TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Použití databáze časových řad v serverové aplikaci pro ukládání dat kvality elektrické energie

Bakalářská práce

Studijní program: B2646 – Informační technologie
Studijní obor: 1802R007 – Informační technologie

Autor práce: **Pavel Pilař**
Vedoucí práce: Ing. Tomáš Bedrník

Liberec 2020

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání bakalářské práce

Použití databáze časových řad v serverové aplikaci pro ukládání dat kvality elektrické energie

Jméno a příjmení: **Pavel Pilař**
Osobní číslo: M17000086
Studijní program: B2646 Informační technologie
Studijní obor: Informační technologie
Zadávající katedra: Ústav mechatroniky a technické informatiky
Akademický rok: **2019/2020**

Zásady pro vypracování:

1. Seznamte se s API stávající serverové aplikace.
2. Prozkoumejte možnosti databází časových řad. Vyberte nejvhodnější pro nahrazení nebo doplnění stávajícího řešení.
3. Navrhněte způsob, jak vybranou databázi integrovat do stávajícího API s minimem změn.
4. Navrhněte nové API, které z původního vychází, ale rozšiřuje ho o možnosti a výhody databází časových řad.
5. Vybrané řešení implementujte jako ukázkovou serverovou aplikaci.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	30–40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] DUNNING, Ted a Ellen FRIEDMAN, 2014. *Time Series Databases: New Ways to Store and Access Data*. 1 edition. Sebastopol, CA: O?Reilly Media. ISBN 978-1-4919-1472-4.

[2] ANON., nedatováno. Time Series Database (TSDB) Explained | InfluxDB. *InfluxData* [online] [cit. 2019-10-09]. Dostupné z: https://www.influxdata.com/time-series-database/

[3] FREEDMAN, Mike, 2019. TimescaleDB vs. InfluxDB: Purpose built differently for time-series data. *Timescale Blog* [online] [cit. 2019-10-09]. Dostupné z: https://blog.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/

Vedoucí práce:	Ing. Tomáš Bedrník Ústav mechatroniky a technické informatiky
Datum zadání práce:	10. října 2019
Předpokládaný termín odevzdání:	18. května 2020

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 10. října 2019

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

31. 5. 2020 Pavel Pilař

Použití databáze časových řad v serverové aplikaci pro ukládání dat kvality elektrické energie

Abstrakt

Tato práce se zabývá výběrem vhodné databáze časových řad a implementací serverové aplikace, která vybranou databázi využívá. Cílem bylo především zjistit výhody a nevýhody time series databází a možnosti jejich využití místo stávajícího systému, který je postavený na relační SQL databázi. Konkrétně jsou v práci porovnány databáze InfluxDB, TimescaleDB a OpenTSDB. Při výběru byl pak kladen důraz hlavně na jednoduchost implementace požadovaných funkcí aplikace a možnosti dalšího rozšiřování.

Výsledkem porovnání je výběr databáze TimescaleDB, založené na relační databázi PostgreSQL. Mezi její výhody patří zejména plná podpora jazyka SQL, díky kterému je možné většinu funkcionality aplikace realizovat přímo v databázi. Ostatní databáze časových řad nabízejí daleko menší flexibilitu a implementaci složitějších funkcí je tak třeba přenést do serverové aplikace.

V ukázkové implementaci aplikace je proto kladen důraz na využití co nejvíce možností vybrané databáze. Samotná serverová aplikace tak má na starosti prakticky pouze ověřování uživatelů a jinak funguje jen jako proxy server, který zajišťuje správný formát dotazů. Všechny složitější funkce, jako je například agregace dat a další práce s měřeními probíhá automaticky přímo v databázi.

Klíčová slova: Databáze časových řad, internet věcí, API, SQL, serverová aplikace

Server Application for Storing Power Quality Data Using Time Series Database

Abstract

This thesis is focused on choosing an appropriate time series database and implementing a server application that uses the chosen database. The target was mainly to determine the pros and cons of time series databases and possibilities of using them instead of the current system, which uses a relational SQL database. Specifically, the databases compared in this work are InfluxDB, TimescaleDB, and OpenTSDB. During the selection, emphasis was placed mainly on the simplicity of implementation of the required functions of the application and the possibility of further expansion.

The result of the comparison is the selection of the TimescaleDB database, based on the PostgreSQL relational database. Its advantages include mainly full SQL support, thanks to which it is possible to implement most of the application's functionality directly in the database. Other time series databases offer far less flexibility, so the implementation of more complex functions needs to be done in the server application.

Therefore, in the example implementation of the application, emphasis is placed on using as many tools of the selected database as possible. The server application itself is thus responsible only for user authentication and otherwise acts only as a proxy server, which ensures the correct format of queries. All complex functions, such as data aggregation and other work with measurements, take place automatically directly in the database.

Keywords: Time series database, internet of things, API, SQL, server application

Obsah

1 Úvod **9**

2 Databáze časových řad **10**
- 2.1 InfluxDB . . . 11
  - 2.1.1 TICK stack . . . 11
- 2.2 TimescaleDB . . . 13
  - 2.2.1 Licence . . . 14
- 2.3 OpenTSDB . . . 14
- 2.4 Další možnosti . . . 15

3 Porovnání databází **17**
- 3.1 Možnosti implementace funkcí . . . 17
  - 3.1.1 Správa uživatelů a uživatelských práv . . . 17
  - 3.1.2 Tagování dat a vyhledávání . . . 18
  - 3.1.3 Nasazení na ARM zařízeních . . . 18
  - 3.1.4 Agregace dat . . . 18
  - 3.1.5 Přepočty hodnot . . . 18
  - 3.1.6 Další funkce . . . 19
- 3.2 Výběr databáze . . . 19

4 Nastavení databáze **21**
- 4.1 Schéma databáze . . . 21
- 4.2 Tabulky pro měření . . . 24
  - 4.2.1 Agregace dat . . . 24
- 4.3 Zpracovávání dat . . . 27

5 Serverová aplikace **29**
- 5.1 Komunikace s databází . . . 29
- 5.2 API endpointy . . . 31

6 Možnosti rozšíření **33**
- 6.1 Rozšíření databáze . . . 33
- 6.2 Rozšíření aplikace . . . 33
  - 6.2.1 Implementace API . . . 34

7 Závěr 36

Použitá literatura 38

Přílohy 39

A Nastavení a spuštění aplikace 39

B Přiložené soubory 40

Seznam obrázků

2.1 Ukázka GUI Chronograf 12
2.2 Ukázka GUI OpenTSDB 15
4.1 Konceptuální schéma databáze. 22

Seznam zdrojových kódů

4.1 Kontrola uživatelských práv 22
4.2 Trigger pro automatické tagování dat 23
4.3 Automatická agregace dat 25
4.4 Pohled nad daty 25
4.5 Trigger pro vkládání dat 26
4.6 Nastavení komprese starých dat 27
5.1 Generování SQL dotazů 30
5.2 Implementace API endpointu 31
5.3 Ukázka chybového objektu 32
6.1 Naměřené hodnoty události 34
6.2 Přepočet naměřené hodnoty 34
6.3 Streamování odpovědi 35
A.1 Konfigurační soubor aplikace 39

Seznam zkratek

API	Application Programming Interface, rozhraní pro programování aplikací
CLI	Command Line Interface, příkazový řádek
DB	Database, databáze
DML	Data Manipulation Language, jazyk pro manipulaci s daty
GPS	Global Positioning System, globální polohový systém
GUI	Graphical User Interface, grafické uživatelské rozhraní
HTTP	Hypertext Transfer Protocol, protokol pro přenos hypertextu
IoT	Internet of Things, internet věcí
JSON	JavaScript Object Notation, JavaScriptový objektový zápis
JWT	JSON Web Token, způsob ověření uživatele
ORM	Object-relational Mapping, objektově relační mapování
RDBMS	Relational Database Management System, systém řízení relační báze dat
REST	Representational State Transfer, architektura rozhraní webových služeb
SQL	Structured Query Language, strukturovaný dotazovací jazyk
TSDB	Time Series Database, databáze časových řad

1 Úvod

Počet zařízení, která měří a sbírají různé hodnoty, stále roste. Nejde již jen o průmyslová zařízení nebo monitorování IT infrastruktury, ale také o stále se rozrůstající segment chytrých domácností. Kromě měření a sbírání těchto dat je ale stejně důležité, nebo i důležitější, jak je efektivně zpracovávat a jak tyto výsledky využít.

Vzhledem k tomu, jak rychle se toto odvětví vyvíjí, je nutné myslet nejen na to, jak tyto data ukládat a zpracovávat nyní, ale také počítat s postupným růstem jejich počtu a tím i zvyšující se složitostí práce s nimi. Je proto vhodné zvolit takové technologie, které toto škálování zvládnou.

Při vylepšování nebo nahrazování části již existujícího softwarového systému je třeba myslet na zpětnou kompatibilitu. Často není možné vytvořit nové rozhraní a je tak nutné kompletně napodobit původní funkcionalitu, i za cenu neoptimálního využití nových součástí. I při výběru vhodné náhrady tak nejsou důležité jen její funkce a možnosti, ale také kompatibilita se zbytkem systému a složitost implementace případných změn.

Cílem této práce proto není pouze vybrat nejlepší řešení pro ukládání dat kvality elektrické energie, ale také takové, které umožní zachovat již existující API. To klade na výběr specifické požadavky, které by při vytváření nového API neexistovaly.

2 Databáze časových řad

Databáze časových řad (time series databáze) jsou databáze optimalizované pro ukládání a práci s časovými řadami. Časové řady jsou série měření nebo událostí, které jsou sledovány, ukládány a zpracovávány v čase [1].

Hlavním důvodem jejich vzniku bylo, že již existující databáze nebyly schopné efektivně zpracovat množství dat, které stále se zvětšující množství IoT zařízení vytváří. Například relační databáze se pro časové řady nehodí, protože se po každé vložené hodnotě musí aktualizovat indexy celé tabulky. S rostoucím počtem řádků v tabulce roste také časová náročnost této operace a tím klesá rychlost vkládání nových dat, zejména jakmile začne při práci s indexy docházet k jejich swapování na disk [2]. NoSQL databáze se pro časové řady hodí o něco více, zejména díky jejich vysoké škálovatelnosti. Ty naopak ale naráží na to, že neobsahují prakticky žádné nástroje ani pomůcky ke zpracování takto velkého množství dat. V případě jejich použití je tak třeba všechnu takovou funkcionalitu implementovat mimo databázi v aplikacích, které s daty pracují.

Základními vlastnostmi těchto databází tak je hlavně schopnost přijímat velké množství dat a efektivně s nimi pracovat. Mají proto na rozdíl od obecných databází speciální funkce, které umožňují s uloženými daty pracovat podle času. Je tak vhodné je použít i v případě, že jejich optimalizace nejsou potřeba, právě díky těmto funkcím, které usnadňují zpracovávání uložených dat. Často také nabízejí možnost propojení s nástroji pro vizualizaci a analýzu dat, jako je například Grafana.

Právě kvůli jejich optimalizacím pro časové řady jsou ale také méně flexibilní a při jejich výběru je tak nutné zvážit všechny výhody a nevýhody, které jejich použití přináší. To je důležité zejména pokud není cílem navrhnout celý nový systém, ale pouze databázi časových řad využít v již existující implementaci.

2.1 InfluxDB

InfluxDB je NoSQL databáze speciálně vytvořená pro časové řady. Je tedy zaměřená na rychlý zápis, kompresi a čtení dat. Pro práci s daty obsahuje dva různé jazyky, InfluxQL a Flux. InfluxQL je jazyk podobný SQL a umožňuje používat na data některé jeho příkazy, jako např. `WHERE` nebo `GROUP BY`. Vzhledem k rozdílům mezi InfluxDB a SQL databázemi ale není možné provádět operace jako spojování tabulek. Flux je poté funkcionální jazyk, pomocí kterého se s daty pracuje řetězením různých funkcí, například pro omezení času a filtrování.

Každý zápis v databázi je definován jménem, tagy a časem a může mít několik hodnot. Jméno je obdoba tabulky v jiných databázích a slouží k rozdělení uložených hodnot do skupin. Pomocí tagů se rozlišují jednotlivé časové řady a vyhledává se mezi nimi. Tagy mají formát *klíč-hodnota*. Čas je ukládán jako unixový čas v nanosekundách. Pro hodnoty se používá stejný formát jako pro tagy, ale narozdíl od nich nejsou indexované. Validní datové typy hodnot jsou integery, floaty, stringy a booleany. Hodnoty s plovoucí řádovou čárkou mají vždy jednoduchou přesnost, dvojitou přesnost nelze ukládat [3].

Data uložená v databází není možné měnit. Je tak nutné, aby měření obsahovaly všechny potřebné tagy a hodnoty již při jejich nahrání do databáze. Jakmile jsou v databázi, změnit je lze pouze jejich přečtením, smazáním a po provedení změn opětovným nahráním.

InfluxDB poskytuje velké množství funkcí a nástrojů pro práci s časovými řadami a pro jejich zpracovávání. Umožňuje například automaticky snižovat rozlišení starých dat nebo stará data mazat. K tomu obsahuje velké množství matematických funkcí, používaných pro analýzu dat.

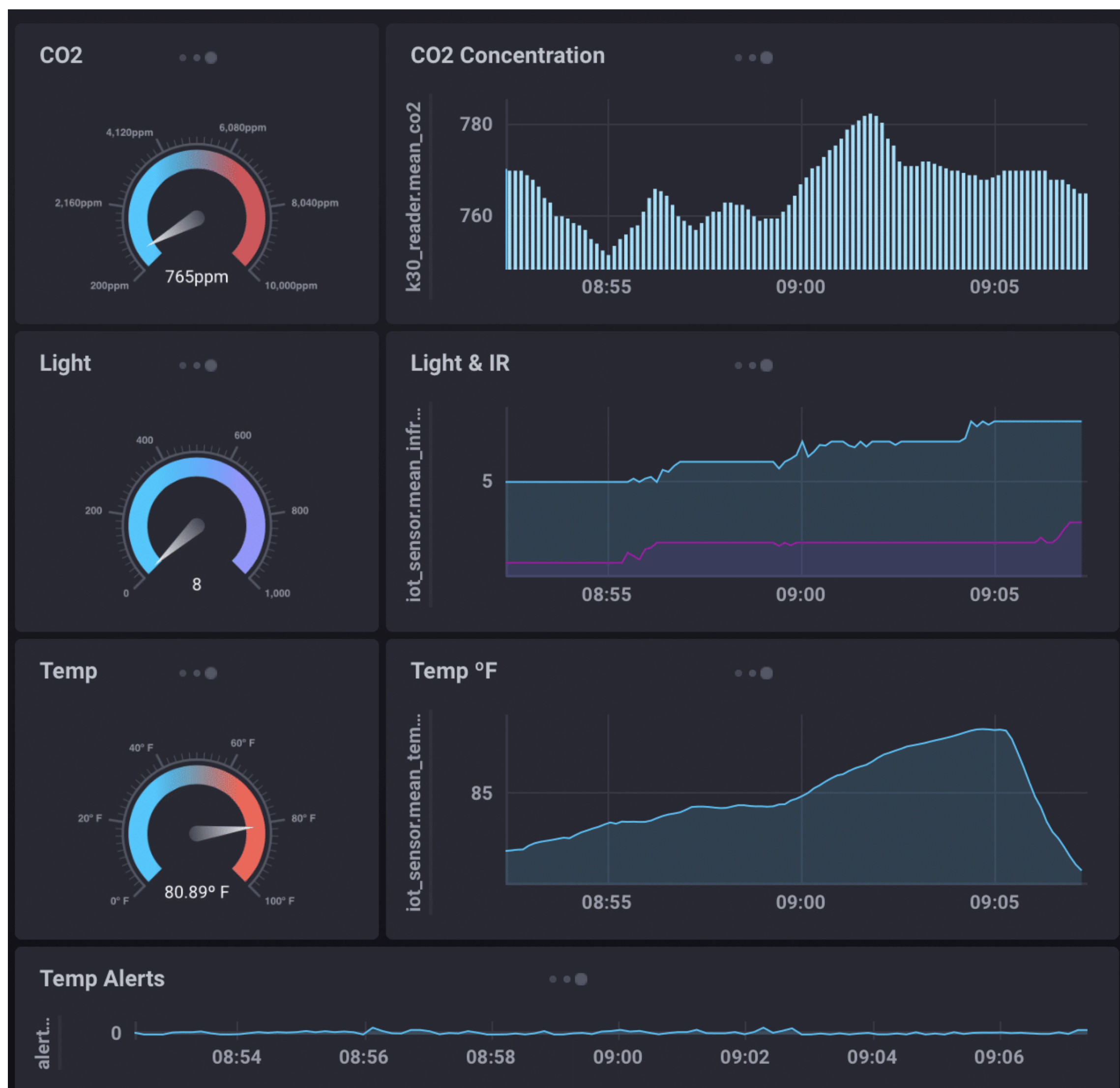
Komunikace s InfluxDB se provádí pomocí HTTP API, obsahuje ale také CLI klienta, který toto API využívá. Kromě toho nabízí knihovny pro většinu nejrozšířenějších programovacích jazyků, jako je JavaScript, Python i Java.

2.1.1 TICK stack

TICK stack je soubor programů pro sběr, ukládání, analýzu a zobrazování time series dat. Telegraf pro sběr a odesílání měření, InfluxDB jako úložiště, Chronograf jako uživatelské rozhraní a Kapacitor pro zpracování dat.

Ukázka grafického rozhraní Chronograf je na obrázku 2.1. To umí z dat vytvářet různé typy grafů, ale také tabulky či samostatné hodnoty a umožňuje je libovolně rozestavit po pracovní ploše.

Kapacitor umožňuje zpracovávat a analyzovat data v databázi. Umí například upozorňovat na překročení povolených hodnot nebo na změny za určitý časový interval. Informace o těchto událostech poté odešle, nebo uloží. Data umí také přepočítávat a spojovat, kromě InfluxDB i z jiných zdrojů, pro vytvoření nových časových řad. Tyto akce se definují pomocí jazyku TICKscript. Jeho nastavení se provádí pomocí rozhraní Chronograf, nebo HTTP API.



Obrázek 2.1: Ukázka GUI Chronograf [4]

2.2 TimescaleDB

TimescaleDB je databáze časových řad založená na relační databázi PostgreSQL. Umožňuje díky tomu používat všechny možnosti RDBMS a rozšiřuje a optimalizuje je pro využití s time series daty. Narozdíl od ostatních databází časových řad tak poskytuje možnost ukládat kromě číselných dat například i data typu JSON, nebo GPS souřadnice.

TimescaleDB řeší neoptimálnost relačních databází automatickým dělením tabulek s časovými řadami do tzv. *chunků*. Ty pak mohou mít takovou velikost, aby se jejich indexy vešly do operační paměti a práce s nimi tak byla rychlejší. Při práci s takto rozdělenou tabulkou však uživatel nepozná žádný rozdíl, neboť TimescaleDB ji prohledává, jako by rozdělená nebyla [5]. Díky tomu, že i takto rozdělené tabulky s časovými řadami, nazvané *hypertables*, obsahují plnou podporu jazyka SQL, je možné na ně také použít všechny nástroje PostgreSQL, jako jsou například referenční či doménové integrity, ale i triggery či procedury. Je tak také možné použít i všechny existující knihovny pro komunikaci a práci s SQL databázemi, nebo pro vizualizaci dat.

Díky chunkům je možné také optimalizovat další funkce databáze, například díky mazání celých chunků narozdíl od mazání po řádcích. Databáze se tak těmito operacemi nemusí zdržovat a může více svého výkonu využít pro zpracovávání nově příchozích dat.

Rozšíření TimescaleDB také obsahuje několik funkcí pro analýzu a úpravu dat. Jednou z nich je například `histogram` nebo agregační funkce `first` a `last` pro zjišťování první a poslední hodnoty časového intervalu. Další funkce je poté možné jednoduše přidat, díky možnostem SQL.

Pro práci s daty podle času obsahuje funkci `time_bucket`. Ta umožňuje zadat libovolný časový interval a pomocí agregačních funkcí hodnoty seskupit do těchto intervalů. V případě chybějících hodnot je možné použít funkci `time_bucket_gapfill`, která umožňuje hodnoty dopočítat například pomocí lineární interpolace.

Stejně jako InfluxDB i TimescaleDB umožňuje automatickou agregaci dat. Ta se definuje pomocí `time_bucket` funkcí, které určují, jaké intervaly se mají agregovat. Díky podpoře jazyka SQL je kromě již existujících agregačních funkcí možné definovat i vlastní. Mezi další funkce patří také komprese dat, která automaticky převádí data do formátu, který efektivněji využívá úložiště. Nevýhodou komprese je, že zkomprimovaná data již nejde dále měnit nebo k nim přidávat další.

2.2.1 Licence

TimescaleDB je dostupná ve 3 verzích s různými funkcemi a omezeními. Základní verze Open-source s licencí Apache 2.0, verze Community pod licencí TSL (Timescale License) a Cloud s komerční licencí. Verze Open-source, která díky své licenci neklade žádná omezení na použití, oproti Community neobsahuje některé důležité funkce pro práci s time series daty. Licence Community verze zamezuje nabízení samotné databáze jako služby [6]. Cloud poté nabízí služby jako cloudový hosting a větší podporu. Funkce, které ve verzi Open-source oproti Community chybí, jsou automatická agregace, interpolace chybějících dat, komprese dat a další.

2.3 OpenTSDB

OpenTSDB je založená na databázi Apache HBase. S tou komunikuje pomocí Time series daemonů, které zprostředkovávají zápis a čtení dat. Funguje tak spíše jako vrstva mezi aplikací a databází, která zpřístupňuje různé optimalizace pro práci s časovými řadami. Oproti ostatním databázím časových řad je více zaměřená jen na jejich ukládání a nabízí tak menší možnosti pro jejich analýzu. Obsahuje tak například menší množství matematických funkcí, které je možné na data používat, nebo méně možností pro nastavení mazání starých dat.

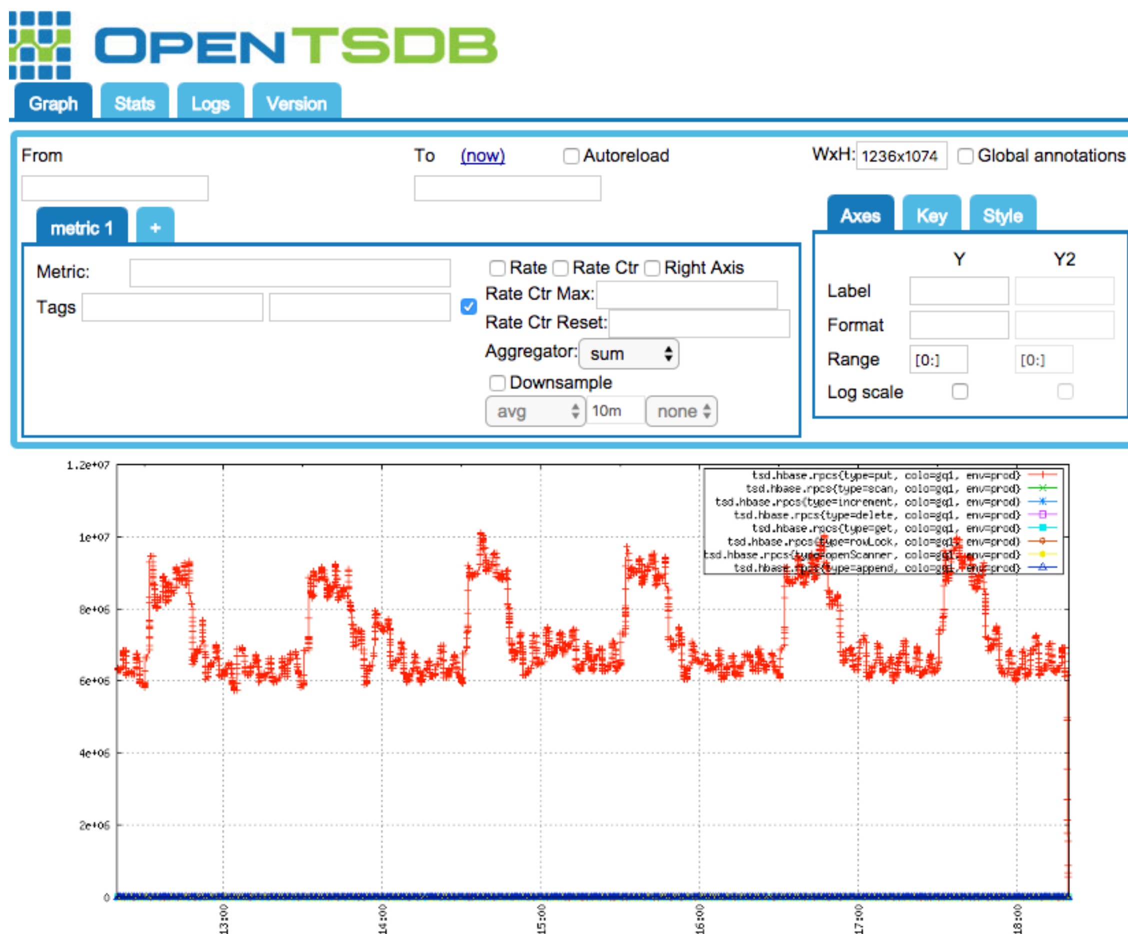
Data v databázi jsou tvořeny názvem, časem, hodnotou a tagy. Čas může mít až milisekundovou přesnost, hodnoty mohou být buďto celočíselné, nebo s plovoucí řádovou čárkou. Pro celočíselné hodnoty databáze používá proměnlivou velikost, 1-8 bytů podle potřeby. Hodnoty s plovoucí řádovou čárkou mají jednoduchou přesnost. Tagy mají stejný formát jako v InfluxDB, tedy *klíč-hodnota* [7].

Oproti ostatním databázím obsahuje jednoduché webové rozhraní. Jeho ukázka je v obrázku 2.2, který obsahuje i vygenerovaný graf. To umožňuje z jednotlivých časových řad vygenerovat graf a zobrazit ho v prohlížeči, ale také zpřístupňuje logy a statistiky databáze. Kromě možností pro změnu vzhledu grafů umožňuje také nastavení rozlišení dat i rozlišení grafu samotného.

Na rozdíl od ostatních databází ale také neumožňuje jedním požadavkem získat více časových řad najednou. V případě, že cílem dotazu je více časových řad, musí být součástí dotazu i agregační funkce, kterou se všechny spojí do jedné. Pro přečtení více řad je tak nutné pro každou vytvořit vlastní požadavek na databázi.

Dále umožňuje rozšíření svojí funkcionality pomocí pluginů. Ty umožňují databázi rozšířit například o další protokoly pro komunikaci, serializaci dat, nebo vyhledává-

ní mezi nimi. Rozšířit jde také zobrazování grafů, například o možnost generování histogramů. Time series daemony OpenTSDB jsou napsány v Javě. I vlastní pluginy je tak možné vytvářet pomocí Javy a dále jimi rozšiřovat možnosti databáze.



Obrázek 2.2: Ukázka GUI OpenTSDB [7]

2.4 Další možnosti

Tato práce se soustředí na databáze časových řad, které umožňují měřená data ukládat a později je zpracovávat a analyzovat pomocí jejich nástrojů. Existují ale také takzvané *in-memory* databáze. Ty místo na disk všechny informace ukládají do operační paměti, kde je práce s nimi rychlejší [8]. Vzhledem k výrazně menším velikostem operačních pamětí oproti pevným diskům jsou ale omezené svojí kapacitou pro dlouhodobé ukládání dat. Jsou tak vhodné zejména pokud je důležitá nízká odezva a vysoká efektivita práce s daty nebo není potřeba uchovávat přijatá data dlouhodobě.

Další možností jsou *big data* databáze. Ty jsou navržené a optimalizované pro velké množství dat a možnosti škálování. Patří mezi ně například i Apache HBase, na které je založená time series databáze OpenTSDB. Při jejich použití se ale předpokládá, že práce s daty a jejich analýza bude probíhat mimo databázi a poskytují tak kromě zápisu a čtení dat jen malé množství dalších nástrojů.

V případě, že by žádná databáze časových řad nevyhovovala požadavkům aplikace, big data databáze by za cenu výrazně složitější implementace serverové aplikace poskytla dostatečný výkon pro práci s časovými řadami.

3 Porovnání databází

Cílem aplikace není jen měření shromažďovat, ale také řídit k nim přístup a umožnit s nimi efektivně pracovat. Při výběru vhodné databáze je tak třeba brát v úvahu ne jen její použití pro ukládání časových řad, ale také jednoduchost propojení s ostatními funkcemi. Tato kapitola se proto zabývá výběrem databáze, která se pro použití v aplikaci hodí nejvíce a hodnotí, které funkce jdou implementovat přímo v jednotlivých databázích, případně kolik dalšího zpracování v serverové aplikaci by vyžadovaly.

3.1 Možnosti implementace funkcí

3.1.1 Správa uživatelů a uživatelských práv

Jednou z hlavních funkcí aplikace je řízení přístupu uživatelů k měřením z různých zařízení. To se provádí pomocí uživatelských skupin a skupin zařízení. Každý uživatel může být v několika uživatelských skupinách, z nichž každá má definovaná práva jako zápis a čtení pro určitou skupinu zařízení.

Není tak možné uživatelům povolit přístup přímo do databáze a všechny požadavky na data musí jít skrze serverovou aplikaci. Stejně tak není možné použít dostupná uživatelská rozhraní pro zobrazování dat v databázi, pokud není možné i jejich požadavky zpracovávat v aplikaci. Není také možné využít již hotové kompletní řešení, jako je například TICK stack, neboť takovéto řízení uživatelů neumožňuje.

Vzhledem k tomu, že NoSQL databáze časových řad nedokáží ukládat obecná data, bylo by třeba mít při jejich použití pro informace o uživatelích a skupinách v aplikaci ještě jednu databázi. SQL databáze v tomto ohledu tak mají převahu, protože do nich jde uložit libovolný typ dat.

3.1.2 Tagování dat a vyhledávání

Vyhledávání mezi daty je v aplikaci implementováno pomocí tagů. Tag je definován svým jménem a má k sobě přiřazenou jednu či více veličin nebo zařízení. Tagy se také dají kombinovat pomocí konjunkce a disjunkce.

Tento systém je nekompatibilní s NoSQL databázemi, které tagy ukládají jako dvojici *klíč-hodnota*. Pro vyhledávání podle tagů je tak třeba znát obě části dvojice a nelze vyhledávat pouze pomocí jeho názvu. Dalším problémem je, že data musí být otagovaná již při jejich nahrání do databáze a měnit je poté lze jen složitě.

3.1.3 Nasazení na ARM zařízeních

IoT a průmyslová embedded zařízení jsou často založená na architektuře ARM. Je proto nutné, aby i databáze a serverová aplikace podporovaly tuto architekturu.

Všechny tři databáze jsou open-source a je tak možné je zkompilovat pro libovolnou architekturu. Všechny také nabízejí připravené balíčky pro některé Linuxové distribuce.

3.1.4 Agregace dat

Při shromažďování velkého množství dat je třeba uvažovat, jak je zpracovávat a ukládat. Kromě komprese a mazání starých dat je další možnost jejich agregace. Při agregaci dochází ke spojení více hodnot v pravidelném časovém intervalu do jedné a tím na úkor rozlišení k zmenšení jejich velikosti.

Specifikem práce s měřeními elektrické energie je nutnost používat pro agregaci různé funkce, kromě aritmetického průměru například i průměr kvadratický. Je proto důležité, aby kromě samotné možnosti agregace dat nabízela databáze i potřebné funkce.

InfluxDB všechny takové funkce poskytuje, TimescaleDB dokonce umožňuje definovat vlastní. OpenTSDB však nabízí pouze agregaci pomocí aritmetického průměru nebo součtu.

3.1.5 Přepočty hodnot

U některých měřených hodnot je třeba pro získání dalších informací provést přepočet na jinou jednotku. Například při měření výkonu solárních panelů je měřená hodnota v kWh. Ta ale sama o sobě nevypovídá nic o jejich efektivitě, je proto nutné znát i

plochu, kterou panel pokrývá a vypočítat z ní hodnotu v kWh/m^2. Tuto plochu je tedy potřeba v databázi uložit vedle ostatních informací o měřené veličině a umožnit získávání jak původní naměřené hodnoty, tak i té přepočítané.

TimescaleDB umožňuje používat všechny funkce relační databáze a není tak problém spojit měření s dalšími hodnotami a provádět nad nimi matematické operace. Oproti tomu NoSQL databáze mohou pouze tyto hodnoty uložit jako tagy a vrátit je společně s měřeními, samotný přepočet pak tedy musí proběhnout v serverové aplikaci.

3.1.6 Další funkce

Kromě funkcí, které jsou nutné pro správnou funkčnost API, je dobré zhodnotit i všechny méně důležité funkce, které ale budou ovlivňovat výslednou implementaci. Jsou to například datové typy, konkrétně možnost rozlišovat typy `float` a `double`. To NoSQL databáze InfluxDB neumí, což je pro toto využití značnou nevýhodou.

Další je způsob komunikace mezi aplikací a databází. Díky tomu, že TimescaleDB je rozšíření PostgreSQL, je možné pro komunikaci s ní využít všechny dostupné knihovny a nástroje pro práci s SQL databázemi. InfluxDB pak knihovny pro práci s databází nabízí vlastní. OpenTSDB ale zpřístupňuje pouze HTTP API, jejíž použití je složitější než použití nativní knihovny.

3.2 Výběr databáze

Vzhledem k tomu, že původní API je navržené pro práci s SQL databází, některé jeho funkce by se na time series NoSQL databáze převáděly obtížně, nebo dokonce vyžadovaly pro některá data druhou, obecnější databázi. V tabulce 3.1 jsou porovnány možnosti implementace jednotlivých funkcí. Informace neurčují, zda je nebo není možné dané funkce implementovat, ale spíše jestli jdou vytvořit přímo na úrovni databáze, nebo vyžadují jen malé množství dalšího zpracování v aplikaci.

Jednou z hlavních výhod relační databáze TimescaleDB je možnost napojovat na měření libovolný počet a typ dalších dat. Použití druhé databáze pro data jako uživatelská práva a informace o měřených hodnotách by výrazně zvyšovalo složitost celé aplikace.

Tabulka 3.1: Porovnání databází

Funkce	InfluxDB	TimescaleDB	OpenTSDB
Data o veličinách a zařízeních	✓	✓	✓
Řízení přístupu uživatelů	×	✓	×
Podpora architektury ARM	✓	✓	✓
Agregace dat	✓	✓	×[1]
Přepočty hodnot	×	✓	×

[1] Umožňuje agregaci, ale neposkytuje všechny potřebné funkce

Další výhodou je pak celkově větší flexibilita SQL databází. NoSQL databáze časových řad jsou vhodnější v případě, že je funkcí systému pouze sbírat data a umožnit s nimi pracovat. Nutnost definovat pevné schéma je v tom případě spíše na obtíž. Jakmile je však třeba přidávat další funkcionalitu, dostupnost různých nástrojů a možnosti rozšíření SQL databází zjednodušují výslednou implementaci.

4 Nastavení databáze

Databáze PostgreSQL nemá defaultně optimální nastavení pro práci s časovými řadami [5]. Je proto třeba změnit její nastavení v souboru `postgresql.conf`. Jde zejména o hodnoty pracovních pamětí a bufferů, ale také o nastavení procesů běžících na pozadí a zámků tabulek. TimescaleDB k tomuto účelu obsahuje nástroj `timescaledb-tune`, který tyto hodnoty automaticky nastaví podle hardwaru, na kterém je databáze spouštěna. V případě, že databáze a aplikace běží na různých strojích, je také potřeba databázi nastavit pro vzdálené připojení v souborech `postgresql.conf` a `pg_hba.conf`

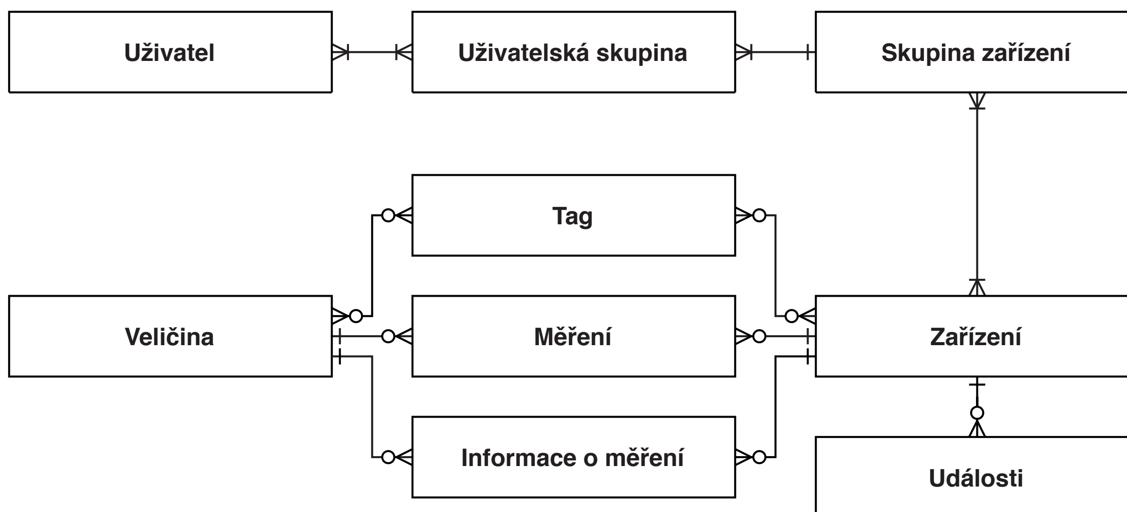
Před použitím se rozšíření TimescaleDB musí přidat do databáze, po jeho instalaci se to provádí příkazem `CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;`.

4.1 Schéma databáze

Návrh databáze se řídí potřebami zadaného API, konceptuální schéma je zobrazeno diagramem na obrázku 4.1. Ten ukazuje, jak jsou řešena uživatelská práva pomocí skupin. Je v něm také znázorněno ukládání měření, kde každá časová řada je definována dvojicí zařízení a veličiny a také k ní příslušící informace a tagy.

Cílem při návrhu však bylo využít co nejvíce možností RDBMS a TimescaleDB a přesunout tak funkcionalitu aplikace přímo do databáze. Pomocí vhodných triggerů a procedur je tak možné většinu akcí provádět jediným požadavkem ze serverové aplikace a tím zjednodušit její logiku.

Ve zdrojovém kódu 4.1 je ukázka databázové procedury, která ověřuje, zda má uživatel přístup k požadované skupině zařízení, a její použití v dotazu. Pokud uživatel potřebná práva nemá, databáze odešle jako odpověď chybu a dotaz se neprovede. Pokud ano, procedura vrátí do dotazu požadované ID a ten se tak může provést.



Obrázek 4.1: Konceptuální schéma databáze.

```
01 | CREATE FUNCTION can_read_device_group(dgid INT, uid INT)
02 | RETURNS INT AS $func$
03 | BEGIN
04 |   IF dgid NOT IN (
05 |     SELECT dg.id FROM device_groups dg
06 |     INNER JOIN user_groups ug
07 |       ON dg.id = ug.device_group_id
08 |     INNER JOIN user_relations ur
09 |       ON ug.id = ur.user_group_id
10 |     WHERE ug.read_data = TRUE AND ur.user_id = uid) THEN
11 |         RAISE 'Unauthorized'
12 |           USING ERRCODE = 'insufficient_privilege';
13 |   END IF;
14 |   RETURN dgid;
15 | END
16 | $func$ LANGUAGE plpgsql;
17 |
18 | SELECT * FROM device_groups WHERE id = x AND
19 |     id = can_read_device_group(x, user_id);
```

Zdrojový kód 4.1: Kontrola uživatelských práv

Díky kontrole uživatelských práv pomocí takovýchto procedur je možné v aplikaci zpracovávat pouze dva různé výsledky. Chybu, kterou jde řešit pro všechny dotazy hromadně, nebo úspěšné přijetí dat a jejich následné odeslání uživateli. Oproti tomu kontrola práv pomocí samostatného dotazu přidává aplikaci na složitosti, protože je potřeba nejdříve čekat na první výsledek a až poté provést požadavek na data, o které uživatel žádal.

Další možnost zjednodušení celé aplikace je použití triggerů pro automatické tagování dat. Jakmile je tak do aplikace vloženo nové zařízení či veličina, automaticky se k nim i přiřadí tagy. Ve zdrojovém kódu 4.2 je ukázka konkrétní implementace triggeru pro tagování nově vložených veličin. Kód využívá možností jazyka PL/pgSQL pro kombinování DML příkazů s podmínkami a cykly a umožňuje tak v jediné proceduře nejen přiřadit k nové proměnné tagy, ale zároveň také tyto tagy vytvořit pokud ještě neexistují. Takovýto trigger tak nahrazuje dokonce až 3 různé dotazy, které by jinak aplikace musela vykonat.

```
01 | CREATE OR REPLACE FUNCTION variable_tags()
02 |   RETURNS trigger AS $variable_tags$
03 | DECLARE
04 |   tag_name TEXT;
05 |   tag_id INT;
06 | BEGIN
07 |   FOR tag_name IN VALUES (NEW.variable_group),
08 |       (NEW.variable_subgroup), (NEW.short_name),
09 |       (NEW.variable_group || '_' ||
10 |        NEW.variable_subgroup || '_' || NEW.short_name)
11 |   LOOP
12 |     SELECT id INTO tag_id FROM tags WHERE name = tag_name;
13 |     IF (tag_id IS NULL) THEN
14 |       INSERT INTO tags(name) VALUES (tag_name)
15 |         RETURNING id INTO tag_id;
16 |     END IF;
17 |     INSERT INTO tag_relations(tag_id, variable_id)
18 |       VALUES (tag_id, NEW.id);
19 |   END LOOP;
20 |
21 |   RETURN NEW;
22 | END;
23 | $variable_tags$ LANGUAGE plpgsql;
24 |
25 | CREATE TRIGGER variable_tags
26 |   AFTER INSERT ON variables
27 |   FOR EACH ROW EXECUTE PROCEDURE variable_tags();
```

Zdrojový kód 4.2: Trigger pro automatické tagování dat

Další triggery se používají pro zabránění vzniku *sirotků*. Například zařízení musí být vždy alespoň v jedné skupině. Nemůže tak vzniknout situace, kdy by k němu nikdo neměl přístup. Kromě triggerů a procedur obsahuje databáze také velké množství integritních omezení. Ty zajišťují například aby nevznikaly duplikáty přiřazení uživatelů a zařízení ke skupinám, kvůli kterým by mohlo docházet ke zmatení uživatele.

4.2 Tabulky pro měření

Pro ukládání časových řad se používají dva různé typy tabulek, *úzké* a *široké* [9]. Při použití úzkých tabulek se pro každý časový bod ukládá pouze jedna naměřená hodnota, zatímco do širokých je možné ukládat hodnot více. Úzké tabulky tak nabízejí větší flexibilitu, naopak široké na úkor flexibility efektivněji využívají úložiště.

Vzhledem k tomu, že do databáze budou zapisovat data různé typy zařízení, z nichž každý měří různou kombinaci veličin, není možné vytvořit takové schéma tabulky, aby všechny hodnoty měřené jedním zařízením byly v jednom řádku a zároveň nevznikalo v databázi velké množství `NULL` hodnot. Druhá možnost vytvoření široké tabulky je ukládat do jednoho řádku více hodnot jedné veličiny v určitém časovém intervalu, například vytvořit nový řádek každých 5 sekund a do něj ukládat 5 hodnot, jednu každou sekundu. Nevýhodou tohoto přístupu je složitější zápis dat a nutnost dodržovat tyto časové intervaly. Není tak například možné některé veličiny měřit s menším časovým rozlišením nebo toto rozlišení zvýšit v případě neobvyklé události. Právě zaznamenávání událostí je jednou z hlavních funkcí aplikace a bez kompromisů se tak pro ukládání měření hodí pouze úzká tabulka.

V zájmu šetření místa se také pro ID zařízení a veličin používá pouze 2 bytový typ `int2`. Obě tyto ID jsou totiž kvůli návrhu tabulek uloženy s každým měřením a určují ke které časové řadě patří. Je také nepravděpodobné, že by jich bylo potřeba více než 32 767, což je limit tohoto datového typu.

4.2.1 Agregace dat

Tabulky je dále třeba uzpůsobit pro agregaci dat. TimescaleDB má pro automatickou agregaci speciální funkce. Největším omezením této funkcionality je při agregaci zákaz napojování k měřením další tabulky. Při používání různých agregačních funkcí je tak třeba informaci o tom, kterou z nich použít, uložit buďto přímo v tabulce s měřeními, nebo měření rozdělit podle funkcí do různých tabulek. Ukládání této informace přímo s každým měřením by znatelně zvýšilo využití úložiště a vytvoření několika tabulek je tak lepší řešení. Pro 3 různé agregační funkce jsou tedy vytvořeny 3 tabulky, `data_avg`, `data_rms` a `data_last`, pro aritmetický průměr, kvadratický průměr a poslední hodnotu v daném intervalu. Na tyto tabulky je poté použita funkce `create_hypertable`, která je označí jako tabulky s časovými řadami a nastaví pro ně optimalizace jako dělení na chunky.

Ve zdrojovém kódu 4.3 je vytvořen pohled `data_avg_1m`, ve kterém se agregují hodnoty tabulky `data_avg` pomocí aritmetického průměru s rozlišením 1 minuty. Obdobně se agregují i hodnoty tabulek `data_rms` a `data_last` s rozlišením 1 minuta a 1 hodina. Takto vytvořené pohledy s nastavením `timescaledb.continuous` si uchovávají svoje data i po smazání zdrojových dat.

```
01 | CREATE VIEW data_avg_1m
02 | WITH (timescaledb.continuous) AS
03 | SELECT
04 |   time_bucket('1 minute', time) AS bucket,
05 |   device_id,
06 |   variable_id,
07 |   AVG(value)::float4 AS value
08 | FROM data_avg
09 | GROUP BY bucket, device_id, variable_id;
10 |
11 | ALTER TABLE data_avg_1m
12 | RENAME bucket TO time;
```

Zdrojový kód 4.3: Automatická agregace dat

Výsledek funkce `time_bucket`, použité k určení časového intervalu pro agregaci, v tomto případě nelze pojmenovat `time`, protože poté dochází k nejasnosti v `GROUP BY` části dotazu. Je proto nutné použít pro něj dočasné jméno a celý sloupec poté přejmenovat, aby byly názvy sloupců stejné v tabulkách i pohledech. Tím je umožněno používat stejné dotazy na původní i agregovaná data.

Výsledky agregačních funkcí nad čísly s plovoucí řádovou čárkou mají vždy dvojnásobnou přesnost (typ `double`). V případě, že taková přesnost není potřeba, je vhodné výsledky pro úsporu místa přetypovat na jednoduchou přesnost a tím snížit jejich velikost na polovinu.

Rozdělení tabulek a pohledů podle agregačních funkcí zesložiťuje práci s nimi. Ve zdrojovém kódu 4.4 je proto vytvořen pohled, díky kterému se se všemi měřeními dá pracovat, jako by byly ve stejné tabulce. Z pohledu uživatele je tak celý systém agregace schován za tento pohled a není důvod jakkoliv pracovat s rozdělenými tabulkami. Stejně je potřeba ošetřit také pohledy vytvořené agregací.

Zápis dat do správné tabulky je v tomto případě potřeba řídit triggerem zobrazeným ve zdrojovém kódu 4.5. Ten načte informace o vkládané proměnné a podle nich jednotlivá měření rozdělí.

```
01 | CREATE VIEW data AS
02 |     SELECT * FROM data_avg
03 |     UNION ALL
04 |     SELECT * FROM data_rms
05 |     UNION ALL
06 |     SELECT * FROM data_last;
```

Zdrojový kód 4.4: Pohled nad daty

```
01 | CREATE OR REPLACE FUNCTION insert_data()
02 |   RETURNS TRIGGER AS $insert_data$
03 | DECLARE
04 |   fn TEXT;
05 | BEGIN
06 |   SELECT subgroup INTO fn FROM variables
07 |     WHERE variables.id = NEW.variable_id;
08 |   CASE
09 |     WHEN fn = 'avg' THEN
10 |       INSERT INTO data_avg VALUES(NEW.time,
11 |         NEW.device_id, NEW.variable_id, NEW.value);
12 |     WHEN fn = 'rms' THEN
13 |       INSERT INTO data_rms VALUES(NEW.time,
14 |         NEW.device_id, NEW.variable_id, NEW.value);
15 |     WHEN fn = 'last' THEN
16 |       INSERT INTO data_last VALUES(NEW.time,
17 |         NEW.device_id, NEW.variable_id, NEW.value);
18 |   END CASE;
19 |   RETURN NEW;
20 | END;
21 | $insert_data$ LANGUAGE plpgsql;
22 |
23 | CREATE TRIGGER insert_data INSTEAD OF INSERT ON data
24 |   FOR EACH ROW EXECUTE PROCEDURE insert_data();
```

Zdrojový kód 4.5: Trigger pro vkládání dat

Navržené schéma databáze používá pro ukládání času měření typ `timestamp`. Ten umožňuje čas ukládat s až mikrosekundovou přesností. Časový vstup ve formátu unixového času tak může mít až 6 desetinných míst, narozdíl od celočíselného typu `integer`. V případě, že by však taková časová přesnost nebyla potřeba, je možné použít například třicetidvou bitové celé číslo, narozdíl od 64 bitového typu `timestamp`, a tím ušetřit místo.

4.3 Zpracovávání dat

Dělení tabulek s časovými řadami na chunky se provádí automaticky po uplynutí určitého časového intervalu. Pro optimální fungování databáze je nutné tento interval nastavit tak, aby aktivní chunky zabíraly dohromady maximálně 25% operační paměti [5]. Záleží tedy jak na operační paměti stroje, tak na množství dat, která se ukládají.

Tento interval se nastavuje funkcí `set_chunk_time_interval`, např. `set_chunk_time_interval('data_avg', interval '24 hours');` pro vytvoření nového chunku tabulky `data_avg` každých 24 hodin.

Pro snížení objemu dat umožňuje TimescaleDB kromě agregace také automaticky stará data komprimovat. Při kompresi se jednotlivé řádky tabulky spojují do jednoho, ve kterém jsou poté všechny hodnoty uloženy a vzniká tak široká tabulka. V době komprese jsou již známé všechny časové hodnoty a není tak třeba mít definované pevné schéma, ani časové intervaly mezi měřeními.

Nevýhodou tohoto řešení je, že zkomprimovaná data již nelze měnit ani k nim přidávat další. Je proto důležité zvolit dostatečný časový odstup, aby nedocházelo ke kompresi časového intervalu, ve kterém ještě nejsou nahrána všechna data. Komprese se provádí vždy na celé chunky, nastavení odstupu se tak definuje pomocí stáří chunků.

Ve zdrojovém kódu 4.6 je ukázka nastavení komprese chunků tabulky `data_avg` starších než 7 dní. Možnost `compress_segmentby` definuje kombinace sloupců, podle kterých se budou hodnoty rozdělovat do řádků. Ve vytvořených tabulkách jsou jednotlivé časové řady rozlišeny dvojicí zařízení a veličiny, i při kompresi proto pro každou takovou dvojici bude vznikat nový řádek.

```
01 | ALTER TABLE data_avg SET (
02 |  timescaledb.compress,
03 |  timescaledb.compress_orderby = 'time DESC',
04 |  timescaledb.compress_segmentby = 'device_id, variable_id'
05 | );
06 |
07 | SELECT add_compress_chunks_policy(
08 |     'data_avg', interval '7 days');
```

Zdrojový kód 4.6: Nastavení komprese starých dat

I přes všechny funkce jako agregace a komprese dat však není možné ukládat měření nekonečně a je tak nutné vyřešit i kdy mazat stará data. Stejně jako ostatní funkce se i mazání v TimescaleDB provádí po chuncích. Je tak daleko rychlejší než mazání každého řádku samostatně pomocí `DELETE` příkazů ve standardních SQL databázích. Mazání starých chunků se provádí funkcí `drop_chunks` se třemi parametry. První dva jsou interval, jak staré chunky se mají smazat a název tabulky, ze které se mají smazat. Poslední je `cascade_to_materialization`, který určuje, zda smazat jen zdrojová data, nebo i všechna agregovaná ze zadané tabulky. Je tak možné používat pro agregovaná měření jiné intervaly mazání, nastavením parametru na `FALSE` při mazání zdrojových dat a poté zadáváním názvu agregačního pohledu místo původní tabulky pro smazání agregovaných dat.

5 Serverová aplikace

Ukázková serverová aplikace je naprogramovaná v jazyce JavaScript pomocí frameworku Express. Prostředí Node.js je multiplatformní a umožňuje tak spouštět stejný kód na různých architekturách. JavaScript má pro psaní serverových aplikací několik výhodných vlastností. Jednou z nich je práce s daty ve formátu JSON. Při jejich čtení z požadavku a odesílání odpovědi je tak není třeba nijak parsovat a to snižuje složitost aplikace. Další je syntaxe `async/await`, která umožňuje jednoduchou práci s asynchronními funkcemi, jako je například komunikace s databází.

Zdrojový kód umožňuje spouštět aplikaci a databázi zvlášť na různých strojích nebo v kontejnerech. Aplikace proto obsahuje pro nastavení svých parametrů a připojení k databázi konfigurační soubor popsaný v příloze A.

5.1 Komunikace s databází

Díky tomu, že TimescaleDB je rozšíření PostgreSQL, umožňuje použítí všech již existujících nástrojů pro práci s SQL databázemi. Aplikace tak pro komunikaci s databází využívá SQL builder `Knex.js`. Použití SQL builderu je kompromis mezi samostatnými SQL dotazy a ORM knihovnami. Použití ORM přidává další úroveň abstrakce mezi aplikaci a databázi a neumožňuje tak použití některých jejich funkcí, jako jsou například procedury. Oproti tomu práce s čistě textovými SQL dotazy je obtížná a v případě operací jako `UPDATE` vyžaduje jejich složité generování. SQL builder umožňuje pomocí svých metod vytvářet libovolné dotazy ve formátu podobném SQL, včetně poddotazů a procedur.

Využití SQL builderu pro generování parametrizovatelného dotazu pro získávání měření z databáze je zobrazeno ve zdrojovém kódu 5.1. Ten obsahuje například výběr, z jaké tabulky (pohledu) se má číst, použití databázových procedur, nebo poddotazy. Pomocí parametrů požadavku je tak možné v jedné metodě vytvářet dotazy s různými podmínkami a výstupy.

```
01 |  let query = db.queryBuilder()
02 |
03 |  if(req.timeStart) query.where('time', '>=',
04 |    db.functions.toTimestamp(req.timeStart))
05 |  if(req.timeEnd) query.where('time', '<=',
06 |    db.functions.toTimestamp(req.timeEnd))
07 |
08 |  if(req.eventID)
09 |    query.with('evnt', db.getEvent(req.eventID, req.userID))
10 |      .where('time', '>=',
11 |        db.queryBuilderOn('evnt').select('time'))
12 |      .where('time', '<=',
13 |        db.queryBuilderOn('evnt')
14 |          .select("time + duration * interval '1 second'"))
15 |      .where('device_id',
16 |        db.queryBuilderOn('evnt').select('device_id'))
17 |
18 |  if(req.andTags)
19 |    query.with('tags', db.functions.andTags(req.andTags))
20 |  if(req.orTags)
21 |    query.with('tags', db.functions.orTags(req.orTags))
22 |  if(req.andTags || req.orTags)
23 |    query.where((q) =>
24 |       q.whereNotExists(
25 |         db.queryBuilderOn('tags')
26 |           .select('variable_id')
27 |           .whereNotNull('variable_id'))
28 |       .orWhereIn('variable_id',
29 |           db.queryBuilderOn('tags').select('variable_id')))
30 |     .where((q) =>
31 |       q.whereNotExists(
32 |          db.queryBuilderOn('tags')
33 |            .select('device_id').whereNotNull('device_id'))
34 |       .orWhereIn('device_id',
35 |           db.queryBuilderOn('tags').select('device_id')))
36 |
37 |  query.column('time', 'device_id', 'variable_id')
38 |  if(!req.withoutData) query.column('value')
39 |
40 |  if(req.aggregate1m) query.from('data_1m')
41 |  else if(req.aggregate1h) query.from('data_1h')
42 |  else query.from('data')
43 |
44 |  query.where('device_id', 'in',
45 |    db.functions.readableDevices(req.userID))
46 |
47 |  const rows = await query
```

Zdrojový kód 5.1: Generování SQL dotazů

Díky takto všestrannému dotazu lze použít pro většinu práce s měřeními jediný endpoint a tím také zjednodušit práci s API. Data z databáze je možné přijmout jako JSON objekty, nebo jako pole hodnot. Díky tomu, že JavaScript pracuje nativně s objekty ve formátu JSON, je možné s přijatými daty ihned pracovat jako s objekty i bez použití ORM. Vzhledem k tomu, že všechna práce s daty ale probíhá již v databázi, není žádné jejich zpracování třeba a je možné ihned je odeslat.

5.2 API endpointy

Framework Express obsahuje pro zjednodušení implementace API endpointů tzv. *middleware* funkce. Ty umožňují před vykonáním samotného endpointu předzpracovat požadavek, například ověřit příhlášení uživatele, a případně vykonávání ukončit a odeslat chybu. Takovou funkci je pak možné použít hromadně pro všechny endpointy a není potřeba jí volat v každém zvlášť. Například v ukázce endpointu s metodou `PUT` 5.2 je to funkce `isLoggedIn`. Ta přečte z HTTP hlaviček token uživatele a podle něj nastaví pro požadavek proměnnou `userID`. Díky té může následně databáze určit, zda má uživatel právo měnit požadované zařízení.

```
01 | dvc.put('/:id', auth.isLoggedIn, ah(async (req, res) => {
02 |   await db.editDevice(req.params.id, req.body, req.userID)
03 |   return res.status(200).send()
04 | }))
```

Zdrojový kód 5.2: Implementace API endpointu

Aplikace používá pro ověření uživatelů JSON Web Token (JWT). JWT v sobě umožňuje uložit JSON objekt s informacemi a poté ověřovat jeho pravost. Aplikace tak v tokenech ukládá ID uživatelů, které následně používá v endpointech pro řízení přístupu.

Stejně jako middleware funkce je pro všechny endpointy možné použít také hromadné řešení chyb a výjimek. Aplikace toho využívá pro další zjednodušení implementace endpointů, které tak nemusí obsahovat žádné ošetření vstupu ani chyb. Knihovna Knex.js se stará o zamezení útoků jako SQL injection. Je proto možné databázi poslat přímo data z požadavku a přenechat jejich kontrolu na ní. Databáze se pak tyto data pokusí zpracovat a v případě, že se v nich bude vyskytovat nějaký problém, odešle zpět popis chyby. Tyto databází vygenerované chyby mají pevně danou strukturu a je tak možné vytvořit funkci, která bude tyto chyby umět zpracovat a naformátovat odpověď na požadavek. Funkce použítá jako error handler pro všechny endpointy je implementovaná v modulu `err.js`.

API tak kromě typu chyby, pokud je to možné, obsahuje i název sloupce a hodnotu, kvůli které chyba nastala. Ukázka takového objektu je ve zdrojovém kódu 5.3, po pokusu vložit do databáze zařízení s již existujícím seriovým číslem. Díky těmto informacím je pak možné v uživatelském rozhraní například zvýraznit, který vstup byl chybný. Je také jednodušší vytvořit lokalizaci těchto chyb a do lokalizovaného textu pak jen vložit přijaté hodnoty.

```
01 |  {
02 |      "type": "Unique violation",
03 |      "text": "Unique constraint violation -
04 |          Key (serial_number)=(1234) already exists.",
05 |      "table": "devices",
06 |      "column": "serial_number",
07 |      "value": "1234"
08 |  }
```

Zdrojový kód 5.3: Ukázka chybového objektu

Aby takovýto systém mohl fungovat, musí mít databáze pro všechny tabulky přesně nastavená integritní omezení, která tak nahradí podmínky v jednotlivých endpointech API. Je důležité zejména aby uživatel nemohl měnit ID záznamů, ale také které hodnoty musí být `UNIQUE` a které jsou povinné, či další doménová omezení.

6 Možnosti rozšíření

6.1 Rozšíření databáze

Vytvořená databáze, popsaná v části 4.2 *„Tabulky pro měření“*, používá pro ukládání měření 3 různé tabulky, podle funkcí, kterými se mají agregovat. Tento systém je možné dále rozvést a tabulky tak rozdělit i podle datového typu, který ukládaná měření mají, nebo podle jejich důležitosti. Je tak možné rozlišit například měření s jednoduchou a dvojitou přesností, čímž se velikost části z nich zmenší na polovinu. Dále je také možné řídit, která měření se smažou dříve a která je naopak v databázi nutné ponechat delší dobu.

TimescaleDB ve verzi Community nedovoluje automaticky mazat stará data. Toto omezení je však možné obejít použitím časovačů v aplikaci nebo přímo v systému. Ty pak mohou posílat v pravidelných intervalech `drop_chunks` požadavky a tím tuto funkcionalitu nahradit.

Díky tomu, že databáze PostgreSQL umožňuje nativně ukládat data ve formátu JSON a pracovat s nimi, je možné zjednodušit návrh některých tabulek. Zejména při ukládání informací o událostech, kdy každý typ události může mít několik různých důležitých informací, je vhodné tyto informace uložit jako JSON objekt a předejít tím vytváření pevné struktury tabulky. I mezi takto uloženými daty je možné vyhledávat a tím prakticky pro jeden sloupec tabulky napodobit NoSQL dokumentovou databázi, jako je například MongoDB.

6.2 Rozšíření aplikace

Způsob, kterým aplikace řídí přístup k datům, nedovoluje uživatelům psát vlastní SQL dotazy. Místo toho tedy musí být jakákoliv práce s daty zprostředkována pomocí předpřipravených dotazů s nastavitelnými parametry. Pro přehlednost ukázky neobsahují omezení časového intervalu ani kontrolu uživatelských práv.

Jednou z hlavních částí aplikace je ukládání informací o neobvyklých událostech. Kromě informací o nich ale ukládá také všechna měření z doby jejich trvání. Zdrojový kód 6.1 obsahuje dotaz, který umožňuje podle informací o události vyhledat všechna data, která v jejím průběhu zařízení zaznamenalo. Tento dotaz je možné vidět také ve zdrojovém kódu 5.1, kde je již použitý k vyhledávání.

```
01 | WITH event AS ( SELECT * FROM events WHERE id = x )
02 | SELECT time, variable_id, value FROM data
03 | WHERE device_id = (SELECT device_id FROM event) AND
04 |   time >= (SELECT time FROM event) AND
05 |   time <= (SELECT time FROM event) +
06 |       (SELECT duration * interval '1 second' FROM event);
```

Zdrojový kód 6.1: Naměřené hodnoty události

Ve zdrojovém kódu 6.2 je ukázka přepočítávání hodnot. Tato funkcionalita je více popsaná v části 3.1.5. Každá časová řada je definována dvojicí veličiny a zařízení. Proto i dodatečné informace o ní obsahují tuto dvojici a umožňují tak jednoduché spojení obou tabulek a použití jejich hodnot. Je možné také například vynechat omezení veličiny a přepočítat tak všechny hodnoty zařízení, pro které je to možné.

```
01 | SELECT time, data.value/md.area AS value
02 | FROM data
03 | INNER JOIN measurement_data md ON
04 |     data.variable_id = md.variable_id AND
05 |     data.device_id = md.device_id
06 | WHERE md.area IS NOT NULL AND
07 |     data.variable_id = x AND data.device_id = y;
```

Zdrojový kód 6.2: Přepočet naměřené hodnoty

6.2.1 Implementace API

Dalším vhodným rozšířením API je použití protokolu WebSocket místo HTTP pro příjem měření. Existující implementace endpointu musí pro každý požadavek ověřit, zda má účet povoleno zapisovat data do požadovaného zařízení. Při použití socketů by stačilo toto ověření provést pouze jednou a poté už jen využívat otevřené spojení. Snížily by se tím jak nároky na výpočetní výkon serveru, tak i objem přenesených dat, protože by nebylo třeba posílat s každým měřením také všechny HTTP hlavičky.

Pro framework Express jsou dostupné balíčky, které umožňují použití protokolu WebSocket společně s HTTP endpointy. Není tak problém je jednoduše zakomponovat do vytvořené aplikace a používat je současně.

Další rozšíření je použítí streamování pro odesílání dat. Při odesílání HTTP odpovědí se musí nejdříve všechna odesílaná data nahrát do operační paměti serveru a poté jsou odeslána najednou. Pokud by tak uživatel požádal o velké množství dat, například požadavkem bez omezení časového intervalu, mohla by serveru při jejich načítání dojít paměť. S použitím streamování je možné data odesílat po částech a tomuto tak předejít. Další výhodou streamování je, že pokud i klient dokáže příjmout data jako stream, může je zpracovávat postupně a nemusí tak čekat, než bude jejich odesílání dokončené.

Node.js obsahuje nativní rozhraní pro práci se streamy dat. Je tak možné využívat je pro čtení, zápis i transformování. Kromě HTTP požadavků a odpovědí je možné streamy používat například i pro práci se soubory [10]. Streamování se však hodí pouze pro velké množství dat, pro většinu dotazů je vhodnější i rychlejší použít normální odesílání.

Ve zdrojovém kódu 6.3 je ukázka obou řešení, na prvních dvou řádcích odeslání celé odpovědi najednou, na posledních jejich postupné streamování. Z SQL dotazu je nejprve vytvořen stream, který pomocí kurzoru postupně čte data z databáze. Před jejich odesláním je nutné je nejdříve převést z formátu JSON na text pomocí knihovny `JSONStream`. Díky implementaci HTTP protokolu v Node.js, kde požadavek i odpověd jsou automaticky streamy, je pak možné data ihned odesílat.

```
01 |  const rows = await query
02 |  return res.status(200).send(rows)
03 |
04 |  const stream = query.stream().on('error', e => next(e))
05 |  stream.pipe(JSONStream.stringify()).pipe(res)
```

Zdrojový kód 6.3: Streamování odpovědi

7 Závěr

Použití relační databáze PostgreSQL s rozšířením TimescaleDB pro časové řady umožnilo jednoduché nahrazení původní SQL databáze a tím i zachování již existujícího API. V aplikaci, která kromě uchovávání time series dat zároveň i řídí přístup uživatelů, mohou být NoSQL databáze časových řad pouze jednou ze součástí celku, zatímco relační databáze dokáže poskytnout většinu potřebné funkcionality. Rozšíření TimescaleDB k tomuto přidává požadované optimalizace a další užitečné funkce pro práci s časovými řadami.

Hlavní překážkou pro použití NoSQL databází je jejich zaměření na ukládání pouze časových řad. Ukládání většího množství metadat je tak třeba uzpůsobit formátu *klíč-hodnota*, který dané databáze používají. I přesto však takto některé informace uložit nejdou, zejména protože mezi nimi není možné vytvářet $m{:}n$ relace. Alternativní řešení by bylo použít stejné schéma, jako to, zobrazené v obrázku 4.1 a NoSQL databází nahradit pouze tabulky s měřeními. V tom případě je však třeba pro ostatní data použít druhou databázi a udržovat obě databáze navzájem konzistentní pomocí logiky v aplikaci, na rozdíl od nástrojů relačních databází jako je např. referenční integrita.

Navržené schéma databáze využívá většinu funkcí poskytovaných rozšířením TimescaleDB. Automaticky tak provádí například agregaci dat nebo jejich následnou kompresi. Díky agregaci a kompresi je možné ukládat naměřené hodnoty samostatně a jednodušeji s nimi pracovat, na rozdíl od původního řešení, které hodnoty ukládalo měření do binárních blobů. Pomocí procedur a triggerů, umožněných plnou podporou jazyka SQL, také mohou v databázi probíhat další funkce aplikace, jako tagování dat či kontrola uživatelských oprávnění.

Vytvořený create script databáze však není možné použít beze změn. Obsahuje nastavení, například intervaly tvoření chunků a jejich komprese, které je nutné určit pro každé nasazení databáze zvlášť.

Díky tomu, že většinu funkcí bylo možné implementovat již v databázi, vytvořená serverová aplikace má na starost pouze několik málo odpovědností. Patří mezi ně zejména ověření uživatele a ochrana před útoky jako SQL injection. Jakmile jsou data ošetřená a bezpečná, aplikace je pro veškeré další zpracování předá databázi. Ta poté zkontroluje, zda má uživatel dostatečná práva na provedení požadované akce a zda jsou přijaté informace kompletní a validní.

Pro ověření funkčnosti byla aplikace i databáze nasazena na embedded ARM zařízení. Vytvořené REST API bylo poté otestované pomocí programu Postman. Vzhledem k objemu dat, které by databáze TimescaleDB a obecně databáze časových řad měly zvládnout zpracovat, nebylo možné v rozsahu této práce zjistit skutečné možnosti a limity tohoto řešení.

K významnému rozšíření API nedošlo zejména proto, že použití TimescaleDB umožnilo většinu požadovaných funkcí provádět automaticky. Došlo tak spíše k jejímu zjednodušení, například díky možnosti ukládání měření z událostí společně s normálními měřeními.

Použitá literatura

[1] Time series database (TSDB) explained. *InfluxData* [online]. InfluxData [cit. 17.2.2020]. Dostupné z: https://www.influxdata.com/time-series-database/.

[2] KIEFER, Rob. *TimescaleDB vs. PostgreSQL for time-series: 20x higher inserts, 2000x faster deletes, 1.2x-14,000x faster queries.* In: *Timescale Blog* [online]. 10.8.2017 [cit. 21.5.2020]. Dostupné z: https://blog.timescale.com/blog/timescaledb-vs-6a696248104e/.

[3] InfluxDB 1.8 documentation. *InfluxData Documentation* [online]. InfluxData [cit. 21.5.2020]. Dostupné z: https://www.docs.influxdata.com/influxdb/v1.8/.

[4] SIMMONS, David G. *How To: Building Flux Queries in Chronograf*. In: *Blog / InfluxData* [online]. 15.11.2018 [cit. 21.5.2020]. Dostupné z: https://www.influxdata.com/blog/how-to-building-flux-queries-in-chronograf/.

[5] TimescaleDB Documentation. *TimescaleDB Docs / Main* [online]. Timescale [cit. 21.5.2020]. Dostupné z: https://docs.timescale.com/latest/main.

[6] Timescale License. *Products / Timescale* [online]. Timescale [cit. 21.5.2020]. Dostupné z: https://www.timescale.com/legal/licenses.

[7] Documentation for OpenTSDB 2.3. *OpenTSDB 2.3 documentation* [online]. OpenTSDB [cit. 21.5.2020]. Dostupné z: http://www.opentsdb.net/docs/build/html/index.html.

[8] In-Memory Database Questions & Answers. *McObject* [online]. McObject [cit. 21.5.2020]. Dostupné z: https://www.mcobject.com/in_memory_database/.

[9] DUNNING, Ted a Ellen, FRIEDMAN. *Time Series Databases: New Ways to Store and Access Data*. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1-4919-1472-4.

[10] PARODY, Liz. *Understanding Streams in Node.js*. In: *The NodeSource Blog* [online]. 22.11.2019 [cit. 21.5.2020]. Dostupné z: https://www.nodesource.com/blog/understanding-streams-in-nodejs/.

A Nastavení a spuštění aplikace

Serverová aplikace vyžaduje runtime prostředí Node.js ve verzi alespoň v8. Před prvním spuštěním je třeba nainstalovat ve složce `server` příkazem `npm install` všechny potřebné moduly. Aplikace se poté spouští příkazem `npm start`.

Konfigurace aplikace se provádí v souboru `app/config.js`. Ten umožňuje měnit všechny potřebné parametry pro připojení k databázi a spuštění serveru. Nastavení hesla databáze a klíče pro šifrování JWT tokenů je možné také pomocí proměnných prostředí, nemusí tak být uloženy přímo ve zdrojovém kódu. Pokud jsou použity oba způsoby, prioritu má proměnná prostředí.

```
01 | module.exports = {
02 |   //Database connection configuration
03 |   "url": "localhost",
04 |   "user": "postgres",
05 |   "password": "",         //Env variable DBPASSWORD or this
06 |   "db_port": 5432,
07 |   "db_name": "db",
08 |
09 |   //Server configuration
10 |   "port": 80,
11 |   "jwt_secret": ""        //Env variable JWTSECRET or this
12 | }
```

Zdrojový kód A.1: Konfigurační soubor aplikace

B Přiložené soubory

`server`	Složka s vytvořenou serverovou aplikací
`postman.json`	Kolekce API endpointů programu Postman
`create.sql`	Create script vytvořené databáze