala, je lineární vyhledávání v souboru, při kterém bychom postupně iterovali nad všemi prvky souboru. Tento přístup je však velmi neefektivní a v databázových systémech se nepoužívá. K přistupování k jednotlivým záznamů se využívá index. Princip funkce indexu v databázových systémech je obdobný s principem, se kterým se setkáme v knihách. Index je část knihy, ve které najdeme námi hledaný pojem a pomocí čísla strany dohledáme informace k tomuto pojmu.

V SŘBD se setkáme s dvěma druhy indexu. Prvním druhem je tzv. *uspořádaný index*, který pracuje na stejném principu, jako index v knize. Pomocí hodnoty indexu se vyhledá odkaz na místo, kde se vyskytují data. Druhým druhem je *hašovaný index*. Tento druh indexu pracuje na základě tzv. hašovací funkce, která na základě hodnoty indexu vygeneruje odkaz na data v souboru [20].



Obrázek 2.4: Uspořádaný a hašovaný index.

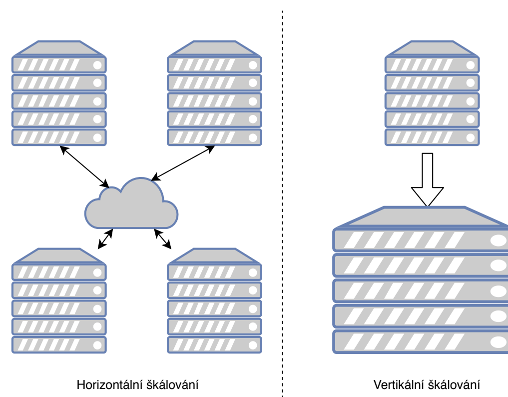
Soubor, ve kterém jsou uloženy data záznamů se označuje *primárním souborem* a sloupec, který slouží pro vyhledání záznamu se označuje *vyhledávací klíč*. Vyhledávací klíč se dělí na dva druhy a to primární a sekundární vyhledávací klíč. Podle primárního vyhledávacího klíče jsou řazeny záznamy v primárním souboru [20].

2.1.5 Škálovatelnost

Databáze pojímají velké množství dat, a proto je možnost jejich rozšíření velice žádanou vlastností. Rozlišujeme dva druhy škálovatelnosti – horizontální a vertikální. Vertikální škálování pracuje principu, kdy se do systému nepřidávají nové prvky, ale navyšuje se výkon stávajících. Jedná se tedy o kvalitativní změnu. Tato škálovatelnost je značně omezená. Hardware je výkonnostně limitován a jeho vylepšování je finančně náročné.

Horizontální škálování reguluje výkon systému na základě přidávání, či odebírání prvků ze systému. Jedná se o kvantitativní změnu. Zátěž se distribuje na více zařízení, které tak nevyžadují špičkové hardwarové vybavení, ale je zapotřebí zajistit jejich synchronizaci. Horizontálním škálování systému se stává velice náročné na splnění požadavků konzistence dat. Uzly často obsluhují jinou oblast sítě, ale musí zajistit dostupnost všech dat. Na jednotlivých

uzlech mohou existovat repliky záznamů, které nejsou v době aktualizace replikovaných dat na jiném uzlu konzistentní a jejich aktualizace se projeví až s časovým odstupem [11].



Obrázek 2.5: Horizontální a vertikální škálování.

2.1.6 PostgreSQL

Jedním ze zástupců relačních databází je PostgreSQL. PostgreSQL je objektově-relační databázový systém. Je vydán pod volně šiřitelnou licencí a na jeho vývoji se podílí komunita vývojářů od roku 1986 v čele s Kalifornskou univerzitou v Berkeley [16].

Jednou z předností databáze PostgreSQL je možnost vytváření funkcí (podobné uloženým procedurám). Umožňují spouštět části kódu přímo na serveru a tím zrychlit a zjednodušit práci s daty. Funkce mohou být implementovány v zabudovaném jazyce PL/pgSQL, který se podobá procedurálnímu jazyku PL/SQL od společnosti Oracle. Příkladem dalších jazyků, které se mouhou ve funkcích používat, jsou běžné jazyky jako Python (PL/Python), Lua, PHP (plPHP), C, C++, Java (PL/Java) a další. PostgreSQL umožňuje funkcím vracet celé tabulky nebo jejich části, čímž umožňuje definici vlastních agregačních a okenních (window) funkcí.

V oblasti indexování PostgreSQL podporuje indexy se strukturou B+ stromu, či hašované indexy. Dále umožňuje vytvářet vlastní (uživatelské) indexy. Jednou možností je vytvoření *indexů nad výrazy*, které vytvářejí indexy nad výsledky výrazů a funkcí. Další možností definice vlastního indexu je tzv. *částečný index*. Tento index se vytváří nad částí tabulky jen za předpokladu splnění definované podmínky (v definici indexu se uvede klauzule WHERE).

V určitých situacích je nutné zkontrolovat platnost dat ještě před vložením, či upravením. Běžně se tato činnost provádí na straně aplikace. PostgreSQL nabízí definici tzv. *triggerů*, které kontrolu dat provádí na straně databázového serveru. Trigger umožňuje reagovat na příkazy SQL směřující na námi definovanou tabulkou. Nejčastěji se triggery aplikují pro příkazy typu INSERT a UPDATE [17].

2.1.7 Shrnutí vlastností relačních databází

Relační databáze mají pevně definované schéma, které jednoznačně určuje podobu dat ukládaných v databázi. Této vlastnosti se využívá v aplikacích, u kterých je změna struktur dat vyloučena, nebo k nim dochází velice vyjímečně. Další významnou vlastností tohoto

druhu databází je umožnění vytváření relací, které zachycují vztahy mezi objekty. Relace tak umožňují oddělit data, která mají vzájemný vztah, a rozdělit je do logicky oddělených částí – tabulek. S problémy se setkáme v oblasti škálovatelnosti. Horizontální škálování, neboli přidání více strojů pro rozložení zátěže způsobují problémy při udržování konzistentního stavu databáze. V oblasti počtu zápisů, či čtení za sekundu mají databáze značné omezení. Tato skutečnost je zapříčiněna velkou režií, kterou musí databáze provádět při práci s daty. Jedná se především o kontrolu integritních omezení (kontrola datových typů a kontrola správnosti cizího klíče) [1].

Kapitola 3

NoSQL

NoSQL je databázový koncept, kdy databáze používá k zpracování dat jiné přístupy než tradiční relační databáze založené na tabulkových strukturách. Pro tyto databáze je typické, že porušují některé přístupy relačních databází. Příkladem může být způsob zpracování transakcí, kdy relační databáze používají princip ACID (viz. 2.1.2). Tento přístup zajišťuje spolehlivost operací nad daty v databázi, opravy a detekce chyb. Databázový systém tak zajišťuje jistotu, že se databáze bude nacházet v konzistentním stavu. Provádění těchto operací je poměrně náročné a vytváří limity v oblasti počtu dotazovaných dat. NoSQL databáze tento přístup nedodržují a dovolují tak zvýšit výkon databázového systému na úkor spolehlivosti (viz 3.1). Umožňují distribuovat zátěž napříč *clusterem*, který je tvořen uzly, které mezi sebou navzájem komunikují.

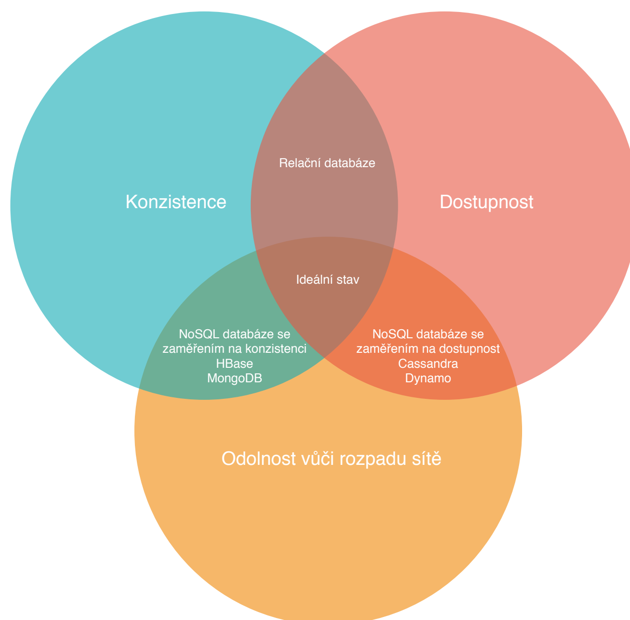
Zdrojem informací pro problematiku NoSQL databází mi byly zdroje z internetu a kniha *Big Data a NoSQL databáze* [11]. Pokud nebude v textu uvedeno jinak zdrojem informací je právě tato kniha.

Vzhledem k rozložení zátěže na uzly v síti se musí řešit problémy, které přináší výpadky v síti.

3.1 CAP teorém

Jak jsem již zmínil v úvodu, NoSQL databáze nepodporují vlastnosti ACID. Hlavní výhodou tohoto přístupu je usnadnění horizontální škálovatelnosti (viz. 2.1.5). Při tvorbě tzv. distribuovaného systému se setkáváme s problémem, kdy replika dat nemusí být z důvodu chyby sítě doručena na uzel, na který se distribuuje. Nicméně i u tohoto druhu databází vyžadujeme určitou garanci správnosti dat v databázi. NoSQL databáze aplikují odlišný přístup, ve kterém se databázový systém snaží dosáhnout ideálních vlastností. Jsou to tyto vlastnosti:

- Konzistence (**c**onsistency), která zaručuje, že v systému se vyskytuje pouze jedna aktuální verze dat.
- Dostupnost (**a**vailability), která garantuje obsloužení všech požadavků ke čtení, či zápisu.
- Odolnost proti rozpadu sítě (**p**artion tolerance), která zajistí funkčnost systému i v případě izolace jednotlivých částí distribuovaného systému z důvodu rozpadu sítě.



Obrázek 3.1: CAP teorém.

Jak lze vidět na obrázku 3.1, tradiční centralizované databáze s podporou ACID transakcí zajišťují pouze konzistenci a dostupnost. NoSQL databáze však pracují na decentralizovaném principu a hlavním řešeným problémem se stává odolnost vůči rozpadu sítě. Z tohoto důvodu je nutné omezit podporu zbývajících dvou vlastností (dostupnost, či konzistenci) [11].

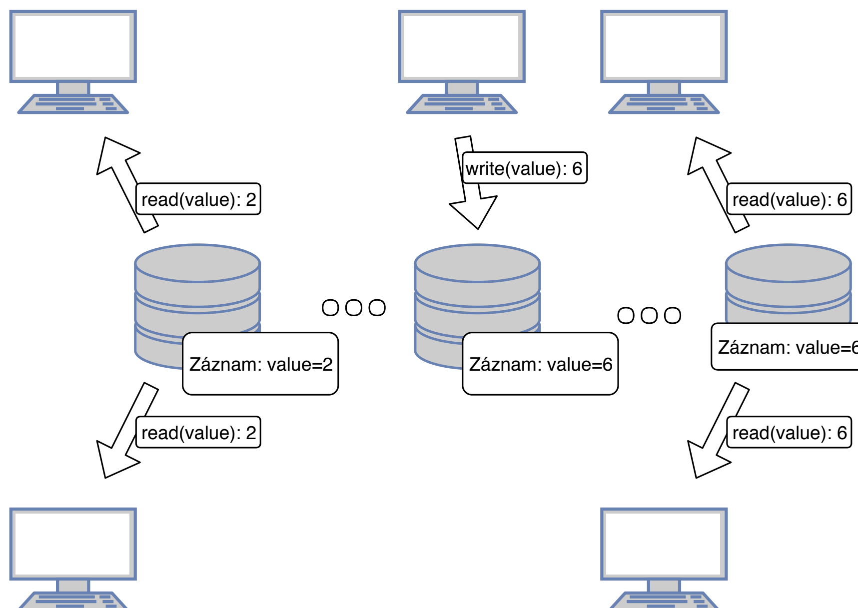
3.2 Občasná konzistence

Decentralizované databázové systémy nahrazují principy ACID modelem BASE, který na úkor konzistence umožňuje horizontální škálovatelnost. Model BASE předpokládá pro databázový systém tyto tři vlastnosti.

- Převážná dostupnost (**b**asically **a**vailable), která počítá s nedostupností některých uzlů v rámci clusteru z důvodů výpadků sítě, ale nepředpokládá nedostupnost clusteru jakožto celku.
- Volný stav (**s**oft state), který určuje nedeterminističnost a dynamičnost systému. Tedy v systému dochází k neustálým změnám.
- Občasná konzistence (**e**ventual consistency) definuje vlastnost systému, kdy se systém snaží dosáhnout konzistence dat, ale tato konzistence není vždy zaručena. Je tak možné, že se systém v určitém okamžiku nevyskytuje v konzistentním stavu. Databázové systémy s občasnou konzistencí obsahují rutiny, které identifikují nekonzistenční data a provedou jejich převedení do konzistentního stavu.

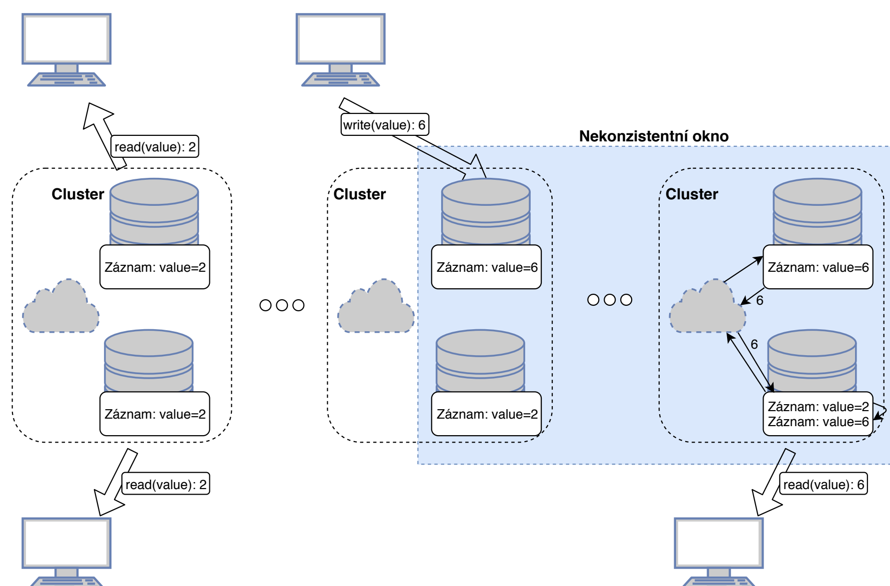
Model BASE přináší alternativu k principu ACID. Silná konzistence, která je zajištěna přístupem na principu ACID, zaručuje konzistenci v každém okamžiku. Tedy k aktualizaci

dat dojde okamžitě po jejich změně (viz Obrázek 3.2). Tento způsob aktualizace dat je z pohledu režie velice náročný a v případech rozsáhlých databázových systémů může znatelně zvyšovat odezvu systému. Důvodem jsou operace aktualizací napříč celým systémem.



Obrázek 3.2: Silná konzistence.

Dočasná konzistence na rozdíl od silné konzistence neprovádí aktualizaci dat okamžitě po provedení změn. Jednotlivé záznamy jsou obvykle zduplikovány a rozmístěny napříč uzly v clusteru v podobě replik (viz. 3.3). Ověření konzistentního stavu a provedení případné aktualizace se oddálí do doby, kdy je konzistence vyžadovaná – do okamžiku čtení těchto dat. Příkladem může být následující situace, která je zachycena na obrázku 3.3. Klient komunikuje s jedním uzlem v síti, kde pracuje s replikou záznamu. Provede aktualizace stavu záznamu a uloží jej. Při silné konzistenci by došlo k okamžitému rozšíření změn do všech uzlů, ve kterých se repliky vyskytují. Při občasné konzistenci se tyto operace neprovedou a je tak umožněn rychlý zápis dat do databáze. Poté se k databázovému systému připojí jiný klient přes odlišný uzel sítě. Chce přečíst data, která byla změněna prvním klientem. Před samotným přečtením dat se v databázovém systému provede kontrola, zda jsou data konzistentní. Při zjištění porušení konzistence se provedou operace, které data převedou do konzistentního stavu a vrátí konzistentní data klientovi. Období mezi provedením změn prvním klientem a provedením aktualizace dat v celém clusteru na základě požadavku čtení dat klientem dvě se nazývá *nekonzistentní okno*.



Obrázek 3.3: Občasná konzistence.

3.3 Distribuce

Z důvodu velkého objemu zpracovaných dat se zátěž distribuuje na uzly v tzv. clusteru, který je tvořen více databázovými servery. Distribuce dat umožňuje s daty provádět dvě operace – *rozdělení a replikaci.*

Rozdělení (*sharding*) rozděluje data na různé uzly a zvyšuje tak kapacitu systému. Pro přístup k různým záznamům je nutné přistupovat do různých uzlů v rámci sítě. Na rychlost práce s daty má zásadní vliv jejich rozmístění v clusteru, kdy se při procházení více uzlů zvyšuje latence systému z důvodu zvýšené režie na komunikaci mezi uzly. Organizaci dat lze provést třemi různými způsoby, přičemž se v praxi volí jejich kombinace.

- V rámci clusteru se data rozmísťují rovnoměrně, aby každý uzel nesl stejný objem dat a jejich zátěž byla přibližně stejná.
- Při práci s daty se snažíme dosáhnout minimálního počtů uzlů, které je nutné procházet při přístupu k datům.
- Data se rozmísťují v souvislosti k jejich geografickému výskytu. Data budou umístěna na uzlu, který se nachází nejblíž oblasti, ve které se s nimi bude pracovat.

Replikace dat rozmísťuje v rámci clusteru kopie jednotlivých dat, čímž se snaží minimalizovat dopady z důvodu výpadku sítě, či selhání uzlu a následné nedostupnosti dat. Replikace dat může probíhat na dvou různých principech – *master-slave* a *peer-to-peer.*

Replikace na principu master-slave vytváří hierarchii uzlů, ve které jsou uzly *primární* a *sekundární.* Při čtení dat ze systému můžeme přistoupit k jakémukoli uzlu, který obsahuje daná data. Při požadavku na zápis dat je požadavek delegován na primární uzel, který zápis provede a o změně informuje všechny sekundární uzly. Výhodou tohoto přístupu je jednotná kontrola zápisů dat, která zabraňuje vzniku write-write konfliktů. Při potřebě zvýšení počtu

replik a přístupových míst pro čtení se jednoduše přidají sekundární uzly. Nevýhodou je výskyt pouze jednoho uzlu pro zápis dat, který disponuje limity pro zápis dat za jednotku času a při výpadku omezuje systém pouze na možnost čtení dat ze sekundárních uzlů. Replikace master-slave nachází uplatnění v systémech, kde čtení výrazně převyšuje zápis a modifikaci dat.

Omezení zápisu při výpadku primárního uzlu je možné odstranit použitím druhého způsobu replikace – replikace *peer to peer*. Replikace peer to peer umožňuje provést zápis kterémukoli uzlu v clusteru. Systém tak není omezen na jeden zapisovací uzel, který je při zvyšování požadavků značně omezen pro škálovatelnost. Při práci s daty se řeší dva konflikty – *read-write* a *write-write*.

Konflikt *write-write* vzniká v případě, kdy se tentýž záznam modifikuje pomocí dvou a více požadavků současně. Hrozí tak trvalé poškození dat v systému, protože změny provedené z jednoho místa v síti budou přepsány daty, které mohou způsobit nekonzistentní stav systému (např. při ztrhávání dvou různých částek z jednoho účtu z dvou různých poboček bank by se bezchybně provedly, ale systém by z účtu odečetl pouze jednu srážku). Řešením tohoto konfliktu je zavedení mechanismu, který umožní provedení pouze jedné změny. Obvykle se v distribuovaných databázových systémech nesetkáme s případem, kdy se repliky dat rozmísťují na všechny uzly v clusteru, ale jen na některé z nich. V tomto případě se musí v rámci clusteru definovat tzv. *replikační faktor*, který určí, kolik replik se má pro jeden záznam celkově vytvořit. Pro eliminaci možnosti vzniku konfliktu write-write je možné zvolit zamykání záznamu v rámci systému, aby nebylo možné záznam upravovat souběžně dvěma požadavky. Tento způsob vynucuje velkou systémovou režii, která je zapotřebí pro komunikaci mezi uzly a zpomaluje tak odezvu systému. Alternativou pro zamykání záznamů je princip, při kterém je nutné provést změnu na většině uzlů v clusteru. V systému se provede pouze ta změna, která se aplikuje na nadpoloviční většině uzlů vzhledem k replikačnímu faktoru – tzv. *kvórum zápisu*. Změna se šíří přes replikační uzly v clusteru. Změna se smí aplikovat pouze na data, která nejsou zabrána operací zápisu na uzlu v rámci jiného požadavku (záznam je uzamčen). Pokud je umožněn zápis, data jsou zabrána požadavkem, který zápis provádí. Operace zápisu dat je označena za úspěšnou, pokud se provede zabrání a zápis na **W** replikách na alespoň $(N/2) + 1$ uzlů, kde N označuje replikační faktor. Nová data se následně rozšíří do zbytku replik v clusteru, na které se zápis doposud neaplikoval. V opačném případě bude operace zápisu označena za neúspěšnou a zápis se neprovede.

Konflikt *read-write* nastává ve situaci, kdy požadavek na čtení dat probíhá současně s požadavkem na zápis dat. V tomto případě hrozí přečtení neaktuálních replik dat, které ještě nebyly pozměněny. Každý záznam musí obsahovat metadata (časovou značku), která označují, kdy byl záznam naposledy upraven. Řešením tohoto konfliktu je nastavení počtu požadovaného počtu replik na čtení (tzv. *kvórum čtení*). Pro počet čtených replik se nastavuje takové číslo R, aby platila nerovnost $R + W > N$, kde W je hodnota kvóra zápisu a N je replikační faktor. Nerovnost garantuje čtení aktuálních dat tím, že alespoň jedna replika dat bude spadat do obou množin uzlů kvóra zápisu i kvóra čtení. Tato replika bude mít nejaktuálnější časovou značku, čímž se zajistí výběr konzistentních dat.

3.4 Členění NoSQL databází

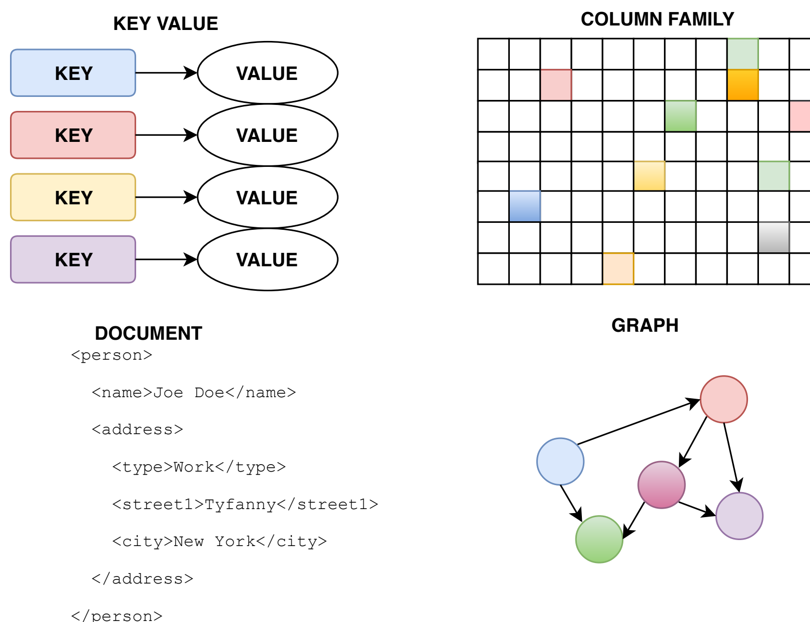
V současné době existuje mnoho NoSQL databázových systémů, které se od sebe odlišují způsobem práce s daty, při čemž se rozlišují čtyři základní proudy. Příkladem může být úložiště typu klíč–hodnota. Další možností realizace databází NoSQL jsou *column family*, *document* a *graf* (obrázek 3.4).

Klíč-hodnota (key-value) databáze pracuje na principu, kdy k jednomu klíči (key) jsou přiřazena konkrétní data (value). Tento způsob přístupu umožňuje rychlé vyhledávání dat, ale neumožňuje mezi nimi charakterizovat jakékoli vztahy a provádět velké množství operací. Jedná se tedy o přímého protějška relačním databázím, v kterých jsou data reprezentována pomocí tabulek se vzájemnými vztahy.

Dokumentové databáze (document-oriented databases) jsou tvořeny prostřednictvím dokumentů. Na rozdíl od relační databáze jsou všechna data patřící jednomu objektu uložena v jednom souboru, přičemž každý objekt může být tvořen libovolnou strukturou. Výhodami tohoto přístupu jsou flexibilita, kterou umožňuje změna struktury jednoho dokumentu (bez nutnosti měnit dokumenty označující stejný objekt), a absence prázdných polí, takže entita relace nemusí obsahovat všechny atributy schématu (tedy i prázdné atributy, tzv. NULL), ale pouze ty, které nesou nějakou hodnotu. Samotné schéma nemusí být definované.

Sloupcové databáze (column family) vytváří struktury podobné tabulkám, přičemž každý záznam (řádek) v tabulce může obsahovat jiný počet sloupců.

Databáze typu graf (graph) vytváří graf pomocí uzlů a hran. Každý uzel označuje jednu entitu (objekt) a je ekvivalentem pro řádek (row) v relační databázi. Uzel disponuje určitými vlastnostmi (properties), které tvoří ekvivalent pro sloupec (column) v relační databázi. Hrany propojují jednotlivé uzly a charakterizují vztahy mezi těmito uzly. Používají se především pro zachycení vztahů mezi daty.



Obrázek 3.4: Realizace NoSQL databází.

Jednotlivé druhy NoSQL databází popíši v následující části, kde se zaměřím hlavním přednostem, datovému modelu a rozmístění záznamů a správě replik v clusteru.

3.4.1 Databáze typu klíč–hodnota

Tento typ úložiště přiřazuje na základě hodnoty klíče odkaz na místo, kde se bude záznam nacházet. Jedná se o velice jednoduchý způsob ukládání, který efektivně vyhledává pomocí klíče, ale nenabízí efektivní prohledávání dat podle jiných atributů. Jednoduchost těchto

úložišť definuje i skromnou nabídku operací, které je s daty možné provádět. Jsou to tři operace. Operace `PUT` vloží data, která jsou identifikována pomocí klíče, `GET` data čte a `DELETE` data maže. Klasickým příkladem využití tohoto typu úložiště je ukládání dat o klientech, kteří přistupují na webové stránky v tzv. *session*.

V případě kdy potřebujeme přistupovat k různým datům, která jsou spjata se stejným klíčem a volba jiného klíče by byla problematická, je možné vytvořit tzv. *jmenný prostor*. Příkladem jmenného prostoru může být třída ve škole, kde název jmenného prostoru nabývá označení třídy. Záznamy žáků a třídních učitelů nepotřebují nést informace o tom, v které třídě působí. V případě výběru studentů z konkrétní třídy je ušetřeno sekvenční vyhledávání mezi všemi záznamy studentů celé školy.

Hodnotu klíčů udává konkrétní hodnota, nebo hodnota hašovací funkce. Při práci v rámci clusteru je zapotřebí vybrat, na který uzel se záznam vloží. Nejjednodušší metodou je využití výsledku operace modulo nad celočíselnou reprezentací hodnoty přístupového klíče. Nevýhodou této metody je vnesení chyby při výpočtu uzlu po rozšíření clusteru o nový uzel. Vložením nového uzlu dojde k zvětšení celkového počtu uzlů a výsledky operace modulo nejsou stejné, takže je zapotřebí provést náročnou operaci přeuspořádání dat v rámci clusteru.

Další možností jak rozmístit záznamy a umožnit jednoduché rozšiřování clusteru je metoda *konstantního hašování*. Hašovací funkce má pevně definovaný obor hodnot. Při vytváření clusteru se obor hodnot rovnoměrně rozdělí mezi uzly v clusteru. Při přidávání nového uzlu se vybere jeden interval, který se rozdělí na dva intervaly a záznamy, které jsou nyní pod správou nového uzlu, se přesunou pod jeho správu. Přeuspořádání záznamů probíhá v rozsahu jednoho intervalu a ostatní uzly jsou o změnách v clusteru informovány pomocí tzv. *gossip* protokolů. Nevýhodou tohoto přístupu je vlastnost hašovací funkce, která nezajišťuje stejnou velikost intervalů mezi jednotlivými uzly po rozšíření clusteru a rovnoměrné rozložení záznamů v celém oboru hodnot, takže dochází k nevyváženému vytížení uzlů.

Nevýhodu konstantního hašování odstraňuje koncept *virtuálních uzlů*. Koncept rozděluje obor hodnot hašovací funkce na předem (při inicializaci) specifikovaný počet virtuálních uzlů. Tyto uzly jsou sekvenčně přidělovány serverům, které se participují v clusteru tak, že jeden server nespravuje dva virtuální uzly jdoucí po sobě. Dochází tak k rovnoměrnému rozdělení záznamů mezi uzly i po změně jejich počtu. Koncept také usnadňuje rozložení replik na uzly v clusteru. Repliky se uloží na daný počet virtuálních uzlů jdoucích za sebou na základě replikačního faktoru.

Databáze typu klíč–hodnota jsou často nasazeny v systémech, v kterých je nutné zajistit vysokou propustnost zápisových operací. Pro replikaci dat se používá peer to peer replikace za využití zápisových a čtecích kvór a pokročilejších technik. Příkladem pokročilé techniky umožňující zajištění konzistence dat je technika *vektorových razítek (vector stamps)*. Každý uzel si pro spravovaný klíč udržuje vektor čítačů. Jednotlivé prvky vektoru odpovídají hodnotám čítačů uzlů, které spravují repliky stejného klíče. Při zápisu na klíč se hodnota čítače zápisového uzlu inkrementuje. Při synchronizaci uzlů se rozšíří hodnota uzlu, který obsahuje nejaktuálnější hodnotu čítače. Případný konflikt se rozpozná při neporovnatelnosti vektorů.

Redis

Příkladem databáze pracující na principu klíč–hodnota je databáze Redis (Remote Dictionary Server).

Hlavním předností je průběžné ukládání dat do operační paměti. Jedná se o tak zvanou *in–memory* databázi, která umožňuje vysokou propustnost zápisových operací. Data se ukládají do operační paměti a po určité době nebo po dosažení definované objemu jsou případně přesunuta na disk v podobě zálohy. Zálohování dat může být provedeno dvěma různými způsoby. Prvním způsobem je pravidelné vytváření souborů RDB *(Redis Database Backup)*. Tento soubor obsahuje kompletní obraz databáze ve stavu, v kterém se při vytváření nachází. Jsou zde uložena uživatelská data a serializované záznamy včetně časových razítek pro kompletní obnovení po restartu databázového serveru. Druhým způsobem je vytváření souborů APF *(Append Only File)*, které obsahují záznamy příkazů, které byly v databázi vykonány.

Databáze Redis nabízí 5 datových typů. Základním typem je `řetězec` (`string`), který zajišťuje binární konzistenci zapsaných dat, takže může být použit k zápisu jakýchkoli dat do maximální velikosti 512 MB. Příkladem použití je čítač, pro které nabízí atomické operace `INCR`, `DECR` a `INCRBY`. Prvky datového typu řetězec lze dále vložit do datových struktur typu `seznam (list)`, `množina (set)`, `haš (hash)`, `uspořádané množiny (sorted set)` a do specifického datového typu `bitmaps`, či `hyperLogLogs`. Ke každé datové struktuře je poskytnuta sada operací pro snadnou obsluhu (`LPUSH`, `SADD` a další).

Pro svoji rychlost a jednoduchost se databáze Redis nejčastěji používá pro cache sezení (session cache), frontu zpráv a počítadlo přístupů [18].

3.4.2 Dokumentové databáze

Tento typ úložiště člení ukládaná data do souborů v datových strukturách, které popisují charakter dat (metadata). Nejčastějším formátem pro ukládání dat je JSON, BSON (JSON v binární podobě) a XML.

Hlavní jednotkou je dokument, který může mít libovolný formát. V praxi se upřednostňuje přístup, kdy se společně ukládají data pouze stejného druhu (např. data o zákaznících, zboží, ...). Tyto databáze umožňují provést ukládání stavů objektů (tzv. *objektově dokumentové mapování – ODM*) dvěma způsoby.

Prvním způsobem je využití *vnořených objektů* (*embedded documents*), jehož zápis ve formátu JSON je uveden níže. Výhodou tohoto přístupu je zpřístupnění dat pomocí jednoho dotazu, protože data jsou uložena na jednom místě. Tento přístup umožňuje zachytit vztah 1:1 nebo 1:N. Nevýhodou je výrazné zpomalení přístupu při velkém nárůstu dat napříč více dokumenty.

```
{
    "sensor": "temperature and humidity"
    "measurement": {
        "temperature": "5",
        "humidity": "45"
    }
}
```

Příklad vnořených dokumentů (objektů).

Druhým způsobem je využití odkazů. Tento přístup umožňuje rozdělit data do více dokumentů. Jak je uvedeno na příkladu níže. Jednotlivé dokumenty obsahují jednoznačný identifikátor s označením *_id*, které současně slouží jako jejich primární klíče. Vzájemně lze dokumenty provázat pomocí odkazů, které jsou identifikovány pomocí podřetězce *_id*.

Tento přístup umožňuje vyjádřit vztah N:M. Nevýhodou je rozmístění spolu souvisejících dokumentů dál od sebe, což při získávání odpovědi na dotaz může vést k zvýšení odezvy.

```
{
    "_id": <ID1>,
    "sensor": "temperature and humidity"
}

{
    "_id": <ID2>,
    "sensor_id": <ID1>,
    "temperature": "5",
    "humidity": "45"
}
```

Příklad odkazování v dokumentech.

MongoDB

Typickým zástupcem dokumentových databází je databáze MongoDB. Tato databáze pro ukládání dat využívá formát JSON. Základní jednotkou je dokument, s který umožňuje pracovat oběma způsoby zmíněnými výše.

Pro replikaci využívá principu master–slave, aby umožnila vysokou propustnost čtecích operací nad *množinou replik* (*replica set*). Ovladače sloužící k připojení klientských aplikací k databázi a poskytují mechanizmy k obsloužení jak čtecích, tak i zápisových požadavků bez nutnosti specifikace primárního (master) uzlu. Klientská aplikace uvede adresu známému uzlu a ovladač samostatně provede nalezení primárního uzlu, na který následně deleguje veškeré zápisové operace.

Jak je uvedeno na obrázku 3.1, databáze MongoDB upřednostňuje konzistenci dat a odolnost proti rozpadu sítě. V případě rozpadu sítě a znemožnění komunikace mezi uzly je zajištěna konzistence dat na úkor zápisových operací tak, že v jedné izolované části clusteru není možné zapisovat. Část, které bude umožněno zapisování, je určena na základě nadpoloviční většiny. Druhá část bude podporovat pouze čtení. Pokud se v clusteru vyskytuje sudý počet uzlů, je možné přidat tzv. *arbitra*. Tento uzel nenese žádná data, ale poskytuje hlas při rozhodování o nadpoloviční většině při rozpadu clusteru. Vytvoření repliky je provedeno pomocí *operačního logu* (*oplog*), který může být aplikován okamžitě po doručení na podřazený uzel nebo po vypršení časového limitu ve formě *zpožděné kopie*.

Při velkém množství položek v kolekci se data dělí do menších celků (tzv. sharding). Data jsou rozdělena pomocí primárního nebo jiného klíče (tzv. sharding key), nad kterým je definovaný index. O správu rozdělení záznamů v kolekci se na základě informací z konfiguračního serveru stará proces `mongos`.

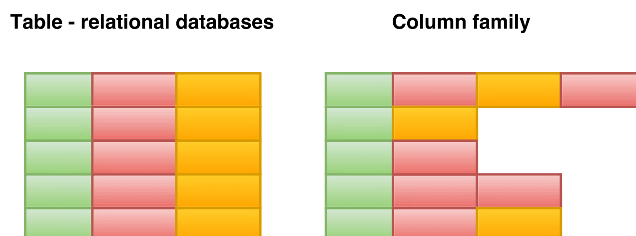
MongoDB umožňuje definici kvór (viz. 3.3), které v kombinaci s master–slave replikací a nastavením šíření replik okamžitě po vytvoření záznamu podporují ACID vlastnosti systému [12].

V současnosti nachází uplatnění v oblasti IoT (Internet of things) a webových aplikací, u kterých je zapotřebí velká propustnost čtecích operací a bezproblémové editace velkého množství dat.

3.4.3 Sloupcové databáze

Počátek vývoje sloupcových databází je v roce 2008, kdy společnost Google prezentovala článek Bigtable [6], ve kterém definuje jejich datový model. Sloupcové databáze se z NoSQL databází nejvíce podobají tradičním relačním databázím. Na rozdíl od relačních databází umožňují velkou volnost schématu, ale oproti databázím typu klíč–hodnota a typu dokument definují přesnější schéma dat.

Rodina sloupců (column family) se v podstatě skládá ze stejných složek jako tabulky relačních databází, obsahuje řádky (rows) a sloupce (columns), ale disponuje vlastnostmi, které jsou pro NoSQL databáze typické. Nad column family není možné provádět operace typu JOIN a poddotazy (subqueries), jako tomu je u relačních databází. Dalším a zásadním rozdílem je, že rozměry jednotlivých řádků (rows) nemusí být vždy shodné, jako je běžné u relačních databází. Jak lze vidět na obrázku 3.5 u vytvoření záznamu (řádku) v sloupcových databázích není nutné vložit data do předem definovaných sloupců, ale je možné přidávat libovolné sloupce. Při definici rodiny sloupců pouze určujeme, které sloupce spolu logicky souvisí a budou dotazovány společně. Databázový systém pak udržuje záznamy rodiny sloupců se stejným identifikátorem řádku blízko sebe (na jednom uzlu), aby zvýšil efektivitu přístupu.



Obrázek 3.5: Rozdíl mezi tabulkou v relační databázi a rodinou sloupců

Každý řádek obsahuje libovolný počet sloupců, které nesou informace o svém jméně, hodnotě a času vytvoření (timestamp). Řádek je identifikovaný pomocí svého unikátního klíče (row key).

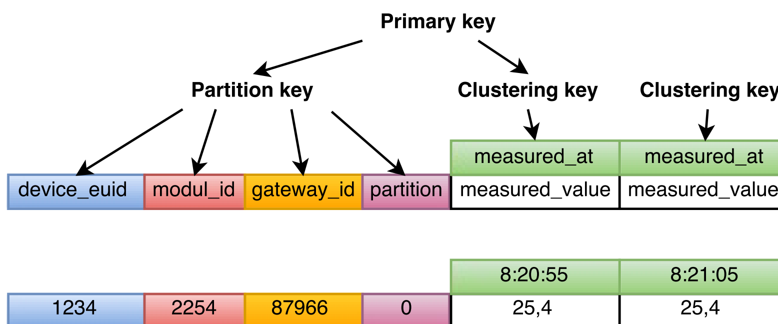
Cassandra

Apache Cassandra je open source systém pro správu databází NoSQL. Cassandra je navržena pro práci s velkým objemem dat, přičemž zajišťuje jejich vysokou dostupnost. Cassandra využívá svůj vlastní dotazovací jazyk CQL (Cassandra query language), který od verze 3 přistupuje k datovému modelu jako k řídké tabulce (na rozdíl od předchozích verzí, ve kterých byla struktura column family interpretována jako multidimenzionální pole). Jazyk CQL vychází ze syntaxe dotazovacího jazyka SQL, ale nepodporuje vnořené dotazy. Dále zavádí pojmy tabulka (pro column family) a buňka (pro column). Jazyk modifikuje datový model sloupcových databází a neumožňuje kdykoli přidávat nové sloupce. Pro umožnění přidávání nových sloupců databáze nabízí datové typy `set` a `map`.

Každý řádek je určen unikátním klíčem (primary key), který tvoří hash hodnot jeho sloupce nebo sloupců. Hlavní složkou primary key je tzv. partition key, který je tvořen prvním sloupcem zapsaným ve výčtu sloupců primary key. Partition key určuje, které záznamy

budou uloženy blízko u sebe, aby k nim bylo možné efektivně přistupovat. Další nepovinnou částí primary key je clustering key, který definuje uspořádání v rámci oddílu (řádku). Partition key i clustering key mohou být tvořeny prostřednictvím composite key, který je vytvořen hodnotami dvou a více sloupců [4].

Nyní uvedu příklad tvorby primary key při vytváření tabulky z návrhu pro IoTCLoud. Při vytváření column family (table od CQL 3) je zapotřebí určit jméno tabulky, která se bude vytvářet (zde *log*), a jméno KEYSPACE, v kterém se bude nacházet (mezi různými KEYSPACE lze přepínat příkazem USE). Dále je nutné definovat alespoň jeden sloupec (column). Takto nadefinujeme 6 sloupců a definujeme PRIMARY KEY. V příkladě 3.4.3 a na obrázku 3.6 je PRIMARY KEY tvořen s využitím partition key a clustering key. Partition key obsahuje composite key (`device_euid`, `module_id`, `gateway_id` a `partition`). Tyto čtyři hodnoty jednoznačně určují řádek, na který se bude pomocí tohoto partition key zapisovat. Clustering key tvoří pouze jedna položka – `measured_at`. Hodnota položky definuje, která část řádku se bude adresovat. V tomto případě `measured_at` tvoří něco jako indexy pole hodnot typu float (`measured_value`).



Obrázek 3.6: Grafické znázornění funkcí partition key a clustering key.

```
CREATE TABLE IF NOT EXISTS beeeon.log (
    device_euid decimal,
    module_id smallint,
    gateway_id decimal,
    partition tinyint,
    measured_at timestamp,
    measured_value float,
    PRIMARY KEY (
        ( device_euid, module_id, gateway_id, partition ),
        measured_at
    )
)
```

Příklad vytvoření tabulky v systému Cassandra.

Vlastnosti databáze Cassandra jsou navrženy za účelem spolehlivě uložit velké množství dat a zajistit jejich vysokou dostupnost. Databáze Cassandra při obsluze požadavků využívá

oba typy paměti – RAM a ROM. Při operaci zápisu se provádí zápis do dvou datových struktur. První z nich je *zápisový log*, který je uložen na disku a je odolný vůči ztrátě dat při výpadku napájení. Je to textový soubor typu *append only*, ve kterém se rychle ukládají příkazy typu `INSERT`. Druhou strukturou je tzv. *memtable*. Memtable je držena v operační paměti, aby zápis proběhl v nejkratším možném čase a byla umožněna vysoká propustnost požadavků. Po dosažení definované velikosti je provedena operace přesunutí dat z operační paměti na disk (operace `flush`). Na disku jsou data uložena v strukturách zvaných *SSTable*. Tyto struktury jsou v čase neměnné (*immutable*). Proces modifikace je možný pouze přes operace, které jsou nejdříve zaznamenány v zápisovém logu a v memtable.

Záznamy jsou v systémem vyhodnocený čas (nízká zátěž atd.) nahrávány do operační paměti a přeorganizovány pro zlepšení čtení. Současně jsou porovnána se zápisem v operačním logu a s obsahem memtable. Změny nad daty se provedou (úprava, mazání) a data jsou opět zapsána ve formě pozměněných SSTable na disk. Tento proces se označuje jako *konsolidace* (*consolidation*).

Jednotlivé uzly v clusteru disponují stejnou rolí a nejsou vůči sobě hierarchicky nadřazené, či podřazené (peer to peer replikace). Data jsou distribuována v rámci clusteru na základě jeho nastavení, takže zodpovědnost za uchování informace se rozděluje mezi více uzlů a celý cluster je odolnější vůči ztrátě dat zapříčiněné selháním jednoho a více uzlů. Kontrola konzistence dat je zajištěna pomocí zápisových a čtecích kvór a data jsou distribuována na základě replikační strategie.

Replikační strategie určuje, na které uzly budou kopie uloženy. Existují dvě replikační strategie – **SimpleStrategy** a **NetworkTopologyStrategy**

SimpleStrategy používá pouze jedno datové centrum a jeden rack. První kopie je uložena na základě rozhodnutí řadiče a všechny další kopie jsou umístěny na uzly po směru hodinových ručiček bez ohledu na topologii.

NetworkTopologyStrategy se aplikuje při užití dvou a více datových center. Tato strategie umožňuje vybrat počet kopií pro každé datacentrum zvlášť. Při rozhodování o počtu kopií v datovém centru se snaží najít nejlepší kompromis mezi redundancí s časem přístupu (možnost číst z lokálního datového centra a zamezit tak latenci způsobené přístupem do centra jiného) a možností selhání uzlu.

V nižším příkladě je uveden proces vytvoření keyspace Test s užitím NetworkTopologyStrategy se třemi datacentry (center1, center2, center3). U každého datacentra se udává nenulové přirozené číslo, které přiděluje danému datovému centru replikační faktor (viz. 3.3).

```
CREATE KEYSPACE "Test"
  WITH REPLICATION = {'class' : 'NetworkTopologyStrategy',
 'center1' : 3, 'center2' : 2, 'center3' : 3};
```

Databáze Cassandra nachází uplatnění v aplikacích, které požadují vysokou propustnost zápisových operací. Nejčastějším druhem ukládaných dat jsou serializovaná data a data z IoT (Internet of Things). Příkladem použití je společnost Netflix, která v databázi Cassandra ukládá metadata (historie, záložky atd.). V současnosti je cluster schopný za vteřinu zpracovat 10 milionů operací [8].

3.4.4 Grafové databáze

Grafové databáze ukládají data jiným principem. Tento přístup nevyužívá klíče, ale vstupní uzly do grafu. Hlavní výhody, které přináší oproti relačním databázím, jsou rychlost operací a možnost modifikace modelu dat. Grafové databáze jsou specializované pro operace průchodu grafem, který oproti klasickým relačním databázím nevyžaduje spojení většího množství tabulek. Uzel je reprezentován jako ukazatel na paměťový prostor, takže může obsahovat libovolný formát dat, který je v daný okamžik potřebný.

Graf je složen z uzlů a hran. Uzel slouží k zachycení nějaké entity z reálného světa. Může obsahovat atributy, které zachycují jeho vlastnosti. Hrany slouží k propojení uzlů. Jsou ve většině případů orientované a mohou obsahovat atributy, které určují vlastnosti vztahu jako typ vztahu, podmínky existence a čas platnosti.

Grafy mohou být *orientované* nebo *neorientované*. Dále rozlišujeme grafy podle počtu druhů hran, které v nich existují. Grafy *jednovztahové* obsahují hrany pouze jednoho typu, jejichž typ v databázi nemusí být specifikovaný. Oproti tomu v grafech *vícevztahových* (*ohodnocených*) se hrany rozlišují a musí obsahovat atributy definující jejich typ. V případě kdy vytvoříme graf, ve kterém dva uzly propojujeme různými hranami, hovoříme o tzv. *multigrafu*. Pokud jedna hrana propojuje více než dva uzly grafu, jedná se o *hypergraf*.

Neo4j

Neo4j je v současné době jedním z nejpopulárnějších zástupců grafových databází. Je vyvíjena společností Neo Technology od roku 2007. Systém je implementován v jazyce Java.

Distribuce dat v clusteru není možná, protože pro správnou funkci systému je vždy nutné udržovat celý graf na jednom uzlu. Tato skutečnost přináší omezení počtu uzlů, které může graf obsahovat. Řádově se jedná o 2^{35} uzlů a 2^{35} hran. Definice clusteru je však umožněna, aby bylo možné rozložit zátěž při čtení dat. Replikace je založena na principu master–slave, kdy se na každém uzlu ukládá celý graf a následné změny jsou synchronizovány dvěma způsoby. Prvním způsobem je občasná distribuce změn, která probíhá při zápisu dat na uzel typu master. Druhým způsobem je zápis na uzel typu slave. V tomto případě jsou změny rozšířeny okamžitě, aby se zabránilo nekonzistenci dat.

Datový model je definovaný pomocí uzlů a hran. Neo4j podporuje vytváření orientovaných multigrafů, přičemž v případě potřeby lze orientaci hran zanedbat. Uzly i hrany mohou obsahovat atributy, které jsou identifikovány řetězcem. Atribut nabývá jednoduchého datového typu.

K přístupu do databáze je možné využít více způsobů. Prvním z nich je přístup z aplikace pomocí aplikačního API (např. v jazyce JAVA, PHP, Python atd.). Dalším způsobem je využití jazyků, které umožňují pracovat s grafy. Příkladem pro databází Neo4j jsou jazyky Gremlin a Cypher. Jazyk Gremlin je určen pro iterativní průchod grafem pomocí jednoduchých a efektivních příkazů. Naopak jazyk Cypher je určen pro deklarativní práci s databází, kdy je specifikováno, jaká data chceme z databáze získat bez specifikace kroků, které mají být provedeny. Syntaxe jazyka je podobná SQL.

Databáze Neo4j se stejně jako další grafové databáze využívá k ukládání dat s velkým počtem vztahů, které v případě použití tradičních relačních databází znamenají nutnost zavedení velkého počtu tabulek zachycujících vztahy a při dotazování vyžadují spojení těchto tabulek. Využití nachází v sociálních sítích, vyhledávacích nástrojích a v řízení a mapování přístupu [13].

3.5 Shrnutí vlastností NoSQL databází

NoSQL databáze nevynucují pevnou strukturu schématu. Tedy jejich záznamy mohou být různé velikosti a jednotlivé domény nemusí obsahovat pouze jednoduché datové typy, ale také typy složené (pole, viz. 3.4.3).

NoSQL databáze nepodporují tvorbu cizích klíčů, takže neumožňují zachytávat vztahy mezi záznamy v databázi. Tato skutečnost snižuje potřebnou režii při přistupování k záznamům (není dovoleno používat příkazy JOIN). Absence relací je výhodná při přístupu k velkému objemu dat, která se nevztahují k jiným záznamům v databázi. Příkladem mohou být periodicky ukládaná data ze senzorů. Pokud bychom v databázi chtěli zachytit vztahy, musíme tuto režii provést na aplikační úrovni. Tato režie je velice náročná a při zpracování velkého množství dat, které je při použití NoSQL databází typické, je velice náročné na zdroje i výpočetní čas, a proto se v těchto případech NoSQL databáze nepoužívají.

Data jsou soustředěna v několika kolekcích, při čemž každá kolekce se obvykle vztahuje k jedné části aplikace. Kolekce jsou uspořádány tak, aby často dotazovaná skupina záznamů byla sdružena co nejblíže u sebe.

NoSQL díky ustoupení od přístupu ACID umožňuje provádět horizontální škálování. Zátěž je rovnoměrně distribuovaná na jednotlivé uzly v distribuované architektuře. Tyto databáze jsou navrženy pro vysoký výkon v oblasti čtení a zápisu dat [1].

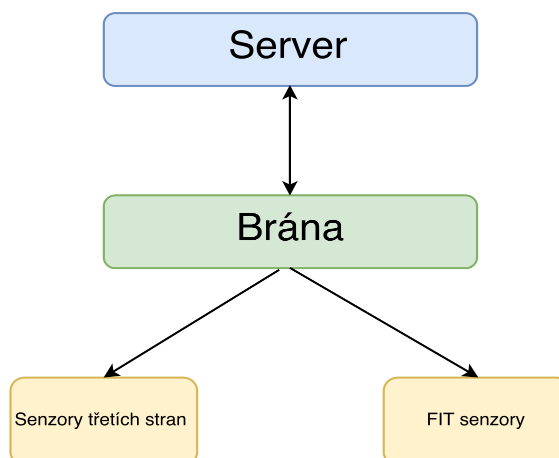
Kapitola 4

Projekt IoTCloud

IoTCloud je projekt, který se zaměřuje na zpracování dat v oblasti IoT (Internet of Things). Cílem projektu je vytvoření systému, který zpracovává data v centrálním místě na internetu (v rámci projektu označeného jako cloud). Data jsou podrobena analýze umělou inteligencí, která na základě jejich výsledků může provést případné akce. Data jsou sbírána z bezdrátových senzorů, která jsou podporované a pochází od výrobců třetích stran.

4.1 Architektura projektu

Systém se skládá ze tří částí (viz obrázek 4.1). Nejnižší vrstvu tvoří zařízení třetích stran. Tato zařízení jsou spravována pomocí serveru (cloudu). Mezivrstvu mezi serverem a zařízeními třetích stran tvoří Brána, která mezi nimi zajišťuje komunikaci.



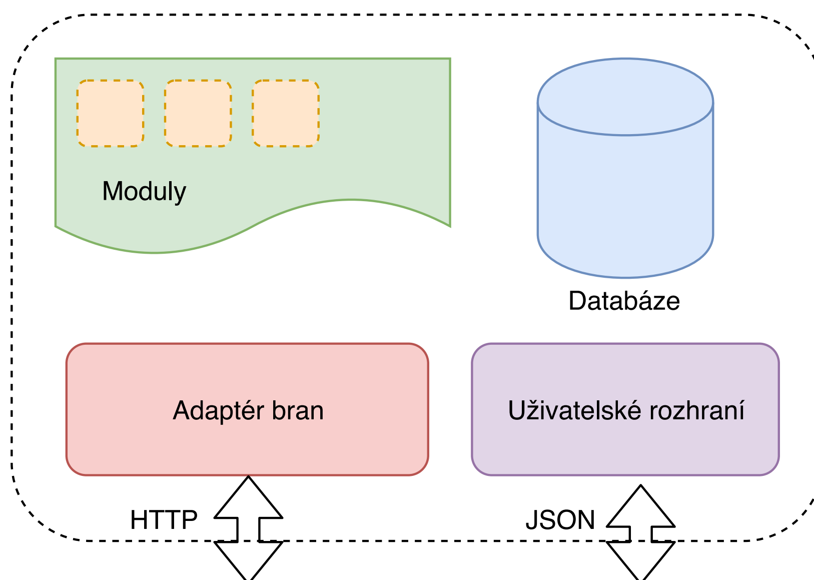
Obrázek 4.1: Architektura systému.

Na nejnižší vrstvě se nacházejí senzory a aktory. Příkladem mohou být senzory vyvinuté při FIT, které komunikují s bránou pomocí FIT protokolu [3]. Senzory jsou jednoduchá zařízení, která slouží k získávání dat z okolního prostředí. Může se jednat například o senzory

tlaku, vlhkosti, teploty nebo hlásiče CO (oxidu uhelnatého). Aktory jsou zařízení, která svojí činností ovlivňují podmínky v okolním prostředí. Například můžou spustit vytápění, větrání nebo osvětlení místnosti.

Brána obsahuje software z projektu BeeeOn (část BeeeOn gateway). Projekt BeeeOn je vydán pod volně šiřitelnou licencí a je vyvíjený na Fakultě informačních technologií Vysokého učení technického v Brně [2]. Brána je zařízení zodpovědné za zabezpečení komunikace s cloudem a za detekci anomálií v bezdrátových sítích. Brána vždy navazuje spojení s cloudem pomocí technologie WebSocket, která zasílá HTTP požadavky. Tyto požadavky jsou zabezpečené technologií TLS. Data jsou posílána ve formátu JSON. Dále je zodpovědná za sběr dat a ovládání aktorů.

Server registruje senzory, brány a uživatele. Dále sbírá a vyhodnocuje data přijatá z těchto senzorů a je zodpovědný za zasílání příkazů do koncových stanic, které jsou závislé na činnosti uživatele, nebo na automatických úlohách nastavených uživatelem. Architektura serveru je zachycena na obrázku 4.2. Důležitou část tvoří databáze, která slouží pro ukládání dat a v současné době prochází řadou změn. Databázová část, která doposud obsahovala pouze databázi PostgreSQL, by se měla postupně rozšířit o databázi Cassandra. Dále obsahuje moduly, které provádí inteligentní analýzu dat a rozhodují o provedení akcí.



Obrázek 4.2: Architektura serveru.

Uživatelské rozhraní je zprostředkováno pomocí webové aplikace. Frontend aplikace je implementován pomocí javascriptové knihovny React. Frontend komunikuje s backendem pomocí technologie AJAX. Frontend server je vytvořen na frameworku Django a spravuje vlastní databázi pro uchování informací o nastavení uživatele. Frontend zajišťuje komunikaci se serverem přes rozhraní REST-UI a umožňuje tak konfigurovat zařízení z uživatelského rozhraní.

Kapitola 5

Analýza možností urychlení práce s daty

Současný systém IotCloud ukládá data v databázi PostgreSQL. Senzorická data jsou ukládána v tabulce následujícím způsobem. Nasnímané hodnoty ze senzorů jsou uloženy v tabulce `sensor_history_recent`. Primární klíč je složen ze čtyř atributů – `gateway_id`, `device_id`, `module_id` a `at`. První dvojice atributů tvoří cizí klíče. Prvním atributem z nich je `gateway_id`, která odkazuje do tabulky `gateways` a vyjadřuje, na které bráně je senzor připojen. Druhý atribut odkazuje na záznam, který popisuje zařízení. Dále následuje identifikátor senzoru v rámci zařízení. Poslední složkou je časové razítko získaných dat. Jedinou novou informaci, kterou s každým záznamem ukládáme je hodnota atributu `value`. Všechny ostatní informace jsou redundantní a slouží jako metadata pro naměřenou hodnotu. Pokud uvážíme příchod 1 000 hodnot za 5 minut, znamená to, že na disku tato implementace zabere 500 MB. To je do budoucna neudržitelné. Optimalizace provedené v době psaní tohoto textu umožnili uspořit cca 20 % prostoru.

Nad senzorickými daty jsou nejčastěji prováděny dotazy zaměřené na výběr dat. Doba jejich vykonání silně závisí na délce časového období, přičemž nejkomplikovanější dotazy trvají až 0,5 s.

Pomocí databáze Cassandra je možné stejný objem dat ukládat s následujícími parametry. Provedl jsem vložení 1 000 senzorických dat do databáze pro testovací senzor v celkovém časovém rozmezí 5 minut (po 300 milisekundách). Tento počet záznamů spotřeboval zhruba 71 KB diskového místa.

Dotazy pro databázi Cassandra lze optimalizovat dobře navrženou strukturou tabulky, která zakládá na dvou pravidlech. První pravidlo požaduje rovnoměrné rozložení dat po clusteru. Rozložení dat v úložišti ovlivňuje první prvek primary key – partition key. Partition key jednoznačně identifikuje oddíl (partition), ke které se bude přistupovat. Tento oddíl tvoří řádky, přičemž každý určuje hash jeho primary key. Pro optimalizaci je tedy nutné správně zvolit správný primary key tak, aby data, která se nejčastěji dotazují, byla sdružena v jednom oddíle, ale současně byla rovnoměrně rozložena po clusteru.

Druhé pravidlo požaduje minimalizaci počtu čtecích oddílů. Dodržování tohoto pravidla snižuje latenci čtecích dotazů, protože každý oddíl může být umístěn na jiném uzlu. V tomto případě koordinátor dotazů vydává samostatné dotazy každému uzlu, který obsahuje oddíl s odpovídajícím partition key. Vzrůstá tak potřebná režie a zvyšuje se latence prováděného dotazu. Tyto dvě pravidla jsou navzájem v konfliktu, takže se vždy musí zvolit kompromis odpovídající požadavkům ze strany aplikace [10].

Kapitola 6

Porovnání databází

V této části se zaměřím na porovnání jednotlivých databází. Pro porovnání uvedu test SQL databáze MySQL. Dále budu pracovat se zástupci skupin NoSQL databází. Byli vybráni tito zástupci: Redis, MongoDB, Cassandra a Neo4j.

Pro testování výkonnosti byl vytvořen program, který do databází vloží testovací data a provede nad nimi množinu definovaných operací. V průběhu testování budou sledovány tyto klíčové parametry:

- Množství spotřebované operační paměti před a po vložení dat.
- Paměťové nároky pro uložení na disku.
- Propustnost operací zápisu za sekundu.
- Rychlost editace dat.
- Rychlost výběru dat daného senzoru v časovém intervalu (poslední den, týden a měsíc).

Testovací data tvoří množina hodnot, které představují hodnoty získané při měření senzory v periodickém intervalu. Interval měření je stanoven na 30 sekund. Do databáze budou vložena data, které odpovídají 2 rokům měření. Na jeden senzor připadá přibližně 2 102 400 naměřených hodnot. Budou zpracována data z dvou trojic senzorů, přičemž se každá trojice nachází na jiné bráně. Každý senzor bude data nahrávat paralelně, aby vyprodukovaný provoz simuloval skutečnou situaci. Data jsou identifikovaná pomocí identifikátoru brány, `euid` zařízení, `id` modulu a časem, kdy byla naměřena. Hodnotu dat tvoří náhodné desetinné číslo (`float`).

Testovací stroj

- AMD FX-8320E Eight-Core Processor 3.20 GHz
- 8 GB RAM
- Souborový systém NTFS
- ST2000DX 001-1CM164 SATA Disk Device, SATA, 158 MB/s, 1 TB

Redis

Redis je databáze, která pracuje v operační paměti (tzv. in–memory). Předpokladem tedy je velice rychlý zápis dat. Jako klíč byl použit řetězec utvořený z hodnot identifikující měření kromě časové známky. Pod tímto klíčem je vytvořena hašovací mapa. Jedná se tedy opět o správu dat typu klíč–hodnota. Klíčem je časové razítko naměřených dat a hodnotou výsledek měření. Obě hodnoty jsou převedeny do textové podoby.

Při nahrávání dat byla sledována zátěž operační paměti a procesoru. Maximální hodnota RAM během testu dosáhla 950 MB. Poté následovala rutina, která uloží data z operační paměti na disk. Tato rutina vytížila procesor na 15 %. Databáze Redis umožňuje konfigurovat chování při práci s daty. Při prvním testu bylo použito základní nastavení. Data se na disk ukládaly po 10 000 záznamech. Při zápisu byla zapnuta komprese dat. Výsledný soubor měl velikost přibližně 418 MB (33,2 B na záznam).

Při nahrávání dat byla sledována propustnost zápisu, přičemž výstup měřícího programu vykázal následující výkonnost operací. Je vybráno minimum a maximum. Průměr trvání zápisové operace je 229 us při průměrné propustnosti 4 366 zápisů za sekundu pro senzor. Celkem bylo zpracováno 12 614 400 požadavků za 515 sekund (24 493 zápisů za sekundu).

Dotazy na data z databáze byly zdlouhavé. Důvodem je nemožnost specifikovat interval, ve kterém chceme data vybírat. Tento výběr bylo nutné provést skrze aplikaci, která musela pracovat s časovým intervalem záznamů a sama počítat časová razítka. Dalším důvodem je nahrávání dat z disku. Dotazování trvalo kolem 8 sekund.

Po této sadě testování byla databáze restartována a uvedena do původního stavu. Server byl spuštěn s nastavením, které neukládalo data na disk. Nároky na paměť se zvýšily na 1.6 GB. V oblasti zápisu došlo k zlepšení na hodnoty 210 us, tedy 4 761 zápisů za sekundu pro senzor a celkově 28 566 zápisů za sekundu. V oblasti dotazování nad daty nebylo zjištěno zlepšení. Z pozorování vyplývá, že průběžné ukládání na disk nemá na výkonnost vkládání dat příliš velký význam vzhledem k rostoucím nárokům na operační paměť.

Cassandra

Databáze Cassandra je sloupcová databáze, která se oproti ostatním NoSQL databázím vyznačuje nutností definice schématu. Primární klíč byl vytvořen z dvojice partition key a clustering key. Partition key je tvořen pomocí composite key, který obsahuje hodnoty identifikátoru brány, `euid` zařízení, `id` modulu a hodnoty `partition`, která umožňuje aplikačně rozlišovat bloky dat jednotlivých měření. Clustering key je tvořen časovým razítkem.

Data, která odpovídají sběru hodnot ze šesti senzorů za dva roky v intervalu 30 sekund, byla vložena a proběhlo sledování nároků na systémové prostředky. Během vkládání dat dosáhlo maximální vytížení RAM 1 585 MB a procesoru 20 %. Zápis dat proběhl se základní konfigurací. Ukládání dat na disk probíhá v periodě 10 sekund nebo po zaplnění datové struktury v operační paměti (memtable) 128 MB dat. Data byla ukládaná do formátu SSTable tak, aby bylo umožněno efektivnější čtení dat. Naměřená propustnost systému byla přibližně 3 313 zápisů za sekundu. Velikost databáze byla 187,05 MB (zjištěno pomocí příkazu `nodetool stats`). Na záznam je zapotřebí 14,82 B.

Pro testování výběru dat bylo provedeno více měření za různých podmínek. První měření bylo provedeno nad daty, která nebyla nijak indexována. Poté byl zaveden index na čas měření, aby se urychlilo prohledávání nad těmito daty. Velikost tabulky narostla na 200 MB (15,86 B na záznam). Bylo opětovně provedeno měření, které potvrdilo efektivnější práci s daty. Dotazování s indexem dosahuje rychlosti 12,5 ms. Při dotazování nad neindexovanými daty se délka dotazu pohybuje kolem 16,5 ms. Při vykonávání dotazu se

musí v obou případech přistoupit do stejného `partition`, ve kterém jsou data seřazena pomocí `clustering key`, neboli času měření. Odlišnost při vykonávání dotazu je tedy ve vyhledání hraničního intervalu, kdy se při indexování přistupuje přímo a při neindexovaném přístup spouští iterativní vyhledávání.

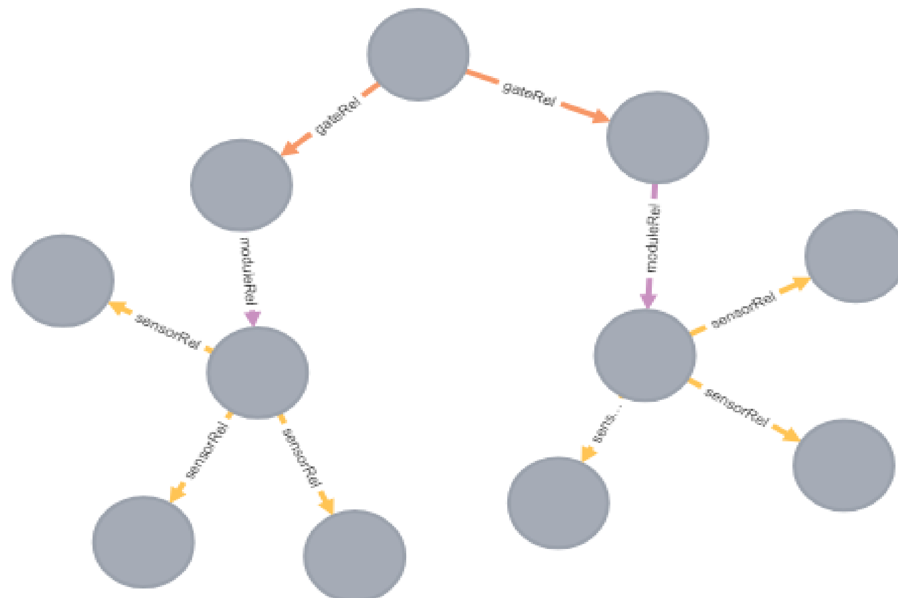
MongoDB

Databáze MongoDB vytváří dokumenty a data v nich ukládá ve formátu JSON. Jeden dokument reprezentuje jeden senzor. Dokument obsahuje atributy, které identifikují senzor, a pole `values`, které obsahuje jednotlivá měření. Přidávání prvků probíhá pomocí operace update a příkazu `$push`, který vloží data na konec pole. Zápis dat tímto způsobem probíhal velice pomalu, proto bylo vykonávání přerušeno a vložen menší počet záznamů. Proces vkládání dat zabral v maximu až 2 156 MB a vytížil procesor na 68 %. Propustnost byla celkově naměřena pouhých 966 zápisů za sekundu, takže by vkládání zabralo kolem 217,5 minut. Na jeden záznam připadá 47 B.

V druhém testu byla data vložena ve formě dokumentů, kdy jednomu dokumentu odpovídá jedno měření. Naměřená propustnost byla 16 446 zápisů za sekundu. Průměrná velikost záznamu je 133 B. Tento přístup je podle měření velice efektivní. Databáze byla před měřením restartována.

Neo4j

Databáze Neo4j ukládá data do grafů, které jsou reprezentované pomocí uzlů a hran. Struktura ukládání dat je navržena tak, aby umožnila rychlé procházení napříč grafem. Pro testování byl vytvořen graf. Struktura grafu obsahuje kořenový uzel, ke kterému jsou připojeny brány. Každá brána obsahuje modul, který je propojen se senzorem. Senzor je propojen pomocí ohodnocených hran s uzly, které vyjadřují naměřené hodnoty. Hrany jsou ohodnocené pomocí časového razítka. Graf má strukturu, kterou můžeme vidět na obrázku 6.1.



Obrázek 6.1: Grafové znázornění v Neo4j browseru.

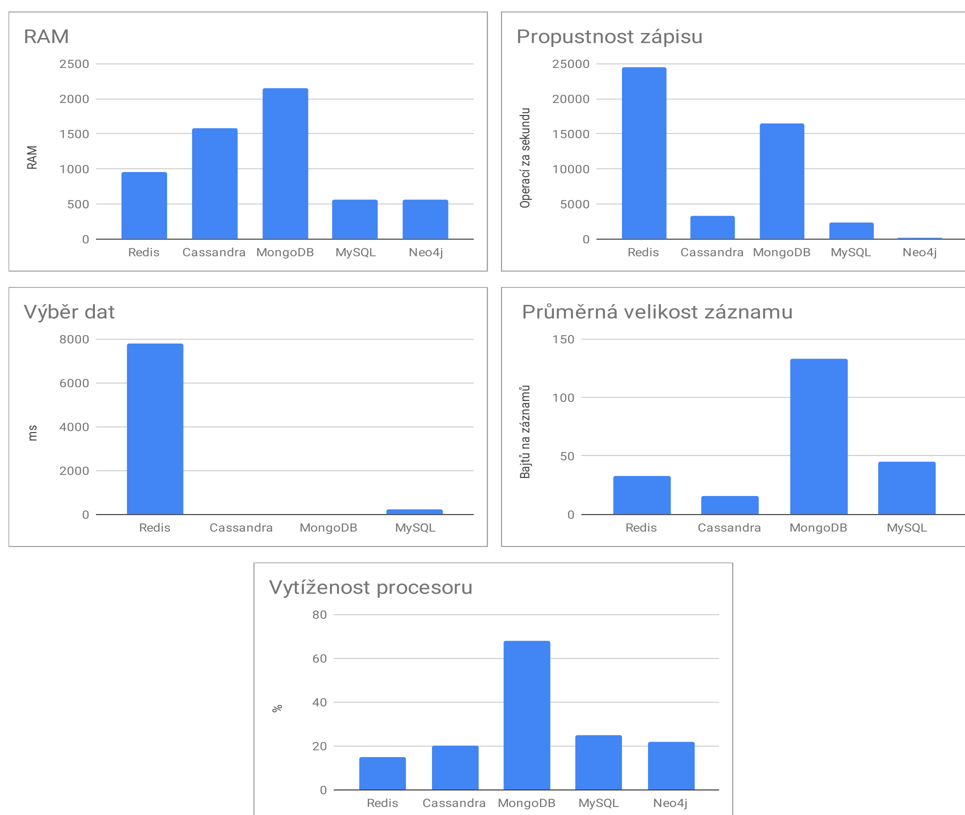
Při vkládání dat bylo nutné omezit časový interval, ve kterém se data nahrávala, z důvodu velmi nízké propustnosti. Při testování zápis 60 000 vzorků trvalo 293 sekund. Naměřená propustnost byla 204 zápisů za sekundu. RAM byla vytížena na 556 MB a procesor z 22 %.

MySQL

Pro srovnání bylo provedeno měření výkonnosti databáze MySQL. Jako primární klíč byla zvolena čtveřice – identifikátor brány, `euid` zařízení, `id` modulu a čas naměření dat. Naměřená propustnost byla 2 334 zápisů za sekundu. Záznamy na disku spotřebovaly 571,395 MB, tedy 45,3 B na záznam. Vytížení RAM bylo maximálně 556 MB a procesu kolem 22 %. Při výběru dat bylo provedeno pozorování, z kterého vyplývá, že odezva dotazování dat v určitém intervalu je přímo úměrná k délce dotazovaného intervalu. Naměřené hodnoty byly v rozsahu od 21,66 ms pro dotazování z jednoho dne až 428,85 ms pro data z posledního měsíce.

Shrnutí výsledků

Naměřená data korespondují s vlastnostmi jednotlivý databází. Výsledky testů jsou zachyceny na obrázku 6.2.



Obrázek 6.2: Výsledky testů.

Databáze Redis, která poskytuje minimální množství operací nad daty se vyznačuje vysokou propustností zápisu. Nevýhodou tohoto přístupu je nutnost selekci dat provádět

z aplikačního prostředí. Veškeré výpočty přístupových parametrů musí být spočítány na straně aplikace. Dalším negativním jevem je značné prodloužení odezvy na výběr dat, které je způsobeno velkým množstvím dotazů na menší objem dat, takže velké množství času je spotřebováno na režii přenosu dat. Z tohoto důvodu je využití databáze Redis pro uchovávání senzorických měření nevhodné. Možné využití se naskýtá v použití na místech, u kterých lze očekávat velké množství příchozích dat s možností průběžného přenášení záznamů do jiného úložiště.

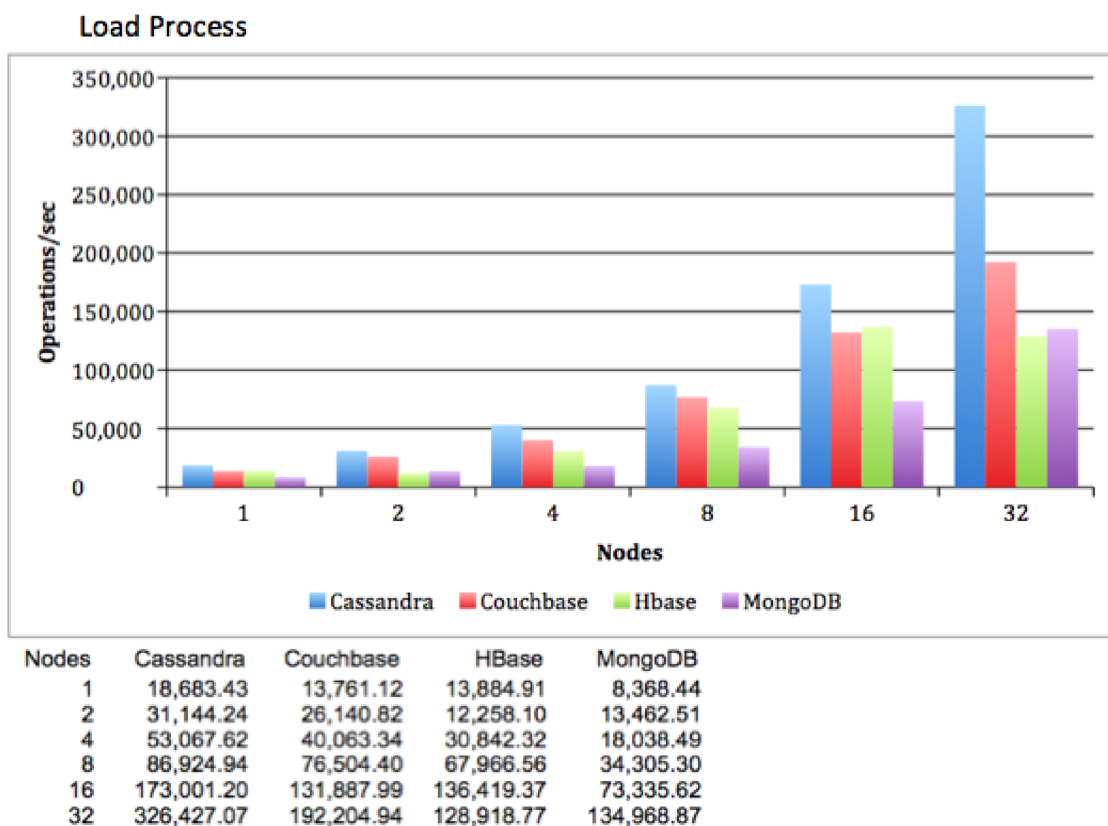
Databáze MySQL nabízí klasický přístup k ukládání dat. Při vkládání dat nebylo nutné provádět velké množství kontrol integritních omezení, které by zpomalovaly zápis. Nevýhoda spočívá ve vysoké redundanci a neukládání dat do menší lépe přístupných sekcí (tzv. partitions), což má za následek prodlužování doby potřebné pro průchod většího množství dat.

Během průběhu měření pro databázi MongoDB bylo při vkládání hodnot do pole vloženo pouze zlomek hodnot. Propustnost zápisu při provádění operace `$push` je natolik nízká, že pro vkládání dat není vhodná. O mnoho efektivnější bylo ukládání jednotlivých měření jako samostatných dokumentů. Výsledky vkládání i vybírání dokumentů vykázalo nejlepší poměr v celém testování. Nevyhovujícím parametrem testu byly vysoké paměťové nároky na ukládání záznamů.

Databáze Neo4j vykázala nejhorší výsledky a pro správu dat získaných sběrem dat ze senzorů je nevhodná. Propustnost zápisu byla velice nízká. Další velkou nevýhodou je nemožnost distribuce grafu napříč clusterem, protože na každém uzlu musí být přítomný celý graf, který není možné rozdělit. Databáze Neo4j je vhodná pro data, která mezi sebou mají velké množství vztahů a jsou častěji čtena než zapisována.

Poslední porovnávanou databází byla databáze Cassandra. Na rozdíl od ostatních zástupců NoSQL databází vyžaduje definice schématu. Tato vlastnost však v případě ukládání senzorických dat nevadí a poskytuje možnost efektivního přístupu podle sledovaných parametrů. Z měření vyplývá, že Cassandra dosahuje stejné časové odezvy při výběru dat z různě dlouhých časových intervalů. Tato vlastnost je umožněna díky přihrádkám (`partition`), do kterých jsou data členěna podle složky `partition key`. Shlukování v těchto blocích s aplikací kompresních algoritmů vytváří prostor pro pokles paměťových nároků. Propustnost zápisu dat je dostačující a díky možnosti horizontálního škálování je možné dosáhnout velice vysoké propustnosti. Příkladem využití potenciálu databáze Cassandra je její nasazení ve společnosti Netflix pro provoz dosahující až 10 milionů požadavků za sekundu (viz. [8]).

Testování nasazení v clusteru provedla univerzita v Torontu, společnost Netflix (viz[7]) a společnost END POINT v publikaci *Benchmarking Top NoSQL Databases* [5], kde databáze Cassandra dosáhla nejlepších výsledků v porovnání s konkurenčními databázemi Couchbase, HBase a MongoDB. Příklad porovnání propustnosti zápisu v závislosti na počtu uzlů v síti je vidět na obrázku 6.3. Proto byla pro ukládání senzorických dat vybrána databáze Cassandra.



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	18,683.43	13,761.12	13,884.91	8,368.44
2	31,144.24	26,140.82	12,258.10	13,462.51
4	53,067.62	40,063.34	30,842.32	18,038.49
8	86,924.94	76,504.40	67,966.56	34,305.30
16	173,001.20	131,887.99	136,419.37	73,335.62
32	326,427.07	192,204.94	128,918.77	134,968.87

Obrázek 6.3: Propustnosti zápisu v závislosti na počtu uzlů [7].

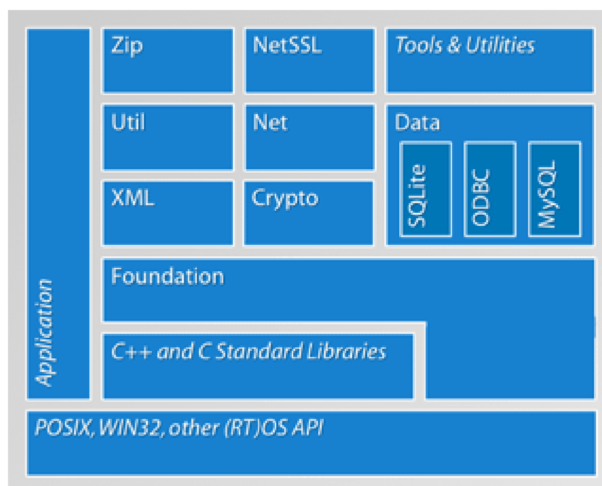
Kapitola 7

Ovladač pro databázi Cassandra

V rámci bakalářské práce jsem se primárně zaměřil na časově náročnou implementaci ovladače pro databázi Cassandra v knihovně Poco. Využil jsem ovladač od společnosti Datastax, který je implementován v jazyce C.

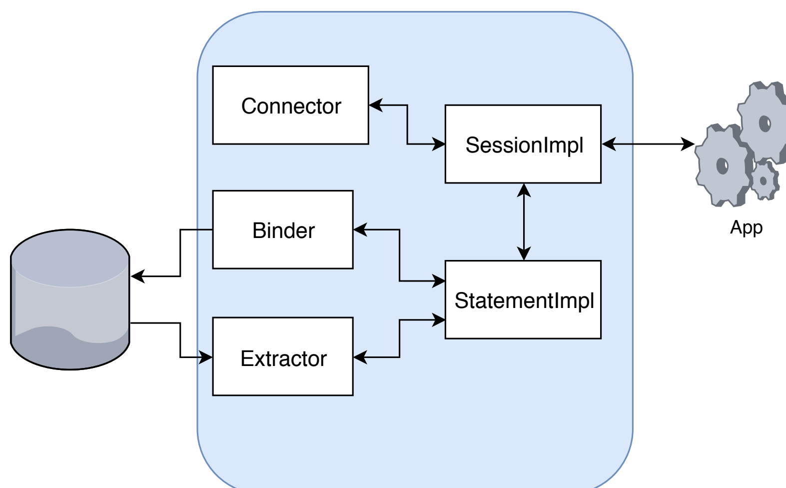
Poco je open source knihovna, která integruje standardní knihovny jazyka C++. Umožňuje urychlit vývoj aplikacích zaměřených na komunikaci po síti díky poskytnutí vysokoúrovňových předností objektového programování a efektivní nízkoúrovňové obsluhy vstupních a výstupních zařízení a přerušení. Struktura knihovny je zobrazena na obrázku 7.1. Jádro tvoří čtyři knihovny. Jsou to knihovny Foundation, XML, Util a Net. Dále knihovna Poco poskytuje další moduly jako jsou NetSSL, Crypto a Data [14].

Během práce jsem se podrobně zaměřil na modul Data, který zapouzdřuje obsluhu připojení na jednotlivé databáze. Jedná se o ovladače k relačním databázím. Pro účely projektu bylo zapotřebí vytvořit ovladač pro databázi Cassandra, která byla vybrána jako vhodná NoSQL databáze pro ukládání senzorických dat.



Obrázek 7.1: Struktura knihovny Poco [14].

Tvorba ovladače spočívala v implementaci rozhraní, které knihovna Poco obsahuje. Celkem bylo zapotřebí implementovat pět tříd, které stručně popíši v následujících bodech. Byly to třídy Binder, Extractor, StatementImpl, SessionImpl a Connector [15].



Obrázek 7.2: Architektura ovladače.

Třída **Binder** slouží k namapování datových typů C++ na datové typy databáze Cassandra. Implementace této třídy spočívá v definici přetěžované funkce *bind*, která přiřadí předanou hodnotu na zadanou pozici v dotazu. Princip definování této funkce spočíval v dosazení funkce z ovladače od společnosti Datastax, která odpovídala danému datovému typu. Případnou komplikaci může způsobit konverze datových typů.

Extractor zprostředkovává selekci hodnot z databáze. Zpracovává řádek, který se získal dotazem na databázi, na zadané pozici. Definice této třídy je obdobná jako definice třídy Binder.

Třída **StatementImpl** obsahuje instance tříd Binder a Extractor a je zodpovědná za veškeré procedury spojené s vykonáváním dotazu na databázi (přípravu dotazu, vložení hodnot do dotazu tzv. binding, zpracování a extrakce výsledků).

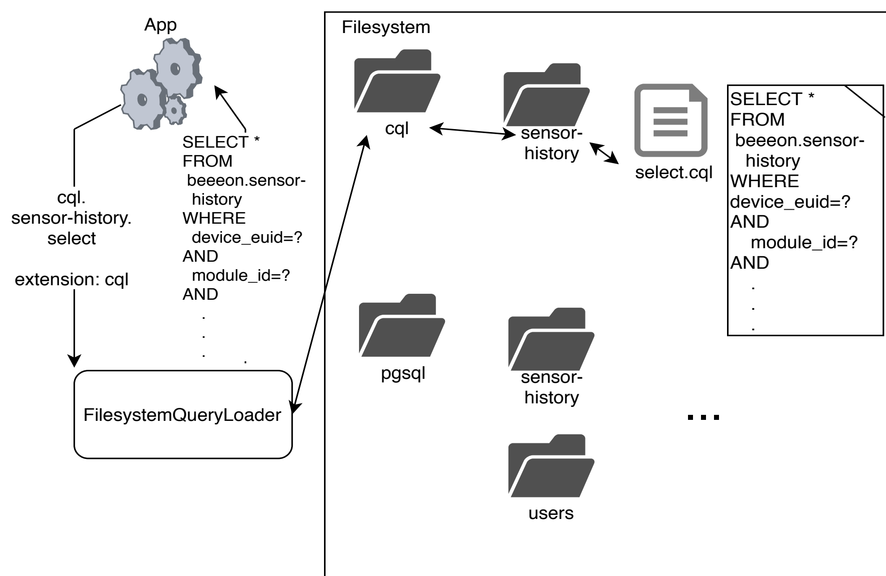
SessionImpl je třída, která v konstruktoru otevírá a v destruktoru uzavírá spojení s databází. Obsahuje i metody k obsloužení transakcí. SessionImpl určuje implementaci třídy Session, která je důležitou součástí při komunikaci s databází. Je přes ní zprostředkováno vykonávání veškerých dotazů. Jelikož je navázání spojení s databází časově náročná operace, používá se k uchování instancí této třídy tzv. SessionPool.

Třída **Connector** rozhoduje o tom, ke které databázi se bude připojovat, podle zvoleného klíče. Pravidlem pro tvorbu klíče pro třídu Connector je řetězec odpovídající jménu databáze, ke které se Connector připojuje.

Kapitola 8

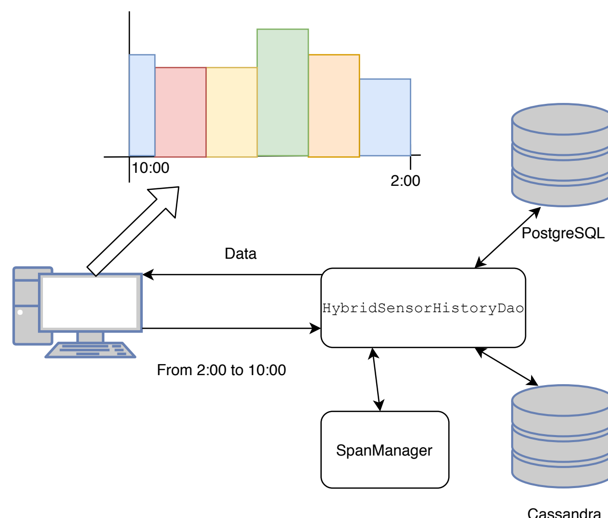
Integrace databáze Cassandra

Pro integraci databáze Cassandra do systému IoTCloud bylo zapotřebí implementovat ovladač pro knihovnu Poco, která se v projektu využívá (viz 7). Rozšíření serveru o databázi Cassandra vedlo k přepracování tvorby dotazů na databázi. V původním provedení se dotazy tvořily z *ini* souboru a byly složeny z volání uložených procedur, které jsou součástí schématu a zkracují tak zápis tohoto dotazu i jeho round-trip, který zpomaluje systém. Schéma je spravováno systémem Sqitch, který umožňuje správu schématu při vývoji i nasazení. Databáze Cassandra nemůže být tímto systémem spravována a nepodporuje uložené procedury, jaké se používají pro dotazování PostgreSQL. Struktura ukládání dotazů se transformovala z podoby *ini* souborů do souborového systému, který umožňuje sjednocení pravidel pro tvorbu a vytváření dotazů na obě databáze. Došlo k odstranění uložených procedur a tedy i k zkrácení sqitch plánu. Načítání dotazů, které je znázorněna na obrázku 8.1, probíhá pomocí FilesystemQueryLoaderu. FilesystemQueryLoaderu přistupuje k jednotlivým dotazům přes jejich klíč. Klíč je určen relativní cestou k souboru s dotazem oddělenou tečkami (adresář.adresář.soubor tedy např. cql.sensor-history.select). Loader umožňuje nastavit příponu souboru (file extension), který se bude hledat.



Obrázek 8.1: Načítání dotazu ze souborového systému.

Databáze Cassandra spravuje agregovaná data za určitý časový úsek a používají se mimo jiné i k vykreslování grafů v uživatelském rozhraní. Byl implementován selektor dat (HybridSensorHistoryDao), který vybírá data dle potřeby z databáze PostgreSQL, nebo Cassandra. Zpracování dotazu na data je znázorněno na obrázku 8.2. Selektor dat při výběru agregovaných dat (v databázi Cassandra) vybírá vhodný agregační interval pro výběr dat, která nejpřesněji vystihují hodnoty v daném časovém rozmezí. Všechny agregační intervaly jsou spravovány správcem agregačních intervalů (SpanManager).



Obrázek 8.2: Výběr dat pro požadovaný rozsah.

Při tvorbě schématu databáze se struktura jednotlivých tabulek podřídila dotazům, které nad nimi budou prováděny. V systému IotCloud tvoří nejčastější případ dotazování dotazy nad daty z jednoho senzoru za určité časové období s určitou úrovní agregace. Byla vytvořena tabulka, která bude obsahovat všechna agregovaná data různých agregačních úrovní.

```
TABLE beeeon.sensor-history (
    device_euid decimal,
    module_id smallint,
    gateway_id decimal,
    partition tinyint,
    span int,
    measured_at timestamp,
    measured_value float,
    PRIMARY KEY (
        ( device_euid, module_id, gateway_id, partition, span ),
        measured_at
    )
)
```

Jako partition key (viz 3.4.3) byl určen složený atribut `( device_euid, module_id, gateway_id, partition, span )`. Tento partition key distribuuje data jednoho senzoru s konkrétní agregační úrovní na jedno místo v clusteru. Struktura tabulky upřednostňuje

rychlost čtení oproti rovnoměrnému rozložení záznamů po clusteru. Důvodem volby tohoto řešení je fakt, že současný cluster je tvořen pouze jedním uzlem, takže všechna data se budou vždy vyskytovat právě v tomto uzlu.

Pro testování dotazů v rámci projektu vznikl testovací skript. Skript generuje testovací data simulující provoz a ukládání agregovaných dat (s různými úrovněmi agregace) za posledních 14 dní. Dále jsou postupně testovány zadané dotazy s různým časovým rozsahem.

V současné době byla integrace testována lokálně na serveru bez napojení na reálný provoz. Jednalo se o krátkodobé běhy nad množinou generovaných dat. Snížení výkonu na straně aplikace nebylo pozorováno. K zpomalení běhu dochází při spuštění v testovacím režimu, ve kterém se startuje databázový server. Při startu a připojení k již běžícímu databázovému serveru nebylo zjištěno zpomalení běhu aplikace. Dotazy směřující na databázi PostgreSQL byly při vývoji značně optimalizovány, takže dosahují lepších výsledků než dotazy směřující na databázi Cassandra, která tento nedostatek kompenzuje úsporou paměťové prostoru.

Kapitola 9

Závěr

Cílem práce bylo seznámit se s NoSQL databázemi a navrhnout vhodné databáze pro ukládání senzorických dat pro projekt IoTCloud, implementace ovladače databáze a integrace do systému IoTCLoud.

Pro výběr vhodné databáze do prostředí IoT bylo nutné se seznámit s rozdíly mezi SQL a NoSQL databázemi. Dále bylo zapotřebí porozumět odlišnostem mezi jednotlivými druhy těchto databází, aby je bylo možné porovnávat. Byly charakterizovány problémy dosavadního úložiště a proběhla specifikace požadavků pro nové úložiště. Bylo provedeno testování a porovnání výkonností vybraných zástupců NoSQL databází, ze kterého byla pro integraci do projektu vybrána databáze Cassandra.

Při vypracování práce se podařilo zprovoznit databázi Cassandra na serveru a vytvořit schéma s potřebnou strukturou. Knihovna Poco byla rozšířena o ovladač pro databázi Cassandra, přičemž byl otestován v rámci jednotkových testů dle konvencí knihovny Poco. Dotazy směřující na PostgreSQL nyní nepoužívají uložené procedury a byly přemístěny z *ini* souboru do souborového systému, kde jeden soubor odpovídá jednomu dotazu. Všechny dotazy (včetně dotazů směřujících na databázi Cassandra) se ukládají do filesystemu a jsou načítány pomocí nově implementované funkcionality. Server nyní dokáže spravovat intervaly časových agregací, přičemž tato nová vlastnost je využita v nové komponentě serveru pro selekci dat. Úlohou komponenty je výběr vhodného úložiště pro získání požadovaných dat (viz 8.2).

Proběhla definice prvních (testovacích) dotazů na databázi Cassandra. Provedl jsem i první měření rychlosti dotazování a paměťových nároků. Z měření paměťových nároků vyplynula značná úspora paměťového prostoru. Rychlost dotazování naměřená při provádění měření nad simulovaným provozem přináší drobné zlepšení, kdy data získáváme do maximálně 100 milisekund. V porovnání se současnými výsledky v databázi PostgreSQL tato rychlost nepředstavuje významné zlepšení, ale při uvážení paměťových nároků představuje zajímavou alternativu vůči databázi PostgreSQL. Hlavní zlepšení výkonnosti serveru umožňuje zavedení škálovatelnosti, kdy se zátěž dá rozprostřít mezi více strojů. Vlastnost se projeví při nutnosti zvýšení propustnosti zápisových operací, které podle očekávání budou časem značně narůstat.

V rámci IoTCloud se Cassandra využije jako úložiště velkého množství senzorických dat (big data). Budou se implementovat a nasazovat agregační skripty, které budou průběžně vytvářet agregace vyšších úrovní. Poté bude testováno nasazení v rámci clusteru s více uzly v síti.

Literatura

[1] Academind: *SQL vs NoSQL or MySQL vs MongoDB*. [Online; navštíveno 23.01.2019].
URL https://www.youtube.com/watch?v=ZS_kXvOeQ5Y

[2] BeeeOn: *beeeon.org*. [Online; navštíveno 15.01.2018].
URL https://beeeon.org/wiki/Main_Page

[3] BeeeOn: *FIT protocol*. [Online; navštíveno 15.01.2018].
URL https://beeeon.org/index.php?title=FIT_protocol

[4] Bertuccini, C.: *Primary key in Cassandra*. [Online; navštíveno 15.01.2018].
URL https://stackoverflow.com/questions/24949676/difference-between-partition-key-composite-key-and-clustering-key-in-cassandra

[5] END POINT *Benchmarking Top NoSQL Databases*. [Online; navštíveno 06.05.2018].
URL http://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf

[6] Chang, F.; Dean, J.; Ghemawat, S.; aj.: Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, ročník 26, č. 2, 2008: s. 1–26, ISSN 1557-7333.

[7] Datastax: *Apache Cassandra NoSQL Performance Benchmarks*. [Online; navštíveno 06.05.2018].
URL https://academy.datastax.com/planet-cassandra/nosql-performance-benchmarks

[8] DataStax: *Netflix Personalizes Viewing for Over 50 Million Customers with DataStax*. [Online; navštíveno 21.04.2019].
URL https://www.datastax.com/wp-content/uploads/2011/09/CS-Netflix.pdf

[9] *Základy relačních databází, jejich využití v programování webu*. [Online; navštíveno 30.05.2018].
URL https://gml.vse.cz/data/oppa-webdesign/zaklady-db.html

[10] Hobbs, T.: *Basic Rules of Cassandra Data Modeling*. [Online; navštíveno 02.06.2018].
URL https://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling

[11] Holubová, I.: *Big Data a NoSQL databáze*. Profesionál, Praha: Grada, první vydání vydání, 2015, ISBN 978-80-247-5466-6.

[12] *MongoDB*. [Online; navštíveno 16.4.2019].
URL https://docs.mongodb.com/manual/

[13] Neo4j: *When Connected Data Matters Most*. [Online; navštíveno 23.04.2019].
URL https://neo4j.com/use-cases/

[14] *A Guided Tour Of The POCO C++ Libraries*. [Online; navštíveno 23.04.2019].
URL https://pocoproject.org/docs/00100-GuidedTour.html

[15] POCO: *POCO Data Connectors Developer Guide*. [Online; navštíveno 20.01.2018].
URL https://pocoproject.org/docs/00300-DataDeveloperManual.html

[16] *PostgreSQL*. [Online; navštíveno 06.05.2019].
URL https://www.postgresql.org/about/

[17] *PostgreSQL*. [Online; navštíveno 06.05.2019].
URL https://www.postgresql.org/about/featurematrix/

[18] *Redis*. [Online; navštíveno 9.4.2019].
URL https://redis.io/

[19] Wikipedia: *Databáze*. [Online; navštíveno 15.01.2019].
URL https://cs.wikipedia.org/wiki/Datab%C3%A1ze

[20] Zendulka, J.: *Databázové systémy : IDS*. Brno: Fakulta informačních technologií, 2008, [Online; navštíveno 30.05.2018].
URL http://docplayer.cz/2879793-Databazove-systemy-ids.html