VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VERZOVÁNÍ DATABÁZE PŘI VÝVOJI APLIKACÍ

DATABASE VERSION MANAGEMENT FOR APPLICATION DEVELOPMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Jan Kotráš**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. Marek Rychlý, Ph.D.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Kotráš Jan**
Obor: Informační technologie
Téma: **Verzování databáze při vývoji aplikací**
Database Version Management for Application Development
Kategorie: Databáze

Pokyny:

1. Seznamte se s aktuální problematikou a se systémy správy verzí zdrojových kódů a jiných artefaktů procesu vývoje software. Zaměřte se na problémy a řešení verzování schémat různých typů databází.
2. Navrhněte metodiku pro správu verzí relační schéma databáze s podporou větvení při jeho vývoji. Umožněte dělení a slučování verzí schéma během vývoje. Vytvořte sadu vzorových příkladů, které metodika řeší.
3. Po konzultaci s vedoucím implementujte pro databázový systém MySQL/MariaDB nástroj, který umožní verzování schémat databází dle navržené metodiky.
4. Výsledný nástroj vyzkoušejte na vzorových příkladech a, pokud možno, i v praxi.
5. Zdokumentujte výsledky a zveřejněte projekt pod svobodnou licencí jako open-source.

Literatura:

- John F. Roddick. *A survey of schema versioning issues for database systems*. Information and Software Technology, Volume 37, Issue 7, 1995, pp. 383-393. ISSN 0950-5849. [http://dx.doi.org/10.1016/0950-5849(95)91494-K]
- Erhard Rahm, Philip A. Bernstein. *An online bibliography on schema evolution*. SIGMOD Rec. 35, 4, 2006, pp. 30-31. [http://dx.doi.org/10.1145/1228268.1228273]
- Vít Palarczyk. *Verzování databáze při vývoji aplikací v Eclipse*. Bakalářská práce, FIT VUT v Brně, 2012. [http://www.fit.vutbr.cz/study/DP/BP.php?id=13288]

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2, a započatá práce na řešení bodu 3.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L.S.

Abstrakt

Cílem této práce je usnadnění verzování databázových systémů pracujících na systému řízení báze dat MySQL. Záměr aplikace je zefektivnění a zjednodušení verzování databázové vrstvy při vývoji aplikací v menších týmech.

Abstract

The purpose of this work is to facilitate database version working on the database management system MySQL. The intention of application is to get efficiency and make simpler versioning of database layer for application development in smaller teams.

Klíčová slova

Databáze, vývoj software, verzování databází, verzování, Python, Mysql, Git, SVN, informační systémy

Keywords

Database, software development, database version, versioning, Python, Mysql, Git, SVN, information systems

Citace

Jan Kotráš: Verzování databáze při vývoji aplikací. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Verzování databáze při vývoji aplikací

Prohlášení

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

……
Jan Kotráš
12.května 2017

Poděkování

Děkuji všem, kteří mi jakoukoliv formou pomohli na této práci. Zejména bych jsem chtěl poděkovat svému vedoucímu moji bakalářské práce RNDr. Marku Rychlému, Ph.D. za jeho vedení, připomínky a směrování celé této práce jasným cílem. Také bych rád poděkoval firmě Izon s.r.o za poskytnutí podmínek pro testování. A v neposlední řadě bych také rád poděkoval svým rodičům, nejbližší rodině a kolegům.

Obsah

1 Úvod

V současnosti je poměrně ojedinělý vývoj aplikací jedním vývojářem. Aplikace se stávají komplexní a složitější na implementaci. Z tohoto důvodu je při jejich vývoji potřeba více vývojářů, kde každý z nich se zaměří na konkrétní úsek problematiky vyvíjené aplikace. Pro práci v týmu vývojářů je však nutné zavézt efektivní způsob, kterým jsi vývojáři budou navzájem sdílet části aplikace mezi sebou, verzovat zdrojový kód a případně jej slučovat. Pro tento účel byly vyvinuty verzovací systémy, které problematiku velice efektivně řeší. Jistý problém však nastává, pokud je součásti vyvíjené aplikace databáze, neboť verzovací systémy pracují nad soubory a o databázi jako o souboru je velice těžké hovořit.

Úkolem této práce je prozkoumat současné možnosti verzování databází. Zhodnotit jejich klady a zápory. Následně na těchto poznatcích navrhnout a implementovat systém pro verzování databáze. Výsledný systém následně otestovat, pokud možno v ostrém provozu.

Práce se zaměřuje na vytvořením nástroje pro podporu verzování databází používajícím systém řízení báze dat MySQL v kombinaci s libovolným verzovacím systémem. Cílem je vytvořit takový systém, který bude efektivně podporovat verzování databázové struktury v menších vývojářských týmech.

2 Problematika verzování databází

V této kapitole představím obecné způsoby verzování databázových systémů. Zejména se zaměřím na způsoby verzování založené na převodu databázové struktury do textového souboru. Dále pak zhodnotím systémy již vytvořené a používané.

Běžné verzovací systémy zdrojového kódu pracují nad textovými soubory. To jim umožňuje tyto soubory porovnávat, řešit vzájemné kolize a případné je slučovat. Avšak databáze není textový soubor, proto ji nelze přímočarým způsobem verzovat. Databáze je uložena v závislosti na systému řízení báze dat (dále jen SŘBD) v binárních souborech. Jejich obsah je často čitelný pouze systémem řízení báze dat a tak je nečitelný pro ostatní aplikace a už vůbec ne pro vývojáře. Často jsou také soubory SŘBD závislé i na provozované platformě a tedy nejsou přenositelné na jiné platformy [1].

Z tohoto důvodu se v oblasti verzování databází vytvořily dva základní přístupy jak databáze a jejich schémata verzovat. První přístup je složitější jak na implementaci a údržbu, vhodný pro větší databázové systémy. Tento směr nebudu nijak podstatně rozebírat, neboť se zaměřuje na větší databázové systémy na které v této práci není cíleno. Pouze nastíním hlavní vlastnosti a principy daného přístupu. Druhý přístup je obecnější a vyhovuje požadavkům verzovacích systémů, pro to aby verzované soubory byly v textovém formátu. Bohužel obecnost tohoto způsobu sebou nese spousty nedokonalosti a problémy o kterých budu diskutovat a optimalizovat je.

První přístup je uplatňovaný ve velkých databázových systémech sebou nese specifické požadavky na systém řízení báze dat a často také na samotný verzovací systém. Jeho princip spočívá ve verzování pomocí specializovaných verzovacích nástrojů spolupracujících přímo ze SŘBD. Z čehož plyne jeho velká efektivita ale také závislost na konkrétní platformě SŘBD. Hlavním rysem je také verzování databáze bez nutnosti iniciativy vývojáře v SQL kódu během verzování. Tedy téměř veškerá obsluha běžných činnosti probíhá prostřednictvím grafického rozhraní. Do této skupiny patří aplikace vytvořené pro větší databázové systémy od firem Oracle a Microsft, kde se můžeme setkat například z aplikací SQL Source Control pro SQL Server či Schema Version Control For Oracle pro databázový server od firmy Oracle.

Druhý způsob verzování databáze je založený na převodu databázové struktury do textového souboru [2]. Textový soubor či soubory obsahují popis vytvoření databázového obsahu. Popis je tvořen nejčastěji sekvencí SQL příkazu. Způsob kterým lze databázovou strukturu převézt na sekvenci SQL příkazů umožňuje téměř jakýkoliv typ databáze, jedná se o tzv. „snímky“ často také nazývány jako „snapshoty“ či „dumpy“. Tyto snímky jsou následně vloženy do adresářové struktury do adresáře, který je pod správou verzovacího systému, a tím i zpřístupněný pro ostatní vývojáře, kteří jsi soubory reprezentující databázi mohou naimportovat na svá lokální zařízení. Snímky jsou tvořeny nejprve úvodními příkazy pro vyčištění databáze a následné importními příkazy, které nová data zapíšou do již zcela prázdné databáze.

2.1 Obecné přístupy

Systémy založené na exportu databázové struktury do souboru či souborů, lze považovat za obecný způsob verzování databází. Jejich implementace a provoz má několik nedostatků, které zde postupně rozeberu.

Pokud zvolíme metodu exportu do jednoho souboru, tak se velice brzy můžeme potýkat s problémy týkající se velikosti souboru. Exportovaný soubor může velice rychle dosáhnout velikosti stovek MB. S takovýmto souborem se velice špatně pracuje. Je téměř needitovatelný. Jeho pouhé otevření může trvat spousty času. Řešení konfliktů a slučovaní na úrovni verzovacího systému je tak téměř nemožné. Proto se velice často volí varianta exportu databáze do několika souborů. Zpravidla se jeden soubor týká jednoho elementu v databázi. Např. Jeden soubor pro jednu tabulku.

Další částí této problematiky je začlenění vytvořených souborů do verzovaného prostředí. Což se na první pohled může zdát bezproblémové, neboť vyexportovaný soubor reprezentující

databázovou strukturu vložíme pod správu libovolného verzovacího systému. Avšak exportované soubory reprezentující databázovou strukturu musí být efektivně rozděleny a rozřazeny tak, ať umožňují jednoduché a efektivní verzování. Zejména pak řešení konfliktů a správu větví.

Dílčím nedostatkem je neimplicitnost provádění exportu, tedy neimplicitnost vytváření snímků databázové struktury při vytváření nové revize ve verzovaném prostředí. Vývojář musí tedy před nově vytvořenou revizí ve verzovaném prostředí, která obsahuje i změnu databázové struktury, vyexportovat aktuální verzi databáze. V případě nedodržení tohoto kroku by mohlo dojít k nekonzistenci databáze vůči aplikaci. Neboť aplikace by byla umístěna ve verzovaném prostředí v novější verzi než opomenutá databáze.

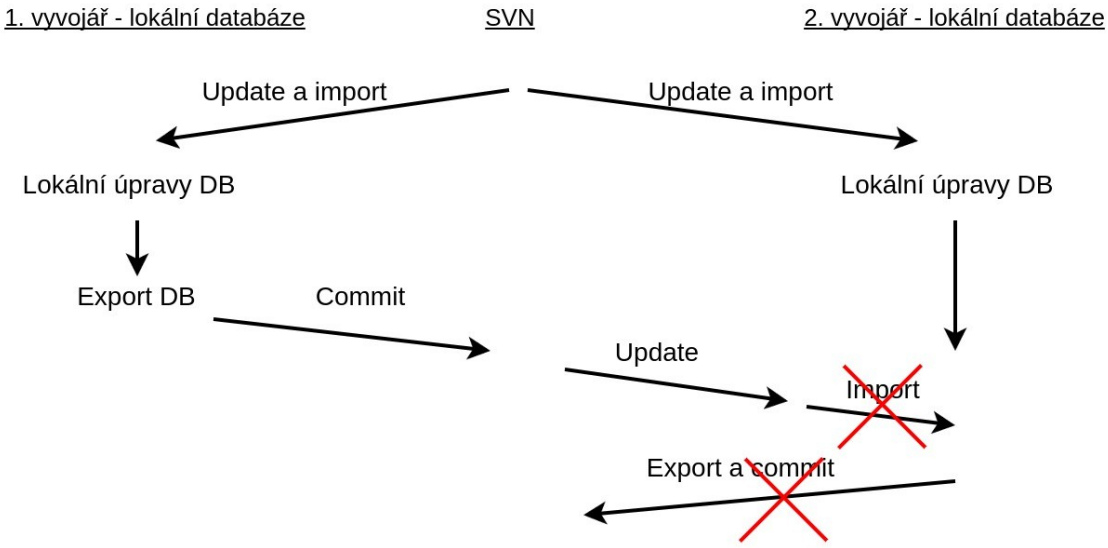
Dalším bodem je rychlost provádění způsobu verzování. Databáze může obsahovat velké množství záznamů. Jednotky či desítky tisíc záznamu můžou být mírně problematické při exportu do souborů, avšak protože se jedná o čtení z databáze společně ze zápisem do souboru je proces dostatečně rychlý i při větším počtu záznamů. Mnohonásobně kritičtějším bodem je import. Import neboli zápis do databáze je časově mnohonásobně složitější než čtení, proto je nutné při vytváření postupů pro verzovaní databáze také pamatovat na rychlost importu.

Obecným nedostatkem exportu a importu, je provádění nadbytečného množství SQL příkazů. Při jakékoliv změně v databázi, ať už při sebemenší změně je nutné celou databázi vyexportovat. Ostatní vývojáři jsi musí svoji lokální databázi kompletně přepsat vyexportovaným souborem. Neboť vyexportovaný soubor obsahuje SQL příkazy pro vyčistění databáze a následné naplnění novými daty.

Další problém vzniká při souběžné práci dvou a více vývojářů. Mohou vznikat kolize a nechtěné přepsání úprav jednoho z vývojářů. Tyto problémy budu demonstrovat za použití verzovacího systému SVN, při souběžném vývoji dvou vývojářů. Následně problémy budu využívat jako sadu vzorových příkladů pro analýzu současných systémů a návrh vyvíjené aplikace.

První typ kolize, graficky doplněna obrázkem 1:

- Vývojář 1 jsi naimportuje aktuální verzi databáze.
- Vývojář 2 jsi naimportuje aktuální verzi databáze.
- Oba vývojáři začnou upravovat databázovou strukturu dle svých požadavků.
- Vývojář 1 ukončí práci, exportuje databázi a provádí commit do centrálního repozitáře.
- Vývojář 2 chce úpravy provedené vývojářem 1. ve své lokální databázi také. Proto updatuje a získává data ze serveru.
- Vývojář 2 se však nyní dostává do kolize, neboť databázi nemůže ze novějšího snímku importovat, neboť by jsi přepsal své úpravy, avšak nemůže ani exportovat neboť by přepsal úpravy vývojáře 1.

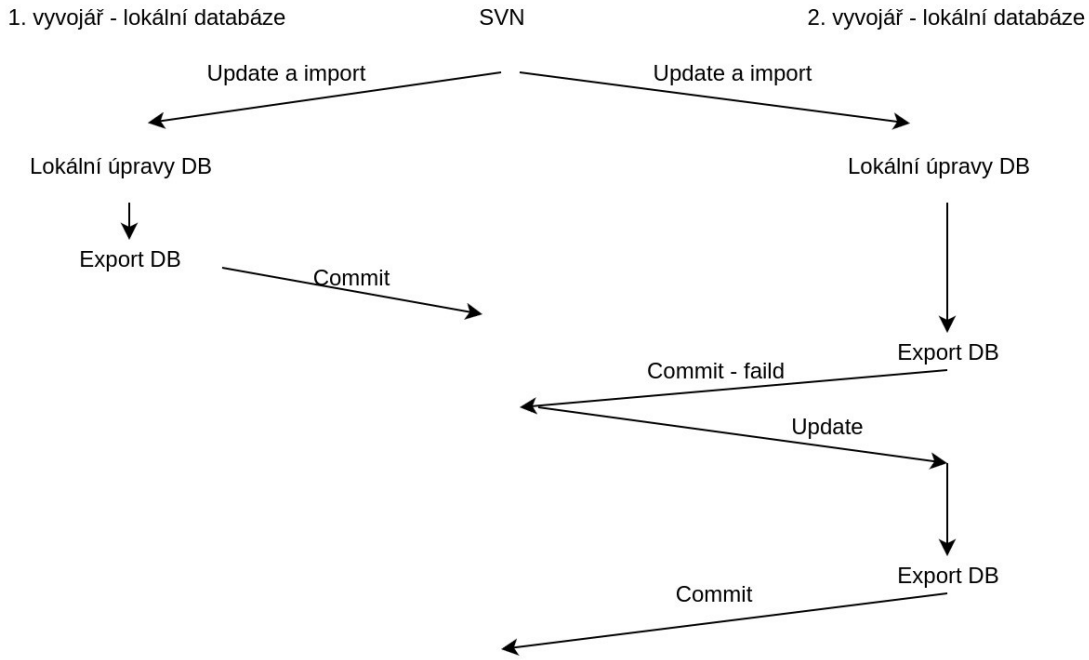


Obrázek 1: Kolize při použití export/importu databáze

Řešení výše uvedeného problému je velice problematické. Nejoptimálnější řešení je kdy vývojář 2 vloží SQL příkazy, které chce uveřejnit, do exportu od vývojáře 1. Toto řešení je však značně rizikové, protože může dojít ke spoustě kolizí a nekonzistenci exportovaného souboru (ať už z pohledu syntaxe či sémantiky). Proto se v praxi nejčastěji vývojář 2 uchýlí k importu od vývojáře 1 a provedení svých úprav znovu.

Další kolize, která může vznikat při verzování databází při použití metody importu a exportu je velice podobná předcházející kolizi, obrázek 2:

- Vývojář 1 jsi naimportuje aktuální verzi databáze
- Vývojář 2 jsi naimportuje aktuální verzi databáze
- Oba vývojáři začnou upravovat databázovou strukturu dle svých požadavků.
- Vývojář 1 ukončí práci, exportuje databázi a commituje do centrálního repozitáře.
- Vývojář 2 ukončí práci později, svá data exportuje a chce ji také commitovat, avšak nemůže neboť v repozitáři je již novější verze od vývojáře 2.
- Vývojář 1 se tak rozhodne pro update, aby jsi zpřístupnil novou verzi repozitáře.
- Při update nastane SVN kolize, která i manuálně velice těžce řešitelná. Při řešení kolize by mohlo dojít k porušení syntaxe, proto vývojář akceptuje jednu z verzí (svoji nebo vývojáře číslo 2)
- Vývojář 2 následně provede export a commit. Tím ale přepsal změny vývojáře 1.



Obrázek 2: Přepsání data při použití importu/exportu databáze

Již z popisu tohoto konfliktu je jasné že by ke konfliktu nemuselo docházet, neboť vyřešení kolize se pohybuje na úrovni verzovacího systému. V praxi se však setkáváme s tím že vývojář akceptuje pouze některou verzi a tím tu druhou přepíše. Koná tak z důvodu toho, že SQL snímky jsou často nepřehledné a řešení kolize na úrovni verzovacího systému by přineslo potencionálně ještě více problémů něž přepsání uprav jinému vývojáři. Zejména se jedna o porušení konzistence snímku, porušení syntaxe, protože kolize může být velice rozsáhlá a nepřehledná.

2.2 Aplikace pro verzování Databází

2.2.1 Komerční produkty

Pro velké databázové systémy vyvíjené firmami Microsoft či Oracle jsou k dispozici tzv. development nástroje, které umožňují verzování databázových schémat pro danou platformu SŘBD. Práce s nimi je velice přehledná, často na úrovni GUI. Při použití těchto systémů se můžeme z velké části vyvarovat od přímého kontaktu s SQL příkazy, což je jistým plusem. Avšak tyto systémy, jejich implementace a work-flow je velice svázaná z konkrétním produktem, což nám znemožňuje implementovat postupy na obecnější úrovni, či na úrovni volně dostupných databázových systému, jak je databázový systém MySQL.

2.2.2 Laravel

Laravel je velice rozšířený PHP framework. Na poli verzování databázových schémat v oblasti PHP fremeworků jej lze považovat za nejpropracovanější systém. Jeho základním stavebním kamenem jsou tzv. Migrace.

Migrace [3] je speciální typ PHP třídy obsahující zpravidla dvě metody, *up* a *down*. Metody jsou psané v pseudo SQL jazyce v konvencích frameworku. Metoda *up* definuje způsob přechodu

databázového schématu k novější migraci. Metoda *down* definuje opačný postup, tedy způsob navráceni verze. Součásti této migrace mohou být i meta-informace pro přehlednější verzování. Jedná se datum vytvoření, autora atd. Ilustraci migrace je možné vidět na obrázku 3.

Každá migrace je uložená v separátním souboru, který je unikátně pojmenovaný. Pojmenování je založeno na kombinaci textu a timestampu (časového razítka). Migrace jsou uloženy ve společné složce na souborovém systému, kde se předpokládá že daná složka je pod správnou verzovacího systému.

Laravel následně nabízí, mnoho rutin zpřístupněných pomocí konzole. Mezi tyto rutiny patří nejčastěji používané – přechod na novou verzi a rollback, tedy návrat ve verzi.

Koncept verzování databáze ve frameworku Laravel se velice osvědčil na poli menších databázových systémů. Proto jej lze nalézt i v jiných frameworcích pro PHP či jiné skriptovací jazyky. Jedná se o framework Symfony[4], avšak až ve verzi 2, a nebo také framework Django pro jazyk Python[5].

Jak vyplývá z celého popisu tohoto konceptu, tak tento způsob verzování je velice propracovaný a ověřený. Nese sebou však jisté nedostatky. Vývojář musí do každé migrace manuálně zasahovat respektive ji vytvářet, tedy se nepřímo setkává z SQL kódem nejen při řešení konfliktů. Často práce vývojáře spočívá v provedení změn v lokální databázi, nejčastěji pomocí grafického rozhraní, testování a následné uveřejnění ve verzovacím systému. Tento koncept práce však Laravel nabourává, protože vývojáře nutí provedené změny v databázi přepsat do migrací, což je pro vývojáře velice nepříjemný krok, kde musí SQL kód redundantně vytvářet. Také druhým a podstatným nedostatkem je nutnost použíti takovéhoto frameworku, což skupinu vývojářů pro které software bude užitečný, velice omezuje.

```
class CreateUserTable extends Migration
{
    /**
     * Run the migrations.
     * @table user
     *
     * @return void
     */
    public function up()
    {
        Schema::create('user', function (Blueprint $table) {
            $table->engine = 'InnoDB';
            $table->increments('id');
            $table->string('name', 45)->nullable();
            $table->string('email', 45)->nullable();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
      Schema::dropIfExists('user');
    }
}
```

Obrázek 3: Ukázka migrace - framework Laravel

2.2.3 Liquibase

Liqubase [6] je volně dostupný software napsaný v jazyce Java pro verzování databází z různým systémem řízení báze dat. Podporuje i systém řízení báze dat MySQL. Pracuje na principu změnových logů. Změnové logy mohou být ukládány v formátu *XML*, YAML, JSON či přímo v podobě sekvence *SQL* příkazů. Opět jsou tyto logy ukládány na souborový systém a následně verzovány. Velkou výhodou Liqubase je nezávislost na platformě databáze a na verzovacím systému. Liquibase není nijak závislý na programovacím jazyku ve kterém se vyvíjí samotná aplikace.

Celá struktura databáze a je tak tvořena na základě změnových souboru, kde každý soubor definuje příslušnou změnu verzi databáze. Změnu lze považovat za inkrementální popis přechodu od starší verze k novější. Celé databáze se skládá z jednotlivých změnových souborů. Kde každý změnový soubor tvoří rozdíl (inkrement) mezi jednotlivými verzemi databáze, viz. obrázek 4.

Obrázek 4: Skládání verzí databáze pomocí změnových souborů - Liquibase

Při vytváření změnových logů využívá podobného konceptu jako Laravel. Změnové soubory jsou napsány vývojářem, proto jsou jednoduše čitelné a sledovatelné. Liqubase nad změnovými logy umožňuje spousty rutin, které podobně jako v případě Laravel mají usnadnit vývojáři práci.

Liquibase je určený na středně velké databáze, proto se velice hodí pro porovnání s mojí prací. Také reflektuje problémy verzování databází, kterém jsem uvedl na začátku této kapitoly a snaží se je odstranit či alespoň zmírnit ku prospěchu přehlednosti a šetření času vývojářů.

2.2.4 Phpmyversion a DBV

PhpMyVersion a DBV(Database version control) jsou aplikace vytvořené pro databázi MySQL a programovací jazyk PHP, což systémy předurčuje a omezuje na vývoj webových aplikací na platformě PHP + Apache + MySQL.

DBV [7] je vkusně zpracovaná aplikace se kterou probíhá komunikace pomocí webového rozhraní. Je přehledná a jednoduchá proto se hodí pro malé projekty. Princip fungování je založený na prostém exportu a importu databáze. Což vynucuje přehrání celé databáze i v případě malé změny.

PhpMyVersion [8] je také ovládané pomocí webového rozhraní. Avšak naproti DBV využívá rozdílové, inkrementální soubory popisující jednotlivé změny mezi verzemi databáze pomocí SQL příkazů. Velice zajímavý prvkem je získávání dat pro inkrementální soubory. Inkrementy jsou získávány z binárních logů, které databáze MySQL umožňuje vytvářet. Aplikace sama o sobě má zajímavý potenciál. Bohužel její poslední vydaná verze je verze 0.1.1 v roce 2013, tedy aplikace již není vyvíjena a obsahuje spousty nedostatků a chyb pro praktické použití.

3 Použité technologie

Tato kapitola se zabývá použitými technologiemi při řešení práce. Podkapitoly vždy obsahují zběžný úvod do dané technologie a následně podkapitola pokračuje výkladem specifik technologie uplatněných v této bakalářské práci.

3.1 MySQL

MySQL[9] je databázový systém řízení báze dat založený na relačním databázovém modelu. Je vytvořený a spravovaný švédskou firmou MySQL AB. SŘBD je k dispozici pod licencí GPL či pod placenou komerční licencí. MySQL je multiplatformní, což jej předurčuje, společně s licencí GPL, pro projekty menšího rozsahu. Jedná se nejčastěji o webové aplikace a menší informační systémy. A tak je MySQL nejčastěji kombinováno z Linoxovým operačním systémem, PHP programovacím jazykem a Apache webovým serverem.

MySQL bylo až do nedávno prezentováno jako velice jednoduchý databázový software, avšak v řádu několik posledních let byly doimplementovány funkce známe z velkých databázových systémů. Jedná například o triggery, pohledy a další. Tento krok posouvá databázi i pro uplatnění ve větších databázových aplikacích.

Systém nabízí několik typů datových úložišť, tzv. Enginu. Mezi nejznámější patří enginy MyISAM a InnoDB. Tyto dva enginy mají mezi sebou podstatné rozdíly. MyISAM je engine optimalizovaný pro čtení velkého množství dat. K tomuto mu napomáhá full-textové indexování. Avšak engine MyISAM sebou nese jistá negativa. MyISAM nepodporuje transakce a kontrolu cizích klíčů. Naproti tomu InnoDB podporuje provádění transakcí i kontrolu cizích klíčů. To vše ale za cenu nižšího výkonu při čteni, zejména při operacích týkající se fulltextové vyhledávání (platí pouze pro starší verze, ve verzi MySQL 5.6 již InnoDB podporuje full-text) a počítání počtu záznamů v tabulce. Nicméně pro to aby byla výkonost enginu MyISAM výrazně vyšší než InnoDB musel by poměr čtení:zápis výrazně převyšovat ve prospěch čtení. Důvod velkého poklesu výkonu enginu MyISAM při zápisu je uzamykání celé tabulky při vkládání nového záznamu[9]. Což je také důvod proč se v obecných aplikacích postupně odstupuje od enginu MyISAM a přechází se na engine InnoDB.

Velice zajímavým prvkem systému řízení báze dat je možnost logování prováděných DDL a DML SQL dotazů. Dotazy jsou ukládaný v podobě binárních logů a jsou primárně určeny pro zrcadlení na záložní sekundární server. Vzhledem k tomu že jsou data uloženy v binární formě, tak nejsou běžně čitelná. Pro jejich čtení slouží utilita mysqlbinlog, která poskytuje data z těchto logu v textové formě. Utilita je pro čtení záznamů optimalizována a také nabízí množství volitelných parametrů. Na jejich základě je sestaven výstupní textový soubor. Nejčastěji se jedná o parametry určující název sledované databáze či omezení výstupních hodnot dle časového razítka.

MySQL používá několik formátů pro zaznamenávání dat v binárním logu. K dispozici jsou tři formáty zápisu. První a nejstarší z jich je formát *STATEMENT.* V případě zápisu v STATEMENT formátu jsou do binárního logu ukládány příkazy DML a DDL v textové podobě, tedy přesně v té podobě jak byly provedeny na MySQL serveru. Další formát je *ROW,* kde jsou jednotlivé záznamy ukládány jako binární hodnoty popisujících změnu tabulky a způsob ovlivnění jejího obsahu. Třetím formátem je formát *MIXED.* Tento formát je částečně automaticky a kombinuje oba předchozí formáty. Standartně formát MIXED používá logování ve formátu *STATEMENT,* avšak v jistých případech se automaticky přepíná na režim ROW. Automatické přepnutí probíhá při provádění operací, které mohou být pro formát STATEMNT rizikové a nekonzistentní. Nejtypičtějším příkladem může být použití tabulky obsahující sloupec, jehož hodnoty jsou určovány na základě auto inkrementu, společně v kombinaci s triggerem on update[9].

Společně se systém řízení báze dat je také k dispozici utilita mysqldump. Utilita je určena pro jednoduché a rychlé vytvoření snímku databáze či jiného databázového elementu. Chování a ovládání je velice podobné utilitě mysqlbinlog, tedy se nastavuje dle vstupních parametrů.

Jednou z velice důležitých vlastností pro tuto práci je transakčnost procesu provádění SQL příkazů. Bohužel MySQL podporuje transakce pouze nad SQL příkazy typu DML, tedy pouze na příkazy typu vytvoření,smazání, upravení záznamu. Nad příkazy typu DDL databáze MySQL nepodporuje transakce, či lépe řečeno nad tímto typem příkazů uplatňuje tzv. implicit commit. Což znamená, že ihned pro provedení jednoho DDL příkazu je příkaz označený jako transakčně potvrzený.

3.2 MariaDB

MariaDB[10] je relační databáze založená na databázi MySQL. Jedná se o větvi vývoje databáze MySQL, která vznikla z obavy ztráty licence GNU GPL při odkoupení MySQL společností Oracle. MariaDB ve verzi 5.5 je plně kompatibilní s MySQL též ve verzi 5.5. V novějších verzích může vznikat jistá nekompatibilita, avšak do současné verze MariaDB 10.1 a verze MySQL 5.7 jsou rozdíly tak minimální, že je téměř nelze nazývat nekompatibilitou.

3.3 Verzovací systémy Git a SVN

Smyslem verzovacích systémů je vytvořit dobrou podporu vývojářů pro ukládání jednotlivých verzí zdrojových kódu vytvářené aplikace a jejich následné sdílení s ostatními vývojáři. Dnešní verzovací systémy tuto funkčnost doplňují o spousty dodatků vyplývajících z požadavků a běžné praxe vývojářů. Verzovací systémy jsou tak již standartně vybaveny prostředky pro řešení kolizí, podporu větvení, tagování a dalším.

V této kapitole pojednávám o dvou nejrozšířenějších používaných verzovacích systémech. Na tyto dva systémy jsem se zaměřil z důvodů jejich značně protichůdnosti a použití zcela jiných postupů verzování zdrojových souborů.

3.3.1 Git

Je distribuovaný systém správy verzí. Byl vytvořen pro vývoj jádra linuxu v roce 2005. Postupně byl rozšiřován o další funkčnost a nyní se jedná o jeden z nejrozšířenějších verzovacích systémů. Jeho úspěch tví v jeho architektuře, která je plně decentralizovaná. Což sebou nese různé přínosy tak i negativa.

Při použití verzovacího systému git je na každém zařízení uložený částečný či kompletní repozitář. Výhoda tohoto řešení je nepotřebnost připojení k centrálnímu prvku.

GIT nepoužívá inkrementální soubory ale snímky [11], tedy každý verzovaný soubor je v jednotlivých verzích uložený samostatně, nemá žádné vazby týkající se obsahu na svého předchůdce. Tato volba velice zrychluje provádění veškerých verzovacích operací, zejména větvení. Díky tomuto kroku se kompletní podpora a správa větvení se stává daleko dostupnější a hlavně rychlejší. A tak větvení tvoří základní kámen celé „GITovské“ ideologie verzovaní zdrojových souborů.

3.3.2 SVN

Apache Subverzion (SVN) je centralizovaný systém pro správu verzí. Nahrazuje dřívější a starší CVS a snaží se vyvarovat jeho nedostatkům. SVN je často používaný nástroj pro malé projekty o jejichž implementaci se stará pouze několik vývojářů.

Naproti Gitu se na lokálních vývojové zařízení nekopíruje kompletní repozitář, avšak pouze kopie aktuálně stažené verze. Repozitář je uloženy na centrálním bodu odkud je dostupný. Tento přístup sebou nese výhodu v podobně malých požadavků na velikost paměti na lokálních zařízeních. Avšak nese sebou nevýhodu v podobě nutnosti stálého připojení k centrálnímu bodu. Což v praxi znamená stále připojení k serveru poskytující kompletní centrální repozitář.

Přesto že se může zdát že výhody verzovacího systému GIT značně přesahují jeho nedostatky, je systém SVN stále vyvíjený a podporovaný spousty vývojáři, zejména pro svoji jednoduchost.

3.4 Python

Python je multiplatformní skriptovací jazyk. Vznikl v roce 1991. Aktuální verze jazyka je 3.5. Jazyk je hybridní a tak podporuje různá paradigmata, zejména pak objektově orientovaný či funkcionální přístup. Je vyvíjený jako open source projekt. Python je dynamicky interpretovaný jazyk.

Python má vynikající vyjadřovací schopnosti. Kód programu je ve srovnání s jinými jazyky krátký a dobře čitelný. Což je jeden z důvodů proč je Python využíván pro výukové potřeby. A tak se Python často stává bránou k vývoji software začátečnických programátorů.

Python je považován za velice univerzální jazyk. Neboť lze na tomto jazyku, díky široké podpoře modulů, postavit téměř jakoukoliv aplikaci. Avšak hlavním zaměřením Pythonu je zpracovávání velké množství dat pro které byl určen. Proto jeho hlavní použití bývají informační systémy.

Python je dobře optimalizovaný pro výkon. Jeho optimalizace spočívá v tom že Python je napsán v jazyce C. A také některé kritické knihovny jsou napsány v tomto jazyce. Díky tomuto způsobu implementace je jazyk Python znatelně rychlejší ve většině aplikaci než srovnatelné PHP.

4 Návrh řešení

V této kapitole uvedu návrh řešení zakládající na datech zjištění v kapitole 2. Kapitola obsahuje popis mého způsobu řešení a návrhu aplikace.

4.1 Řešení problematiky

Jak bylo probráno v kapitole 2, ve verzování databázových systémů, existuje spousty problematických částí. Pro to aby jsem problémy odstranil či aspoň redukoval, tak jsem se rozhodl navrhnout aplikaci s klíčovými prvky, které nyní podrobněji popíšu.

Bodově jsem jsi vytyčil během návrhu tyto cíle:

- Výsledná aplikace nebude závislá na konkrétním verzovacím systému.
- Verzování bude probíhat na základě obecného principu importu a exportu databáze.
- Exportovaná data budou v podobě inkrementálních změn.
- Exporty budou vloženy pod správu verzovacího systému.
- Komunikace z vývojářem pouze na úrovni příkazů pro aplikaci.
- Minimální kontakt vývojáře z SQL kódem.
- Podpora větvení při vývoje databáze.
- Vyřešení kolizních situací, nastíněných v kapitole 2.

Návrh aplikace zakládá na použití obecných verzovacích systému, které jsou nejlépe připraveny na práci ze soubory textového typu. Proto navrhovaná aplikace bude pracovat na obecném principu importu dat do databáze a exportu do textových souborů, které budou obsahovat sekvenci SQL příkazů. Díky této volbě budou moci být exportované soubory vloženy pod správu jakéhokoliv verzovacího systému. Čímž je docíleno nezávislosti na verzovacím systému.

Avšak systém založený na principu exportu snímků (dumpů) databáze a jejich následném importů sebou nese značný nedostatek, a to je rychlost přechodu z jedné verze databáze k druhé. Rychlost přechodu je pomalá z důvodu provádění nadměrného množství SQL příkazů na databázovém serveru, protože při přechodu na novější verzi musí být celá databáze vymazána a následně naimportována. Některé aplikace které jsem uvedl v kapitole 2 tento problém obsahovaly ale neřešili jej, avšak některé se snažily tuto problematiku řešit. Mezi aplikace které tento problém řešily byl PHP framework Lavavel, který využíval tzv. inkrementů neboli migrací. Migrace, či blíže metoda *up* v dané migraci, je právě onen inkrement sestávající se ze sekvence příkazů pro databázi psaný v SQL/PHP jazyce. Na myšlence inkrementálních sekvencí SQL příkazů [15] bude také postavena navrhovaná aplikace. Tedy pod pojmem export již nebude rozuměno vytvoření snímku celé databáze ale vytvoření sekvence SQL příkazů, které budou reprezentovat inkrement. Následně při importu se budou spouštět pouze příkazy jedno či více inkrementů, které budou nutné k přechodu lokální databáze k nejaktuálnější verzi. A tak se při importu, tedy přechodu na novější verzi, bude spouštět znatelně menší množství databázových příkazů než v případě importu založeném na vymazáním všech záznamů a následném kompletním importu, což velice přispívá k rychlosti vykonání celé operace.

Dalším cílem vyplývající ze zadání bakalářské práce je navrhnout a vytvořit aplikaci, která bude podporovat větvení vývoje databáze. Způsob začlenění možnosti větvení do této aplikace jsem dlouze uvažoval. Konečným rozhodnutím, které bylo právě podpořeno požadavkem na podporu větvení vývoje, byl návrh počítající z aplikací bez přímé účastni na verzování, tím i na větvení vývoje databázového obsahu. Veškeré postupy a chování související z verzováním budou primárně spočívat na verzovacím systému. Z čehož plyne jediný požadavek na aplikaci,a to ten aby aplikace verzovacímu systému zprostředkovávala a předkládala data v takové podobě, která budou snadno verzovatelná i v případě podpory větvení.

Nezavilost výsledné aplikace na použitém verzovacím systému již částečně uvozuje předchozí odstavec. Protože veškeré úkony související z verzováním budou přenechány v režii verzovacího systému. Což vytvoří prostředí pro autonomnost aplikace vůči použitému verzovacímu softwaru. Zbývající části nezávislosti bude docíleno, pomocí odstínění komunikace mezi aplikací a verzovacím systémem. Odstínění bude zakládat na neimplementování jakýchkoli akcí, které by přímo souvisely s konkrétním verzovacím systémem. Odstínění bude tak spočívat v jednostranné komunikaci aplikace s verzovacím systém. Kdy aplikace bude verzovacímu systému předkládat data k verzování, bez jakékoliv bližší či vzájemné komunikace.

Mnoho vývojářů pro vývoj a práci nad databází používá různé grafické klienty, pro MySQL je to nejčastěji webové rozhraní phpMyAdmin. Díky grafickému rozhraní je úprava databáze prováděna zejména na pokyny myší, nikoliv zadáváním SQL příkazů. Grafické rozhraní velkou měrou přispívá k snadnosti úpravy a značně redukuje možné logické i syntaktické chyby. Mezi body sčítající cíle při návrhu aplikace je také požadavek na minimální kontakt vývojáře s SQL kódem. Tento požadavek grafická rozhraní přímo reflektují. Navrhovaná aplikace se bude zaměřovat i na podporu vývoje databáze v těchto grafických rozhraních. Tím jsi aplikace dává za řešení problematiku, která vzniká ve frameworku Laravel, kde je nutné „bezmyšlenkovité“ opisování pseudo SQL příkazů do jednotlivých migrací.

V návrhu aplikace je také myšleno na sadu vzorových příkladů. Návrh aplikace se bude primárně zaobírat tím, aby vyřešil tyto kolize, které byly na běžném systému neřešitelné.

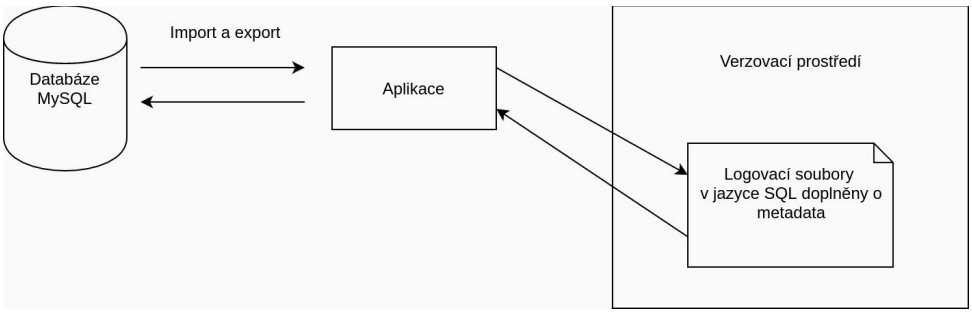
Řešení konfliktů na vrstvě verzovacích systému, je u souborů vytvořených na nákladě snímků databáze nepřehledné. Často je řešení konfliktů téměř nemožné, protože je velice těžké se zorientovat ve velkém množství po sobě jdoucích SQL příkazů. Proto je nutné při návrhu aplikace myslet také na tento poznatek, který vede na efektivní a přehledné řazení verzovaných souborů. Např. Systém založený na řazení či strukturování velkého množství SQL textu. Strukturování lze dosáhnout odsazováním a používáním komentářů, které nám budou odlišovat jednotlivé sekce SQL kódu.

Problémem zběžně vytyčených cílů byl jistý rozpor mezi požadavkem na inkrementální změny a požadavkem na nepředkládání přímého SQL kódu vývojáři. Druhý požadavek (na nepředkládání přímého SQL kódu vývojáři) počítá s využitím různých grafických rozhraní, které umožňují pohodlně pracovat s databázovou strukturou. Tímto však přicházíme o možnost zjištění toho jakým způsobem, který by byl nejlépe zapsaný způsobem SQL příkazů, proběhla změna DML(data manipulation) či DDL(data definition) změna v databázi. Tím že o prováděné příkazy přicházíme, neboť SQL příkazy se tvoří a provádí na pozadí GUI, docházíme k nutnosti požadavku na vývojáře o opsání SQL příkazů, které chce začlenit jako inkrementální změnu. Bohužel opisování SQL příkazů je jeden z bodů, kterému se chci ve vytvořené aplikaci vyvarovat. Problém má v databází MySQL řešení. Jedná se o binární logy. Tyto logy jsou primárně určené pro zrcadlení dat na sekundární (záložní) MySQL server. Obsahují binárně uložená data o spuštěných DML a DDL příkazech nad databází. Při správném nastavení a použití utility mysqlbinlog jsou binární logy čitelné i v podobě textové včetně přímých provedených SQL příkazů, v seznamu dle pořadí vykonání.

Čtení dat z binárních logů je způsob, jak zajistit možnost práce v jakémkoliv prostředí. Neboť do binárních logů jsou ukládány data přímo systémem řízení báze dat. Data z binárních logů mohou být dokonce za použití aplikace mysqlbinlog filtrována dle data vytvoření. Tímto se dostáváme k tomu, že při správném nastavení aplikace mysqlbinlog jsou výstupní data v podstatě námi potřebné inkrementální příkazy.

4.2 Návrh aplikace

Aplikaci jsem navrhl jako prostředníka mezi databází a verzovacím systémem. Aplikace bude tímto zprostředkovávat jejich vzájemnou nezávislost a nepotřebnost přímé komunikace. Aplikace bude tedy tvořit jakýsi most mezi těmito vzájemně standartně nekomunikujícími prvky, viz obrázek 5.



Obrázek 5: Architektura komunikace: MySQL - aplikace - Verzovací prostředí

Komunikace aplikace s databázovou vrstvou bude probíhat ve dvou kanálech. První část komunikace bude obstarávat připojení na MySQL databázi. Na tomto kanále budou databázi předkládaný SQL příkazy k provedení. Bude se jednat o řídící příkazy nutné pro funkci aplikace a zejména na tomto kanále budou zpracovávány příkazy týkající se importu dat do konkrétní databáze. Druhý kanál bude jednosměrný, zde se budou zpracovávat data uložená v binárním logu. Data budou aplikaci poskytnuta pomocí utility mysqlbinlog, následně budou v aplikaci data roztřízena do vnitřních struktur a posléze budou připravena k zápisu do logů aplikace, které budou uloženy ve složce, která je pod správou verzovacího systému

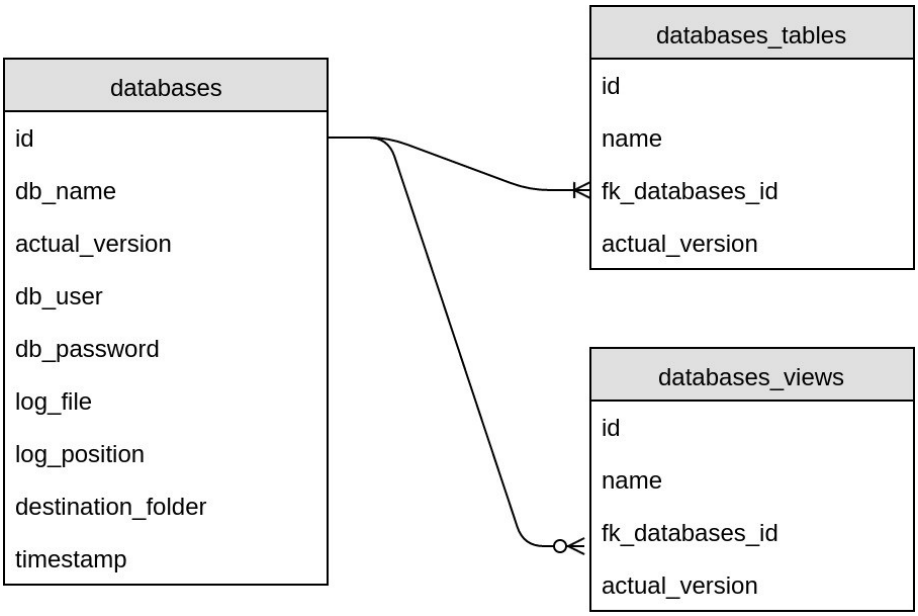
Komunikace s verzovacím systémem nebude oboustranná, dalo by se říct že minimální. Verzovací systém vůbec nebude vědět od tom, že tyto data jsou specifická pro verzování a tak se k nim bude chovat zcela běžně, což je můj požadavek. Aplikace bude verzovacímu systému předkládat textové soubory, kde každý soubor bude reprezentovat nově vytvořenou verzi databáze. Jeden soubor bude tedy obsahovat inkrementální SQL příkazy, které bude reprezentovat přechod mezi verzemi. Soubor bude obsahovat samotné SQL příkazy definované pro změnu verze a také bude obsahovat členěná meta data v podobě komentářů, které budou zajišťovat strukturovanost a bližší informace o dané verzi. Struktura jednotlivých verzí je následující.

```
/*
Start version
version: xxx
date: xxx
author: xxx
*/
SQL příkazy verze
/*
End version
*/
```

Aplikace jsi bude veškeré informace o verzovaných databázích udržovat ve své databázi. Bude se jednat o databázi, která bude sčítat několik entit. Dle ER-diagramu (obrázek 6) se jedná o entity: *databases*, *databases_tables*, *databases_views*.

Kde entita *databases* bude uschovávat informace o verzované databázi. Jedná se o jméno databáze, které bude využíváno jako slovní identifikátor verzované databáze. Dále přihlašovací jméno a heslo pro uživatele, který bude mít nad databází plná práva. Entita bude obsahovat i informace o binárních logovacích souborech, jde o jméno posledního logovacího souboru a pozici v souboru, na které bylo naposledy čteno. Tím pozice označuje začátek pro čtení další inkrementální změny. Součástí entity bude i hodnota aktuálně importované verze v rámci aplikace. V neposlední řadě entita obsahuje i cestu k cílové složce, kam se budou data (logovací soubory aplikace) ukládat.

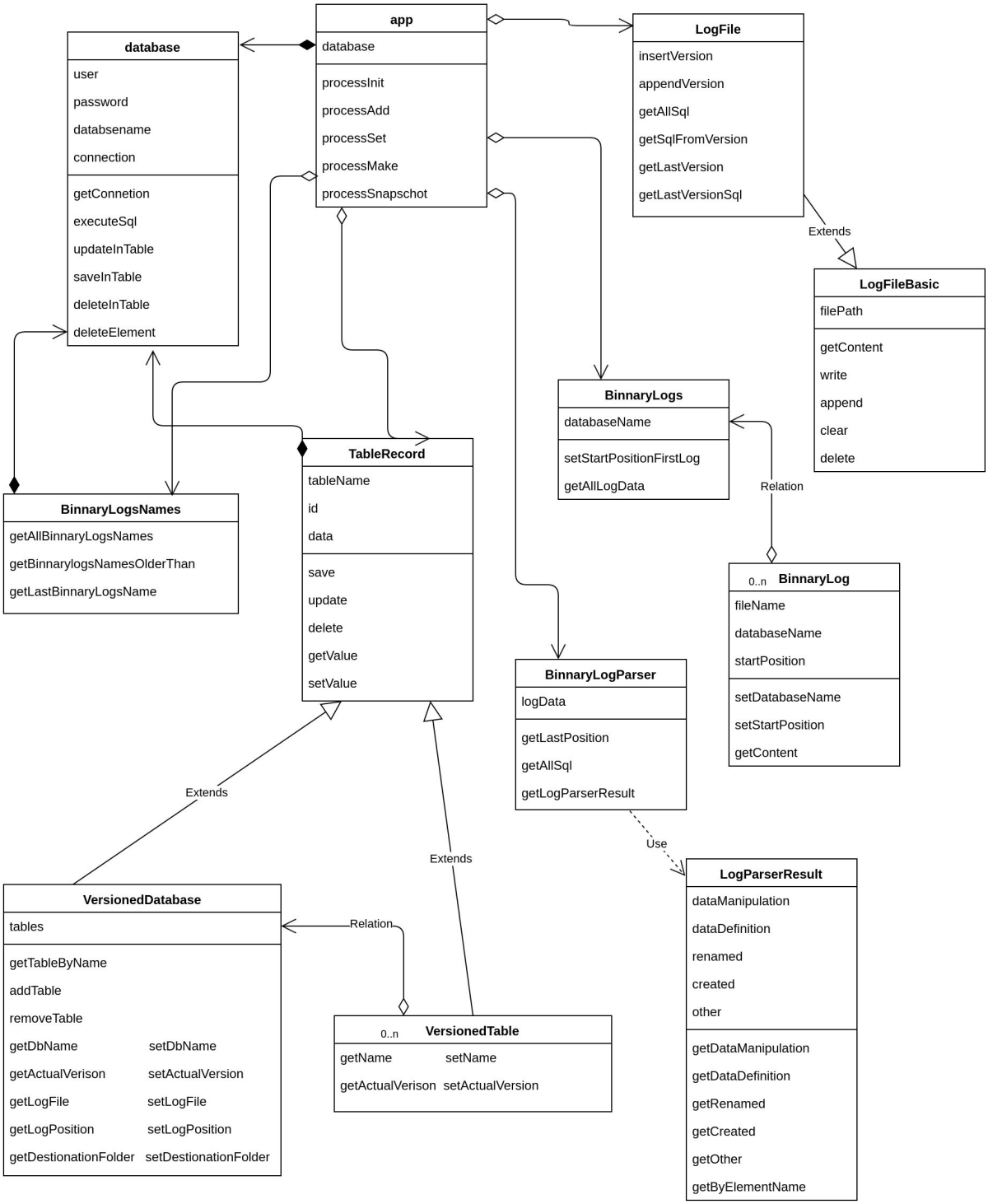
Entity *databases_tables* a *databases_views* budou reprezentovat tabulky a pohledy verzované databáze. Nejpodstatnější informací o tabulkách, respektive pohledech, bude jejich jméno a aktuální verze ve které jsou importovány v systému řízení báze dat. Verze se však bude vždy schodovat s verzí importované databáze.



Obrázek 6: ER-diagram, návrh databáze aplikace

Data uloženy v této malé databázi budou představovat popis verzované databáze. V jaké verzi je momentálně naimportovaná. Jaká je pozice ukazatele v binárních logovacích souborech. Kam aplikační logovací soubory bude aplikace ukládat. Díky těmto meta datům se může aplikace rozhodovat při řešení různých kolizí a nekonzistencí, a také se bude moci rozhodovat v případě řešení kolizí plynoucí z sady vzorových příkladů.

Aplikaci jsem se rozhodl implementovat v jazyce Python ve verzi 3.5 s primárním zaměřením na operační systém Linux. Volbu jazyka jsem zvolil v důvodu jeho multi-platformnosti a jistým výhodám jazyka Python pro zpracovávání velkého množství dat v textové podobě. Volba jazyka byla také podpořena tím že aplikaci jsem měl navrženou pro objektově orientovaný přístup, obrázek 7. Python je hybridní jazyk, který podporuje i mnou požadované objektově orientované paradigma.



Obrázek 7: Návrh objektové struktury aplikace - diagram tříd

Komunikaci aplikace z vývojářem jsem navrhl nejednodušším a nejčitelnějším způsobem. Aplikace bude terminálová. Rozhraní aplikace bude obsluhováno přes terminál (příkazovou řádku) . V tomto použití aplikace nemá význam dělat use-case diagramy, neboť účastníci diagramu by byly pouze dva. Samotná aplikace a vývojář obsluhující aplikaci. Proto komunikaci mezi aplikaci a vývojářem navrhnu výčtem dostupných funkcí, které bude aplikace nabízet:

Inicializační metoda – funkce pro inicializaci aplikace v rámci lokálního běhu. Nainicializuje potřebné nastavení aplikace a také vytvoří nutné databázové struktury.

Metoda pro přidání databáze do verzování – funkce vloží záznamy do databáze aplikace a také vytvoří první logovací soubor obsahující aktuální snímek databáze.

Metoda pro přidáni již verzované databáze – funkce pro vložení databáze do lokálního verzování. Databáze je naimportovaná z existujících logovacích souboru, které jsou dostupné ve verzovacím prostředí.

Metoda pro vytvoření nové verze – aplikace zjistí inkrement z binárních logu a následně jej převede a zapíše do svých logovacích souboru.

Metoda pro import nových verzí – funkčnost, která na základě lokální verze databáze a poslední verze logovacího souboru rozhodne zdali, jsou z dispozici nějaká data určená k importu. Případně tyto data (inkrementy) importuje do databáze.

Metoda pro vytvoření snímku databáze – funkce vytvoří snímek (snapshot) celé lokální databáze do jednoho souboru. Například pro vložení databáze na jiný server, který není svázaný s vývojem a tedy i s touto aplikací.

5 Implementace

V kapitole Implementace se zaměřím na implementační aspekty, které provázely vývoj aplikace. Nastíním některé nejdůležitější části implementace a také rozeberu problémy, které bylo nutné při implementaci řešit.

5.1 Způsob implementace

Způsob kterým budu aplikaci implementovat jsem založil na ideii unit testů. Nejprve tedy určím a přesněji specifikuji rozhraní jednotlivých tříd, které následně implementuji a posléze otestuji zdali splňují definované rozhraní a jeho chování.

Bohužel jsem v rámci unit testů použil pouze jejich myšlenku (jednotkového testování), praktické automatické testy jsem neimplementoval. Důvod byl zcela prostý. Unit testy nad aplikacemi používající více zdrojů, zejména zdroje databázové, jsou částečně hůře implementovatelné, proto jsem myšlenku unit testů přetvořil na následující postup. Před implementací třídy jsi definují její parametry, chování, rozhraní. Na základě tohoto popisu jednotlivých třídy, jsi definuji, kterými manuálními postupy budu jednotlivé metody třídy testovat. Po té metody třídy a vnitřní implementaci vytvořím. Následně započnu s jednotkovým testování, kde testuji manuálně jednotlivé metody (rozhraní) třídy dle mnou definovaných postupů a porovnávám s očekávanými výsledky dle specifikace metody.

5.2 Detaily vývoje

Součásti implementace aplikace bylo vyřešení transakčnosti jednotlivých funkcí, které aplikace provádí. Transakčnost byla zajištěna jak na úrovni databáze tak i na úrovni logovacích souborů.

Na databázové vrstvě bylo využito samotné vlastnosti transakčnosti databáze. Databáze MySQL je však schopna transakce provádět pouze nad příkazy typu DML. Zajištění transakčnosti příkazů DDL je docílil různými způsoby. U nekritických operací, tedy operací které nejsou často prováděny, jsem zvolil postup ukládání stavu databáze před prováděním DDL příkazů. Tedy před každou sekcí obsahující DDL příkazy jsi aplikace na dočasnou dobu uloží aktuální stav databáze a v případě chyby v provádění úseku kódu obsahující DDL příkazy tento stav využije a databázi na základě něj obnoví. Způsob vytvoření transakčního chování nelze označit za plnohodnotnou transakci neboť může vzniknout stav, kdy vznikne havarie v úseku kódu obsahující DDL a následně se nepodaří databázi obnovit ze snímku databáze. Avšak riziko vzniku této chyby je zanedbatelné, neboť stav (snímek) databáze je vytvořený MySQL systémem řízení báze dat, a tedy by měl být i tím stejným systémem řízení báze dat úspěšně zpracovaný. Riziko je ještě umenšeno tím že logovací soubory aplikace jsou umístěny ve verzovacím prostředí, které také využívá transakčnosti. A proto v případě kolize je možné databázi obnovit z těchto logů.

V případě kritických funkcí aplikace – zejména pak u funkce *set*, je nutné brát zřetel na čas provádění takovéhoto postupu. Z důvodu že tato metoda bude využívána asi nejčastěji, jsem zvolil jiný způsob zajištěni pseudo transakčnosti. Již se před operací nezjišťuje stávající stav databáze. DDL příkazy se provádějí bez jakýchkoliv předchozích příprav. V případě havarie je databáze obnovena z kombinace logovacích souborů aplikace (použijí se jen starší verze, nikoliv verze které se nyní importovaly) a binárních logů databáze.

Transakčnost souborů. Transakčnosti souborů, všech operacích aplikace, bylo dosaženo stejným principem. Princip zakládá na stejné myšlence jako u databáze. Tedy před jakoukoliv prací s jedním souborem je jeho obsah dočasně uložen do paměti a v případě havárie jsou do souboru zapsána (obnovena) data právě z paměti. Tento způsob nijak výrazně aplikaci při běžném použití neomezuje, neboť práce ze soubory není tak časově náročná jako práce z databází. Také na zajištěni

transakčnosti nad aplikačními logy není nutné vytvářet nijak komplikovaný mechanismus, který řeší i detailní případy havárie. Neboť v případě ojedinělé havarie, kterou tento způsob nebude schopen odstranit se lze ještě spolehnout na verzovací prostředí, které transakčnost a případnou historii souborů poskytuje.

Pro práci aplikace bylo nutné implementovat i funkčnost podporující komunikaci aplikace s terminálem operačního systému. Zpracování parametrů bylo zajištěno pomocí modulu *argparse [14]*. Dále byly implementovány metody, které jsou součástí třídy *TerminalCommand*, pro spouštění příkazů v terminálu. Jednalo se zejména o spouštění aplikací mysqlbinlog a mysqldump, které mají terminálové rozhraní. Komunikaci s terminálem na pozadí obstarává aplikace s dopomocí modulu *subprocess [14]*.

Důležitým bodem implementace bylo zpracování binární logů MySQL databáze. Binární logy ve formátu STATEMENT mají následující podobu, pří výpisu pomocí aplikace mysqlbinlog.

```
BEGIN
/*!*/;
# at 2086
# at 2118
#170502  7:17:41 server id 1  end_log_pos 2118 CRC32 0xf86cd64a      Intvar
SET INSERT_ID=2/*!*/;
#170502  7:17:41 server id 1  end_log_pos 2412 CRC32 0x18b3e222     Query
        thread_id=36    exec_time=0     error_code=0
use `myDatabase`/*!*/;
SET TIMESTAMP=1493702261/*!*/;
INSERT INTO `articles` (`id`, `title`, `slug`, `perex`, `content`, `date_publish`, `active`)
VALUES (NULL, 'Nový článek', 'novy-clanek', 'Perex k novému článku', 'Obsah nového
článku', '2017-05-03', '1')
/*!*/;
# at 2412
#170502  7:17:41 server id 1  end_log_pos 2443 CRC32 0xf1e54dd2      Xid = 277
COMMIT/*!*/;
```

Proto je nutné tento výstup předzpracovat. Pro zpracování binárních logů jsem použil zpracování textových dat na základě regulárních výrazů. Vytvořené regulární výrazy však dle svého rozsahu naznačovaly, že jejich běh nebude nijak snadný z pohledu rychlosti provádění. Z tohoto důvodu jsem zvolil nejprve předzpracování na základě klíčových slov, které v textové podobě výpisu binárních logů vyhledám. Jedná se například o slova typu *insert*, *update*,… Hledaní slova v řetězci není tak časově náročné jako porovnávání řetězce vůči regulárnímu výrazu. A následně po předzpracování a identifikace slova v řádku se nad daným řádkem spustí příslušný regulární výraz, který je podstatně jednodušší a vytvořený pouze pro daný SQL příkaz. Ten odhalí veškerou syntax, editované objekty a jejich názvy. Zde je uvedený jednoduchý regulární výraz pro zpracování SQL příkazu *insert*.

```
insert (low_priority |delayed | high_priority )?(ignore )?(into )?(?P<dbname>[\w`\'\-"]*\.)?
(?P<name>[\w`\'\-"]*).+
```

Pro spuštění regulárních výrazů používám balíček jazyka Python s názvem *re*. Jelikož klíčová slova v MySQL nejsou case-sensitive [9], proto je nutné při spouštění a vyhledávání regulárních výrazů použít přepínač *re.IGNORECASE*. Potom příkaz pro aplikování a vyhledávání regulárního výrazu nad textem pro jazyk Python je následující.

```
re.search(regularExpresion, sqlQuery, re.IGNORECASE)
```

Pro jednoduchost používání databázové vrstvy v kódu aplikace jsem využil jednoduché implementace objektově relačního mapování. Kdy se tabulky databáze mapují na objekty v aplikaci. Objekty následně mají atributy odpovídající sloupcům tabulky. Atributy nejsou však dostupné přes veřejné rozhraní třídy. Atributy jsou označeny na jako privátní (v jazyce Python označovány dvěma podtržítky – *self.__name)*. Na přístup a manipulaci s atributy slouží tzv. g*ettery a settery,* což jsou veřejné metody pro získání a zapsání hodnoty. Pro implementaci mapování jsem naprogramoval třídu *DatabaseTableRecord*, která je obecným rozhraním pro objektově relační mapování. Obsahuje metody *save* a *delete*. Kde metoda save provede uložení aktuálního stavu třídy do databáze dle mapování a metoda *delete* smaže záznam. Od třídy následně dědí třídy, které mají nastavené konkrétní mapování. V této aplikaci se jedná o třídy *VersionerDatabase* pro tabulku *ver_databases* a *VersionedTable* pro tabulku *ver_databases_tables*. Třídy pak obsahují zmiňované *gettery a settery*.

Aplikace po jakémkoliv připojení na databázi spouští několik konfiguračních příkazů týkající se nastavení komunikace z databází. Jedná se o příkaz „SET sql_log_bin=0“, pro to aby veškeré změny provedené aplikací nebyly zapisovány do binárních logů a posléze i do logovacích souborů aplikace, a příkaz „SET FOREIGN_KEY_CHECKS = 0“, pro to aby SŘBD neprováděl kontrolu cizích klíčů.

5.3 Implementační změny

Po vyhotovený základní funkčnosti aplikace, tedy po vyhotovení aplikace ve verzi nazvané alfa. Byly zřetelné některé omezení v návrhu aplikace, na které nebylo z počátku myšleno. Jedním z těchto bodů byla nutnost změny konceptu logovacích souborů. V návrhu bylo počítáno s tím že každý logovací soubor, který aplikace vytvoří bude reprezentovat jeden inkrement, přechod na novější verzi databáze, který bude obsahovat SQL příkazy pro provedení přechodu. Avšak po hlubším zamyšlení byly logovací soubory přepracovány do stavu, kdy každý logovací soubor reprezentuje databázový element, nejčastěji tabulku. Takovýto logovací soubor se bude skládat z jednotlivých verzí tohoto elementu. Každá verze bude obsahovat, stejně jako tomu bylo v návrhu, SQL příkazy pro přechod z nižší verze k novější. Příkazy se však budou vztahovat pouze k danému DB elementu. Ostatní logy netýkající se konkrétních databázových elementu budou logovány do separátního logu, který pro tento účel bude určen. Jednotlivé logy budou odděleny specifickými komentáři obsahující metadata o verzi. Komentáře budou téměř identické s těmi, které byly navrhženy v návrhu. Díky tomuto kroku se docílí přehlednosti logů, protože každý logovací soubor bude patřit k jednomu logickému celku. A tak při případném zásahu, např. při řešení kolize, bude vývojář jen díky souboru ve kterém se bude pohybovat, obeznámen s elementem nad kterým pracuje, čímž bude orientace v kódu daleko jednodušší než předchozí navržené řešení. Jednotlivé logovací soubory se budou pojmenovávat podle názvu elementu který logují. Pro logovací soubor týkající se ostatních SQL dat jsem vyhradil jméno *otherLog,* čímž jsem znemožnil vytvoření tabulky s tímto jménem. Jako koncovku logovacích souborů jsem zvolil koncovku *.versql*. Koncovku jsem zvolil z důvodu, aby bylo hned z názvu jasné, že tento soubor je pod správou verzovací aplikace a také že tento soubor obsahuje textová SQL data. Tedy celý název souboru pro tabulku *articles* bude *articles.versql.* Obsah jednotlivých souborů je řazen následovně.

```
/*
Start version
version: xxx
date: xxx
author: xxx
*/
Sql příkazy a dotazy dané verze
/*
End version
*/

/*
Start version
version: xxx
date: xxx
author: xxx
*/
Sql příkazy a dotazy dané verze
/*
End version
*/
```

Po vyhotovení beta verze a následném prvotním testování byly vydefinovány nedostatky, které se odrazily v nutnosti implementovat několik dalších funkcí aplikace.

Jednalo se o:

- Plný import databáze – funkčnost aplikace bude mít za úkol odstranit veškeré stávající záznamy v databázi, celou databázi vyčistit, a následně naimportovat veškeré verzované logy s SQL příkazy. Tímto krokem bude dosaženo synchronizace obsahu databáze a aplikačních logovacích souborů. Také při spuštění funkce budou zapomenuty veškeré binární logy, které byly dosud provedeny. Pod pojmem zapomenutí je blíže chápáno posunutí ukazatele. Čímž budou starší data z binárních logů pro aplikaci znepřístupněna. Funkčnost byla dodělána ze dvou důvodu: 1) Častá potřeba vymazat změny v databázi, které se prováděly z důvodů testování. 2) Reflektovaní změn týkajících se větvení na verzované vrstvě, bližší popis je uvedený v následující kapitole 6.
- Plný export databáze – Princip této funkce je pročistit veškeré aplikační logy od nadbytečných dat. Metoda aplikace promaže logy a následně do nich vloží SQL příkazy (snímky jednotlivých elementů) v poslední verzi. Díky tomuto kroku budou logy vyčištěny od redundantních dat. Jedná se o mrtvé úseky příkazů jako například *insert* záznamu a jeho následný *delete.*
- Slučování verzí databáze – Častým případem, který je nejčastěji používaný ve verzovacím prostředí git, je vytvoření několika commitů (a také verzí databáze), a až po té následný export (*push*) do veřejného repozitáře. Avšak z důvodů vytvoření několik verzí databáze je log poměrně nepřehledný. Tato metoda sloučí logy od zadané verze. Krok docílí zpřehlednění logovacího souboru a umožní vývojáři čitelnější zásah do aplikačních logů.

6 Testování

Důležitým bodem vývoje aplikace bylo otestování její funkčnosti a následné zhodnocení navržené workflow. Testování jsou rozdělil do několika části. První část testování jsem již rozebíral v předcházející kapitole. Toto testování zakládalo na myšlence jednotkového testování. Další části bylo otestování vzájemné komunikace jednotlivých komponent (tříd) aplikace. A posledním bodem bylo testování v týmu zaměřené na týmovou spolupráci, využití v kombinaci s verzovacími systémy, a zdali aplikace splňuje požadavky, které jsem jsi určil v kapitole 4 -Návrh. V rámci všech testů jsem jsi definoval, že pokud bude objevena v některé fázi testování chyba, tak veškeré testy budou přerušeny, chyba opravena. Následně se bude pokračovat zopakováním naplánovaných testů od začátku.

Testování komunikace jednotlivých komponent nebylo prováděno automatizovaně. Veškeré testování bylo prováděno manuálně. Před každým manuálním testem jsem určil vstupní hodnoty testu a také počáteční stav aplikace. Následně probíhal samotný manuální test. Výsledky testu byly porovnány s očekávanými hodnotami a výstupy. Na základě těchto hodnot jsem pokračoval k další iteraci testování.

Pro poslední část testování jsem určil zběžný testovací plán, který se skládal z několika fází testování. První fáze se zaměřovala na lineární vývoj dvou vývojářů. Na této fázi bylo otestováno základní funkčnosti a zejména chování inkrementálních změn. Na další fázi jsem se během testů zaměřoval na již dříve v této práci definované kolize (sadu vzorových příkladů) , které mohou nastat. Jedná se o kolize ústící ve vzájemné přepisování databázových verzí. Po ukončení předchozí fáze, následovalo testování funkčnosti v rámci podpory větvení databázového schématu. Tato poslední část testování probíhala ve dvou rovinách. První předběžná úroveň probíhala pouze interně, tedy aplikaci a výše zmíněné oblasti jsem testoval osobně. Pokračoval jsem navazující úrovní, kde jednotlivé postupy byly testování v rámci vývojového týmu.

Pro interního testování jsem použil verzovací systém git. Emulování týmové spolupráce, tedy vytvoření dvou nezávislých vývojářů jsem docílil pomocí dvou kopii repozitáře nezávisle na sobě umístěných v adresářové struktuře v kombinaci se dvěma databázemi běžící nad jedním SŘBD.

V průběhu interního testování aplikace vykazovala je drobné logické chyby, které byly jednoduše odstraněny. Avšak po započetí interních testů týkající se podpory větvení vývojového schéma databáze, byla odhalena chyba v nedostatečně propracovaném návrhu aplikace. Chyba se týkala nemožnosti reflektování změny větve ve verzovaném prostředí vůči lokální databázi, tedy po změně větve ve verzovácím systému nebylo součástí lokální databáze schéma, které by odpovídalo obsahu v aktuální větvi. Dosažení obsahu lokální databáze, tak aby databáze reflektovala aktuální větev, nebylo možné, neboť při přepnutí vývojové větve nemá aplikace k dispozici SQL příkazy vedoucí k změně lokálního databázového schématu dle současné větve ve verzovaném prostředí. Chyba vedla k analýze a návrhu s cílem přizpůsobení současné implementace požadavkům na větvení. Pod delším zkoumání možnosti začlenění do stávající aplikace jsem došel k závěru že větvení samo o sobě porušuje myšlenku inkrementálních sekvencí SQL příkazů. Proto bude nutné v případě přepnutí větve celou databázi přizpůsobit. Což v praxi znamená celou databázi naimportovat, dle aktuální větve. Řešení není nijak optimální, avšak jiná řešení vedou na komplikované řešení, která by přesahovala rozsah této práce. S tímto cílem vznikla také funkce aplikace s názvem *forceSet*.

V návrhu aplikace bylo počítáno s tím, že vytvořená aplikace nebude schopna redukovat tzv. mrtvý SQL kód, tedy např. *create table* a následný *delete table*. Avšak společně s vytvořením funkce *forceSet*, byla nasnadě i funkce *forceMake*. Funkce měla za úkol vytvoření snímku aktuální lokální databáze, respektive jednotlivých elementu. Snímky elementů jsou pak vloženy jako poslední verze do aplikačních logovacích souborů, s tím že veškeré předchozí verze jsou smazány. Díky takovéto funkčnosti bude odstraněný veškerý mrtvý kód.

Po ukončení interního testování následovalo testování v rámci vývojového týmu nad ostrými daty. Aplikaci jsem testoval v 7 členném týmu ve firmě Izon s.r.o., při vývoji jedno z jejich produktů na kterém pracovali. Tedy aplikace byla testována v prostředí pro které byla vytvořena. Testování

bylo opět rozděleno do tří částí: testování inkrementálních změn, testování řešení kolizí a testování podpory větvení. Pro ostré testování jsem však zavedl jednu drobnou změnu. A to že tento způsob testování bude proveden ve dvou iteracích. S tím že první iterace bude otestování na fyzických dvou členech týmu a v druhé iteraci se do testování zapojí celý vývojářských tým pracující nad jedním projektem.

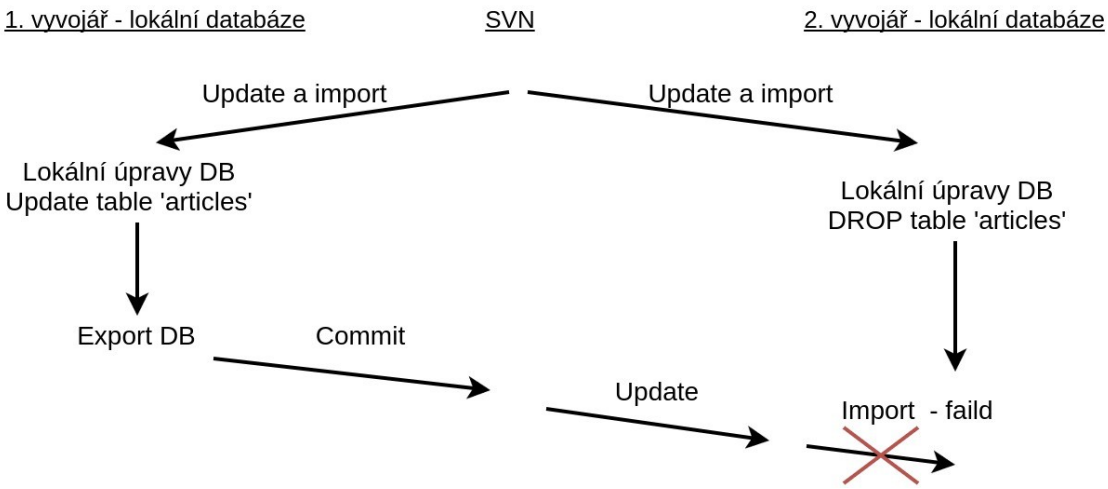
Součástí požadavků pro testování v daném týmu byla nutnost aplikaci zprovoznit a ozkoušet i na systému Windows, pro který aplikace nebyla primárně vyvíjena. Nutnost zprovoznění aplikace na systémem Windows bylo to, že někteří vývojáři z týmu systém primárně používali pro vývoj. Vzhledem k tomu že aplikace byla napsána v jazyce Python, tak nebyla nutnost nijak výrazného zásahu do kódu aplikace. Pouze jsem setkával s problémy týkající se kódování textu a reprezentace cest v adresářové struktuře. Aplikace byla proto poměrně rychle odladěna i pro operační systém Windows.

Velkým pozitivem bylo využívání jiného verzovacího systému. Vývojový tým pro svou práci využívá verzovacího systému SVN. Proto bude otestování komplexnější a případně zřetelněji odhalí problémy, které by se vázaly na konkrétní verzovací prostředí. Bohužel negativem bylo to že vývojový tým téměř vůbec nepoužívá větvení pří vývoji aplikací. Proto bude testování větvení nad ostrými daty spíše okrajové. Pouze pro úplnost bych poznamenal: Vývojový tým do teď využíval systému verzování databáze založeném na importu a exportu celé databáze.

Testování inkrementálních změn aplikace v rámci vývojového týmu proběhlo až nad očekávání úspěšně. Veškerý vývoj, který byl bez jakéhokoliv cíleného konfliktu probíhal bezproblémově a aplikace nevykazovala chybové stavy. Pouze pohled vývojového týmu byl zpočátku mírně skeptický, protože se aplikace tvářila jako klasická černá skříňka (*black box*). Což tvořilo jistý počáteční pocit „co a jak se v aplikaci děje“ a tím pádem i jistou nedůvěru.

V rámci testování kolizí při verzování databáze jsem se zaměřil na testováni již zmíněných nejběžnějších kolizí které nastávají. S těmito běžnými kolizemi se verzovací systém či aplikace vypořádala dle očekávání. Tedy v případě kolize na úrovni verzovacího systému bylo vše v režii verzovacího systému. Při testování případů ze sady kolizních případů se aplikace chovala následovně. První kolize, která byla nastíněna v kapitole 2 obrázkem 1, kdy jeden vývojář chtěl do své neexportované databáze začlenit úpravy jiného vývojáře, byla aplikací vyřešena. Vzhledem k tomu že aplikace pracuje na inkrementálních změnách tak s tímto krokem neměla žádný problém. Inkrement jiného vývojáře naimportovala do lokální databáze druhého vývojáře a změnila aktuální verzi databáze, s tím že veškeré úpravy lokálního vývojáře zůstaly beze změny. V druhé kolizi, obrázek 2, byla kolize plně řešena na úrovni verzovacího systému. Její řešení bylo podstatně přehlednější než v případě rozebíraném v kapitole 2, toto díky strukturovanosti logovacích souborů.

Bohužel aplikace nebyla schopna řešit okrajovější případy, pouze na ně upozornila a samotné řešení nechala na vývojáři. Jeden z těchto případů je ilustrovaný na obrázku 8. V této kolizi oba vývojáři pracují nad stejným databázovým elementem, konkrétně v tomto případě se jedná o tabulku *articles*. Což samo o sobě nemusí být předpokladem pro kolizi,pokud oba vývojáři vloží data do tabulky, tak se oba záznamy bezproblémově sloučí. Však v následujícím problému je tabulka *articles* vývojářem 1 upravena a následně cummitnuta. Vývojář 2 tabulku smaže, updatuje logovací soubory a chce je importovat. Aplikace však zahlásí SQL chybu neboť nemůže nalézt tabulku *articles.* Díky transakčnosti aplikace se nově importovaná verze neprovede, tedy vývojář 2 může kolizi vyřešit a následně znovu importovat. Na takovéto chování nelze pohlížet kriticky, protože i u verzovacích systému se často setkáváme z kolizemi, které nejsou řešitelné automaticky. Často se jedná o kolize na stejném principu, tedy práce nad stejným elementem, souborem.



Obrázek 8: Kolize aplikace nad DB elementem

Souhrnně testování potvrdilo správnost a cílenost návrhu,implementace aplikace. Aplikace pracuje velice dobře na úrovni inkrementálních změn databáze. Testování odhalilo jisté nedokonalosti při využití větvení vývoje databáze. Které se, i přes řešení problém v rámci začleňování do již existující částečné implementace, podařilo vyřešit dostatečně úspěšně.

7 Závěr

Cílem této práce bylo vytvoření aplikace pro podporu verzování databázových systémů, založeném na systému řízení báze dat MySQL, při vývoji aplikace. Před samotným návrhem a implementací aplikace byly prozkoumány již existující nástroje pro podporu verzování databází. Zejména bych jsem zmínil systémy phpMyVersioner a Laravel, které mnou vyvíjené aplikaci daly prvotní směr a myšlenku. V návrhu aplikace jsem reflektoval všechny jejich klady i zápory. Součástí návrhu byl také brán zřetel na podporu větvení při vývoji databáze. V rámci implementace jsem se zaměřil na použitelnost a praktičnost aplikace. Aplikace byla navrhována s cílem minimalizace závislosti na jakémkoli verzovacím systému, výsledkem je absolutní nezávislost a možnost použití jak systému Git, SVN či dalších. Aplikace byla implementována v jazyce Python 3.5 a je spustitelná jak pod operačním systémem Windows či Linux. Aplikace byla také otestována v rámci týmu pracující na ostrými daty v reálném provozu.

Jako přínos aplikace bych vyzdvihl několik aspektů, které z návrhu a následného vývoje plynou. Zejména se jedná o inkrementální změny databáze, které přinášejí rychlost zpracování a přehlednost při verzování oproti klasickým systémů. Dalším přínosným bodem je podpora větvení při vývoji databáze, kterou většina dostupných aplikací vůbec nepodporuje. Pozitivní je také snaha aplikace o přehlednost a snadnost jejího použití. Pro běžnou práci z aplikací není třeba téměř žádná hlubší znalost fungování verzovacích systémů, a také hlubší znalost SQL jazyka.

Aplikace obsahuje plně funkční základ, na kterém je možné dále stavět a zdokonalovat aplikaci. Pro zdokonalení je určitě vytvoření jednoduchého grafického rozhraní, které by snadnost obsluhy aplikace ještě více umocňovalo a dalo k dispozici i méně zkušeným uživatelům. Velmi zajímavým prvekem, se kterým je částečně počítáno v návrhu a implementaci aplikace, je možnost udržování různých databázových elementů v jiných vývojových verzích. Implementování této funkčnosti by vytvářelo jistou dynamičnost a „ohebnost“ celého vyvíjeného databázového schématu. Databázový vrstva i částečně aplikační vrstva jsou pro tento způsob práce před-připraveny.

Aplikace je uveřejněna na stránkách *github.com*[1]. Je veřejně dostupná pod licencí GNU. Aplikaci jsem označil verzí 0.1, protože se jedná o funkční základ na kterém lze aplikaci dále rozšiřovat. Nicméně i v této verzi aplikace přináší odlišný pohled na možnosti verzování databázových systémů. Jisté myšlenky upravuje a také vytváří nový způsob práce v týmu.

1 Repozitář je dostupný na adrese https://github.com/honza66/databaseVersioner

Literatura

[1] Abraham Silberschatz, Henry F. Korth, S. Sudarshan: Database System Concepts, 6th Edition, 2011, ISBN 978-0-07-352332-3

[2] K. Scott Allen, Versioning Databases – The Baseline, Ode to Code [online]. 2008 [cit. 1.5.2017] Dostupny z WWW: <http://odetocode.com/blogs/scott/archive/2008/02/01/versioning-databases-the-baseline.aspx>

[3] Laravel - Migrations [online]. 2017 [cit. 1.5.2017]. Dostupný z WWW: <https://laravel.com/docs/5.4/migrations>

[4] Symfony - Documentation [online]. 2017 [cit. 1.5.2017]. Dostupný z WWW: <http://symfony.com/doc/current/index.html>

[5] Django - Documentation [online]. 2017 [cit. 1.5.2017]. Dostupný z WWW: <https://docs.djangoproject.com/en/1.11/topics/migrations/>

[6] LiquiBase – source control for your database, [online]. 2016 [cit. 1.5.2017]. Dostupny z WWW: <http://www.liquibase.org/>

[7] Database version control, made easy, [online]. 2016 [cit. 1.5.2017]. Dostupny z WWW: <https://dbv.vizuina.com/>

[8] phpMyVersion,[online]. 2013 [cit. 1.5.2017]. Dostupny z WWW: <http://phpmyversion.sourceforge.net/>

[9] MySql – Oracle Corporation, The world‘s most popular open source database [online]. 2017 [cit. 1.5.2017]. Dostupný z WWW: <https://dev.mysql.com/>

[10] MariaDB, The Fastest Growing Open Source Database [online]. 2017 [cit. 1.5.2017]. Dostupný z WWW: <https://mariadb.com/>

[11] Git, --everything-is-local, [online]. 2017 [cit. 1.5.2017].

Dostupný z WWW: <https://git-scm.com/>

[12] SVN, Apache Subversion, [online]. 2011 [cit. 1.5.2017]. Dostupny z WWW: <http://subversion.apache.org/>

[13] Collins-Sussman, B.; Fitzpatrick, B.; Pilato, C.: Version Control with Subversion. O'Reilly Media, 2011. Dostupné z WWW: <http://svnbook.red-bean.com/>

[14] Python – Python Software,[online]. 2017 [cit. 1.5.2017]. Dostupný z WWW: <https://www.python.org/>

[15] K. Scott Allen, Versioning Databases – Change Scripts, Ode to Code [online]. 2008 [cit. 1.5.2017] Dostupny z WWW: <http://odetocode.com/blogs/scott/archive/2008/02/02/versioning-databases-change-scripts.aspx>

[16] K. Scott Allen, Versioning Databases – Views, Stored Procedures, and the Like, Ode to Code [online]. 2008 [cit. 1.5.2017] Dostupny z WWW: <http://odetocode.com/blogs/scott/archive/2008/02/02/versioning-databases-views-stored-procedures-and-the-like.aspx>

[17] Jon Loeliger, Matthew McCullough: Version Control with Git, 2nd Edition, O'Reilly Media, 2012. ISBN 978-1449316389

[18] Kriegel, A.; Trukhnov, B.: SQL Bible. Indianapolis, Indiana: Wiley Publishing, 2003, ISBN 0-7645-2584-0

[19] John F. Roddick: A survey of schema versioning issues for database systems. Information and Software Technology, Volume 37, Issue 7, 1995, pp. 383-393. ISSN 0950-5849

Seznam příloh

Příloha A

Použití aplikace

Spuštění aplikace

Pokyny pro nastavení a spuštění aplikace jsou zapsány v souboru README.txt, který je obsažený ve zdrojových kódech. Aplikace se sestává z jednoho package s názvem *databaseVersioner.* Package obsahuje zdrojové kódy pro aplikaci. Pro samotné spuštění aplikace je nutné spustit soubor *databaseVersioner.py* pomocí interpreta Python 3.5. Soubor se nachází v kořeni adresáře repozitáře.

Příklad spuštění aplikace v terminálu na operačním systému Ubuntu:

```
$ python3 databaseVersioner.py
```

Použití aplikace

Chování aplikace je určeno dle zadaných parametru. Seznam a popis jednotlivých parametru je možné získat použitím přepínače „--help“.

Kde jednotlivé parametry spouštějí následující úlohy:

--help
: Výpis nápovědy k programu.

--init
: Inicializace aplikace, používá se pouze při prvotním běhu aplikace.

--version
: Výpis aktuální verze aplikace.

--addNonExist dbName folder
: Přidá databázi s názvem *dbName (*kde *dbName* je parametr určující název databáze*)* do lokálního verzování pomocí aplikace. Předpokladem je že přidávaná databáze nebyla zatím nikým přidána do verzování, proto nejsou k dipozici aplikační logy. Parametr *folder* určuje cílovou složku, do které budou aplikační logy ukládány.

--addExist dbName folder
: Přidá databázi s názvem *dbName (*kde *dbName je* parametr určující název databáze*)* do lokálního verzování pomocí této aplikace. Tato databáze již však byla přidána (pravděpodobně jiným programátorem) do verzování pomocí aplikace a aplikační logy jsou k dispozici ve složce, která je určena parametrem *folder*. Předpokladem je, že složka je pod kontrolou verzovacího systému.

--snapshot dbName destinationFile
: Vytvoří snímek databáze. Snímek je reprezentovaný jedním souborem. Parametr *dbName* určuje název databáze a parametr *destinationFile* určuje cestu k cílovému, vytvářenému souboru.

--make dbName
: Vytvoří inkrementální změnu databáze a zapíše ji do aplikačních logů.

--set dbName

Naimportuje nové inkrementy, které nejsou naimportovány ale jsou k dipozici v aplikačních logovacích souborech.

--forceMake dbName

Pročistí všechny aplikační Logy. Vyčištění logů je založené na snímcích jednotlivých DB elementů.

--forceSet dbName

Pročistí databázi od změn. Databáze se uvede do stavu, reflektující poslední verzi databáze. Veškeré lokální změny jsou zapomenuty. Tato funkčnost se využívá pro testování. Také je nutné ji spustit po přepnutí větve ve verzovacím prostředí.

--merge dbName fromVersion

Sloučí více verzí databáze do jedné. Sloučí se verze od *fromVersion(*parametr) do poslední verze. Tím se docílí větší „čistoty“ aplikačních logů.

--databasesInfo

Vypíše základní informace o verzovaných databázích.

Příklady spuštění některých funkcí:

```
$ python3 databaseVersioner.py --help
$ python3 databaseVersioner.py --addNonExist world ./temp
$ python3 databaseVersioner.py –make world
```

WorkFlow

Běžná práce z aplikací:
Programátor který databázi založí spustí funkci *addNonExist*, tím pro aplikaci vytvoří aplikační logy a záznamy v lokální databázi. Tyto aplikační logy následně commituje. Ostatní programátoři jsi databázi lokálně zpřístupní pomocí funkce *addExist.* Všichni pracují na vývoji databáze. Při práci využívají funkci *make* pro vytváření nových verzí, které následně commitují. Či updatují aplikační logy, jejichž inkrementy jsi importují pomocí funkce *set.*

Pokud chce nějaký programátor vyzkoušet změnu databáze, která se posléze neověří, využije funkce *forceSet.*

Pro udržení „čistoty“ aplikačních logů, využívají programátoři funkce *merge*. A také některý z nich, po dokončení logického celku, použije funkci *forceMake* pro plné vyčištění logů.

V případě nutnosti importu databáze na server, který nepodporuje tuto aplikaci, je možné použít funkce *snapshot,* která vygeneruje snímek databáze v podobě SQL kódu, který je možný spustit na serveru.

Pokud se ve vývojové týmu používá větvení je nutné pro přepnutí větve ve verzovacím prostředí, spustit metodu *forceSet*.

Příloha B

Virtualizovaný operační systém

Přiložené DVD obsahuje virtualizovaný operační systém Linux (Ubuntu verze 16) pro virtuální prostředí VirtualBox s nastaveným systémem pro běh aplikace.

Systém obsahuje nainstalovaný systém řízení báze dat MariaDB ve verzi 10.0. V SŘBD je umístěna databáze *world*[2], naplněna testovacími daty, určená k testování aplikace. Systém také obsahuje nainstalovaného interpreta Python 3.5 včetně veškerých doplňků, nutných pro chod aplikace. Součástí je i verzovací systém git. Pro snadnou editaci databáze je nainstalováné grafické prostředí phpMyAdmin, které je dostupné pomocí prohlížeče na adrese 127.0.0.1/phpmyAdmin.

Na ploše jsou umístěny dvě složky. Složka *databaseVersioner* je lokální kopii repozitáře dostupného na github.com. Z této složky je aplikace spustitelná dle pokynů v příloze A. Druhá složka *logfiles* je určena pro ukládání aplikačních logů.

Systém je již nastavený pro běh aplikace. SŘBD je též nastavený včetně povolení binárního logování. Aplikace je na vituálním systému již inicializovaná a databáze *world* je již touto aplikací verzovaná.

Přístupy pro virtuální systém:

Uživatel systému (správcovský účet)
Jméno: databaseVersioner
Heslo : heslo

Uživatel databáze
Jméno: root
Heslo: heslo

Uživatel phpMyAdmin
Jméno: root
Heslo: heslo

2 Volně dostupná na url: https://downloads.mysql.com/docs/world.sql.gz

Příloha C

Obsah DVD

- Virtualizovaný operační systém s nastaveným prostředím pro běh aplikace: adresář virtual/
- Zdrojové kódy programu – snímek repozitáře: adresář databazeversioner/
- Text práce v elektronické podobě: adresář text/