VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATIZOVANÉ VELKOOBJEMOVÉ ZPRACOVÁNÍ TESTOVACÍCH VÝSLEDKŮ

AUTOMATIC LARGESCALE PROCESSING OF TESTING RESULTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **RICHARD KLEM**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Klem Richard**
Program: Informační technologie
Název: **Automatizované velkoobjemové zpracování testovacích výsledků**
Automatic Largescale Processing of Testing Results
Kategorie: Analýza a testování softwaru

Zadání:

1. Seznamte se s formáty výstupů z běžných testovacích nástrojů (JUnit, Jest, Go testing, Pytest, apod.) a s aktuálními možnostmi jejich zpracování a vizualizace (raw data nebo BI vizualizace).
2. Pro vybrané nástroje z bodu 1 prostudujte již existující aplikační řešení a tato řešení porovnejte a vyhodnoťte jejich vlastnosti.
3. Na základě poznatků z bodu 2 navrhněte vlastní řešení zpracování testovacích výsledků.
4. Navržené řešení implementujte za použití relační databáze a ověřte testovacími daty.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Harrison, G. (2015). *Next Generation Databases: NoSQLand Big Data*. Apress.
- Petrov, A. (2019). *Database Internals: A Deep Dive into How Distributed Data Systems Work*. O'Reilly Media, Inc.
- Turkington, G., Deshpande, T., & Karanth, S. (2016). *Hadoop: Data Processing and Modelling*. Packt Publishing Ltd.
- Okken, B. (2017). *Python Testing with Pytest: Simple, Rapid, Effective, and Scalable*. Pragmatic Bookshelf.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

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

Vedoucí práce: **Hruška Tomáš, prof. Ing., CSc.**
Konzultant: Dolíhal Luděk, Ing., CODASIP
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 24. října 2020

Abstrakt

Tato práce se zabývá problematikou automatizovaného zpracování velkého objemu dat v prostředí relační databáze. Práce se zaměřuje na aktuální situaci ve firmě Codasip a řeší nedostatečně rychlé vykonávání aktualizací materializovaných pohledů. Výsledkem práce je návrh na úpravy aktuálního systému, tento návrh úspěšně řeší problémy s pomalým zpracováním dat. Implementované řešení pak přináší průměrné zrychlení aktualizace materializovaných pohledů o 93 % a u některých materializovaných pohledů zrychlení o 99 %. Toto řešení pomůže firmě Codasip zvýšit efektivitu vývoje procesorových jader, ale i dalších produktů. Zároveň tato práce může sloužit jako případová studie optimalizace pomalých databázových dotazů při zpracování velkého objemu dat.

Abstract

This thesis deals with the issue of automated processing of large volumes of data in a relational database environment. The thesis focuses on the current situation in the Codasip company and solves the insufficiently fast implementation of materialized views updates. The result of the thesis is a draft to modify the current system. This draft successfully solves problems with slow data processing. The implemented solution then brought an average acceleration of the update of materialized views by 93 % and for some materialized views an acceleration of 99 %. This solution will help Codasip increase the efficiency of processor core development, as well as their other products. At the same time, this thesis can serve as a case study of optimizing slow database queries when processing large amounts of data.

Klíčová slova

testování, automatizace, testovací výsledky, report, MySQL, databáze, databázové schéma, SQL dotazy, pytest, zpracování dat, optimalizace, partitioning, materializovaný pohled, vizualizace, Metabase

Keywords

testing, automation, test results, report, MySQL, databases, database schema, SQL queries, pytest, data processing, optimization, partitioning, materialized view, visualization, Metabase

Citace

KLEM, Richard. *Automatizované velkoobjemové zpracování testovacích výsledků*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, CSc.

Automatizované velkoobjemové zpracování testovacích výsledků

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Další informace mi poskytli Ing. Luděk Dolíhal, Ph.D a Ing. Jiří Hynek, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Richard Klem
10. května 2021

Poděkování

Děkuji svému vedoucímu, profesoru Tomáši Hruškovi za vedení mé práce, dále doktoru Luďkovi Dolíhalovi za odbornou konzultaci, doktoru Jiřímu Hynkovi za pomoc a rady k psaní technické zprávy. Firmě Codasip děkuji za možnost vypracovat bakalářskou práci na takto zajímavé a praktické téma a také zaměstnancům firmy Codasip za ochotné poskytnutí drobných konzultací a odpovědí na mé otázky.

Obsah

Kapitola 1

Úvod

Při automatizovaném testování se může generovat až několik milionů testovacích záznamů za jeden testovací běh. Pokud je testování založeno na parametrizovaných testech, kdy je na vstupu několik kombinací parametrů, nebo se používají náhodně generované vstupní parametry, je objem výsledků ještě větší. Výsledky z testování zpravidla interpretujeme bezprostředně po dokončení testů, ale zároveň je žádoucí výsledky uložit, abychom z nich později mohli získávat statistická data. Zde vyvstává otázka, jaký je vhodný způsob uložení a zpracování těchto dat. Nejrozšířenější způsob uložení dat určených k dalšímu zpracování je uložení do databází. Nejrozšířenějším typem databází jsou dle Google Trends[1] a žebříčku DB-Engines[2] relační databáze. S ohledem na typ dat, která jsou v databázi uložena, může taková databáze snadno přesáhnout i 100 GB dat. Čtení takto objemných dat a zejména následná manipulace, vyhledávání a třídění může být velmi náročné. Škálování takovéhoto systému implikuje vynaložení finančních prostředků na pořízení dalších výpočetních jednotek nebo na pořízení výkonnějších zařízení. Dalším možným řešením je optimalizace daného systému, kdy se zvýší efektivita systému za použití aktuálních výpočetních zdrojů.

Tato práce řeší problém pomalých dotazů nad relační databází, která uchovává data z testování ve firmě Codasip. Cílem práce je na základě analýzy stávající situace ve firmě Codasip urychlit provádění aktualizace pohledů a dotazů nad testovacími výsledky. Výsledkem této práce je řešení, které naplňuje cíl a splňuje zadaná netechnická kritéria. Důležitým kritériem je mimo jiné i snadná integrovatelnost navrženého řešení ve firemním prostředí firmy Codasip nebo preferované využití volně dostupného a open–source softwaru[3].

V kapitolách 2 a 3 rozebírám aktuální přístupy k testování a nástroje, které se při testování využívají. V kapitole 4 představuji problematiku ukládání a vizualizace dat a jaké jsou vlastnosti jednotlivých možností. Zaměřil jsem se na výstupy z testovacího nástroje pytest a relační databáze, protože tyto nástroje se využívají ve firmě Codasip při testování a zpracování výsledků. Aktuální situaci a přístup k problematice ve firmě Codasip rozebírám v kapitole 5. Návrh mého řešení je popsán v kapitole 6, kde popisuji konkrétní kroky vedoucí k zrychlení zpracování testovacích výsledků. Implementace mého řešení je popsána v kapitole 7. O způsobech testování mého řešení a dosažených výsledcích pojednávám v kapitole 8. Práce je zakončena kapitolou 9.

[1] https://trends.google.com/trends/explore?date=today%205-y&q=sql,nosql,mongodb,mysql,elasticsearch
[2] https://db-engines.com/en/ranking
[3] programové vybavení s otevřeným zdrojovým kódem

Kapitola 2

Testování software

Tato kapitola čerpá informace z 4. vydání publikace *Software Testing* [7]. Typický životní cyklus vývoje software (česky programové vybavení) zahrnuje i část, během které se daný produkt testuje. Pro pochopení, co to je testování, je potřeba znát některé pojmy či definice. Testování je dle Hamblinga a kolektivu [7] chápáno jako proces, který zjišťuje, zda testovaný program, aplikace nebo systém neobsahuje chyby, chová se dle očekávání, splňuje definované požadavky na funkčnost i na nefunkční vlastnosti. Funkční vlastnosti jsou například správnost výstupu při daných vstupech. Nefunkční vlastnost je například čas (doba trvání), ve kterém těchto výsledků dosáhneme. Hambling [7] zavádí i další jiné dělení testování. Statické testování je testování, při kterém se nevykonává kód, ale pouze se staticky tento kód hodnotí, posuzuje a kontroluje. Statické testování umožňuje velmi brzy odhalit řadu chyb. Dynamické testování je naopak od statického založeno na provádění kódu programu. Jedná se tedy o testování programu v určitém kontextu. Velké projekty jsou složité systémy a je potřeba provádět testování na více úrovních, které odpovídají určitým úrovním testovaného systému. Více informací o těchto úrovních se lze dočíst v kapitole 2.1. Při vývoji software je potřeba brát v úvahu míru automatizace testování tak, aby zvolený způsob nebo kombinace vyhovovala potřebám projektu. Více o automatizaci testování v sekci 2.3.

2.1 Úrovně testů

Při testování softwaru se můžeme setkat s několika úrovněmi testování. Každá úroveň je v něčem důležitá, ale ne všechny projekty vyžadují všechny druhy testů. V následujících podsekcích proberu jednotlivé úrovně testů.

2.1.1 Jednotkové testy (testy komponent)

Na nejnižší úrovni nalezneme tzv. jednotkové testy (anglicky *unit tests*) nebo testy komponent (anglicky *component tests*). Jednotkovými testy se testují jednotlivé části programu – komponent (proto i název testy komponent). Jednotkové testy můžou testovat funkční nebo nefunkční požadavky. Funkčním požadavkem je například to, že volání určité funkce vrací očekávané výstupní hodnoty. Správná práce s pamětí je příklad nefunkčního požadavku. Hlavním cílem jednotkových testů je odhalit, jestli jsou nejmenší části programu napsány korektně, prohloubit důvěru v nejmenší části programu. Zároveň se tak jasněji oddělí testování na vyšších úrovních. Vyšší úrovně testů se pak nezabývají nejmenšími detaily, ale soustředí se na složitější problémy.

Mezi problémy, které lze odhalit jednotkovým testováním patří například:

- komponenta produkuje chybný výstup,
- komponenta nevrací výstup, i když by měla,
- kód komponenty je nečitelný (odhalí se při psaní testů),
- kód komponenty obsahuje logické chyby.

Existují dva přístupy k psaní jednotkových testů. První přístup je takový, kdy se nejprve píše kód komponenty a až následně se píšou jednotkové testy. Při využití druhého přístupu se jako první píšou jednotkové testy a až následně se píše samotný kód komponenty. Druhý způsob je znám jako *Test-driven development (TDD)*, tedy vývoj řízený testy. Za tento druh testů odpovídá vývojář, který si jednotkovými testy ověřuje, že jeho nově přidaný nebo upravený kus kódu funguje dle očekávání. Když je na této úrovni vše v pořádku, přichází na řadu testy komponent.

2.1.2 Regresní testy

Regresní testy (anglicky *regressison tests*) slouží k zpětnému testování. Kontrolují, jestli se přidáním nové funkce nebo upravením stávající nepoškodila funkčnost celé aplikace. Regresní testy lze aplikovat na všechny úrovně testování, nejčastěji jsou ale použity na úrovni jednotkových testů. Množina testů regresního testování se skládá z testů z předchozí iterace vývoje a z nových testů nových přírůstků.

2.1.3 Další druhy testů

Další úrovně testování nejsou tolik důležité pro tuto práci, protože tyto úrovně již neprodukují tak velké množství dat a nejsou tedy z tohoto hlediska důležité jako testy jednotkové a regresní. Informace kromě kontrolních testů opět pochází z výše zmíněné publikace [7].

Integrační testy (anglicky *integration tests*) slouží k otestování, zda všechny komponenty spolu komunikují korektně a zda celá aplikace běží v pořádku. Integrační test může například testovat komunikace mezi přihlašovací stránkou, resp. stisknutí tlačítka Login, a back-end aplikací, která autorizuje uživatele.

Kontrolní testy (anglicky *sanity tests*) jsou v publikaci *Software Testing and Continuous Quality Improvement* od William E. Lewise [16] definovány jako testy sloužící k nenáročné, rychlé kontrole funkčnosti. Cílem kontrolních testů je otestovat produkt kompletně, ale rychle. Kontrolní testy jsou často podmnožinou regresních testů a pouští se po integračních testech, tedy již nad stabilně sestaveným produktem. Po úspěšně zakončeném integračním testování se při menších změnách pouští místo rozsáhlých regresních testů pouze tzv. *sanity check*, tedy kontrolní testy, aby se ověřilo, že chyby odhalené při minulém testování jsou již opraveny a neobjevily se nové.

Systémové testování (anglicky *system testing*) slouží pro otestování hotového produktu v produkčním prostředí. Často tento druh testování provádí nezávislí testeři, kteří se neúčastní vývoje, aby byla zajištěna věrohodná simulace uživatelského použití. Se systémovým testováním jsou spojeny i akceptační testy (anglicky *user acceptance tests*, které slouží k otestování aplikace koncovými uživateli. Nejedná se tedy již o testery, ale o cílové uživatele. Tento druh testů neslouží již primárně na to, aby odhaloval chyby (kterých by v této fázi vývoje mělo být v produktu co nejméně), ale aby se otestovalo, jak produkt splňuje uživatelská očekávání.

2.2 Vývojové přístupy k testování

Kromě výše zmíněných úrovní testů existují i jiné způsoby dělení. Viktor Farcic a Alex Garcia [4] zavádějí další způsob dělení a to na tzv. *Black-box* a *White-box* testování.

2.2.1 Black-box testování

Černá skříňka (anglicky *black-box*) využívá metody odstínění vývojáře od detailů implementace komponenty a vývojář pracuje pouze s vstupy a výstupy dané komponenty. Testy zpravidla testují korektnost dvojice vstup-výstup. Jedná se tedy o funkční testování.

Výhody, které *black-box* testování nabízí:

- odstínění testerů od implementace,
- pro testování není nutný přístup ke kódu,
- lze takto testovat větší kusy programu,
- při změně implementace uvnitř černé skříňky není nutné psát nové nebo upravovat stávající testy.

Nevýhody, které *black-box* testování obnáší:

- omezené možnosti pokrytí zdrojového kódu,
- riziko neodhalení chyby – kvůli skryté implementaci.

Testování pomocí metodiky černé skříňky je nejčastěji realizováno na úrovni jednotkových testů.

2.2.2 White-box testování

Bílá skříňka (anglicky *white-box*) pracuje s přístupem, kdy vývojář zná implementaci dané komponenty a píše testy s ohledem na tuto skutečnost. V anglicky psané literatuře se můžeme setkat i s názvy *clear-box* nebo *glass-box*. Tyto názvy lépe vystihují podstatu této vývojové metodiky. Výhody, které *white-box* testování nabízí jsou:

- snazší odhalování chyb,
- komplexnější pokrytí zdrojového kódu,
- možnost optimalizace nebo úpravy zdrojového kódu (v *black-box* testování bylo nemožné, kvůli skrytému kódu).

Nevýhody, které *white-box* testování obnáší:

- od vývojáře se vyžaduje porozumění kódu,
- je nutný přístup ke zdrojovému kódu,
- při změně kódu, je zpravidla nutná okamžitá změna v testech.

Obě výše zmíněné metodiky lze dobře automatizovat.

2.3 Automatizace testů a testování

Brian Hambling a kolektiv [7] definují kromě úrovní testů (sekce 2.1) i způsob, jakým se testy provádí. Způsoby, na které lze testy z tohoto pohledu rozdělit existují podle autorů dva – manuální a automatické testování. Každý z přístupů má své výhody i nevýhody a specifické požadavky na zavedení, provoz a údržbu.

Manuální testy je možné bez dalších nástrojů a programů. Testy provádí tester a analyzuje výstupy testů. Výhodou manuálního testování je nižší cena (není potřeba pořizovat nebo vyvíjet automatizované testovací nástroje), vyšší flexibilita, protože tester může bezprostředně reagovat na situace a také lze lépe detekovat chyby v reálném uživatelském prostředí. Mezi nevýhody manuálního testování patří přirozeně časová neefektivnost v případě opakujících se testů (například regresní 2.1.2) nebo v případě velkého množství testů. Další z nevýhod je vliv subjektivity a expertízy testera na výsledky testů, špatná reprodukce problémů apod.

Druhým přístupem k testování jsou automatizované testy. Díky automatizovaným testům lze následně zavést automatizaci testování. Tyto dva podobné pojmy s velmi rozdílným významem vysvětluje článek *Test Automation vs. Automated Testing* [24]. Automatizované testy jsou testy, které vykonávají akce testu automaticky, samostatně pouze pomocí počítače podle předem definovaných pravidel. Při manuálním testování právě tyto akce vykonává tester. Oproti tomu automatizace testování spočívá v automatizaci samotné správy a řízení testování.

Mezi výhody automatizovaných testů patří efektivní regresní testování nebo testování velkého množství testů. Reprodukce odhalených chyb je u automatizovaných testů snazší než u manuálních. Výsledky automatizovaných testů jsou objektivní, nezaujaté a vždy stejně přesné. Nevýhodou automatizovaných testů je například vyšší počáteční finanční a časovou zátěž. Nástroje pro automatizované testování je nutné nejprve zakoupit nebo vyvinout, následně zprovoznit a po celou dobu jejich provozu udržovat. Některé procesy nebo akce, které je potřeba testovat lze testovat automatizovaně s obtížemi nebo dokonce vůbec. Výhodou i nevýhodou zároveň je, že do průběhu automatizovaných testů lze obtížně zasahovat.

V článku *Best Practices for Test Automation / 2021 Tester's Checklist* [11] společnosti *Katalon* se lze dočíst, že ne všechny testy jsou vhodné pro automatizaci. Mezi vhodné testy k automatizaci autoři řadí mimo jiné i jednotkové a regresní testy. V článku se zájemci mohou dočíst i popis sedmi kroků vedoucích k nasazení automatizovaného testování.

Kapitola 3

Testovací nástroje a jejich výstupy

V anglickém jazyce je rozšířený název *framework* (česky aplikační rámec) pro nástroje nebo služby, které poskytují ucelenou sadu programů, aplikací, příkazů a další řadu služeb, které umožňují určitý typ akce. V případě testování se tedy setkáváme s tzv. testovacími frameworky. Tento druh softwaru slouží k snadnému návrhu, implementaci a spouštění testů, zpravidla automatizovaných.

V publikaci [8] zavádí Paul Hamill celou třídu frameworků pracujících s jednotkovými testy jako hlavním druhem testů, kterou autor nazval *xUnit Family*. Do této skupiny patří frameworky jako:

- JUnit – slouží jako vzorová implementace frameworku pro jednotkové testy a více jsem o tomto frameworku napsal v sekci 3.1,
- CppUnit – jedná se o implementaci JUnit frameworku pro jazyk C++[1] nebo
- utPLSQL – framework napsaný v jazyce PL/SQL[2] a určený pro jednotkové testy v tomto databázovém jazyce firmy Oracle[3].

Všechny frameworky z rodiny xUnit mají společné charakteristiky popsané dříve v této práci v sekci 2.1.1. Podle Arnaba Roye Chowdhuryho [1] patří mezi pět nejlepších frameworků pro jazyk Python frameworky PyUnit a pytest. PyUnit patří do rodiny xUnit frameworků a má také jejich společné vlastnosti. Nástroj pytest je testovací framework, který není specializovaný na určitý druh testů. V odlišné rovině pak stojí například framework Selenium[4], který slouží pro automatizaci uživatelských akcí v prostředí internetových prohlížečů. Pro tuto práci jsou ale nejvíce relevantní právě jednotkové testy a tedy i frameworky určené pro tento druh testování. Pro uvedení detailnějších příkladů jsem v následujících dvou sekcích více rozebral frameworky JUnit (sekce 3.1) a pytest (sekce 3.2).

Nejdůležitějším aspektem daného frameworku je pro pro tuto práci výstup, který framework generuje, protože tento výstup slouží jako vstup pro další zpracování a analýzu. Syntax frameworků nebo způsob použití nejsou pro tuto práci stěžejní. V následujícím textu jsem uvedl krátké ukázky testů a výstupů, které daný framework vygeneroval.

[1] http://www.cplusplus.org/
[2] https://www.oracle.com/database/technologies/application-development-pl/sql.html
[3] https://www.oracle.com/index.html
[4] https://www.selenium.dev/

3.1 JUnit

JUnit[5] je jednoduchý testovací framework pro psaní a spouštění testů. Jak název napovídá, framework se zaměřuje na jednotkové testy. Framework lze využít k testování například v jazyce Java, Kotlin nebo Javascript. Každý test je psán jako samostatná metoda testovací třídy. Každá tato metoda reprezentuje jeden konkrétní scénář běhu testovaného programu. Ověření správnosti je prováděno pomocí porovnávání očekávaného a reálného výstupu, tedy tak, jak je pro jednotkové funkční testování obvyklé. [4]

Zdrojový kód JUnit (sekce 3.1) testů lze vidět níže. Původní kód [20] z ukázky jsem zjednodušil pro potřeby této práce. Testy jsou ilustrativního charakteru a testují pouze operaci sčítání. K parametrizaci testů a dalším akcím jsou použity JUnit anotace[6]. Základním formátem výstupu je prostý text. Jak vypadá výstup, který je uživateli zobrazen na standardní výstup je zobrazeno v kódu 3.2. JUnit generuje i XML[7] *report* (česky hlášení), jehož příklad lze vidět v ukázce 3.3, a HTML[8] *report*, jehož ukázku můžete vidět na obrázku 3.1 za použití nástroje Gradle[9]. Výstupní soubory sestavení a testování ze složky `build` lze použít ke generování vlastních *reportů* nebo pro další vlastní zpracování testovacích výsledků.

```
class CalculatorTests {

	@Test
	@DisplayName("1 + 1 = 2")
	void addsTwoNumbers() {
		Calculator calculator = new Calculator();
		assertEquals(2, calculator.add(1, 1), "1 + 1 should equal 2");
	}

	@ParameterizedTest(name = "{0} + {1} = {2}")
	@CsvSource({
			"0, 1, 0",
			"49, 51, 100",
	})
	void add(int first, int second, int expectedResult) {
		Calculator calculator = new Calculator();
		assertEquals(expectedResult, calculator.add(first, second),
			() -> first + " + " + second + " should equal " + expectedResult);
	}
}
```

Kód 3.1: Zdrojový kód dvou testů napsaných ve frameworku JUnit. Druhý test je ukázkou parametrizovaného testu.

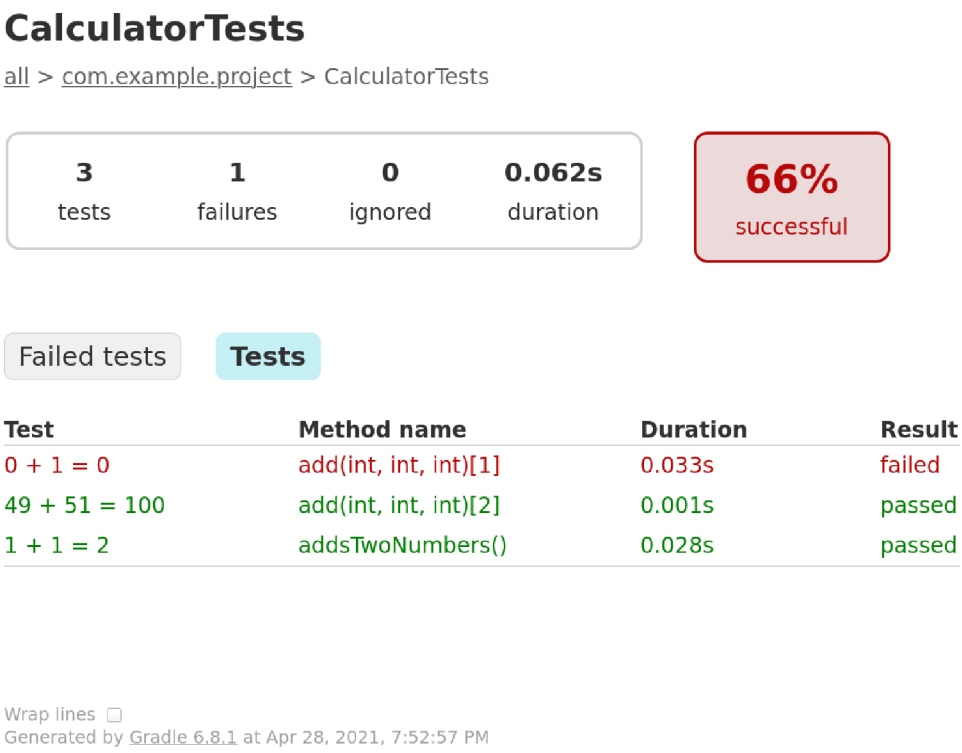
```
> Task :test
CalculatorTests > 1 + 1 = 2 PASSED
CalculatorTests > 0 + 1 = 1 PASSED
CalculatorTests > 49 + 51 = 100 PASSED
BUILD SUCCESSFUL in 1s
3 actionable tasks: 3 executed
```

Kód 3.2: Příklad standardního výstupu JUnit testu, který je tištěn do konzole. Jedná se o jednoduchý přehled základních informací o výsledku testovacího běhu.

[5] https://junit.org/junit5/
[6] https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations
[7] Extensible Markup Language
[8] Hypertext Markup Language
[9] https://gradle.org/

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="com.example.project.CalculatorTests" tests="3" skipped="0" failures="0" errors="0"
          timestamp="2021-04-28T17:49:49" hostname="pc322" time="0.097">
    <properties/>
    <testcase name="1 + 1 = 2" classname="com.example.project.CalculatorTests" time="0.029"/>
    <testcase name="0 + 1 = 1" classname="com.example.project.CalculatorTests" time="0.026"/>
    <testcase name="49 + 51 = 100" classname="com.example.project.CalculatorTests" time="0.002"/>
    <system-out>
        <![CDATA[]]>
    </system-out>
    <system-err>
        <![CDATA[]]>
    </system-err>
</testsuite>
```

Kód 3.3: Příklad XML *reportu*, který nabízí základní informace o testovacím běhu a informace o výsledku testování.



Obrázek 3.1: Ukázka HTML *reportu* jako výstupu z JUnit testovacího běhu. *Report* nabízí souhrnné informace o testovacím běhu i konkrétních testech. JUnit HTML *report* umožňuje procházet tento *report* podobně jako webovou stránku, a lze tak zobrazit i podrobný výpis chybového hlášení k jednotlivých neúspěšným testům.

3.2 pytest

Nástroj pytest[10] je určen pro testování programů/skriptů v jazyce Python. Tento nástroj není specializovaný na konkrétní druh testů a umožňuje tak psát jednotkové, integrační i

[10] https://docs.pytest.org/

další druhy testů. Testy pro pytest (dále jen testy) jsou psány jako testovací metody nebo testovací třídy umístěné v souborech se specifickým názvem. Je důležité dodržet předepsaný formát jména souboru, ve kterém jsou napsané testy, protože pytest využívá prefixu a sufixu jména souboru pro vyhledávání testů. V základním nastavení a bez dodatečných parametrů pytest vyhledává v celém adresářovém stromu počínaje kořenovým adresářem, odkud je pytest spouštěn.

Nástroj pytest pracuje z velké části automatizovaně, což umožňuje rychlé psaní jednoduchých testů. Pro složitější a pokročilejší použití se využívají tzv. *hooks* a další pokročilé možnosti nástroje pytest. Více se lze dočíst v oficiální dokumentaci [14], z které jsem čerpal informace o tomto frameworku. Nástroj pytest nabízí snadné použití výrazů `assert`, které poskytují již v základním nastavení vestavěně podrobné informace o zachycené aserci a podporu pro nejčastěji používané výrazy jako např. *volání funkcí, přístupy do polí, aritmetické i objektové porovnaní, binární i unární operátory a další.*

Základním formátem výstupu je prostý text vypsaný do příkazové řádky. Existují moduly, které umožňují generovat *reporty* ve formátu XML nebo HTML. Podobně jako u frameworku JUnit 3.1 lze využít těchto souborů pro další zpracování výsledků. Na příkladě 3.4 lze vidět zdrojový kód testů napsaných ve frameworku pytest. V ukázce 3.5 pak výstup ve formátu XML a na obrázku 3.2 *report* ve formátu HTML.

```
import pytest


@pytest.fixture(params=["a", "b"])
def letters_fixture(request):
    yield request.param


@pytest.fixture(params=[1])
def numbers_fixture(request):
    yield request.param


def test_fixtureception(letters_fixture, numbers_fixture):
    coordinate: str = letters_fixture + str(numbers_fixture)
    print('\nRunning test_fixtureception with "{}"'.format(coordinate))
    assert coordinate.startswith('b')
```

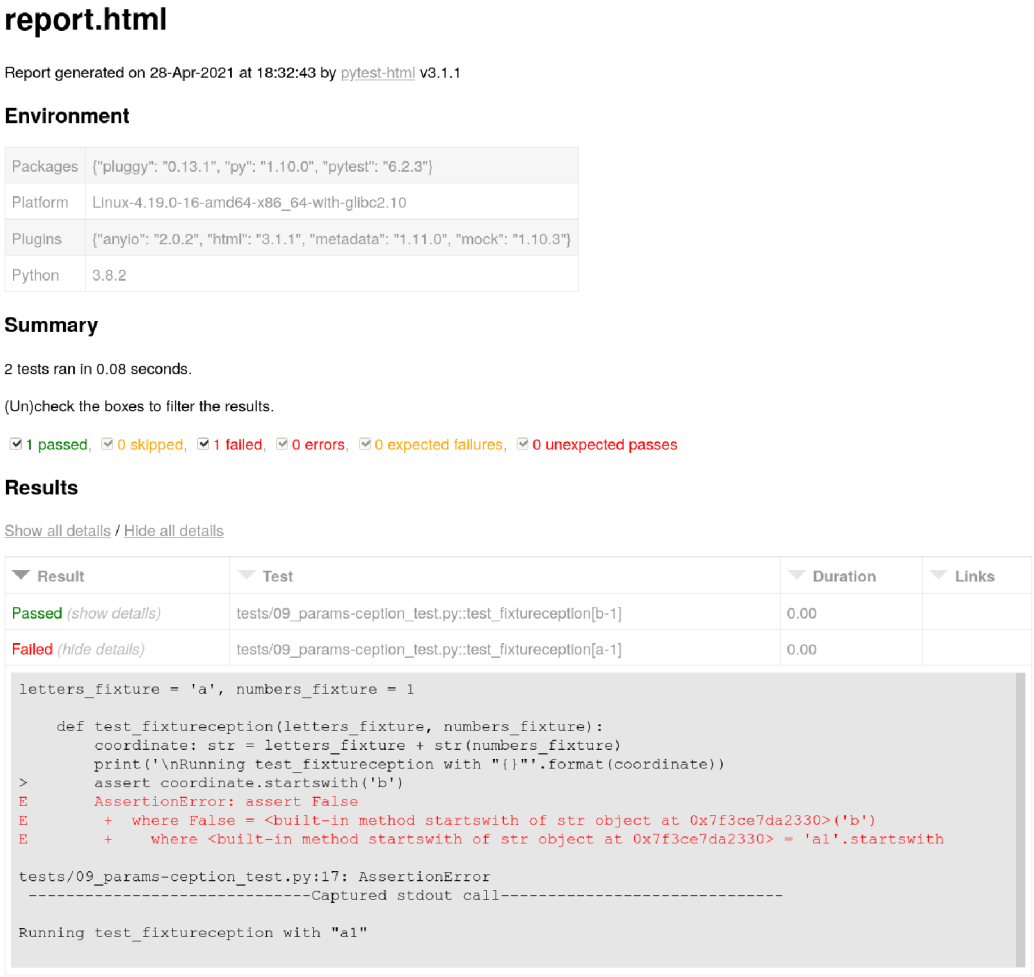
Kód 3.4: Příklad využití frameworku pytest k testování. Využití tzv. *fixtures* k pokročilé parametrizaci testů. Zjednodušeno z původní verze [23].

```
<?xml version="1.0" encoding="utf-8"?>
<testsuites>
    <testsuite name="pytest" errors="0" failures="1" skipped="0" tests="2" time="0.079"
               timestamp="2021-04-28T17:57:04.640611" hostname="pc322">
        <testcase classname="tests.09_params-ception_test" name="test_fixtureception[a-1]"
                  time="0.001">
            <failure message="AssertionError: assert False&#10; +  where False = &lt;
                              built-in method startswith of str object at 0x7fa7f7b243b0&gt;
                              (’b’)&#10; +    where &lt;built-in method startswith of str
                              object at 0x7fa7f7b243b0&gt; = ’a1’.startswith">
                              letters_fixture = ’a’, numbers_fixture = 1

    def test_fixtureception(letters_fixture, numbers_fixture):
        coordinate: str = letters_fixture + str(numbers_fixture)
        print(’\nRunning test_fixtureception with "{}"’.format(coordinate))
        &gt;       assert coordinate.startswith(’b’)
E       AssertionError: assert False
E       +  where False = &lt;built-in method startswith of str object at 0x7fa7f7b243b0&gt;
        (’b’)
E       +    where &lt;built-in method startswith of str object at 0x7fa7f7b243b0&gt; =
        ’a1’.startswith

tests/09_params-ception_test.py:17: AssertionError</failure>
        </testcase>
        <testcase classname="tests.09_params-ception_test" name="test_fixtureception[b-1]"
                  time="0.001" />
    </testsuite>
</testsuites>
```

Kód 3.5: Příklad XML *reportu* testovacích výsledků. V textu si lze všimnout základních informací o testovací, běhu a členění výsledků jednotlivých testů.



report.html

Report generated on 28-Apr-2021 at 18:32:43 by pytest-html v3.1.1

Environment

Packages	{"pluggy": "0.13.1", "py": "1.10.0", "pytest": "6.2.3"}
Platform	Linux-4.19.0-16-amd64-x86_64-with-glibc2.10
Plugins	{"anyio": "2.0.2", "html": "3.1.1", "metadata": "1.11.0", "mock": "1.10.3"}
Python	3.8.2

Summary

2 tests ran in 0.08 seconds.

(Un)check the boxes to filter the results.

1 passed, 0 skipped, 1 failed, 0 errors, 0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration	Links
Passed *(show details)*	tests/09_params-ception_test.py::test_fixtureception[b-1]	0.00	
Failed *(hide details)*	tests/09_params-ception_test.py::test_fixtureception[a-1]	0.00	

```
letters_fixture = 'a', numbers_fixture = 1

    def test_fixtureception(letters_fixture, numbers_fixture):
        coordinate: str = letters_fixture + str(numbers_fixture)
        print('\nRunning test_fixtureception with "{}"'.format(coordinate))
>       assert coordinate.startswith('b')
E       AssertionError: assert False
E        +  where False = <built-in method startswith of str object at 0x7f3ce7da2330>('b')
E        +    where <built-in method startswith of str object at 0x7f3ce7da2330> = 'a1'.startswith

tests/09_params-ception_test.py:17: AssertionError
 ------------------------------Captured stdout call------------------------------

Running test_fixtureception with "a1"
```

Obrázek 3.2: Ukázka HTML *reportu* jako výstupu z pytest testovacího běhu. Lze vidět základní informace jako datum a čas, proměnné prostředí, souhrnnou rekapitulaci testovacího běhu a jako poslední podrobné výsledky, ke kterým lze rozbalit podrobnější popis.

Kapitola 4

Ukládání a vizualizace dat

Účelem testování je zjistit, zda v programu nejsou chyby a jestli se chová dle očekávání. Výsledky časově kratších testů a malého rozsahu vyhodnotí vývojář zpravidla ihned po dokončení testování a zahájí následné kroky podle výsledku z testů. Pokud je testů velké množství, nemusí být vývojář schopen bez dalšího zpracování výsledky vyhodnotit. Tato kapitola pojednává zejména o různých typech databází a systémech řízení báze dat[1], což jsou systémy umožňující vytvoření, úpravy a správu databází.

4.1 Databáze

Databáze je datová struktura, která je organizovaná a slouží k modelování určité hierarchie dat nebo k modelování organizace dat. Ke sběru a uchování těchto dat dnes slouží téměř výhradně specializovaný počítačový program. Dle Hernandeze [10] lze použití databází rozdělit na dva typy podle jejich použití. Prvním typem použití je systém zaměřený na transakční zpracování dat neboli systém OLTP[2] a druhým typem je systém zaměřený na analýzu dat neboli systém OLAP[3].[4]

V paralele k OLTP a OLAP lze již samotné databáze korektně rozdělit na operační a analytické, kdy operační databáze jsou využívány při OLTP a analytické databáze jsou využívány při OLAP. Operační databáze slouží k uložení a modifikaci dat, ke kterým je denní přístup. Data musí být aktuální v každém okamžiku a také se často mění. Využití operačních databází a OLTP je například v obchodech, továrnách nebo třeba v knihkupectví. Nejčastějšími operacemi v OLTP databázích jsou `INSERT`, `UPDATE` a `DELETE`. Analytické databáze a OLAP slouží k uchování historických dat a k jejich analýze. Data se zpravidla příliš nemění a nejčastěji používanou operací je `SELECT`. Data v analytické databázi se využívají pro analytické zpracování, plánování dalších firemních postupů a pro různé prognózy. Uplatnění najde například v marketingu, chemických laboratořích nebo v geologických společnostech. Data z operačních databází většinou slouží jako hlavní zdroj dat pro analytické databáze, avšak návrh a struktura operačních a analytických databází jsou velmi odlišené. [10]

[1] SŘBD = sytém řízení báze dat, anglicky RMDBS = Relational Database Management System.

[2] on-line transaction processing

[3] on-line analytical processing

[4] OLTP a OLAP jsou přístupy k práci s daty a databázemi, nikoli přímo typy databází. Pojmy se ale využívají k definování zaměření databáze. Například pojmem „OLTP databáze“ se myslí databáze, která je primárně využívána pro transakční zpracování požadavků.

Databáze lze také dělit na *relační* (SQL) a *nerelační* (NoSQL), jak popisuje John Hammink [9]. Protože se relační databáze aktivně používají více jak 40 let, tak existuje více odborníků na tento druh databází, velké firmy udržují nejpoužívanější SŘBD a standard jazyka SQL (více o jazyku SQL v podsekci 4.2.2) je propracovaný a dobře udržovaný. Relační databáze jsou vhodné zejména pro strukturovaná data. Protože testovací systém ve firmě Codasip generuje právě strukturovaný typ dat, zaměřil jsem se v následující sekci na podrobnější popis relačních databází. NoSQL a další databáze jsem zevrubněji pospal v sekci 4.3.

4.2 Relační databáze

Dle Google Trends[5] a webových stránek DB-Engines[6] a Johna Hamminka [9] jsou relační databáze doposud nejrozšířenějším typem databází. Relační databáze byly poprvé popsány v roce 1969 a k dnešnímu dni se stále jedná o nejvíce zastoupenou formu databáze ve světě. Relační databáze jsou založeny na relačním modelu, jehož autorem je doktor Edgar F. Codd, který relační model představil oficiálně v roce 1970 ve své práci *A Relational Model of Data for Large Shared Databanks* [2]. Relační model vychází ze dvou matematických odvětví, a to teorie množin a predikátové logiky prvního řádu. Slovo „relační“ v názvu relačních databází odkazuje na matematický termín „relace“. Tabulka relační databáze představuje tuto matematickou relaci. Data jsou uložená v n-ticích – záznamech (v řádkově orientovaných databázích jsou to řádky) složených z atributů – hodnot (v řádkově orientovaných databázích jsou to hodnoty v jednotlivých sloupcích). V následující tabulce 4.1 můžete vidět ukázku jednoduchého uložení dat z testovacího běhu.

id	name	type	passed
1	test_addition	unit	1
2	test_division	unit	0
3	test_division	unit	1

Tabulka 4.1: Příklad tabulky v databázi s výsledky testů.

Více tabulek a vazby mezi tabulkami tvoří tzv. schéma databáze. Jedná se znázornění struktury a hierarchie dat v databázi, více se databázovému schématu věnuji v podsekci 4.2.1.

4.2.1 Schéma relační databáze

Databázové schéma a jeho tvorbu popisuje ve své knize Keith Gordon [6]. Databáze reflektuje data a stav určitého systému. Při návrhu takovéto databáze se nejprve tvoří tzv. ER[7] diagram, který modeluje, jaká data mají mezi sebou jaké vztahy. ERD se někdy nazývá jako schéma. Toto schéma je složeno ze dvou typů prvků: entitní množiny a vazebné množiny. Entitní množiny jsou prakticky tabulky a vazební množiny pak vazby mezi tabulkami. Vazební množiny jsou tří typů mocnosti: „*jedna* ku *jedné*“, „*jedna* ku n“ a „m ku n“. U neurčité mocnosti (m nebo n výše) lze definovat i minimální hodnotu. O jakou hodnotu se jedná, lze určit z notace mocnosti vazební množiny. Lze použít například číselnou notaci

[5] https://trends.google.com/trends/explore?date=today%205-y&q=sql,nosql,mongodb,mysql,elasticsearch
[6] https://db-engines.com/en/ranking
[7] Entity Relationship

a psát „0..n“ nebo „1..n“. Návrh databázového schématu je důležitý pro následnou efektivní a korektní práci s databází. Špatný návrh databáze může vést na její nepoužitelnost.

4.2.2 SQL

Jak již bylo zmíněno výše, jazyk SQL je dotazovací jazyk, který je využívaný relačními databázovými systémy. Dle knihy *Understanding the New SQL* [17] je SQL deklarativní programovací jazyk, kterým lze provádět dotazy nad danou databází. Pomocí SQL jazyka lze vytvářet, upravovat, mazat a jinak manipulovat s celými databázemi, schématy, tabulkami, sloupci nebo konkrétními záznamy a jejich atributy. Připravil jsem příklad pro demonstraci jazyka SQL. V kódu 4.1 lze vidět příkazy v jazyce SQL pro vytvoření databáze a k použití této databáze. Vytvoření tabulky a vložení záznamů se nachází v ukázce 4.2. Následné dotazování se nad tabulkou je provedeno v kódu 4.3. Jako poslední část příkladu figuruje úprava záznamu a ověření, že se změny provedly, což lze pozorovat v ukázce 4.4. Všechny dílčí kódy ukázky jazyka SQL jsou napsané v syntaxi jazyka MySQL. Více se lze o jazyce SQL dočíst na webových stránkách organizace ANSI [12], která spravuje SQL standard, další příklady SQL syntaxe pak v příspěvku organizace Free Code Camp [13].

```
-- Vytvoreni databaze.
CREATE DATABASE znamky;

-- Prikaz pro pouziti nove databaze.
USE znamky;
```

Kód 4.1: Vytvoření databáze a provedení příkazu pro použití této nově vytvořené databáze.

```
-- Vytvoreni tabulky pro predmet algoritmy.
CREATE TABLE algoritmy(
    id bigint auto_increment
        primary key,
    jmeno varchar(15),
    prijmeni varchar(24),
    zadano datetime null,
    znamka varchar(1)
);

-- Vlozeni zaznamu o~znamkach.
INSERT INTO algoritmy (jmeno, prijmeni, zadano, znamka)
VALUES
       ('Petr', 'Solny', '2021-03-28 12:45:57', 'B'),
       ('Antonin', 'Novotny', '2021-03-30 12:05:14', 'D');
```

Kód 4.2: Vytvoření tabulky pro zaznamenávání známek pro předmět Algoritmy a vložení záznamů do této tabulky.

```
SELECT CONCAT_WS(' ', jmeno, prijmeni) as student, znamka
FROM algoritmy
WHERE znamka >= 'C';
-- Vystup prikazu:
# +---------------+------+
# |student |znamka|
# +---------------+------+
# |Antonin Novotny|D |
# +---------------+------+
```

Kód 4.3: Dotaz nad tabulkou na studenty se známkou C a horší.

```
-- Uprava zaznamu, odhaleno plagiatorstvi na projektu u~studenta Solneho.
UPDATE algoritmy
SET zadano = NOW(), znamka = 'F'
WHERE jmeno = 'Petr' AND prijmeni = 'Solny';

-- Zmeny se provedly.
SELECT CONCAT_WS(' ', jmeno, prijmeni) as student, znamka
FROM algoritmy
WHERE znamka >= 'C';
-- Vystup prikazu:
# +---------------+------+
# |student |znamka|
# +---------------+------+
# |Petr Solny |F |
# |Antonin Novotny|D |
# +---------------+------+
```

Kód 4.4: Upravení hodnoty atributu `znamka` u vybraného studenta. Ověření, že se právě provedené změny aplikovali.

4.2.3 ACID

Problematiku transakčního zpracování požadavků a tzv. *ACID* pravidel definuje dokumentace Oracle [18]. ACID je zkratka *Atomicity*, *Consistency*, *Isolation* a *Durability*. Jednotlivá slova zkratky představují určitá pravidla, která definují práci daného SŘBD s transakcemi:

- Atomicita zaručuje, že buď se provedou všechny transakce nebo se neprovede žádná a již provedené změny budou vráceny nazpět do původního stavu.
- Konzistence zaručuje, že před provedením a po provedení transakce jsou data uložená v databázi v konzistentním stavu. Například po provedení převodu finančních prostředků mezi dvěma účty je suma částek na obou účtech stále stejná.
- Izolace je vlastnost, která zaručuje, že stav určité transakce v každý okamžik jejího provádění je neviditelný pro ostatní transakce. Například při převodu peněz ostatní transakce uvidí převáděnou částku vždy pouze na jednom anebo na druhém účtu, nikdy ne na obou zároveň nebo ani na jednom z nich.
- Trvalost znamená, že po dokončení transakce jsou změny trvalé, stálé i při vypnutí či výpadku systému.

Tato pravidla definují korespondující vlastnosti, které jsou typické pro relační databáze. Vlastnosti ACID jsou nutné pro korektní transakční zpracování dat typické pro OLTP typ databází nebo pro kritické aplikace například z oblasti bankovnictví. Implementace těchto vlastností je realizována v rámci daného databázového systému.

4.2.4 Partitioning

Ačkoli SQL standard nezavádí žádnou správu dat na fyzické úrovni, protože ANSI SQL definuje SQL jako jazyk, který je nezávislý na datových strukturách a typu úložiště. Databázové *enginy*[8] a SŘBD ale musí aspekty fyzického uložení, souborového systému a samotného hardware řešit. *Partitioning* umožňuje uživateli ovlivnit rozdělení dat na fyzické

[8] Bez českého ekvivalentu, dle svého principu lze přeložit jako „systém správy uložení dat“.

úrovni. Rozlišujeme dva druhy technologie *partitioning* – horizontální a vertikální. Horizontální *partitioning* umožňuje rozdělit data podle řádků. Rozdělení se provádí na základě předem definovaného kritéria, nejčastěji se jedná o rozdělení na základě hodnot určitého sloupce. Různé záznamy/řádky tedy budou fyzicky uloženy do různých *partitions*. Druhým typem technologie *partitioning* je vertikální *partitioning*, při kterém se data dělí podle sloupců. Různé sloupce jsou tedy fyzicky uloženy v různých *partitions*. U vertikálního verze technologie *partitioning* se pak jedná zpravidla o přesnou definici sloupců, zatímco u horizontálního verze technologie *partitioning* se jednalo o výraz určující specifické záznamy.

Každý SŘBD má (pokud vůbec) vlastní implementaci technologie *partitioning*. Podle serveru *DB Engines* podporují všechny tři nejpoužívanější SŘBD (Oracle, MySQL a MS SQL Server) horizontální i vertikální technologii *partitioning*. Ukázku MySQL technologie *partitioning* lze vidět v kódu 4.5

```
CREATE TABLE t1 (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    region_code TINYINT UNSIGNED NOT NULL,
    dob DATE NOT NULL
)
PARTITION BY RANGE( region_code ) (
    PARTITION p0 VALUES LESS THAN (64),
    PARTITION p1 VALUES LESS THAN (128),
    PARTITION p2 VALUES LESS THAN (192),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);

-- Tento dotaz vyuzije partitions a bude prohledavat pouze partition p1.
SELECT fname, lname, region_code, dob
    FROM t1
    WHERE region_code > 125 AND region_code < 130;
```

Kód 4.5: Vytvoření tabulky `t1` rozdělené do 4 *partitions* podle hodnoty v sloupci `region_code`. Následuje dotaz nad tabulkou `t1`, kdy databázový systém může efektivně využít *partition* p1, protože dle klauzule `WHERE` se hledají záznamy, kde atribut `region_code` leží mezi 125 a 130, čemuž odpovídá jen a pouze *partition* p1.

4.2.5 Příklady relačních databázových systémů

Podle žebříčku serveru *DB Engines*[9] mezi první tři nejpoužívanější databázové systémy patří Oracle, MySQL a MS SQL Server. Každý z databázových systémů (nejen tyto tři) zpravidla používá vlastní verzi SQL jazyka, která vychází ze standardu SQL. Modifikované jazyky SQL bývají podmnožinou ANSI SQL rozšířenou o další funkce. Oracle využívá PL/SQL[10], MySQL implementuje také svoji verzi a MS SQL Server využívá opět vlastní Transact–SQL[11] (TSQL). Všechny tři zmíněné relační SŘBD zaručují ACID zpracování transakcí, nabízejí podporu datového formátu XML nebo technologie *partitioning*.

[9] https://db-engines.com/en/ranking
[10] https://www.oracle.com/database/technologies/application-development-pl/sql.html
[11] https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver15

4.3 NoSQL a další databáze

Podle článku „What is NoSQL?“ [22] je technologie NoSQL popsána jako mladá technologie z přelomu tisíciletí. Název NoSQL znamená *non SQL* nebo *not only SQL*. Nehledě na interpretaci názvu, databáze NoSQL pracují na odlišném principu než tradiční relační databáze. Databáze NoSQL jsou schopny ukládat relační data, přestože název NoSQL může evokovat, že tomu tak není. Hlavní motivací pro vznik NoSQL databází byl zvyšující se objem dat a také fakt, že již se jedná pouze o částečně strukturovaná nebo zcela nestrukturovaná data, nikoli o striktně strukturovaná data, jak tomu bylo do té doby. Napomohlo tomu také to, že peněžní cena ukládání dat se výrazně snížila až na 0.01 dolaru za 1 MB v roce 2000[12] a bylo proto ekonomicky možné začít pracovat s velkými objemy nestrukturovaných dat. NoSQL databáze lze klasifikovat do čtyř hlavních typů.

4.3.1 Databáze typu klíč-hodnota

Struktura dat je jeden velký slovník dvojic klíč-hodnota. Všechna data jsou uložena v jediném souboru. Typy hodnot můžou být celé číslo, textový řetězec nebo nejčastěji JSON[13] struktura. Dotazování nad databází typu klíč-hodnota se provádí pomocí referencí na klíče. Příkladem databáze typu klíč–hodnota je databáze Redis.

```
> set mykey somevalue
OK
> get mykey
"somevalue"
```

Kód 4.6: Příklad vložení hodnoty do Redis databáze pod daným klíčem a následné získání této hodnoty referencí pomocí klíče. [21]

4.3.2 Databáze založené na dokumentech

Dokumentově orientované databáze ukládají data podobně jako databáze typu klíč-hodnota. Zpravidla se jedná o JSON strukturu. Každý z dokumentů představuje určitou množinou dvojic klíč-hodnota. Dokumentově orientované databáze umožňují pohodlné horizontální škálování. Stejně jako databáze klíč-hodnota, dokumentově orientované databáze zpravidla poskytují přímé ORM[14] pro většinu nejpopulárnějších programovacích jazyků, tedy k datům získaným z databáze lze přistupovat přímo jako k objektům. Díky metadatům jednotlivých dokumentů jsou dokumentové databáze schopny dalších optimalizací a v určitých případech tedy i efektivnějších operací oproti prostým databázím typu klíč-hodnota. [22]

4.3.3 Wide-column store

Wide-column[15] databáze jsou podobné tradičním relačním databázím. Data ukládají v tabulkách a sloupcích. Sloupce jsou ale u tohoto typu databází dynamické. To znamená, že dva řádky v jedné tabulce můžou mít různý počet sloupců (atributů) nebo různé datové typy sloupců. Díky tomu jsou *wide-column* databáze v porovnání s relačními více flexibilní. Běžným užitím tohoto typu databází je ukládání IoT (anglicky *Internet of Things*, česky

[12] v roce 2017 to je již pouze 0.00003 dolaru za 1 MB
[13] JavaScript Object Notation
[14] Object Realtion Mapping
[15] Nemá český překlad, doslovně přeložit lze jako „široký sloupec“.

Internet věcí). dat. Příkladem *wide-column store*/databáze je Apache Cassandra[16] nebo Apache HBase[17].

4.3.4 Grafové databáze

Grafové databáze jsou specifický druh databází vyvinutý pro speciální případy užití, kdy je potřeba zkoumat vztahy mezi určitými entitami. Schéma grafové databáze se skládá z uzlů a hran (vychází z matematické definice grafu). Uzly představují určité entity a hrany pak vlastnosti vztahu těchto entit. Typickým případem užití jsou sociální sítě, algoritmy pro doporučování obsahu nebo detekce podezřelého (podvodného) chování – trasování peněžních toků apod. Příkladem grafové databáze je Neo4j[18] nebo JanusGraph[19]. [22]

4.3.5 NewSQL databáze

NewSQL databáze jsou relační databáze, které zároveň poskytují efektivní horizontální škálování jako NoSQL databáze a zároveň garantují vlastnosti ACID (sekce 4.2.3). Mezi zástupce NewSQL databází patří například CockroachDB[20], který nabízí například *partitioning* podle geografické lokace a ochranu před ztrátou dat na vysoké úrovni nebo NuoDB[21], což je distribuovaná SQL databáze zaměřená na ochranu dat podobně jako CockroachDB.

4.4 Vizualizace dat

Kniha *Hands-On Data Visualization* [3] popisuje, jakým způsobem vytvořit správnou vizualizaci. Pomocí kvalitně vypracované vizualizace dat totiž lze uživateli předat více informací, než by bylo možné pouhým textem. Slovy lze popsat mnoho, ale vizuální podání informace poskytuje další možnosti, jak interpretovat data. Správně vizualizovaná data přitahují čtenářovu pozornost k důležitým údajům nebo aspektům dat. Analýza cílových dat tak může být efektivnější a rychlejší. Tvorba dobré vizualizace ale není jednoduchý úkol. Jak píše Jack Dougherty a Ilya Ilyankou [3], nesprávně vytvořená vizualizace může být dokonce i zavádějící.

Při vizualizování dat je důležité dobré porozumění zkoumané domény i samotným datům, která jsou analyzována. Pokud se například v množině zpracovávaných dat nachází extrémy, které se zcela vymykají běžným datům (anglicky *outliers*, tedy „data, která leží mimo“), je dobré tato data z naší vizualizace úplně odstranit nebo použít takové metody, které jsou odolné proti vlivu extrémů. Jako příklad problematiky extrémních dat uvádí autoři v [3] průměr vs. medián čísel 1, 2, 3, 4 a 100. Průměr je 22, kdežto medián je 3. Jak je vidět u tohoto jednoduchého příkladu, je důležité znát charakter dat, se kterými pracujeme.

Kniha zmiňuje mimo jiné i pravidla, která by dobrá vizualizace měla splňovat a rady, čemu se při vytváření vizualizace vyhnout a na co si dát pozor. Další detailní informace ohledně problematiky vizualizace dat nejsou pro potřeby této práce nutné. Uvedu proto dále jen dva příklady vizualizačních nástrojů: Tableau jako zástupce čistě komerčního řešení a Metabase jako zástupce open–source řešení[22].

[16] https://cassandra.apache.org/
[17] https://hbase.apache.org/
[18] https://neo4j.com/
[19] https://janusgraph.org/
[20] https://www.cockroachlabs.com/
[21] https://nuodb.com/
[22] Metabase nabízí mimo open–source verzi svého software i mimo jiné zpoplatněné *enterprise* řešení.

4.4.1 Tableau

Tableau[23] je nástroj pro komplexní správu, analýzu a vizualizaci dat. Tableau je dostupné jako počítačová aplikace, v prohlížeči, jako aplikace pro mobilní zařízeni a i jako vestavěná aplikace. Kromě uživateli vytvořených analýz a pohledů umožňuje Tableau využít i analýzu a interpretaci dat autonomně pomocí umělé inteligence. Pro provoz Tableau aplikací lze využít cloudových služeb Tableu, vlastního cloudového řešení anebo čistě lokálního hostingu. Tableau definuje ve své platformě tři typy uživatelů:

- *Creators* – Tvůrci obsahu, spravují datové zdroje, sestavují analytické pohledy, navrhují vizualizace a nástěnky (dashboardy).
- *Explorers* – Disponují omezeným přístupem k datům. K dispozici mají data, která jsou předzpracována uživateli *Creators*, nad těmito daty můžou provádět analýzy, vizualizace apod.
- *Viewers* – Tato úroveň uživatelů může vyhledávat a zobrazovat dostupné vizualizace a nástěnky. Můžou se také přihlásit k odběru novinek u konkrétních vizualizací.

Kromě analýzy a vizualizace dat nabízí Tableau i dvojici aplikací pro přípravu a správu datových zdrojů s názvem *Tableau Prep Builder* a *Tableau Prep Conductor*. Tableau tedy nabízí řešení s širokým zaměřením. Jedná se ale o čistě placený nástroj[24].

4.4.2 Metabase

Metabase[25] je nástroj běžící v prostředí prohlížeče určený zejména k vizualizaci dat. Nabízí i analytické možnosti nebo možnosti správy zdrojových dat. Metabase podobně jako Tableau nabízí automatickou analýzu a vizualizaci dat pojmenovanou jako *X-Rays*. Zde se však jedná o méně komplexní zpracování dat a Metabase tak poskytuje vhled spíše na charakter dat jako počet unikátních hodnot v tabulce apod. než komplexní analýzu dat. Metabase lze provozovat buď ve vlastní režii – na vlastních serverech/zařízeních nebo na libovolném cloudovém řešení, anebo s využitím hostingu od firmy Metabase. Metabase nabízí i open–source verzi aplikace, která je určená pouze pro vlastní hosting.

[23] https://www.tableau.com/
[24] Podrobnosti o zpoplatnění pro organizace lze nalézt zde: https://www.tableau.com/pricing/teams-orgs
[25] https://www.metabase.com/

Kapitola 5

Analýza problému

Tato práce řeší problematiku pomalých nebo nedostatečně rychlých databázových dotazů nad testovacími výsledky. V této kapitole řeším celkový systém testování, fyzické charakteristiky, které můžou mít vliv na rychlost zpracování výsledků a celkový průběh procesu od samotného spuštění testů až po konečnou vizualizaci výsledků. Zaměřuji se na dosavadní datový model, charakteristiku uložených dat a v neposlední řadě na formát a vlastnosti výstupů z testování, protože tyto výstupy jsou vstupem pro další databázové zpracování. Číselné a jiné údaje uváděné v této kapitole se vztahují k období psaní této práce.

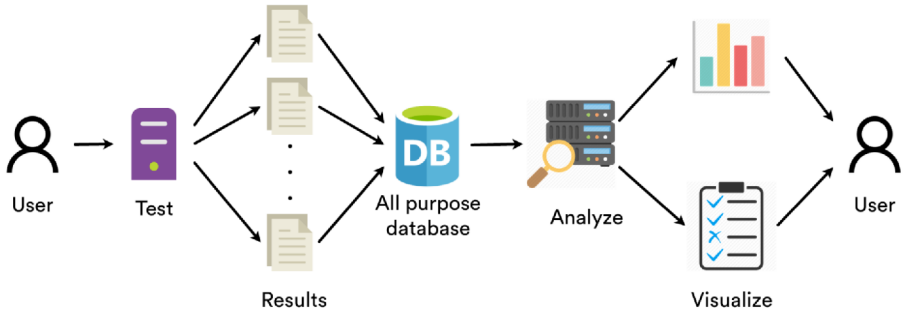
5.1 Analýza současného systému

Celkový systém je znázorněn na obrázku 5.1. Jako první bod testovacího systému stojí samotné testování. Toto testování je řízeno automatizovaně pomocí interního testovacího frameworku a nástroje Jenkins[1]. Každý den je provedeno 2,5 až 3,5 milionu testů. Testovací framework obstarává i plnění databáze základními daty. Celkový počet záznamů zapsaných do databáze je pak o něco větší. Testovací data jsou zpracovávána jednou instancí MySQL. V rámci databázového systému se základní data dále zpracovávají do různých pohledů (anglicky *views*), které jsou často agregované nebo filtrované přes různé atributy. Z důvodu dlouhého trvání provádění některých pohledů jsou pomalé pohledy uloženy v podobě materializovaných pohledů (anglicky *materialized views*). Vizualizace dat z testování pro koncové uživatele[2] je prováděna pomocí nástroje Metabase[3]. Struktura vizualizací v nástroji Metabase se skládá z vizualizačních pohledů zvaných *Questions*. Tyto vizualizace ve formě tabulek, grafů apod. jsou komponovány do nástěnek (anglicky *dashboards*). Z nástěnek i vizualizací jsou pak vytvořeny kolekce, v Metabase zvané *Collections*. Koncoví uživatelé pak využívají vytvořených nástěnek s vizualizacemi k monitorování průběhu vývoje produktů firmy Codasip.

[1] https://www.jenkins.io/

[2] Koncoví uživatelé systému jsou zejména týmoví vedoucí, kteří sledují, zda vývoj produktů probíhá dle očekávání.

[3] https://www.metabase.com/



Obrázek 5.1: Diagram celkového systému testování ve firmě Codasip od samotných testů až po vizualizaci.

V následujících sekcích proberu jednotlivé části systému podrobněji. Sekce 5.2 je věnována shrnutí analýzy a identifikaci problému, tedy příčiny, proč jsou dotazy nedostatečně rychlé. Řešení problému je pak navrženo v kapitole 6.

5.1.1 Testovací framework – generování dat pro další zpracování

Testovací framework je napsán v jazyce Python s intenzivním využitím testovacího frameworku pytest. V rámci jedné testovací relace pouští jeden běh nástroje pytest další sadu podřazených relací, jedná se zde o určité zanoření běhů tohoto nástroje. Framework je spouštěn pravidelně každý den včetně víkendů a běží zpravidla přes noc. Cílem je otestovat denní přírůstky vývoje případně větší změny v produktech. Přes noc proběhne testování a následující den by měli mít vývojáři zpřístupněny výsledky. Některé druhy testů z důvodu jejich náročnosti ale často končí až v průběhu následujícího dne, například během dopoledních hodin. Je žádané, aby i přesto měli vývojáři výsledky dostupné jakmile to bude možné. Framework pytest generuje *reporty* ve formátu HTML a XML. XML *reporty* podřazených testovacích relací jsou slučovány a vkládány do XML *reportu* nadřazené relace. HTML *reporty* se nijak neupravují. Hlavním výstupem každého testu jsou ale strukturovaná data o právě proběhnutém testu, která se vkládají do databáze.

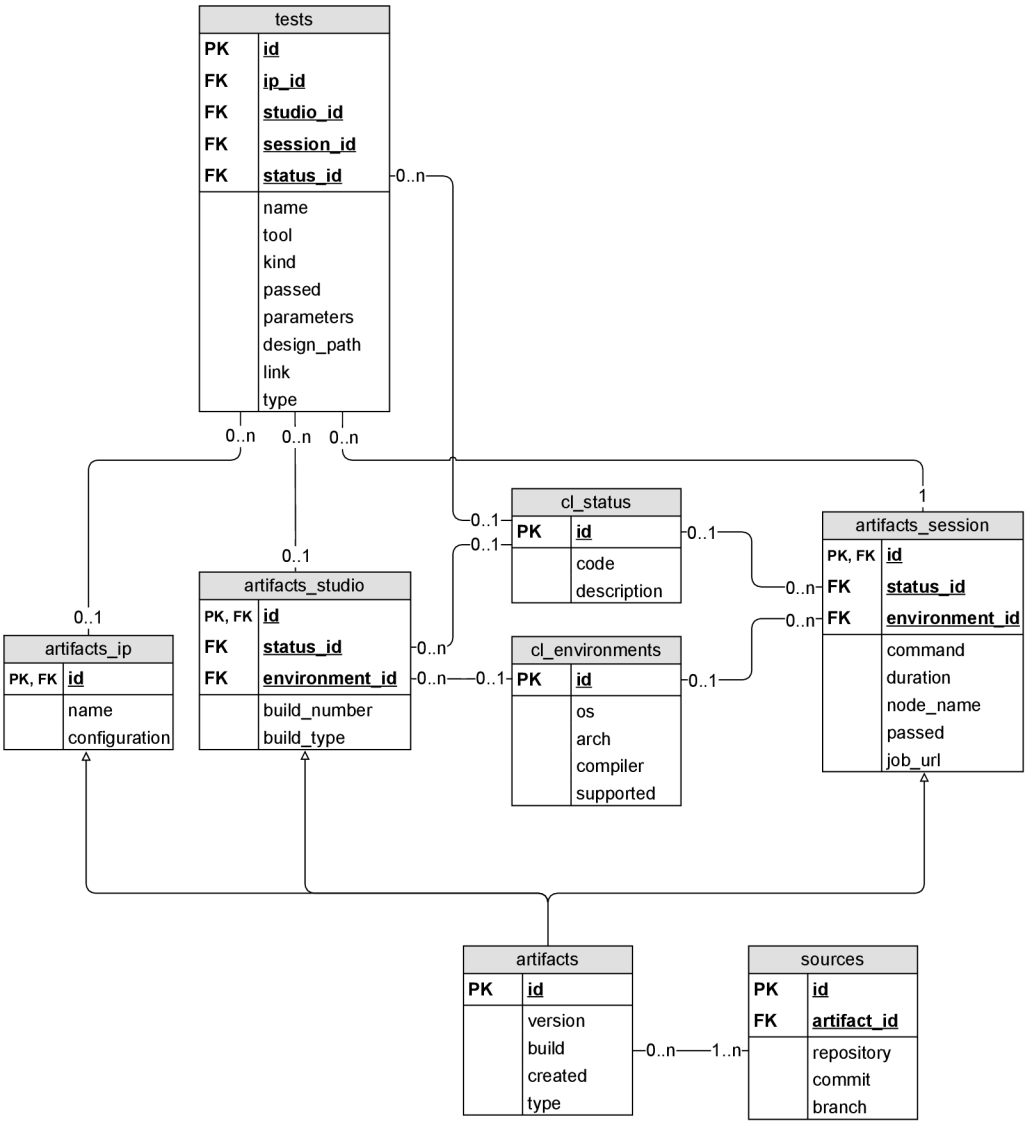
5.1.2 Datový model

Referenční ER diagram databáze pro ukládání testovacích výsledků ve firmě Codasip je možné vidět na obrázku 5.3. Databáze je plněna strukturovanými daty z testovacího frameworku. Databázový *engine* je InnoDB. Dle mojí analýzy vykazují data určité charakteristiky. Většinu objemu dat představuje tabulka `tests`, konkrétně 98 %. Tabulku `tests` pak tvoří z více jak 99 % záznamy o regresních testech překladače. Tabulka `sources` uchovává informace o git[4] repositářích, větvích (anglicky *branches*) a revizích, které jsou spjaty s určitým testovacím během. Kardinalita vazby na tabulku `artifacts` je *n*-*m*. Vazba je im-

[4] https://git-scm.com/

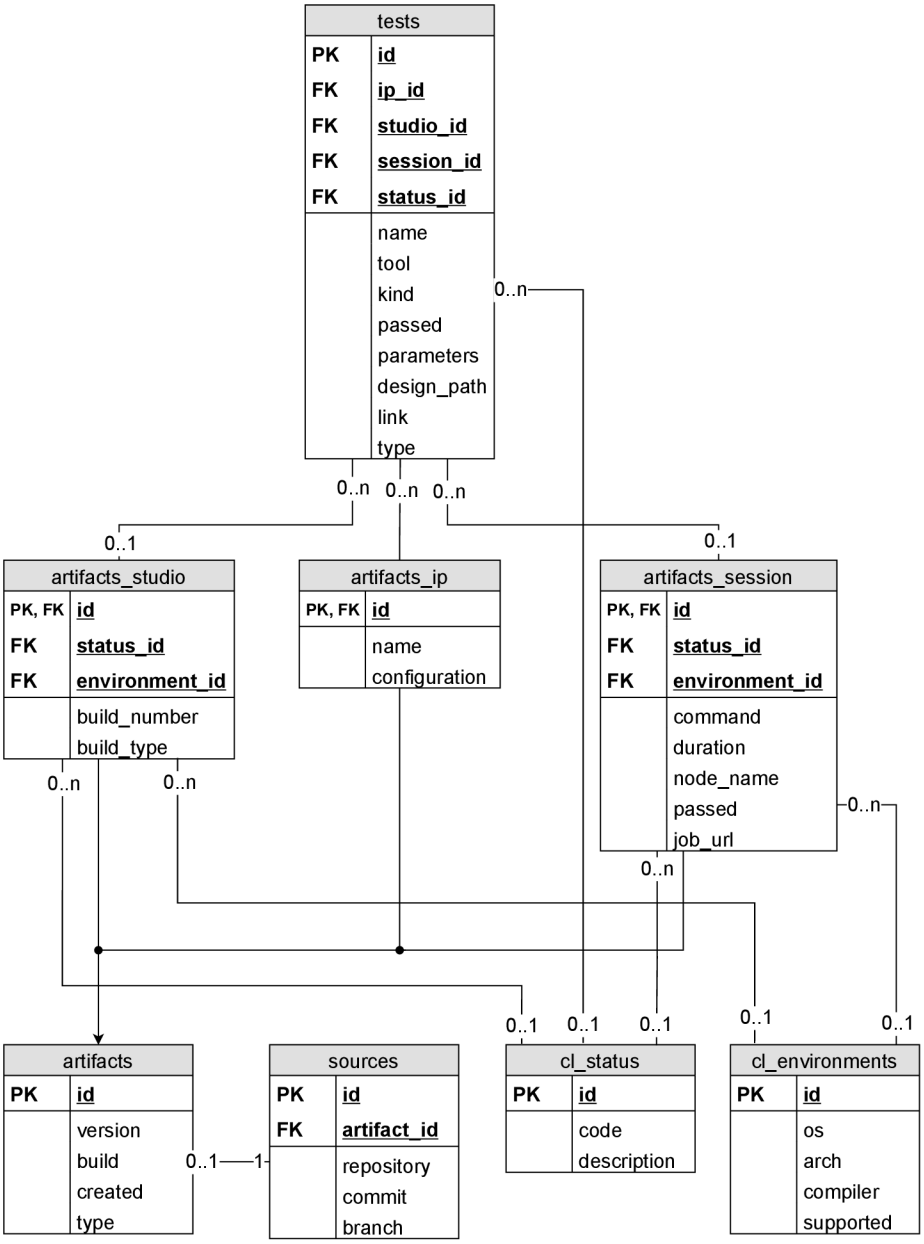
plementována pomocí cizího klíče v tabulce `sources` odkazujícího do tabulky `artifacts`. Dochází zde proto k částečné redundanci dat.

Během své analýzy jsem identifikoval, že vazba `artifacts-sources` není *0..1-1*, ale je *m-n*. Jednou ze specializací entitní množiny `artifacts` je totiž `artifacts_studio`. Atribut `artifacts_studio` reprezentuje sestavení produktu Codasip Studio™. Codasip Studio™ je sestavováno z mnoha repositářů. Proto je kardinalita vazby u `sources` rovna *m*. Dále pak jedny a ty samé „zdroje“ (repositáře) můžou být zdrojem pro více testovacích běhů. Například zmiňované Codasip Studio™ je složeno mimo jiné i z repositáře LLVM[5] překladače. Konkrétní větev a revize LLVM repositáře může být ale použita ve více různých verzích Codasip Studia™. To znamená, že kardinalita vazby u `artifacts` je rovna *n*. Výsledkem je tedy vazební množina `artifacts-sources` s kardinalitou *m-n*. Reálné schéma lze vidět na obrázku 5.2.



Obrázek 5.2: Diagram databázového schématu, které je použito pro ukládání a zpracování dat z testování se správnými (reálnými) kardinalitami.

[5] https://llvm.org/



Obrázek 5.3: Diagram databázového schématu, které je použito pro ukládání a zpracování dat z testování. Tento diagram byl poskytnut firmou Codasip jako referenční.

5.1.3 Zpracování dat

Mezi perzistentní data uložená v databázi patří i pohledy a materializované pohledy. Aktualizaci materializovaných pohledů obstarávají uložené procedury (anglicky *stored procedures*), které jsou pravidelně spouštěny plánovanými událostmi. Obnova všech materializovaných pohledů trvá několik hodin. Z tohoto důvodu jsou data v těchto materializovaných pohledech aktuální pouze v určité časové úseky v průběhu dne. Dle mého průzkumu je přístup firmy takový, že se pohledy, které již svojí rychlostí provádění nevyhovují, převádí na materializované pohledy. Nicméně těchto materializovaných pohledů stále přibývá a je tedy nutné vyřešit příčinu, proč je aktualizace pohledů potažmo provádění dotazů pomalé.

Většina sloupců ve všech tabulkách je indexována indexem typu B–strom [19]. Tento index je vhodný pro takové atributy, které nabývají co nejvíce různých hodnot. Například primární klíče jsou v databázovém systému MySQL automaticky indexovány právě tímto typem indexu. Indexem typu B–strom jsou ale indexovány i atributy jako `tool` nebo `kind`, které mají 13, respektive 27 unikátních hodnot. Dle internetových stránek společnosti SAP[6] jsou pro atributy, které mají pouze nízký poměr unikátních hodnot vhodné bitmapové indexy.

Dále jsem zjistil, že některé dotazy jsou napsány nevhodně vůči specifikaci daného dotazu/pohledu. Jednalo se zejména o časté použití `LEFT JOIN`, kdy pohled zobrazoval i záznamy, které byly pro svůj obsah zcela irelevantní pro daný pohled. Nekompletní a jinak poškozené záznamy určitě do databáze patří. Jedná se o důležitou informaci, že pro konkrétní test nastala nějaká neočekávaná situace a chyba buď při samotném testování nebo v testovacím frameworku. Pro analýzu testovacích výsledků, ale nemají takovéto záznamy žádný význam. U dotazů s využitím `LEFT JOIN` se pak následně v klauzuli `WHERE` často filtrovali nalezené záznamy tak, že výsledek odpovídal využití `INNER JOIN`.

Časté použití *wildcard* v klauzulích `WHERE` na začátku i konci vyhledávaného řetězce (např. `'%codasip_urisc%'`) může značně zpomalovat filtrování podle tohoto atributu. Také si lze všimnout přebytečných spojení tabulek – pravděpodobně vlivem tvorby nových pohledů kopírováním pohledů stávajících a jejich následnou úpravou. Zbytečné spojování tabulek nepříznivě ovlivňuje jakýkoli typ dotazu.

5.1.4 Vizualizace dat

Pro účely vizualizace je použitý nástroj Metabase v open–source verzi. Tato verze, ačkoli je zdarma, nabízí poměrně široké možnosti. Například SQL editor, kde je možné psát SQL dotazy přímo v rozhraní Metabase. Této možnosti Codasip intenzivně využívá a jedná se tak o další vrstvu zpracování dat. SQL editor je použit pro tvorbu filtrů, které jsou zobrazeny v grafickém rozhraní. Tyto filtry pak slouží uživatelům pro dodatečné filtrování nebo řazení. Jak vypadá prostředí Metabase s uživatelskými filtry je vidět na obrázku 5.4.

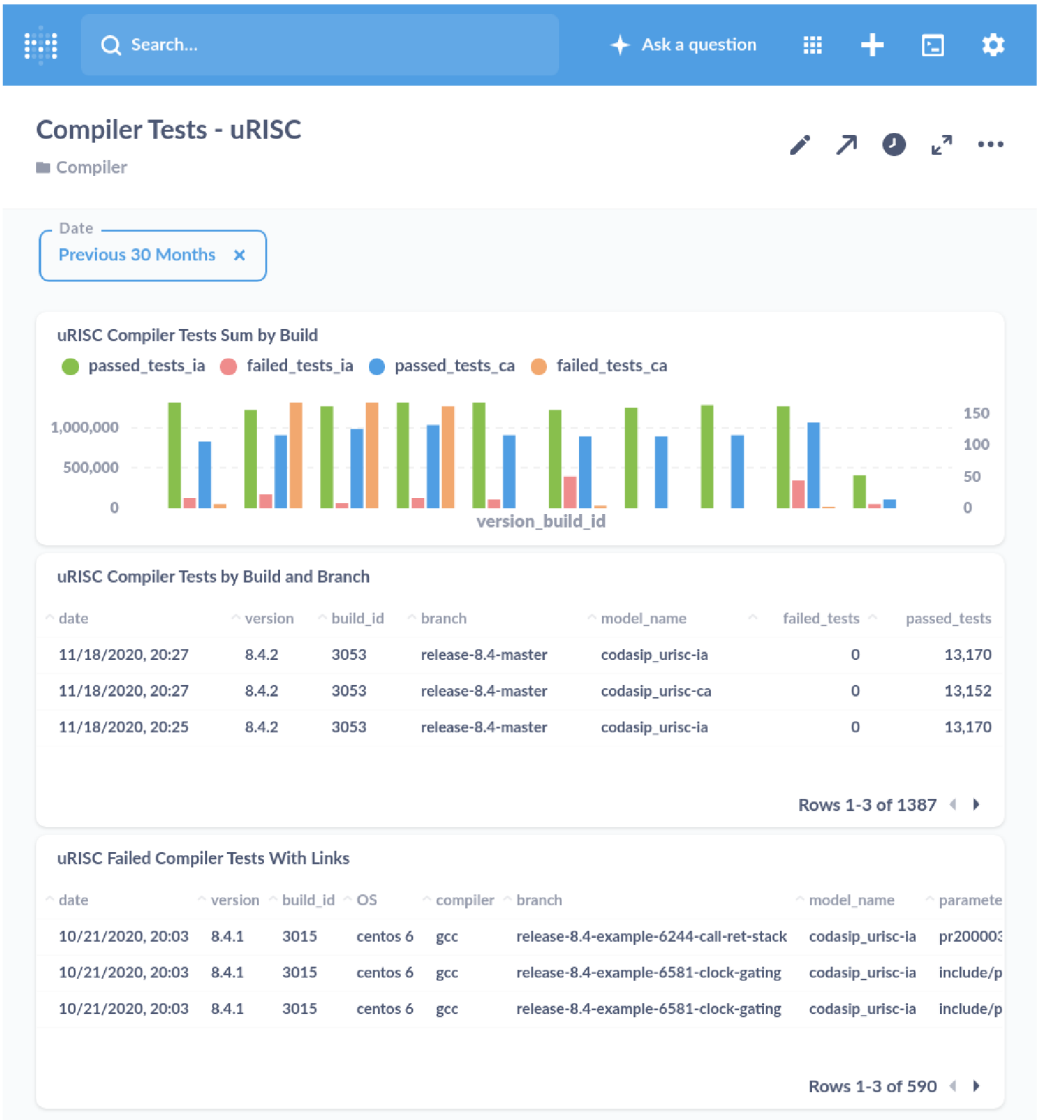
5.1.5 Parametry hardware a ostatní charakteristiky systému

Kromě výše uvedených oblastí jsem se při analýze současné situace ve firmě Codasip zaměřil i na vlastnosti hardware strojů, na kterých běží provoz databáze a dalších služeb. Motivace k analýze těchto údajů je fakt, že nízká kapacita operační paměti, pomalé vstupně–výstupní operace na discích nebo malá propustnost síťového rozhraní můžou být faktory ovlivňující výkonnost celkového systému, zejména pak v případě využití systému více uživateli současně.

Nejvíce mě zajímalo, na jaké počítačové konfiguraci se nachází databáze, případně Metabase aplikace. Dle provedené analýzy se MySQL i Metabase aplikace nachází na stejném stroji. Jedná se ale o virtuální stroj umístěný na jednom z firemních infrastrukturních serverů. Důležitá je tedy nejen konfigurace virtuálního stroje, ale i fyzického stroje, jehož hardware je virtualizovaný a sdílený.

Fyzický stroj je založen na HPE[7] serveru *ProLiant DL385 Gen10*. Server disponuje 256 GiB operační paměti s frekvencí 2666 MHz, dvěma šestnácti–jádrovými procesory AMD EPYC 7281 taktovanými na 2400 MHz s vyrovnávací pamětí L2 o velikosti 8 MiB a paměti

[6] https://wiki.scn.sap.com/wiki/display/EIM/Bitmap+Indexes
[7] https://www.hpe.com/us/en/home.html



Obrázek 5.4: Dashboard v prostředí nástroje Metabase, který vizualizuje analyzované výsledky regresních testů překladače na procesorovém jádře uRISC.

na úrovni L3 o velikosti 32 MiB. Tento server je připojen k NFS[8] SSD poli linkou o přenosové rychlosti 10 Gbit/s. SSD pole je založeno na komplexním řešení firmy NetApp[9], jedná se o vysoce výkonné řešení složené z 960 GB SSD disků s rozhraním SAS 12 GB/s. Virtuálnímu stroji, na kterém běží aplikace MySQL a Metabase je vyhrazeno osm procesorových jader v 100% výkonu a 14 GB operační paměti.

5.2 Identifikace problému

Zanalyzoval jsem různé faktory, které mají vliv na rychlost zpracování databázových dotazů. Nyní lze určit, co je příčinou pomalých dotazů. Parametry fyzického serveru jsou více než

[8] https://docs.microsoft.com/en-us/windows-server/storage/nfs/nfs-overview
[9] https://www.netapp.com/

dostačující. Virtuální stroj ovšem disponuje pouze 14 GB operační paměti. Toto je plně dostačující pro transakční zpracování testovacích výsledků – tedy pro vkládání nových dat. Pro analytické zpracování dat již tato kapacita dostačující být nemusí. Například tabulku `tests`, která má 25 GB velikost dat a 45 GB velikost indexu, nelze nahrát do 14 GB paměti a je nutné stránkování dat, což přirozeně prodlužuje dobu provádění dotazu.

Transakční funkce databáze pracuje dle očekávání a v této oblasti nebyly zaznamenány žádné problémy. Hlavní problém je v následném zpracování dat. Tabulka `tests` je obrovská a je tvořena z naprosté většiny záznamy o regresních testech překladače. Pohledy jsou diferencovány podle typu procesorových jader (atribut `tests.design_path`) a podle zaměření testů (atributy `tests.tool` a `tests.kind`). V každém z dotazů, který analyzuje regresní testy překladače musí počítač filtrovat podle atributu `tool` a atributu `kind` tak, aby vyfiltroval výsledky dle podmínky `(tests.tool = 'compiler') AND (tests.kind = 'regression')`. Podobně tomu je pak i u atributu `design_path`, který určuje o jaký typ procesorového jádra se jedná. Dotazy, které naopak analyzují něco jiného, než jsou regresní testy překladače, musí prohledat zbytečně o několik řádů více záznamů, než je potřeba. Sloupce jsou sice indexovány, ale indexy typu B–strom, kterými jsou sloupce indexovány, nejsou pro tyto účely vhodné.

Co se týká vizualizace, tak zde lze očekávat určité zpomalení, protože se jedná o další vrstvu zpracování dat společně s grafickým zobrazením. Agregace, filtrování a následné vykreslování grafů, tabulek a dalších prvků grafického rozhraní může způsobit sníženou odezvu maximálně několik málo jednotek sekund. Výrazné zpomalení na úrovni vizualizace by znamenalo, že je nutné provést optimalizaci na úrovni databáze.

5.3 Současné přístupy

Práce řeší velmi komplexní problém složený z několika částí. Generování testovacích záznamů, jejich následné zpracování a konečnou vizualizaci uživatelům. Codasip využívá pro tyto účely sadu nástrojů, které společně tvoří určitý celek, ten je podrobněji popsán již výše v sekci 5.1. Ve zdrojích, které jsem prohledal, jsem nebyl schopen nalézt vyhovující řešení celkového problému, který by se podobal problému ve firmě Codasip. Pokusil jsem se tedy rozložit celý systém na dílčí části. Pro dílčí části jsem ovšem opět v hledání neuspěl. Autoři článků nebo blogů zpravidla řeší velmi konkrétní problémy nebo diskutují problém na příliš jednoduché úrovni. Například ve firmě `Qxf2 Services` je zpracování testovacích výsledků z nástroje pytest (sekce 3.2) realizováno pomocí HTML *reportů*. Interakce s uživateli, kterých se testy týkají, je provedena pomocí zasílání HTML *reportů* cílovým uživatelům elektronickou poštou. [5] Toto je ovšem pro potřeby firmy Codasip nedostatečná úroveň zpracování výsledků.

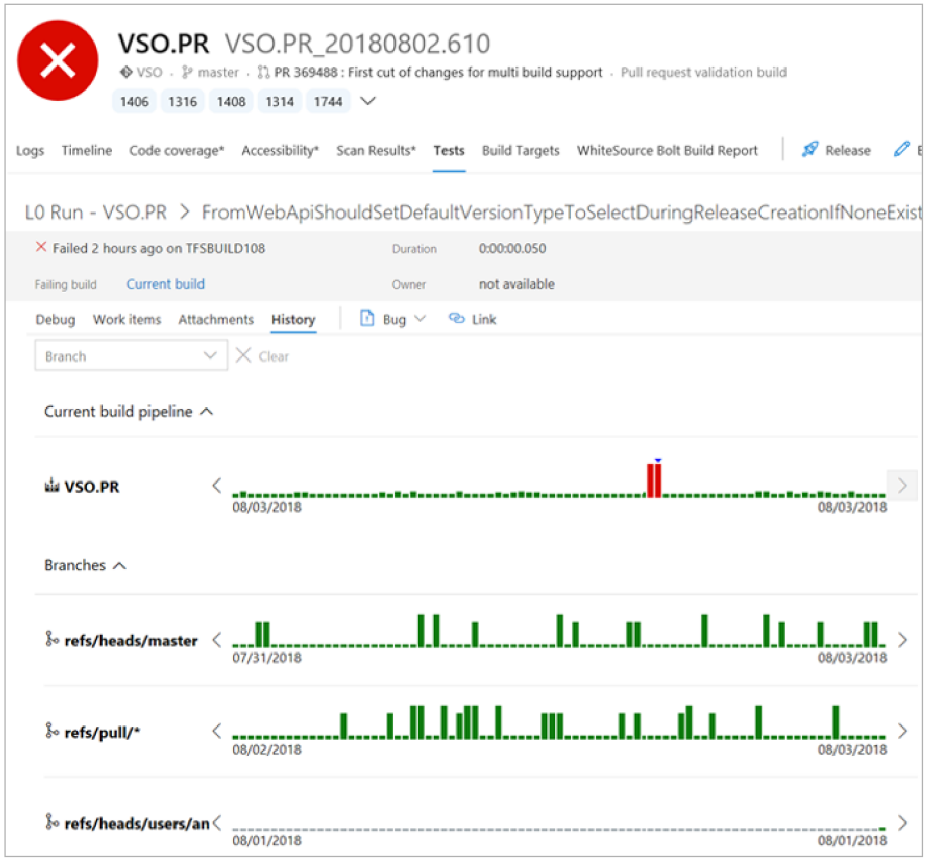
Velké úsilí jsem věnoval analýze sady nástrojů *Microsoft Azure DevOps*[10] (dále jen Azure). Azure tvoří komplexní ekosystém vlastních nástrojů a řešení, které uživatelům poskytují široké možnosti pro automatizaci vývojových procesů, zajištění výpočetního výkonu apod. Aby bylo ale využití Azure výhodné, bylo by potřeba implementovat co nejvíce částí systému v prostředí Azure. V případě firmy Codasip by do prostředí Azure bylo možné přesunout automatizované testování a další procesy, které jsou nyní spravovány v nástroji Jenkins. Azure dále nabízí podporu zpracování testovacích výsledků z několika frameworků různých programovacích jazyků. Pro Python to je pouze modul *unittest*, a to ne zcela

[10] https://docs.microsoft.com/en-us/azure/devops

úplně. Microsoft ve své dokumentaci k Azure uvádí příklad sestavení a testování Python aplikace. [15]

Vizualizace v Azure těchto testovacích výsledků umožňuje uživateli zobrazit například srovnání různých vývojových větví v závislosti na počtu úspěšných a neúspěšných testů – jak je vidět na obrázku 5.5 z dokumentace k nástroji Azure[11]. Azure je schopen automaticky zpracovávat data definovaná základním výstupem z daného nástroje. Uživatelsky definované výstupy zpracovávat neumožňuje. Azure nativně nenabízí možnosti uživatelem definovaného zpracování dat, jako by bylo v případě firmy Codasip například zpracování dat na základě typu nebo konfigurace jádra. Z tohoto důvodu by bylo možné využít Azure DevOps s výhodou na automatické testování a sestavování a nasazení produktů firmy Codasip. Dále by Azure mohl poskytnout vhled do vývojového cyklu na vyšší úrovni abstrakce, např. pro technicky zaměřený vyšší management. Na pokročilé zpracování dat, jejich analýzu a vizualizaci ale Azure není, dle mého názoru, vhodně koncipován.

Pro potřeby pokročilého zpracování a analýzy dat je zapotřebí jiné řešení. Přechod na Azure technologie by navíc představoval finanční zátěž, protože služby Azure jsou zpoplatněny[12]. Protože se mi vyhovující řešení nepodařilo najít, bylo nutné, abych vhodné řešení navrhl sám. Toto řešení je popsáno v kapitole 6.



Obrázek 5.5: Srovnání úspěšnosti testování na více vývojových větvích v prostředí Microsoft Azure.

[11] https://docs.microsoft.com/en-us/azure/devops/pipelines/test/media/review-continuous-test-results-after-build/view-historical-trend.png?view=azure-devops

[12] https://azure.microsoft.com/en-us/pricing/

Kapitola 6

Návrh řešení

Na základě analýzy v kapitole 5 v této kapitole předkládám návrh řešení. Na úvod bych rád zmínil požadavky na řešení, které jsem získal na základě konzultací s vývojáři firmy Codasip. Jde o další požadavky mimo řešení samotného problému pomalých dotazů, což se implicitně od mého řešení očekává. Jedná se o následující:

- Řešení by mělo být založeno na nástrojích, jejichž použití je zdarma i pro větší počet uživatelů. Nemusí se jednat přímo o open–source.
- Řešení by mělo využívat takové technologie, které lze hostovat na vlastních zařízeních uvnitř firemní infrastruktury – kvůli správě aplikací, integraci a ochraně dat.
- Integrace řešení by měla být co nejsnadnější, aby bylo možné řešení ihned začít používat.
- Jsou preferované takové technologie, jejichž použití nevyžaduje vývojáře se speciální expertízou na danou technologii.

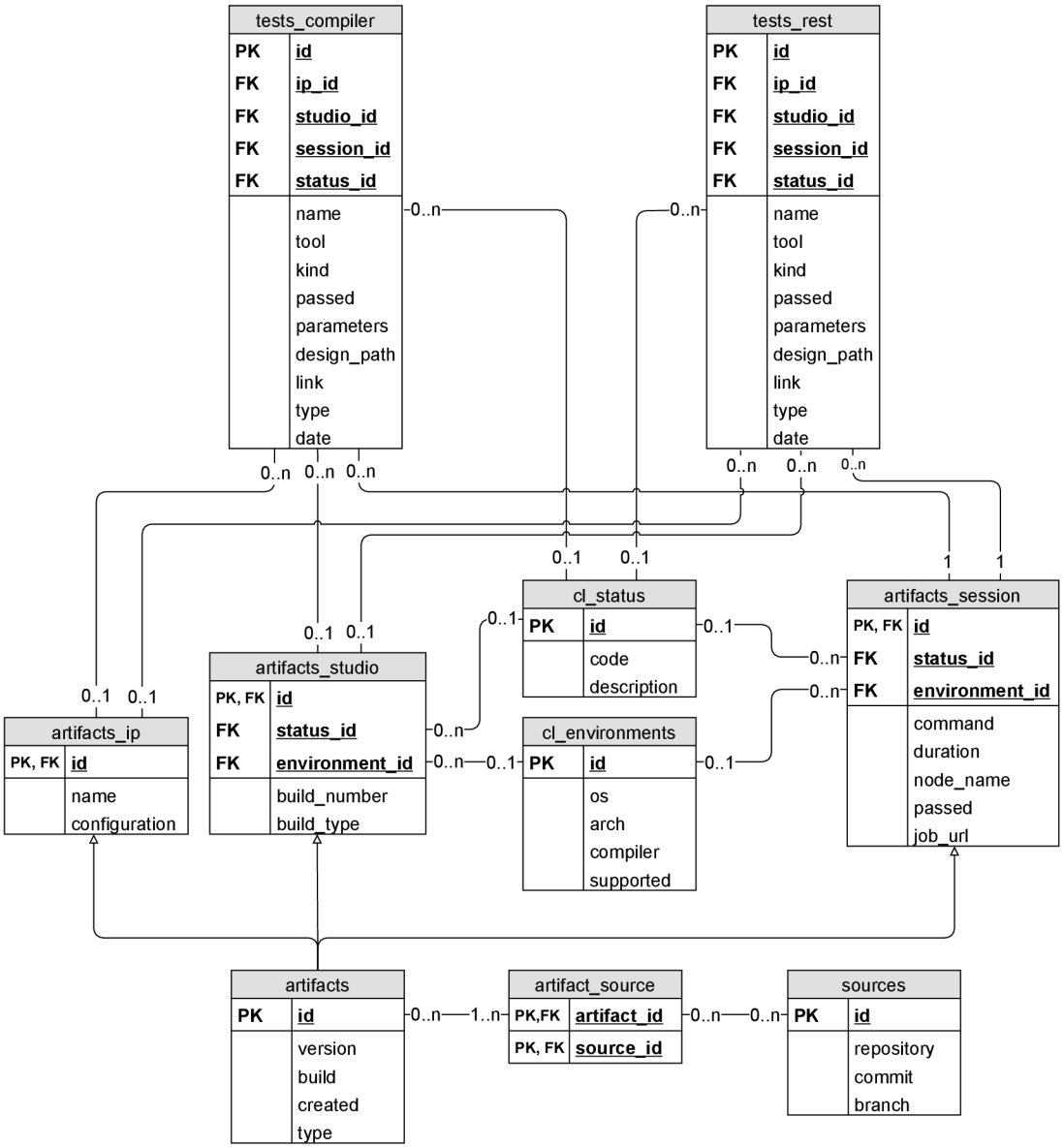
Moje výsledné řešení, při jehož návrhu jsem zohlednil výše uvedené požadavky, se skládá z několika dílčích částí. Návrhy jednotlivých dílčích částí postupně rozeberu v samostatných sekcích. Analogicky ke struktuře popisu návrhu je pak napsána i kapitola 7.

Identifikoval jsem, že struktura dat, se kterými systém pracuje, je vhodná pro zpracování relačním typem databází. Data jsou řádně strukturovaná na úrovni výstupu z testovacího frameworku. Zároveň ve firmě již řešení s využitím relační databáze existuje, což usnadní zavedení nového řešení. Stávající řešení ale bylo realizováno před několika lety a nebyly provedeny průběžné změny a vylepšení tak, aby systém odpovídal komplexnosti nových dat. Rozhodl jsem se proto ponechat relační typ databáze a stávající systém upravit tak, aby reflektoval velikost databáze i charakter dat, která jsou v databázi uložena.

Nové technologie typu NewSQL slibují snadné nasazení, použití i správu, maximální výkon apod. Pro firmu Codasip byla ale prioritou integrovatelnost do praxe, řešení mělo za cíl firmě ihned pomoci. Použití nových technologií je tedy dle mé analýzy pouze lákavým únikem od reálného problému. Vhodnějším přístupem je použít relační SŘBD a zaměřit se na pokročilou správu takovéhoto systému. Návrh, implementace a následná správa relačního SŘBD je nelehkým úkolem. Na základě studia existujících řešení, dostupných technologií a podrobné analýzy situace ve firmě Codasip jsem dospěl k závěru, že nejlepším řešením problému je použití stávajícího relačního SŘBD MySQL, upravit datový model a využít možností, které tento systém nabízí jako je například `partitioning` nebo uložené procedury a události.

6.1 Úprava datového modelu

Důležitým prvkem mého řešení je úprava datového modelu. Změny nejsou nikterak složité, ale měly by přinést zrychlení mnoha pohledů. Diagram upraveného schématu lze vidět na obrázku 6.1.



Obrázek 6.1: ER diagram upraveného schématu. Lze vidět rozdělení původní tabulky `tests` na `tests_compiler` a `tests_rest` a modelace vztahu `artifacts` a `sources` pomocí vazební tabulky `artifact_source`

Tabulku `tests` rozdělím na dvě části. První tabulka nazvaná `tests_compiler` bude obsahovat pouze výsledky regresních testů překladače. Druhá tabulka se jménem `tests_rest` bude obsahovat všechny ostatní záznamy. Tabulka `tests_compiler` bude stále tvořit většinu objemu dat, ale důležité je toto logické rozdělení. Atributy `tool` a `kind` v tabulce `tests_compiler` můžou působit redundantně, protože všechny záznamy budou mít stejné

hodnoty – vždy to budou řetězce `'compiler'` a `'regression'`. Sloupce ale ponechám, aby bylo možné provádět dotazy nad celou množinou testovacích záznamů bez nutnosti opětovně vkládat tyto sloupce a data do nich. Spojení tabulek `tests_compiler` a `tests_rest` lze snadno provést pomocí operace `UNION`. Rozdělení dat by šlo řešit i zavedením *partitions* na regresní testy překladače a zbytek (dále členěný nebo dohromady). Já jsem se rozhodl pro přímé rozdělení databázového modelu, kvůli snazší integraci do systému ve firmě Codasip a hlavně bez nutnosti správy *partitions*.

Dále jsem zavedl vazební tabulku `artifact_source` pro modelování vazby tabulek `artifacts` a `sources`. Tato úprava lze pozorovat ve spodní části diagramu 6.1. Tabulky `artifacts` i `sources` mají řádově pouze desítky tisíc záznamů, ale dle [6] se vazby n ku m modelují pomocí vazební entitní množiny a implementují právě vazební tabulkou.

6.2 Úprava a optimalizace SQL dotazů

Po důkladné analýze dotazů jsem zjistil, že dotazy obsahují různé množství drobných i větších nedostatků, které je třeba vyřešit. Jedná se o úpravu klauzulí `JOIN` a `WHERE`. Také se pokusím zefektivnit pohledy s podmíněnými `SUM` výrazy v klauzuli `SELECT`.

Podle analýzy dat v databázi a příkazu `SELECT DISTINCT name FROM artifacts_ip;` jsem zjistil, že každý typ procesorového jádra má vždy pouze jeden název. Pro jádro `codasip_uvliw` je to tedy vždy pouze řetězec `codasip_uvliw` a nevyskytují se žádné varianty jako by bylo například `codasip_uvliw_reduced` nebo `codasip_uvliw_version_2` a podobné. S tímto zjištěním jsem dále pracoval. V referenčních dotazech, jak je vidět na ukázce 6.1, se ale vyskytuje použití tzv. zástupného znaku – v jazyce MySQL znak procenta `%` pro název jádra. To je zbytečné a zejména použití zástupného znaku na začátku textového řetězce může zpomalovat vykonání dotazu. Databázový systém totiž nemůže použít prefixového porovnávání řetězců, hledaný řetězec může být přítomen kdekoli v porovnávaném řetězci. V případě zástupného znaku pouze na konci vyhledávaného řetězce lze použít prefixové porovnávání a použití zástupného znaku na konci vyhledávaného řetězce má tedy menší vliv na zpomalení rychlosti provedení dotazu.

Další z úprav bylo sjednocení jmenné konvence atributu `design_path` týkající se tabulek `tests_compiler` a `tests_rest`. Po konzultaci s vývojáři firmy Codasip jsem zjistil, že vkládání do databáze se děje na více místech zdrojového kódu testovacího frameworku. Vznikají zde dvě varianty názvu testovaného procesorového jádra – verze s tečkou (např. `codasip_urisc.ia`) a verze s pomlčkou (např. `codasip_urisc-ia`). Naprostá většina výsledků používá ale pomlčku, proto jsem se rozhodl sjednotit tuto jmennou konvenci na využití pouze pomlčky.

```
SELECT
    ‘artifacts‘.‘created‘ AS ‘date‘,
.
.
.
WHERE
       (‘tests‘.‘tool‘ = ’compiler’) AND
       (‘tests‘.‘kind‘ = ’regression’) AND
       (‘artifacts_ip‘.‘name‘ LIKE ’%codasip_uvliw%’)
GROUP BY ‘artifacts‘.‘created‘, ‘build_id‘;
```

Kód 6.1: Ukázka chybného použití zástupných znaků `%` v klauzuli `WHERE` v referenčním dotazu od firmy Codasip.

V rámci optimalizací SQL dotazů jsem navrhl i odlišný přístup k využití sumačních příkazů v klauzuli `SELECT`. V aktuální verzi dotazů (u těch, u kterých se sumace vyskytuje v podobě jak je vidět na ukázce 6.2) je prováděno vyhodnocení podmínek a případné přičítání k úhrnné hodnotě pro každý záznam, který je databází zpracováván. Já jsem navrhl nejprve provést agregaci přes patřičná data a to včetně atributu `passed`, který udává, zda test byl úspěšný nebo ne. Následně stačí pouze sloučit dílčí výsledky do stejné podoby, jako tomu bylo v původním dotazu.

```
SELECT artifacts.created AS date,
       CONCAT_WS('-', artifacts.version, NULL, artifacts.build) AS build_id,
       SUM((CASE
              WHEN tests.passed = 0 AND tests.design_path = 'codix_berkelium-ia' THEN 1
              ELSE 0 END)) AS failed_tests_ia,
       .
       .
       .
       SUM((CASE
              WHEN tests.design_path = 'codix_berkelium-ca' THEN 1
              ELSE 0 END)) AS sum_tests_ca
FROM (((tests_compiler AS tests
       INNER JOIN artifacts ON tests.studio_id = artifacts.id)
       INNER JOIN artifacts_studio ON tests.studio_id = artifacts_studio.id)
       INNER JOIN artifacts_ip ON tests.ip_id = artifacts_ip.id)
WHERE DATE(artifacts.created) >= @last_date AND
     artifacts_ip.name = 'codix_berkelium'
GROUP BY artifacts.created, build_id;
```

Kód 6.2: Kód původního využití příkazu `SUM` v klauzuli `SELECT`. Pro každý záznam jsou vyhodnoceny podmínky a případně je navýšena úhrnná hodnota.

6.3 Inteligentní správa materializovaných pohledů

Jako hlavní omezení aktuálního řešení vidím mazání záznamů, které se již nebudou měnit a následné opětovné zpracování těchto výsledků. Tento přístup není efektivní a jedná se o jednu z příčin pomalých dotazů. Navrhl jsem proto systém správy dat vložených do materializovaných pohledů, který efektivně pracuje s již uloženými daty a zbytečně nepřepisuje neměnná data.

Tento systém automaticky identifikuje, jaké agregované záznamy již nemůžou být změněny a s těmito záznamy již nijak nemanipuluje. Systém zjistí, jaké je datum záznamů, které můžou být ovlivněny nově příchozími daty. Takovéto záznamy jsou pak aktualizovány stejně jako v původním řešení. Tím, že se místo například půlroční historie zpracovává zpravidla pouze poslední den dochází k výraznému urychlení dotazů a tím se urychluje i vykonání celé aktualizační procedury.

Systém je odolný proti delšímu výpadku aktualizací a je schopen doplnit i více chybějících dnů. Celý proces je automatizován a součástí každé z procedur, které jsem v rámci této práce upravoval. Systém nevyžaduje žádné nadstandardní údržby nebo správy. Při vzniku nového materializovaného pohledu a procedury pro jeho aktualizaci, lze tento systém inteligentní správy dat snadno integrovat.

6.4 Partitioning

Pokročilejší práci s databázovým systémem představuje zavedení *partitions*. Datový model firmy Codasip nabízí dvě oblasti využití této technologie. První možností je využití *partitions* v tabulkách ukládajících data. Z hlediska objemu připadá v úvahu tabulka `tests`. Data jsou již rozdělena na regresní testy překladače a zbytek testů. Další často využívaná rovina rozdělení testů je atribut `design_path` definující druh procesorového jádra. Rozdělení do *partitions* lze tedy realizovat na základě tohoto atributu.

Druhou možností využití jsou materializované pohledy. S využitím `partitions` by bylo možné vylepšit systém inteligentní správy dat. Například mazání dat by bylo možné realizovat pomocí *partitions*, pokud by byly pohledy rozděleny do *partitions* podle kalendářních dat. Případně by se mohly starší – nepotřebné výsledky transportovat do archivního databázového schématu.

Kapitola 7

Implementace řešení

Řešení jsem na základě návrhu implementoval v podobě sady skriptů v jazyce SQL. Adresářová struktura a další podrobnosti týkající se zdrojových souborů těchto skriptů se nachází v souboru `README.me` na přiloženém médiu. Implementované změny a SQL skripty lze popsat následovným seznamem:

- Sada SQL skriptů pro transformaci datového modelu. Skripty provedou vytvoření nových tabulek a jejich naplnění patřičnými daty. Indexy jsou vytvořeny na vhodné atributy a vždy typu B–strom, protože jiný typ než B–strom index umožňuje pro MySQL pouze *engine Memory*[1]. Velikost databáze tohoto *engine* je limitována velikostí paměti RAM. Pro potřeby systému ve firmě Codasip proto nelze *engine Memory* použít. Použitý *engine InnoDB* nabízí pouze indexy typu B–strom.
- Sjednocení jmenné konvence atributu `design_path` obou tabulek `tests_compiler` a `tests_rest` jsem implementoval opět jako několik příkazů v jazyce SQL. Nyní se tedy v databázi nachází záznamy pouze s pomlčkou (např. `codasip_urisc-ia`).
- Optimalizované uložené procedury pro aktualizaci materializovaných pohledů:
  - V rámci úpravy procedur jsem implementoval i inteligentní správu již zpracovaných dat v materializovaných pohledech. Skripty automaticky zjistí datum nejstaršího záznamu. Následně smažou všechny záznamy z daného dne a nahradí je nově zpracovanými daty. V průběhu dne můžou dobíhat testovací běhy, a proto je nutné nejmladší záznamy zpracovat znova s novými daty. Díky této implementaci aktualizace pohledů jsou data vždy aktuální bez zbytečného opětovného zpracování historických záznamů.
  - Mimo inteligentní správu zpracovaných dat jsem upravil procedury tak, aby reflektovaly nový datový model. U regresních testů překladače došlo ke zjednodušení klauzulí `WHERE`, kde už není potřeba filtrovat výsledky podle atributů `tests_compiler.tool='compiler'` a `tests_compiler.kind='regression'`.
- Odstranil jsem nepotřebné podmínky v klauzuli `WHERE` u několika pohledů, kde byla podmínka `` `tests`.`studio_id` IS NOT NULL `` nicméně z povahy dotazu byly použity pouze spojení tabulek `tests` a `artifacts_studio` typu `INNER JOIN` a nelze tedy dostat z takového spojení prázdnou hodnotu cizího klíče (v tomto případě `studio_id`). Z tohoto důvodu je takováto podmínka zbytečná a byla odstraněna.

[1] https://dev.mysql.com/doc/refman/5.7/en/memory-storage-engine.html

- Přepracovaní příkazů `SUM` v klauzuli `SELECT` nevykázalo směrodatné zrychlení provedení dotazu. Tuto změnu jsem tedy neimplementoval více než na dvou zkušebních, respektive testovacích dotazech. Jak změna vypadá lze vidět v ukázce 7.1.

```
SELECT created AS date,
       build_id AS build_id,
       SUM((CASE
              WHEN passed = 0 AND design_path = 'codix_berkelium-ia' THEN count
              ELSE 0 END)) AS failed_ia,
       .
       .
       .
       SUM((CASE
              WHEN design_path = 'codix_berkelium-ca' THEN count
              ELSE 0 END)) AS total_ca
FROM (SELECT artifacts.created,
            CONCAT_WS('-', artifacts.version, NULL,
                 artifacts.build) AS build_id,
            design_path AS design_path,
            passed AS passed,
            count(passed) AS count
   FROM (((tests_compiler AS tests
          INNER JOIN artifacts USE INDEX (ix_artifacts_created)
             ON tests.studio_id = artifacts.id)
          INNER JOIN artifacts_studio ON tests.studio_id = artifacts_studio.id)
          INNER JOIN artifacts_ip ON tests.ip_id = artifacts_ip.id)
   WHERE DATE(artifacts.created) >= @last_date
          AND artifacts_ip.name = 'codix_berkelium'
   GROUP BY artifacts.created, build_id, design_path, passed) AS t
GROUP BY created, build_id;
```

Kód 7.1: Kód nového přístupu k využití příkazu `SUM` v klauzuli `SELECT`. Nejprve se provede předvýběr s agregací potřebných atributů a následně sloučení dílčích výsledků pro získaní kompletních dat. Tento přístup ale nepřinesl výraznější zrychlení dotazu a nebyl proto použit ve více pohledech. Dále lze vidět použití `USE INDEX (ix_artifacts_created)`, což je nápověda pro MySQL optimalizátor.

- Podmínky některých dotazů neměly dostatečnou selektivitu. Selektivitu lze v procentech vyjádřit vzorcem:

$$selektivita = kardinalita/(pocetZaznamu) * 100.$$

  Tato veličina určuje jaké množství záznamů bylo vyfiltrováno. Hodnota 100 % znamená, že navrácen nebyl žádný záznam. Hodnota 0 % analogicky znamená, že daná podmínka neměla žádný vliv a proces filtrování vrátil všechny záznamy. Je vhodné používat indexy s co nejvyšší selektivitou, protože navracejí nejmenší množství záznamů a to implikuje rychlejší zpracování dotazu. MySQL optimalizátor se o toto snaží, ale ne vždy je schopen vybrat ten nejlepší index.

  V pohledech, kde podmínky v klauzuli `WHERE` byly méně selektivní než u jiných pohledů jsem použil konstrukci jazyka SQL `USE INDEX (<index_name>)` v klauzuli `JOIN`, abych napověděl optimalizátoru MySQL, jaký index použít pro efektivnější filtrování dat. Jednalo se o využití indexu `ix_artifacts_created` pro sloupec `artifacts.created`, který určuje datum a čas vytvoření záznamu. Protože nový systém aktualizací materializovaných pohledů obnovuje data pouze za poslední den, je toto využití indexu efektivní v případech, kdy selektivita ostatních částí dotazu není

dostatečně vysoká a zároveň není jiná a lepší varianta využití jiného indexu tabulky `artifacts`. Jaké indexy jsou optimalizátorem vybrány a jaká je míra selektivity dotazu, jsem analyzoval pomocí příkazu `EXPLAIN` a sloupce `filtered`, který určuje míru filtrování záznamů. Pro ilustraci problematiky uvádím dvě ukázky, kdy bylo využití nápovědy `USE INDEX` výhodné, a kdy naopak bylo výhodné využít jiného indexu, respektive bylo výhodné ponechat výběr indexu na MySQL optimalizátoru.

V příkladě 7.1 lze vidět kód dotazu, který aktualizuje data materializovaného pohledu `mv_compiler_regression__urisc_sum_by_build`. V tomto dotazu je možné vidět v klauzuli `INNER JOIN` použití příkazu `USE INDEX (<index_name>)` pro nápovědu použití indexu. Protože záznamů pro jádro `codix_berkelium` je v tabulce `tests_compiler` velké množství, vyplatí se vyfiltrovat výsledky pomocí data vytvoření, protože z charakteru dat a vlastností systému vyplývá, že zpravidla potřebuji pouze výsledky za jeden den. Optimalizátor tuto informaci nemůže nijak získat, proto optimalizátoru napovím formou konstrukce `USE INDEX (ix_artifacts_created)`. Kompletní počáteční naplnění všech záznamů trvá s tímto přístupem déle než u původního referenčního dotazu. Nicméně s novým systémem se již materializovaný pohled nemaže celý, ale aktualizují se pouze výsledky z posledního dne. Díky této nápovědě pro použití jiného indexu jsem výrazně urychlil aktualizaci některých materializovaných pohledů.

V rámci testování jsem odhalil, že pro výrazné zrychlení je dostatečný můj systém na správu dat v materializovaných pohledech, který dle testování funguje spolehlivě. Zavedení *partitions* tedy vnímám jako volitelné řešení. Ačkoli jsem tuto část návrhu neimplementoval, považuji zavedení *parititons* jako jeden z prvních kroků při budoucí práci na systému pro zpracování testovacích výsledků ve firmě Codasip.

Kapitola 8

Testování a výsledky

Testování svého řešení jsem prováděl na dvou úrovních. Nejprve bylo potřeba ověřit, že moje úpravy generují validní výsledky. Ověření, že nové materializované pohledy obsahují správná data jsem prováděl manuálně a to tak, že jsem porovnával výsledky referenčních dotazů se svými výsledky. V některých případech byly očekávané výsledky zcela totožné i ve stejném počtu. U dotazů, kde jsem měnil klauzule `LEFT JOIN` na `INNER JOIN` jsem očekával rozdílný počet výsledných záznamů, ale data společných záznamů by měla zůstat stejná.

Druhou rovinou testování bylo ověření, že moje řešení je rychlejší než řešení referenční. Pro účely tohoto nefunkčního testování (kapitola 2 jsem vytvořil speciální uloženou proceduru `refresh_wrapper`. Tato procedura slouží ke spouštění procedur, které aktualizují jednotlivé materializované pohledy (dále jen aktualizační procedury). Uložená procedura `refresh_wrapper` očekává jeden vstupní argument, a to název procedury, která se má spustit. Procedura po doběhnutí spuštěné aktualizační procedury zaznamená do tabulky `refresh_event_log` datum a čas spuštění aktualizační procedury, údaje o návratovém kódu a době trvání aktualizační procedury a další informace. Tyto informace lze pak dále analyzovat a provádět různé statistiky.

Pro vizuální zobrazení dosažených výsledků jsem sestavil vizualizaci v prostředí aplikace Metabase. Metabase jsem spustil v kontejneru pomocí technologie Docker[1]. Vizualizace, jejíž podobu je možné vidět na obrázku 8.2, zobrazuje v grafu průměrnou dobu trvání vybraných procedur.[2] Graf je seřazen podle doby trvání jednotlivých procedur a obsahuje rozdělení sloupců podle verze dané procedury – stará verze první sloupec a nová verze druhý sloupec.

Dále jsem vytvořil v aplikaci Metabase dashboard (obrázek 8.1), který zobrazuje informace o rychlosti provádění procedur, které aktualizují materializované pohledy. Průměrná hodnota je počítána jako průměr průměrů všech trvání provádění procedur v tabulce `refresh_events_log`. Doby trvání nejpomalejších procedur jsou průměr všech běhů dané procedury. Tato vizualizace slouží jako ilustrativní příklad využití aplikace Metabase i k další správě dat, nejenom pro vizualizaci testovacích výsledků.

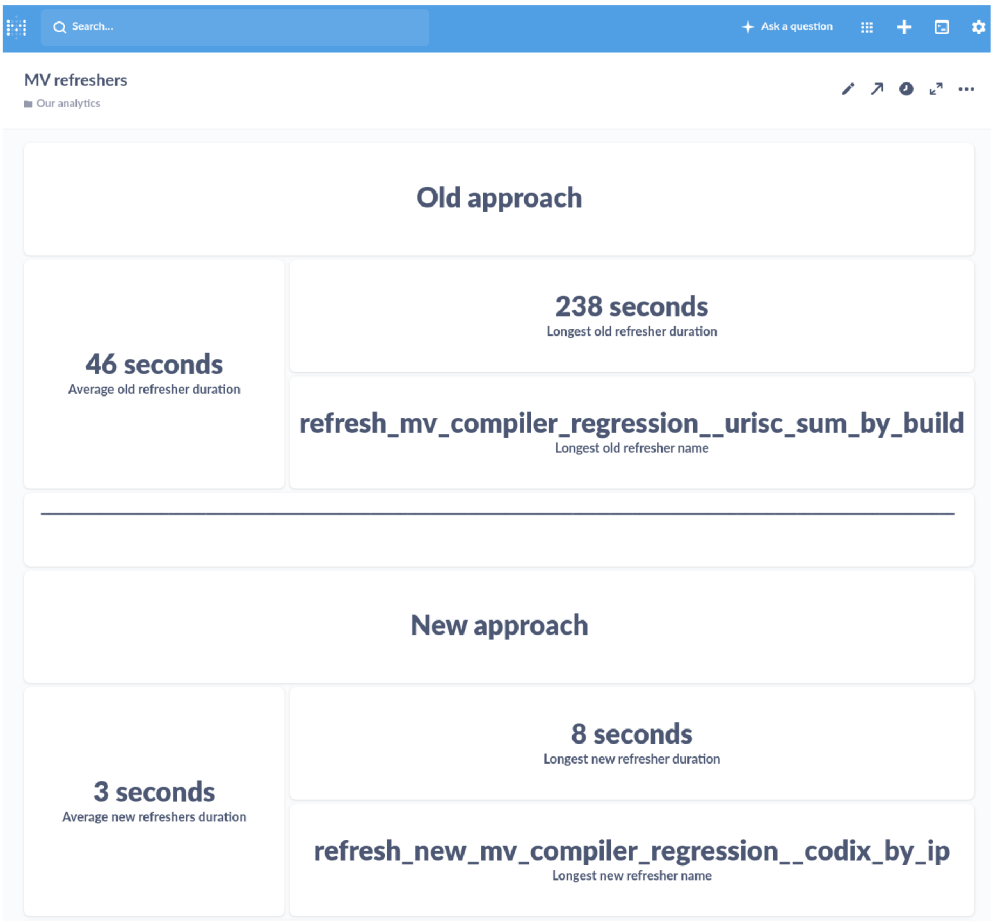
[1] https://www.docker.com/

[2] Zobrazeno je šest nejpomalejších, dalších jedenáct pak bylo rychlejších než 7 sekund.

8.1 Zhodnocení výsledků

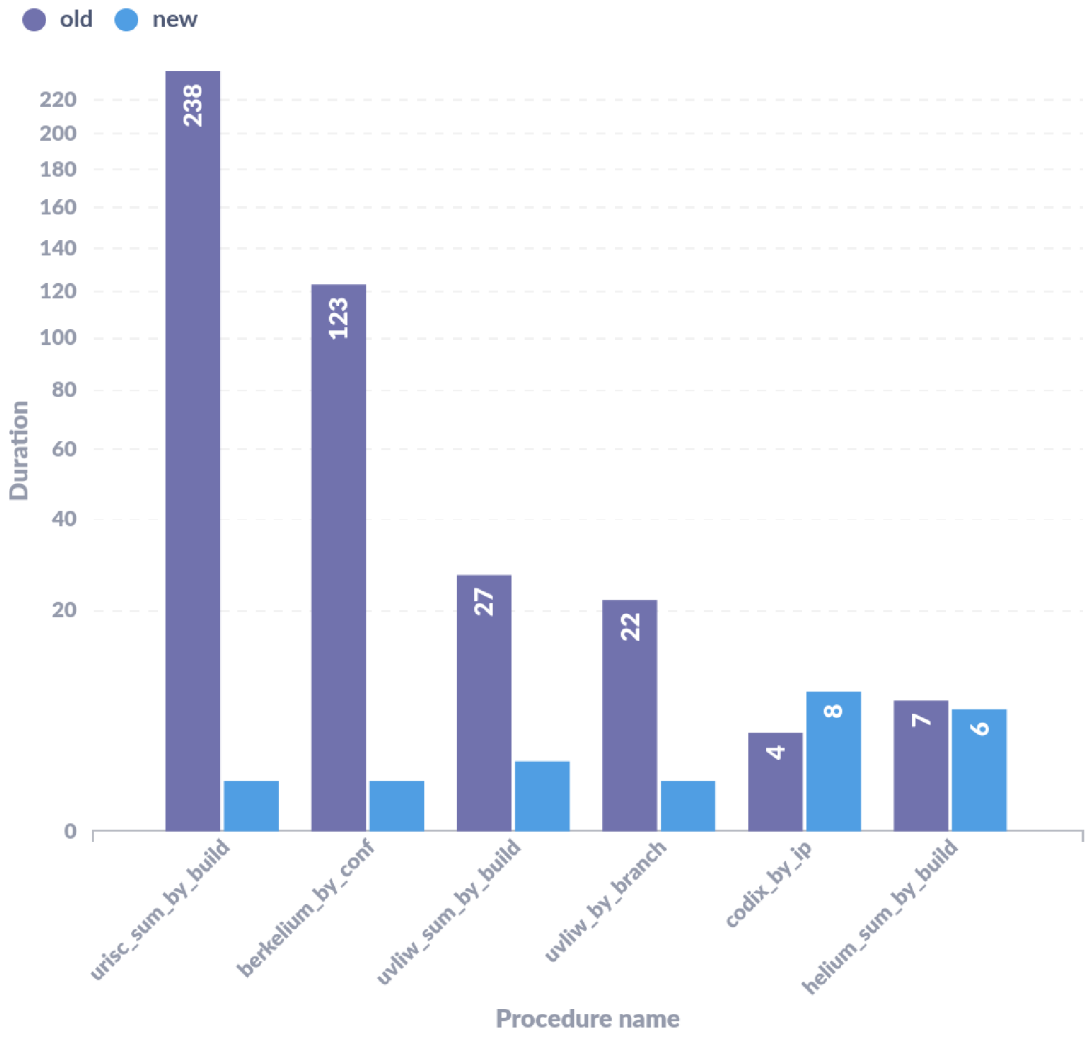
V této sekci jsem zhodnotil dosažené výsledky mého řešení. Dle dosažených výsledků, které lze vidět na obrázcích 8.1 a 8.2 je možné tvrdit, že nové řešení přineslo významné zrychlení v oblasti aktualizace materializovaných pohledů. Zároveň se mi podařilo splnit požadavky na řešení, které využívá nástroje, které jsou zdarma.

Maximální zrychlení se mi podařilo dosáhnout pro materializovaný pohled `mv_compiler_regression__urisc_sum_by_build`, který zobrazuje informace o regresních testech překladače pro procesorové jádro uRISC agregované na základě ID sestavení (verze Codasip Studia™). U tohoto pohledu jsem zrychlil jeho aktualizaci o více jak 99 %, protože průměrná doba aktualizace původní verze je 238 sekund a verze nové 1 sekunda[3] Následuje pohled `mv_compiler_regression__berkelium_by_conf` se zrychlením opět více jak 99 %, pohled `mv_compiler_regression__uvliw_sum_by_build` se zrychlením 93 % a `mv_compiler_regression__uvliw_by_branch` s průměrným zrychlením 95 %. Celkové průměrné zrychlení napříč všemi pohledy lze spočítat z hodnot v dashboardu na obrázku 8.1 a toto celkové průměrné zrychlení činí 93%.



Obrázek 8.1: Dashboard, který zobrazuje průměrné doby trvání procedur, které aktualizují materializované pohledy původního a nového řešení. Dále je zobrazena pro obě řešení nejpomalejší procedura a její doba trvání.

[3]Hodnoty menší než jedna sekunda jsou zaokrouhleny nahoru na jednu sekundu, zrychlení tak může být ještě větší.



Obrázek 8.2: Průměrná doba vykonávání šesti nejpomalejších aktualizací materializovaných pohledů. První sloupec je původní verze firmy Codasip a druhý sloupec je nová verze podle mého řešení.

Kapitola 9

Závěr

Práce řeší problém pomalých aktualizací materializovaných pohledů. Problém plyne z potřeby zpracovávat velký objem dat generovaných při testování ve firmě Codasip. Aktuální řešení nebylo dostatečně rychlé. Nejprve jsem nastudoval teoretické základy týkající se problematiky generování, ukládání, zpracování a vizualizace dat z testování software. Poznatky jsem sepsal do prvních teoretických kapitol. Následně jsem analyzoval a zhodnotil dostupné nástroje a existující řešení. Protože jsou požadavky firmy Codasip specifické a nepodařilo se mi najít existující řešení, které by bylo dostatečně vyhovující, navrhl jsem vlastní nové řešení.

Moje řešení se zaměřuje na úpravu datového modelu podle vlastností obsažených dat. Nový datový model je lépe strukturovaný s ohledem na následné zpracování dat. Dále jsem navrhl efektivní správu již zpracovaných dat a automatické doplňování pouze chybějících dat. Další částí návrhu byla optimalizace samotných dotazů. Navržené řešení jsem úspěšně implementoval a otestoval pomocí reálných referenčních dat, ke kterým mi dala firma Codasip přístup.

Za stěžejní části této práce, a oblasti, na které je dle mého názoru potřeba se zaměřit při řešení podobného problému, považuji detailní analýzu konkrétní situace, systému a dat, se kterými daný systém pracuje. Dále je důležité znát možnosti, silné stránky a omezení konkrétních použitých technologií. S komplexním pochopením technologií i dat pak lze úspěšně navrhnout a implementovat kvalitní řešení. Jsem názoru, že toto se mi v mé práci podařilo.

Dosáhl jsem velmi dobrých výsledků v oblasti zpracování nových dat do aktuálních materializovaných pohledů. Průměrné zrychlení činí 93 %, nejvyšší zrychlení pak více než 99 %. Předpokládám, že firmě Codasip toto zrychlení pomůže v efektivnějším vývoji procesorových jader i ostatních produktů. Zároveň jsem optimalizoval dotazy, které materializované pohledy aktualizují.

Jako další možnosti rozšíření řešení představeného v této práci vidím zavedení *partitions*, které umožní ještě více zrychlit databázové dotazy a optimalizovat práci s materializovanými pohledy. Protože datový model obsahuje často sloupce, jejichž hodnoty mají charakter vhodný k indexování bitmapovým indexem, bylo by vhodné detailně prostudovat jiné databázové systémy, který tento druh indexu nabízejí (například Oracle).

Literatura

[1] Chowdhury, A. R. *Best Python Testing Frameworks* [online]. LambdaTest, březen 2019 [cit. 2021-04-28]. Dostupné z: https://www.lambdatest.com/blog/top-5-python-frameworks-for-test-automation-in-2019/.

[2] Codd, E. F. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. červen 1970, sv. 13, č. 6, s. 377–387. DOI: 10.1145/362384.362685. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/362384.362685.

[3] Dougherty, J. a Ilyankou, I. *Hands-On Data Visualization*. 1. vyd. O'Reilly Media, Inc., 2021. ISBN 978-1492086000.

[4] Farcic, V. a Garcia, A. *Test-Driven Java Development*. 1. vyd. Packt Publishing, 2015. ISBN 978-1783987429.

[5] Gopal, R. *Email your pytest results as html report* [online]. Qxf2 Services, prosinec 2019 [cit. 2021-04-26]. Dostupné z: https://qxf2.com/blog/email-your-pytest-results-as-html-report/.

[6] Gordon, K. *Modelling Business Information: Entity relationship and class modelling for Business Analysts*. 1. vyd. BCS Learning & Development Limited, 2017. ISBN 978-1780173542.

[7] Hambling, B., Morgan, P., Samaroo, A., Thompson, G. a Williams, P. *Software Testing - An ISTQB-BCS Certified Tester Foundation guide*. 4. vyd. BCS Learning & Development Limited, 2019. ISBN 978-1780174921.

[8] Hamill, P. *Unit Test Frameworks*. 1. vyd. O'Reilly Media, Inc., 2004. ISBN 978-0596006891.

[9] Hammink, J. *The Types of Modern Databases* [online]. Alooma, Inc., březen 2018 [cit. 2021-04-29]. Dostupné z: https://www.alooma.com/blog/types-of-modern-databases.

[10] Hernandez, M. J. *Database Design for Mere Mortals™*. 2. vyd. Addison-Wesley Professional, 2003. ISBN 978-0201752847.

[11] Katalon Solution. *Best Practices for Test Automation* [online]. Katalon, Inc., září 2020 [cit. 2021-04-30]. Dostupné z: https://www.katalon.com/resources-center/blog/test-automation-best-practices/.

[12] Kelechava, B. *The SQL Standard – ISO/IEC 9075:2016 (ANSI X3.135)* [online]. ANSI, říjen 2018 [cit. 2021-04-30]. Dostupné z: https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/.

[13] Khoja, S. *What is SQL? What is a Database? Relational Database Management Systems (RDBMS) Explained in Plain English.* [online]. Free Code Camp, Inc., prosinec 2020 [cit. 2021-04-29]. Dostupné z: https://www.freecodecamp.org/news/sql-and-databases-explained-in-plain-english/.

[14] Krekel, H. a pytest-dev tým. *Pytest: helps you write better programs* [online]. Holger Krekel a pytest-dev tým, 2020 [cit. 2020-12-23]. Dostupné z: https://docs.pytest.org/en/stable/.

[15] Kulla Mader, J. *Build Python apps* [online]. Microsoft, duben 2019 [cit. 2021-04-27]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/pipelines/ecosystems/python?view=azure-devops.

[16] Lewis, W. E. *Software Testing and Continuous Quality Improvement.* 3. vyd. Auerbach Publications, 2017. ISBN 978-1351722209.

[17] Melton, J. a Simon, A. R. *Understanding the New SQL.* 1. vyd. Morgan Kaufmann, 1993. ISBN 978-1558602458.

[18] Oracle and/or its affiliates. *Transactions* [online]. Oracle and/or its affiliates, březen 2017 [cit. 2021-04-29]. Dostupné z: https://docs.oracle.com/database/121/CNCPT/transact.htm#CNCPT016.

[19] Petrov, A. *Database Internals.* 1. vyd. O'Reilly Media, Inc., 2019. ISBN 978-1492040347.

[20] Philippe, M. a ostatní. *Junit5-jupiter-starter-gradle* [online]. Marc Philippe a ostatní, 2021 [cit. 2021-02-27]. Dostupné z: https://github.com/junit-team/junit5-samples/tree/r5.7.1/junit5-jupiter-starter-gradle.

[21] Redis.io. *An introduction to Redis data types and abstractions – Redis* [online]. Redis.io, 2021 [cit. 2021-03-03]. Dostupné z: https://redis.io/topics/data-types-intro.

[22] Schaefer, L. *What is NoSQL?* [online]. MongoDB, Inc., 2021 [cit. 2021-03-02]. Dostupné z: https://www.mongodb.com/nosql-explained.

[23] Sturgis, D. a ostatní. *Intro-to-pytest* [online]. David Sturgis a ostatní, září 2020 [cit. 2021-02-27]. Dostupné z: https://github.com/pluralsight/intro-to-pytest.

[24] Tricentis team. *Test Automation vs. Automated Testing: The Difference Matters.* Tricentis, leden 2017 [cit. 2021-04-30]. Dostupné z: https://www.tricentis.com/blog/test-automation-vs-automated-testing-the-difference-matters/.