VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE ZPRACOVÁNÍ KLÍČOVÝCH INDIKÁTORŮ VÝKONU
OPTIMIZATION OF KPI PROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ONDŘEJ ŠULC**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JIŘÍ HYNEK, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

148486

Ústav: Ústav informačních systémů (UIFS)
Student: **Šulc Ondřej, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Kybernetická bezpečnost
Název: **Optimalizace zpracování klíčových indikátorů výkonu**
Kategorie: Databáze
Akademický rok: 2022/23

Zadání:

1. Prostudujte oblast internetu věcí (*Internet of Things*, IoT) se zaměřením na zpracování a vyhodnocování dat. Studujte principy klíčových indikátorů výkonu (*key performance indicator, KPI*).
2. Prozkoumejte existující relační databázové systémy, způsoby použití objektově relačního mapování (ORM) a dostupné knihovny určené pro tento účel. Rozhodněte, v čem jsou tyto knihovny klíčové a zda je vhodné je používat pro rychlé přístupy do databází. Knihovny porovnejte.
3. Analyzujte existující koncepty pro vyhodnocování KPI nad množinou dat ve firmě Logimic. Zaměřte se na rychlost vyhodnocování KPI a popište nedostatky.
4. Dle výsledků analýzy navrhněte nový způsob vyhodnocování KPI, případně navrhněte modifikaci stávajícího způsobu.
5. Navržené rozšíření a úpravy implementujte.
6. Proveďte testování funkčnosti a použitelnosti implementace přímo na platformě Logimic Smart City.

Literatura:

- Greengard, S. (2015). *The Internet of Things*. MIT Press.
- Parmenter, D. (2015). *Key performance indicators: developing, implementing, and using winning KPIs*. John Wiley & Sons.
- Eckerson, W. W. (2010). *Performance dashboards: measuring, monitoring, and managing your business*. John Wiley & Sons.
- Interní dokumentace firmy Logimic.

Při obhajobě semestrální části projektu je požadováno:
Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hynek Jiří, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 17.5.2023
Datum schválení: 25.10.2022

Abstrakt

Tato práce se zabývá optimalizací zpracování dat z IoT senzorů chytrých měst do podoby klíčových indikátorů výkonu (zkr. KPI). KPI jsou prostředkem pro monitorování velkého množství dat a vyjádření stavu výkonnostních faktorů, ovlivňujících prosperitu celého města. Zpracování dat do této podoby je výpočetně náročný proces, který se ale skládá z velkého množství vzájemně nezávislých výpočtů. Cílem této práce tedy bylo provést optimalizaci využitím paralelizace. Při paralelním zpracování lze výpočty rozdělit mezi více vláken a mohou tak být plně využívány všechny dostupné výpočetní prostředky (jádra CPU). Tento koncept byl prakticky implementován v projektu Smart City od firmy Logimic. Projekt je ovšem vybudován na platformě Node.js a při využití paralelizace dochází ke komplikacím s využíváním knihoven pro objektově relační mapování (zkr. ORM). Knihovny pro ORM na platformě Node.js nejsou vždy připraveny pro fungování v paralelním prostředí. Problém je v práci řešen vytvořením samostatné instance použité knihovny pro každé paralelní vlákno. Práce se zaměřuje na snížení režie s tím spojené a také na správné rozdělování práce mezi paralelní vlákna, aby docházelo k rovnoměrnému využití všech jader. Výsledky této práce dokazují, že optimalizace zpracování dat z IoT využitím paralelizace vede k významnému zrychlení, které odpovídá Amdahlovu zákonu, protože problémy s režií je možné snížit na zanedbatelné minimum.

Abstract

This thesis deals with the optimization of data processing from IoT sensors of smart cities into the form of key performance indicators (abbr. KPI). KPIs are a mean of monitoring a large amount of data and expressing the status of performance factors affecting the prosperity of the entire city. Data processing in this form is a computationally demanding process, but it consists of a large number of mutually independent calculations. Therefore the goal of this thesis was to perform optimization using parallelization. In parallel processing, calculations can be divided between multiple threads, enabling all available computing resources (CPU cores) to be fully used. This concept was practically implemented in the Smart City project of Logimic company. However, the project is built on the Node.js platform, and when using parallelization there are complications with the use of libraries for object-relational mapping (abbr. ORM). ORM libraries on the Node.js platform are not always ready to work in a parallel environment. This problem is solved by creating a separate instance of the used library for each parallel thread. The thesis focuses on reducing the overhead associated with this and also on the correct distribution of work between parallel threads so that all cores are used equally. The results of this work prove that optimizing IoT data processing using parallelization leads to a significant speedup that conforms to Amdahl's law, as overhead problems can be reduced to a negligible minimum.

Klíčová slova

klíčové indikátory výkonu, KPI, chytrá města, internet věcí, IoT, databázové systémy, objektově relační mapování, ORM, optimalizace, paralelizace, multithreading, Typescript, Node.js

Keywords

key performance indikators, KPI, smart cities, internet of things, IoT, database systems, object-relational mapping, ORM, optimization, parallelization, multithreading, Typescript, Node.js

Citace

ŠULC, Ondřej. *Optimalizace zpracování klíčových indikátorů výkonu.* Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Hynek, Ph.D.

Optimalizace zpracování klíčových indikátorů výkonu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Hynka, Ph.D. Další informace mi poskytl pan Ing. František Mikulu, CTO firmy Logimic. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Ondřej Šulc
14. května 2023

Poděkování

Chtěl bych tímto poděkovat svému vedoucímu práce, panu Ing. Jiřímu Hynkovi, Ph.D. především za jeho stálou pomoc, ochotu a trpělivost a za jeho cenné rady při psaní této práce. Zároveň bych chtěl poděkovat panu Ing. Františkovi Mikulů za možnost pracovat na zajímavém projektu Smart City a za rozšíření obzorů v oblasti IoT a chytrých měst.

Obsah

Kapitola 1

Úvod

Stále vyšší popularita Internetu věcí (angl. Internet of Things, zkráceně IoT) způsobuje, že se začíná rozšiřovat z úrovně podniků a průmyslu do podstatně větších rozměrů, a to až na úroveň celých měst. S rozšiřováním přichází nové generace senzorů, díky kterým lze sbírat velké množství užitečných údajů. Mezi takové mohou patřit data o kvalitě ovzduší a životním prostředí, infrastruktuře, využití veřejných služeb nebo péči o veřejné prostranství. Sběr dat umožňuje použít metody k odvození dalších informací, na jejichž základě lze provést úkony vedoucí k rozvoji města a zlepšení kvality života jeho obyvatel.

Neuvážené zavádění vlastních odvozovacích metod však může vést ke špatným rozhodnutím a nesprávnému hospodaření se zdroji. Proto je vhodné použít některou z již osvědčených metod. Jednou z takových jsou [12] doporučované *klíčové indikátory výkonu* (zkr. KPI). Tato metoda umožňuje přesně vyjádřit úspěšnost celého města a identifikovat faktory, které jsou klíčové pro jeho prosperitu.

Metodu KPI používá i projekt Smart City od firmy Logimic. V tomto projektu, vytvořeném na platformě Node.js, dochází ke sběru dat z firemních IoT senzorů, které jsou rozmístěny po městě. Data jsou ukládána do databáze, kde jsou nad nimi prováděny další úpravy. Zpracování naměřených nebo předupravených dat do podoby KPI je důležitý, avšak výpočetně velmi náročný proces. Z výsledků práce [17] vyplývá, že aktuální implementace převodu není dostatečná pro budoucí rozvoj projektu. S rostoucím počtem senzorů by se celková doba potřebná pro zpracování neúnosně prodloužila.

Cílem práce proto bylo provést optimalizaci procesu zpracování. Podle provedené analýzy se skládá z velkého množství vzájemně nezávislých výpočtů. Hlavní technikou optimalizace proto byla zvolena paralelizace. Při paralelním zpracování je možné rozdělit výpočty mezi více vláken, která pak mohou být vykonávána na více dostupných jádrech CPU. Celkové zrychlení dosažitelné paralelizací popisuje Amdahlův zákon. Pro přístup k databázi je v projektu využívána knihovna pro objektově relační mapování (zkr. ORM). Práce tedy zároveň řeší, jak ORM v paralelním prostředí efektivně používat.

Práce nejprve v kapitole 2 podrobněji popisuje koncept chytrých měst, blíže seznamuje s možnostmi využití KPI a zkoumá techniky jejich zpracování. Data, která jsou podkladem pro KPI, mohou být velmi různorodá. V navazující kapitole 3 jsou proto probrány způsoby jejich uchování v různých typech databázových systémů. Kapitola se také zaměřuje na metodu objektově relačního mapování, která pomáhá efektivněji pracovat s relačními databázovými systémy. Kapitola 4 už provádí analýzu implementace zpracování KPI v projektu Smart City. Popisuje architekturu systému, přičemž se zaměřuje na využívání dostupných výpočetních prostředků a na efektivitu práce s databází. V kapitole 5 je pak navrženo několik kroků, kterými lze vyřešit nedostatky objevené během analýzy a provést tak optimalizaci

procesu zpracování KPI. Implementace navržených kroků je podrobně popsána v kapitole 6. Nakonec je nový proces zpracování podroben výkonnostnímu testování a výsledky jsou srovnány s původním řešením. Celý proces testování je popsán v kapitole 7.

Kapitola 2

Chytrá města a klíčové indikátory výkonu

Tato kapitola seznamuje s konceptem chytrého města a IoT v podkapitole 2.1. Dále v podkapitole 2.2 popisuje metodu vyhodnocení výkonu pomocí klíčových indikátorů výkonu (KPI) a nakonec v podkapitole 2.3 uvádí některé způsoby měření výkonnosti chytrých měst.

2.1 Chytré město

Chytré město je koncept, který se snaží pomocí moderních technologií zlepšit životní podmínky městských obyvatel, které se vlivem urbanizace[1] a rozšiřováním průmyslu pomalu zhoršují. Světová zdravotnická organizace ve svém článku [41] řadí urbanizaci mezi hlavní trendy 21. století. Celosvětově žije v městských oblastech kolem 55% všech obyvatel a toto číslo stále roste. Kolem roku 2050 se očekává celkový podíl až 68%. Většina z celkového počtu 4.2 miliardy lidí trpí nedostatkem kvalitního ubytování, transportu, hygieny a kvality vzduchu. Další problémy velkých měst, jako jsou nadměrný hluk, světelný smog, nedostatek prostoru k aktivnímu životu a neekologické nakládáním s odpady, přispívají k šíření nepřenosných nemocí[2]. Vzhledem k této předpokládané koncentraci populace je důležité, aby byla města v budoucnu schopna efektivně hospodařit se svými zdroji a zajišťovala příznivé podmínky pro život.

Technologie IoT je vhodným prostředkem pro řešení tohoto problému. Umožňuje totiž systému města přímou interakci s fyzickým světem. Na základě této interakce pak může systém analyzovat okolní prostředí a provádět potřebné úkony k zajištění kvalitních životních podmínek.

2.1.1 Definice chytrého města

Úplný koncept chytrého města je stále předmětem diskusí a neexistuje obecně přijatá přesná definice. Existuje ale několik vlastností, které jsou napříč literaturou často zmiňovány.

Washburn a spol. [40] definují chytré město jako využití softwarových systémů, síťové a serverové infrastruktury a klientských zařízení pro propojení hlavních komponent a služeb infrastruktury města. Mezi takové služby patří administrativa, vzdělání, zdravotnictví,

[1] Dle slovníku cizích slov: "soustřeďování hospodářského i kulturního života do velkých měst na úkor venkova".

[2] Mezi nepřenosné nemoci (angl. Noncommunicable diseases, zkr. NCDs) patří například nemoci srdce, respirační onemocnění, neurologické poruchy, cukrovka, rakovina a jim podobné.[8]

veřejná bezpečnost, správa nemovitostí nebo doprava. Vše inteligentnější, propojenější a efektivnější.

Chourabi a spol. [9] sestavili seznam vlastností, kterými by chytré město mělo disponovat:

1. Pečlivě monitoruje a spravuje kritickou infrastrukturu (silnice, mosty, tunely, koleje, metro, letiště, rozvody energií).

2. Stará se i o optimální využívání zdrojů ve velkých veřejných budovách, plánuje jejich údržbu, monitoruje bezpečnostní faktory a snaží se maximalizovat jejich službu občanům.

3. Propojuje fyzickou, ICT, sociální a obchodní infrastrukturu za účelem zlepšení rozhodovacích schopností města.

4. Využívá ICT a princip Web 2.0 (společně s dalšími nástroji pro organizaci, design a plánování) pro účely zjednodušení, zlevnění a zrychlení byrokratických procesů a pro účely hledání nových efektivních řešení pro složitou správu města.

5. Využívá technologie Smart Computing k propojení a zlepšení kritické infrastruktury a městských služeb, jako jsou městská správa, školství, zdravotnické služby, veřejná bezpečnost nebo pronájem veřejných nemovitostí.

6. Výborně si vede z perspektivního hlediska v oblasti ekonomiky, počtu a vzdělání obyvatel, vlády, dopravy, životního prostředí a bydlení.

7. Snaží se být stále „chytřejší“ (efektivnější, obyvatelnější, udržitelnější a ke svým občanům spravedlivé).

Některé zdroje uvažují i o mnohem propracovanějších funkcích. Rajab a spol. [32] navrhují funkce řízení denního rozvrhu každého občana, kontrolování jeho zdravotního stavu a případně automatické plánování návštěvy lékaře nebo zavolání záchranné služby. Článek [20] také sepisuje různé způsoby využití moderních technologií, jako je umělá inteligence, nebo poslední dobou velice populární Internet věcí.

2.1.2 Internet věcí

Internet věcí (angl. *Internet of Things*, zkr. IoT) je skupina fyzických zařízení, které jsou spolu vzájemně propojeny komunikační sítí a jejichž cílem je vzájemná spolupráce na celkovém cíli systému. Cíle mohou být nejrůznějšího druhu od jednoduchého snímání dat (IoT senzory), až po fyzickou kooperaci více zařízení.

Rajab a Cinkelr [32] popisují IoT jako infrastrukturu, kterou tvoří moderní vozidla, budovy a další důležitá, denně používaná elektrická zařízení. Tato vzájemně propojená zařízení pak mohou společně shromažďovat data a vzájemně je sdílet. Umí spolupracovat, komunikovat a tvořit hierarchie a to vše bez nutného zásahu člověka.

S IoT se už dnes často setkáváme v nejrůznějších oblastech. Největším průkopníkem v této oblasti je automatizace průmyslu. Díky ní dnes existují plně automatizované továrny a chytré průmyslové stroje, což umožňuje efektivnější produkci a cenově dostupnější výsledný produkt. Různé způsoby využití IoT v průmyslu jsou popsané v [7].

Ostatní oblasti nicméně nezůstávají pozadu. Ve městech se setkáváme s chytrými parkovacími domy. V domácnostech se objevují chytré spotřebiče, které rozumně hospodaří

s energií. V oblasti zdravotnictví je v České republice známá např. aplikace Záchranka[3], která umožňuje uživateli efektivní komunikaci se složkami integrovaného záchranného systému. Kromě odeslání přesné polohy při tísňovém volání lze také v telefonu vyhledat nejbližší AED[4] nebo vyhledat pohotovost.

Využití pro IoT je spousta a popularita tohoto tématu stále roste. Dokazuje to například článek IoT Analytics [16], který popisuje stále rostoucí počet zařízení na trhu. V roce 2021 počet zařízení vzrostl o 8%, což představuje cca 12,2 miliardy nových zařízení. Tento nárůst je nižší než v předchozích letech z důvodu probíhající pandemie COVID-19 a krize ohledně nedostatku čipů. Podle odhadů z května 2022 se však do konce roku 2022 očekává nárůst až 18% a kolem roku 2025 se očekává, že bude existovat až 27 miliard připojených zařízení.

2.2 Měření výkonu

Od chytrých měst se očekává, že budou na základě dostupných dat (např. získaných od zmíněných IoT senzorů) provádět úkony, které vždy povedou k lepším životním podmínkám obyvatel. Aby však bylo možné vyhodnotit správnost těchto úkonů, je potřeba definovat metriky hodnotící celkový výkon města. Tento výkon by se měl s každým provedeným úkonem vždy zvyšovat. Neuvážené zavádění měřících metod však může vést ke špatným rozhodnutím a nevhodnému hospodaření se zdroji.

Eckerson [12] uvádí příklad špatně nastavených metrik přepravní společností, která svůj výkon měří jako procento včas dodaných zásilek. Výsledkem takové strategie pak může být vysoké navýšení dopravních nákladů, protože dispečerům nic nebrání v odesílání poloprázdných vozidel. Aby tedy strategie byla efektivní, společnost musí zavést druhou metriku, která bude nevyužitý nákladní prostor zohledňovat. Tímto dojde opět ke stabilizaci využívání zdrojů a dispečeři jsou automaticky navedeni se například pokusit se zákazníkem vyjednat jiný termín dodání.

Jednou z možných metod měření výkonu jsou klíčové indikátory výkonu.

2.2.1 Klíčové indikátory výkonu

Klíčové indikátory výkonu (angl. *Key Performance Indicators*, zkr. KPI) se snaží co nejpřesněji vyjádřit celkový výkon/úspěšnost celého města nebo jednotlivých obyvatel a identifikovat výkonnostní faktory, které jsou klíčové pro budoucí úspěch.

Tato metoda je často používaná i v menším měřítku – např. na úrovni firem a jejích zaměstnanců. Stala se tedy vcelku oblíbenou metodou měření mezi manažery a dalšími lidmi zodpovědnými za výkonnost různých projektů.

Parmenter [27] tuto metodu rozděluje na 3 podtypy měřítek výkonnosti:

1. *Klíčové ukazatele výsledků* (angl. *Key Result Indicators*, zkr. KRI) – jsou měřítka, která jsou typicky výsledkem mnoha činností a je z nich ihned patrný správný vývoj projektu. Obvykle prezentují výsledky za delší měřený časový úsek (měsíc nebo čtvrtletí) a nemění se ze dne na den. Nevyčteme z nich však informaci, jakými způsoby celkové výsledky zlepšit. Toto měřítko se tedy spíše používá jako nástroj pro vykazování výsledků lidem, kteří nejsou zapojeni do každodenního managementu (např. vedení firmy, investoři). Všechna KRI je vhodné ve výsledku společně prezentovat např. prostřednictvím řídící desky (angl. dashboardu), která umožní rychlou analýzu.

[3] https://www.zachrankaapp.cz
[4] automatický externí defibrilátor

2. *Ukazatele výkonnosti* (angl. *Performance Indicators*, zkr. PI) – jsou především pomocná měřítka, která se dají využít buď pro doplnění KPI, nebo k objasnění výsledků zobrazených KRI.

3. *Klíčové ukazatele výkonnosti* (angl. *Key Performance Indicators*, zkr. KPI) – jsou měřítka zaměřená na nejkritičtější výkonnostní faktory, na jejichž stavu závisí budoucí úspěch celého projektu. KPI se v průběhu chodu nemění a to, že by nějaká nová přibývala nebo naopak byla odstraněna se stává velmi zřídka. Pokud k tomu dojde, je příčinou spíše jejich špatná původní identifikace. Z toho důvodu také rovnou popisuje 7 typických vlastností měřítek KPI:

    - Nejsou finančního charakteru (nevyjadřují se v korunách a ani jiné měně).
    - Jsou měřena často a opakovaně (každý den nebo dokonce neustále), protože KPI se zaměřuje na současnost a budoucnost, k čemuž jsou potřeba nejaktuálnější data. Naopak na data z delší minulosti se nebere ohled.
    - Zabývá se jimi ředitel a tým vrcholového vedení nebo vedoucí projektu.
    - Pochopení daného měřítka a opatření k nápravě je požadováno po všech pracovnících.
    - Za dané KPI je odpovědný zvolený jednotlivec nebo tým.
    - Mají významný dopad na úspěch. Ovlivňují většinu kritických faktorů úspěchu.
    - Mají pozitivní dopady. Zlepšení daného KPI pozitivním způsobem ovlivní i ostatní měřítka výkonnosti.

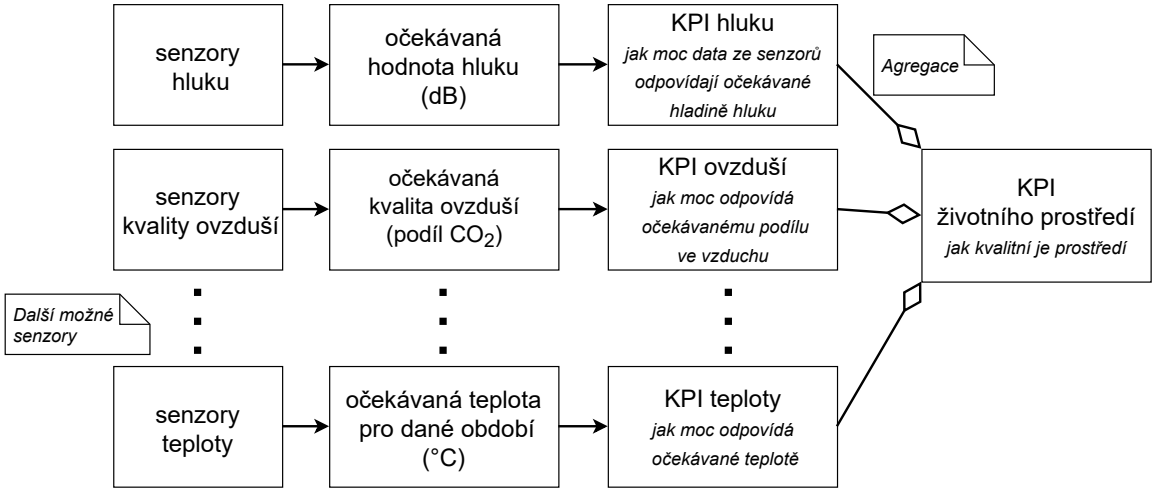
K popsání vztahu těchto tří měřítek pak používá analogii s cibulí. Vnější vrstva představuje KRI, protože je na první pohled vidět celkový stav. Vnitřní vrstvy představují PI a samotným jádrem jsou KPI, které musí být v dobrém stavu, aby cibule dále rostla.

Oproti tomu Eckerson ve své knize [12] spíše preferuje rozdělení KPI na *leading* a *lagging* (v překladu do češtiny lze podle [34] použít výrazy *prediktivní* a *reflexivní*). Prediktivní indikátory se zaměřují na aktivity, které mají zásadní vliv na budoucí výkon, zatímco reflexivní zkoumají výsledky aktivit z minulosti. Podle jeho průzkumu však pouze 15% firem po definování svých KPI je už nadále nemění. Ostatní firmy pravidelně provádí úpravy buď každý rok nebo čtvrtletí. Nejčastějším důvodem (77%) je změna obchodní strategie, která vyžaduje právě i úpravu KPI. Zbylé důvody změn jsou pak nejčastěji způsobeny chybným počátečním definováním.

2.3 Implementace KPI pro chytrá města

Jak už popisují předchozí podkapitoly, hlavním cílem chytrého města je obecná prosperita jeho obyvatel a technologie IoT představují vhodný nástroj pro automatické řešení mnoha požadavků. Různé typy IoT senzorů rozmístěné po městě snímáním svého okolí generují obrovské množství cenných dat. Tato data je ale potřeba správně zpracovat a získat z nich důležité informace – např. právě v podobě KPI.

Podle [12] je jedním ze způsobů vyhodnocení KPI stanovení intervalů očekávaných hodnot. Z toho vyplývá, že systému pak stačí u naměřených dat kontrolovat, zda odpovídají očekávanému intervalu (intervaly jsou různé pro každý typ dat). Tyto výsledky pak lze snadno agregovat do jedné hodnoty a tím navíc umožnit rychlou kontrolu celkového stavu. Tento princip je graficky znázorněn na obrázku 2.1.



Obrázek 2.1: Příklad vyhodnocení KPI na základě dat ze senzoru

Mezi KPI chytrých měst však nepatří pouze ty, které lze vyčíst ze senzorů. Takovými jsou například informace o obyvatelích města (jejich počet, stáří nebo vzdělání). Tyto další KPI je nutné v rámci celého systému také propojit a zajistit přehlednou kontrolu nad všemi částmi systému. Nicméně vzhledem k tomu, že už se jedná o jiný druh dat, tak jednoduchá agregace nemusí být možná. Přístupy k problematice takového propojení jsou různé. Angelakoglou a spol. [2] navrhují rozdělení KPI do kategorií podle tří dimenzí. První dimenze rozděluje KPI podle 4 zainteresovaných stran:

1. distributoři elektrické energie,
2. koncoví uživatelé (občané),
3. poskytovatelé technologií a služeb (soukromé firmy),
4. vládní instituce.

Druhá dimenze rozděluje KPI na následující oblasti:

1. KPI technického výkonu – kontrolují například spotřebu energie a podíl obnovitelných zdrojů.
2. KPI životního prostředí – kontrolují množství CO_2 v atmosféře, okolní hluk nebo světelný smog.
3. KPI ekonomiky – jsou zaměřená na správu finančních prostředků a efektivní. obchodování.
4. KPI sociálních záležitostí – jsou zaměřená na spokojenost občanů a veřejné mínění.
5. KPI ICT – jsou zaměřená na využívání chytrých aplikací, které uživatelům umožňují komunikovat se systémem.
6. KPI legislativy – jsou zaměřená na efektivní integraci nových poznatků a technologií do právního rámce.

Třetí dimenze rozděluje KPI podle geografické rozlohy na tyto úrovně:

1. budova,
2. řada budov,
3. napájecí jednotka,
4. řada napájecích jednotek,
5. městská část,
6. celé město.

Každý KPI by navíc měl být škálovatelný a replikovatelný. Škálovatelnost znamená možnost využití na různých geografických úrovních (3. dimenze), aniž by došlo ke snížení efektivity. Replikovatelnost znamená možnost využití daného KPI v jiných geografických podmínkách (jiných městech). Nakonec navrhují celkem 75 KPI spadajících do různých kategorií. Zde jsou příklady některých z nich:

1. Cenová efektivita redukce CO_2 – snižování uhlíkové stopy za přijatelné finanční náklady:
   - zainteresované strany: poskytovatelé technologií, vládní instituce,
   - oblast: KPI ekonomiky,
   - geografická úroveň: budova, řada budov, městská část, celé město,
   - měrná jednotka: eura/(tuna CO_2 * 1 rok).
2. Využití open-source softwaru – levnější cena softwarového řešení, možnost komunity se podílet na vývoji:
   - zainteresované strany: poskytovatelé technologií, vládní instituce,
   - oblast: KPI ICT,
   - geografická úroveň: celé město,
   - měrná jednotka: 5 stupňová Linkertova stupnice[5].
3. Bezpečnost dat – zabezpečení systému před kybernetickými útoky všech typů:
   - zainteresované strany: koncoví uživatelé, poskytovatelé technologií, vládní instituce,
   - oblast: KPI ekonomiky,
   - geografická úroveň: celé město,
   - měrná jednotka: útok.

[5] 5 stupňová stupnice na které respondenti uvádí svůj souhlas s probíraným tématem typicky 1 = zcela souhlasím – 5 = zcela nesouhlasím

4. Hluková zátěž – dlouhodobé vystavení nadměrnému hluku:
    - zainteresované strany: distributoři elektrické energie, koncoví uživatelé, poskytovatelé technologií, vládní instituce,
    - oblast: KPI životního prostředí,
    - geografická úroveň: městská část, celé město,
    - měrná jednotka: dB nebo %.

Tento způsob však není jediný, jak lze s KPI chytrých měst pracovat. Další možný způsob popisují Bosch a spol. [6]. Ti nejprve uvádí hlavní oblasti zájmu a jejich kategorie, které jsou vypsány v tabulce 2.1.

Obyvatelstvo	**Planeta**	**Prosperita**	**Vláda**	**Propagovatelnost**
zdraví	energie	zaměstnanost	organizace	škálovatelnost
bezpečnost	materiály, voda, půda	spravedlnost	zapojení komunity	replikovatelnost
dostupnost služeb	klimatická odolnost	zelená ekonomika	více vrstvá vláda	
vzdělání	odpady a znečištění	ekonomický výkon		
etnická rozmanitost	ekosystém	inovace		
kvalita ubytování		konkurenceschopnost		

Tabulka 2.1: Hlavní oblasti zájmu a jejich kategorie podle Bosch a spol. [6]

V rámci své práce zmiňují, že je důležité zmiňované KPI využít nejen pro vyhodnocení celkového výkonu chytrých měst, ale i přímo jeho dílčích projektů – ty totiž představují kroky, kterými se dané město zlepšuje.

Projekty se dělí na jednotlivé dílčí *aktivity*. KPI projektů jsou pak rozděleny na vstupní, procesní, výstupní a výsledné.

- *Vstupní* – monitorují zdroje potřebné k provedení aktivity. Především měří množství, kvalitu a stáří zdrojů. Zdroje nemusí být jen materiální, ale i humanitní a finanční.
- *Procesní* – měří míru skutečného provádění plánovaných aktivit. Příkladem může být uskutečnění školících kurzů nebo provedení instalace chytrých čidel.
- *Výstupní* – kontrolují a posuzují výstup prováděné aktivity. Například množství čidel, které byly nainstalovány nebo počet proškolených lidí.
- *Výsledné* – kontrolují výsledky dílčích aktivit, které směřují ke splnění celkového cíle projektu. Měly by také odkazovat, proč byly dané dílčí aktivity uskutečněny.

Nakonec, podobně jako [2] v předchozím případě, uvádí výčet různých městských KPI, které jsou zařazeny do jednotlivých kategorií z tabulky 2.1. Zde jsou opět některé příklady:

1. Podpora zdravého životního stylu:
   - oblast: obyvatelstvo,
   - kategorie: zdraví,
   - měrná jednotka: 5 stupňová Linkertova stupnice.
2. Snížení množství dopravních nehod:
   - oblast: obyvatelstvo,
   - kategorie: bezpečnost,
   - měrná jednotka: %.
3. Zvýšení lokální produkce obnovitelné energie:
   - oblast: planeta,
   - kategorie: energie,
   - měrná jednotka: % v kWh.

Podobný způsob je aplikován i v dalších článcích, jako např. [13] nebo [15], které také navrhují vlastní KPI a jejich strukturální dělení.

Metoda popsaná v této podkapitole nabízí poměrně intuitivní a systematický přístup k měření výkonnosti chytrých měst a při správně navržené architektuře systému umožňují dynamické přidávání a odebírání jednotlivých měřítek – protože jak bylo zmíněno, často se nepodaří všechny KPI správně identifikovat na první pokus.

Na závěr je vhodné zdůraznit, že KPI jsou jen jednou z mnoha metod, jak lze výkon chytrých měst měřit. Existují metody založené na zcela odlišných přístupech, jako např. Agbali a spol. [1], kteří navrhují model využívající systémovou dynamiku.

Kapitola 3

Databázové systémy

V kapitole 2 bylo popsáno, jakým způsobem lze vyhodnocovat výkon chytrých měst. Podkladem ke správnému vyhodnocování jsou však vždy kvalitní data. Ta mohou pocházet z různých senzorů rozmístěných po městě, ale může se jednat i o další např. statistické údaje. Veškerá data je potřeba někde uchovat, a protože jich je obrovské množství, je nutné při ukládání zajistit jistou systematičnost a přehlednost. K tomu slouží databázové systémy (zkr. DBS). Existují různé typy DBS a jejich využití je závislé na tom, jaký typ dat je zrovna potřeba ukládat.

Relační DBS jsou navrženy pro zajištění věrné reprezentace aktuálního stavu a snaží se ukládat a uchovávat všechna data tak, aby byla vždy zajištěna jejich konzistentnost. K tomu slouží speciální procedury, které však využívají nemalé výpočetní prostředky (více v podkapitole 3.1). Vyhodnocovaná data ale někdy mají povahu rychlého a častého zápisu, přičemž kontrola zápisu všech dat při daném množství nehraje velkou roli (např. data ze senzorů). Ve výsledku tak může vlivem snahy o zajištění konzistentnosti docházet spíše k zahlcování databáze zmíněnými procedurami. Zároveň z pohledu výpočtu prediktivních a reflexivní KPI je nutné, aby systém uměl uchovávat jak aktuální stav prostředí, tak i jeho historii. Uchovávání historie přímo v relačních DBS je však spekulativní řešení, které jistě není vždy optimální. Kromě relačních DBS je tedy nutné uvažovat i nad jinými druhy systémů, jako jsou NoSQL databáze, které jsou navržené pro specifické případy užití.

Nakonec je důležité uvažovat i nad samotným přístupem k těmto databázím ze zdrojového kódu. Například paradigma přístupu k datům v objektově orientovaných jazycích (zkr. OOJ) se značně liší od přístupu k datům v relačních databázích. Existují proto nástroje, jako je objektově relační mapování, které provádí automatickou konverzi mezi těmito přístupy.

Tato kapitola nejprve v podkapitole 3.1 zkráceně připomíná základní koncepty relačních databází a v podkapitole 3.2 popisuje přístup k nim pomocí objektově relačního mapování. Následně v podkapitole 3.3 popisuje základní typy NoSQL databází a způsoby jejich využití. Na závěr v podkapitole 3.4 popisuje způsoby využití zmíněných systémů pro zpracování KPI.

3.1 Relační databáze

Relační databáze jsou podle [11] nejvyužívanějším typem databází. Jsou totiž základem všech robustních aplikací a jejich architektura umožňuje univerzální použití. Tato podkapitola seznamuje s koncepty, na kterých tento typ databází stojí.

3.1.1 Relační model

Z knihy [29] jsou relační databáze založené na *relačním modelu dat*. Ten definuje pojem *relace* jako *n-tici* ve tvaru $(a_1,a_2, \ldots ,a_n)$, kde n je řád relace. Jednotlivé komponenty a_i jsou z domény atributu A_i. *Doména* je specifikovaná množina hodnot, kterých může atribut nabývat. Samotné A_i je pouze název daného atributu. Doména atributu je vyjádřena jako $dom(A_i)$ – zápisem $A_i{:}dom(A_i)$ je pak atributu přidělena jeho doména. Relaci pojmenovanou jako R lze tedy popsat jako $R(A_1{:}D_1,A_2{:}D_2, \ldots ,A_n{:}D_n)$, kde $D_i{=}dom(A_i)$, pro $i \in \langle 1,n \rangle$. Tento zápis tvoří *schéma relace*.

Integritní omezení (IO) jsou popsány jako logické podmínky, které musí data uložená v databázi splňovat. Jedním z hlavních IO je specifikace klíče schéma relace. *Klíč* schéma $R(A)$ je minimální množina atributů z A, jejichž hodnoty budou jednoznačně určovat n-tice relace R – minimální znamená, že z ní nelze odebrat žádný atribut, aniž by došlo k narušení schopnosti jednoznačného určování. Klíčů v daném schéma může být více (někdy bývají nazývané také jako *alternativní klíče* nebo *kandidátní klíče*) a je proto nutné vybrat z nich jeden *primární klíč* – vybírá se nejčastěji klíč s nejmenším množstvím atributů. Dalším důležitým IO je referenční integrita, která popisuje vztah mezi daty obsaženými ve dvou relacích. Atribut, ovlivněný referenční integritou se nazývá *cizí klíč*. Hodnota cizího klíče musí být v každé n-tici relace buď prázdná, nebo musí být obsažena jako hodnota primárního klíče jiné relace.

Pojmem *relační schéma databáze* je označena dvojice (R,I) kde R je množina schémat relací a I je množina integritních omezení. Relační databází se schématem (R,I) pak nazýváme takovou množinu relací, jejíž prvky vyhovují integritním omezením I – alternativně lze danou množinu relací označit za *konzistentní*. *Stav databáze* pak slouží ke zdůraznění proměnlivosti databáze v čase.

3.1.2 Jazyk SQL

Jazyk SQL je v [29] popsán jako základní prostředek komunikace s databázovým systémem pro relační a nerelační databáze. Je využitelný jak pro definici dat, tak i pro manipulaci s nimi a to prostřednictvím různých operací:

- založení databáze,
- definování tabulek a indexů,
- stanovení integritních omezení,
- vkládání a změny dat,
- výběr dat z databáze,
- zobrazení dat.

Syntaxe jazyka je odvozena z angličtiny a připomíná věty přirozeného jazyka – například příkaz:

```
SELECT Autor FROM Knihy WHERE Cena > 100;
```

lze přirozeně přečíst a přeložit jako „Vyber autory knih, jejichž cena přesahuje 100 Kč“. Jazyk SQL se v současné době používá ve většině relačních databázových systémů, jako jsou například OracleDB, MySQL nebo PostgreSQL.

3.1.3 Relační databázové systémy

Relační databázové systémy (zkr. RDS) jsou založené na relačním modelu. Analogicky k relačnímu modelu je využívána tzv. tabulková terminologie – schéma relace je záhlaví tabulky, n-tice relace jsou řádky tabulky a atributy jsou sloupce tabulky. Pro komunikaci s nimi se nejčastěji využívá jazyk SQL.

Důležitou vlastností RDS je tzv. transakční zpracování požadavků. *Transakce* tvoří jednotku práce na databázi. Jedná se o posloupnost databázových operací, přičemž tyto kroky převedou databázi z jednoho konzistentního stavu do druhého. Princip transakčního zpracování spočívá v zajištění provedení všech operací v dané posloupnosti za všech okolností. Pokud by nastala situace, že některé operace nebyly provedeny, nastává tzv. zotavení – tedy návrat do původního konzistentního stavu. Bližší popis těchto procesů je popsán v [29].

Samotná databáze je ale v praxi jen inteligentní úložiště. Řídící logiku musí zajistit jiná aplikace, která pak s databází potřebuje komunikovat. K této komunikaci sice lze využít zmíněný jazyk SQL, avšak pokud dojde k nesprávné implementaci, pak tato metoda vede ke vzniku různých zranitelností a nepřehlednosti ve zdrojovém kódu. Typický je například útok SQL Injection[1]. Proto vznikají metody, jako je objektově relační mapování, které komunikaci s databází zprostředkovávají.

3.2 Objektově relační mapování

Článek [42] popisuje *objektově relační mapování* (angl. *Object-relational mapping*, zkr. ORM) jako programovací techniku, která umožňuje plynulou konverzi dat mezi dvěma typově nekompatibilními systémy – relační databází a objektově orientovaným programovacím jazykem, která tak vytváří efekt „virtuální objektové databáze“.

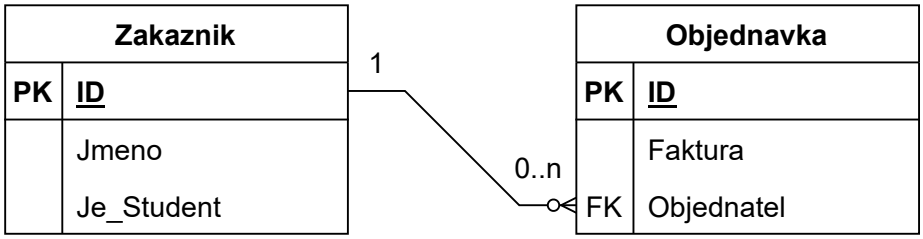
Tento koncept přístupu k relačním databázím je z pohledu objektově orientovaných jazyků (zkr. OOJ) pro programátora přirozenější. Programátor by měl totiž pro práci s daty využívat objekty a nikoli text (resp. SQL dotazy). Koncept ORM spočívá v definování speciálního typu objektů, které pak reprezentují záznamy uložené v databázi. Zároveň jakákoli úprava takových objektů automaticky způsobí úpravu i odpovídajících záznamů bez dalšího zásahu programátora. V této diplomové práci jsou tyto speciální objekty označovány jako objekty ORM.

3.2.1 Použití ORM

V programovacím jazyce je ORM zpravidla zajištěno speciální *knihovnou pro ORM*. Konkrétní implementace se může napříč jazyky a jednotlivými knihovnami lišit. V rámci této sekce je tedy představen jen hlavní koncept využití těchto knihoven.

[1] https://owasp.org/www-community/attacks/SQL_Injection

Na obrázku 3.1 je pomocí ER diagramu zobrazen vztah *Zákazník-Objednávka*. Tento vztah bude nyní využit v několika příkladech, které se nachází v této podkapitole.



Obrázek 3.1: ER diagram vztahu `Zákazník-Objednávka`

Když je znám datový model, je možné začít s implementací ORM. Nejprve je nutné daný model vyjádřit i ve zdrojovém kódu. To se typicky dělá vytvořením tříd. Při tvorbě je nutné neopomenout zavedený vztah `1:N`. Ten je typicky vyjádřen definováním příslušných parametrů v **obou** třídách. Zároveň se v rámci vztahu nepoužívá typ hodnoty v databázi (např. `int ID`), ale **celá třída** (např. `Zakaznik`). V závislosti na použitém programovacím jazyce může třída navíc obsahovat specifický konstruktor nebo další syntaktické prvky.

```
1  Class Zakaznik
2  {
3     [DatabaseGenerated]
4     int ID;
5
6     string Jmeno;
7     bool Je_Student;
8     List<Objednavka> Objednavky;
9  }
```

Výpis 3.1: Datový model `Zákazník`

```
1  Class Objednavka
2  {
3     [DatabaseGenerated]
4     int ID;
5
6     string Faktura;
7     Zakaznik Objednatel;
8  }
```

Výpis 3.2: Datový model `Objednávka`

Instance těchto tříd jsou pak zmíněnými objekty ORM a reprezentují jednotlivé záznamy v tabulkách databáze. Definovaný datový model je nyní předán knihovně pro ORM. Způsob předání je individuální pro každou knihovnu, nicméně většinou se tak děje využitím dědičnosti a speciálních tříd definovaných danou knihovnou.

```
1  Class ApplicationDbContext : DefaultDbContext
2  {
3     Table<Zakaznik> Zakaznik;
4     Table<Objednavka> Objednavka;
5  }
```

Výpis 3.3: Databázový kontext

Instance třídy `ApplicationDbContext` zprostředkovává přístup k objektům ORM. Pomocí nich pak lze automaticky přistupovat do databáze bez definování SQL dotazu. Následující kód znázorňuje příklad vypsání počtu objednávek od zákazníka s ID 0001.

```
1 main(ApplicationDbContext db)
2 {
3    Zakaznik zakaznik_0001 = db.Zakaznik.WithID(0001)
4    int pocet_objednavek = zakaznik_0001.Objednavky.Count();
5    print(pocet_objednavek);
6 }
```

Výpis 3.4: Využití databázového kontextu

V kódu se tedy vypisuje počet položek v seznamu `List<Objednavka> Objednavky`, nicméně knihovna zajišťuje, že je tento počet shodný s počtem záznamů v databázi.

3.2.2 Dopad ORM na výkonnost relačních dotazů

Jak již bylo popsáno v podkapitole 3.1, pro komunikaci s relačními databázemi se používá jazyk SQL. Využití ORM neznamená, že pro komunikaci s databází se tento jazyk přestane využívat. Pouze dochází k zajištění automatického generování potřebných SQL dotazů na základě práce s objekty ORM. Z toho důvodu je tedy důležité se při využití ORM důkladně seznámit s využívanou knihovnou a při práci s objekty ORM dodržovat jisté zásady. V opačném případě hrozí vznik režie, kterou by pak programátor těžko odhaloval. Následujících dva příklady *Problém N+1* a *Problém redundantních dat* demonstrují 2 nejčastější chybné způsoby práce s knihovnou pro ORM. V těchto příkladech je využit stejný model `Zákazník-Objednávka` jako v předchozí sekci 3.2.1.

Problém N+1

Problém N+1 vzniká při práci s objekty ORM v rámci cyklu. Při špatné formulaci příkazů dochází ke generování zbytečně velkého množství SQL dotazů, které dramaticky zatěžují databázi.

```
1 List<Zakaznik> studenti = db.Zakaznik.Where(Zakaznik.Je_Student);
2                         //db je databazovy kontext
3 foreach (Zakaznik student in studenti)
4 {
5    foreach (Objednavka obj in student.Objednavky)
6    {
7       //Cinnost se vsemi objednavkami zakazniku, kteri jsou studenti.
8    }
9 }
```

Výpis 3.5: N+1 problém

Problém N+1 je ve výpisu 3.5 způsoben použitím chybné strategie načítání objektů z databáze. V kódu nejprve dojde k zaslání SQL dotazu pro načtení skupiny zájmu z tabulky `Zakaznik` (ř. 1) a následně je využita relace `Zakaznik.Objednavky` pro iteraci nad objekty z tabulky `Objednavka` (ř. 5) – tím dochází ke generování dotazu pro každý objekt `obj`,

protože tyto objekty zatím nebyly načteny z databáze (tzv. *lazy loading* – načítání objektů až když jsou potřeba). Ve výsledku je tedy zasláno N+1 dotazů (N dle počtu objednávek).

Tomuto problému však lze předejít. Z pohledu programátora je jasné, že objekty z tabulky `Objednavka` budou potřeba a je tedy vhodné zajistit jejich načtení již při prvním SQL dotazu (tzv. *eager loading* – načítání objektů při prvním uvedením do kontextu). Eager loading lze ve většině knihoven zajistit přidáním volání metody *Include*. Kód na řádku 1 lze tedy upravit takto:

```
List<Zakaznik> studenti = db.Zakaznik.Where(Zakaznik.Je_Student)
                                     .Include(Zakaznik.Objednavky);
```

Metoda `Include` tedy slouží ke specifikaci dat, které se mají **také** načítat.

Problém redundantních dat

Problém redundantních dat je přesným opakem problému N+1. Způsobuje načítání dat, které nejsou potřeba k výpočtu, čímž zbytečně plýtvá pamětí a zahlcuje komunikaci s databází.

```
1 foreach (Objednavka obj in db.Objednavka)
2 {
3    print("Id objednavky: ", obj.ID);
4 }
```

Výpis 3.6: Problém redundantních dat

Kód ve výpisu 3.6 pouze vypisuje ID všech objednávek v tabulce `Objednavka`. Na pozadí však dochází k načítání celých objektů, i když je potřeba vždy jen jeden parametr – v tomto případě zbytečně dochází i k načítání parametru `Faktura` (viz. ER diagram na obrázku 3.1) . Opět se tedy jedná o chybnou strategii načítání objektů. V tomto případě tedy načítání celých objektů (eager loading) chceme nahradit načítáním jen potřebných částí (lazy loading). Lazy loading lze ve většině knihoven zajistit přidáním volání metody *Select*. Kód na řádku na řádku 1 lze tedy upravit takto:

```
foreach (Objednavka obj in db.Objednavka.Select(Objednavka.ID))
```

Metoda `Select` tedy na rozdíl od předchozí metody `Include` slouží ke specifikaci dat, které se **jako jediné** mají načítat.

Další aspekty ovlivňující výkonnost ORM

Článek *Efficient Querying* [22] uvádí seznam hlavních aspektů, se kterými by se měl programátor seznámit, než začne danou knihovnu pro ORM využívat. Mezi tyto aspekty patří:

- správné využívání indexování,
- správné využívání eager loading a lazy loading,
- omezení velikosti výsledku a stránkování velkých odpovědí,
- využívání asynchronního programování.

Dále uvádí, že v některých případech se dokonce může stát, že pro dynamicky vygenerovaný SQL dotaz sice existuje efektivnější formulace, nicméně omezené vyjadřovací prostředky knihovny neumožňují jejího dosažení. Pro takové případy většina knihoven poskytuje možnost daný SQL dotaz definovat ručně.

Článek [10] upozorňuje, že navzdory intuici samotný způsob práce s objekty ORM ale není to jediné, co má na konečný výkon dopad. Velmi důležitá je i vhodná konfigurace použité knihovny pro ORM, správné schéma datového modelu, a dokonce i konfigurace samotné SQL databáze. Souhrnně pak jako hlavní nedostatky, které se u dynamicky generovaných SQL dotazů objevují, uvádí:

- eager loading nepotřebných atributů (nejčastěji ID),
- zbytečné vnořování poddotazů (subqueries),
- zbytečné využívání seřazování (ORDER BY),
- obtížná čitelnost vygenerovaného SQL dotazu,
- větší výpočetní plán (query plan),
- duplikátní kód při čtení více záznamů.

Využití ORM tedy může zásadně zrychlit vývoj aplikace a usnadnit práci s daty uloženými v databázi. Je však důležité se vždy seznámit s nástroji a vyjadřovacími prostředky použité knihovny, aby nedocházelo ke vzniku nežádoucí režie a jen v extrémních případech přistoupit k vlastnímu definování SQL dotazu.

3.2.3 Srovnání knihoven pro ORM v jazyce TypeScript

V běžně používaných objektově orientovaných jazycích (např. jazyky Java a .NET) je zvykem, že pro ORM se používá jedna známá a prověřená knihovna (pro Javu je to knihovna Hibernate a pro .NET Entity Framework). Díky tomu jsou zvyky a přístupy programátorů v těchto jazycích jednotné. To umožňuje soustředit se na zdokonalování jednoho nástroje a také vede k efektivnějšímu vývoji.

V jazyce TypeScript však žádná taková nejčastěji využívaná knihovna není. Naopak knihoven pro ORM je na výběr více a je tedy na programátorovi, aby si vhodně vybral. V této sekci jsou proto porovnány vlastnosti 3 knihoven, které podle [26] patří za časové období od dubna do října 2022 k nejpoužívanějším.

Při srovnávání byl kladen důraz na:

- způsob definice schéma databáze,
- způsob formulace dotazů na databázi,
- kontrola eager a lazy loadingu,
- možnosti limitování obsahu a stránkování,
- podpora cache (lokální uchovávání obrazu databáze),
- seznam podporovaných databází.

Sequelize

V knihovně Sequelize[2] lze model databáze definovat dvěma způsoby. První možností je definovat entitu jako třídu, která dědí z třídy `Model`. Druhou možností je využít funkci `sequelize.define()`, která jako parametry přijímá název entity, seznam atributů a parametr s integritními omezeními. Dotazy na SQL databázi mají strukturu *JSON* (angl. *JavaScript Object Notation*) objektu se specifickou strukturou. Eager a lazy loading jsou řešeny vkládáním objektů s parametry `include` a `select`. Také limitování velikosti odpovědí a stránkování odpovědí je podporováno jednoduchým vložením objektu s parametry `limit` a `offset`. Knihovna nepodporuje cache, ale existují komunitní rozšíření. Další informace jsou dostupné v oficiální dokumentaci [33].

Podporované databáze jsou: PostgreSQL, MySQL, MariaDB, SQLite a Microsoft SQL Server.

TypeORM

V knihovně TypeORM[3] jsou entity opět definovány pomocí tříd, ovšem se silnou závislostí na klíčových slovech. Třída představující entitu tedy musí být označena slovem `@Entity()` a její parametry slovem `@Column()`. Dotazy na SQL databázi mají stejně jako v Sequelize strukturu JSON objektu se specifickou strukturou. Eager a lazy loading je možné řešit už na úrovni definice relací. Relace je tedy vždy označena jako eager nebo lazy v závislosti na použití datového typu `Promise`. Při samotné tvorbě dotazu je možné dodatečně způsob načítání ovlivnit vložením objektů `select` a `relations` v metodě `find`. TypeORM podporuje limitování velikosti odpovědi i stránkování pomocí metod `skip` a `take` při tvorbě dotazu. Cache je podporována na úrovni dotazů prostřednictvím parametru `cache`. Další informace je možné dohledat v oficiální dokumentaci [37].

Podporované databáze jsou: MySQL, MariaDB, PostgreSQL, CockroachDB, SQLite, Microsoft SQL server, OracleDB, SAP Hana, MongoDB.

Prisma

Prisma[4] patří k nejmladším knihovnám pro ORM jazyka TypeScript. Narozdíl od ostatních knihoven, Prisma pro definování entity nepoužívá třídy, ale speciální typ objektů `model`. Tyto objekty navíc mohou být uloženy do speciálního souboru `schema.prisma`. V rámci tohoto modelu umožňuje konfiguraci vlastností pomocí klíčových slov. SQL dotazy jsou opět generovány ze struktury JSON objektu. Eager a lazy loading jsou řešeny vkládáním objektů include a select – stejně jako v Sequelize. Limitování velikosti odpovědi i stránkování je podporováno pomocí parametrů `skip` a `take`. Prisma přímo nepodporuje cache, ale je možné dodatečně nainstalovat `prisma-cache-middleware`, který tuto funkci dodává. Další informace jsou dostupné v oficiální dokumentaci [30].

Podporované databáze: PostgreSQL, MySQL, MariaDB, SQLite, AWS Aurora, Microsoft SQL server, Azure SQL, MongoDB, CockroachDB.

[2] https://sequelize.org/
[3] https://typeorm.io/
[4] https://www.prisma.io/

3.3 NoSQL databáze

Relační databáze vznikly za účelem vytvoření stabilní datové základny, která uchovává strukturovaná data přesně podle definovaného schéma, zajišťuje jejich konzistenci a poskytuje přirozené dotazování. Za tyto vlastnosti ale relační databáze platí vysokou daň v podobě výpočetní náročnosti. Někdy je ale rychlé úložiště klíčové a tradiční databáze jsou pro takové úlohy až příliš „těžkopádné“.

Princip NoSQL databází je přímo vyjádřen jejich názvem – „žádné SQL“. Cílem NoSQL databází je zvolit alternativní přístup ke zpracování dat, který místo obecně využitelné architektury bude zaměřen a optimalizován pro ukládání jednoho druhu dat.

Podle [36] existují 4 hlavní typy NoSQL databází:

- **Databáze klíč-hodnota**

  Tento typ databází funguje na principu asociativního pole – pole jehož položky jsou typu klíč-hodnota. Klíče jsou v rámci celé databáze unikátní. Díky tomu, že vyhledávání pouze spočívá v nalezení konkrétního klíče tak tyto databáze disponují velice rychlým vyhledáváním.

  Příklad databází: Cassandra, Voldemort.

- **Sloupcově orientované databáze**

  Relačních databáze ukládají data do záznamů, přičemž všechny záznamy v tabulce vždy obsahují stejný počet atributů (tedy ukládají data po řádcích). To někdy vede k plýtvání pamětí z důvodu používání hodnot *null* tam, kde neexistuje žádná relevantní hodnota. Sloupcově orientované databáze proto jednoduše ukládají data po sloupcích. Díky tomuto způsobu ukládání je možné šetřit paměť, protože se lze vyhnout ukládání zbytečných hodnot (jako je *null*). Jednoduše se daný sloupec neuloží, pokud neobsahuje hodnotu. Každá položka ve sloupcově orientované databázi je dvojice klíč-hodnota. Záznam je množina hodnot všech položek se stejným klíčem. Tento klíč je analogicky s relačními databázemi nazýván primární klíč.

  Příklad databází: MariaDB, Hbase.

- **Dokumentové databáze**

  Dokumentové databáze místo ukládání dat do řádků nebo sloupců provádí ukládání do dokumentů. Dokumentem však v tomto případě není myšlen klasický počítačový soubor (i když ten může být také uložen), ale volně uspořádaná množina položek typu klíč hodnota. Typicky je dokument typu JSON. Dokumentové databáze pak umožňují vyhledávání nejen na základě klíče, ale i na základě obsahu jednotlivých dokumentů.

  Příklad databází: MongoDB, CouchDB.

- **Grafové databáze**

  Grafové databáze sice autor zmiňuje, ale jejich samotný koncept již dále nerozvádí. Nicméně podle [24] jsou v grafové databázi data ukládána do uzlů grafu a vzájemné relace jsou vyjádřeny pomocí hran grafu. Tato architektura umožňuje velice snadné zavádění relací a navíc lze při práci s daty využít nejrůznější grafové algoritmy.

  Příklad databází: Neo4j.

3.3.1 Databáze časových řad

Databáze časových řad (angl. *Time Series Database*, zkr. TSDB) jsou podle [18] speciálním druhem NoSQL databází určené k ukládání údajů proměnlivých v čase. Všechna data jsou při zápisu opatřena časovým razítkem. Databáze pak umožňuje nad uloženými daty provádět dotazování pomocí specializovaného jazyka. TSDB jsou nejčastěji využívány právě pro ukládání měření senzorů, aby bylo možné u naměřených hodnot zpětně kontrolovat jejich vývoj. V listopadu 2022 mezi nejpoužívanější[5] TSDB patří InfluxDB, Kdb+, Graphite a Prometheus.

3.4 Ukládání a zpracování KPI

Data, která slouží ke zpracování KPI mohou být velmi odlišných druhů a podle toho budou ukládána v příslušných typech databází. Systém pro zpracování KPI tedy musí být na tato různorodá data připraven.

Při implementaci takového systému by se tedy mělo postupovat následujícími kroky:

1. **Definovat KPI** – Slovem „definovat“ není myšleno jen vytvářet zcela nové KPI, ale především jasně stanovit cíle a následně vybrat takové, které budou tyto cíle správně monitorovat. Konkrétních KPI pak už existuje celá řada, jak bylo demonstrováno v kapitole 2, a je tedy možné jen vybrat ty vyhovující. Eckerson [12] v tomto ohledu doporučuje použít metodu „brainstorming“ s nejvyšším vedením.

2. **Provést průzkum dostupných dat** – tzn. zjistit, jaká data budou pro vyhodnocování k dispozici. Mohou to být různé databázové systémy s různou strukturou ukládaných dat. Například informace o probíhajících městských projektech budou nejspíše uložené v relační databázi, zatímco data ze senzorů budou ukládána v TSDB. Různé statistické údaje, jako je věkové rozložení populace, zase budou spíše ve formátu CSV nebo XML v dokumentové databázi[6].

3. **Definovat vyhodnocovací proces** – následně je potřeba vytvořit proces převodu dostupných dat na zvolené KPI. Tento proces je individuální pro každé KPI, ale běžné skládá z těchto fází:

   (a) čištění dat – odstranění neúplných, nesmyslných nebo duplicitních dat,
   (b) integrace dat – spojení dat z několika zdrojů do jednoho místa,
   (c) transformace dat – agregace, normalizace, diskretizace numerických hodnot, transformace kategorických dat a další úlohy vedoucí vytvoření výsledné hodnoty KPI.

   Bližší informace o možných operacích s daty jsou dostupné například v knize [14].

4. **Integrovat výstup vyhodnocovacího procesu** – neboli jakým způsobem se bude reagovat na výsledné hodnoty. Je možné, že se budou pouze zobrazovat na nějaké řídící ploše nebo budou na základě těchto hodnot prováděny jiné automatické úkony.

Konkrétní způsob zpracování popisuje např. článek [28], ve kterém autoři prezentují metodu vyhodnocení pro energetické a enviromentální problémy. Tuto metodu následně demonstrují na skutečných městech.

[5] https://db-engines.com/en/ranking/time+series+dbms

[6] Právě data o věkovém rozložení populace jsou v tomto formátu volně dostupná na stránkách Českého statistického úřadu

Kapitola 4

Analýza aktuálního řešení

Cílem této diplomové práce je optimalizovat aktuální řešení zpracování KPI v projektu Smart City. Jedná se o projekt vyvíjený českou firmou Logimic[1], která se specializuje na poskytování cloudových IoT řešení pro města a obce, průmysl, zdravotnictví a další oblasti. Firma má pro projekt Smart City vytvořenou aplikaci se serverless architekturou, která je implementovaná v jazyce TypeScript. Tato aplikace využívá dva databázové systémy: relační databázi PostgreSQL a databázi časových řad InfluxDB. Komunikace s relační databází probíhá prostřednictvím knihovny TypeORM. Pro hosting je využíván systém AWS Lambda.

Tato kapitola nejprve v podkapitolách 4.1 a 4.2 seznamuje s projektem Smart City a řešeným problémem. V podkapitole 4.3 popisuje výkonnostní analýzu aktuálního řešení. V podkapitolách 4.4 a 4.5 pak podrobně rozebírá aktuální stav řešení a jeho nedostatky.

4.1 Struktura projektu Smart City

Smart City slouží k propojení menších IoT systémů na úrovni budov, objektů, městských částí či jiné úrovni dle aktuální potřeby. Součástí každého z těchto IoT systémů je pak několik IoT zařízení (senzorů), které disponují řadou parametrů, které mohou být statického (typ zařízení, geografické souřadnice) nebo dynamického (síla signálu, míra nabití baterie) typu. Tyto senzory sbírají data, která následně pod záštitou svého IoT systému posílají do databáze PostgreSQL. Nad informacemi v databázi jsou pak prováděny výpočty KPI. Ty jsou prezentovány uživateli (správci) prostřednictvím webové aplikace. Na obrázku 4.1 je příklad 3 výstupů (KPI detektoru kouře, chytrých pastí a chytrých dveří a oken).

Obrázek 4.1: KPI v systému Smart City

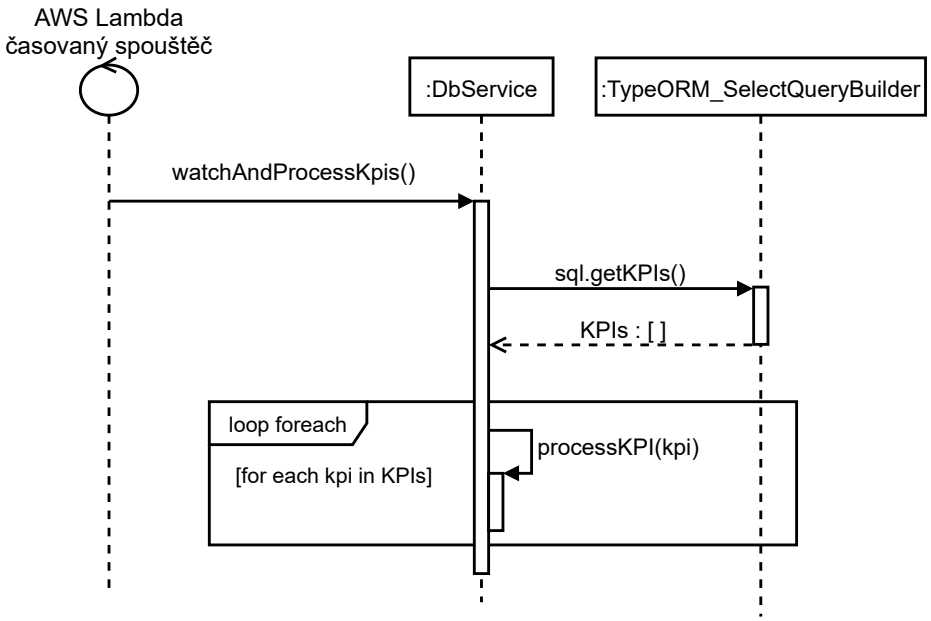
[1] https://www.logimic.com

Uživatel má tedy například možnost ihned vidět, že všechny chytré detektory kouře mají nabité baterie, stabilní signál a nehlásí žádné nebezpečí. Podobných výstupů se v celém Smart City nachází velké množství[2] a je potřeba zajistit pravidelnou aktualizaci hodnot, aby odpovídaly aktuálnímu stavu senzorů. Pravidelné vyhodnocení KPI provádí aplikace *lgmcOrmDbApi*.

4.2 Vyhodnocovací aplikace

Platforma Smart City využívá serverless architekturu. Tu [21] popisuje jako jednu z možných služeb, které může nabízet poskytovatel cloudového hostingu. Alternativami jsou pak *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), *Software as a Service* (SaaS) a *Containers and Functions as a Service* (FaaS). Bližší informace k těmto službám jsou dostupné v tomto článku. Využití serverless architektury znamená, že zdrojový kód je implementován ve formě speciálních *funkcí*. Ty jsou pak z hostujícího prostředí (AWS Lambda) volány prostřednictvím *spouštěčů* (angl. *triggers*). Po skončení funkce pak mohou být uvolněny její prostředky. Spouštěč může reagovat na různé podněty: manuální spuštění, časovač, přijetí HTTP požadavku, nahrání souboru a mnoho dalších. Výstupem spouštěče je tedy zavolání programátorem definované funkce. Tyto funkce mohou být navíc rozděleny do více aplikací (balíků funkcí), což vede k přehlednějšímu zdrojovému kódu.

Jednou z aplikací je zmíněná `lgmcOrmDbApi`. V této aplikaci je implementována funkce *watchAndProcessKpis*, která provádí aktualizaci hodnot KPI. Tato funkce je v pravidelných intervalech volána časovačem z prostředí AWS Lambda (nicméně může být volána i jiným způsobem). V rámci této funkce pak dochází k dotazu na databázi, který vrátí seznam všech KPI, které mají být zpracovány. Na každý KPI je pak volána funkce *processKPI*, která provádí další dotazy na databázi (např. získává data ze senzorů relevantních pro daný KPI). Průběh této funkce je znázorněn sekvenčním diagramem na obrázku 4.2.



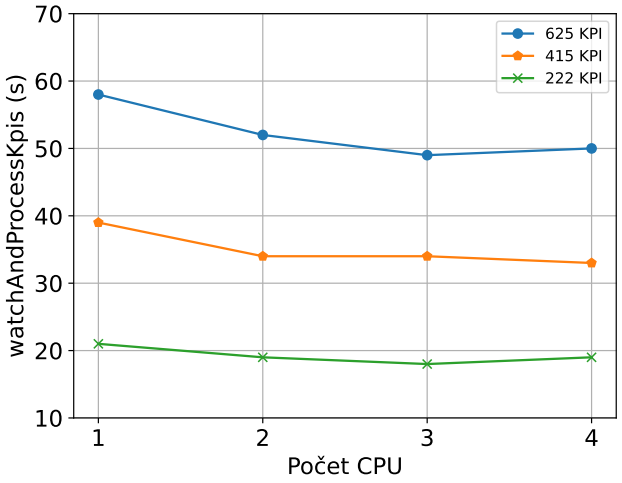
Obrázek 4.2: Zjednodušený sekvenční diagram funkce `watchAndProcessKpis`

[2] https://www.logimic.com/smart-city-dashboards/

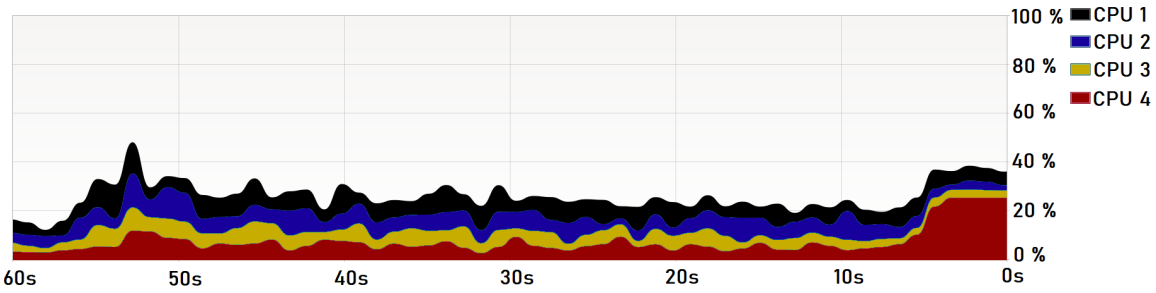
4.3 Výkonnostní analýza

V rámci bakalářské práce [17] byl sestaven nástroj, který provádí zátěžové testování systému v produkčním nasazení. Tento nástroj simuluje existenci IoT senzorů a generuje požadavky na systém podle předdefinovaných scénářů. Podle výsledku testu č. 3 z této práce se při vysokém zatížení systému nestíhá funkce `watchAndProcessKpis` dokončit před začátkem dalšího plánovaného volání. Zatížení systému v systému odpovídá 700 senzorům. Interval volání funkce je přitom dlouhých 5 minut. Podle vyhodnocení tohoto testu je příčinou málo výkonný hardware platformy AWS a také způsob implementace této funkce. Na základě tohoto vyhodnocení bylo firmou rozhodnuto, že je nutné provést její optimalizaci. Pouhý cíl zrychlení je však sám o sobě z pohledu dosažitelné optimalizace zavádějící, protože hraje roli použitý hardware, který musí být schopen obsluhovat i jiné požadavky.

Proto bylo vytvořeno vlastní testovací prostředí. Toto prostředí je podrobně popsáno v podkapitole 7.1. V tomto prostřední bylo provedeno měření doby výpočtu funkce `watchAndProcessKpis` s různými počty jader CPU a počty KPI v databázi. Výsledky měření jsou zobrazeny v grafu na obrázku 4.3. Z grafu je možné vyčíst, že se zvyšujícím se počtem dostupných jader nedochází k časovému zlepšení. To je způsobeno tím, že aplikace nevyužívá všechna dostupná jádra, jak ukazuje graf na obrázku 4.4. Vytížení jednoho jádra na konci měření je způsobeno pouze aktivním čekáním na další iteraci. Optimalizovaná funkce tedy musí dobře využívat všechny dostupné výpočetní prostředky.



Obrázek 4.3: Doby běhů funkce `watchAndProcessKpis`

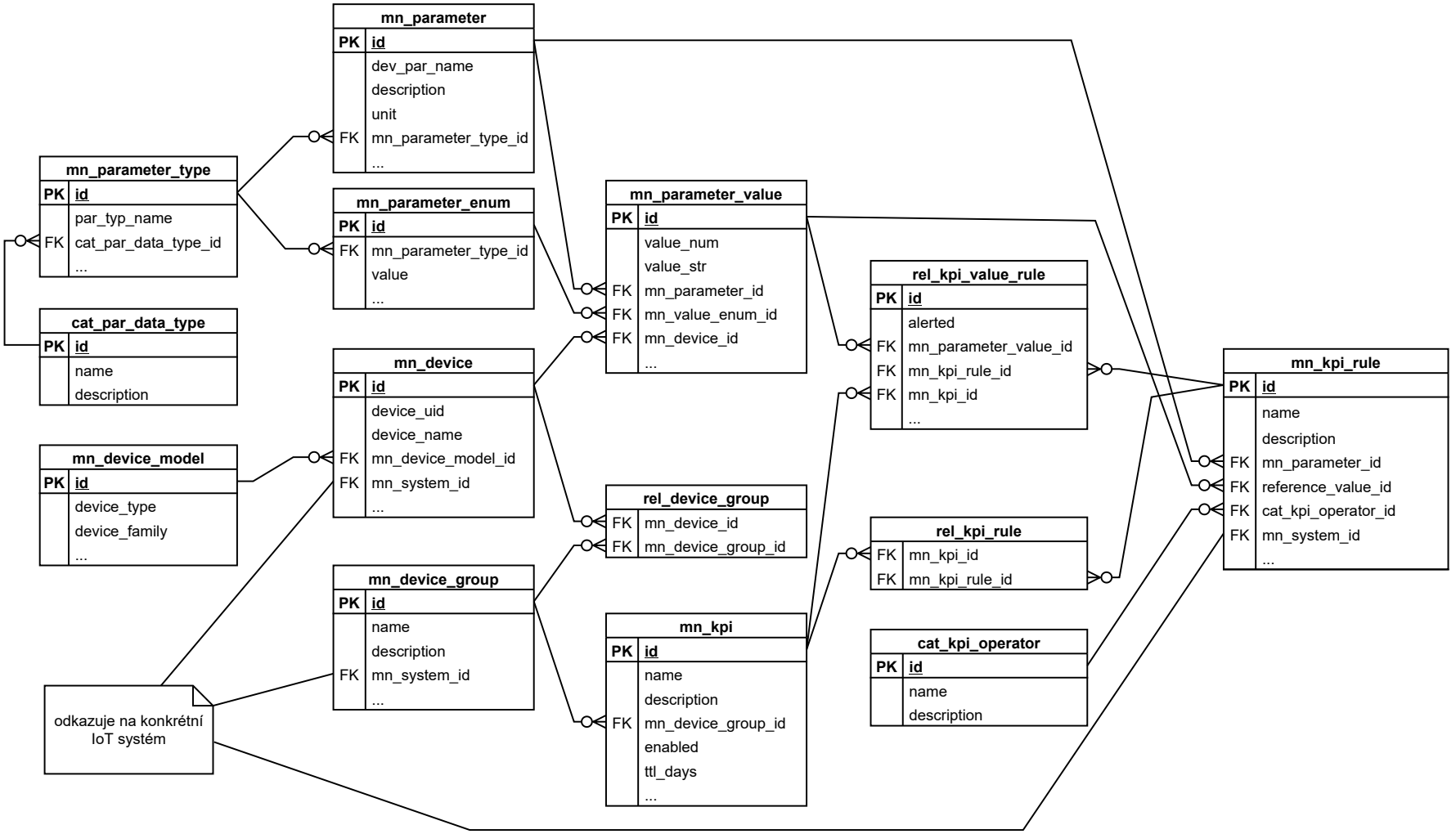


Obrázek 4.4: Graf využití CPU během jedné iterace funkce `watchAndProcessKpis`

4.4 Datový model

Platforma Smart City pro ukládání dat využívá relační databázi PostgreSQL. Tato podkapitola popisuje část databáze, která souvisí s výpočtem KPI představených v předchozích podkapitolách. Tento datový model je definován ve zdrojovém kódu pomocí syntaktických konstrukcí knihovny TypeORM, které byly popsány v sekci 3.2.3. Následující seznam popisuje jednotlivé tabulky z ER diagramu na obrázku 4.5 – počínaje v diagramu vlevo nahoře:

- **cat_par_data_type** – datové typy hodnot parametrů senzorů, které jsou používány v systému (number, string, enum).
- **mn_parameter_type** – typy hodnot parametrů (nikoli datový typ) – např. kapacita a typ baterie, procenta, stav otevřeno/zavřeno, RSSI, kvalita ovzduší, měření vzdálenosti a další,
- **mn_parameter_enum** – seznam možných hodnot pro parametry, jejichž datový typ hodnoty (`mn_parameter_type`) je nastaven jako enum (`cat_par_data_type`) – např. AA, AAA, C a D jako typy baterií, stav otevřeno, stav zavřeno, nízká jako stav nabití baterie a další,
- **mn_parameter** – konkrétní parametry, kterými mohou disponovat jednotlivé senzory – např. baterie, teploměr nebo dálkoměr,
- **mn_device_model** – modely senzorů, které mohou být v systému – např. detektor kouře v1.0, odpadkový koš v2.3,
- **mn_device** – instance senzoru odpovídající záznamu v `mn_device_model` (tedy již konkrétní zařízení) – např. odpadkový koš v2.3 s konkrétním ID na daných geografických souřadnicích,
- **mn_parameter_value** – obsahuje konkrétní hodnoty konkrétních parametrů (`mn_parameter`) všech senzorů v systému
- **mn_device_group** – seznam skupin senzorů (`mn_device`) – seskupování je `M:N` a probíhá přes tabulku `rel_device_group`
- **rel_device_group** – propojovací tabulka mezi `mn_device` a `mn_device_group`
- **mn_kpi** – obsahuje definice KPI, které jsou později zobrazeny uživateli – např. baterie detektoru kouře, signál detektoru kouře, signál pastí na hlodavce a další (viz. obrázek 4.1),
- **mn_kpi_rule** – obsahuje pravidla, jak se mají jednotlivé KPI vyhodnocovat
- **cat_kpi_operator** – operátory využitelné při výpočtu KPI (větší, menší, rovno, ...)
- **rel_kpi_rule** – propojuje KPI s příslušným pravidlem pro vyhodnocení (Vztah `M:N` tedy v datovém modelu znamená, že pro dané KPI může existovat více pravidel pro vyhodnocení. Konkrétní záznamy v tabulce však nasvědčují tomu, že se jedná o vztah `1:N` – tedy dané KPI je vyhodnocováno jen na základě jednoho pravidla)
- **rel_kpi_value_rule** – obsahuje výsledky zpracování KPI

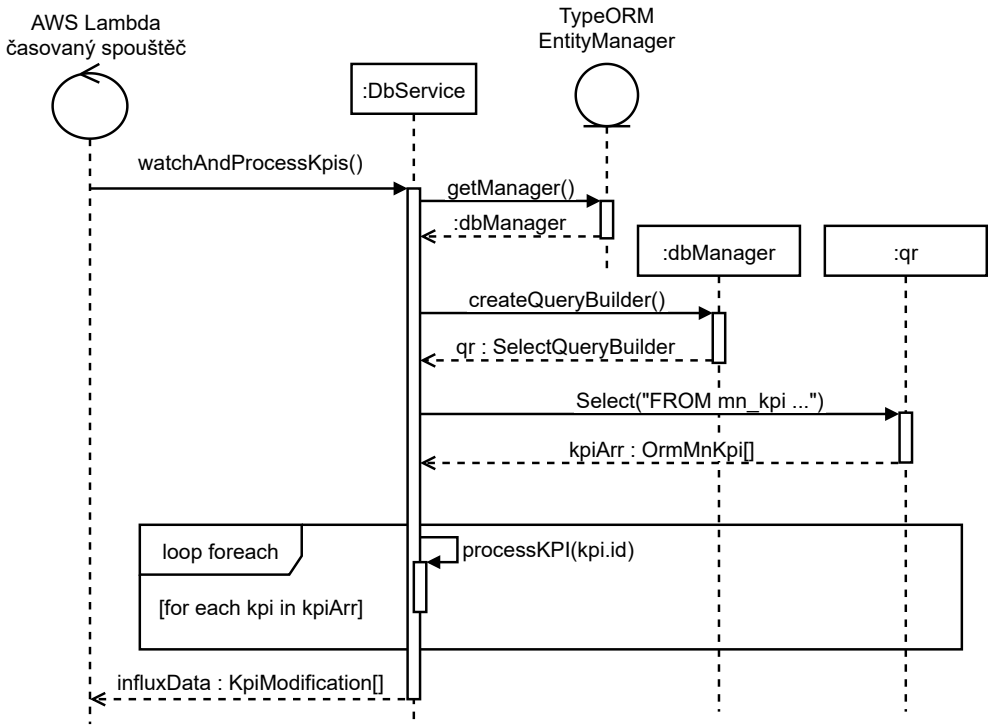


Obrázek 4.5: ER diagram části databáze firmy Logimic

4.5 Analýza funkce zpracování KPI

Náplní této diplomové práce je úprava již zmíněné funkce `watchAndProcessKpis`. Princip této funkce byl popsán v podkapitole 4.2, ale tato podkapitola rozebírá funkci mnohem podrobněji a hledá nedostatky, jejichž odstranění povede k její optimalizaci.

Volání této funkce provádí v pravidelných intervalech časovaný spouštěč platformy AWS Lambda. Funkce pro komunikaci s databází využívá knihovnu TypeORM. Z této knihovny se pro správu databáze používá instance třídy *EntityManager* (instance je pak nazvána `dbManager`). Pomocí `dbManager` je vytvořena instance typu `SelectQuerryBuilder` nazvaná `qr`, která umožňuje manuální skládání SQL dotazů ve zdrojovém kódu. Pomocí `qr` jsou získány všechny aktuální KPI z tabulky `mn_kpi`. Pro každý získaný KPI je následně sekvenčně volána funkce `processKPI`, která přijímá parametr ID daného KPI, který má být zpracován. Výstupem `processKPI` jsou pak informace o změnách provedených v tabulce `rel_kpi_value_rule` (tedy tabulce s výsledky zpracovaných KPI). Tyto informace jsou shromážděny ze všech iterací a následně jsou funkcí `watchAndProcessKpis` vráceny jako výsledek volání. Tato data jsou pak ukládána do InfluxDB – to už ale `watchAndProcessKpis` neřeší. Průběh celé funkce je znázorněn sekvenčním diagramem na obrázku 4.6.



Obrázek 4.6: Sekvenční diagram funkce `watchAndProcessKpis`

4.5.1 Analýza funkce processKPI

Funkce `processKPI` tedy slouží ke zpracování jednoho konkrétního KPI z tabulky `mn_kpi`. Zdrojem pro zpracování jsou data ze senzorů uložená v databázi. Postup odpovídá principu z obrázku 2.1 v podkapitole 2.3. Pro přístup k databázi se opět používá instance `SelectQuerryBuilder`. Celý průběh funkce je pak znázorněn sekvenčním diagramem na obrázku 4.7. Nyní následuje slovní popis kroků, ze kterých se tato funkce skládá:

1. Je zahájena transakce zpracování daného KPI.
2. **qr** – SELECT – Dochází k opětovnému načtení KPI z databáze, tentokrát včetně informací o dané skupině, pro kterou je KPI vyhodnocováno (z `mn_device_group`) a pravidlu pro vyhodnocení (`mn_kpi_rule`). Bylo by vhodné tento dotaz více sesynchronizovat s počátečním dotazem pro získávání všech KPI, aby nedocházelo ke zbytečnému dvojímu načítání stejných dat.
3. **qD** – DELETE – Je prohledána tabulka s hodnotami KPI (`rel_kpi_value_rule`) a pro zpracovávané KPI jsou z ní vymazány všechny záznamy, u kterých chybí odpovídající pravidlo z tabulky `mn_kpi_rule` (tedy odstraní záznamy, pro které bylo odstraněno vyhodnocovací pravidlo).
4. **qD1** – DELETE – Znovu je prohledána tabulka s hodnotami KPI (`rel_kpi_value_rule`) a pro zpracovávané KPI jsou vymazány všechny záznamy, u kterých tentokrát chybí odpovídající propojení se skupinou zařízení v tabulce `rel_device_group` (tedy odstraní záznamy, pro které byla odstraněna skupina zařízení).
5. Následuje cyklus pro všechna pravidla pro vyhodnocení (rule) zpracovávaného KPI. Tento cyklus je zde ovšem zbytečný, protože KPI má v praxi vždy jen jedno pravidlo pro vyhodnocení (viz. popis datového modelu v podkapitole 4.4). Cyklus se aktuálně skládá z těchto kroků:
   (a) **q2** – SELECT – Jsou získány záznamy s hodnotami všech parametrů od všech senzorů ze stejné skupiny zařízení jako je zpracovávané KPI, pro které platí následující:
      - neexistuje u nich propojení na pravidlo vyhodnocení KPI (přes tabulku `rel_kpi_value_rule`),
      - toto spojení lze odvodit, protože spadají pod dané KPI přes skupinu zařízení (`rel_device_group`).

   (b) **qv** – INSERT – Pro každou takto nepropojenou hodnotu je zaslán dotaz, aby bylo propojení v tabulce `rel_kpi_value_rule` vytvořeno. Zde však dochází k cyklickému vkládání vždy jen po jedné hodnotě. Místo toho by bylo vhodné vytvořit jeden dotaz, který všechna propojení vloží současně.

   (c) **q3** – SELECT – Z doplněné tabulky `rel_kpi_value_rule` je provedeno opětovné získání všech záznamů odpovídající zpracovávaném KPI. Zde se jedná o redundantní dotaz neboť dochází k načítání informací, které už jsou na straně serveru (byly ukládány do databáze v předchozím příkazu `qv`).

   (d) Nyní je pro všechny načtené hodnoty parametrů provedeno příslušné porovnání s referenčními hodnotami (tedy dle načtených pravidel). Během vyhodnocování je ukládáno, které hodnoty odpovídaly referenčním hodnotám a které ne.

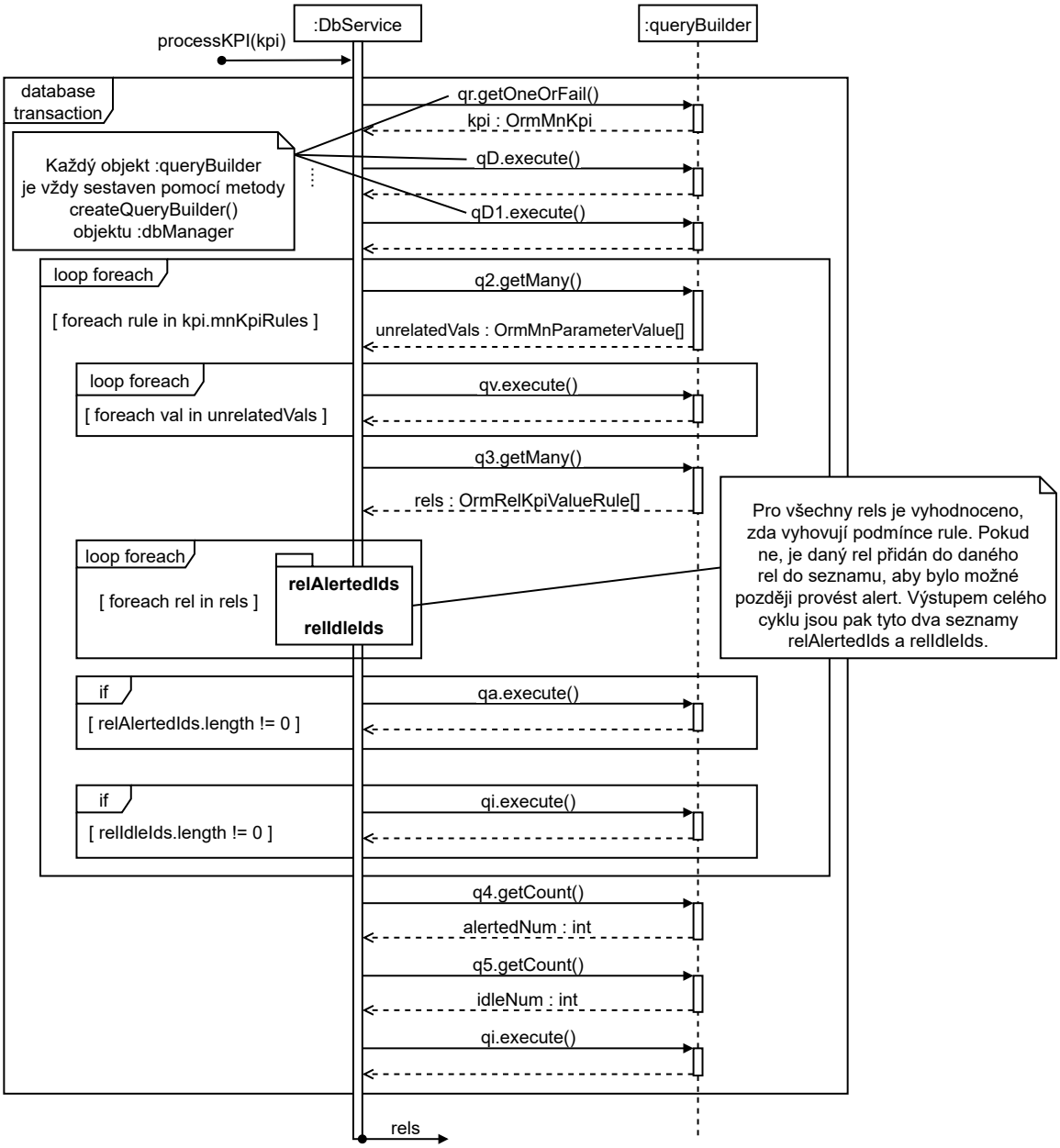
   (e) **qa** – UDPATE – Do `rel_kpi_value_rule` je uloženo, které hodnoty neodpovídaly referenčním hodnotám (jsou `alerted`).

   (f) **qi** – UPDATE – Do `rel_kpi_value_rule` je uloženo, které hodnoty odpovídaly referenčním hodnotám (jsou `idle`).
6. **q4** – SELECT – Z `rel_kpi_value_rule` je získán počet všech `alerted` hodnot. Tento dotaz je zjevně redundantní neboť načítá data, která jsou dostupná na straně serveru. Počet `alerted` hodnot lze zjistit zavedením čítače do cyklu s vyhodnocením pravidel.

7. **q5** – SELECT – Z `rel_kpi_value_rule` je získán počet všech idle hodnot. Stejně jako dotaz `q4` je tento dotaz redundantní a lze nahradit zavedením čítače.

8. **qi** – UPDATE – Do tabulky `mn_kpi` jsou u zpracovávaného KPI vloženy počty hodnot `alerted` a `idle` (`q4` a `q5`).

9. Transakce je ukončena.



Obrázek 4.7: Sekvenční diagram funkce `processKPI`

V této funkci je jedna velká transakce využívána zbytečně. Ve funkci `processKPI` jsou prováděny dotazy pouze v rámci jednoho KPI. Jednotlivá volání funkce tedy upravují jiné záznamy a nehrozí tak vytvoření nekonzistentní databáze. Za zvážení stojí pouze zamknutí databáze pro hromadné čtení hodnot parametrů senzorů, které by mohly být jednotlivými senzory změněny. Z principu využití funkce je však možné uvažovat, že pokud by k takové změně došlo, jednoduše by se změna promítla uživateli až v další iteraci volání funkce `watchAndProcessKpis`.

Zároveň je ze sekvenčního diagramu patrné, že pro komunikaci s databází jsou SQL dotazy sestavovány manuálně. Nedochází k správnému využívání principu ORM (který byl popsán v podkapitole 3.2), a to i když je využívána knihovna TypeORM. To má mimo jiné také za následek zhoršenou čitelnost zdrojového kódu.

V neposlední řadě je z prováděných dotazů patrné, že dotazy `qD`, `qD1`, `q2` a `qv` slouží k zajištění konzistentních dat v databázi. Zasílání těchto dotazů až ve funkci `processKPI` naznačuje, že se během běhu aplikace dostává databáze do stavu, který není (minimálně pro funkci `processKPI`) očekávaný. Z toho důvodu by provádění těchto dotazů mělo být přeneseno do jiných částí aplikace.

4.5.2 Analýza závislostí dotazů

Velmi nákladnou činností, která je ve funkci `watchAndProcessKpis` prováděna je síťová komunikace – přesněji komunikace s databází. Ze sekvenčního diagramu je zřejmé, že všechny dotazy probíhají sériově – tzn. funkce vždy čeká na odpověď z databáze, než provede další dotaz. K čekání však dochází i v případě, kdy existuje jiná, na prováděném dotazu nezávislá činnost, která mezitím může být vykonávána. To má za důsledek nedostatečné využívání výpočetních zdrojů a plýtvání časem. Pokud má dojít ke zlepšení této situace, je potřeba, aby funkce místo pouhého čekání vykonávala jinou užitečnou činnost – například paralelně zahájila další dotazy. Paralelní provádění však není možné provést v momentě, kdy jsou dotazy závislé.

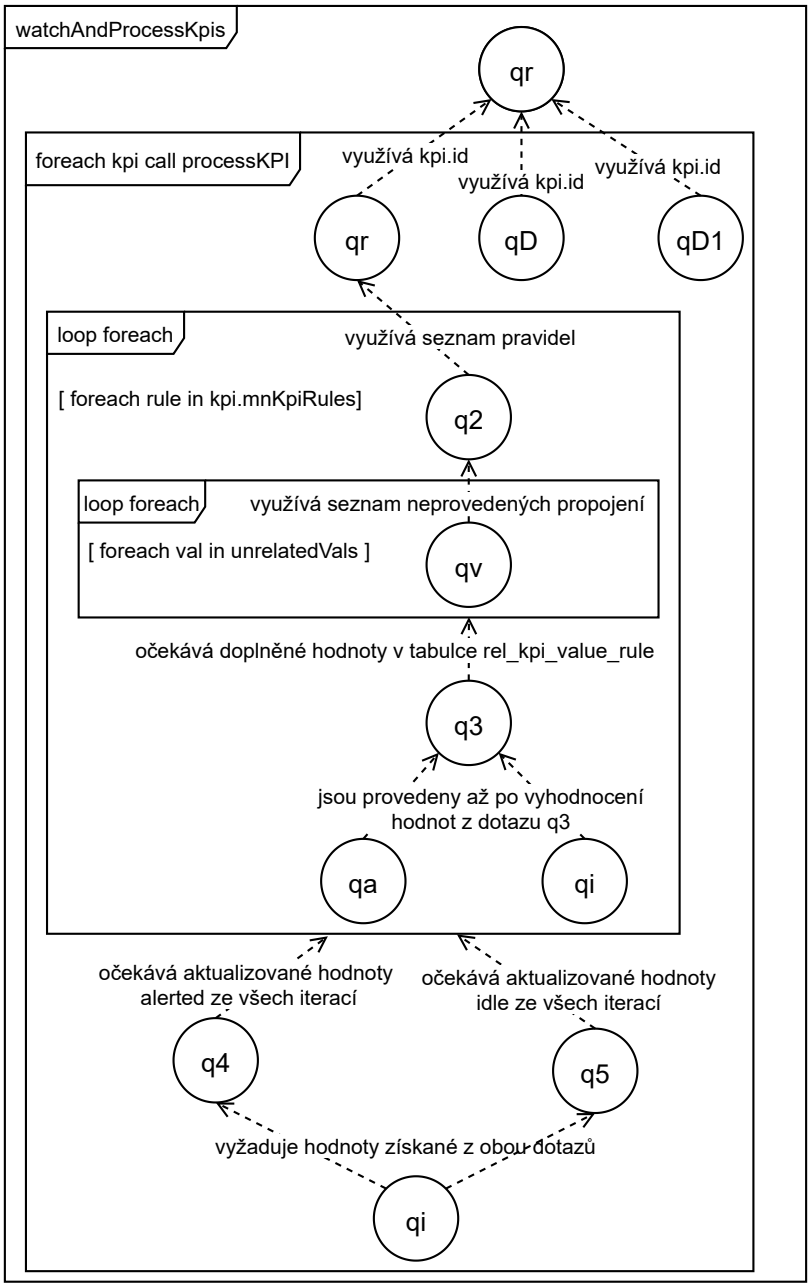
Definice závislosti SQL dotazů: *SQL dotaz Q2 je závislý na SQL dotazu Q1 právě tehdy, když je potřebné dokončení provádění dotazu Q1 ještě před provedením dotazu Q2.* Pro určení typů závislostí SQL dotazů byl využit koncept jaký je u závislostí instrukcí na úrovni procesoru. O těchto závislostech je možné se dočíst například v [5]. Mezi dotazy tedy mohou existovat tři typy závislostí:

- RAW (*read after write*) – Závislost typu RAW vzniká právě tehdy, když Q2 je závislý na Q1 tím, že Q2 čte záznamy, které Q1 předtím modifikuje.
- WAR (*write after read*) – Závislost typu WAR vzniká právě tehdy, když Q2 je závislý na Q1 tím, že Q2 modifikuje záznamy, u kterých si ale Q1 musí nejprve přečíst původní hodnoty.
- WAW (*write after write*) – Závislost typu WAW vzniká právě tehdy, když Q2 je závislý na Q1 tím, že Q2 modifikuje záznamy, které předtím modifikuje i Q1 a záměna těchto dvou dotazů bude mít za následek jiný výsledný stav databáze.

Výsledek analýzy závislostí dotazů funkce `processKPI` je znázorněn grafem závislostí na obrázku 4.8. Z grafu je patrné, že se ve funkci nachází několik dotazů, které jsou **nezávislé**:

- dotazy `qr`, `qD` a `qD1` jsou vzájemně nezávislé,
- dotazy `qa` a `qi` v rámci iterace cyklu pro dané pravidlo jsou vzájemně nezávislé,

- dotazy `q4` a `q5` jsou po skončení cyklu nad pravidly vzájemně nezávislé,
- dotazy `qv` napříč iteracemi cyklu nad nepropojenými hodnotami jsou vzájemně nezávislé,
- dotazy napříč iteracemi vnitřního cyklu nad pravidly jsou vzájemně nezávislé,
- dotazy napříč iteracemi volání funkce `processKPI` jsou vzájemně nezávislé.



Obrázek 4.8: Diagram závislostí SQL dotazů

4.6 Shrnutí analýzy

Analýza aktuálního řešení odhalila nedostatky v následujících oblastech:

- Výkon – zpracování probíhá sekvenčně, což omezuje aplikaci ve využívání více vláken a znemožňuje úplné využívání dostupných výpočetních prostředků.
- Datový model – datový model neodpovídá skutečným požadavkům. Záznamy v tabulce `rel_kpi_rule` nasvědčují tomu, že vztah mezi *mn_kpi_rule* a `mn_kpi` je spíše `1:N` místo `M:N`.
- Práce s databází – funkce na několika místech provádí načítání dostupných nebo odvoditelných dat. Také dochází k provádění dotazů v rámci cyklu, čímž dochází ke generování nadbytečného množství dotazů. Práce s databází je navíc prováděna manuálním sestavováním SQL dotazů, což je v rozporu s principy popsanými v 3.2.

Na řešení těchto nedostatků se zaměřuje následující kapitola.

Kapitola 5

Návrh

V kapitole 4 bylo popsáno několik problémů aktuální implementace zpracování KPI. Tato kapitola navrhuje způsob jejich řešení, které lze rozdělit na několik částí podle objevených problémů. Datový model neodpovídá reálným datům a je vhodné jej upravit. Způsob provedení této změny je popsán v podkapitole 5.1. Dále se v kódu nachází zbytečné dotazy na SQL databázi, protože načítají data, která jsou dostupná na serveru. Řešení tohoto problému je rozebráno v podkapitole 5.2. Posledním problémem je sekvenční provádění a nedostatečné využívání dostupných výpočetních prostředků. Zde je potřeba zavést paralelní vykonávání. Návrh řešení je popsán v podkapitole 5.3. Na závěr je v podkapitole 5.4 proveden odhad očekávaných výsledků.

5.1 Úprava datového modelu

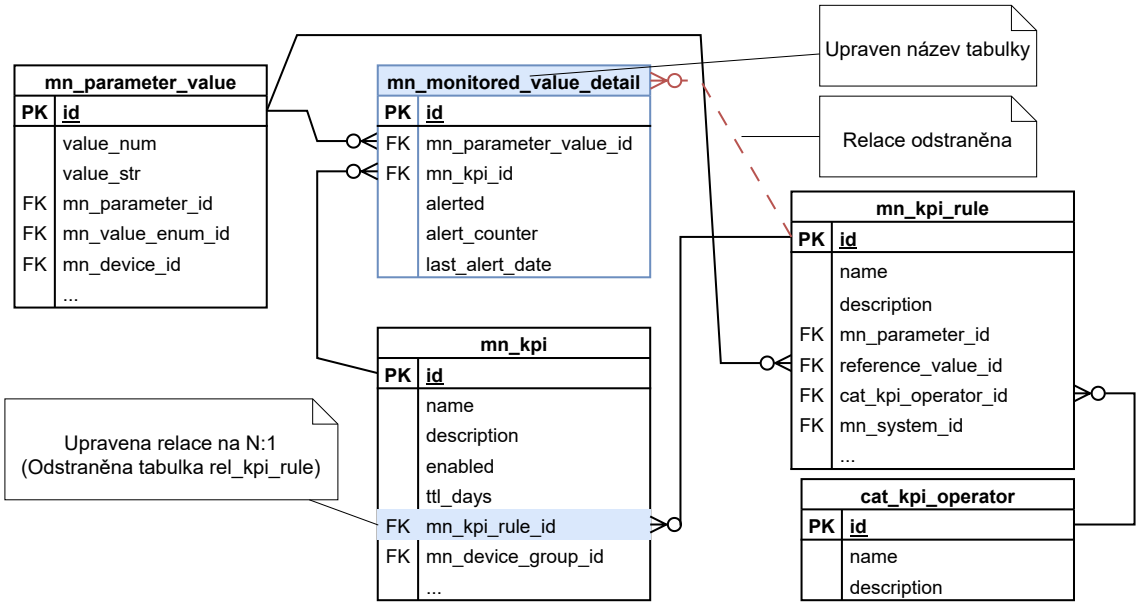
Z dat uložených v databázi vyplývá, že vztah mezi entitami `mn_kpi_rule` a `mn_kpi` je spíše vztah typu `1:N` místo aktuálního `M:N`. Úpravu kardinality relace lze udělat tak, že bude odstraněna tabulka `rel_kpi_rule` a do tabulky `mn_kpi` bude přidán atribut obsahující ID příslušného vyhodnocovacího pravidla (z tabulky `mn_kpi_rule`). Zároveň je možné z tabulky `rel_kpi_rule` odstranit relaci `1:N` (atribut `mn_kpi_rule_id`), protože dané vyhodnocovací pravidlo lze odvodit na základě relace s `mn_kpi`.

Tento skutečnosti neodpovídající datový model se však už nachází v produkční databázi. V takové fázi už není možné jednoduše upravit datový model změnou definice ve zdrojovém kódu. Lze totiž očekávat, že z bezpečnostních důvodů bude automatická propagace změn do databáze vypnuta. Řešení v takové situaci je vytvoření tzv. *migrace*. Migrace je podprogram, který zajišťuje shodu datového modelu ve zdrojovém kódu a na straně databáze pomocí dvou funkcí, které bývají typicky nazývány `up` a `down`. Postup přechodu na novou verzi databáze (aktualizaci modelu) zajišťuje funkce `up`. Funkce `down` pak slouží k návratu do původního stavu před provedením funkce `up`. Funkci `down` však není nutné implementovat, pokud se nepředpokládá potřeba návratu. Pro návrat databáze do původního stavu během ověřování implementace migrace lze jednoduše použít zálohu dané databáze. Zdrojový kód migrace může být buď celý vytvořený programátorem nebo může být automaticky vygenerován úpravou definice datového modelu. Nicméně i při automatickém vygenerování pak programátor musí zkontrolovat znění migrace před jejím aplikováním. Další informace o implementaci migrací jsou dostupné v [23] nebo v [39].

V případě úpravy aktuálního datového modelu lze očekávat, že se bude migrace skládat především z následujících 3 kroků:

1. přidání sloupce do tabulky `mn_kpi` pro FK vyhodnocovacího pravidla `mn_kpi_rule`,
2. přenos hodnot cizích klíčů z tabulky `rel_kpi_rule` do nově vytvořeného sloupce k příslušným záznamů v tabulce `mn_kpi`,
3. odstranění tabulky `rel_kpi_rule`.
4. odstranění relace mezi `rel_kpi_value_rule` a `mn_kpi_rule`

Zároveň bude přejmenována tabulka `rel_kpi_value_rule` na `mn_monitored_value_detail`, aby její název lépe vystihoval význam jejích záznamů. Uchovává detail monitorování hodnoty parametru daným KPI. Navrhované úpravy jsou zobrazeny na obrázku 5.1.



Obrázek 5.1: Navrhované úpravy datového modelu

5.2 Úprava dotazů na databázi

Jak již bylo popsáno v sekci 4.5.1, funkce `processKPI` provádí zbytečné dotazy na databázi. Možným řešením by bylo tyto zbytečné dotazy odstranit. Nicméně tento problém je částečně způsoben absencí ORM. V aktuálním kódu totiž dochází k manuálnímu sestavování SQL dotazů pomocí instance `SelectQuerryBuilder`. Tato instance sice pochází z knihovny TypeORM, ale její časté využívání je v rozporu s principy popsanými v podkapitole 3.2. Navíc jsou v tomto kódu střídavě využívány dva identicky definované datové modely (pojmenované *Entities.ts* a *OrmEntities.ts*, přičemž `OrmEntities.ts` je zřejmě zamýšlen pro knihovnu TypeORM a všechny jeho entity dědí z odpovídajících entit modelu `Entities.ts`). Ve funkci jsou však střídavě využívány oba modely, což má za následek nízkou čitelnost kódu. Pro programátora pak může být v takové situaci obtížné udržet si informaci o tom, jaká data již byla v daných částech kódu z databáze získána. Knihovna pro ORM by měla tuto věc řešit za něj, a podobným problémům tak předcházet.

Funkce `processKPI` navíc provádí dodatečnou činnost (dotazy `qD`, `qD1`, `q2` a `qv`), která však přímo nesouvisí s business logikou této funkce. Ta by měla pouze aktualizovat hodnoty v tabulce `rel_kpi_value_rule`, protože na tyto nové hodnoty čeká uživatel (kontrolující webovou aplikaci – viz. 4.1). Jakékoli další procesy, které tato funkce provádí (jako právě zmíněné dotazy) způsobují zpomalení a znepříjemňují tak využívání aplikace. Tuto nesouvisející činnost je tedy vhodné převést do jiných částí aplikace, které pak budou volány samostatně – popř. zvážit či je tato činnost vůbec v této podobě potřeba. U daného KPI by mělo být už při jeho vložení jasné, jak bude vyhodnocováno. Tato problematika je však už nad rámec této diplomové práce.

5.3 Paralelizace kódu

Velkým nedostatkem funkce `watchAndProcessKpis` je sekvenční provádění, což má také za následek nedostatečné využívání výpočetních zdrojů. Tento problém lze řešit zavedením asynchronního programování. Stávající kód je možné rozdělit na několik vzájemně výpočetně nezávislých bloků instrukcí – v tomto případě například podle analýzy závislostí na obrázku 4.8. Tyto nezávislé bloky pak lze provádět paralelně.

5.3.1 Asynchronní kód

Podle knihy [19] existují v jazyce TypeScript 3 způsoby, jak psát asynchronní kód.

- Prvním způsobem jsou tzv. *callback* funkce. Ty využívají možnosti, že funkci lze předat jako parametr jiné funkci a zároveň lze funkci vrátit (jako jakoukoli jinou hodnotu). Asynchronní kód je pak sestaven navazováním jednotlivých funkcí. Díky tomu pak může interpret před zahájením další funkce snadno přepínat mezi prováděnými funkcemi a přitom si zachovat kontext všech paralelních běhů. Nicméně využití tohoto způsobu u větších projektů vede k problému s udržitelností známému jako *callback hell* – tedy postupné zanořování se časem stane příliš nepřehledným.
- Novějším konceptem jsou tzv. *Promises* (doslovně přeložitelné jako „přísliby“). Tento koncept zavádí datový typ *Promise*, který obaluje libovolnou akci a přímo „slibuje“, že tato akce bude vykonána. Objekt typu `Promise` může nabývat 3 stavů (*pending* (akce čeká na zahájení nebo probíhá), *resolved* (akce je dokončena) a *rejected* (akce selhala) – v závislosti na dokončení akce), se kterými může programátor pracovat.
- Nejnovější verze jazyka TypeScript obsahují *asynchronní funkce*. Jedná se o pomocné syntaktické konstrukce, které jen usnadňují práci s objekty typu `Promise`. Pro práci s asynchronním kódem slouží klíčová slova *async* a *await*. Slovo `async` je přidáváno do definice hlavičky funkce, čímž automaticky její výsledek obaluje typem `Promise`. Slovo `await` automaticky odstraňuje obalující typ `Promise`, přičemž pokud daná akce ještě není dokončena, tak je automaticky provedeno čekání. Z pohledu vlákna se však nejedná o aktivní čekání, ale přepnutí kontextu mezi paralelními běhy. To znamená, že dokud není daná akce, na kterou bylo použito `await`, dokončena, tak vlákno vykonává jinou práci. Podrobnější popis použití je popsán v samotné knize.

5.3.2 Vícevláknové provádění

Převedení funkce `processKPI` na asynchronní funkci (a provedení dalších z toho vyplývajících úprav) ovšem celý problém zatím nevyřeší. Jazyk TypeScript provádí výpočet převážně

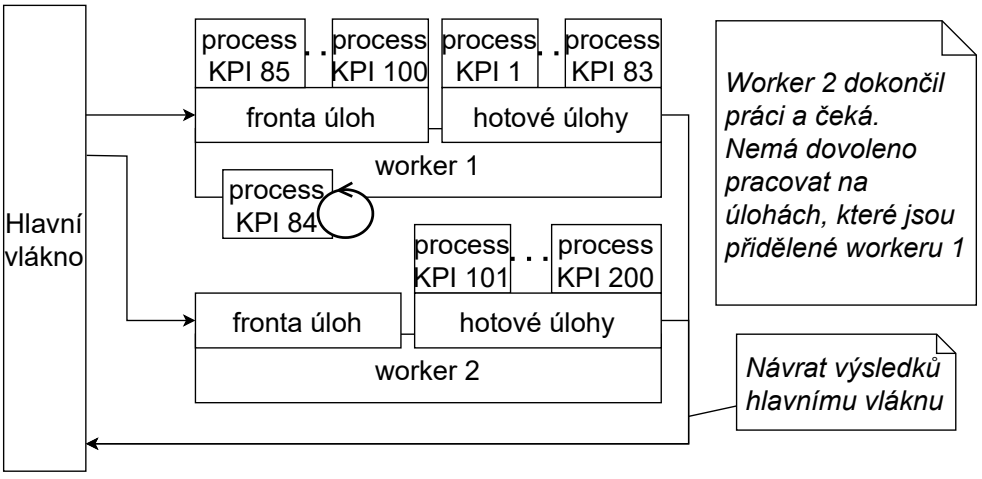
na jednom vlákně a tedy jen na jednom jádře (tzv. *singlethread*). Některé asynchronní funkce zabudované v jazyce TypeScript sice automaticky spouští pomocné paralelní vlákno (typicky práce s diskem, kryptografické operace nebo síťová komunikace), nicméně vytvoření asynchronní funkce nijak nezaručuje její spuštění na jiném vlákně. Pouze je vláknu umožněno přepínat kontext v rámci programu podle potřeby, jako například upřednostňovat funkce obsluhující uživatelské vstupy a výstupy nebo právě provádět jinou aktivitu místo čekání na provedení dotazu.

Ke spuštění více vláken slouží podle dokumentace [25] moduly *child process*, *cluster* a *worker threads*. Moduly `child process` a `cluster` jsou však podle popisu pro paralelní spouštění funkcí nevhodné a slouží k paralelizaci v jiných oblastech. Naopak modul `worker threads` je přímo určen k provádění výpočetně náročných úloh, aby nedocházelo k zatěžování hlavního vlákna. Tento modul umí vytvořit více paralelně pracujících vláken, přičemž každému takovému vláknu se říká *worker* (pracovní vlákno).

Činnost pracovního vlákna je vždy definována v samostatném souboru. Z hlavního vlákna je možné předávat parametry práce zasíláním zpráv. V aktuální aplikaci by tedy činností pracovního vlákna mohlo být provedení funkce `processKPI`. V takovém případě by pro každé provedení funkce bylo vytvořeno samostatné vlákno. To by ale znamenalo, že budou vytvořeny řádově stovky pracovních vláken – v systému však může současně běžet pouze tolik vláken, kolik je jader procesoru. Z toho důvodu je vhodné vytvořit menší počet vláken (maximálně tolik, kolik je jader) a jednotlivá volání `processKPI` mezi ně rozdělit.

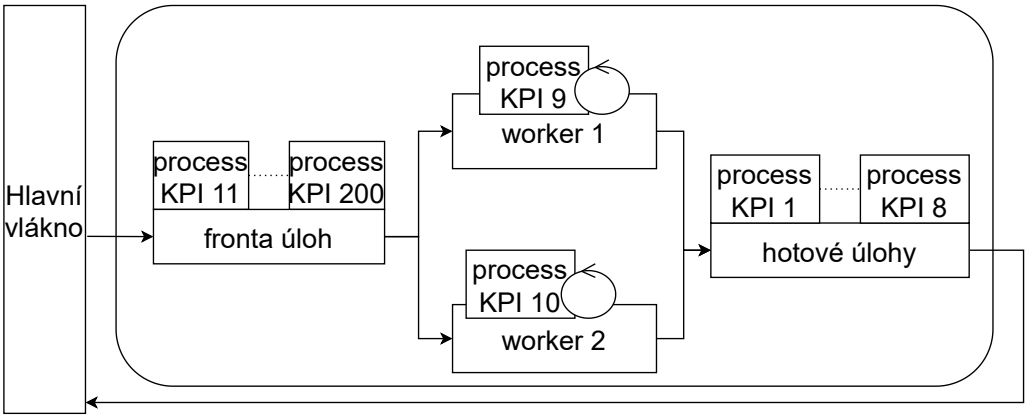
5.3.3 Dělení práce mezi vlákna

Jednoduchým způsobem by bylo statické rozdělení podle počtu pracovních vláken – např. pro 2 vlákna by z 200 KPI zpracovalo 1. vlákno KPI 1-100 a 2. vlákno KPI 101-200. Toto řešení by bylo v pořádku, pokud by každé volání `processKPI` bylo vždy stejné. V praxi se ale pro každé KPI liší počet parametrů, které musí být zkontrolovány. Některé iterace volání jsou z toho důvodu náročnější. V takovém případě hrozí, že by jedno z vláken mohlo svou práci dokončit dříve a skončit, zatímco by druhé by stále pracovalo. Například by tato situace pravidelně nastala, pokud by KPI 1-100 byly vždy složitější. Tato situace je znázorněná na následujícím obrázku 5.2.



Obrázek 5.2: Vizualizace nevyvážené zátěže pracovních vláken při statickém rozdělení práce

Tento problém lze vyřešit tak, že všechna volání `processKPI` budou umístěna do společné fronty odkud budou jednotlivými pracovními vlákny postupně odebírána a obsluhována. Vlastní implementace synchronizační logiky je však riskantní. Místo toho je možné využít některou z dostupných knihoven, která bude synchronizaci zajišťovat. Od takové knihovny se pak očekává, že si vytvoří *zásobu vláken* (angl. *worker pool*), kde budou vlákna vyčkávat na práci. V zásobě by jich měl být takový počet, jaký je dostupný počet jader procesoru. Nicméně by mělo být možné nastavit množství menší. Důvod k takovému snížení bude probrán v následující podkapitole 5.4. Hlavní vlákno pak musí mít možnost knihovně předat úlohu (typicky voláním specializované funkce). Knihovna pak tuto úlohu sama předá volnému pracovnímu vláknu a po jejím dokončení předá výsledek zpět hlavnímu vláknu. Princip funkce knihovny je znázorněn na obrázku 5.3



Obrázek 5.3: Vizualizace dynamického rozdělení práce

Velkou výhodou tohoto způsobu je navíc možnost měnit počet pracovních vláken bez jakékoli změny zdrojového kódu, jelikož veškerou správu zajišťuje daná knihovna.

5.4 Očekávané výsledky

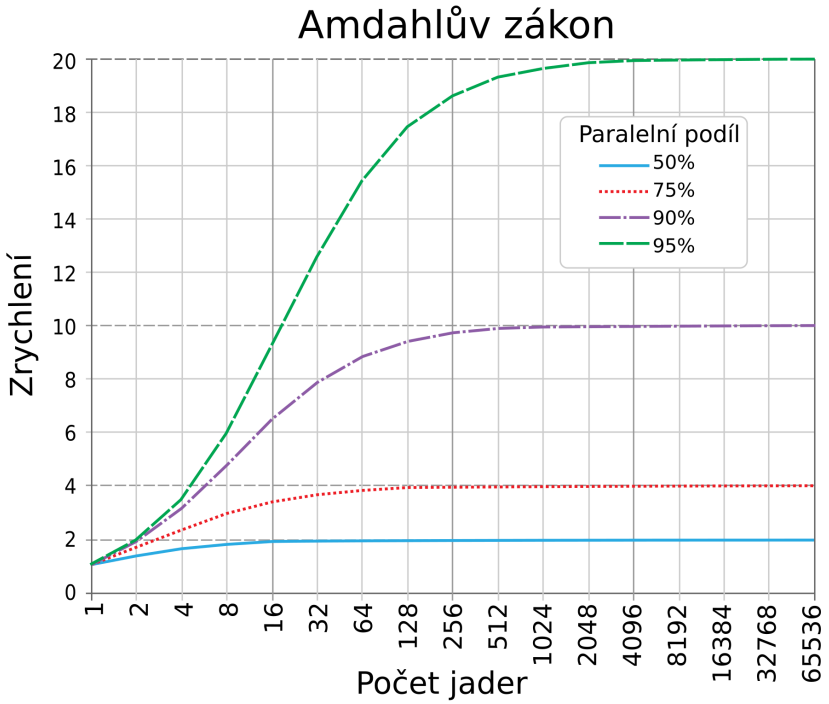
Z výkonnostní analýzy v podkapitole 4.3 vyplývá, že v aktuálním řešení při využití více jader nijak nedochází ke zkrácení celkové doby provádění funkce `watchAndProcessKpis`. Očekávaným výsledkem optimalizace popsané v předchozí podkapitole je tedy vytížení všech dostupných jader a zkrácení doby výpočtu. Míru jak moc lze kód zrychlit popisuje Amdahlův zákon. Tento zákon zavádí pojmy *zrychlení* a *efektivita zrychlení*, které jsou počítány následovně:

$$S = \frac{T_1}{T_n} \qquad E = \frac{T_n}{n},$$

kde je: T_1 sekvenční čas (doba běhu programu na jednom jádře), T_n paralelní čas (doba běhu programu na n jádrech), n počet jader, S zrychlení (na n jádrech), E efektivita zrychlení (pro n jader).

Podle tohoto zákona lze prováděný kód rozdělit na část paralelizovatelnou a sekvenční. Očekávané zrychlení je pak závislé na poměru času stráveného v těchto částech a počtu dostupných jader. Ve funkci `watchAndProcessKpis` je paralelizovatelnou částí cyklus volání

funkce `processKPI`. V tomto cyklu je tráveno nejvíce času a celkový podíl byl naměřen přibližně kolem 95 %. Z grafu Amdahlova zákona na obrázku 5.4 lze tedy vyčíst, že například při využití 4 jader je teoretické zrychlení oproti jednomu jádru téměř 4násobné. Reálné zrychlení však lze očekávat nižší, jelikož se do výsledku musí zahrnout i režie způsobená rozdělováním práce a střádáním výsledků. Tato režie bude úměrně růst s rostoucím počtem jader. Pro daný podíl paralelizovatelné části kódu pak exituje maximální možný počet využitelných jader, po jehož dosažení bude podíl režie tak vysoký, že už bude využívání dalších jader neefektivní.



Obrázek 5.4: Amdahlův zákon[1]

[1]převzato a přeloženo z https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg

Kapitola 6

Implementace

Kapitola 5 popisuje několik návrhů na úpravu stávající implementace zpracování KPI firmy Logimic. Tato kapitola se zabývá tím, jak byly tyto návrhy realizovány. Mezi navrhované úpravy patří úprava datového modelu, aby více odpovídal reálným datům. Tomuto návrhu se věnuje podkapitola 6.1. Dále je navrženo zavedení vícevláknového provádění, aby byly efektivně využívány dostupné výpočetní prostředky. Přístup k implementaci je popsán v podkapitole 6.2. Poslední navrženou úpravou je přepis zdrojového kódu, aby byla využívána knihovna pro ORM v souladu s principy popsanými v podkapitole 3.2. Postup při přepisu je popsán v podkapitole 6.3.

6.1 Datový model a migrace

Z kapitoly 4 je známé, že se v projektu pro komunikaci s databází používá knihovna TypeORM. Tato knihovna byla také popsána v podkapitole 3.2. V této podkapitole je rozebrána implementace úpravy datového modelu, jejíž princip již byl popsán v podkapitole 5.1. Úprava se skládá ze dvou částí: změny definice modelu ve zdrojovém kódu (popsané v sekci 6.1.1) a vytvoření migrace pro úpravu přímo relační databáze (popsané v sekci 6.1.2).

6.1.1 Změna definice ve zdrojovém kódu

Všechny entity jsou pro knihovnu TypeORM definovány v souboru `OrmEntities.ts`[1]. Úprava tohoto souboru podle návrhu se týká následujících entit:

- `OrmMnKpi` – odpovídá tabulce `mn_kpi`,
- `OrmMnKpiRule` – odpovídá tabulce `mn_kpi_rule`,
- `OrmRelKpiValueRule` – odpovídá tabulce `rel_kpi_value_rule`,
- `OrmMnParameterValue` – odpovídá tabulce `mn_parameter_value`,
- `OrmRelKpiRule` – odpovídá tabulce `rel_kpi_rule` (Tato entita nebyla upravena ale zcela odstraněna).

Nyní následuje detailní popis provedených úprav. Popis je doplněn o výpisy kódu, které přímo znázorňují provedené změny. V těchto výpisech se mohou nacházet dvě části, které

[1] src/db/entity/OrmEntities.ts

jsou pojmenovány následovně: *původní*, která obsahuje originální kód a *nové*, kde se nachází buď nově přidaný kód nebo úprava, kterou byla původní část nahrazena.

OrmMnKpi

U entity byla nejprve změněna kardinalita vztahu s entitou `OrmMnKpiRule` na `N:1`. Změna byla provedena úpravou datového typu u parametru `mnKpiRules` z `OrmMnKpiRule[]` na `OrmMnKpiRule`. Změnu kardinality je pak potřeba provést i v entitě `OrmMnKpiRule`. Zároveň byl přidán nový parametr `mnKpiRuleId`, který nově ponese hodnotu ID příslušného záznamu vyhodnocovacího pravidla z tabulky `mn_kpi_rule`. Parametr tak usnadní případné manuální sestavování SQL dotazů (za předpokladu, že to bude v souladu s principy popsanými v podkapitole 3.2). Výsledkem je úprava znázorněná ve výpisu 6.1.

```
1  @Entity("mn_kpi")
2  export class OrmMnKpi extends eif.MnKpi
3  {
4     //puvodni
5     @ManyToMany(type => OrmMnKpiRule)
6     @JoinTable({ name: "rel_kpi_rule" })
7     mnKpiRules: OrmMnKpiRule[];
8
9     //nove
10    @Column({ name: "mn_kpi_rule_id", type: "bigint", nullable: false })
11    mnKpiRuleId: number;
12    @ManyToOne(() => OrmMnKpiRule, (rule) => rule.mnKpis, { nullable:
          false, onDelete: 'NO ACTION' })
13    mnKpiRule: OrmMnKpiRule
14 }
```

Výpis 6.1: Úprava kardinality mezi entitami `OrmMnKpi` a `OrmMnKpiRule`

Dále byl v entitě přejmenován parametr `relKpiValueRule` na `monitoredParameterValues-Details`, aby jeho název přesněji popisoval jeho význam. Datový typ tohoto parametru byl upraven tak, aby odpovídal přejmenování entity `OrmRelKpiValueRule` na `OrmMnMonitored-ValueDetail` (bude popsáno dále). Poslední úpravou této entity je přidání nového parametru `monitoredParameterValues`, který odkazuje rovnou na entity `OrmMnParameterValue`. Tento redundantní odkaz je použit i v oficiální dokumentaci [38] demonstrující správný postup implementace vztahu `M:N`. Umožňuje tak přistupovat přímo na monitorované parametry bez načítání detailů. Výsledkem je úprava znázorněná ve výpisu 6.2

```
@Entity("mn_kpi")
export class OrmMnKpi extends eif.MnKpi
{
   // puvodni
   @OneToMany(() => OrmRelKpiValueRule, rel => rel.mnKpi)
   relKpiValueRule: OrmRelKpiValueRule[]

   //nove
   @ManyToMany(() => OrmMnParameterValue, (paramValue) => paramValue.
       monitoringKpis)
   @JoinTable({ name: "mn_monitored_value_detail" })
   monitoredParameterValues: OrmMnParameterValue[];

   @OneToMany(() => OrmMnMonitoredValueDetail, (monitoredValueDetail) =>
       monitoredValueDetail.monitoringKpi)
   monitoredParameterValuesDetails: OrmMnMonitoredValueDetail[];
}
```

Výpis 6.2: Úprava vztahu entit `OrmMnKpi` a `OrmMnParameterValue`

OrmMnKpiRule

V entitě bylo nutné zohlednit změnu kardinality vztahu s entitou `OrmMnKpi` – aby definice modelu byla v souladu s principy z podkapitoly 3.2. To bylo provedeno přidáním nového parametru `mnKpis`, který odkazuje na všechny instance `OrmMnKpi`. Přes tento parametr tedy lze přistoupit na entity `OrmMnKpi`, které dané pravidlo využívají k vyhodnocení svých monitorovaných hodnot. Způsob přidání ukazuje výpis 6.3.

```
@Entity("mn_kpi_rule")
export class OrmMnKpiRule extends eif.MnKpiRule
{
   //nove
   @OneToMany(() => OrmMnKpi, (kpi) => kpi.mnKpiRule)
   mnKpis: OrmMnKpi[];
}
```

Výpis 6.3: Úprava entity `OrmMnKpiRule`

OrmRelKpiValueRule

Tato entita byla přejmenována na `OrmMnMonitoredValueDetail`, aby stále odpovídala zavedené pojmenovávací konvenci a zároveň přesněji vyjadřovala svůj význam – detail relace mezi KPI a monitorovanou hodnotou parametru. Z této entity byl podle návrhu odstraněn parametr `mnKpiRule`, který zajišťoval relaci s entitou `OrmMnKpiRule`. Taktéž byly přejmenovány parametry `mnKpi` a `mnKpiId` na lépe vysvětlující `monitoringKpi` a `monitoringKpiId`. Na stejném principu pak byly přejmenovány parametry `mnParameterValue` a `mnParameterValueId` na `monitoredParameterValue` a `monitoredParameterValueId`. Pro zachování přehlednosti obsahuje výpis 6.4 jen výsledný stav kódu.

```
@Entity("mn_monitored_value_detail")
export class OrmMnMonitoredValueDetail
{
   //nove
   @Column({ name: "mn_kpi_id", type: "bigint", nullable: false })
   monitoringKpiId: number;
   @ManyToOne(() => OrmMnKpi, (kpi) => kpi.
       monitoredParameterValuesDetails, { nullable: false, })
   @JoinColumn({ name: "mn_kpi_id" })
   monitoringKpi: OrmMnKpi

   @Column({ name: "mn_parameter_value_id", type: "bigint", nullable:
       false })
   monitoredParameterValueId: number;
   @ManyToOne(() => OrmMnParameterValue, (paramValue) => paramValue.
       monitoredParameterValuesDetails, { nullable: false})
   @JoinColumn({ name: "mn_parameter_value_id" })
   monitoredParameterValue: OrmMnParameterValue
}
```

Výpis 6.4: Úprava entity `OrmRelKpiValueRule`

OrmMnParameterValue

Do této entity byly pouze přidány parametry k vyjádření relace s entitou `OrmMnKpi`, aby definice odpovídala principům z 3.2. Předchozí definice této entity propojení s `OrmMnKpi` nijak nezohledňovala. Při definici relace je nutné opět brát v úvahu, že relace je určena entitou `OrmMnMonitoredValueDetail`. Stejně jako při úpravě `OrmMnKpi` je i zde využit redundantní odkaz, kterým je možné přistupovat přímo na entity `OrmMnKpi`. Úprava je znázorněna ve výpisu 6.5.

```
@Entity("mn_monitored_value_detail")
export class OrmMnMonitoredValueDetail
{
   //nove
   @ManyToMany(() => OrmMnKpi, (paramValue) => paramValue.
       monitoredParameterValues)
   @JoinTable({ name: "mn_monitored_value_detail" })
   monitoringKpis: OrmMnKpi[];

   @OneToMany(() => OrmMnMonitoredValueDetail, (monitoredValueDetail) =>
       monitoredValueDetail.monitoredParameterValue)
   monitoredParameterValuesDetails: OrmMnMonitoredValueDetail[];
}
```

Výpis 6.5: Úprava entity `OrmMnParameterValue`

Upravením entit ve zdrojovém kódu (v souboru `OrmEntities.ts`) nedojde ke změně datového modelu v samotné PostgreSQL databázi, jelikož se nachází v produkčním prostředí. Pro změnu datového modelu v databázi je potřeba vytvořit migraci.

6.1.2 Migrace

Koncept migrace již byl popsán v podkapitole 5.1 – spočívá v definování několika SQL dotazů, které převedou datový model databáze na novou verzi. Tyto SQL dotazy lze vygenerovat automaticky na základě úpravy modelu ve zdrojovém kódu nebo je lze vytvořit manuálně. Při automatickém vygenerování však vždy došlo ke tvorbě nepřehledného kódu (pravděpodobně i díky dvojici modelů) a z toho důvodu bylo přistoupeno jen k manuálnímu vytvoření migrace pomocí následujících 7 dotazů:

1. vytvoření atributu pro FK z `mn_kpi_rule`,

   ```
   ALTER TABLE mn_kpi
   ADD COLUMN mn_kpi_rule_id int;
   ```

2. přidání integritního omezení FK na právě přidaný atribut,

   ```
   ALTER TABLE mn_kpi
   ADD CONSTRAINT FK_kpi_kpirule
   FOREIGN KEY (mn_kpi_rule_id) REFERENCES mn_kpi_rule(id);
   ```

3. doplnění hodnot do vytvořeného atributu z relační tabulky `rel_kpi_rule`,

   ```
   UPDATE mn_kpi kpi
   SET mn_kpi_rule_id = rel.mn_kpi_rule_id
   FROM rel_kpi_rule rel
   WHERE kpi.id = rel.mn_kpi_id;
   ```

4. smazaní relační tabulky mezi `mn_kpi` a `mn_kpi_rule`,

   ```
   DROP TABLE rel_kpi_rule
   ```

5. přejmenování tabulky `rel_kpi_value_rule` na `mn_monitored_value_detail`,

   ```
   ALTER TABLE rel_kpi_value_rule
   RENAME TO mn_monitored_value_detail
   ```

6. odstranění atributu relace s tabulkou `mn_kpi_rule`,

   ```
   ALTER TABLE mn_monitored_value_detail
   DROP COLUMN mn_kpi_rule_id
   ```

7. přidání integritního omezení, aby záznamy v `mn_monitored_value_detail` pro danou kombinaci `mn_kpi` a `mn_parameter_value` byly unikátní.

   ```
   ALTER TABLE mn_monitored_value_detail
   ADD CONSTRAINT UQ_kpi_monitored_value
   UNIQUE (mn_kpi_id, mn_parameter_value_id)
   ```

Postup implementace migrace s využitím knihovny TypeORM je popsán v oficiální dokumentaci [39]. Podle této dokumentace je možné využít program `cli.js`, který se nachází ve složce `node_modules/typeorm` po importování TypeORM přes správce balíčku *npm*[2]. Nejprve byl tedy vytvořen soubor s migrací, která byla pojmenována jako `kpi_rule_to_1-n`.

[2] https://www.npmjs.com/

Migrace byla vytvořena pomocí příkazu:

```
$ cli.js migration:create kpi_rule_to_1-N
```

Tímto byl vytvořen nový soubor `1679140608315-kpi_rule_to_1-N.js`, jehož název automaticky začíná časovou značkou vytvoření souboru. V tomto souboru bylo implementováno zasílání 7 popsaných SQL dotazů pomocí instance třídy `QuerryRunner`. Struktura souboru je znázorněna ve výpisu 6.6.

```
1  import { MigrationInterface, QueryRunner } from "typeorm"
2
3  export class kpiRuleTo1N1679140608315 implements MigrationInterface {
4
5     public async up(queryRunner: QueryRunner): Promise<void> {
6
7        // prvni SQL dotaz
8        await queryRunner.query(
9           "ALTER TABLE mn_kpi
10          ADD COLUMN mn_kpi_rule_id int",
11       );
12
13       // dalsi dotazy
14    }
15 }
```

Výpis 6.6: Soubor s migrací knihovny TypeORM

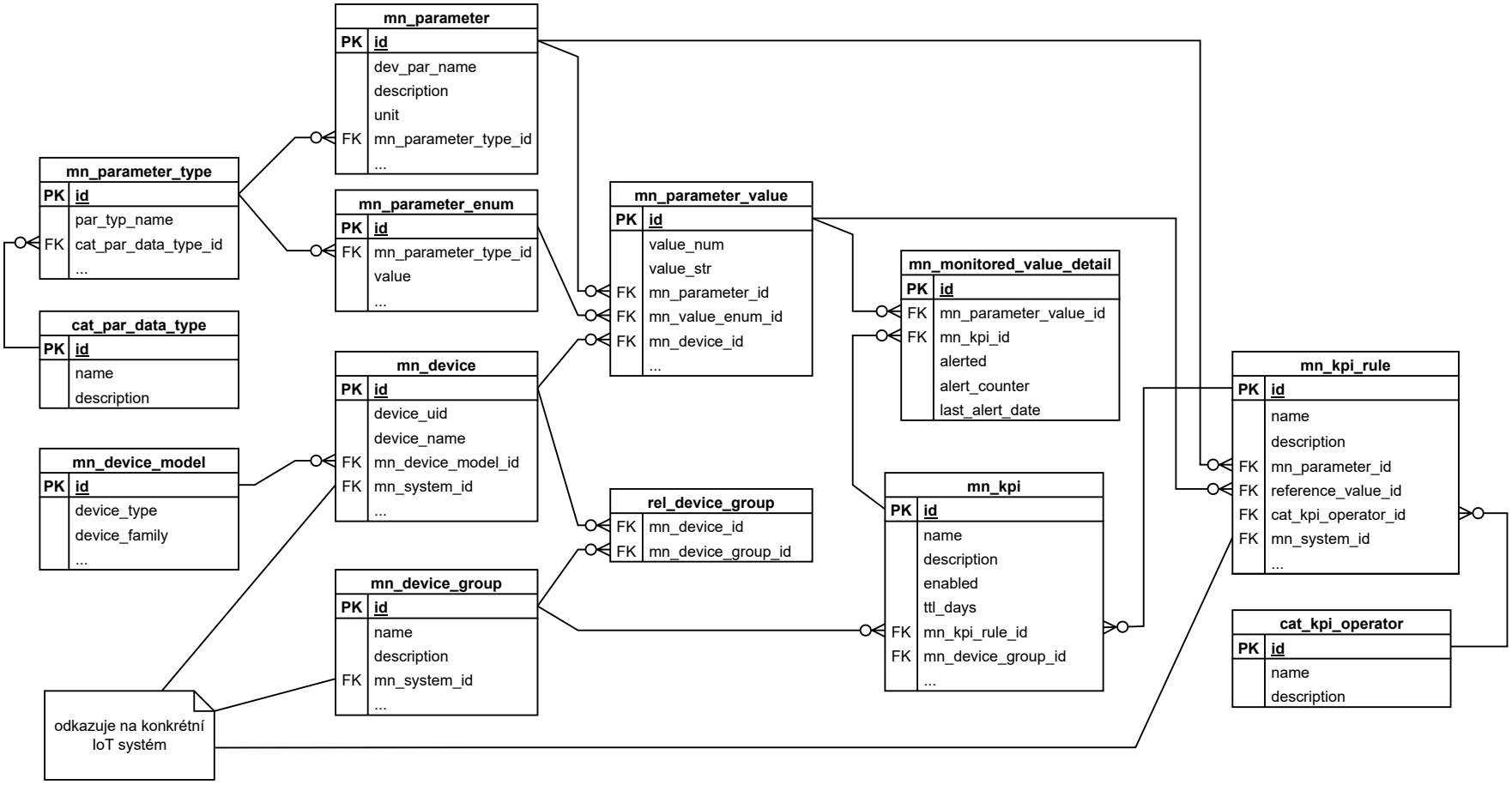
V projektu se navíc nachází konfigurační soubor pro knihovnu TypeORM nazvaný `ormconfig.js`. V tomto souboru se především nachází informace, jako je:

- specifikace databáze a přihlašovacích údajů,
- specifikace souboru s definicí entit,
- specifikace složky se soubory s migracemi.

Do specifikované složky pro migrace byl přidán vytvořený soubor `1679140608315-kpi_rule_to_1-N.js`. Následně bylo možné spustit provedení migrace databáze pomocí následujícího příkazu:

```
$ cli.js migration:run -d ormconfig.js
```

Tímto byla dokončena úprava datového modelu. Celé schéma výsledného modelu je na obrázku 6.1. Následující podkapitola se věnuje další části návrhu, kterou je zavedení vícevláknového provádění.



Obrázek 6.1: Upravený ER diagram části databáze

6.2 Zavedení vícevláknového provádění

Tato podkapitola se zabývá zavedením vícevláknové ho provádění, správě pracovních vláken a jejich využití při zpracování KPI. V podkapitole 5.3 byly popsány dva možné způsoby jakými lze pracovní vlákna spravovat:

- statické – kde hlavní vlákno přímo pracuje s instancemi jednotlivých vláken,
- dynamické – kde jsou vlákna umístěna do zásobníku a mají sdílenou frontu úloh.

Pro realizaci bylo vybráno dynamické spravování, neboť umožňuje snadno měnit počet pracovních vláken podle potřeby a to navíc bez vážných zásahů do zdrojového kódu. Celý postup zavedení vícevláknového provádění je popsán v sekci 6.2.1. Dále se tato podkapitola v sekci 6.2.2 zabývá chováním knihovny TypeORM ve vícevláknovém prostředí a popisuje způsob jejího využití.

6.2.1 Knihovna vícevláknového provádění

Jak je popsáno v podkapitole 5.3, pro implementaci dynamické správy vláken je vhodné využít některou z volně dostupných knihoven, jelikož implementace vlastní synchronizační logiky je poměrně náročná. Na základě popularity a četnosti využití ve správci balíčků *npm*[3] byly uvažovány knihovny *threads*[4] a *piscina*[5]. Nakonec byla vybrána knihovna `threads`, která podle dokumentace [35] nabízí přirozenější podporu jazyka TypeScript.

Stejně jako samotný modul `worker threads` (popsaný v 5.3), i tato knihovna vyžaduje, aby implementace pracovního vlákna byla v samostatném souboru. Proto byl vytvořen soubor `worker.ts`, do kterého byla přesunuta implementace funkce `processKPI`. Zároveň je podle knihovny nutné použít speciální funkci `expose`, která umožní funkci `processKPI` spouštět paralelně. Tato úprava je znázorněna ve výpisu 6.7.

```
1 //import knihovny
2 import { expose } from "threads/worker"
3
4 // specialni funkce expose z~knihovny threads
5 expose(processKpi);
6
7 async function processKpi(kpiId: number) : OrmMnMonitoredValueDetail[]
8 { ... }
```

Výpis 6.7: Ukázka implementace v souboru `worker.ts`

Na straně hlavního vlákna (funkce `watchAndProcessKpis`) je pak nejprve potřeba vytvořit instanci zásobníku s vlákny. Při vytváření je možné nastavit zásobníku počet vláken. Pokud počet není nastaven, je vytvořeno tolik vláken, kolik je jader procesoru. Instanci zásobníku je pak možné zadávat práci voláním funkcí, na které byla použita funkce `expose`. Instance pak bude zadanou práci delegovat na pracovní vlákna. Podle předchozího výpisu lze takto nyní použít funkci `processKpi`. Funkce pro zadání práce vrací objekt typu `Promise`

[3] https://www.npmjs.com/search?q=worker%20threads&ranking=popularity
[4] https://www.npmjs.com/package/threads
[5] https://www.npmjs.com/package/piscina

(který byl popsán v podkapitole 5.3). To znamená, že hlavní vlákno může pomocí slova `await` vyčkat na dokončení zadané úlohy. Princip využití demonstruje následující výpis 6.8.

```
1  //import knihovny
2  import { spawn, Pool, Worker } from "threads";
3
4  //vytvoreni zasobniku ze souboru ./worker.js
5  pool = Pool(() => spawn(new Worker("./worker")));
6
7  var kpis : OrmMnKpi = /* ziskani KPI z~databaze */;
8  for (let kpi of kpis)
9  {
10    //zadan ukol zpracovat KPI s~danym id + vyckani na vysledek
11    var result = await pool.queue(w => w.processKpi(kpi.id))
12 }
```

Výpis 6.8: Zpracování KPI na pomocném vlákně

Řádek č. 11 z tohoto výpisu znamená, že hlavní vlákno zadá úlohu na zpracování KPI s daným ID. Po zadání práce bude hlavní vlákno na základě slova `await` v dané funkci pozastaveno (nejedná se však o aktivní čekání – viz. 5.3). Z toho vyplývá, že takto napsaný paralelní kód má stále vlastnosti sekvenčního kódu, protože se vždy čeká na zpracování jednoho KPI před zahájením zpracování druhého. Proto je potřeba tento kód ještě upravit způsobem znázorněným ve výpisu 6.9.

```
1  var processKpiPromises = [] //fronta (FIFO)
2
3  for (let kpi of kpis)
4  {
5    //zadan ukol zpracovat KPI s~danym id
6    let promise = pool.queue(w => w.processKpi(kpi.id))
7    processKpiPromises.push(promise);
8  }
9
10 for(let kpi of kpis)
11 {
12   //vyckani na vysledek
13   var result = await processKpiPromises.pop();
14 }
```

Výpis 6.9: Správné využití paralelizace v hlavním vlákně

V takto upraveném kódu jsou nejprve zadána všechna KPI ke zpracování a až potom dochází k neblokujícímu čekání. Toto je základní koncept používání více vláken v jazyce Typescript. V následující sekci je probráno jeho použití v kombinaci s knihovnou TypeORM.

6.2.2 Knihovna TypeORM ve vícevláknovém prostředí

V knihovně TypeOrm slouží pro správu spojení s databázemi třída `ConnectionManager`. Jedna instance této třídy obsluhuje vždy jedno připojení k databázi. Konfigurační údaje daného připojení jsou buď načteny ze souboru `ormconfig.js` nebo je lze dané instanci předat přímo. Po úspěšném vytvoření spojení je možné prostřednictvím této instance získat přístup k instanci třídy `EntityManager`. Instance `EntityManager` už zprostředkovává přístup jednotlivým objektům ORM, které reprezentují záznamy uložené v databázi. Při načtení objektů ORM z databáze dochází k lokálním uchovávání jejich stavu v databázi, což v některých případech umožňuje filtrovat zbytečnou komunikaci (např. uložení entity, která však nebyla nijak změněna) – viz. cache v podkapitole 3.2.3.

Podle [25] mohou vlákna mezi sebou komunikovat pouze předáváním zpráv, jejichž obsah je předán kopírováním (s výjimkou objektů typu *transferable*). Instance třídy `ConnectionManager` a `EntityManager` ale existují pouze v kontextu daného vlákna (hlavního nebo pracovního) a nelze je mezi vlákny předávat. Pro každé vlákno tak musí být vytvořeno samostatné spojení s databází. Stejně tak mezi vlákny nelze předávat ani samotné objekty ORM, neboť instance `EntityManager` hlídá jejich synchronizaci s databází.

Aby pracovní vlákno mohlo vytvořit vlastní spojení s databází, musí k tomu dostat úkol. To bylo implementováno tak, že v souboru `worker.ts` byla vytvořena nová funkce `connectDb`, která vytvoří nové spojení a instanci `EntityManager` uloží do globální proměnné modulu. Opět bylo důležité zavolat funkci `expose`, aby bylo možné funkci zavolat z hlavního vlákna. Implementace je znázorněna ve výpisu 6.10.

```
1  var global_entity_manager : EntityManager;
2  var conn_set = false
3  expose({ connectDb, processKpi });
4
5  async function connectDb(credentials): Promise<boolean>
6  {
7    if(!conn_set)
8    {
9        //vytvoreni noveho spojeni a jeho ulozeni do globalniho kontextu
10       let connection = await createConnection(credentials);
11       global_entity_manager = connection.getEntityManager();
12       conn_set = true;
13       return true;
14   }
15   else
16   {
17       // pokud je dany worker jiz pripojeny, pak po urcite dobe
18       // (napr. 2s) vrati false
19       wait(2000)
20       return false;
21   }
22 };
```

Výpis 6.10: Funkce pro připojení workera k databázi

Knihovna `threads` v případě využití dynamického spravování neposkytuje přístup k jednotlivým pracovním vláknům, ale pouze ke sdílené frontě. Aby si však všechna vlákna úspěšně vytvořila spojení s databází, musí každé z nich obdržet úkol k provedení funkce `connectDb`. To lze řešit přidáním tolika úkolů do fronty, kolik je pracovních vláken. V extrémních případech však může nastat situace, kdy jedno vlákno stihne obsloužit více takových úkolů. To by znamenalo, že na některé z vláken úkol k připojení nezbude a nebude připojeno. Toto lze ošetřit hlídáním návratových hodnot z jednotlivých úkolů a počítáním případů, kdy se vlákno připojilo (úkol vrátil `true` – viz. výpis 6.10 ř. 13). Pokud by se stalo, že po dokončení všech úkolů nebude počet navrácených hodnot `true` roven počtu pracovních vláken, spustí se zadávání úkolů k připojení znovu. Všechna již připojená vlákna budou muset počkat po danou dobu (výpis 6.10 ř. 19) a nebudou moci přijímat další úkoly k připojení. Tímto bude zajištěno, že se úkol k připojení postupně dostane ke všem dosud nepřipojeným vláknům. Po provedení této akce jsou všechna spojení s databází udržována po celou dobu běhu pracovnách vláken. Vlákna jsou přitom mezi iteracemi zpracování udržována a znovu používána, takže se tato operace provede pouze jednou. Dodatečnou kontrolu spojení už si pak provádí vlákna sama. Výpis 6.11 obsahuje ukázku kódu pro rozdělení úkolů k připojení. V tomto kódu je místo čekání na výsledek pomocí `await` použita funkce `then` (viz. 5.3).

```
1  pool = Pool(() => spawn(new Worker("./worker")));
2  var succ_conns_count: number = 0; // pocet pripojeni
3  var expected_conns_count = cpu.count(); //pocet workeru = pocet jader
4  var credentials = /* prihlasovaci udaje k~databazi */
5
6  //kontrola poctu pripojenych workeru
7  while (succ_conns_count < expected_conns_count)
8  {
9     for (let i = 0; i < expected_conns_count; i++){
10       pool.queue(w => w.connectDb(connectionCredentials))
11           .then(result =>
12           {
13             if (result) {
14                succ_conns_count += 1;
15             }
16           });
17    }
18    await pool.completed(); //cekani na dokonceni vsech ukolu
19 }
20 //v teto casti kodu jsou vsechna vlakna uspesne pripojena
```

Výpis 6.11: Funkce pro zadání úkolů k připojení vláken k databázi

Vzhledem k právě popsaným omezením knihovny pro ORM v paralelním prostředí je potřeba provést v aktuálním kódu pro zpracování KPI určité změny, aby bylo vícevláknové zpracování efektivní. Tyto změny se týkají především oblasti rozdělování práce mezi vlákna a způsob práce s ORM knihovnou. Úpravu zbývajícího zdrojového kódu popisuje následující podkapitola.

6.3 Úprava hlavního kódu

Nyní, když je upraven datový model a je znám způsob implementace vícevláknového provádění, je možné přistoupit k úpravě zdrojových kódů funkcí `watchAndProcessKpis` a `processKPI`. Jednou z hlavních navrhovaných změn je odstranění manuálního sestavování SQL a odstranění dotazů, které nesouvisí s významem těchto funkcí (viz. 5.2). Úprava práce s knihovnou ORM je popsána v sekcích 6.3.1 a 6.3.2. Sekce 6.3.3 pak popisuje vyřešení problému s nadbytečnými dotazy.

6.3.1 Využití ORM v hlavním vlákně

Z předchozí sekce 6.2.2 už je jasné, že pro práci s entitami slouží instance třídy `EntityManager`. Funkce `watchAndProcessKpi` na začátku musí z databáze získat informace o KPI, které mají být vyhodnoceny (viz. diagram 4.6). To doposud prováděla pomocí manuálně sestaveného dotazu `qr`. Nově má být zpracování KPI delegováno na pracovní vlákna. Zároveň je zbytečné načítat celé objekty ORM, protože si je vlákna musí z databáze získat samy. Z toho vyplývá, že hlavní vlákno bude načítat pouze ID příslušných KPI, která následně přerozdělí ke zpracování.

Pokud by však hlavní vlákno zadávalo práci jako „zpracování samostatného KPI“ (viz. výpis 6.9 z předchozí podkapitoly), pak by na databázi bylo při každém volání funkce `watchAndProcessKpi` zasláno přibližně dvojnásobné množství dotazů než kolik je KPI v databázi (vždy jeden pro načtení KPI a jeden pro jeho uložení). Z toho důvodu je nutné zadávat práci ve větších počtech KPI, aby pak bylo možné v jeden okamžik načítat z databáze více informací a tím klesl celkový počet dotazů.

Základním způsobem je sloučení KPI do tolika skupin, kolik je pracovních vláken. U tohoto rozdělení je však důležité předejít problému nevyvážené zátěže statického dělení práce, který byl popsán v 5.3.3. Z postupu zpracování KPI vyplývá, že náročnost jednoho zpracování je především určena počtem parametrů, které daný KPI monitoruje. Řešením je tedy získat všechny KPI seřazené podle jejich náročnosti. To je provedeno následujícím manuálně sestaveným SQL dotazem, neboť TypeORM nepodporuje načítání samostatných parametrů z relací:

```
SELECT mn_kpi_id
FROM mn_monitored_value_detail
WHERE mn_kpi_id IN (SELECT id
                    FROM mn_kpi
                    WHERE enabled=true)
GROUP BY mn_kpi_id
ORDER BY count(id) DESC;
```

Následně je možné takto seřazené KPI postupně rozdělit mezi vlákna. I přes toto pečlivé rozdělení se však může stát, že některým bude přiděleno více práce než ostatním (např. pokud jenom 2 KPI budou extrémně náročné). Z toho důvodu je nutné vytvořit o něco více skupin KPI. Více skupin umožní pracovnímu vláknu s náročným KPI izolovaně pracovat, zatímco ostatní si budou moci brát další práci. Tímto se případný nepoměr v rozdělení práce stane zanedbatelným.

Zvolit výsledný počet skupin je poněkud záludný úkol. Je potřeba najít rovnováhu, protože s rostoucím počtem skupin poroste počet dotazů na databázi. S menším počtem skupin se zase může více projevovat nepoměr v rozdělení práce. Spolehlivým způsobem, jak se k této rovnováze dopracovat je vyzkoušet různé počty skupin a vybrat nejrychlejší

variantu. Jedná se však o poměrně pracný postup a navíc se změnou počtu monitorovaných parametrů tato rovnováha může posunout. Z toho důvodu bylo přistoupeno k řešení podle implementace zásobníku vláken v *cpython* [31]. Výpis 6.12 obsahuje část ze zdrojového kódu cpython, ve kterém dochází k určení **velikosti jedné skupiny** práce.

```
if chunksize is None:
   chunksize, extra = divmod(len(iterable), len(self._pool) * 4)
   if extra:
      chunksize += 1
if len(iterable) == 0:
   chunksize = 0
```

Výpis 6.12: Výpočet velikosti jedné skupiny práce v cpython [31]

Z tohoto kódu tedy vyplývá, že výsledný **počet skupin** práce je počet jader vynásobený číslem 4. Stejný způsob byl zvolen při implementaci nové verze `watchAndProcessKpis`.

6.3.2 Úprava pracovního vlákna a využití ORM

U pracovních vláken je potřeba upravit funkci `processKpi`, aby místo jednoho ID jich přijímala více, protože tak je bude posílat hlavní vlákno. Zároveň je potřeba zajistit, aby si pracovní vlákno z databáze získalo všechna potřebná data pro vyhodnocení přiřazených KPI. Aby však byl co nejvíce zachován účel funkce `processKpi`, je lepší pro zadávání práce vytvořit funkci zcela novou. Nechť se tato nová funkce nazývá `processKpiGroup`. Cílem nové funkce bude přijmout seznam ID od hlavního vlákna a z databáze získat potřebná data. Každý KPI pak bude zpracován pomocí funkce `processKpi` a výsledné hodnoty uloženy do databáze. Účelem funkce `processKpi` tak zůstane pouze ověření, zda je monitorovaná hodnota ve správném vztahu s referenční hodnotou.

Výpis 6.13 obsahuje implementaci této funkce. Funkce přijme seznam ID jako parametr volání. Následně pomocí globální instance `EntityManager` získá KPI jejichž ID je v dodaném seznamu. To je zajištěno speciální funkcí `In` z knihovny TypeORM. Načtení dalších relací daného KPI je pak specifikováno pomocí parametru `relations`, k němuž je přidělen objekt `KpiSearchRelationsStructure` typu JSON. Popis tohoto objektu je pak uveden v následujícím samostatném výpisu 6.14.

```
async function processKpiGroup(kpiIdsArray: number[]) : Promise<any>
{
   var kpiArr: OrmMnKpi[];
   //ziskani vsech kpi a potrebnych dat
   kpiArr = await global_entity_manager.find(OrmMnKpi, {
      where: {
         id: In(kpiIdsArray)
      },
      relations: KpiSearchRelationsStructure
   });
   //zbytek funkce
}
```

Výpis 6.13: Získání KPI ve funkci `processKpiGroup`

```
const KpiSearchRelationsStructure = {
   mnKpiRule: {
      referenceValue: {
         mnParameter: { /* dalsi relace */ },
         mnValueEnum: true
      },
      catKpiOperator: true
   },
   monitoredParameterValuesDetails: {
      monitoredParameterValue: { /* dalsi relace */ }
   }};
```

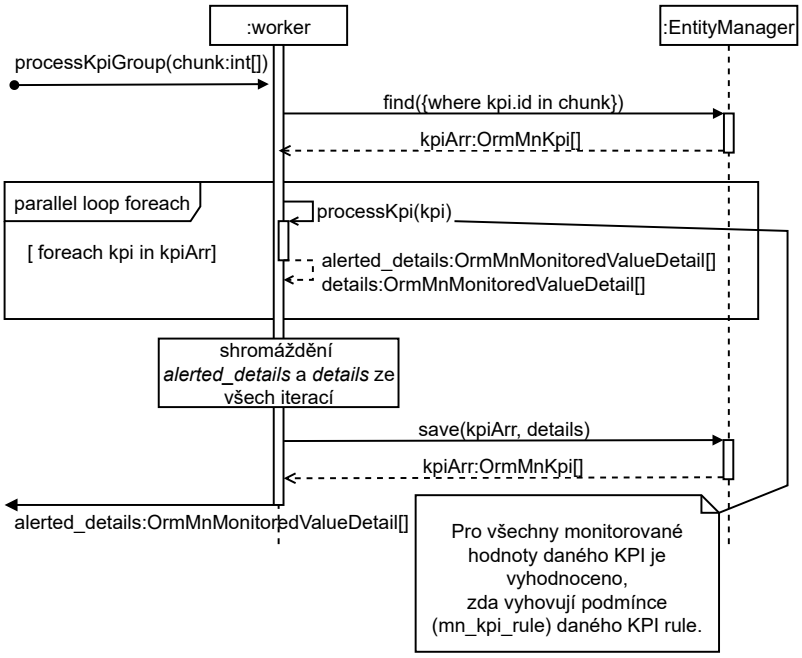
Výpis 6.14: Objekt typu JSON pro definici lazy loading relací

Ve zbytku funkce `processKpiGroup` je použit stejný princip paralelního zpracování, který byl využit ve výpisu 6.9. V této funkci už se však nejedná o vícevláknové zpracování, protože neprobíhá za účasti dalších vláken, ale vše se děje jen na daném pracovním vlákně. Zbytek funkce je znázorněn ve výpisu 6.15. Celá funkce `processKpiGroup` je pak ještě znázorněna sekvenčním diagramem na obrázku 6.2.

```
async function processKpiGroup(kpiIdsArray: number[]) : Promise<any>
{
   // nacteni kpi
   // zavolani processKpi na vsechna KPI
   var processKpiTasks = new Array();
   kpiArr.forEach( async (kpi) => {
      let task = processKpi(kpi); // volani processKpi na jednotliva KPI
      processKpiTasks.push(task);
   });

   var influxData = new Array();
   var changedKpis: OrmEntitites.OrmMnKpi[];
   var monitoredValues: OrmEntitites.OrmMnMonitoredValueDetail[];
   //zpracovani navracenych hodnot
   for (let processKpiTask of processKpiTasks)
   {
      let [alerted_details, all_details] = await processKpiTask;
      changedKpis.push(kpi);
      monitoredValues.push(all_details);
      influxData.push(alerted_details);
   }
   // ulozeni upravenych dat do databaze
   await global_entity_manager.save(monitoredValues);
   await global_entity_manager.save(changedKpis);
   return influxData;
}
```

Výpis 6.15: Paralelní zpracování KPI a ukládání změn do databáze v `processKpiGroup`



Obrázek 6.2: Sekvenční diagram funkce `processKpiGroup`

6.3.3 Přesun nadbytečných úkonů

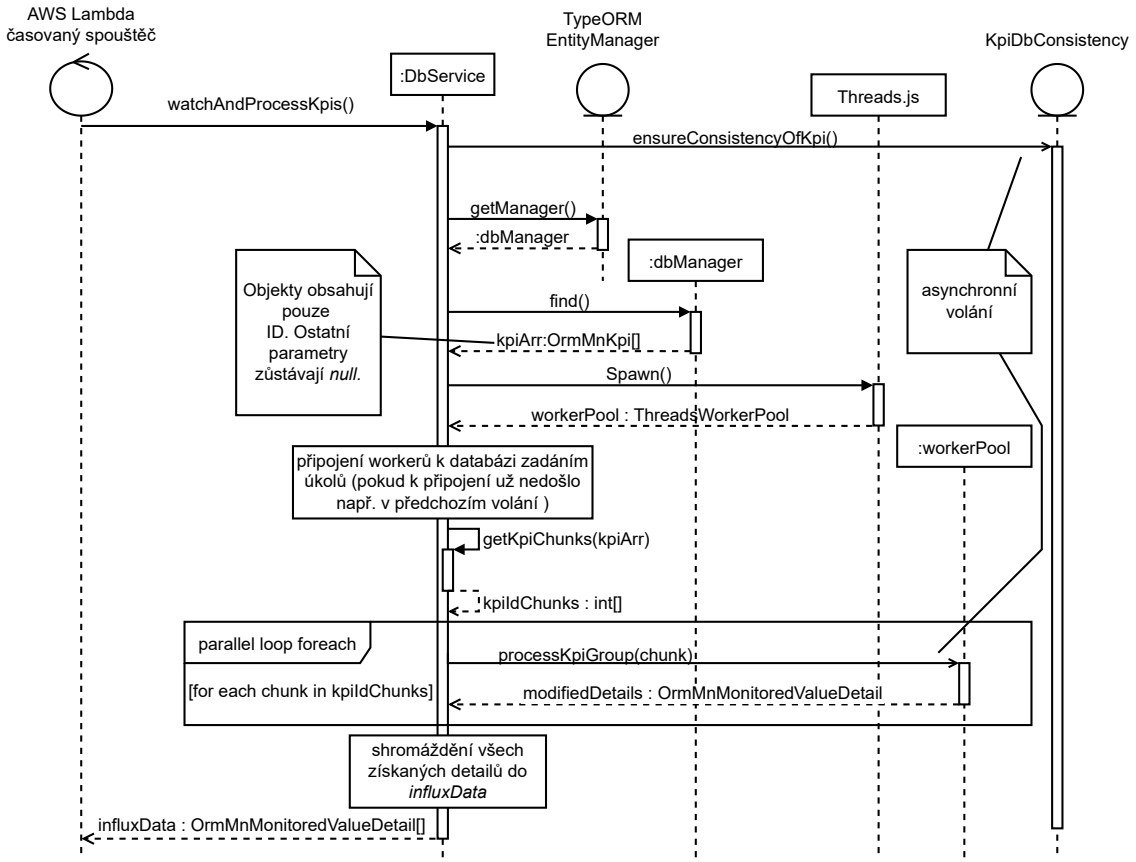
Poslední úpravou je přesun provedení dotazů `qD`, `qD1`, `q2` a `qv`. Jak již bylo popsáno v podkapitole 5.2, provádění těchto dotazů spíše souvisí se zachováním konzistentních dat v databázi než zpracováním KPI. Ideálním řešením by bylo, aby se databáze nikdy nedostávala do nekonzistentního stavu (např. vykonání dotazu `qD` ihned při odstranění vyhodnocovacího pravidla – viz. 4.5.1). Takové řešení však vyžaduje hlubokou znalost celého rozsáhlého systému, což je vysoce nad rámec zadání této práce. Z tohoto důvodu bylo implementováno jednodušší řešení.

Provádění těchto dotazů bylo přesunuto do nového modulu `KpiDbConsistency.ts`, ve kterém byla implementována funkce `ensureConsistencyOfKpi`. Tato funkce pak může být volána stejně jako `watchAndProcessKpi` (např. časovačem na platformě AWS Lambda). Aby však výsledek vykonání funkce `watchAndProcessKpi` zůstal oproti její původní verzi stejný, bylo volání `ensureConsistencyOfKpi` přidáno na její začátek. Díky této změně struktury kódu bude možné tento modul snadno odstranit, až bude vyvinuto vhodnější řešení.

6.4 Shrnutí provedených úprav

V této kapitole byla implementována optimalizace stávající funkce `watchAndProcessKpi` a pomocné funkce `processKpi`. Optimalizace se skládá z:

- úpravy datového modelu,
- zavedení zpracování KPI na pracovních vláknech,
- úpravy kódu, aby byla správně využívána knihovna TypeORM,
- izolace nadbytečných dotazů, aby je bylo možné později snadno přesunout nebo odstranit.



Obrázek 6.3: Sekvenční diagram upravené funkce `watchAndProcessKpis`

Na obrázku 6.3 je sekvenční diagram upravené funkce `watchAndProcessKpi`. Tato funkce nejprve paralelně spustí funkci pro zajištění konzistence databáze (viz. 6.3.3). Následně provede získání ID všech KPI, které mají být zpracovány. V dalším kroku jsou vytvořena pracovní vlákna a je ověřeno (případně navázáno) jejich připojení k databázi. Následně jsou ID rozdělena do několika skupin (`kpiIdChunks`) s přibližně stejnou náročností zpracování. Vlákna dané skupiny KPI zpracovávají paralelně pomocí funkce `processKpiGroup`. Po skončení všech vláken jsou získány detaily monitorovaných hodnot (`OrmMnMonitoredValueDetail`), jejichž stav `alerted` byl změněn. Tyto detaily jsou z této funkce vráceny zpět.

Nová funkce `processKpiGroup` je paralelně vykonávána pracovními vlákny. Vlákno vykonávající tuto funkci nejprve od hlavního vlákna přijme seznam ID všech KPI, která má zpracovat a z databáze načte potřebná data. Na každé KPI je zavolána funkce `processKpi`, která provede kontrolu, zda monitorovaná hodnota (`OrmMnParameterValue`) odpovídá pravidlu (`OrmMnKpiRule`). Výsledek této kontroly zapíše do příslušného detailu (`OrmMnMonitoredValueDetail`). Po skončení všech zpracování jsou provedené změny uloženy do databáze. Zároveň detaily, kterým byl změněn stav parametru `alerted` jsou vráceny zpět hlavnímu vláknu.

Následující kapitola popisuje testování nového řešení. Zaměřuje se na srovnání výkonů s původní verzí implementace. Také na novém řešení provádí výkonnostní testy ověřující jeho schopnost efektivně využívat všechna jádra.

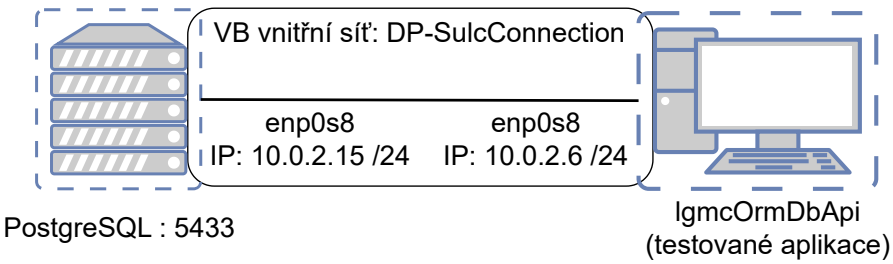
Kapitola 7

Testování

V předchozí kapitole 6 je popsána implementace nové verze funkce zpracování KPI. Tato kapitola se zaměřuje na srovnání výkonů této nové verze oproti původní. Pro tyto účely bylo vytvořeno virtuální testovací prostředí, které je popsáno v podkapitole 7.1. Samotné testování se pak skládá ze dvou částí. První část, popsána v podkapitole 7.2, provádí stejné testování jaké je v podkapitole 4.3 a prezentuje celkové zrychlení v porovnání s původní implementací. Druhá část popsaná v podkapitole 7.3 ověřuje, že nové řešení škáluje v souladu s očekáváním z podkapitoly 5.4.

7.1 Testovací prostředí

Pro účely testování bylo vytvořeno virtuální testovací prostředí. Toto prostředí se skládá ze dvou virtuálních strojů, které běží na virtualizačním prostředí VirtualBox[1]. Na obou strojích byl nainstalován operační systém Debian GNU/Linux 11. Na jeden ze strojů byl nainstalován databázový systém PostgreSQL společně s nástrojem pgAdmin[2]. Na tento stroj byla umístěna kopie testovací databáze, která byla dodána firmou Logimic. Na druhý stroj byla nainstalována platforma Node.js a byly na něj umístěny testované aplikace (s původní a optimalizovanou funkcí). Stroje byly propojeny vnitřní sítí přes dedikovaná virtuální síťová rozhraní (enp0s8). Bezpečnostní mechanismy databáze a síťová rozhraní byly nakonfigurovány tak, aby byl testovaným aplikacím umožněn přístup k databázi. Obrázek 7.1 obsahuje schéma sítě tohoto prostředí.



Obrázek 7.1: Schéma virtuálního testovacího prostředí

[1] https://www.virtualbox.org
[2] https://www.pgadmin.org

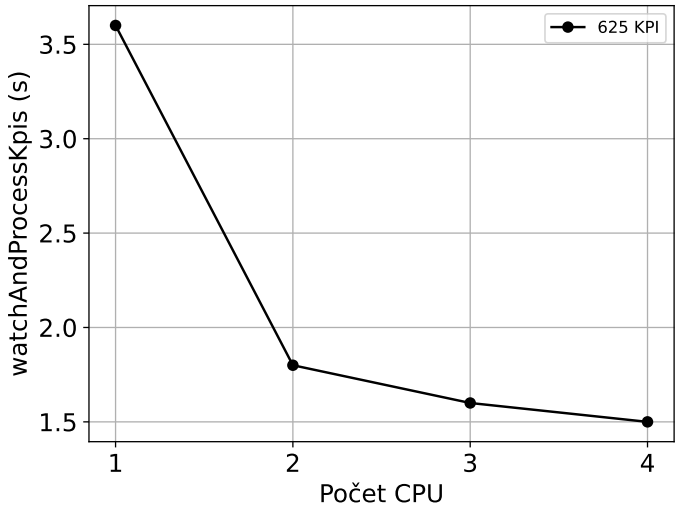
Hardware použitý na AWS Lambda není možné přímo replikovat, neboť nelze dohledat jeho specifikaci. Podle dokumentace [4] se počet dostupných jader odvíjí od množství zakoupené paměti. Ovšem podle [3] lze očekávat, že maximální použitelný počet jader je 6. Tato informace zajišťuje, že optimalizovaná aplikace nevytvoří příliš velké množství spojení s databází. V opačném případě by bylo nutné v implementaci omezit počet vytvořených vláken. Testovací prostředí tedy bylo nasazeno na PC s 4 jádrovým procesorem Intel Core i5-4590 3.30GHz a 16GB operační pamětí. Virtuálním PC bylo každému zprostředkováno 6GB operační paměti.

Takto vytvořené prostředí umožňuje dobře sledovat míru využití dostupných výpočetních prostředků obou strojů (např. pomocí nástroje `htop`). U testované aplikace je totiž potřeba sledovat, že během zpracování KPI dochází k vytížení všech jader. U databáze je pak potřeba ověřit, že během výpočtu nedochází k jejímu zahlcení.

Dodaná testovací databáze obsahuje 398 záznamů v `mn_kpi_rule` (vyhodnocovacích pravidel), 625 záznamů v `mn_kpi` (KPI), 420 záznamů v `rel_kpi_value_rule` (monitorovaných hodnot) v a 106 záznamů v `mn_device` (IoT senzorů).

7.2 Výsledky provedené optimalizace

Na obrázku 7.2 jsou zobrazeny doby běhů optimalizované funkce na různých počtech jader. Oproti původnímu řešení, jehož měření bylo provedeno v podkapitole 4.3, je vidět velmi významné zrychlení. Při využití všech 4 jader je zrychlení až 27násobné (z původních 50s na 1.8s).

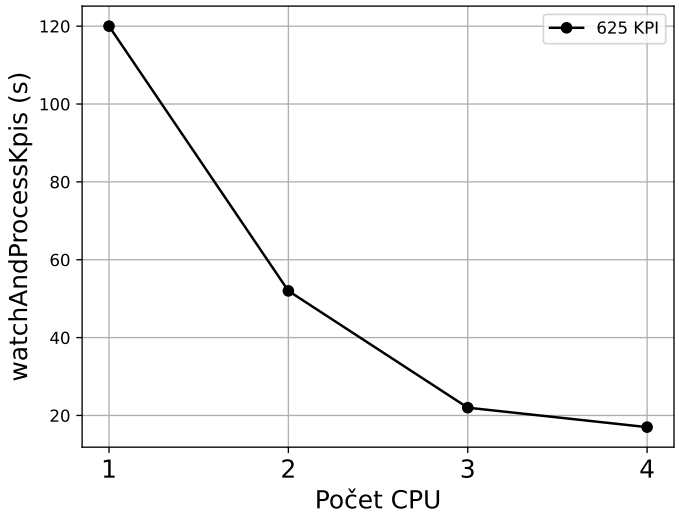


Obrázek 7.2: Výsledné doby běhů funkcí `watchAndProcessKpis` pro 625 KPI

Hlavním faktorem, který vedl k takto dramatickému zrychlení je správná komunikace s databází a eliminace sekvenčního zpracování a zbytečných dotazů. Podle počtu transakcí zaznamenaných na databázi jejich celkový počet klesl z původních 587 na 90 pro 4 vlákna a pro 1 vlákno dokonce na pouhých 33. Počet transakcí byl získán z databáze `pg_stat_database`, která je běžně součástí PostgreSQL serveru. Bohužel počet samotných dotazů tato databáze nezaznamenává. Zrychlení na jednom vlákně potvrzuje, že i v takovém případě může paralelní kód vést ke zvýšení efektivity zpracování. V tomto případě zřejmě vlákno mohlo zpracovávat KPI místo čekání na dokončení dotazů na databázi.

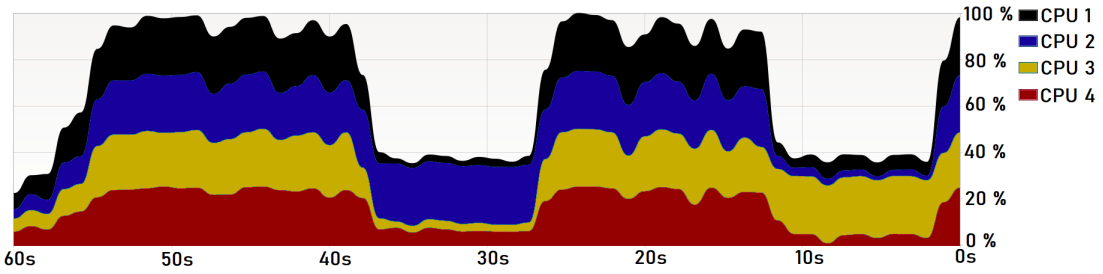
7.3 Výkonnostní analýza

Poslední věcí, kterou je potřeba zkontrolovat je schopnost využívat dostupné výpočetní prostředky. Testovací databáze poskytnutá firmou Logimic obsahuje omezené množství monitorovaných hodnot (420). Na tomto malém vzorku se škálování s počtem CPU projevuje jen mírně. Z toho důvodu byl počet monitorovaných hodnot uměle navýšen na 6750. Výsledky tohoto testování jsou na obrázku 7.3.



Obrázek 7.3: Doby běhů nové funkce `watchAndProcessKpis` s 6750 monit. hodnotami

Na tomto grafu je vidět, že s rostoucím počtem jader klesá celková doba výpočtu. Toto zrychlení odpovídá výsledkům, které byly očekávány v podkapitole 5.4 na základě Amdahlova zákona. Těchto výsledků je dosaženo díky téměř úplnému využití výpočetních prostředků, což znázorňuje graf na obrázku 7.4.



Obrázek 7.4: Graf využití výpočetních prostředků nové funkce `watchAndProcessKpis`

Během zpracování dochází v průměru k 90-100% využívání. Zároveň je možné vyčíst, že všechna jádra jsou rovnoměrně vytížena. V úsecích mezi iteracemi zpracování dochází k aktivnímu čekání, které je prováděno hlavním vláknem. Toto chování však slouží pouze pro testovací účely. V produkčním prostředí je zpracování časováno pomocí časovače AWS Lambda.

Kapitola 8

Závěr

Tato diplomová práce se zabývala optimalizací funkce pro zpracování klíčových indikátorů výkonu na platformě Smart City od firmy Logimic. Výstupem práce je nová funkce, která správně využívá dostupné výpočetní prostředky a plánuje komunikaci s databází pomocí knihovny pro objektově relační mapování. Během analýzy původní funkce byly objeveny nedostatky především v těchto dvou zmíněných oblastech.

Zpracování KPI probíhalo sekvenčně, což omezovalo zpracování pouze na jedno výpočetní vlákno. Analýza závislostí prováděných operací vedla k nalezení více paralelně zpracovatelných úseků. Díky tomu bylo možné do procesu zpracování zapojit více výpočetních vláken, mezi která byly tyto úseky rozděleny, a celý proces tak významně zrychlit.

Druhým odhaleným nedostatkem byla neefektivní komunikace s databází. Ve funkci docházelo k manuálnímu sestavování SQL dotazů vedoucímu k nepřehlednostem ve zdrojovém kódu. Výsledkem byla tvorba nadbytečných databázových transakcí, které načítaly již známá nebo odvoditelná data. Zároveň docházelo k zasílání dotazů v rámci cyklu, což množství těchto transakcí ještě navyšovalo. Efektivní komunikace byla zajištěna využitím objektově relačního mapování skrze knihovnu TypeORM, která pomohla snížit celkový počet provedených transakcí řádově ze stovek na několik desítek.

Průběžné testování odhalilo, že běžný způsob použití knihovny TypeORM není možné použít v paralelním prostředí, neboť kontext spojení s databází je omezen vždy pouze na jedno vlákno. Tento problém byl vyřešen použitím samostatného spojení s databází (samostatné instance této knihovny) pro každé výpočetní vlákno. Nicméně vzhledem k očekávanému počtu použitelných jader v produkčním prostředí je jisté, že množství paralelních spojení nebude tak velké, aby negativním způsobem ovlivňovalo výkon databáze.

Optimalizovaná funkce z této práce je připravena na reálné využití na platformě Smart City. Do budoucna je možné uvažovat nad možnými způsoby snížení počtu potřebných spojení s databází pro případ rostoucího počtu jader a výpočetních vláken. Může se však stát, že časem bude v použité knihovně TypeORM implementována přímá podpora pro paralelní prostředí. Pokud by se tak stalo a zároveň by byl zachován princip připojení na základě názvu spojení, pak bude nová implementace na toto vylepšení připravena a bude možné ji využívat bez zásahu do zdrojových kódů.

U platformy Smart City lze očekávat její postupný rozvoj a především přibývání nových parametrů, které musí být zpracovávány. I přes provedenou optimalizaci se stane, že zpracování postupně časem překročí přijatelnou dobu. V takové situaci bude vhodné změnit samotný způsob zpracování KPI. Nyní jsou vždy zpracovávána všechna, která se nachází v systému. Pro různá KPI se ovšem liší doba za kterou je nutné provést jejich aktualizaci. Například enviromentální by stačilo aktualizovat jednou za den, zatímco infrastrukturní

jednou za hodinu. Pokud by s tímto počítal i samotný systém, došlo by v daný okamžik k redukci počtu zpracovávaných KPI. Jednalo by se i o relativně snadno implementovatelné řešení. V databázovém systému by se u každého KPI navíc uchovával interval jeho zpracování. Na základě parametru poslední provedené aktualizace by bylo možné vypočítat čas příští. V dané iteraci zpracování by byly filtrovány jen ty, u nichž uplynul daný interval od poslední aktualizace.

Výsledky práce byly publikovány na studentské konferenci Excel@FIT 2023 [43].

Literatura

[1] Agbali, M., Trillo, C., Fernando, T., Ibrahim, I. A. a Arayici, Y. Conceptual Smart City KPI Model: A System Dynamics Modelling Approach. In: *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. 2018, s. 163–171. DOI: 10.1109/WorldS4.2018.8611565. Dostupné z: https://ieeexplore.ieee.org/abstract/document/8611565.

[2] Angelakoglou, K., Nikolopoulos, N., Giourka, P., Svensson, I.-L., Tsarchopoulos, P. et al. A Methodological Framework for the Selection of Key Performance Indicators to Assess Smart City Solutions. *Smart Cities*. 2019, sv. 2, č. 2, s. 269–306. DOI: 10.3390/smartcities2020018. ISSN 2624-6511. Dostupné z: https://www.mdpi.com/2624-6511/2/2/18.

[3] AWS. *AWS Lambda now supports up to 10 GB of memory and 6 vCPU cores in AWS GovCloud (US) Regions*. 2021 [cit. 1.5.2023]. Online. Dostupné z: https://aws.amazon.com/about-aws/whats-new/2021/08/aws-lambda-supports-10-gb-memory-6-vcpu-cores-aws-govcloud-us-regions/.

[4] AWS. *Configuring Lambda function options*. 2023 [cit. 1.5.2023]. Online. Dostupné z: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html.

[5] Baer, J.-L. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 2010. ISBN 978-0-521-76992-1. Dostupné z: https://perpus.univpancasila.ac.id/repository/EBFT180172.pdf.

[6] Bosch, P., Jongeneel, S., Rovers, V., Neumann, H.-M., Airaksinen, M. et al. CITYkeys indicators for smart city projects and smart cities. *CITYkeys report*. 2017. Dostupné z: https://www.dataplan.info/img_upload/7bdb1584e3b8a53d337518d988763f8d/citykeys.pdf.

[7] Buntz, B. *The Top 20 Industrial IoT Applications*. IOT World Today, 2017 [cit. 27.11.2022]. Online. Dostupné z: https://www.iotworldtoday.com/2017/09/20/top-20-industrial-iot-applications/.

[8] Centers for Disease Control and Prevention. *About Global NCDs*. 2021 [cit. 27.11.2022]. Online. Dostupné z: https://www.cdc.gov/globalhealth/healthprotection/ncd/global-ncd-overview.html.

[9] Chourabi, H., Nam, T., Walker, S., Gil Garcia, J. R., Mellouli, S. et al. Understanding Smart Cities: An Integrative Framework. In: *2012 45th Hawaii International Conference on System Sciences*. 2012, s. 2289–2297. DOI: 10.1109/HICSS.2012.615. Dostupné z: https://ieeexplore.ieee.org/abstract/document/6149291.

[10] Colley, D., Stanier, C. a Asaduzzaman, M. The Impact of Object-Relational Mapping Frameworks on Relational Query Performance. In: *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. 2018, s. 47–52. DOI: 10.1109/iCCECOME.2018.8659222.

[11] DB-Engines. *DB-Engines Ranking*. 2022 [cit. 20.11.2022]. Online. Dostupné z: https://db-engines.com/en/ranking.

[12] Eckerson, W. W. *Performance dashboards: measuring, monitoring, and managing your business*. John Wiley & Sons, 2010. ISBN 978-0-471-72417-9.

[13] Genta, C., Lombardi, P., Mari, V. a Moghadam, S. T. Key Performance Indicators for Sustainable Urban Development: Case Study Approach. In: IOP Publishing. *IOP Conference Series: Earth and Environmental Science*. 2019, sv. 296, č. 1, s. 012009. Dostupné z: https://iopscience.iop.org/article/10.1088/1755-1315/296/1/012009/meta.

[14] Han, J., Pei, J. a Tong, H. *Data Mining: Concepts and Techniques*. Elsevier Science, 2022. The Morgan Kaufmann Series in Data Management Systems. ISBN 9780128117613.

[15] Hara, M., Nagao, T., Hannoe, S. a Nakamura, J. New key performance indicators for a smart sustainable city. *Sustainability*. MDPI. 2016, sv. 8, č. 3, s. 206. Dostupné z: https://www.mdpi.com/2071-1050/8/3/206.

[16] Hasan, M. *State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally*. IoT Analytics, 2022 [cit. 27.11.2022]. Online. Dostupné z: https://iot-analytics.com/number-connected-iot-devices/.

[17] Hlava, J. *Zpracování a ukládání IoT dat do relační databáze s využitím cloudových služeb*. Brno, CZ, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/24979/.

[18] influxdata. *Time series database (TSDB) explained*. 2022 [cit. 27.11.2022]. Online. Dostupné z: https://www.influxdata.com/time-series-database/.

[19] Jansen, R., Vane, V. a Wolff, I. de. *TypeScript: Modern JavaScript Development*. Packt Publishing, 2016. ISBN 9781787287594. Dostupné z: https://books.google.cz/books?id=yc7cDgAAQBAJ.

[20] Kirimtat, A., Krejcar, O., Kertesz, A. a Tasgetiren, M. F. Future trends and current state of smart city concepts: A survey. *IEEE access*. IEEE. 2020, sv. 8, s. 86448–86467. Dostupné z: https://www.researchgate.net/publication/341166746_Future_Trends_and_Current_State_of_Smart_City_Concepts_A_Survey.

[21] Likness, J. a Phillip, C. *Serverless apps: Architecture, patterns, and Azure implementation*. Microsoft Developer Division, .NET, and Visual Studio product teams, 2022 [cit. 10.12.2022]. Online. Dostupné z: https://aka.ms/serverlessbookpdf.

[22] Microsoft. *Efficient Querying*. 2022 [cit. 25.11.2022]. Online. Dostupné z: https://learn.microsoft.com/en-us/ef/core/performance/efficient-querying.

[23] Microsoft. *Migrations*. 2023 [cit. 1.2.2023]. Online. Dostupné z: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli.

[24] Neo4J. *What is a Graph Database?* 2022 [cit. 27.11.2022]. Online. Dostupné z: https://neo4j.com/developer/graph-database/.

[25] Node.js. *Node.js v19.6.1 documentation*. 2023 [cit. 10.2.2023]. Online. Dostupné z: https://nodejs.org/dist/v19.6.1/docs/api/.

[26] npm trends. *Bookshelf vs mikro-orm vs objection vs prisma vs sequelize vs sequencify vs typeorm vs waterline*. 2022 [cit. 25.11.2022]. Online. Dostupné z: https://npmtrends.com/bookshelf-vs-mikro-orm-vs-objection-vs-prisma-vs-sequelize-vs-sequencify-vs-typeorm-vs-waterline.

[27] Parmenter, D. *Key performance indicators: developing, implementing, and using winning KPIs*. John Wiley & Sons, 2015. ISBN 9781118925102.

[28] Picioroagă, I.-I., Eremia, M. a Sănduleac, M. SMART CITY: Definition and Evaluation of Key Performance Indicators. In: *2018 International Conference and Exposition on Electrical And Power Engineering (EPE)*. 2018, s. 217–222. DOI: 10.1109/ICEPE.2018.8559763.

[29] Pokorný, J. a Halaška, I. *Databázové systémy*. Praha : Česká informatická společnost, 2002. ISBN 80-900853-9-3.

[30] Prisma. *Prisma documentation*. 2022 [cit. 25.11.2022]. Online. Dostupné z: https://www.prisma.io/docs/.

[31] python. *Cpython*. GitHub, 2022 [cit. 2.4.2023]. Dostupné z: https://github.com/python/cpython/blob/3.10/Lib/multiprocessing/pool.py#L480.

[32] Rajab, H. a Cinkelr, T. IoT based Smart Cities. In: *2018 International Symposium on Networks, Computers and Communications (ISNCC)*. 2018, s. 1–4. DOI: 10.1109/ISNCC.2018.8530997. Dostupné z: https://ieeexplore.ieee.org/abstract/document/8530997.

[33] Sequelize. *Sequelize v6 documentation*. 2022 [cit. 25.11.2022]. Online. Dostupné z: https://sequelize.org/docs/v6/.

[34] Tealmakers. *Co jsou Leading vs Lagging KPI*. 2020 [cit. 29.11.2022]. Online. Dostupné z: https://www.tealmakers.cz/blog/leading-vs-lagging-kpi.

[35] threads.js. *THREADS.JS Make web workers & worker threads as simple as a function call*. 2019 [cit. 20.3.2023]. Online. Dostupné z: https://threads.js.org/.

[36] Tiwari, S. *Professional NoSQL*. Wiley, 2011. EBL-Schweitzer. ISBN 9781118167809. Dostupné z: https://books.google.cz/books?id=tv5iO9MnObUC.

[37] TypeORM. *TypeORM documentation*. 2022 [cit. 25.11.2022]. Online. Dostupné z: https://typeorm.io/.

[38] TypeORM. *Many-to-many relations*. 2023 [cit. 20.3.2023]. Online. Dostupné z: https://typeorm.io/many-to-many-relations.

[39] TypeORM. *Migrations*. 2023 [cit. 1.2.2023]. Online. Dostupné z: https://typeorm.io/migrations.

[40] Washburn, D., Sindhu, U., Balaouras, S., Dines, R. A., Hayes, N. et al. Helping CIOs understand "smart city" initiatives. *Growth*. 2009. Dostupné z: https://s3-us-west-2.amazonaws.com/itworldcanada/archive/Themes/Hubs/Brainstorm/forrester_help_cios_smart_city.pdf.

[41] World Health Organization. *Urban health*. 2022 [cit. 27.11.2022]. Online. Dostupné z: https://www.who.int/health-topics/urban-health.

[42] Xia, C., Yu, G. a Tang, M. Efficient Implement of ORM (Object/Relational Mapping) Use in J2EE Framework: Hibernate. In: *2009 International Conference on Computational Intelligence and Software Engineering*. 2009, s. 1–3. DOI: 10.1109/CISE.2009.5365905.

[43] Šulc, O. *Optimizing KPI Processing of Smart Cities Using Multithreading in Node.js*. Vysoké učení technické v Brně, Fakulta informačních technologií, 2023. Dostupné z: https://excel.fit.vutbr.cz/submissions/2023/036/36.pdf.

Příloha A

Obsah přiloženého paměťového média

- xsulco01_dp.pdf – tento dokument
- latex – složka obsahující soubory pro sestavení tohoto dokumentu
- README.md – soubor s podrobným popisem média a instrukcemi pro spuštění
- src – složka obsahující implementaci modulu s novou verzí funkce pro zpracování KPI
- src_old – složka obsahující původní implementaci modulu pro zpracování KPI
- test – složka obsahující virtuální testovací prostředí pro VirtualBox