VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

NOSQL DATABÁZE PRO DATA SENZORŮ S PODPOROU ČASOVÝCH ŘAD

NOSQL TIME SERIES DATABASE FOR SENSOR DATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR VIZINA
AUTHOR

VEDOUCÍ PRÁCE — RNDr. MAREK RYCHLÝ, Ph.D.
SUPERVISOR

BRNO 2017

Zadání diplomové práce/19637/2016/xvizin00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Vizina Petr, Bc.**
Obor: Informační systémy
Téma: **NoSQL databáze pro data senzorů s podporou časových řad**
NoSQL Time Series Database for Sensor Data
Kategorie: Databáze

Pokyny:

1. Seznamte se principy NoSQL databází a s charakteristikami senzorových dat. Prozkoumejte možnosti optimálního uložení a dotazování časových řad dat v databázích (tzv. "time series databases").
2. Navrhněte způsob uložení, indexování a dotazování senzorových dat s časovými řady v NoSQL databázi. Navrhněte vlastní NoSQL databázový server či upravte/vylepšete již existující produkt pro uložení senzorových dat.
3. Po konzultaci s vedoucím databází implementujte. Soustřeďte se na rychlost ukládání dat, jejich minimální velikost na disku, ale také na rychlost dotazů (běžných i nad časovými řadami).
4. Řešení důkladně otestujte nad generovanými či reálnými daty, vyhodnoťte rychlost operací s databází a velikost úložiště.
5. Výsledky zdokumentujte a zveřejněte jako open-source pod svobodnou licencí.

Literatura:

- A. MacDonald (2016) *PhilDB: the time series database with built-in change logging*. PeerJ Computer Science 2:e52 [https://doi.org/10.7717/peerj-cs.52]
- D. Ramesh, A. Sinha and S. Singh (2016) *Data modelling for discrete time series data using Cassandra and MongoDB*. In 3rd International Conference on Recent Advances in Information Technology (RAIT), Dhanbad, 2016, pp. 598-601. [https://doi.org/10.1109/RAIT.2016.7507966]
- T. Goldschmidt, A. Jansen, H. Koziolek, J. Doppelhamer and H. P. Breivold (2014) *Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes*. In IEEE 7th International Conference on Cloud Computing, Anchorage, AK, 2014, pp. 602-609. [https://doi.org/10.1109/CLOUD.2014.86]
- T. Dunning and E. Friedman. (2014) *Time Series Databases: New Ways to Store and Access Data*. O'Reilly Media. ISBN 1-4919-1472-6. [http://shop.oreilly.com/product/0636920035435.do]

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a započatá práce v bodě 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá tématem NoSQL databází, jež lze využít pro efektivní ukládání senzorových dat s charakterem časových řad. Cílem je navrhnout a implementovat vlastní řešení databáze určené pro ukládání dat časových řad s využitím právě principu NoSQL.

Abstract

This thesis deals with NoSQL databases, which can be used for effective storage of sensors data with character of time series. The aim is to design and implement own solution for database designed to store time series data, with usage of NoSQL.

Klíčová slova

NoSQL, časové řady, databáze, senzory, IoT, MongoDb, Java

Keywords

NoSQL, time series data, databases, sensors, IoT, MongoDb, Java

Citace

VIZINA, Petr. *NoSQL databáze pro data senzorů s podporou časových řad*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

NoSQL databáze pro data senzorů s podporou časových řad

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Petr Vizina

24. 5. 2017

Poděkování

Chtěl bych poděkovat vedoucímu práce RNDr. Marku Rychlému, Ph.D. za ochotnou odbornou pomoc a užitečné rady k řešení projektu, dále bych rád poděkoval Ing. Janu Kořenkovi, Ph.D., Ing. Tomáši Novotnému a Ing. Janu Viktorinovi za poskytnutí informací k existujícímu systému databázového serveru.

Obsah

Obsah **1**
1. Úvod **2**
2. Internet věcí **4**
2.1 Inteligentní domácnosti 5
3. NoSQL databáze **7**
3.1 Charakteristiky NoSQL databází 8
3.2 Klasifikace NoSQL databází 9
3.3 Nevýhody NoSQL 11
4. Časové řady **12**
4.1 Smysl uložení časových řad 13
4.2 Požadavky na databázi 13
4.3 Databáze časových řad 14
5. Současná databáze **19**
6. Návrh databáze **21**
6.1 Požadavky na databázi 21
6.2 Návrh aplikace 21
6.3 Návrh uložení 23
7. Implementace **28**
7.1 Uložení dat v databázi 28
7.2 Uložení dat v operační paměti 31
7.3 Dotazy nad daty 34
8. Testování **38**
8.1 Rychlost vkládání dat 38
8.2 Rychlost výběru dat 39
8.3 Velikost dat na disku 41
9. Možné rozšíření **42**
9.1 Agregace naměřených hodnot 42
9.2 Snížení granularity dat 43
9.3 Cache dotazů 43
9.4 Rozhraní pro C++ 43
10. Závěr **44**
Literatura **45**
Seznam příloh **47**
A. Manuál k aplikaci **48**
B. Obsah DVD **49**

Kapitola 1

Úvod

Současná doba spěje počítačovému trhu, který se neustále rozvíjí a hledá nové možnosti jeho využití. Pořizovací cena hardwaru jako jsou procesory, nebo celé integrované zařízení na jedné desce, je v dnešní době zlomek pořizovací ceny a nákladnější je samotný vývoj softwaru. Ale není to pouze o pořizovací ceně, rozměry nového hardwaru se razantně zmenšují a tím se hardware stává použitelnějším pro různé měřící zařízení nebo domácí spotřebiče. Dnes se takový hardware využívá v celé řadě domácích spotřebičů jako jsou varné konvice, lednice, televize. Díky jejich využití je pak možné tyto rozdílné zařízení například ovládat z jednoho místa nebo sbírat data o jejich využití, případně od nich získávat informace o okolí. Tyto možnosti umožňují zjednodušení každodenního života spoustě lidem. Typickým příkladem mohou být chytré domácnosti. Zde je využito tohoto potenciálu chytrých zařízení. Jedná se například o automaticky řízené osvětlení, kdy při nepřítomnosti osoby v místnosti se osvětlení automaticky vypne, nebo je ovládané v závislosti na okolní síle osvětlení. Dalším příkladem může být automatické vytápění v domácnosti. Tato zařízení nemusí nutně ovládat jiné předměty v domácnosti, ale mohou provádět sběr dat v domácnosti a poté provést zobrazení uživateli. Nové domácí systémy také přidávají možnost automatického zabezpečení domácnosti.

Takové systémy již existují řadu let. Jejich problémem však byla jejich vysoká pořizovací cena. Dnes se takové systémy stále více rozšiřují do domácností, a to právě díky jejich nižším pořizovacím nákladům než dříve, nižší energetické náročnosti, a hlavně více užitečných funkcí, které někteří ocení. S tím souvisí také používaný pojem *Internet of Things* (zkráceně IoT), přeloženo jako internet věcí. Většina těchto chytrých zařízení vyžadují pro své fungování komunikaci s okolním světem prostřednictvím internetové sítě. Počet chytrých zařízení se neustále zvětšuje a je zde snaha informace z těchto zařízení nějakým způsobem zpracovávat, uchovávat a analyzovat. Za tímto účelem vznikají systémy, které shromažďují a pracují s těmito daty. Problémem je však efektivita a rychlost takovýchto systémů. Požadavkem je obsloužení v reálném čase potenciálně tisíců takových zařízení, které zpřístupňují svá data. Vzniká zde také problém efektivního uložení takovýchto dat, které kontinuálně ve velkém počtu přicházejí od různých zařízení. Do vývoje takového systému pro IoT se zapojila i skupina z fakulty informačních systémů v Brně. Jejich projekt se pak zabývá inteligentní domácností. Součástí tohoto systémů je také databáze pro uložení nasbíraných dat ze senzorů chytré domácnosti, to je tématem této práce.

Cílem práce je vylepšení stávajícího řešení uložení senzorových dat v databází za účelem efektivnějšího uložení dat a zaměření se na rychlost zápisu a čtení dat z databáze. K tomu bude

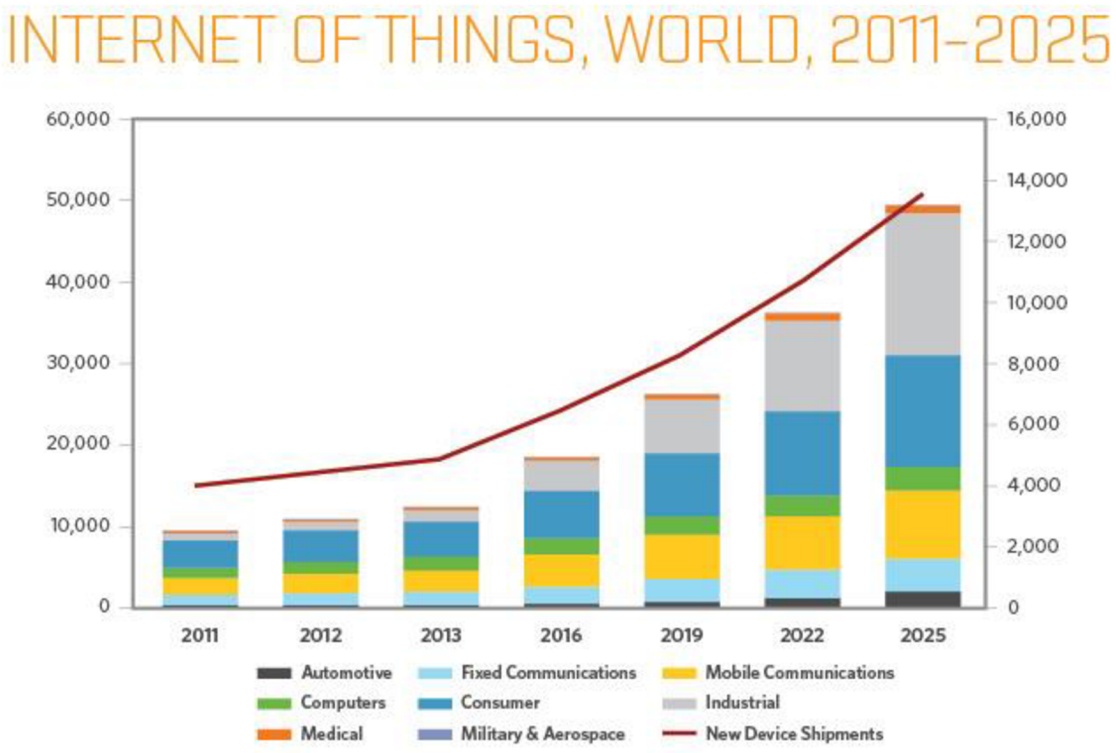
využito NoSQL databázového systému a uložení dat do této databáze založené na poznatcích dat časových řad.

V první kapitole práce bych se rád zaměřil na pojem internet věcí, inteligentní domácnosti, kterých se ve velké části týká právě ukládání a zpracování senzorových dat, které je tématem této práce. Následuje stručný popis systému BeeeOn, který je projektem fakulty informačních technologii v Brně. Poté se provede charakteristika NoSQL databází, čím se liší od klasických relačních databází a charakteristika jejich druhů. Obsáhlou kapitolu poté tvoří datové řady. Zde je popsáno, co to jsou datové řady, jak vypadají, jaké jsou požadavky na jejich ukládání a popis specializovaných databázových systémů, jež na tuto problematiku soustředí. Následuje popis stávajícího řešení ukládání senzorových dat v aplikaci BeeeOn a poté samotný návrh možného řešení problému s využitím NoSQL databáze. Kapitola implementace se zabývá konkrétními postupy řešení, které byly využity pro formát dat k uložení do databázového systému, práci s daty v operační paměti a implementované dotazy nad vyvíjeným systémem. Další kapitola je věnována výkonnostnímu porovnání mezi vlastním řešením založeném na NoSQL a původním řešením využívající relační databázi. V závěru jsou poté uvedeny a popsány další možné rozšíření, které by bylo možné implementovat do řešení.

Kapitola 2

Internet věcí

IoT je dnes vnímáno jako „další velká věc“. Od současných inteligentních domácností může situace dospět až k propojení celých chytrých měst. Není tedy pochyb, že má cenu se tímto tématem zabývat a jedná se o příslib do budoucna. Tvrzení mimo jiné podporuje také níže uvedené [*Obr. 1*] znázornění růstu tohoto trhu. Na obrázku lze pozorovat strmý nárůst v budoucích letech včetně podílu jednotlivých oblastí, které takové systémy nejvíce využívají. Pravá vertikální osa grafu poté odpovídá červené křivce, která odpovídá počtu nově prodaných zařízení v milionech kusů.



Obr. 1: Znázornění růstu chytrých zařízení zapojených do IoT. [1]

Definice tohoto pojmu bohužel není tak jednoduchá a jednoznačná. Jedná se však o koncept vzájemného propojení velkého počtu různých zařízení. Těmi zařízeními mohou být

například měřící senzory, domácí spotřebiče nebo také telefony. Při definici tohoto pojmu záleží na úhlu pohledu. Dle [2] se dá internet věcí chápat třemi způsoby:

Věcně orientovaný pohled, zde je bráno zaměření na jednotlivé věci, počínaje jejich identifikací, fungováním atd.

Internetově orientovaný pohled, zde je zaměření na komunikaci těchto zařízení s ostatními prvky sítě.

Sémanticky orientovaný pohled, jedná se o technologie pro uchování, vyhledávání a správa dat, jež jsou získány ze zařízení v síti.

Jak lze vidět z uvedených pohledů, IoT je velmi široký pojem a není zde zaměření pouze na jeden obor. Věcně orientovaný pohled nám bere v potaz zařízení, které jsou součástí sítě a sbírají data o svém okolí, které nám poté dávají k dispozici. Bez těchto zařízení by internet věcí nebyl vůbec možný. Taková definice dnes nepostačuje.

Pro propojení těchto zařízení je potřeba navrhnout protokoly pro vzájemnou komunikaci a vůbec celou strukturu sítě. To je možno brát jako samostatný pohled, kterým se zabývá spousta společností, kde se vyvíjí komerční, ale i volně dostupné komunikační protokoly.

Podstatnou částí je také uložení těchto sesbíraných dat. K tomu slouží databázové systémy. Zařízení generují velké množství dat, které je potřeba rychle a efektivně ukládat a také s nimi dále pracovat. Problémem je zde jak velké množství zařízení, která svá data poskytují, tak také v součtu jejich velký objem dat, jež je nutné uchovávat a zpracovávat jej. Nejedná se zde pouze o nárazové získání dat. Zařízení data odesílají v konstantních intervalech, kdy se jedná v závislosti na typu zařízení například o jednotky sekund. Všechno tohle má velký dopad na konečnou výkonnost stávajících databázových systémů, a tak vznikají jiné specializované databázové systémy, které řeší problémy uložení časových řad dat.

2.1 Inteligentní domácnosti

Jedná se o pojem, který spadá pod souhrnné označení internetu věcí. Existují řešení domácností, které se zabývají například zabezpečením. Ta poté shromažďují informace od požárních čidel, stavu zámků apod., v případě nastání nějaké nestandardní situace by systém například upozornil uživatele. Pod automatizaci může spadat např. řízené vytápění domu, případně oken, osvětlení a mnoho dalších úkonů. Také v domácnosti umožňují řídit komunikaci a ovládat zábavné systémy z jednoho místa.

V tuto chvíli existují řešení pro chytré domácnosti, které zastřešují sběr dat a řízení komunikace s různými druhy zařízení. Výčtem pár dostupných řešení:

- Nest od společnosti Gooogle
- SmartThings, podpora ZigBee, Z-Wave a další
- WeMo vyvíjeno společností Belkin

Výčet by bylo možné ještě dále rozšířit, ale to není náplní této práce. Většina z těchto systému je však uzavřena. Systém pak dokáže spolupracovat pouze se zařízeními určitých výrobců nebo vlastními. Mnoho z nich je také prodáváno komerčně. Nás však v rámci této práce zajímá řešení vyvíjené skupinou na fakultě informatiky v Brně pod názvem BeeeOn.

2.1.1 BeeeOn

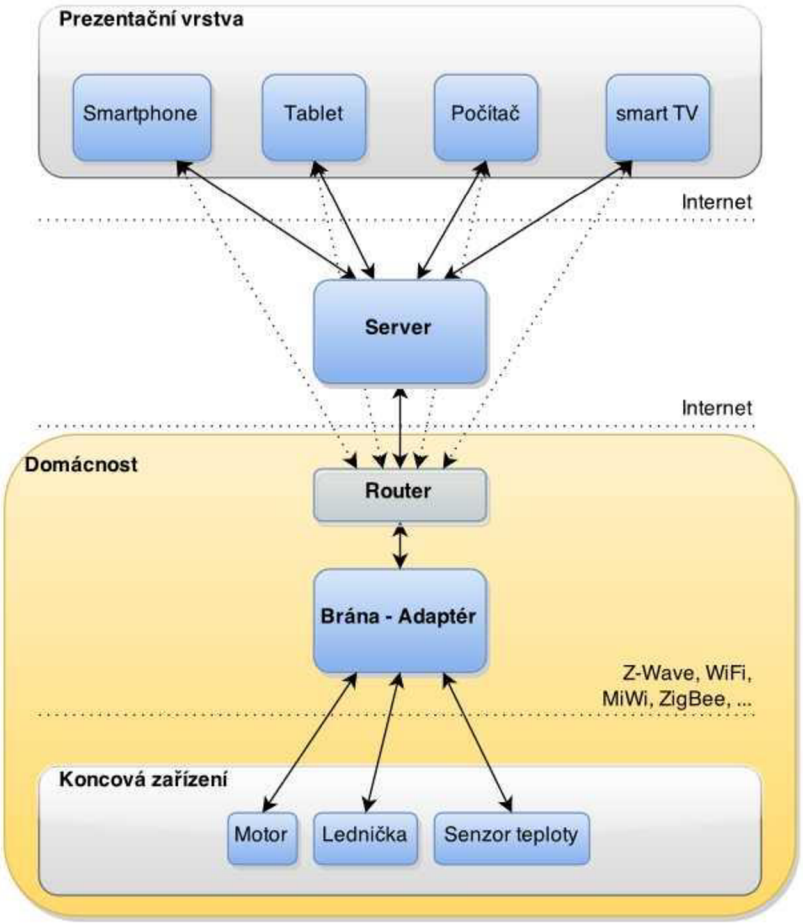
Jedná se o systém, který je vyvíjený tak, aby byl schopen spolupracovat s širokou řadou senzorů, včetně zařízení třetích stran, a shromažďoval jejich data v databázi. Server dále bude moci provádět zpracování a různé výpočty nad senzorickými daty. Důležitým aspektem systému je také prezentační vrstva, kde systém data poskytuje uživateli např. na telefonu, nebo jiném zařízení. Součástí systému je také oznamování událostí uživateli formou notifikací.

Systém BeeeOn se tedy dělí na tři úrovně, kde centrem je serverová část. Ta provádí získání dat včetně jejich zpracování a výpočty nad nimi. Obsahuje také databázi pro uložení dat o samotném systému (topologie sítě senzorů, uživatelé, nastavení atd.) a poté již samotná data senzorů.

Další funkcí serveru je poskytovaní dat prezentační vrstvě, která zajišťuje komunikaci s uživatelem. Zde se jedná jak o ovládání, tak také zobrazení naměřených dat a aktuálních informací o domácnosti. Zmíněná prezentační vrstva znázorňuje zařízení uživatele, na kterém budou zobrazeny informace o systému. K dispozici je například mobilní aplikace.

Server je dále propojen s další úrovní, což je domácnost. Tato vrstva již obsahuje samotné senzory, které se využívají pro sběr informací nebo ovládání okolí.

Již Pro tuto práci je podstatnou vrstvou serverová část, která obsahuje právě databázi s uloženými senzorovými daty.



Obr. 2: Struktura systému BeeeOn. [3]

Kapitola 3

NoSQL databáze

Pro další části práce je důležité porozumění samotnému pojmu NoSQL databáze. Jedná se o zkratku „Not Only SQL“, což volně přeloženo znamená „Nejen SQL“. [4] Tento termín nemá nějakou přesnou definici, jedná se o zkratku vytvořenou Johanem Oskarssonem, který toto označení použil v roce 2009 na sociální síti Twitter „*#nosql*“ pro chystanou konferenci o databázích, které nebyly založeny na tradičním relačním schématu. Toto označení se již poté uchytilo. NoSQL databáze se tedy nekategorizují podle nějakých striktních pravidel, které musí splňovat. Naopak, všechny se od sebe liší. Neplatí zde žádný standardní dotazovací jazyk jako je SQL pro relační databáze. Databázové systémy skrývající se pod tímto označením, ale sdílí některé společné znaky [Obr. 3]. Není však nijak vyžadováno, zda splňují všechny uvedené znaky. Existují NoSQL databáze, které jsou například komerční, nebo které nepodporují distribuovatelnost.

Obr. 3: Společné znaky NoSQL databází.

NoSQL databáze nemají relační schéma, nejedná se o propojení tabulek na základě cizích klíčů. Poskytují však výhody při práci s velkým objemem dat, protože jejich systém uložení dat je jednoduchý. Většina těchto systému umožňuje distribuci dat na více disků, clusterů. Data se typicky nemusí ukládat v nějakém definovaném schématu. U relačních databází se musí první vytvořit tabulky a sloupce této tabulky poté mají definovaný formát a typ. Naopak některé NoSQL databáze tento předpis nevyžadují a získaná data uloží do jednotné tabulky například s využitím formátu JSON pod nějakým klíčem. Nedochází zde tedy k nějakému pochopení dat databázovým systémem, nýbrž pouze jejich uložení. Data jsou poté získány použitím klíče řádku.

Dnes se NoSQL těší velké popularitě na pozadí webových stránek, u internetových firem jako Google, Amazon, nebo případech, kdy se pracuje s tzv. *BigData* (práce s velkým množstvím dat). Použita je také v cloudovém databázovém systému Firebase, jež je k dispozici pro operační systém Android, iOS a webové stránky. Příchodem NoSQL, ale nedochází k nahrazení klasického relačního schématu, naopak dochází k jejich vzájemnému doplnění.

3.1 Charakteristiky NoSQL databází

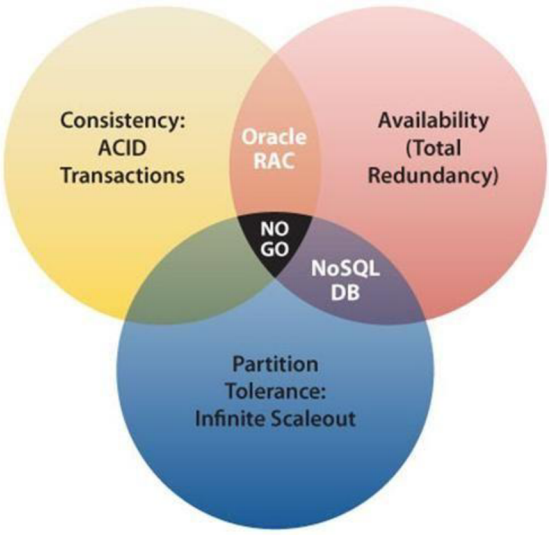
[5] Klasické relační databázové systémy jsou transakční, čímž zaručují integritu dat. Je zaručena také konzistence dat v každé situaci. Tyto transakční charakteristiky jsou také známy pod zkratkou známou jako *ACID* (atomicita, konzistence, izolace, trvanlivost). Nicméně zde dochází k problému, se zachováním těchto pravidel, docílit škálování do šířky takovýchto databázových systému. Dochází zde k nárůstu chyb a konfliktů mezi uloženými daty, při vysoké dostupnosti v distribuovaných datech za účelem vyšší dostupnosti. Jedná se o ne kompletně řešitelný problém nazvaný jako CAP teorém [5].

Silná konzistence (consistency), všichni klienti vidí stejnou verzi dat i po aktualizaci databáze.

Vysoká dostupnost (availability), všichni klienti mohou nalézt alespoň jednu kopii požadovaných dat i v případě, že některý z clusterů je nedostupný.

Tolerance k dělení dat (partition tolerance), celý systém si zachová charakteristiky, i když je přesunut na jiný server.

CAP teorém nám říká, že je možné v jeden čas rozšiřovat pouze dva z těchto tří aspektů, tak aby bylo dosaženo jejich plné funkčnosti. Znázorněno na [Obr. 4]. Většina NoSQL databází se rozhodla vynechat podmínku konzistence dat, aby bylo dosaženo větší dostupnosti a dělení dat do clusterů. Tato skutečnost je poté vyjádřena termínem *BASE (Basically Available, Soft-state, Eventually consistent)*.



Obr. 4: Znázornění CAP teorému. [6]

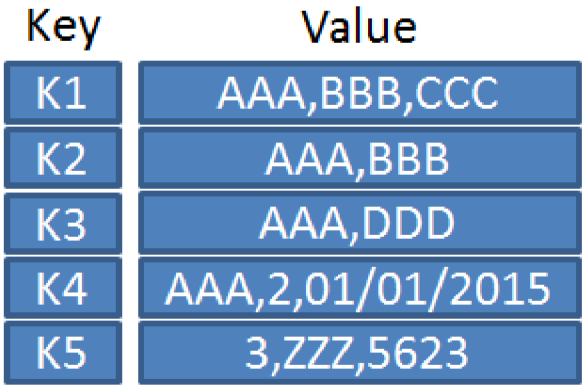
3.2 Klasifikace NoSQL databází

NoSQL databáze lze podle jejich vlastností klasifikovat do čtyřech základních typů. Většinou se jedná o rozdělení do typů podle jejich uložení dat. Každý z nich má své výhody a nevýhody, případně jiný model použití.

3.2.1 Uložení klíč-hodnota

[7] Prvním typem je uložení stylem klíče, ke kterému je přiřazena nějaká hodnota. Klíč je většinou alfanumerického tvaru, pomocí něj se poté odkazuje do tabulky (forma hashovací tabulky). Hodnota je poté uložena formou textového řetězce, nebo jiným více komplexním typem. Výhodou tohoto jednoduchého uložení dat je velmi rychlý přístup k požadovaným datům. Využití nalezne například při správě uživatelských účtů, získání názvu položky. Nevýhodou přístupu pomocí klíče je poté skutečnost, že nelze v databázi vyhledávat podle hodnot. Hodnota v tomto případě není databází nijak pochopena a není možné ji tak například porovnávat. Veškerý přístup k datům se tak děje pouze pomocí klíče.

Příkladem databáze tohoto typu jsou například DynamoDB, Riak, Oracle NOSQL database a mnoho dalších.



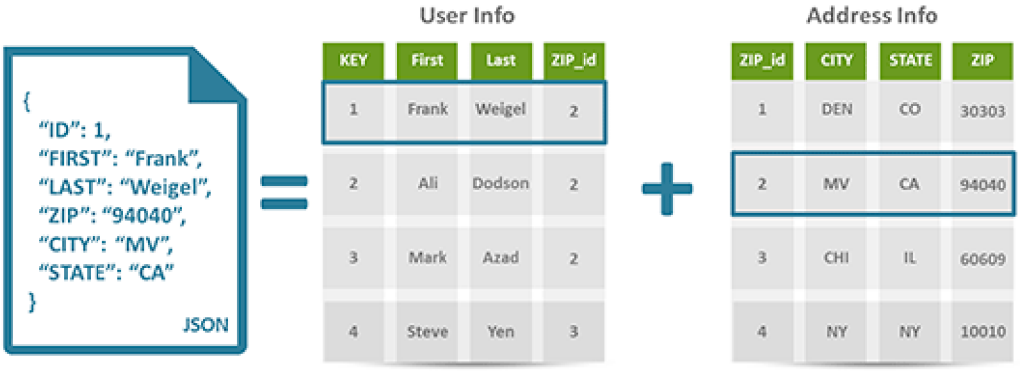
Obr. 5: Schéma databáze klíč-hodnota. [7]

3.2.2 Uložení dokumentů

[7] Koncept NoSQL databází fungující na ukládání dokumentů je velmi podobný databázi typu klíč-hodnota. Dokumenty bývají běžně uloženy ve formě strukturovaného textu, typické formáty, jež se běžně používají jsou XML, YAML, JSON, nebo případně také v binární formě jako je například BSON. Dokumenty jsou v databázi adresovány pomocí nějakého unikátního klíče, který reprezentuje daný dokument. Strukturovaná forma hodnoty je právě odlišná od nestrukturované podoby hodnot u databází typu klíč-hodnota. Díky známé struktuře dat je potom možné provádět vyhledávání dat na základě známého klíče, nebo prohledáním dokumentu. V paměti by byla možná implementace pomocí jisté formy kolekce dokumentů, která umožňuje přístup na základě klíče. V objektově orientovaných jazycích se takovéto typy běžně vyskytují jako hash mapy.

Znázornění lze vidět na [Obr. 6], kde je uložen dokument obsahující osobní informace. Dokument je uložen ve strukturovaném formátu JSON a obsahuje vždy atribut a hodnotu. Ekvivalentní reprezentace v relačním databázi je poté vidět vpravo.

Známými příklady jsou databáze MongoDB, CouchDB, Elastic.



Obr. 6: Znázornění schématu dokumentové databáze obsahující osobní informace. [9]

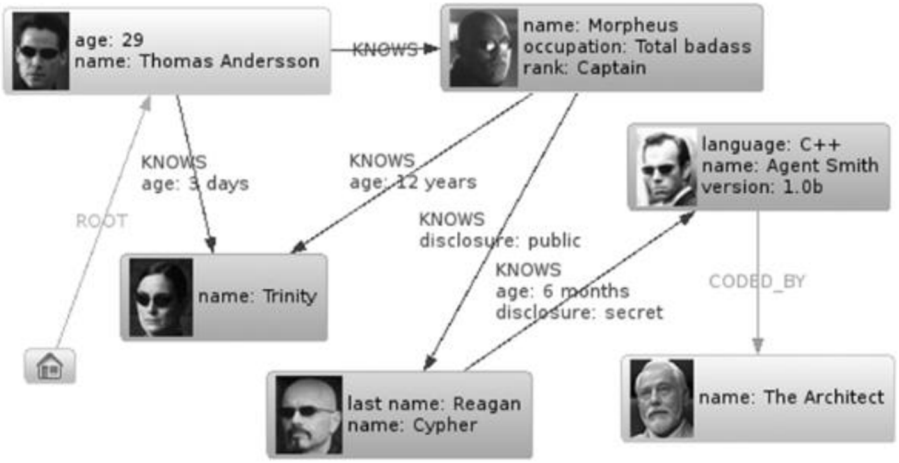
3.2.3 Široké sloupce

[5] Jedná se zde o hybridní přístup k uložení dat, jež je svým typem mezi relačním modelem a databází typu klíč-hodnota. Tento typ databáze ukládá data jako sekce ve sloupci tabulky místo v řádku. Tento typ uložení je využit pro ukládání velkého množství dat rozprostřeného do více clusterů. Výhodou je rychlejší dotazovaní na každý sloupec a jejich zpracování. Samotné vkládání je však pomalejší. Typickými příklady může být databáze Cassandra, Scyla nebo Druid.

Obr. 7: Znázornění databáze se širokým sloupcem. [5]

3.2.4 Grafové databáze

[8] Je užívaný nad daty, které mají povahu vztahů mezi sebou. Například linka veřejné dopravy, vztahy mezi lidmi. Jsou podobné objektově orientovanému typu databáze. Vztah je zde znázorněn hranou mezi objekty a objekty poté reprezentují atributy jako páry klíče a hodnoty. Reprezentanty jsou Neo4J, ArangoDB, InfoGrid.



Obr. 8: Znázornění grafu. [5]

3.3 Nevýhody NoSQL

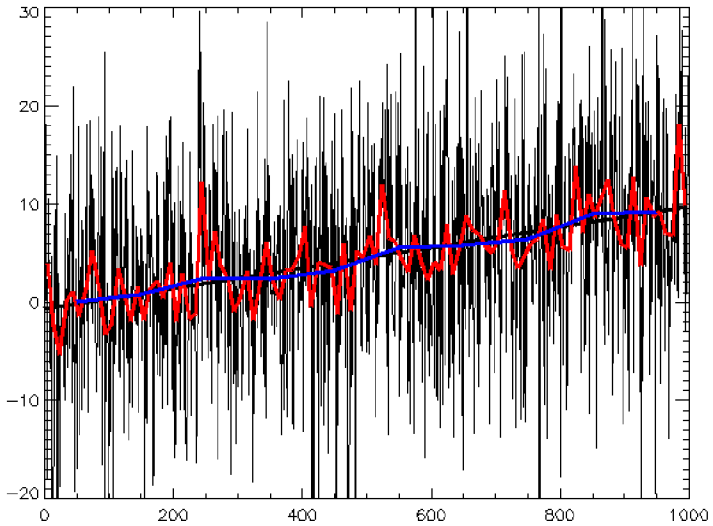
Klasické SQL již má mnohaletou tradici a za ta léta se jeho vývoj ustálil. Koncept uložení do tabulek a provázání tabulek pomocí cizích klíčů je stále stejná a neměnný. Dotazovací jazyky mezi různými typy SQL databází jsou velmi podobné. Proto je přechod z jednoho systému na druhý jednoduchý a není potřeba, žádné větší studování nového databázového systému. NoSQL databáze, jak je patrné z příkladu výše, jsou samy od sebe velmi odlišné, nemají žádný společný dotazovací jazyk a každý typ má jiný formát uložení dat. Tyto vlastnosti poté tvoří problém s přechodem mezi jednotlivými jejich typy, případně i porozuměním, jakým způsobem jsou data reprezentována.

Kapitola 4

Časové řady

Tématem práce je uložení senzorových dat. Data, která tyto senzory produkují mají právě charakter časové řady. Jedná se o sekvenci časových bodů, sbíraných v pravidelných časových intervalech a indexovaných seřazeně podle jejich času vzniku. Časové intervaly pro měření závisí na konkrétním využití, ale mohou se pohybovat v některých případech několik měření za jednu sekundu, až k intervalu v řádech několika sekund, případně hodin nebo dnů. Takováto data se skládají ze dvou hodnot, naměřená hodnota a čas provedení měření. Zjednodušeně je možné vyjádření, že se jedná o všechny data, která obsahují nějaké časové razítko určující jejich vytvoření. To je případ právě již zmíněných senzorových dat, která patří do skupiny IoT. Každá senzorem zaznamenaná hodnota je opatřena mimo jiné kromě identifikátoru původce dat a samotné naměřené hodnoty také časem, kdy bylo provedeno ono měření. [10] Dalšími typy akcí, které produkují časové řady mohou být například hodnoty akcií, hodnota komodit, měření přílivu nebo počty slunečních skvrn.

Pro zjištění více informací se nad těmito daty musí provádět agregace a analýza. Ačkoliv se nejedná o žádný nový typ dat, právě růst v oblasti internetu věcí a využití senzorů ukázal, že je potřeba pro tyto data přizpůsobit nebo navrhnout nové databázové systémy, které zvládnou zpracovat veliké množství dat přicházejících z potenciálně tisíců zařízení. Data časové řady jsou často znázorňována formou grafu [Obr. 9]. Graf na své svislé ose obsahuje naměřené hodnoty a na vodorovné ose čas jejich výskytu.



Obr. 9: Ukázka vykreslení dat časové řady. [11]

4.1 Smysl uložení časových řad

Smysl sběru dat časových řad je závislý na konkrétním oboru. Pro meteorology, seismology, geofyziky mají tyto data cenu pro předpověď počasí. Z pohledu dolování dat je možné tato data použít například pro shlukovací metody, případně klasifikaci.

Provedením analýzy nad těmito daty je tedy možné například předpovídat nějaké budoucí chování sledovaného systému, detekovat různé anomálie, provádět klasifikaci detekovaných vzorů chování do specifických kategorií, provádět regresní analýzu nebo také segmentaci. Konkrétním případem využití těchto dat může být v letectví, kdy jsou data ze senzorů (výškových, motoru atd.) využita k rekonstrukci událostí, jež předcházely chybě.

4.2 Požadavky na databázi

Jedním z požadavků je, že databáze musí zvládnout pracovat s daty, které neustálé přirůstají. NoSQL databáze zde vytláčejí právě díky požadavkům na škálovatelnost tradiční relační databáze, které se nejsou schopny tomuto požadavku přiblížit, tak aby byla zachována vlastnost ACID. Avšak ukládání dat časových řad přináší další problémy, které nejsou ani některými NoSQL databázemi splněny.

Dalšími poznatky pro databázi určenou k uchování dat časových řad mohou být následující. Již uložená historická data se typicky dále v čase nemění, jejich editace ve většině případů není u databáze vyžadována. Často také není potřeba ani jednotlivé mazání záznamů. Požadavkem je však smazaní historických dat od určeného data dále. Tohle je vyžadováno u aplikací, kterým záleží na aktuálních datech, případně na datech určitého stáří. Skladování velkého počtu historických dat by tak nedávalo smysl, protože by nebyla využita a docházelo by ke zbytečnému zatížení databáze.

Vybrat pro řešení tohoto problému vhodnou NoSQL není jednoduché a je zde vyžadováno spousta dalších podmínek, které by měla vhodná databáze splňovat [10]:

Lokace dat: pokud jsou na sobě závislá data uložena na jiných fyzických uložištích, dotazy nad těmito daty mohou být pomalé, případně mohou vést k time-outům. Proto je vhodné, aby databáze udržovala na sobě závislá data na jednom fyzickém uložišti poblíž sebe, což zapříčiní rychlejší vykonání dotazů.

Optimalizace pro dotazy nad časovým rozsahem: při analýze dat časových řad je často vyžadována právě schopnost databáze poskytovat výsledky nad dotazy, které zahrnují výběr dat v časovém rozpětí od-do. K tomu, aby byl takovýto dotaz spolehlivě vykonán, je vyžadována také právě podmínka lokace dat, jež je popsána výše. Výběr v tomto případě může ovlivnit i přívětivost formátu dotazovaní jež aplikace nabízí.

Vysoká rychlost zápisu: spousta databází není schopno rychle obsluhovat požadavky během špiček provozu. Právě distribuované NoSQL databáze jsou dobrou volbou pro zajištění vysoké dostupnosti a výkonu pro zápis i čtení během výkonnostních špiček.

Kompaktní data: spolu se stárnoucími daty se snižuje jejich granularita. U starších dat již není vyžadována taková přesnost. Redukcí dat v databázi dojde ke zvýšení jejího výkonu.

4.3 Databáze časových řad

K dispozici jsou však specializované databázové systémy, které právě problémy týkající se uložení a práce s časovými řadami řeší. To že se jedná o důležitý a netriviální problém, který je třeba řešit, svědčí i to, že některé tyto databázové systémy jsou k dispozici jako komerční nástroje. Benefity těchto specializovaných databází mohou být [10]:

Masivní škálovatelnost a výkon: efektivní databáze dokáže obsloužit až milion zařízení nebo bodů časové řady v neustálém proudu dat včetně vykonání analýzy dat v reálném čase.

Redukce výpadků: v situaci, kdy je výpadek neakceptovatelný je databáze schopna zajistit a být k dispozici. To je například při selhání části sítě, nebo chyby hardwaru.

Redukce nákladů: efektivním provozem a dobrým využíváním zdrojů, může snížit náklady na provoz.

Vylepšení rozhodování řízení: díky analýze dat v reálném čase, může mít pozitivní dopad na řízení a rozhodování firmy.

Většina specializovaných databází, jež jsou určené pro časové řady, jsou většinou založeny na již existující implementaci databázového serveru a jsou rozšířeny o „driver“, který běží právě nad tou databází. Zkoumané databáze časových řad byly ve většině případů založeny na již existujících databázích typu NoSQL. Není to však pravidlem a k dispozici jsou i systémy, které jsou postavené na relační SQL databázi.

4.3.1 PhilDB

Prvním příkladem může být malá a jednoduchá databáze pro časové řady pod názvem PhilDB. Databáze je napsaná s využitím jazyka Python. Jedná se o aplikaci, která je založená na již stávajícím databázovém serveru SQLite. Jeho výhodami by měla být snadnost instalace, kterou je možné provést přes repositář pythonu a snadnost použití. Zde je uveden výpis funkcí PhilDB [12]:

- Metoda pro zápis akceptující sérii objektů, frekvenci výskytu a atribut pro zápis nebo možnost editace časové řady.
- Metoda pro čtení jedné časové řady, založena na identifikátoru časové řady, četnosti výskytu a atributu.
- Pokročilé čtení pro čtení kolekce časových řad.
- Podpora pro data s pravidelnou i nepravidelnou četností výskytu.
- Logování nových a změněných hodnot.
- Metoda pro čtení logu k extrakci časové řady s daným datem.

Formát uložení dat v případě PhilDB vypadá následovně. K datům časové řady jsou získána jejich meta-data, která jsou uloženy v relační databázi SQLite. Data časové řady jsou poté uloženy na disk jako prostý databázový soubor tzv. „flat file“. V tomto případě se jedná o prostý binární soubor, není to však pravidlem a data mohou být v takovém souboru reprezentovány formou prostého textu. Ke zjištění cesty k požadovanému souboru časové řady, jsou soubory indexovány pomocí uložených meta-dat časových řad. V souboru je poté uložení dat formou

třech hodnot typů: „long“, „double“, „int“, pro každý záznam. Typem „long“ je reprezentována časová značka dat, „double“ zde reprezentuje naměřenou hodnotu a „int“ je doplňující meta informace k datům. Individuálně provedené změny v databázi jsou logovány v HDF5 souboru, který je uložen vedle souborů s daty časové řady. Databáze jinak neprovádí žádnou kompresi dat, provádí pouze jednoduché vkládání formou jednotlivých záznamů. Komunikace s aplikací probíhá pomocí poskytnutého rozhraní, není zde žádná implementace pro komunikaci po síťové vrstvě.

4.3.2 KairosDB

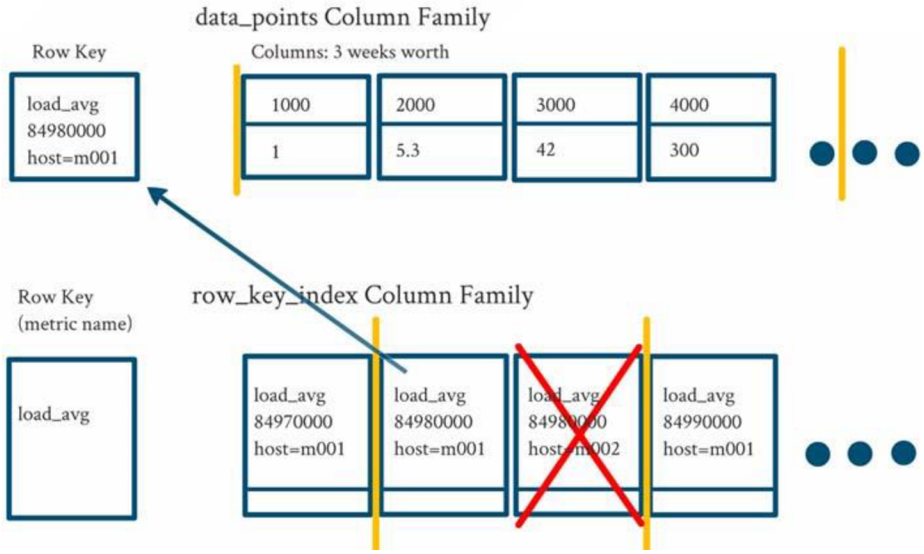
V tomto případě se již jedná o větší projekt. Databáze KairosDB je mezi firmami hojně využívaná a je také k dispozici zdarma. Databáze staví na NoSQL databázi Cassandra, která se řadí do typů databází s širokými sloupci. Samotná aplikace je poté napsaná v jazyce Java a je dobře škálovatelná. Proti PhilDB nabízí KairosDB více funkcí [13]:

- K dispozici mnoho protokolů pro nahrávání dat do databáze: *Telnet*, *Rest*, *Graphite*, případně je zde možnost využití pluginů třetích stran.
- Ve standardní instalaci je k dispozici webové rozhraní.
- Dostupnost agregačních funkcí jako je *min*, *max*, *sum*, *count*, *mean*.
- Nástroje pro import a export, sledování výkonu.
- Rozšiřitelné pomocí pluginů.

Schéma databáze Cassandra se v KairosDB skládá ze třech typů sloupců:

- *data_points:* pro uložení dat
- *row_key_index:* k indexování řádku během dotazu
- *string_index:* použito k informaci o typu atributů a metrik v systému.

Klíč řádku je vytvořen konkatenací názvu metriky, časového razítka a typu datového uložiště. Podle tohoto klíče se získá patřičný řádek v *data_points* sloupcích. Jeden řádek databáze je nastaven tak, aby zvládl uložit data až ze tří týdnů.



Obr. 10: Schéma uložení dat v KairosDB. [14]

4.3.3 OpenTSDB

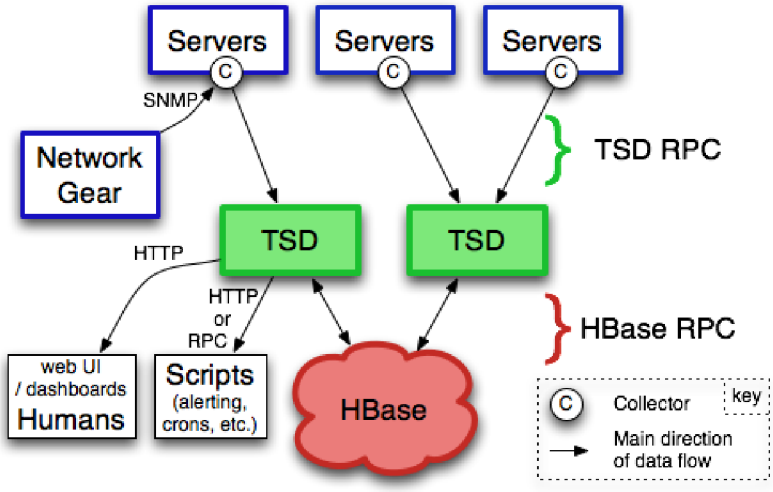
OpenTSDB je distribuovaná a škálovatelná databáze učená právě k uložení dat časových řad. Použití je vhodné pro monitoring a měření, senzorová data, finanční data, výsledky vědeckých experimentů. K databázi uvádí, že je schopna uložit triliony datových bodů, neztrácí přesnost dat a dobře škáluje díky využití HBase [15]. HBase je volná, nerelační, distribuovaná databáze vyvíjená společností Apache Software Foundation. Stejně jako Cassandra u KairosDB se řadí do NoSQL databází s typem ukládání širokých sloupců. Nástroj využívá řada velkých společností jako box, tumblr, ebay, Pinterest. Vkládání dat je možné provádět pomocí:

- *Otevřením socketu podobnému telnetu a zapsaní:* put sys.cpu.user 1234567890 42 host=web01 cpu=0
- *Přidání JSONu:* http://<host>:<port>/api/put
- *Nahrání velkých souboru pomocí CLI*

Není vyžadována definice žádného schématu. Klíč řádku je složen konkatenací „*metriky* + *časového razítka* + *tagk1* + *tagv1* + ... + *tagkN* + *tagvN*“, kde metrika je identifikátor, *tagk* reprezentuje parametr, např. *host* a *tagv* poté jeho hodnotu, např. *web01*. Jeden uložený záznam obsahuje shluk datových bodů zaznamenaných v průběhu jedné hodiny. Ukládáním rozmezí jedné hodiny do jednoho řádku se zajistí, že podobná data budou ve své blízkosti a bude možné je získat např. jedním dotazem. Reprezentace jednoho datového bodu obsahuje:

- *Název metriky*
- *Unixové časové razítko*
- *Hodnotu typu 64-bitového integeru, nebo float*
- *Set tagu (klíč-hodnota), pro popsání časové řady*

Na schématu [Obr. 11] je vidět, že systém se skládá z Time Series Daemon (TSD), kterých může běžet více a jsou na sobě nezávislé. Každé TSD přijímá příchozí dotazy a poté komunikuje s uložištěm HBase, které je distribuované mezi více uložišť.

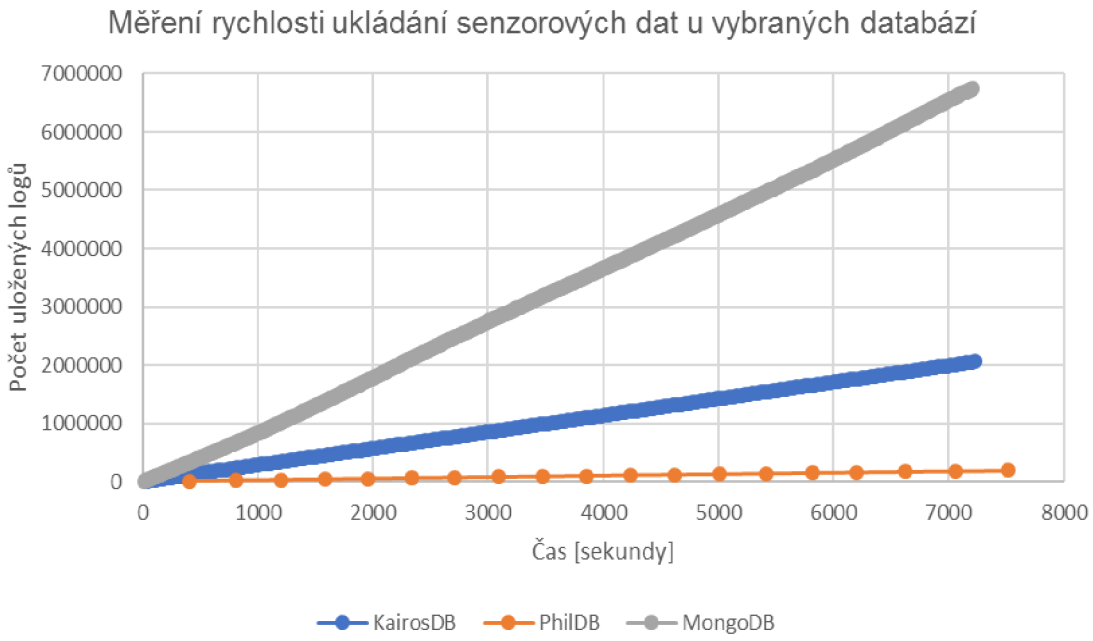


Obr. 11: Schéma struktury OpenTSDB. [15]

4.3.4 Měření rychlosti zápisu

Pro měření výkonnosti při zápisu vybraných databázových systému posloužila originální senzorová data ze systémů BeeeOn, které mám k dispozici. Originální databáze byla načtena do systému PostgreSQL a byla provedena operace *SELECT* nad všemi dostupnými senzorovými daty. Data byla načítána z databáze po 10000 položkách a poté ukládána do testované databáze časových řad. Postup se poté opakoval. Načítání nových položek a ukládání do databáze probíhalo pro každý testovaný systém dvě hodiny. Vždy se po určité době zaznamenal počet již uložených položek a čas běhu. Celý testovací program byl napsán v jazyce Python, pro který má podporu většina databázových systémů, ať již nativní nebo vytvořenou třetími stranami. Měření probíhalo na systému Ubuntu 16.04, který byl virtualizován pomocí nástroje VirtualBox. Systému bylo přiděleno k práci 8 GB RAM a 2 fyzická jádra procesoru.

Testy prošly tři vybrané databázové systémy a to KairosDB, PhilDB a MongoDB. Původním záměrem bylo provést testování i databáze OpenTSDB, tento nástroj se však nepodařilo zprovoznit. K testování byla vybrána i databáze MongoDB, jakožto zástupce dokumentové databáze. MongoDB není přímo určeno pro práci s časovými řadami, ale bylo zařazeno z důvodu porovnání s nástroji specializovanými pro ukládání dat časových řad.



Graf 1: Výsledek měření při vkládání dat do databáze.

Z naměřených hodnot zobrazených v [Graf 1] lze vidět odlišná výkonnost různých systémů. U databáze PhilDB se na jeho efektivitě projevila jeho optimalizace pro nahrávání celých souborů do databáze, místo jednotlivých záznamů. Nebylo tak u ní dosaženo ani 1 milionu uložených měření za dvě hodiny. Konkrétně se jedná o hodnotu 190 tisíc logů. KairoDB si vedla mnohem lépe. Tato databáze byla schopná v průběhu dvou hodin uložit lehce přes 2 miliony datových bodů. Nejlépe si vedla databáze MongoDB, která datové body pouze vkládalo do své dokumentové tabulky ve formě strukturovaného JSONu. Její výkonnost dosahovala k 7 milionům uložených datových bodů. Měření bylo samozřejmě zkresleno prováděním operace *SELECT* nad

systémem PostgresDB a až poté docházelo k jejich uložení do vybraných databázi, podmínky ale měla každá testovaná databáze stejné, a tak jde o chybu měření konstantního charakteru.

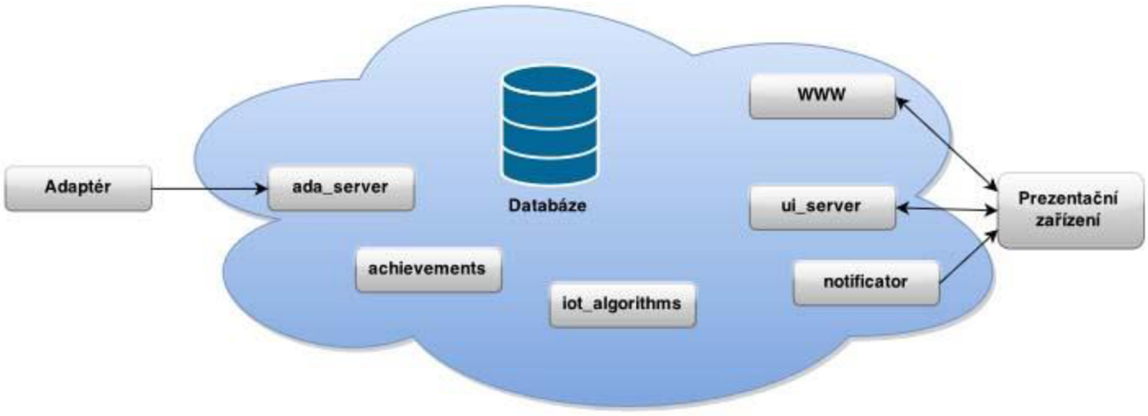
Z měření lze vyvodit, že je možné pro databázi zaměřenou na ukládání časových řad využít dokumentové NoSQL systémy, jako je MongoDB, a v případě implementace efektivní nadstavby nad standardní NoSQL databází, může být tato implementace velmi rychlá a srovnatelně výkonná se specializovanými systémy jako KairosDB.

Kapitola 5

Současná databáze

Práce se zabývá vytvořením vlastního databázového systému, který bude optimalizovaný pro práci s daty časových řad. Použití pro tuto databázi by bylo možné nalézt právě ve školním projektu BeeeOn, který je popsaný výše. Z tohoto důvodu bylo potřeba získat informace o současném databázovém systému, který se v projektu využívá, co obsahuje a jaké jsou požadavky.

Serverová část projektu BeeeOn je implementovaná v jazyce C++. Obsahuje komunikaci se zařízeními pro sběr dat, které jsou součástí chytré domácnost. Komunikace probíhá skrz adaptér a získaná data jsou poté uložena do databáze, která je součástí serverové vrstvy. Server dále poskytuje uložená data dalším zařízením, která slouží jako prezentační vrstva pro uživatele systému. Může se jednat například o telefon. Server taktéž obsahuje implementace metod pro analýzu naměřených dat, správu uživatelů apod.



Obr. 12: Znázornění zjednodušeného schématu serverové aplikace BeeeOn. [17]

Současná implementace databáze v projektu BeeeOn je založena na relační databázi PostgreSQL. Důvodem využití relačního schématu je, že databáze neobsahuje pouze uchování senzorových dat, ale také tabulky pro uložení uživatelů, zařízení, lokalizací, notifikací apod.
Část databázového systému znázorňující uložení senzorových dat je vyobrazena na [Obr. 13].

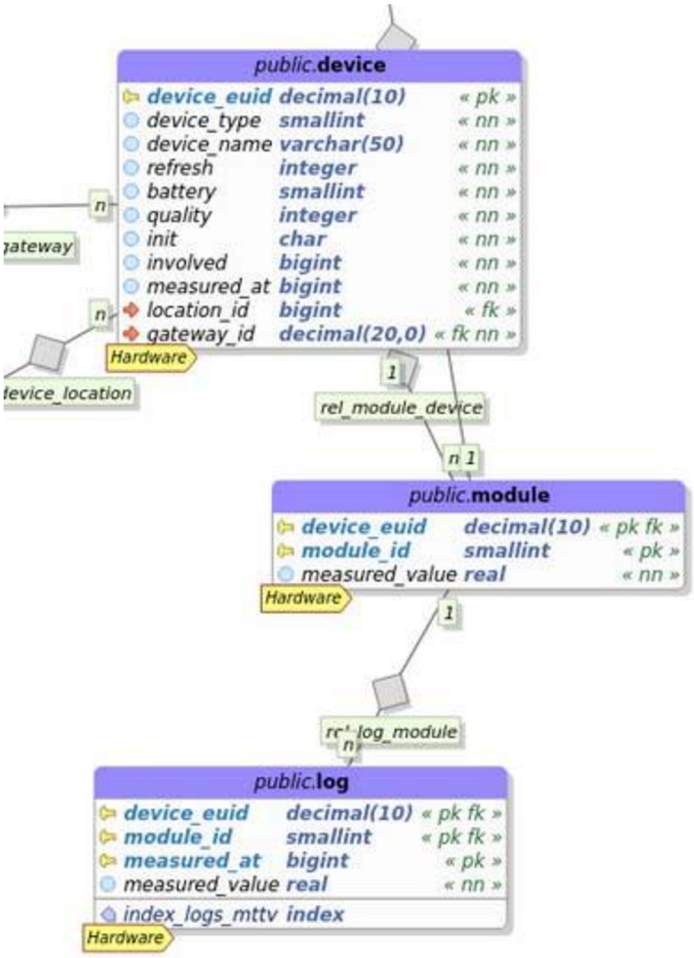
V této části jsou obsaženy tři tabulky:

- device
- module
- log

Tabulka *device* uchovává informace o jednotlivých zařízeních pro sběr dat, které jsou k systému připojeny. Konkrétně se jedná o uchování atributů *UID* tohoto zařízení formou decimální hodnoty, poté *typ_zařízení*, *jméno_zařízení*, *doba_periody_měření,* která určuje, v jakých časových intervalech probíhá měření dat. Atribut *quality* určuje přesnost naměřených dat. Navržená databáze však nebude určena pro správu senzorů, ale pro uložení jejich naměřených dat. Vyjmenované atributy budou však potřeba i pro NoSQL databázi, aby bylo možné s daty pracovat a porozumět jim.

Konkrétní zařízení se skládá z jednoho nebo více modulů. Moduly mají své vlastní *UID* a jsou spojeny se zařízením pomocí cizího klíče *UID_zařízení*. Částí modulu je taktéž poslední naměřená hodnota daného modulu. Ta je zde uložena kvůli rychlejšímu zobrazení této hodnoty v uživatelské aplikaci.

K modulu je dále vázána tabulka *log*, která obsahuje samotný záznam naměřené hodnoty senzoru v určitý čas. Tabulka *log* je provázána s tabulkami *device* a *module*. Vazba mezi tabulkami je vytvořena pomocí cizích klíčů *UID_zařízení* a *UID_modulu*, ke kterým se naměřená hodnota vztahuje. Každé provedené měření senzoru tak je uložena v tabulce *log,* a kromě naměřené hodnoty a časového razítka, které určuje čas provedení měření, obsahuje tabulka také identifikátory zařízení a modulu, které určují konkrétní senzor a jeho modul jemuž uložený data náleží. Senzorem naměřená data jsou typu *real*. Reprezentace časového razítka je pomocí unixového času.



Obr. 13: Diagram současného uložení dat senzorů v relační databázi. [16]

Kapitola 6

Návrh databáze

Na základě zjištěných informací bylo potřeba navrhnout efektivní uložení senzorových dat s využitím NoSQL databáze, v závislosti na vlastnostech senzorových dat, a tak aby byly zajištěny potřebné požadavky na systém.

6.1 Požadavky na databázi

Požadavky pro potřeby projektu BeeeOn jsou následující:

- Efektivní uložení, co se týče využitého prostoru.
- Rychlé dotazovaní nad daty.
- Rychlé ukládání dat.
- Výhodou je také možnost horizontální škálovatelnosti systému.
- Možnost propojení s aplikacemi na serveru.

Databáze si musí umět rychle poradit s dotazy výběru jednotlivých záznamů z databáze a taktéž výběru dat v určeném časovém rozpětí. Dotaz na časové rozpětí měřených dat společně s agregací hodnot se nevyužívá. Patřičné operace jako je změna měřítka, průměr, součet, se nad daty provádí až při vizualizaci v grafech, kdy si uživatel definuje možnosti zobrazení dat.

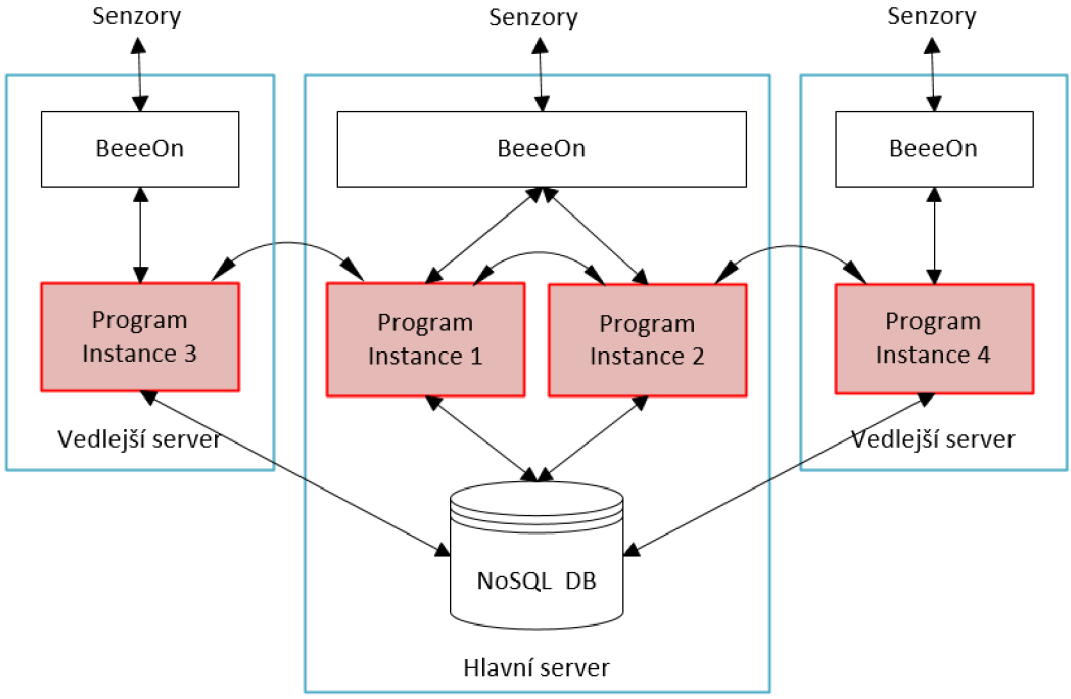
Požadovanou funkcionalitou je dále automatické mazání zastaralých dat. Mazání probíhá v nočních hodinách, kdy má aplikace nízké využití a mažou se data starší 14 dnů.

Pro zjištění podrobnějšího využití databáze, jaké přesně a jak často se provádí různé dotazy nad daty, by bylo vhodné do současné databáze nainstalovat monitorovací systém, který by tyto informace zaznamenal. Na jejich základě by bylo možné provést lepší optimalizaci nové databáze.

6.2 Návrh aplikace

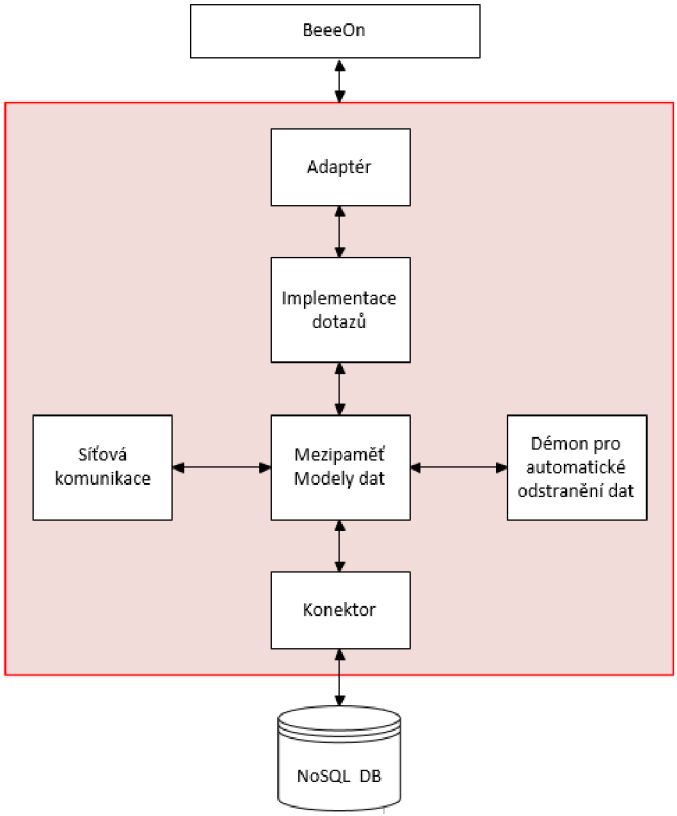
Aplikace je navržena tak, aby fungovala formou knihovny, která se připojí ke stávající serverové aplikaci. Poskytováno bude rozhraní pro jednoduchou manipulaci s daty v NoSQL databázi, jako například vytvoření a smazání senzoru v NoSQL databázi, přidání nové naměřené hodnoty a funkce nad výběrem dat z databáze, získání průměrné hodnoty apod.

Každá instance programu je schopna běžet samostatně a komunikuje s jednou již existující NoSQL databází, která běží na hlavním serveru. Na vstupu poté program komunikuje se systémem IoT, např. BeeeOn. IoT systém bude program využívat pro ukládání dat, které získá od senzorů a následné dotazování nad nimi. V případě potřeby (např. synchronizace) je možná koordinace mezi jednotlivými programy pomocí síťové komunikace. Samotný program poté NoSQL databázi využívá pouze pro potřeby uložení dat na disk a získání uložených dat zpět. Použitím již existujícího databázového serveru je poté zaručena také spolehlivost uložení dat a jejich dostupnost. Vyvíjený program tak vytváří rozšíření pro NoSQL databázi, kde budou v paměti uloženy data, s kterými se aktuálně pracuje a stará se o efektivní uložení, společně s dotazy nad daty.



Obr. 14: Nasazení vyvíjené aplikace („Program“) v reálném systému.

Vyvíjená aplikace [Obr. 15] se skládá z adaptéru, implementací dotazů, modelů dat, mezipaměti, konektoru na databázi. Adaptér poskytuje funkce pro připojenou aplikaci. V závislosti na typu dotazu, bude vybrána konkrétní implementace dotazu, který bude aplikace vykonávat. K databázi bude mít přístup konektor. Ten bude moct načítat data z databáze a vkládat data nové. Pro uložení dat do paměti bude sloužit modul mezipaměti, zde budou data uloženy v připravených modelech tak, aby se zjednodušila práce s daty. Protože se s daty bude pracovat v operační paměti, je potřeba nějakým způsobem řídit její vytížení. To má na starosti démon, který po určité době vyčistí data z paměti. Dále se také stará o mazání starých dat z databáze. V samostatném vláknu je k dispozici také komunikační protokol pro síťovou komunikaci mezi systémy.



Obr. 15: Schéma částí vyvíjeného programu.

6.3 Návrh uložení

6.3.1 MongoDB

K uložení dat jsem se rozhodl využít již existující NoSQL databázi dokumentového typu. Zvolenou databází je MongoDB. Důležitou vlastností vybrané databáze musí být její horizontální škálovatelnost. Nad touto databází poté bude implementována řídící vrstva, která se bude starat o přípravu dat pro jejich uložení a dotazování nad nimi.

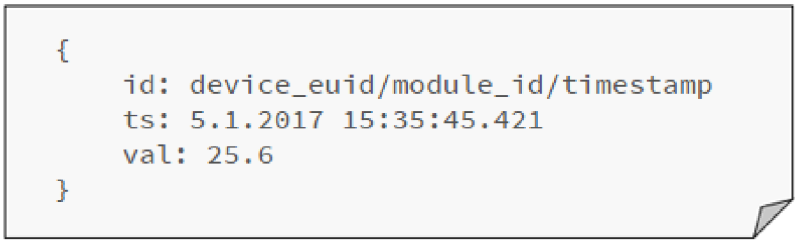
MongoDB je open-source NoSQL databáze, která je vyvíjena od roku 2007. Volně dostupná je poté od roku 2009, kdy uvolněna pod licencí GNU AGPL což povoluje její využití a modifikaci.

Jedná se o dokumentovou databázi, která pracuje s dokumenty typu JSON, které na disk ukládá v jejich binární zakódované podobě BSON. Tyto strukturované dokumenty typu JSON se poté ukládají do databáze pod vlastním atributem *_id*, který slouží jako hashovací klíč pro daný dokument. Atribut *_id* tedy musí být unikátní a povinný. Jeho generování má na starost databáze, ale je možné jej nastavit i ručně. MongoDB umožňuje s dokumenty v databázi pracovat a vytvářet dotazy i nad atributy JSON dokumentu. Dokumenty je poté možné seskupovat do tzv. kolekcí, v kontextu relačních databází si to lze představit jako by se jednalo o tabulku a dokument by byl záznamem.

Nespornou výhodou použití databáze MongoDB je její rozšířenost a kvalitní dostupná dokumentace společně s velkou podporou komunity vývojářů.

6.3.2 Struktura dat

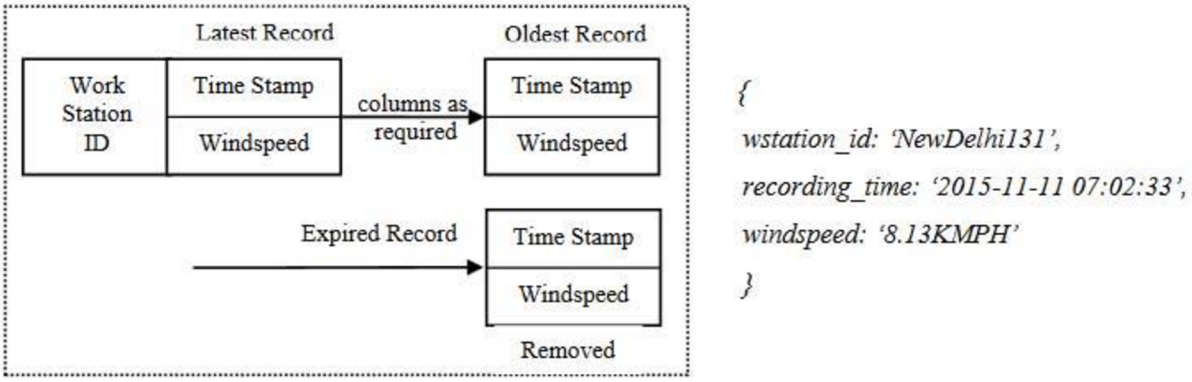
V souvislosti s využitím dokumentového typu databáze je využití strukturovaného jazyka pro uložení dokumentu. Jako nejvhodnější volba se jeví využití jazyka JSON, ve kterém je možné strukturovaně uložit data. Současný relační návrh databáze je možné převést jednoduchým způsobem na dokumentový typ formátu JSON následovně, ilustrace [Obr. 16]. Atributy tabulky *log* se uloží do strukturovaného dokumentu ve formátu JSON. Klíčem pro získání dat by byl složený klíč vytvořený z ID zařízení, modulu a časové známky. Tento dokument by se poté vložil do NoSQL databáze a pomocí složeného klíče by bylo možné k němu přistoupit. Aby bylo možné provádět dotazy nad časovým úsekem, bylo by potřeba zařadit do dokumentu atribut typu *next_id*, který by byl dostupný za účelem serializace dat. Při načtení konkrétního měření by bylo ihned zjištěno *id* dokumentu dalšího výsledku, který by se hledal.



Obr. 16: Naivní uložení dat v dokumentu.

Tento přístup má řadu nevýhod. Data jednotlivých senzorů, nemusí být uloženy za sebou nebo poblíž sebe v jejich chronologickém pořadí. Dotazy pro získání úseku dat mezi dvěma časy by bylo tudíž pomalé. Logy by bylo nutné číst po jednom záznamu. To by vedlo k velkému počtu diskových operací čtení. Jedná se o zbytečné zatížení aplikace a disku. Stejně tak je potřeba vkládat tyto data do databáze jednotlivě. Databáze by byla také přetížena indexováním takhle velkého počtu klíčů.

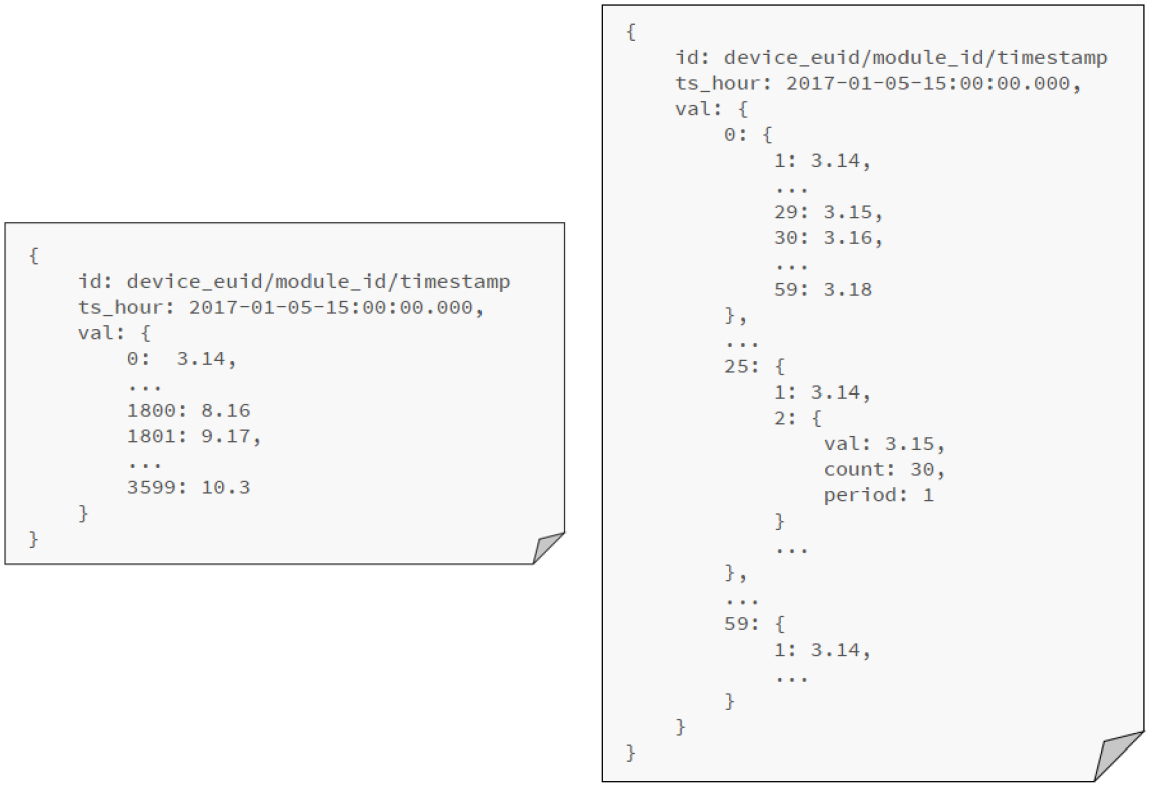
Vylepšené řešení problému je uvedeno v dokumentu [18]. Zde je pro každou stanici vytvořen jeden strukturovaný dokument, který obsahuje kolekci naměřených dat. Prvek kolekce je vždy označena svojí časovou známkou a naměřenou hodnotou. Tohle již řeší problém čtení dat, kdy by bylo možné získat z databáze dokument obsahující více hodnot a poté bez dalších čtení z disku s daty pracovat v paměti. Přidání nové položky provádí tak, že se z vkládaného dokumentu získá *ID* zařízení, v databázi se nalezne potřebný záznam a dokument se přidá na začátek získaného záznamu. Celý dokument se poté uloží do zpět do databáze. Vzniká zde ale nevýhoda při velkém množství zaznamenaných dat. Hledání konkrétní hodnoty by muselo být provedeno sériově. Databáze také musí při každém novém vloženém záznamu ukládat velký dokument. Samotné uložení je ale taktéž neefektivní, je zde u každého záznamu uvedeno časové razítko, které zabírá zbytečné místo.



Obr. 17: Vylepšená možnost uložení senzorových dat. Vpravo je dokument určený pro vkládání. [18]

Výše popsaný princip lze vylepšit tak, že se bude v databázi uchovávat pod jedním klíčem kolekce naměřených hodnot, ale bude určena délka trvání tohoto úseku. Inspirací zde může být právě implementace v databázi OpenTSDB, kde jsou data shlukovány v úseku jedné hodiny. Tím se docílí zmenšení velikosti dokumentů, které se poté rychleji načítají z databáze a ukládají se. Vyhledání uvnitř této struktury je také rychlejší. Ukázka jednoho dokumentu je na [Obr. 18] vlevo. Dokument zůstává v paměti, dokud není doplněn do konce hodinového intervalu. Do dokumentu se vkládají data pouze jednoho senzoru. Časová známka je v tomto případě ořezána na celé hodiny, což pomůže při indexaci, kdy při dotazu nad získání dat v určitém času dojde také k ořezání tohoto data na celou hodinu a je možné tak rychle zjistit v jakém shluku dat hledat. Data jsou označena atributem například minut nebo sekund, relativně. Jedná se o rozdíl jejich časové značky od značky skupiny dat. Tím je možné ušetřit část dat a zlepšit indexaci uvnitř dokumentu. V časovém intervalu však může být spoustu hodnot měření. V případě měření v intervalu jedné sekundy, by tak bylo za sebou 3600 hodnot, ve kterých by nebylo vyhledání patřičné hodnoty tak efektivní.

Proto je možné stávající návrh upravit do podoby jež je znázorněna na [Obr. 18] vpravo. Dokument bude obsahovat ukládání dat do skupin po minutách a uvnitř této skupiny bude výsledná hodnota označena atributem sekund. Tím lze docílit, že případná data, která by měla 3600 záznamů za sebou, budou rozděleny do 60 skupin po maximálně 60 záznamech. Tím lze docílit rychlejšího průchodu touto strukturou. Tato struktura je také vhodná pro dotazování, protože spolu související data jsou v jednom celku uspořádány za sebou.



Obr. 18: Dokumenty s kolekcí naměřených dat s relativním časem.

6.3.3 Optimalizace návrhu

Komprimace

Senzorová data mohou obsahovat z velké části při nízkých rozestupech měření podobná data. Tlak, nebo teplota se nemění skokově, většinou jde o pozvolný nástup změn. Zde se otvírá prostor pro možnosti využití komprimace těchto dat, při případné malé ztráty přesnosti. Do současného návrhu [Obr. 18] je možnost zahrnout jednoduchou komprimaci založenou na principu *Run-length encoding*. Naměřená hodnota desetinného typu se ořeže na daný počet míst, který zajišťuje postačující přesnost výsledků. Pokud nově příchozí hodnota bude stejná jako předcházející, uloží se místo hodnoty objekt s hodnou měření, časovým rozdílem mezi daty a čítačem počtu výskytů stejné hodnoty. Z reálných měření v dostupné databázi byla tato vlastnost u některých modulů vidět, kdy ke změně dat docházelo zřídka.

Agregace dat

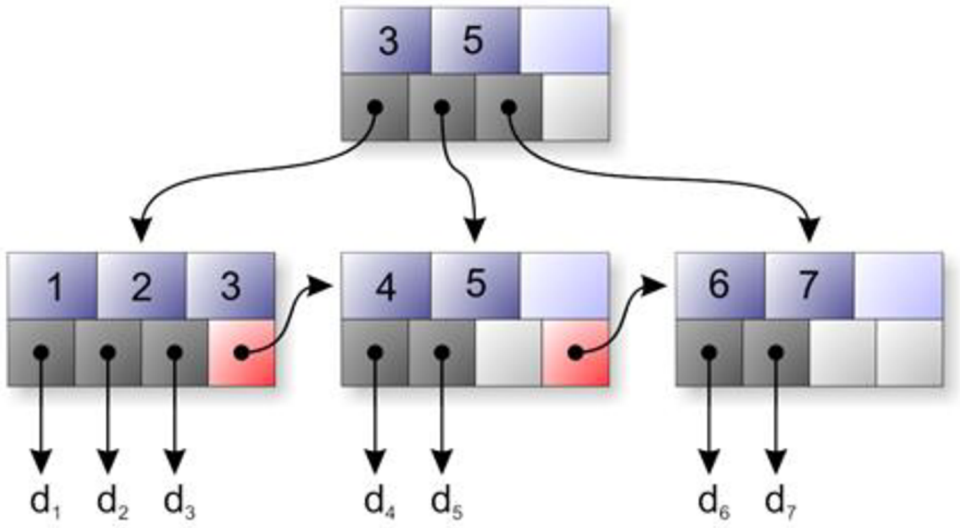
Při dotazování výběru dat v určitém rozmezí časových hodnot je možné přidat k dotazu podmínku k agregaci dat, například *„vrať mi data od X do Y s krokem jedné minuty/hodiny“*. Takovéto dotazy nejsou v systému BeeeOn používané, ale navržená datová struktura v [Obr. 18] umožňuje usnadnit výpočet těchto agregovaných dat. Při ukládání by bylo možné dopočítat agregaci dat pro jednotlivé minuty a také celou hodinu. Jednalo by se tedy o výpočet průměru nad

daty. Tyto hodnoty by se poté uložily dále do struktury tohoto dokumentu a při případném dotazu s touto podmínkou by se tyto před počítané hodnoty využily.

6.3.4 Meta-data

Aby bylo možné mít v ukládaných datech v paměti přehled a provádět mezi nimi vyhledávání, je potřeba si vést informace, které budou naše data popisovat. Uložení těchto meta-dat, může být založené na podobném principu jako ukládání samotných dat na disku. Pro popis struktury se využije opět dokument ve formátu JSON.

Vlastní dokument může mít vytvořeno každé zařízení, které se nachází v systému. Jeho identifikátor bude odpovídat *device_euid*. Pod tímto identifikátorem bude uložen dokument v NoSQL databázi. Struktura dokumentu bude obsahovat kolekci objektů dělených podle *module_id*. V konkrétní větvi je bude rozdělení do objektů na základě časové známky, která bude reprezentovat hodnotu dne. V této struktuře již budou uloženy informace o klíčích k vyhledání patřičného záznamu hodiny. Na základě tohoto dokumentu bude v paměti vytvořena struktura reprezentující tuto hierarchii ve formě B+ stromu.



Obr. 19: Znázornění struktury B+ stromu. [19]

Struktura B+ stromu je znázorněna na [Obr. 19]. V implementaci by se jednalo v první vrstvě o indexaci senzoru a modulu. V další vrstvě by byla k dispozici indexace podle dne a poslední spodní vrstva poté indexaci na základě hodiny. Poslední uzel poté bude obsahovat informaci potřebnou k načtení dokumentu z databáze, případně pokud se již s uzlem pracovalo, tak bude obsahovat před načtenou verzi požadovaného dokumentu. Díky provázání spodní úrovně stromu je poté možné sekvenčně číst po sobě jdoucí dokumenty. Tato hierarchie umožní mít přehled o uložených datech a zároveň v nich rychle vyhledávat. Požadovanou operaci pro odstranění zastaralých dat je možné provést jednoduše odstraněním záznamů v databázi, které bude tato struktura obsahovat pod patřičným dnem.

Kapitola 7

Implementace

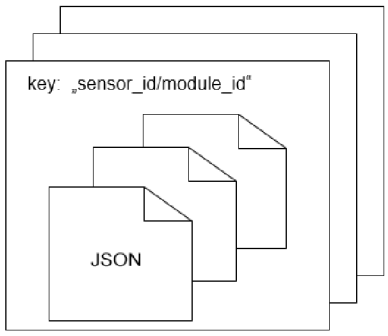
K implementaci systému jsem se rozhodl využít jazyk Java. Výhodou jsou jeho objektové rysy a široké možnosti rozšíření, ať se již jedná o práci s JSON dokumenty, databázovým serverem, nebo také vytvoření síťové komunikace.

7.1 Uložení dat v databázi

Implementace programu je provedena s využitím NoSQL databáze MongoDB. Tento databázový program poskytuje knihovnu pro práci v jazyce Java, což usnadnilo vývoj ohledně komunikace s databázovým serverem.

7.1.1 Využití kolekcí v MongoDB

Díky dostupnosti kolekcí v MongoDB jsem se rozhodl vytvořit si pro snazší práci s dokumenty a senzory hierarchii, kdy každý senzor má k dispozici vytvořenou vlastní kolekci, do které se poté přidávají jednotlivé dokumenty obsahující měření pro konkrétní senzor. Kolekce je identifikovatelná pomocí zvoleného řetězcového klíče, který je možné si ručně zvolit. V mém případě jsem jako klíč zvolil řetězec identifikátoru senzoru a modulu „*id_senzoru/id_modulu*“. Tímto navržením jednotlivé kolekce pro každý senzor jsem získal možnost získat kdykoliv seznam všech senzorů v databázi (MongoDB nabízí možnost získat list dostupných kolekcí v databázi), aniž bych si vedl speciální dokument s výčtem jednotlivých senzorů v databázi a ten neustále s databází synchronizoval během jeho změny. Dojde tím také k nepatrné úspoře prostoru na disku. Nejdůležitější je však skutečnost, že společné JSON dokumenty konkrétního senzoru jsou ve své blízkosti, případně možnost získat uložené dokumenty konkrétního senzoru bez znalosti informací o času měření.



Obr. 20: Kolekce s patřící konkrétnímu senzoru a uložené JSON dokumenty uvnitř.

7.1.2 Informace o senzoru

Aktuální databázový server BeeeOn obsahuje spoustu dodatečných informací o senzoru. Většina z těchto informací pro potřeby databáze pro ukládání měření nebude potřeba, ale vybrané atributy jsou důležité a při vytváření senzoru jsou vyžadovány. Tyto informace jsou uloženy uvnitř kolekce senzoru ve vlastním JSON dokumentu. Pro tento dokument je zvolen statický identifikátor „*_id = sensor_info*“, který má každý senzor stejný. Tím se zjednoduší nalezení těchto informací i v případě, že o senzoru nejsou dostupné žádné informace.

Dalšími atributy jsou *server* a *port*, které identifikují vlastníka tohoto senzoru. Tím je myšleno, další běžící instance této aplikace. V případě potřeby získat nejaktuálnější verzi dokumentu, která ještě není uložena v databázi, je možné tento server upozornit pomocí implementované operace pro síťovou komunikaci, která zajišťuje synchronizaci mezi běžícími instancemi této aplikace. Původním záměrem bylo za tímto účelem využít synchronizačního serveru ZooKeeper od Apache, který sdílí informace mezi servery, ale nakonec bylo přistoupeno k vlastní implementaci.

Informace *lastSave* značí, jakou verzi aktuálního dokumentu má server pro senzor k dispozici. *AutoSave* je informace po kolika nových příchozích měření se má hodinový dokument s měřeními uložit do databáze. Atribut *refresh* poté určuje interval, ve kterém senzor pořizuje měření.

Nejdůležitější informací obsaženou v *sensor_info* je atribut *precision*. Ten určuje přesnost měřené hodnoty *double* na počet desetinných míst. Na základě tohoto čísla je zaokrouhlena každá příchozí hodnota měření. Každý senzor v databázi má tyto hodnoty vlastní, nejedná se o společné nastavení.

```
1     {
2           _id = sensor_info,
3           server = localhost,
4           port = 55123,
5           lastSave = 1492970490,
6           autoSave = 60,
7           precision = 3,
8           refresh = 15
9     }
```

Alg. 1: Znázornění dokumentu s informacemi o senzoru.

7.1.3 Formát dokumentu měření

Uložení dokumentů obsahující naměřené hodnoty vychází z původního navrženého formátu. Jeden dokument tak symbolizuje jednu celou hodinu a obsahuje tak více naměřených hodnot v tomto časovém úseku.

```
1     {
2           _id = 123/321/1492970400,
3           s = 1492970490,
4           v = {
5                 0 = 0.123,
6                 10 = [0.125, 10, 3],
7                 40 = [0.123, 10, 2],
8                 85 = 0.123,
9                 90 = 0.4
10          }
11    }
```

Alg. 2: Znázornění hodinového dokumentu s měřeními.

Jako „*_id*“ dokumentu je zvolen klíč, který se skládá z hodnot id senzoru, id modulu a časové značky, tzv. „*_id = sensor_id/module_id/timestamp*“. Časová značka obsažena v klíči je oseknuta na celou hodinu, tak aby neobsahovala minuty ani sekundy. Dokument poté obsahuje naměřená data od tohoto data až po následující hodinu. Ořezáním značky zajistím, že v případě vyhledávání konkrétního měření, které má daný konkrétní přesný čas, naleznu správný dokument, ve kterém se bude požadovaná hodnota nacházet.

Atribut „*s*“ poté symbolizuje čas posledního přidaného měření do dokumentu a následné uložení do databáze. Tento atribut je potřebný kvůli zjištění verze dokumentu, která je aktuálně načtena v paměti. V případě, že příchozí dotaz na data, které mají vyšší čas než čas uložení dokumentu, bude z databáze načten novější dokument, případně pokud není v databázi není novější dokument, kontaktuje se správce daného senzoru (instance programu na jiném serveru), aby provedl uložení aktuálního dokumentu z paměti do databáze.

Následující atribut znázorňuje kolekci senzorem naměřených hodnot. Jednotlivé záznamy měření jsou zde uloženy společně s časovou značkou jejich pořízení, která je relativní. Jedná se o hodnotu sekund v této hodině, která je získaná odečtením celé hodiny identifikující tento dokument od původní časové značky měření. Pomocí relativního značkování je možné dosáhnout úspory uložení dat, protože nemusí být u každé hodnoty uložena celá časová značka, ale pouze její rozdíl. Ke každé relativní časové značce je přiřazena naměřená hodnota. Pokud v nějaký čas nebylo provedeno žádné měření, není časová známka vůbec uvedena a u dat tak není vyžadováno, aby přicházely v pravidelných časových intervalech, který je zadaný v informaci o senzoru.

Naměřená hodnota uvedená v položce může být buď konkrétní číslo nebo pole tří položek. V případě vloženého typu pole se jedná o uložení dat komprimovanou formou. Pro kompresi je zde využito *Run-length encoding* (zkráceně RLE). Pro kompresi se využívá ořezání naměřené hodnoty na uvedenou přesnost a poté test, zda po sobě jdoucí hodnoty mají stejnou hodnotu. Pokud si hodnoty odpovídají, je vytvořen objekt RLE, který poté danou hodnotu uvede pouze jednou a zaznamená si časový rozestup mezi měřeními a počet hodnot, které jsou v RLE obsaženy. Pokud se stane, že nově příchozí hodnota má stejnou hodnotu, ale jiný časový rozestup, není do komprese zařazena a RLE je ukončeno. Tato komprimace probíhá v aplikaci dynamicky v čase příchodu nové hodnoty měření. Není prováděno až těsně před uložením dokumentu.

7.1.4 Konektor k databázi

Pro přístup k datům z databáze slouží v implementované aplikaci třída *ConnectionDb*. Tato třída poskytuje metody pro načítání, ukládání, mazání a aktualizaci dat v databázi. Třída je implementována podle návrhového vzoru singleton, aby byla dostupná pro všechny ostatní třídy. Tato třída je také zodpovědná za připojení k databázi MongoDB.

```
1     public void updateCollectionItemDb(
2           MongoCollection coll,
3           Document item) {
4                 BasicDBObject prevItem = new BasicDBObject(
5                       Constants.idDbString,
6                       item.get(Constants.idDbString));
7                 coll.replaceOne(prevItem, item);
8     }
```

Alg. 3: Ukázka funkce pro update dokumentu v databázi s využitím funkce z API MongoDB.

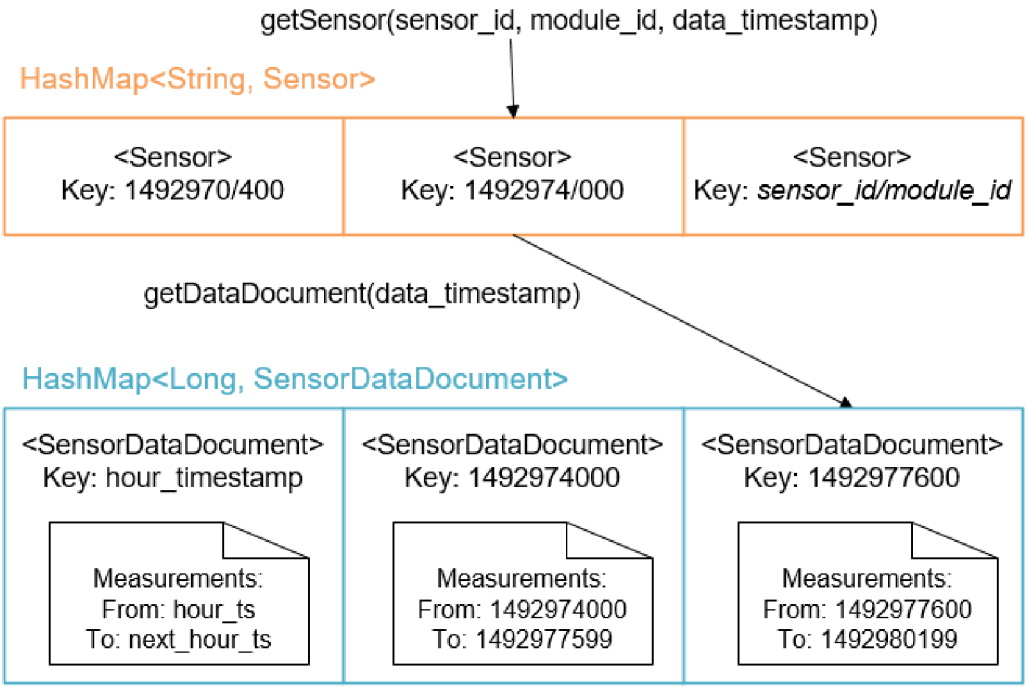
7.2 Uložení dat v operační paměti

Implementovaný databázový program z velké části pracuje s daty v operační paměti, databáze zde slouží pro persistentní uchování dat. Proto bylo potřeba vytvořit strukturu uložení těchto dat v operační paměti tak, aby byla práce s nimi efektivní a pohodlná.

Původní návrh uložení dat v operační paměti, který jsem měl v plánu implementovat, využíval pro organizaci dat strukturu B+ stromu. Stromová struktura by byla složena z hashovací mapy obsahující objekty reprezentující senzory, ke kterým by se přistupovalo pomocí klíče složeného z identifikátoru senzoru. V senzoru by se dálo přistupovalo již k sekvenčnímu seznamu indexů jednotlivých modulů senzoru. Z této vrstvy by se získal patřičný modul. Následující vrstva B+ stromu by dále vyhledávala pomocí indexu dnů a naposled hodin v konkrétním dnu. Konec B+ stromu by poté obsahoval objekt reprezentující JSON dokument reprezentující konkrétní hodinu s provedenými měřeními.

Implementovaná byla nakonec upravená varianta, která není založena na struktuře B+ stromu, ale pouze na dvou hashovacích tabulkách. Důvodem byl značný výkonnostní benefit pro vyhledávání cílových dokumentů v řešení využívající formu hashovací tabulky. U této implementace jsem se rozhodl sjednotit identifikátory senzoru a modulu, výsledný řetězec poté slouží jako klíč pro hashovací tabulku. Hashovací tabulka je implementována pomocí v Javě dostupného. Přístupem pomocí klíče získám objekt *Senzor*, který reprezentuje konkrétní modul senzoru. Tento objekt obsahuje rovněž hashovací tabulku, v tomto případě se jedná o tabulku obsahující objekty typu *SensorDataDocument* reprezentující abstrakci nad hodinovým dokumentem s uloženými měřeními. Vyhledání patřičného *SensorDataDocument* je provedeno pomocí hodinové časové značky dokumentu. Původní návrh založený na B+ zajišťoval možnost sekvenčního průchodu mezi dostupnými *SensorDataDocument*, které by na sebe časově navazovaly. Návrhem dokumentu obsahující celou hodinu provedených měření se však podobného sekvenčního průchodu dá docílit i u hashovací tabulky. Pomocí jednoduchého přičtení hodiny, nebo odečtení hodiny od časového razítka reprezentující hashovací klíč je možné přistoupit k následujícímu nebo předcházejícímu dokumentu. Tato implementovaná struktura

uložení dat v operační paměti má obdobné rysy jako NoSQL databáze MongoDB. K získání dat je využíváno hashovacích tabulek, což je princip dokumentových NoSQL databázových systémů.



Obr. 21: Schéma uchování dat v operační paměti a získání konkrétního senzoru s dokumentem obsahující měření.

7.2.1 Reprezentace datového dokumentu

SensorDataDocument třída, která je vyhledatelná podle jejího klíče složeného z hodinové časové značky. Uvnitř obsahuje reprezentaci JSON dokumentu s uloženými měřeními, společně s metodami pro práci s uloženým dokumentem. JSON dokument je reprezentován typem *Document*, který ve svém API využívá MongoDB. JSON dokument je indexován pomocí klíčů (názvy atributů v JSON souboru). Jedná se tak o princip hashování a přístup k datům v tomto datovém typu neprobíhá sekvenčním průchodem strukturou. Bez znalosti konkrétní hodnoty časové známky není možné získat požadovaná data. Za tímto účelem si vytvářím pole indexů obsažených v JSON dokumentu. V případě dotazu *select* obsahující konkrétní časovou známku měření, které uživatel chce získat se první vykoná v dokumentu vyhledání pomocí klíče (časová známka). Pokud však v daný okamžik nebylo provedeno žádné měření a v dokumentu tak chybí atribut s tímto klíčem a mě zajímají například hodnoty v daném časovém rozpětí je potřeba využít pro vyhledávání vytvořené pole indexů. Pole indexů je seřazené, a tedy ideální pro vyhledání konkrétní hodnoty nebo pokud neexistuje, tak získání předcházející a následující časové značky existujících měření. Vyhledávání není prováděno sekvenčně, indexů může být potenciálně až 3600. Za účelem rychlejšího získání dat bylo potřeba implementovat nad polem binární vyhledávací algoritmus, který poskytuje nejhůře logaritmickou časovou složitost.

Chybějící časový klíč měření v dokumentu může chybět ve dvou situacích:

- Senzor v daný čas neprovedl žádné měření, neexistuje potenciální uložený záznam. Potencionální problém nastane v situaci dotazu s časovým intervalem výběru měření a neexistuje měření odpovídající začátku intervalu. Pomocí binárního vyhledaná je nalezena časová značka následující měření.
- Měření se nachází uvnitř komprimovaných dat v dokumentu, je tedy vedeno pod nadřazenou časovou značkou. V tomto případě je binárním vyhledáním získána přecházející hodnota a proběhne kontrola na existenci hodnoty v RLE objektu.

Původní záměr práce s hodinovým datovým dokumentem bylo provádění zápisu do databáze celého dokumentu v jeden okamžik (zaplnění dokumentu a přechod na další hodinu) a případné načtení z databáze dokumentu opět jako celku do paměti. Za tímto účelem je u senzoru vedený atribut intervalu zálohy do databáze. Tento způsob by byl velmi náchylný na výpadky, při kterých by docházelo ke ztrátě neuložených dat. Nastavený interval pro zálohu, měl tento problém redukovat. Data byla vždy po určitém přidání nových dat uložena do databáze. Nevýhodou by však bylo také neustálé ukládání objemného dokumentu do databáze v případě nastavené nízké hodnoty. Dalším problémem je případ, kdy aplikace ještě neprovedla uložení dat do databáze, ale jiná aplikace již data požaduje a je potřeba řešit synchronizaci mezi aplikacemi.

Tyto problémy řeší implementace okamžitého zápisu nové naměřené hodnoty do databáze. MongoDB umožňuje provádět změny v databázi i uvnitř dokumentu, a ne pouze na úrovni nahrání celého dokumentu. Tato změna se provádí vytvořením JSON dokumentu s názvem „*$set*“ a následným zadáním hodnoty, která se má nastavit společně s uvedením cesty k měněnému atributu v původním dokumentu. Dochází sice k neustálým zápisům, ale nejde zde o ukládání celého dokumentu, pouze malé změny.

```
1     db.coll.update({"_id":"123/321/123457851254"},
2                    { "$set":{"v.30":"new_value"}})
```

Alg. 4: Ukázka aktualizace atributu uvnitř dokumentu v databázi, který je definován „_id“.

U senzoru ukládaný atribut udávající interval zálohy dokumentu do databáze jsem se rozhodl ponechat. Je zde možnost současný stav implementace, kdy dochází k okamžitému uložení jedné hodnoty, rozšířit o možnost zvolit automatickou. Příchozí nové měření by byly skládány a do databáze by byla v určitý čas uložena záloha, která by obsahovala pouze rozdílová data mezi novějším dokumentem a starším. Směrem do databáze by nedocházelo k přenosu celého dokumentu, jako prvotním řešení.

7.2.2 Mezipaměť

V implementované struktuře uložení dat v paměti se v případě vykonávání uživatelského dotazu pomocí dvou přístupů do hashovacích tabulek získá cílový dokument, který bude obsahovat hledaná data, případně data k editaci. V mezipaměti se aplikace snaží držet co nejvíce dostupných dokumentů, aby se s nimi pracovalo primárně v operační paměti a z databáze se data načítaly do paměti co nejméně. Může však nastat situace, že není v paměti načtený patřičný objekt reprezentující hodinový dokument, který obsahuje hledaná data. Při dotazu se automaticky

v případě nenalezení požadovaného objektu v hashovací tabulce tento objekt vytvoří a synchronizuje data z databáze. Následně objekt zůstává v paměti dále přístupný v případě dalších dotazů na data v podobném časovém úseku.

Tímto způsobem by docházelo k neustálému nárůstu před načtených dokumentů v paměti. Odstranění starých dokumentů z operační paměti má na starosti zvláštní běžící vlákno. U vlákna je možné definovat při jeho spuštění v jakém časovém intervalu se má čištění provádět. Hodnota se při inicializaci udává v sekundách. Na základě zadaného údaje se provede naplánování spuštění tohoto vlákna, které poté odstraní z paměti před načtené dokumenty kromě aktuálního.

```
1     Daemon.init(daysForDbCleanup, secsToMemoryCleanup); // spuštění
2
3     public static void init(int dbClearTime, int memoryClearTime) {
4           Timer timer = new Timer();
5           DaemonJob daemonJob = new DaemonJob(dbClearTime,
6                                       memoryClearTime, timer);
7           timer.scheduleAtFixedRate(daemonJob, 0,
8                                       memoryClearTime);
9     }
```

Alg. 5: Vytvoření a inicializace vlákna pro čištění paměti a databáze, provádí uživatel.

Součástí vlákna pro správu mezipaměti je také metoda pro čištění samotné NoSQL databáze. Při inicializaci vlákna se kromě časového intervalu čištění mezipaměti udává také maximální stáří dat v databázi. Tato hodnota se určuje ve dnech. V případě provádění čištění se po denním rozdílu provede i smazání dat, které jsou starší než uvedený počet dnů. Mazání probíhá přímo v databázi pomocí dotazu s určujícím časem stáří dat [Alg. 6]. Dotaz je ve formátu JSON, kde je obsaženo, jaký atribut v dokumentu nás bude zajímat a v objektu označeném tímto atributem bude uveden atribut menší než „*$lt*“ (*less than*) a jeho hodnota. Poté se v kolekci pomocí metody *deleteMany* odstraní všechny dokumenty, které splňují tento předpis.

```
1     BasicDBObject query = new BasicDBObject();
2     query.put(itemKey, new BasicDBObject("$lt", ts));
3     dbCollection.deleteMany(query);
```

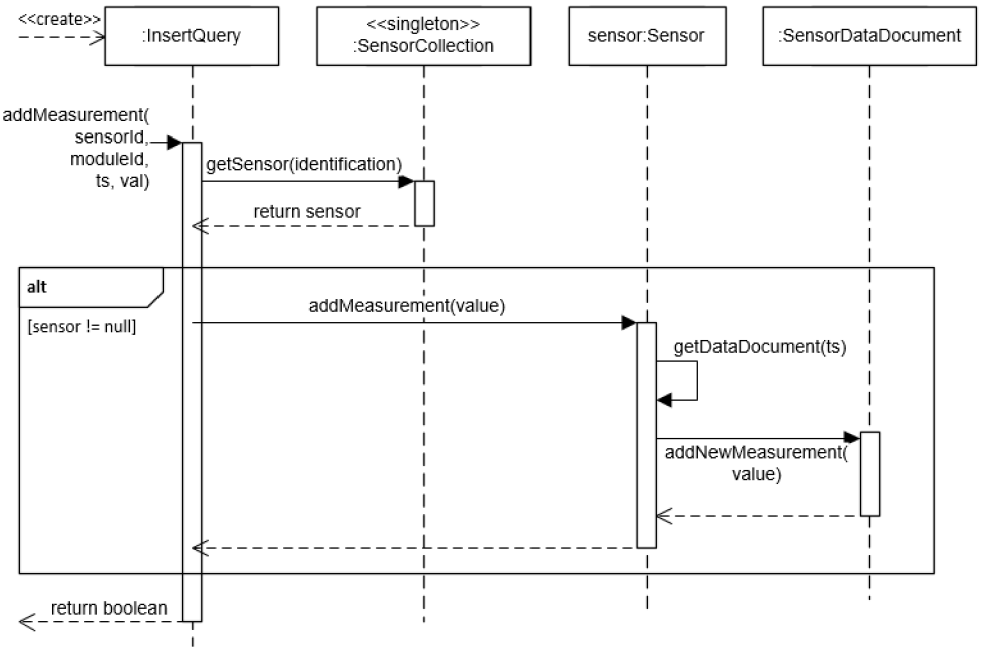
Alg. 6: Dotaz do MongoDB, který smaže všechny dokumenty obsahující položku s názvem „itemKey“, jejichž hodnota je menší než zadaná „ts“.

7.3 Dotazy nad daty

Uživatel databáze bude primárně pracovat s třídami uvnitř implementovaného balíčku „*query*“, který slouží jako adaptér pro připojenou aplikaci. Dostupné třídy (*InsertQuery, SelectQuery, RemoveQuery*) nabízí patřičné metody pro provedení dotazů nad daty z databáze. V následující části popsaná návrh a implementace dotazů dat senzorovými daty.

7.3.1 Dotaz pro vložení

Třída *InsertQuery* nabízí uživateli metody, které mají na starost vkládání dat do databáze. Práce s třídou je znázorněna na sekvenčním diagramu [Obr. 22]. Uživatel pro začátek práce musí vytvořit novou instanci třídy a nad získaným objektem poté již může invokovat implementované metody, které se implementují požadované operace.



Obr. 22: Sekvenční diagram znázorňující prováděné akce při přidávání nové hodnoty měření do databáze a invokaci této akce.

```
1     Boolean addMeasurement(int sensorId, int moduleId,
2           long ts, double val);
3     Boolean addSensor(int sensorId, int moduleId,
4           int dataRefresh, int dataAutoSave, int dataPrecision,
5           String serverIp, Integer serverPort);
```

Alg. 7: Dostupné metody pro vložení do databáze.

Primárním je metoda *addSensor()*, která umožní přidat do systému nový senzor. Vyžadovány jsou parametry s identifikačním číslem senzoru a modulu, rozmezí mezi měřeními, interval ukládání nových dat do databáze, přesnost na desetinná místa, na kterou se má naměřená hodnota zaokrouhlit a kontaktní informace na server, který má senzor pod správou. Přidání senzoru se provede pouze v případě, že senzor ještě neexistuje v paměti ani databázi. Při povedeném přidání senzoru se vytvoří jeho instance v operační paměti, následuje přidání do hashovací tabulky senzorů a uložení do databáze s nastavenými daty.

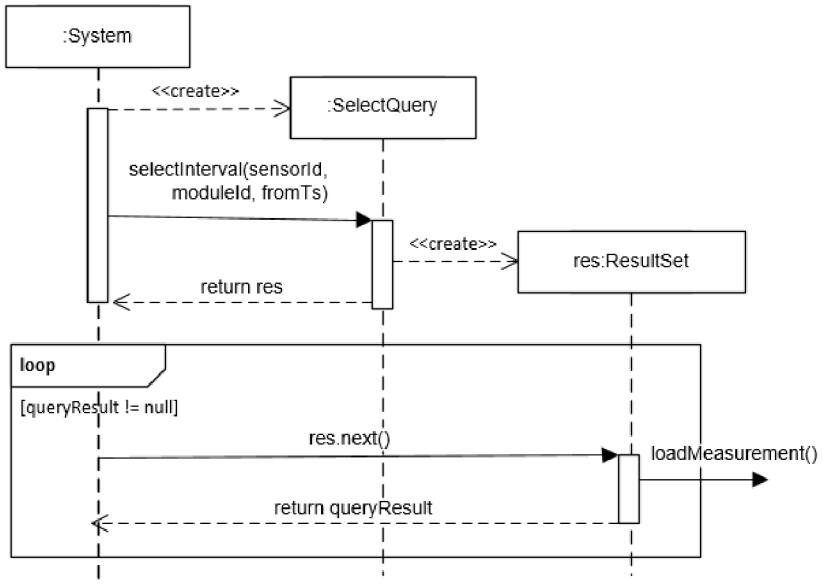
Metoda *addMeasurement()* implementuje přidání dat do databáze a do dokumentu v operační paměti. Diagram na [Obr. 22] znázorňuje zjednodušený průběh akce. Na základě zadaného senzoru se získá jeho instance v paměti a pomocí času měření se nalezne v hashovací tabulce patřičný hodinový dokument. Uvnitř dokumentu poté proběhne přidání samostatné

hodnoty do na konec JSON dokumentu. Vložení probíhá včetně případné komprimace dat, kdy se testuje, zda se příchozí hodnota rovná předchozí. Následně se případně provede vytvoření RLE objektu, který nahradí poslední hodnotu v JSON dokumentu. Nakonec se provede uložení samostatné nové hodnoty i do databáze pomocí databázového konektoru. Pomocí této metody je také možné provést aktualizaci hodnoty měření v konkrétní čas, není však již implementováno řešení situace, kdy se hodnota nachází uvnitř objektu RLE.

7.3.2 Dotaz pro výběr

SelectQuery je třída nabízející metody pro výběr dat z databáze. Zde je implementováno celkem 7 metod z nichž jsou unikátní 4 metody pro různé druhy výběrů.

Diagramu [Obr. 23] ukazuje fungování dotazování z vnějšího pohledu. Každý dotaz nad výběrem má vlastní instanci třídy *SelectQuery*, tím se liší proti předchozímu typu dotazu (*insert*). Objekt *selectQuery* uloží informace o typu dotazu a parametrech dotazu a vytvoří objekt typu *ResultSet*, kterému předá vlastní instanci. *ResultSet* je poté předán nadřazené aplikace, která nad tímto objektem může invokovat metody pro získání hodnoty, které dotaz vrací. *ResultSet* tedy vrací jednotlivé výsledky odpovídající dotazu. V případě, že již není žádná hodnota nalezena, dojde k vrácení hodnoty *null*. Jedná se o obdobný princip nabízený JDBC driverem[1]. *ResultSet* obsahuje seznam do kterého jsou postupně načítány jednotlivé položky odpovídající dotazu, které se vrací uživateli voláním metody *next()*. Pomocí dalších metod *previous()* a *first()* je možné získat již obdržené hodnoty, které jsou uloženy v seznamu. Celý výsledek dotazu se nezískává najednou, ale je implementován tzv. líným přístupem, kdy se následující výsledek získává, až když uživatel invokuje metodu *next()*. Dotazy pracují pouze s daty uloženými v operační paměti. V případě, že není nalezen hledaný hodinový dokument, provede se jeho načtení z databáze do paměti.



Obr. 23: Sekvenční diagram znázorňující dotaz vnějšího systému na zobrazení uložených dat.

[1] Java Database Connectivity (známé spíše jako **JDBC**) je API pro programátory v programovacím jazyku Java, které definuje jednotné rozhraní pro přístup k relačním databázím. **JDBC** je součástí Javy SE („Standard Edition“) od JDK 1.1., definice převzata ze zdroje [20].

```
1  ResultSet selectMeasurement(int sensorId, int moduleId, long ts);
2  ResultSet selectInterval(int sensorId, int moduleId,
3     long fromTs, long toTs);
4  ResultSet selectIntervalAverage(int sensorId, int moduleId, long
5     fromTs, long toTs, int avgIntervalType, long avgIntervalTs);
6  ResultSet selectMax(int sensorId, int moduleId,
7     long fromTs, long toTs);
8  ResultSet selectMin(int sensorId, int moduleId,
9     long fromTs, long toTs);
```

Alg. 8: Implementované metody pro získání dat.

Základním dotazem je *selectMeasurement*, který vyhledá a vrátí jednu hodnotu naměřenou senzorem v určitý čas. Pokud hodnota s daným časovým razítkem neexistuje, není žádná hodnota vrácena.

SelectInterval umožňuje získat z množinu více měření použitím jednoho dotazu. Měření jsou načítána v uživateli zadaném rozsahu časů. Dotazem získané měření jsou jednotlivé po sobě jdoucí senzorem provedené měření, které mají čas pořízení v zadaném intervalu. Tento dotaz je vykonáván zpracováním celého hodinového datového dokumentu najednou a výsledné hodnoty jsou uloženy do seznamu výsledků v *ResultSetu,* odkud je poté při dalším požadavku vrácen uživateli. Dokument se tedy nemusí zpracovávat vícekrát, ale jeho procházení probíhá pouze jednou.

SelectIntervalAverage je vylepšeným intervalovým dotazem. Uživateli nejsou vrácena pouze jednotlivá načtená data z dokumentu, ale dotaz provádí jejich agregaci v zadaném kroku. Uživatel při zadaní načtení dat v časovém intervalu *od-do* zadat také s jakým krokem mají být data získána. Interval kroku lze nastavit na 0 – sekundy, 1 – minuty. A poté je potřeba určit počet vybraných jednotek. Např. „`avgIntervalType = 0; avgIntervalTs = 60;`" značí krok 60 sekundového intervalu hodnot, který bude vrácen jako samostatná hodnota ve formě vypočtené průměrné hodnoty měření v tomto kroku. Vykonání dotazu probíhá získáním hodnot obsažených v následujícím kroku a výpočtem průměru ze získaných hodnot. Jednotlivé kroky musí být uvnitř zadaného časového intervalu.

SelectMin a *selectMax* vracejí měření ve kterém byla naměřená hodnota nejnižší (resp. nejvyšší) v zadaném časovém rozsahu. Při vyhledávání těchto hodnot pomáhá rychlosti zpracování slučování stejných měření do jednoho objektu (RLE), není u něj potřeba procházet jednotlivé hodnoty, ale vím už dopředu následující hodnoty v intervalu, ve kterém bylo RLE provedeno.

7.3.3 Dotaz pro mazání

Kromě dotazů pro vkládání a získávání dat je ještě implementována metoda, uvnitř třídy *RemoveQuery*, pro smazání senzoru z databáze a operační paměti. Mazání přidaných samostatných měření není v současné implementaci dostupné. Její případná implementace by však měla být schopná si poradit s odstraněním záznamu z komprimovaného objektu RLE, kdy by se jejím odstraněním objekt musel rozdělit do dvou samostatných objektů, protože by již nebyl dodržen časový rozestup mezi daty.

Kapitola 8

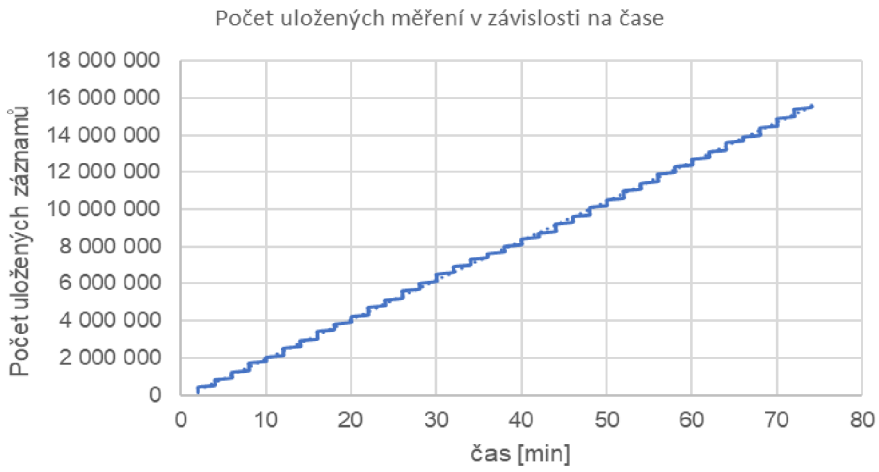
Testování

Testování probíhalo na virtualizovaném systému Ubuntu 16.04. Virtuálnímu systému bylo přidělen o 8 GB RAM a 2 fyzická jádra procesoru. Všechny dotazy probíhaly nad daty ze zálohy databázového systému z projektu BeeeOn. Při testování výkonnosti jsem zaměřil na porovnání původního databázového systému založeného na relační schématu (PostgreSQL) a vlastním navrženém řešení, které využívá NoSQL databázový systém MongoDB.

Databázová záloha obsahuje téměř 16 milionů naměřených záznamů, jedná se tedy o objemnou databázi.

8.1 Rychlost vkládání dat

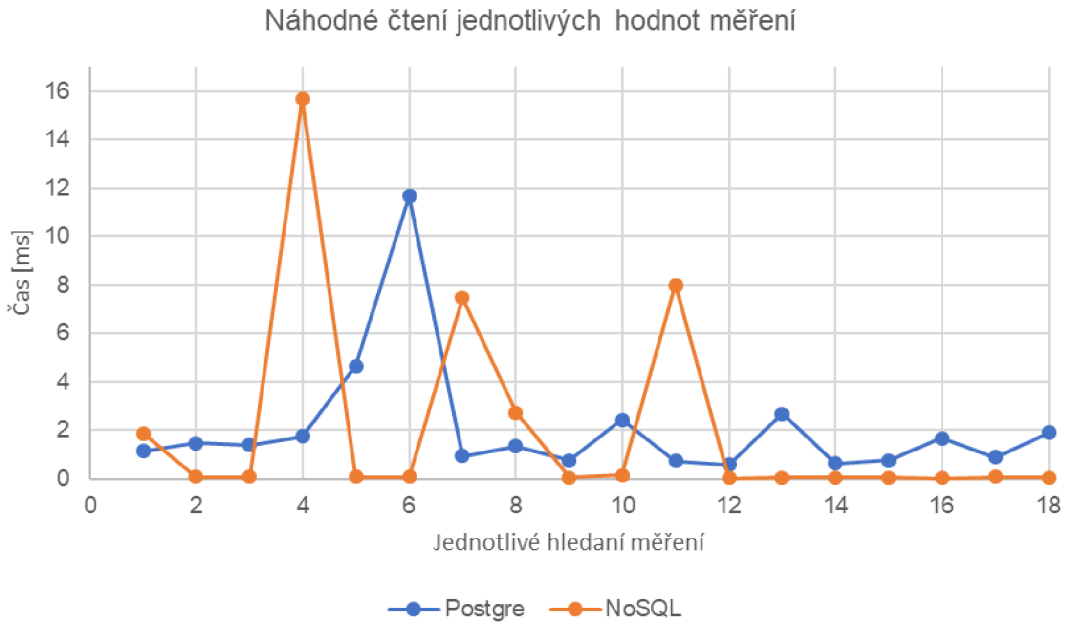
Prvním testem bylo provedení nahraní dat do NoSQL databáze. Celý databázový soubor byl v tomto testu zapsán do NoSQL databáze. Dle grafu [Graf 2] jde vidět, že databáze pro si udržovalo konstantní výkonnost během celého testu. Předchozí měření samotných databází MongoDB, Cassandra a PhilDB, které lze vidět v grafu [Graf 1] si udržovaly taky konstantní výkonnost, ale nedosahovaly takového výkonu. Vlastní implementace zvládla uložit do databáze celý soubor do 80 minut. Zápis probíhal rychlostí téměř 3600 měření za minutu. Při předchozím měření zvládla zapsat databáze MongoDB ve polovinu dat ve skoro 2krát delším čase. Nutno poznamenat, že rozdíl může být zapříčiněn použitím jazyka Java pro načítání dat relační databáze. V předchozím případě se jednalo o využití jazyka Python.



Graf 2: Výsledek vkládání dat měření do implementovaného programu s využitím MongoDB.

8.2 Rychlost výběru dat

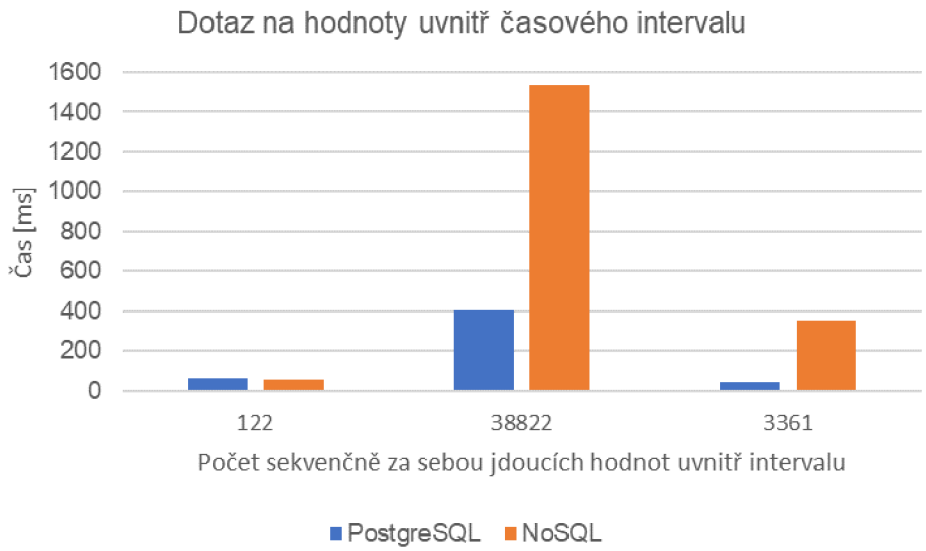
V měřeních rychlosti pro dotaz *select* jsem vybral u dotazů, které jsou definované časovým intervalem, 3 konkrétní hodnoty zpracovávaného intervalu. Kažkou akci vyhledání jsem provedl 30 opakování a naměřené hodnoty jsem zprůměroval. Každý zadaný interval měl uvnitř rozdílný počet záznamů. První interval obsahoval 122 záznamů, druhý interval 38 822 záznamů a poslední 3 361 záznamů. Toto měření již pro porovnání probíhalo současně s relační databází PostgreSQL. Obě databáze obsahovaly stejná data.



Graf 3: Výsledek provedení výběru náhodných hodnot.

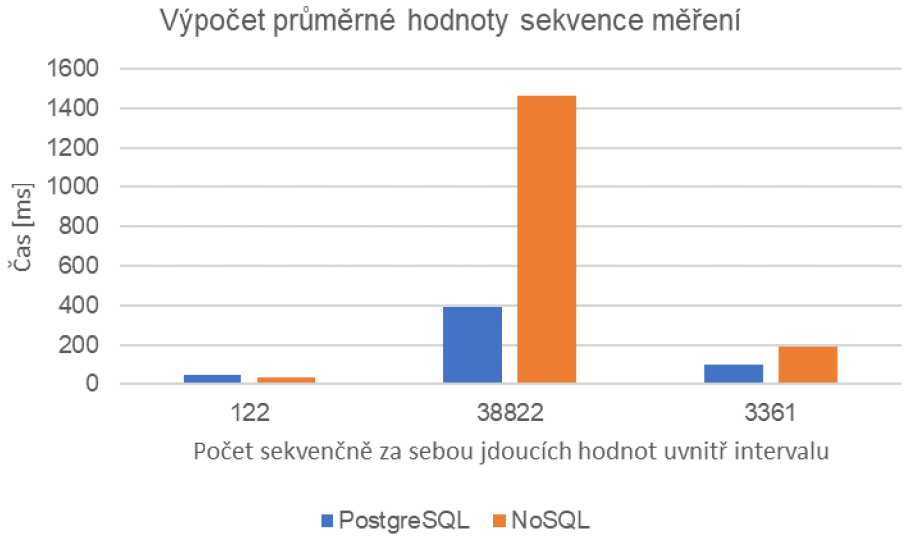
Výsledek měření pro náhodné čtení [Graf 3] jednotlivých hodnot vychází lépe pro NoSQL databázi. U čtení je možné sledovat velké výkyvy času, který je potřeba pro získání dané hodnoty. Jev je patrný zejména u vlastní implementace systému. Výkyvy jsou zapříčiněny načítáním velkého hodinového dokumentu do paměti. V případě, že byly dokumenty již nachystané v paměti, je získání hodnoty hluboko pod 1ms. Relační databáze podává vyrovnaný výkon, ale na řešení s NoSQL nestačí. Jedná se o operaci s náhodným přístupem, kde je formát hashovací tabulky výhodnější.

V případě sekvenčního čtení hodnot uvnitř zadaného intervalu [Graf 4] lze vidět, že se situace mění a v této části převládá klasická relační databáze. Ačkoliv je u NoSQL řešení implementován hodinový dokument, který shromažďuje více hodnot poblíž, tak přesto výkonově znatelně ztrácí. Z grafu je na první pohled zřejmé, že u krátkých sekvencí rozdíl není tak velký, ale společně s nárůstem po sobě jdoucích sekvenčních hodnot je rozdíl téměř čtyřnásobný.



Graf 4: Porovnání dotazu získání sekvence hodnot v daném intervalu.

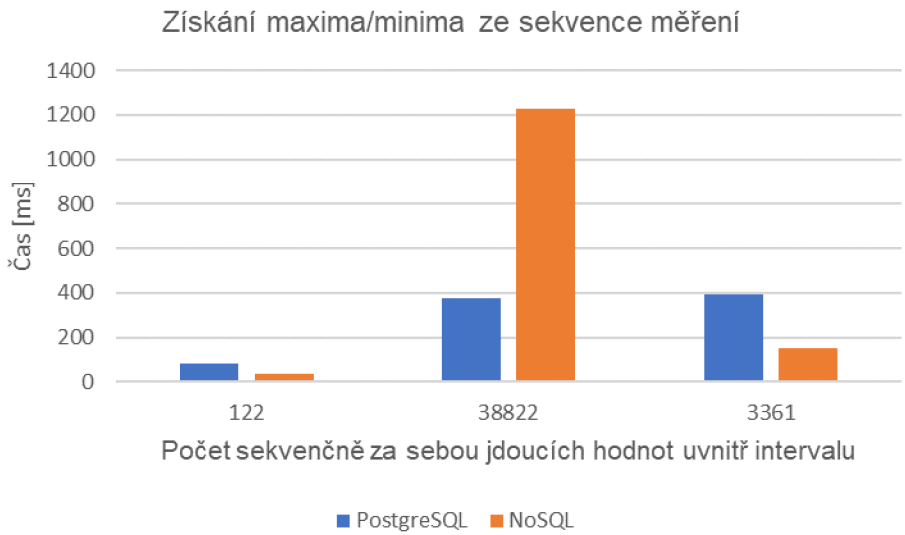
Obdobného výsledku, jaký byl vidět na předchozím grafu, se docílilo i u dotazu na získání průměrné hodnoty na měřeních v intervalu [Graf 5]. Tyto dotazy jsou si velice podobné, oba načítají sekvenci měření, kterou následně zpracovávají (počítají průměr). Identický výsledek se tedy dal očekávat a relační databáze získává převahu. Rychlosti sekvenčního zpracování dat u NoSQL řešení nepomáhá ani nutnost provádět dekompresi dat ze struktury RLE, která přidává režii navíc.



Graf 5: Výsledek výpočtu průměrné hodnoty na intervalu.

Posledním typem dotazu je získání minimální nebo maximální hodnoty [Graf 6]. Dotaz je opět proveden na předchozích datech, které testují jak práci s krátkou sekvencí, tak extrémně dlouhou. V tomto případě se u kratších sekvencí dat projevila optimalizace vyhledávání minimální a maximální hodnoty, která je způsobena provedenou komprimací dat. Sjednocená

data v tomto případě není potřeba procházet a testovat všechny, ale pouze jednu hodnotu reprezentující danou kolekci hodnot. Tímto se docílilo snížení čtení pro všechny délky sekvencí, které se zpracovávají. Avšak u nejdelší sekvence se stále projevuje chybějící výkonnost.



Graf 6: Výsledek dotazu nad sekvenčními daty a užití operací min/max.

8.3 Velikost dat na disku

Na prostorovou úsporu dat při uložení do databáze byl kladen důraz již od počátku návrhu struktury dat. Relační databáze zde není příliš úsporná a návrhem vlastní struktury, bylo by možné docílit značné úspory místa. Používaný formát JSON je ideálním řešením, které poskytuje velkou tvárnost, co se týče schémata uložení. Pro porovnání velikostí jsem využil exportu dat z obou databází a jejich následného porovnání. Data v relační databázi měla 720,9 MB, jedná se pouze o hodnoty z tabulky *log*, obsahující naměřená data. Datové BSON soubory z vlastního řešení, společně se soubory obsahující indexaci dat, zabírají 113 MB. Jedná se o značný rozdíl velikosti dat. Důvodem úspory bude již popsané provádění komprimace dat (RLE) a relativní časové značky dat.

Kapitola 9

Možné rozšíření

9.1 Agregace naměřených hodnot

Při ukládání příchozích senzorových dat se nabízí možnost předpočítání agregovaných hodnot, které by se poté využily při dotazovaní. Typickým příkladem by mohlo být předpočítání průměrných příchozích hodnot např. pro krok 1-5 minut. Zde záleží na hustotě měření. Při provádění měření do 10 záznamů za minutu, nemá smysl počítat průměrnou hodnotu pro každou minutu, výhodnější by to bylo ve větším časovém rozsahu. V implementované aplikaci by bylo možné agregované hodnoty ukládat přímo do hodinového dokumentu pod samostatný index.

Výsledný dokument by se však objemově zvětšil, ale docílilo by rychlejšího vykonání dotazů, které provádí výpočet průměrné hodnoty za běhu. V případě dotazů obsahující stejný interval hodnot pro zprůměrovaní je jako přepočítaný, by bylo získání hodnot pouze otázkou rychlosti čtení. V ostatních případech by se dalo předpočítaného průměru využít také. Je zde možnost skládat průměry a z předpočítaných dat se tedy můžou vytvářet průměry na větším časovém intervalu. Aby bylo možné skládat předchystaný průměr s dalšími příchozími daty, je potřeba si u něj uchovat, kromě hodnoty výsledného průměru, také počet atributů, ze kterých je průměr získán. Tím si zajistíme možnost vypočítat původní součet hodnot, který dále rozšíříme o nové hodnoty.

```
1     {
2           _id = 123/321/1492970400,
3           s = 1492970490,
4           v = {
5                 ...
6           }
7           a = {
8                 59 = [avg_val, count],
9                 119 = [avg_val, count],
10                ...
11          }
12    }
```

Alg. 9: Možná implementace ukládání agregovaných hodnot uvnitř hodinového dokumentu pod indexem „a“.

9.2 Snížení granularity dat

Uložená data se postupem času s příchodem novějších dat stávají neaktuální a pro uživatele méně zajímavá. Současná implementace systému stárnutí dat řeší automatickým odstraněním dat z databáze v případě jejich, že jsou záznamy starší než nastavená limitní hodnota (např. 20 dnů). Kromě toho se nabízí řešení problému pomocí postupného snižování granularity dat. Hodnoty starší, než určený limit je možné sloučit a zprůměrovat. Například až na úroveň jedné hodiny. Případně snížit granularitu o něco méně a data kódovat pomocí rozdílu sousedních hodnot. Výhodou těchto řešení je stálý přístup i k historickým datům, které sice nebudou tak přené, ale budou podávat jistý přehled o situaci v historii.

9.3 Cache dotazů

Pro urychlení dotazů typu *select* se nabízí možnost uchovávat výsledky posledních vykonaných dotazů. S touto možností jsem při návrhu počítal a dotazovaná data jsou uložena uvnitř třídy *ResultSet*. K dispozici jsou i informace o typu původního dotazu a časovém intervalu dat. Instanci této třídy by bylo možné uchovat v paměti a v případě podobného typu dotazu, nad daty se vzájemným časovým průsekem, by došlo k využití hodnot ze staršího dotazu. Nemuselo by se však jednat pouze o pouhé využití dat, ale bylo by možné starší uložené dotazy slučovat do větších jednolitých celků. Nový příchozí dotaz by se porovnal s dostupnými výsledky a ze systému by docházelo k načtení pouze časových úseků v dotazu, které se v historii dotazů nevyskytují.

9.4 Rozhraní pro C++

Aplikace je psána kompletně v jazyce Java. V případě nasazení systému na produkční server uvažovaného IoT systému BeeeOn, na jehož základě byla aplikace navržena, dojde k překážce propojení se serverovou aplikací, která je implementována v jazyce C++. Aby bylo možné implementovanou aplikaci využívat při více příležitostech, bylo by vhodné navrhnout a připravit rozhraní pro volání funkcí aplikace z jazyka C a C++.

Kapitola 10

Závěr

Cílem diplomové práce bylo seznámení se s principy NoSQL databází a charakteristikami senzorových dat. S tím souvisí také téma časových řad v databázích, jehož jsou senzorová data příkladem. Na základě nastudovaných znalostí poté navrhnout vlastní formát uložení dat, tak aby bylo dosaženo snadného indexování uložených dat, nízké latence při přístupu k uloženým datům a rychlého ukládání získaných dat.

Implementovaná aplikace využívá existující NoSQL databází MongoDB pro ukládání dat a stejnými principy převzatými z NoSQL paradigmatu implementuje strukturu a práci s daty v operační paměti. V implementaci se podařilo docílit úspory požadovaného místa pro uložení, rychlého zápisu dat do databáze, společně s nízkou dobou přístupu k náhodným datům. Špatné výsledky pak byly dosaženy u práce se sekvencí dat, která jsou definována jistým časovým úsekem. Zde se projevuje nevýhoda získávání hodnot pomocí hashování. Pro databázi pracující s daty časových řad je práce se sekvencemi důležitá, protože se primárně zpracovávají po úsecích dat. Pro nasazení do produkčního prostředí by bylo potřeba upravit práci s daty v paměti a přidat lepší podporu pro práci se sekvenčními daty.

Aplikace je volně dostupná a šiřitelná se zdrojovými kódy pod volnou licencí.

Literatura

[1] Svět Hardware: *„Internet of Things: Propojená budoucnost“* [online]. 2014, aktualizováno 2014-10-24 [cit. 2017-01-6]. Dostupné na URL: <http://www.svethardware.cz/internet-of-things-propojena-budoucnost/39560-2>

[2] Vampola, P.: Databáze pro inteligentní domácnost., Diplomová práce, str. 6, Brno: 2015 [cit. 2017-01-7].

[3] Vampola, P.: Databáze pro inteligentní domácnost., Diplomová práce, str. 13, Brno: 2015 [cit. 2017-01-7].

[4] Fowler, M.: *„GOTO Conferences: Introduction to NoSQL“* [online]. 2012, [cit. 2017-01-7]. Dostupné na URL: <http://gotocon.com/aarhus-2012/presentation/Introduction%20to%20NoSQL>

[5] Moniruzzaman, A., Hossain, S. A.: *„NoSQL Database: New Era of Databases for Big data Analytics – Classification, Characteristics and Comparison“* [online]. 2014, [cit. 2017-01-8]. Dostupné na URL: <https://arxiv.org/ftp/arxiv/papers/1307/1307.0191.pdf>

[6] Bigdatanerd: *„Why NoSQL – Part1 – CAP theorem“* [online]. 2011, aktualizováno 2011-12-08, [cit. 2017-01-9]. Dostupné na URL: <https://bigdatanerd.wordpress.com/2011/12/08/why-nosql-part-1-cap-theorem/>

[7] Wikipedia: *„Key-value database“* [online]. 2012, aktualizováno 2016-11-03, [cit. 2017-01-9]. Dostupné na URL: <https://en.wikipedia.org/wiki/Key-value_database>

[8] Wikipedia: *„NoSQL“* [online]. 2010, aktualizováno 2017-05-01, [cit. 2017-01-9]. Dostupné na URL: <https://en.wikipedia.org/wiki/NoSQL>

[9] RefreshMyMind: *„Which NoSQL database should I use?“* [online]. 2016, aktualizováno 2016-02-15, [cit. 2017-01-9]. Dostupné na URL: <http://refreshmymind.com/nosql-database-comparison/>

[10] Basho: *„Time Series Databases Explained“* [online]. 2006, aktualizováno 2017-01-9, [cit. 2017-01-10]. Dostupné na URL: <https://en.wikipedia.org/wiki/Time_series>

[11] Wikipedia: *„Time Series“* [online]. 2006, aktualizováno 2017-01-9, [cit. 2017-01-10]. Dostupné na URL:
<https://en.wikipedia.org/wiki/Time_series>

[12] MacDonald, A.: *„PhilDB: the time series database with built-in change logging“* [online]. 2016, [cit. 2017-01-10]. Dostupné na URL:
<https://peerj.com/articles/cs-52.pdf>

[13] KairosDB: *„KairosDB“* [online]. 2015, [cit. 2017-01-10]. Dostupné na URL:
<https://kairosdb.github.io/>

[14] Hawkins, B.: *„KairosDB Cassandra Schema“* [online]. 2014, [cit. 2017-01-10]. Dostupné na URL:
<https://prezi.com/ajkjic0jdws3/kairosdb-cassandra-schema/>

[15] Larsen, Ch.: *„OpenTSDB 2.0“* [online]. 2014, [cit. 2017-01-10]. Dostupné na URL:
<http://www.slideshare.net/HBaseCon/ecosystem-session-6>

[16] Redmine: *„Schéma DB – Server – Redmine“* [online]., [cit. 2017-01-10]. Dostupné na URL:
<https://ant-2.fit.vutbr.cz/projects/server/wiki/Sch%C3%A9ma_DB>

[17] Vampola, P.: Databáze pro inteligentní domácnost., Diplomová práce, str. 28, Brno: 2015 [cit. 2017-01-10].

[18] D. Ramesh, A. Sinha and S. Singh: Data modelling for discrete time series data using Cassandra and MongoDB. In 3rd International Conference on Recent Advances in Information Technology (RAIT), [online], Dhanbad: 2016 [cit. 2017-01-10].
<https://doi.org/10.1109/RAIT.2016.7507966>

[19] Wikipedia: *„B+ strom“* [online]. 2007, aktualizováno 2016-07-27, [cit. 2017-01-11]. Dostupné na URL:
<https://cs.wikipedia.org/wiki/B%2B_strom>

[20] Wikipedia: *„Java Database Connectivity“* [online]. 2006, aktualizováno 2016-12-06, [cit. 2017-05-19]. Dostupné na URL:
<https://cs.wikipedia.org/wiki/Java_Database_Connectivity>

Seznam příloh

Příloha A: Manuál – Spuštění aplikace
Příloha B: DVD

A. Manuál k aplikaci

Návod pro zprovoznění aplikace:

Prerekvizitou pro správné fungování aplikace je potřeba mít v počítači nainstalovanou a spuštěnou databázovou aplikaci MongoDB v nejnovější verzi. Pomocí příkazů:

```
service mongod start
service mongod status
```

je možné aplikaci MongoDB spustit a zkontrolovat, zda běží na pozadí.

Pro správné fungování je dobré využít Javu verze 1.8 a nastavit tuto verzi také jako výchozí pro překlad. Zdrojové kódy je možné importovat do vývojářského prostředí. Projekt využívá Maven, není tedy potřeba řešit závislosti na přídavných balíčcích a importovaná aplikace by měla být spustitelná.

Aplikace obsahuje také třídu *Main*, kde je znázorněna práce s aplikací formou ukázek, jako spuštění, přidávání do databáze, dotazy apod. Každá akce obsahuje vysvětlující komentář a tiskne data na výstup konzole.

Před spuštěním aplikace je důležité správně nakonfigurovat informace o připojení k databázi. Konfigurace se nachází ve třídě *Constants,* kde je důležité vyplnit proměnné znázorněné na obrázku.

```
/**
 * Před spuštěním je potřeba nakonfigurovat.
 * Informace pro připojení k běžící MongoDB databázi.
 */
public static final String databaseName = "test_database";
public static final int databasePort = 27017;
public static final String databaseHost = "localhost";


/**
 * Adresa serveru na kterém poběží tato relace.
 * Port na kterém bude aplikace naslouchat kvůli synchronizaci.
 */
public static final String serverIp = "localhost";
public static final Integer serverPort = 55123;
}
```

Obr. 24: Konstanty pro nastavení uživatelem v třídě Constanstans. Primární je vyplnit údaje pro připojení k MongoDB.

Před samotnou prací s aplikací je důležité zavolat následující funkce:

```
new SensorCollection(); // Vytvoření instance pro paměťovou strukturu.
ServerListener.serverInit(); // Nepovinná, vlákno pro komunikaci s ostatními
                             // servery.
Daemon.init(20, 3600); // Nepovinné, vlákno pro údržbu mezipaměti.
```

B. Obsah DVD

- Zdrojové kódy projektu
- Text práce
- Licence programu