Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-100241-92223

Zsolt Kiss

JDBC ovládač pre súborový systém

Bakalárska práca

Študijný program: Informatika (konverzný)
Študijný odbor: 9.2.1 Informatika
Miesto vypracovania: Ústav počítačového inžinierstva a aplikovanej informatiky, FIIT STU, Bratislava
Vedúci práce: Ing. Gabriel Szabó
Máj 2023

SEM PRÍDE ZADANIE BAKALÁRSKEJ PRÁCE

Čestne vyhlasujem, že som túto prácu vypracoval samostatne, na základe konzultácií a s použitím uvedenej literatúry.

V Bratislave 15. mája 2023 Zsolt Kiss

Poďakovanie

Týmto by som rád poďakoval vedúcemu práce, Ing. Gabrielovi Szabóovi, za konzultácie, ochotu a odbornú pomoc počas vypracovávania bakalárskej práce.

Zsolt Kiss

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Informatika (konverzný)

Autor: Zsolt Kiss
Bakalárska práca: JDBC ovládač pre súborový systém
Vedúci bakalárskej práce: Ing. Gabriel Szabó
Máj 2023

Cieľom práce je vytvorenie JDBC ovládača, ktorý pracuje nad súborovým systémom, uloží a pristupuje k údajom vo formáte, ktorý je ľudsky čitateľný. V analytickej časti sa skúmajú kľúčové triedy a rozhrania rozhrania JDBC API, skúmajú sa rôzne prístupy k práci s databázovými údajmi a porovnávajú sa populárne ľudsky čitateľné serializačné formáty. Identifikujú sa obmedzenia súvisiace s ukladaním údajov v súborovom systéme. V tejto časti sa tiež rozoberajú dve odlišné metódy spracovania príkazov SQL a diskutuje sa o spracovaní transakcií a úrovniach izolácie v rôznych databázach. V návrhovej časti sa zdôvodňujú vybrané metódy a prístupy na riešenie problémov zistených v analýze. Implementačná časť poskytuje podrobné vysvetlenie každej základnej súčasti vyvinutého JDBC ovládača spolu s prehľadom jeho fungovania. V časti venovanej overovaniu riešenia sú opísané testovacie metódy použité na vyhodnotenie funkčnosti jednotlivých komponentov ovládača. Okrem toho sa uvádza porovnanie výkonu a možností vyvinutého ovládača s ovládačom SQLite. V závere práce sú uvedené odporúčania na budúce vylepšenia ovládača s poukázaním na oblasti, v ktorých je možné vykonať ďalšie vylepšenia a optimalizácie.

Annotation

Slovak University of Technology in Bratislava

FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Degree course: Informatika (konverzný)

Author: Zsolt Kiss

Bachelor's thesis: JDBC driver for the file system

Supervisor: Ing. Gabriel Szabó

May 2023

The goal of this work is to create a JDBC driver that operates over the file system, storing and accessing data in a human-readable format. The analytical part examines the key classes and interfaces of the JDBC API, explores different approaches to working with database data, and compares popular human-readable serialization formats. The limitations associated with storing data in a file system are identified. This section also discusses two different methods for processing SQL statements and discusses transaction processing and isolation levels in different databases.The design section justifies the methods and approaches selected to address the problems identified in the analysis. The implementation section provides a detailed explanation of each basic component of the developed JDBC driver along with an overview of its operation. The solution validation section describes the test methods used to evaluate the functionality of each component of the driver. In addition, a comparison of the performance and capabilities of the developed driver with the SQLite driver is presented. The thesis concludes with recommendations for future enhancements to the driver, pointing out areas where further improvements and optimizations can be made.

Obsah

Zoznam použitých skratiek

JDBC – Java Database Connectivity

API – Application Programming Interface (Aplikačné programovacie rozhranie)

LOB – Large Object

BLOB – Binary Large Object

CLOB – Character Large Object

URL – Uniform Resource Locator

NTFS – New Technology File System

EXT - Extended File System

DBMS – Database Management System (Systém riadenia databázy)

FS – File System (Súborový systém)

HRF – Human Readable Format (Ľudsky čitateľný formát)

CSV – Comma Separated Values

JSON – Javascript Object Notation

XML – eXtensible Markup Language

XSD – XML Schema Definition

YAML – Yet Another Markup Language

REGEX – Regular Expressions (Regulárne výrazy)

ANTLR - ANother Tool for Language Recognition

MVCC – Multi-Version Concurrency Control

DTO – Data Transfer Object (Objek na prenos údajov)

1 Úvod

Vývoj technológií priniesol širokú škálu možností databázových systémov, ktoré vyhovujú rôznym požiadavkám a preferenciám. Na ukladanie údajov a prístup k nim sa bežne používajú databázy v binárnom formáte, ale niektoré databázy, ako napríklad Apache Cassandra, vydláždili cestu pre formáty čitateľné pre človeka. Hoci textové databázy nemusia byť pri spracovaní veľkého množstva údajov alebo spracovaní zložitých dotazov také efektívne ako binárne databázy, zostávajú obľúbenou voľbou pre jednoduché aplikácie alebo v situáciách, keď je jednoduchosť používania a prenosnosť dôležitejšia ako výkon.

Cieľom tejto bakalárskej práce je poskytnúť komplexnú analýzu problémov spojených s používaním ľudsky čitateľného formátu na ukladanie a prístup k údajom v databáze a vyvinúť spoľahlivý JDBC ovládač, ktorý tieto problémy rieši.

Analytická časť tohto projektu sa bude zaoberať najdôležitejšími triedami a rozhraniami rozhrania JDBC API a skúmať rôzne spôsoby práce s údajmi v databáze. Porovnáme aj rôzne populárne serializačné formáty, ktoré sú čitateľné pre človeka, a poukážeme na obmedzenia spojené s ukladaním údajov v súborovom systéme. Okrem toho opíšeme dva rôzne prístupy k spracovaniu príkazov SQL a podrobne rozoberieme spôsob spracovania transakcií v rôznych databázach, ako aj rôzne úrovne izolácie.

V časti o návrhu zdôvodníme metódy a prístupy zvolené na riešenie problémov identifikovaných v časti o analýze.

V implementačnej časti sa podrobne sústredíme na každý kľúčový komponent nášho ovládača spolu s prehľadom jeho fungovania.

V časti o overovaní riešenia predstavíme testovacie metódy použité na posúdenie funkčnosti každej zložky nášho ovládača. Okrem toho porovnáme výkon a možnosti nášho ovládača s možnosťami ovládača SQLite.

V závere ponúkneme odporúčania na budúce vylepšenia nášho ovládača, pričom poukážeme na oblasti, v ktorých by bolo možné implementovať ďalšie vylepšenia alebo optimalizácie.

2 Analýza

2.1 Architektúra JDBC

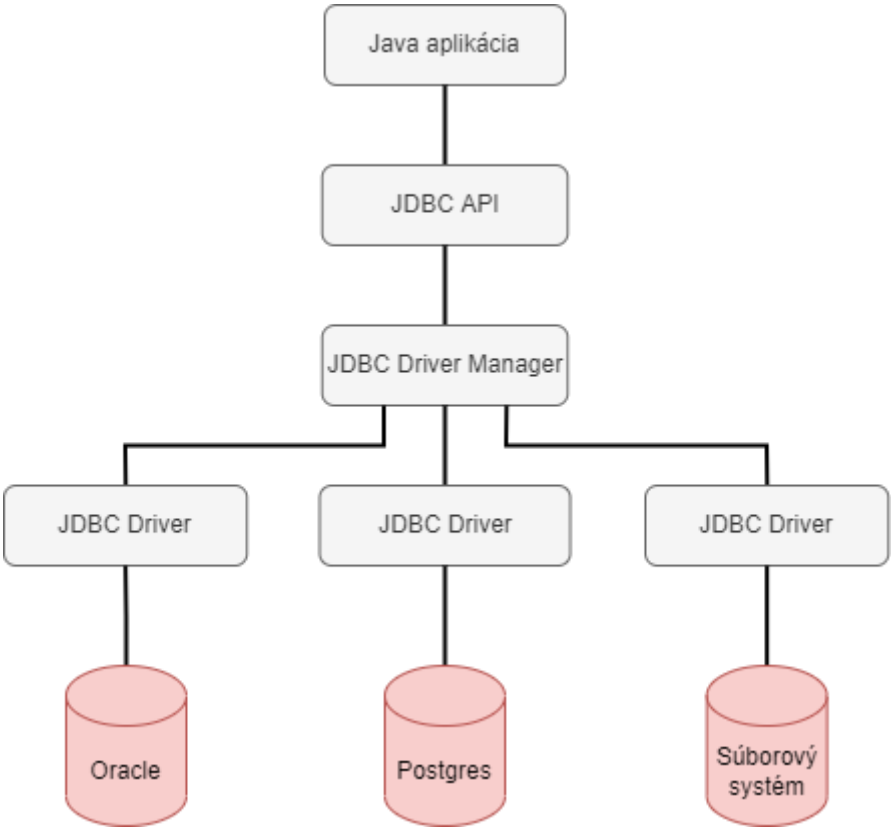
JDBC API poskytuje prístup k relačným databázam pre Java aplikácie. Podporované sú databázy ako SQL server, SQLite, Postgres, Oracle, DB2, avšak je možné pristupovať prakticky k akémukoľvek zdroju údajov od relačných databáz až po tabuľky a ploché databázové súbory (Flat File). [1]

Hlavnou vlastnosťou JDBC je, že ide o štandardné API, čo znamená, že pre rôzne databázy nemusíme vyvíjať odlišný kód. JDBC uľahčuje vývoj programov v jazyku Java, ktoré zvládajú nasledujúce úlohy:

- Pripojiť sa k databáze
- Vykonávať dotazy (query) a príkazy (statement) na aktualizáciu databázy
- Získať výsledok prijatý z databázy.

JDBC API je rozdelené do dvoch balíkov: *java.sql*, čo je hlavné rozhranie API, a *javax.sql*, čo je doplnkový balík JDBC.

- *java.sql* obsahuje základné rozhranie JDBC API na získanie prístupu k údajom uloženým v databáze a na ich úpravu.
- *javax.sql* obsahuje API pre klientov JDBC na prístup k zdrojom údajov na strane servera. [2]



Obrázok 1: Architektúra JDBC

Aplikácia - Je to Java aplikácia alebo servlet, ktorý komunikuje so zdrojom údajov.

JDBC API - Ide o API založené na jazyku Java, ktoré poskytuje množinu tried a metód na vytvorenie pripojenia k databáze, vykonávanie SQL príkazov a spracovanie výsledkov. V nasledujúcej podkapitole sa budeme venovať dôležitejším triedam a rozhraniam.

DriverManager - zabezpečuje, aby na prístup k zdrojom údajov bol použitý správny ovládač. DriverManager je schopný podporovať viacero súbežných ovládačov spojených k viacerým heterogénnym databázam. [3]

JDBC Drivers - Sú to ovládače, ktoré implementujú rozhranie JDBC API a poskytujú mechanizmus na pripojenie k určitej databáze.

Dôležité triedy a rozhrania

Zoznam tried, ktoré musia byť implementované v novovytvorenom JDBC ovládači, závisí od požiadaviek aplikácie, ktorá bude náš ovládač používať. Existuje však základná množina tried, ktoré musia byť vždy implementované, aby ovládač fungoval.

Driver - Rámec Java SQL umožňuje použitie viacerých databázových ovládačov. Rozhranie *Driver* zabezpečuje kľúčové spojenie medzi Java aplikáciou a databázou. Ovládač JDBC transformuje bežné požiadavky API na nízkoúrovňové požiadavky pre konkrétne databázy. Trieda *Driver* by si mala pri načítaní vytvoriť inštanciu seba samého a zaregistrovať ju v *DriverManager*. [4]

Connection – Zapuzdruje pripojenie k určitej databáze. Objekt sa vytvorí zavolaním funkcie *DriverManager.getConnection(),* ktorý vráti objekt typu *Connection*. Jeho primárnym použitím je vykonávanie SQL príkazov v kontexte spojenia. [5]

Statement – Slúži na spúšťanie SQL príkazov a vracanie výsledkov pomocou objektov *ResultSet*. Objekt *Statement* sa vytvorí pomocou metódy *createStatement*() z objektu *Connection*. V každom okamihu obsahuje objekt *Statement* práve jeden objekt *ResultSet*. [5]

PreparedStatement - Skompilovaný SQL príkaz je známy ako *PreparedStatement*. Keďže je skompilovaný, zjednodušuje definovanie hodnôt parametrov SQL, chránia pred útokmi typu SQL injection a v neposlednom rade môže zvýšiť výkon aplikácie. *PreparedStatement* je uložený tak, aby sa mohol neskôr vykonať viackrát. Pomocou metódy *prepareStatement()* rozhrania *Connection* môžeme získať objekt tohto rozhrania. [3]

ResultSet - Objekty *ResultSet* uchovávajú údaje získané z databázy po vykonaní SELECT príkazu. *ResultSet* funguje ako iterátor, ktorý umožňuje pohyb po jeho údajoch. [3]

ResultSetMetaData - Táto trieda sa používa na načítanie údajov z objektu *ResultSet*. Obsahuje metódy, ktoré umožňujú získať dôležité informácie o

výsledku dotazu. Po vykonaní dotazu je možné zavolať metódu *getMetaData()* na získanie objektu *ResultSetMetaData* pre výsledné údaje.

2.2 Spracovanie údajov

DBMS je počítačový softvér, ktorý riadi zhromažďovanie súvisiacich štruktúrovaných údajov. Ten sa používa na ukladanie údajov a efektívne získavanie informácií podľa potreby. Údaje možno získať pomocou dotazov SQL a relačnej algebry. Príkladmi systémov správy databáz sú Oracle, MySQL a MS SQL server.

Súborový systém (FS) je metóda zoskupovania súborov na pevnom disku alebo na inom úložnom prostriedku. FS tvoria rôzne súbory, ktoré sú usporiadané do adresárov. FS vykonáva základné funkcie vrátane správy, pomenovania súborov, udeľovania pravidiel prístupu atď. Medzi súborové systémy patria napríklad NTFS a EXT. [6]

Celkovo možno povedať, že hoci sa DBMS aj súborový systém používajú na ukladanie údajov v počítači, DBMS je výkonnejší a sofistikovanejší nástroj na správu a manipuláciu s veľkým množstvom štruktúrovaných údajov, zatiaľ čo FS je jednoduchší systém, ktorý je vhodnejší na organizovanie a ukladanie menšieho množstva údajov.[7]

Rôzne databázy používajú rôzne formáty databázových súborov na ukladanie tabuľkových údajov. Databázové tabuľky a indexy môžu byť uložené na disku v jednej z viacerých foriem vrátane usporiadaných/neusporiadaných plochých databázových súborov, ISAM, heap súborov, hash bucketov alebo B+ stromov. [8]

Pokiaľ ide o spracovanie údajov, rôzne DBMS sa správajú rôzne. Hoci SQLite môže používať pamäť na účely ukladania do vyrovnávacej pamäte, nenahrá do pamäte celú databázu. Namiesto toho číta a zapisuje údaje na disk podľa potreby. Vďaka tomu je SQLite efektívny z hľadiska využitia pamäte a

umožňuje spracovávať veľmi veľké databázy bez toho, aby sa pamäť vyčerpala. [9]

Alternatívne riešenie ponúka databáza HSQLDB, ktorá načíta z tabuľky pevný počet záznamov (veľkosť stránky) a uloží ich do pamäte. Pri prechádzaní záznamov HSQLDB podľa potreby načíta do pamäte ďalšie strany záznamov. Tento mechanizmus stránkovania umožňuje HSQLDB efektívne spracovávať veľké súbory údajov bez vyčerpania pamäte. [10]

Načítanie údajov zo súborov do pamäte môže mať výhody aj nevýhody v závislosti od veľkosti údajov a dostupnej pamäte.

Výhody	**Nevýhody**
Po načítaní údajov do pamäte je možné k nim pristupovať oveľa rýchlejšie ako pri čítaní zo súboru, najmä pri náhodnom prístupe k rôznym častiam údajov.	Načítanie veľkých súborov údajov do pamäte môže vyžadovať značné množstvo pamäte, ktorá nemusí byť na niektorých počítačoch k dispozícii.
Dáta v pamäti sa dajú ľahko manipulovať a transformovať pomocou rôznych programových knižníc, čo môže zjednodušiť úlohy spracovania údajov.	Načítanie údajov do pamäte môže v závislosti od veľkosti údajov trvať dlho, najmä v prípade pomalších ukladacích zariadení, ako sú pevné disky.

Tabuľka 1: Výhody a nevýhody načítania údajov do pamäte

2.3 Ľudsky čitateľný formát (HRF)

HRF je typ reprezentácie údajov, ktoré sú pre človeka ľahko zrozumiteľné a interpretovateľné. Tieto formáty sa používajú v informatike, aby umožnili ľuďom porozumieť údajom a pracovať s nimi bez potreby špecializovaného softvéru alebo technických znalostí. Ukladanie údajov v HRF v porovnaní s binárnymi formátmi má niekoľko výhod a nevýhod: [11]

Výhody	Nevýhody
Sú jednoduché na čítanie a interpretáciu: Údaje sú uložené tak, aby ich ľudia mohli jednoducho čítať a pochopiť, čo uľahčuje prácu s nimi a odstraňovanie problémov.	V porovnaní s binárnymi formátmi sú formáty HRF často väčšie, pretože potrebujú viac priestoru na uloženie ďalších znakov a formátovania potrebného na to, aby boli údaje čitateľné pre človeka.
Ľahko sa dajú prenášať medzi mnohými počítačmi, pretože sú textové a nezávislé od konkrétnej hardvérovej platformy alebo operačného systému.	Ich spracovanie môže trvať dlhšie ako spracovanie binárnych formátov, pretože sú väčšie a na ich spracovanie je potrebný väčší výpočtový výkon.
Parsovanie HRF je vo všeobecnosti jednoduchšie ako parsovanie binárnych formátov, pretože štruktúra údajov je jasne definovaná a ľahko sa interpretuje.	HRF sú jednoduchšie na čítanie a pochopenie, čo ich robí potenciálne menej bezpečnými, pretože sú viac otvorené neoprávnenej interpretácii.

Tabuľka 2: Výhody a nevýhody HRF

Medzi najpopulárnejšie HRF patria CSV, JSON, XML a YAML. V tejto kapitole budeme analyzovať jednotlivé formáty z hľadiska serializácie a

deserializácie, a potom sa budeme venovať rôznym spôsobom ukladania údajov v týchto formátoch.

CSV – CSV je jednoduchý textový formát na ukladanie tabuľkových údajov. Formát CSV umožňuje voliteľný riadok hlavičky, ktorý sa zobrazuje ako prvý riadok súboru. Ak je prítomný, obsahuje názvy polí pre každú hodnotu v zázname. Tento riadok hlavičky je veľmi užitočný na označovanie údajov a mal by byť prítomný takmer vždy. Aj keď sa CSV považuje za formát, ktorý je do istej miery čitateľný pre človeka, v nasledujúcej tabuľke si môžeme všimnúť, že ak má tabuľka veľa stĺpcov, je veľmi ťažké určiť, ktorá hodnota je priradená ktorému stĺpcu.

JSON - JSON je jedným z najrýchlejších a najpoužívanejších formátov na serializáciu/deserializáciu. Dátové štruktúry formátu JSON presne reprezentujú bežné objekty v mnohých jazykoch. Jazyk podporuje štruktúry typu pole a mapa (objekt).
Podporuje mnoho rôznych dátových typov vrátane reťazcov, čísel, logické typy, null atď. ale nie dátumy. Názov v pároch name/value objektu JSON musí byť vždy reťazec. Má stručnú syntax, hoci vo väčšine situácií nie je taká stručná ako YAML a nepodporuje komentáre. [12]

XML - XML je serializačný formát, ktorý je čitateľný pre človeka. Jednou z nevýhod XML je jeho veľká veľkosť. Jeho opisné koncové značky, ktoré vyžadujú opätovné zadanie názvu obklopujúceho prvku, zvyšujú počet bajtov dokumentu. Jazyk je relatívne rýchly, hoci jeho parsovanie je zvyčajne pomalšie ako parsovanie JSON. Je veľmi flexibilný, pretože každý element môže mať atribúty a ľubovoľné podradené elementy. Pri XML sa tiež odporúča používať XSD, ako primárnu schému na použitie s XML. [12]

YAML - YAML sa považuje za pomerne čitateľný a ľahko sa s ním pracuje, najmä v porovnaní s inými formátmi, ako sú XML alebo JSON. YAML používa jednoduché páry kľúč-hodnota, odsadenie a niekoľko ďalších konvencií na reprezentáciu zložitých dátových štruktúr. Hodnoty môžu byť predvolené ako reťazce, čo umožňuje vynechať úvodzovky.

YAML je zo všetkých formátov najpomalší. Jazyk je dobre štandardizovaný, ale môže byť ťažké nájsť ďalšie funkcie, napríklad validátory schém. [12]

V nasledujúcej tabuľke je uvedený jeden záznam z databázovej tabuľky *catalog* zastúpených vo vyššie uvedených jazykoch.

Jazyk	Formát jedného záznamu
CSV	"id","author","title","genre","price","publish_date","description" "bk101","Gambardella, Matthew","XML Developer's Guide","Computer","44.95","2000-10-01","An in-depth look at creating applications with XML."

JSON	{
	"book": {
	"author": "Gambardella, Matthew",
	"title": "XML Developer's Guide",
	"genre": "Computer",
	"price": 44.95,
	"publish_date": "2000-10-01",
	"description": "An in-depth look at creating applications with XML."
	}
	}

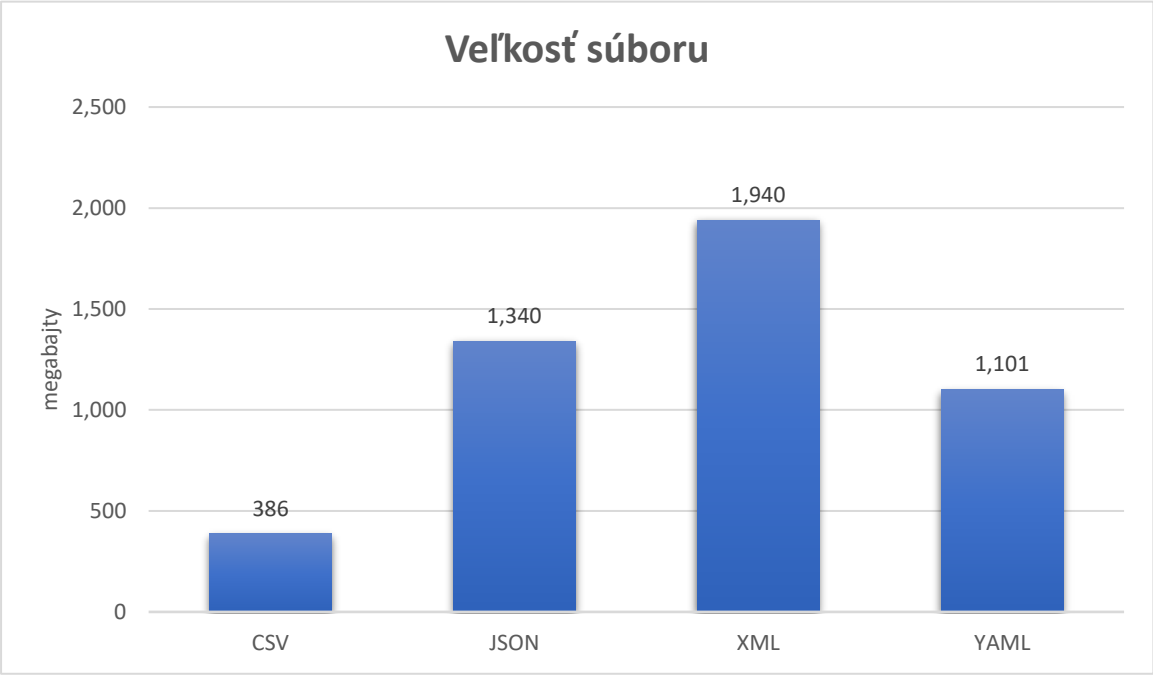
XML	<?xml version="1.0" encoding="UTF-8" ?> <book id="bk101"> <author>Gambardella, Matthew</author> <title>XML Developer's Guide</title> <genre>Computer</genre> <price>44.95</price> <publish_date>2000-10-01</publish_date> <description>An in-depth look at creating applications with XML.</description> </book>
YAML	book: author: "Gambardella, Matthew" title: "XML Developer's Guide" genre: Computer price: "44.95" publish_date: "2000-10-01" description: "An in-depth look at creating applications with XML." _id: bk101

Tabuľka 3: Ukážka jedného záznamu

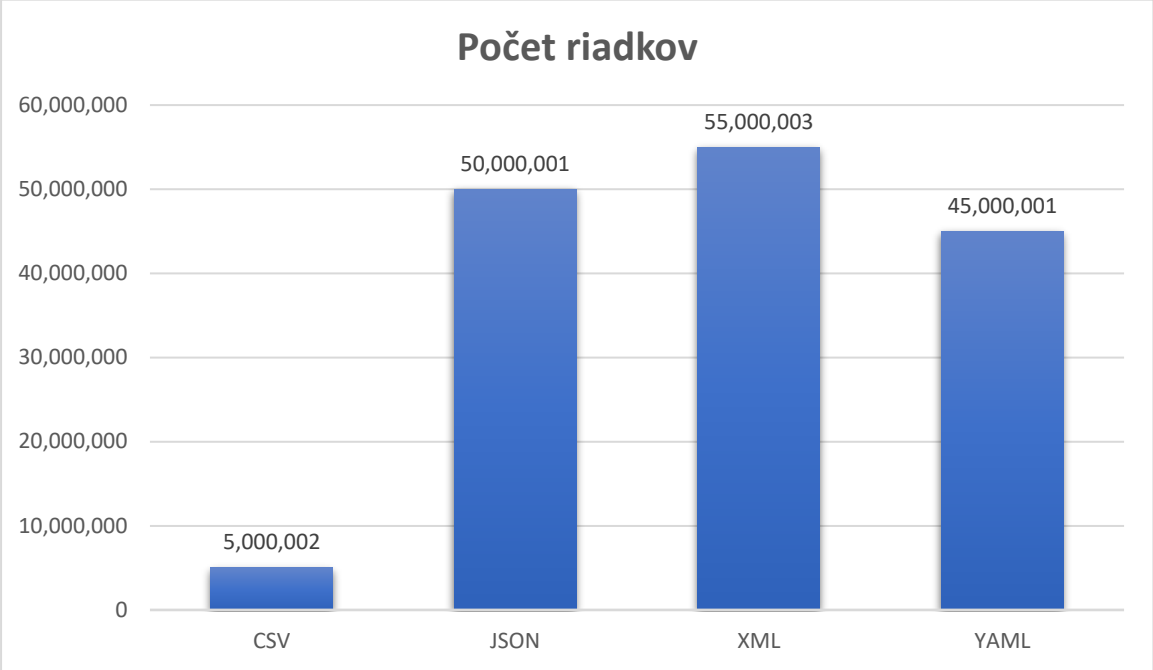
2.3.1 Serializácia a deserializácia

Na tento účel je k dispozícii viacero knižníc, ale kvôli konzistentnosti sme používali pre každý formát verziu Jacksonu. V porovnávacom teste sme vygenerovali zoznam 2 milión objektov s rôznymi náhodnými atribútmi.

Tento zoznam bol najprv serializované do súboru a potom deserializované späť do zoznamu. Nakoniec, aby sme sa uistili, že dostaneme rovnaký výsledok, sme zoznam reserializovali do iného súboru a porovnali obsah týchto súborov.



Tabuľka 4: Veľkosti jednotlivých súborov



Tabuľka 5: Počet riadkov

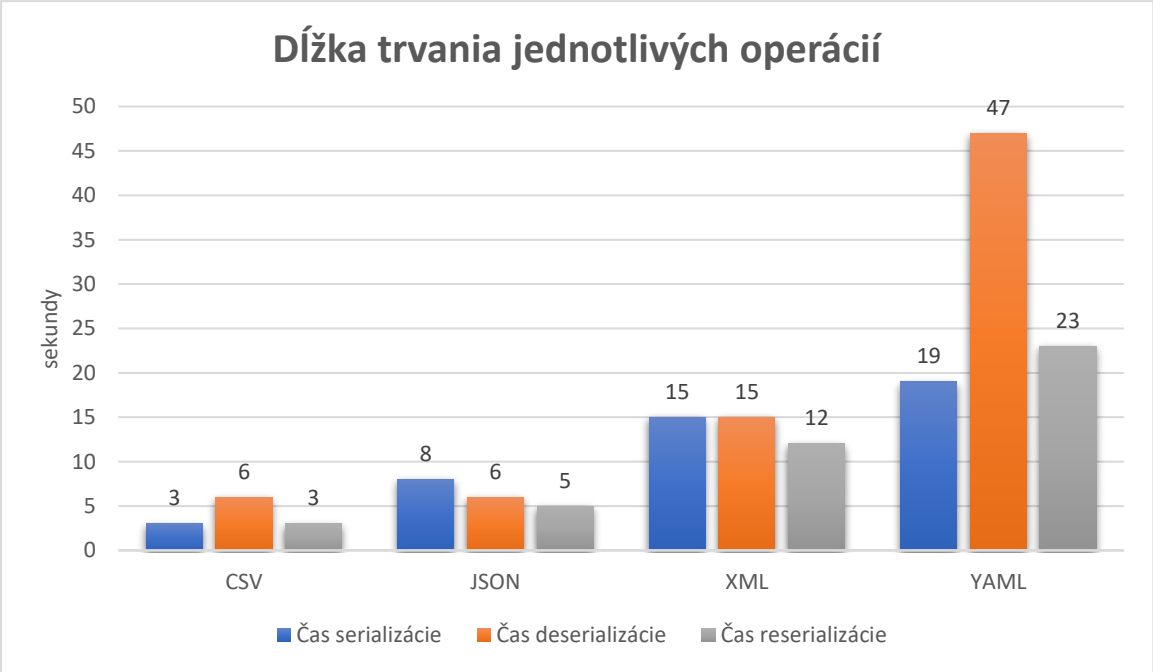
Tabuľka 6: Počet znakov

V prípade CSV sa názov stĺpca definuje len raz, v prvom riadku, pri ostatných formátoch sa definuje pre každý záznam v zozname. To vysvetľuje,

prečo je veľkosť súboru CSV, počet znakov a riadkov v porovnaní s inými formátmi taký nízky.

JSON a YAML mali podobnú dĺžku, ale keďže YAML nepotrebuje kučeravé ani hranaté zátvorky, je o 10-15 % kratší a menší.

Keďže XML potrebuje pre každý neprázdny prvok uzatváraciu značku, má najväčšiu veľkosť súboru a je zo všetkých najväčší.



Tabuľka 7: Čas trvania jednotlivých operácií

Je dôložité zdôrazniť, že rýchlosť serializácie a deserializácie nezávisí iba od veľkosti súboru a počtom riadkov, ale aj od špecifikácie daného jazyka. Čím rozsiahlejšia je gramatika, jazyka, tým je pomalšia deserializácia.

2.3.2 Možné spôsoby ukladania údajov

V tejto kapitole sa budeme venovať porovnaniu rôznych spôsobov ukladania súborov. Na ukážku budeme používať formát XML, ale ak nie je uvedené inak, každý spôsob je použiteľný aj pre ostatné formáty.

2.3.2.1 Každý záznam je samostatný súbor

Môžeme si vybrať možnosť, pri ktorej je každý záznam v databáze uložený v samotnom súbore. V tomto prípade môžeme všetky záznamy z jednej tabuľky uložiť do jedného priečinka. Tento spôsob ukladanie súborov má určité obmedzenia, o ktorých budeme hovoriť neskôr v tejto kapitole.

Názov tabuľky tiež môžeme deklarovať na začiatku súbora, a tak nebudeme obmedzení len na jednu tabuľku:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<catalogTable>
    <Entry>
        <author>Gambardella, Matthew</author>
        <title>XML Developer's Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publishDate>2000-10-01</publishDate>
        <description>An in-depth look at creating applications with XML.</description>
    </Entry>
</catalogTable>
```

2.3.2.2 Každá tabuľka je samostatný súbor

Môže jeden súbor obsahovat záznamy iba z jednej tabuľky. Tento spôsob má za výhodu, že vždy vieme, aký súbor máme otvoriť, ak hľadáme jeden záznam.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<catalogTable>
    <Entry>
        <author>Gambardella, Matthew</author>
        ...
        <description>An in-depth look at creating applications with XML.</description>
    </Entry>
    ...
    <Entry>
        <author>Thurman, Paula</author>
        ...
        <description>Some description here.</description>
    </Entry>
</catalogTable>
```

2.3.2.3 Viacero tabuliek v jednom súbore

Táto metóda je vhodná vtedy, keď chceme mať celú databázu uloženú v jednom súbore. Ak ide o veľký súbor, tak metóda má nevýhodu, že máme prechádzať celý súbor, aj keď hľadáme iba jeden záznam alebo jednu tabuľku.

Je dôležité zdôrazniť, že táto metóda nie je použíteľná pre CSV, dôvod bol vysvetlený v predchádzajúcom kapitole.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Tables>
    <catalogTable>
        <Entry>
            <author>Gambardella, Matthew</author>
            ...
            <description>Creating applications with XML.</description>
        </Entry>
    </catalogTable>
    <authorsTable>
        <Entry>
            <fullName>Zsolt Kiss</fullName>
            ...
            <numberOfPublications>1</numberOfPublications >
        </Entry>
        ...
        <Entry>
            <fullName>Ing. Gabriel Szabó</fullName>
            ...
            <numberOfPublications>2</numberOfPublications>
        </Entry>
    </authorsTable>
</Tables>
```

2.3.2.4 Uloženie LOB-ov

Ak tabuľka obsahuje stĺpec typu BLOB, bolo by možné obsah týchto objektov oddeliť do vlastných súborov a tabuľka by obsahovala len cestu k týmto súborom.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<catalogTable>
    <Entry>
        <author>Gambardella, Matthew</author>
        ...
        <description>An in-depth look at creating applications with XML.</description>
        <file>path/to/lob_001.xml</file>
    </Entry>
    ...
    <Entry>
        <author>Thurman, Paula</author>
        ...
        <description>Some description here.</description>
        <file>path/to/lob_002.xml</file>
    </Entry>
</catalogTable>
```

2.3.3 Obmedzenia

2.3.3.1 Maximálny počet súborov v jednom adresári

V závislosti od súborového systému existujú obmedzenia, ktoré určujú, koľko súborov môže obsahovať jeden adresár.

Súborový systém	Max. počet súborov pre jeden adresár
FAT32	2^16 - 1 (65 535)
exFAT	2 796 202
NTFS	2^32 - 1 (4 294 967 295)
Ext2	1.3×10^20
Ext3	Variabilné, pridelené v čase vytvorenia
Ext4	4 miliardy (určené v čase vytvorenia súborového systému)

Tabuľka 8: Max. počet súborov v jednom adresári pri jednotlivých FS

Z toho vyplýva, že aj keď do toho zahrnieme systémové súbory a ďalšie súbory, ktoré používateľ má uložené na disku, vo väčšine prípadoch je to pre naše účely postačujúce. Vo všeobecnosti sa neodporúča ukladať veľký počet súborov do jedného adresára, pretože to môže viesť k problémom s výkonom a iným problémom. Namiesto toho je zvyčajne najlepšie usporiadať súbory do viacerých adresárov, pričom každý adresár obsahuje menší počet súborov. To môže pomôcť zlepšiť výkon a spoľahlivosť súborového systému a môže to tiež uľahčiť správu a prístup k súborom.

2.3.3.2 Maximálny počet súčasne otvorených súborov

Vo všeobecnosti neexistujú žiadne udané obmedzenia počtu súborov, ktoré môže Java proces otvoriť. Na druhej strane môžeme byť obmedzený množstvom dostupnej pamäte a iných systémových zdrojov, ako aj konkrétnou implementáciou prostredia Java Runtime. Okrem toho niektoré operačné

systémy môžu zaviesť obmedzenia počtu súborov, ktoré môže jeden proces otvoriť súčasne.

Pri našom testovaní sme napísali program, ktorý vytvára konečný počet súborov a potom do nich zapisuje znaky ASCI. Naše testovanie ukázalo, že neexistuje obmedzenie počtu súborov, ktoré možno otvoriť naraz, jediným obmedzujúcim faktorom je množstvo pamäte, ktoré je k dispozícii v operačnom systéme.

2.4 SQL Príkazy

SQL príkazy pozostávajú z viacerých vstupov, ako sú názvy tabuliek, názvy stĺpcov, rôzne funkcie a výrazy. Podľa toho, či sa príkazy nachádzajú vo vyrovnávacej pamäti rozlišujeme dva spôsoby spracovania príkazov.

Ak sa v rámci cache nájde opakovane použiteľný prvok, stačí ho vybrať a použiť na vykonanie príkazu. Tento postup sa nazýva Soft Parsing.

Ak nie je možné nájsť opakovane použiteľný prvok alebo ak sa dotaz nikdy predtým nevykonal, je potrebná parsovanie príkazu. To sa nazýva Hard Parsing.

Parsovanie (syntaktická analýza) je proces analýzy reťazca symbolov, či už v prirodzenom jazyku, počítačových jazykoch alebo dátových štruktúrach, ktorý zodpovedá pravidlám formálnej gramatiky.

Pred vykonaním akýchkoľvek úprav údajov v databáze musíme potvrdiť, že bolo splnené každé z nasledujúcich kritérií:

- Kontrola syntaxe – či daný príkaz je zrozumiteľný pre databázu, a neobsahuje syntax chyby. Syntax sa kontroluje porovnaním celého textu príkazu s podporovanými kľúčovými slovami pre aktuálne používanú verziu databázy.
- Kontrola sémantiky - či je použitý správny názov objektu a tiež či má používateľ správne oprávnenia na vykonanie dotazu.

Ak niektorá z týchto dvoch kontrol zlyhá, proces vykonávania dotazu sa ukončí.

2.4.1 Regulárne výrazy

Príkazy SQL možno parsovať pomocou regulárnych výrazov (ďalej iba REGEX), ale tento prístup má niekoľko nevýhod. Po prvé, REGEX môže byť náročné na čítanie a pochopenie, najmä pri pokuse parsovaní dlhých a zložitých SQL príkazov. Okrem toho môže byť parsovanie SQL príkazov náchylné na chyby, najmä ak vzor nie je dôsledne skontrolovaný alebo ak zmeny v kódovej základni SQL narúšajú vzor.

Napokon, spracovanie REGEX môže byť náročné na výpočet, čo môže mať za následok pomalý výkon a potenciálne ovplyvniť celkový výkon systému. Preto, hoci REGEX možno použiť na parsovanie SQL príkazov, je nevyhnutné uvedomiť si ich obmedzenia a preskúmať iné možnosti, aby sa zabezpečil efektívny a presný rozbor.

2.4.2 Parsovanie pomocou bezkontextovej gramatiky

Dôležité uplatnenie bezkontextových gramatík je pri špecifikácii a kompilácii programovacích jazykov. Bezkontextová gramatika pozostáva zo súboru pravidiel, z ktorých každá vyjadruje spôsoby, akými možno zoskupiť a usporiadať symboly jazyka, a lexikónu slov a symbolov.

Väčšina kompilátorov a interpretov obsahuje komponent nazývaný parser, ktorý extrahuje význam programu pred generovaním skompilovaného kódu alebo vykonaním interpretovanej exekúcie. Ak je k dispozícii bezkontextová gramatika, konštrukciu parseru uľahčuje niekoľko metodík. Niektoré nástroje dokonca automaticky generujú parser z gramatiky.

Parsovací strom je usporiadaný, zakorenený strom, ktorý reprezentuje syntaktickú štruktúru reťazca podľa určitej bezkontextovej gramatiky.

2.5 Transakcie

Transakcia je jednotka práce, ktorá pozostáva z jednej alebo viacerých databázových operácií, ako je vkladanie, aktualizácia alebo mazanie údajov. Transakcie sú atomické, čo znamená, že buď sa úspešne vykonajú všetky operácie v transakcii, alebo sa nevykoná žiadna z nich. [2]

Operácia commit označuje úspešné ukončenie transakcie, čím sa jej zmeny stávajú trvalými a viditeľnými pre ostatné transakcie. Keď sa vykoná operácia commit, DBMS zabezpečí, aby sa všetky zmeny vykonané transakciou zapísali na disk alebo sa inak stali trvalými. [13]

Operácia rollback vzťahuje na operáciu zrušenia alebo vrátenia zmien vykonaných nedokončenou alebo neúspešnou transakciou. Keď sa pri transakcii vyskytne chyba alebo keď ju aplikácia alebo používateľ explicitne vráti späť, DBMS použije operáciu rollback na obnovenie databázy do stavu pred začatím transakcie. Databáza sa tak vráti do známeho a konzistentného stavu, čím sa zabráni tomu, aby boli akékoľvek čiastočne dokončené alebo nesprávne zmeny viditeľné pre iné transakcie. [14]

Pri používaní JDBC môžeme transakciu zrušiť zavolaním funkcie *rollback()* na objekte Connection. [2]

2.5.1 Mechanizmy uzamknutia databázy

Optimistické a pesimistické zamykanie sú dva prístupy k riadeniu súbežného prístupu k údajom v databáze.

Pesimistické zamykanie je mechanizmus, ktorý predpokladá, že konflikty medzi transakciami sa môžu vyskytnúť, a snaží sa im predchádzať tým, že údaje zamyká okamžite, keď k nim transakcia pristupuje. Zámok sa udržiava, kým transakcia neuvoľní údaje, čo môže viesť k nižšej súbežnosti, ale zabezpečuje konzistenciu. Tento prístup sa bežne používa v transakčných systémoch, kde je konzistentnosť údajov veľmi dôležitá.

Na druhej strane optimistické zamykanie je mechanizmus, ktorý predpokladá, že konflikty medzi transakciami sú zriedkavé, a umožňuje viacerým transakciám pristupovať k rovnakým údajom súčasne. Každá transakcia však pred odovzdaním svojich zmien skontroluje, či od posledného čítania údajov nedošlo k ich úprave inými transakciami. Ak sa zistia konflikty, transakcia sa preruší a začne sa znova s aktualizovanými údajmi. Tento prístup môže viesť k vyššej súbežnosti, ale môže mať za následok viac prerušení transakcií a nižšiu konzistenciu. [15] [16], [17]

2.5.2 Úrovne izolácie

To, aké údaje môže aplikácia vidieť v rámci transakcie, je určené úrovňami izolácie transakcií v databáze. Úrovne izolácie transakcií sú opísané z hľadiska toho, či sú na konkrétnej úrovni izolácie povolené tri typy scenárov: [2]

Dirty reads (nečisté čítanie) je situácia, keď transakcia číta údaje, ktoré ešte neboli zapisované. Napríklad transakcia 1 aktualizuje riadok a ponechá ho zapisované, zatiaľ čo transakcia 2 číta aktualizovaný riadok. Ak transakcia 1 zruší zmenu, transakcia 2 bude čítať údaje, ktoré sa považujú za nikdy neexistujúce. [2]

Nonrepeatable reads (neopakovateľné čítanie) nastáva vtedy, keď transakcia číta ten istý riadok dvakrát a zakaždým získava inú hodnotu. Napríklad predpokladajme, že transakcia T1 číta údaje. Z dôvodu súbežnosti iná transakcia T2 aktualizuje tie isté údaje a vykoná commit, Ak teraz transakcia T1 znovu prečíta tie isté údaje, získa inú hodnotu. [2]

Phantom reads - Fantómové čítanie nastáva vtedy, keď sa vykonajú dva rovnaké dotazy, ale riadky načítané týmito dvoma dotazmi sa líšia. Predpokladajme

napríklad, že transakcia T1 načíta sadu riadkov, ktoré spĺňajú určité vyhľadávacie kritériá. Teraz transakcia T2 vygeneruje niekoľko nových riadkov, ktoré vyhovujú vyhľadávacím kritériám transakcie T1. Ak transakcia T1 znovu vykoná príkaz, ktorý načíta riadky, získa tentoraz inú sadu riadkov. [2]

Izolačná úroveň	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITTED	Môžu sa vyskytnúť	Môžu sa vyskytnúť	Môžu sa vyskytnúť
READ COMMITTED	Nevyskytujú sa	Nevyskytujú sa	Môžu sa vyskytnúť
REPEATABLE READ	Nevyskytujú sa	Nevyskytujú sa	Môžu sa vyskytnúť
SERIALIZABLE	Nevyskytujú sa	Nevyskytujú sa	Nevyskytujú sa

Tabuľka 9: Izolačné úrovne

Read Uncommitted je najnižšia úroveň izolácie je. Na tejto úrovni má jedna transakcia možnosť čítať modifikácie vykonané inými transakciami, ktoré ešte neboli zapisované, čo umožňuje špinavé čítanie. Na tejto úrovni nie sú transakcie navzájom izolované. [2]

Read Committed zabezpečí, že údaje čítané sú v čase čítania zaručene zapisované. To zabraňuje špinavému čítaniu. Transakcia bráni ostatným transakciám čítať, upravovať alebo mazať aktuálny riadok tým, že na ňom drží zámok na čítanie alebo zápis. [2]

Repeatable read je najobmedzenejšia úroveň izolácie. Pre všetky riadky, na ktoré sa transakcia odvoláva, drží zámky na čítanie a pre operácie aktualizácie a

vymazania drží zámky na zápis do tých istých riadkov. Vyhýba sa neopakovateľnému čítaniu, pretože iné transakcie nemôžu tieto riadky čítať, aktualizovať ani mazať. [2]

Serializable je najvyššia úroveň izolácie. Je zaručené, že serializovateľné vykonanie je serializované. Serializovateľné vykonávanie je definované ako vykonávanie operácií, pri ktorom sa súčasne vykonávané transakcie javia ako sériovo vykonávané. [2]

2.5.3 Techniky riadenia transakcií v jednotlivých databázach

V typickej databáze založenej na súborovom systéme sa transakcie realizujú pomocou techniky "write-ahead logging", pri ktorej sa všetky zmeny zapisujú do logovacieho súboru predtým, ako sa aplikujú na skutočné dátové súbory. Tento prístup zabezpečuje, že zmeny možno v prípade chýb vrátiť späť a že zmeny pretrvajú aj v prípade havárie databázy. Napríklad SQLite používa na správu transakcií mechanizmus zapisovania do denníka (write-ahead logging).

Na druhej strane databázy v pamäti, ako napríklad HSQLDB a H2, sa pri podpore transakcií spoliehajú na techniku MVCC (Multi-Version Concurrency Control). MVCC umožňuje viacerým transakciám pristupovať k tým istým údajom súčasne tým, že pre každú transakciu vytvára rôzne verzie údajov namiesto uzamykania údajov ako v tradičných mechanizmoch uzamykania.

V rámci MVCC každá transakcia vidí aktuálny obraz (SNAPSHOT) databázy, ktorá existovala na začiatku transakcie. Keď transakcia modifikuje údaje, vytvorí novú verziu modifikovaných údajov, ktorá je viditeľná len pre túto transakciu. Ostatné transakcie naďalej vidia starú verziu údajov, kým ich tiež nezmenia, a vtedy si vytvoria vlastnú verziu údajov.

Keď transakcia vykoná commit, jej zmeny sa zapíšu na disk a stanú sa viditeľnými pre ostatné transakcie. Ak dôjde ku konfliktu medzi dvoma transakciami, jedna z nich je vrátená späť a môže opakovať transakciu s novou verziou údajov. [18][19][20]

3 Návrh

3.1 Spracovanie údajov

Po dôkladnom zvážení sme sa rozhodli načítať údaje najprv do pamäte a potom ich spracovávať. Načítaním údajov do pamäte môžeme dosiahnuť lepší výkon v porovnaní s inými prístupmi. Toto rozhodnutie však so sebou prináša nevýhodu v podobe vyššej spotreby pamäte, keďže pri interakcii s tabuľkou je potrebné načítať do pamäte celý súbor. Napriek možnému zvýšeniu spotreby pamäte sa domnievame, že tento prístup nám poskytne najlepšie výsledky z hľadiska výkonu a vyhovuje našim požiadavkám pri práci so stredne veľkou databázou (napr. 100 MB).

3.2 Ľudsky čitateľný formát

Na základe analýzy sme sa rozhodli použiť formáty JSON a XML z dôvodu rýchlej serializácie, deserializácie a možnosti overenia integrity spracovaných údajov prostredníctvom schém.

3.2.1 Formát databázového súboru

Súbor s metadátami databázy bude tiež používať HRF. V tomto súbore budú uložené cesty k tabuľkám a schémam.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Database name="myDatabase">
    <Table name="myTable">
        <pathToTable>D:\path\to\myDatabase\myTable.xml</pathToTable>
        <pathToSchema>D:\path\to\myDatabase\myTable.xsd</pathToSchema>
    </Table>
</Database>
```

3.2.2 Formát tabuľky

Po analýze rôznych možností ukladania databázových tabuliek sme zistili, že najlepšie by bolo ukladať každú tabuľku ako samostatný súbor. Tento prístup eliminuje akékoľvek obavy z prekročenia povoleného počtu súborov v jednom adresári.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<myTable>
    <Entry>
        <id>1</id>
        <name>Zsolti</name>
        <age>25</age>
    </Entry>
    <Entry>
        <id>2</id>
        <name>Tomi</name>
        <age>24</age>
    </Entry>
    <Entry>
        <id>3</id>
        <name>Ivan</name>
        <age>26</age>
    </Entry>
    <Entry>
        <id>4</id>
        <name>Lukas</name>
        <age>34</age>
    </Entry>
</myTable>
```

3.2.3 Formát schémy

Informácie o tom, akého dátového typu je hodnota, budeme ukladať do schém.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:element name="myTable">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0" name="Entry">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="id" type="xs:long"/>
                            <xs:element name="name" type="xs:string"/>
                            <xs:element name="age" type="xs:long"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

3.2.4 Formát uloženia BLOB-ov

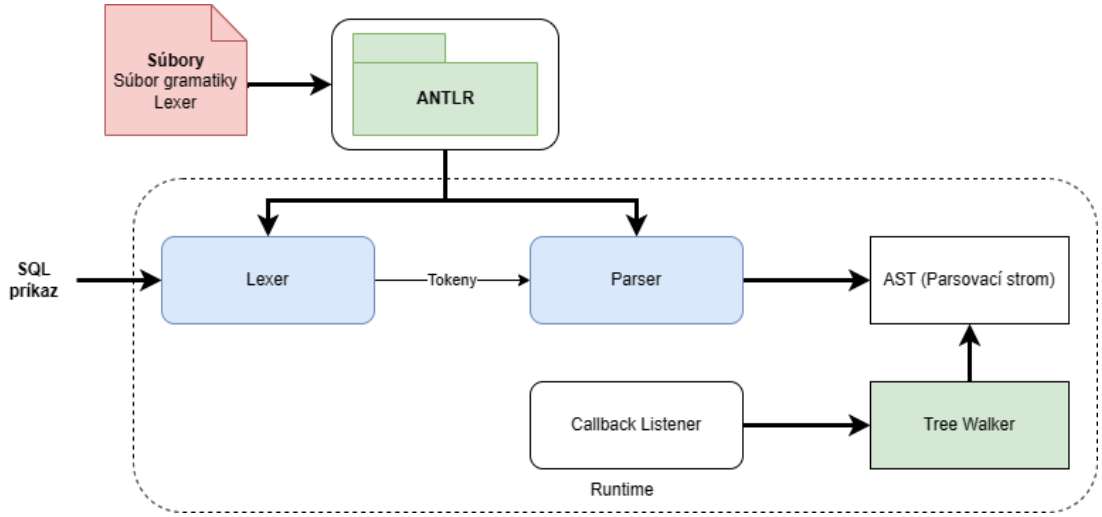
BLOB-y budú uložené v samostatných súboroch vo formáte reťazca (zakódované z polí bajtov) a v tabuľkách budú uložené iba absolútne cesty k súborom.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<myTable>
    <Entry>
        <id>1</id>
        <name>Zsolti</name>
        <age>25</age>
        <file>C:\path\to\myDatabase\blob\blob1.xml</file>
    </Entry>
</myTable>
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<blob>Y29tLmdpdGh1Yi5qZnNxbC5kcml2ZXIuY29yZS5KZnNxbERyaXZlcg==</blob>
```

3.3 SQL Príkazy

ANTLR je silný nástroj na parsovanie jazykov. Zo zadanej gramatiky ANTLR dokáže vygenerovať lexer a parser, ktoré spoločne dokážu vytvoriť strom zo vstupu (v našom prípade reťazec SQL), a poslucháča (listener), ktorý dokáže vykonávať logické operácie pri návšteve tohto stromu.Parsery generované ANTLR štandardne vytvárajú dátovú štruktúru nazývanú parsovací strom alebo syntaktický strom, ktorá zaznamenáva, ako parser rozpoznal štruktúru vstupnej vety a jej zložkových fráz.



Obrázok 2: Fungovanie ANTLR

Najprv sa spustí lexer a rozdelí vstup na tokeny. Potom sa prúd tokenov odovzdá analyzátoru, ktorý vykoná celkové spracovanie. ANTLR nám vygeneruje strom ParseTree, ktorý potom môžeme spracovať pomocou ParseTreeWalker.

Token - postupnosť znakov, ktorá predstavuje zmysluplnú časť vstupu: zvyčajne slovo alebo interpunkčné znamienko, oddelené lexikálnym analyzátorom.

Lexer – Program, ktorý vykonáva tokenizáciu (proces zoskupovania znakov do slov alebo symbolov).

Parser - Program, ktorý rozpozná jazyk, sa nazýva parser alebo syntaktický analyzátor.

Náš ovládač bude podporovať odvodenú syntax z SQLite. Inými slovami, budeme implementovať len menšiu množinu celej syntaxe a z nej extrahovať potrebné údaje. Z týchto údajov vytvoríme objekty na prenos údajov (DTO). SQLite má voľne dostupnú syntax pre ANTLR, ktorá je k dispozícii prostredníctvom githubu.

3.4 Transakcie

Náš prístup k ukladaniu údajov ukladať údaje do pamäte, kým sa neodovzdajú. To nám umožňuje uchovávať nezapisované údaje v objektoch a zapísať ich všetky naraz, keď sa zavolá metóda commit.

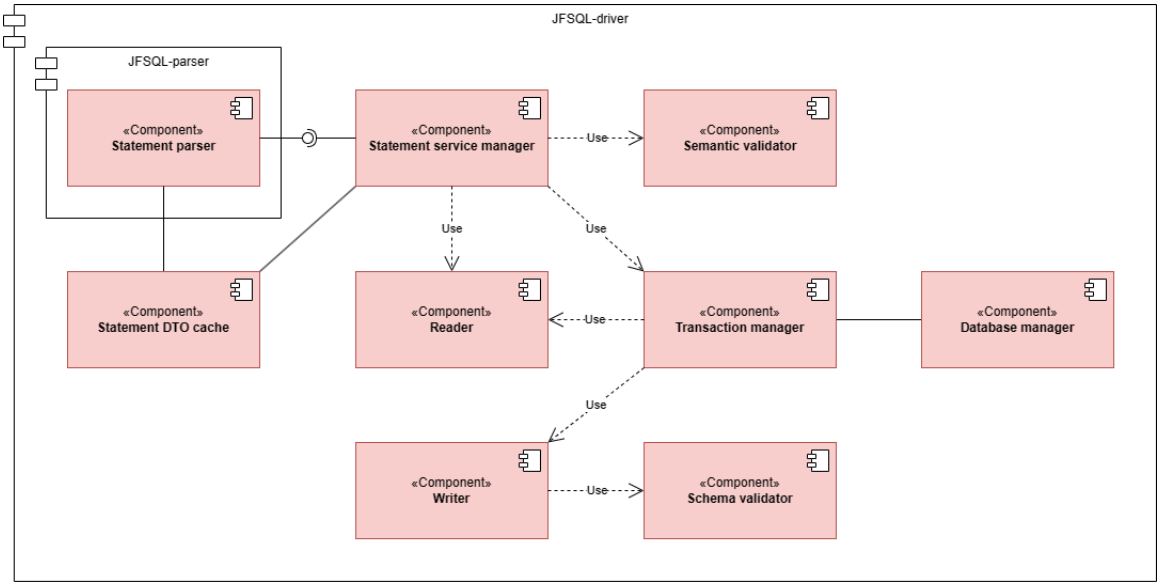
V prípade, že je potrebné použiť metódu rollback, údaje sa môžu jednoducho znovu načítať z perzistovaných súborov do pamäte. Existuje však riziko poškodenia údajov, preto sme sa rozhodli implementovať git ako nástroj na verzovanie databázy, aby sme mohli databázu vrátiť do platného bodu.

Rozhodli sme sa pre úroveň izolácie transakcií "READ COMMITTED", aby sme zabezpečili konzistenciu medzi súbežnými operáciami. Toto nemenné nastavenie zabezpečuje, že všetky údaje načítané zo systému už boli zapísané

do súborov, čím sa predchádza potenciálnym nekonzistentnostiam spôsobeným súbežnými transakciami.

Aby sme ešte viac posilnili konzistenciu nášho ovládača a zabránili konfliktom z viacvláknových transakcií, rozhodli sme sa implementovať pesimistické zamykanie ako náš preferovaný mechanizmus zamykania. Ten zahŕňa získanie zámkov na údaje pred povolením prístupu, čím sa zabezpečí, že údaje môže v danom čase meniť len jedno vlákno.

4 Implementácia



Obrázok 3: Komponenty ovládača

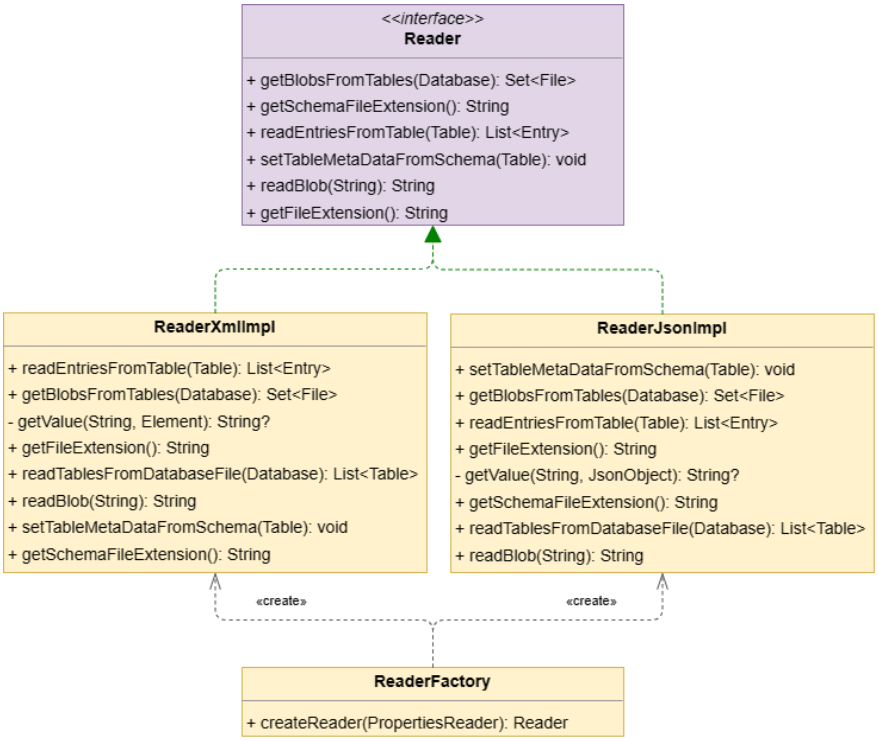
Implementáciu sme rozdelili na nasledujúce moduly:

- Statement parser - je zodpovedný za parsovanie SQL príkazov a vytváranie objektov prenosu dát z nich
- Statement DTO Cache – je zodpovedný na uloženie parsovaných SQL príkazov do pamäte
- Statement service manager – zahŕňa všetky servisy, ktoré riešia zmeny v pamäte
- Semantic validator – je zodpovedný za validáciu príkazu z hľadiska sémantickej validácie
- Reader – je zodpovedný za čítanie údajov
- Writer – je zodpovedný za zápis údajov
- Transaction manager – je zodpovedný za spracovanie tranzakcií
- Database manager – je zodpovedný za vytvorenie novej databázy a načítanie existujúcej databázy
- Schema validator – je zodpovedný za validáciu tabuľky voči schémy

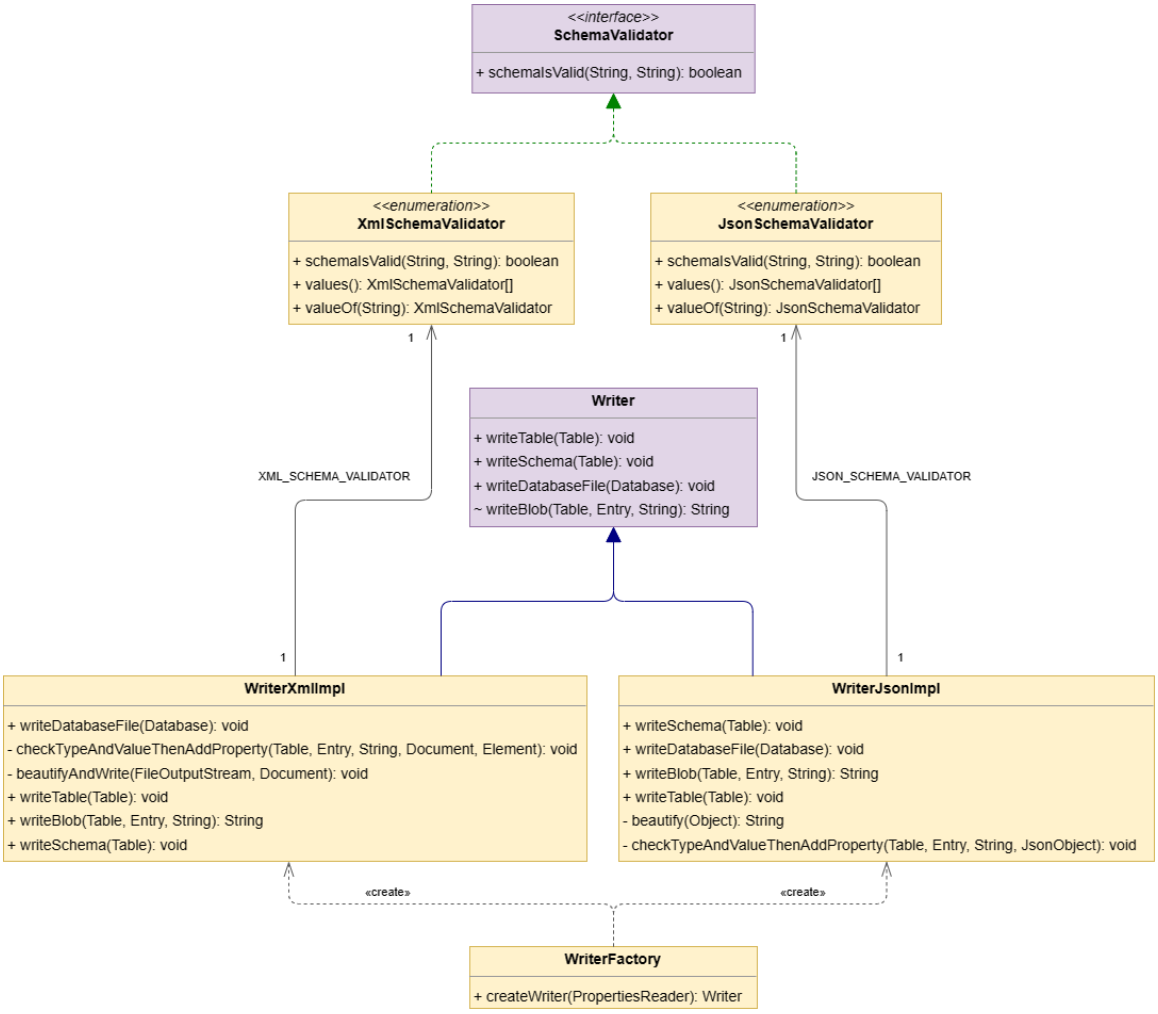
Pomocou rozdelenia sme dosiahli izoláciu modulov, nezávislosť medzi nimi.

4.1 Ľudsky čitateľný formát

Na čítanie a zápis zo súborov používame samotné triedy, ktoré implementujú rozhrania Reader a Writer. Tieto rozhrania poskytujú úplnú nezávislosť od serializačných formátov.



Obrázok 4: Diagram tried rozhrania Reader

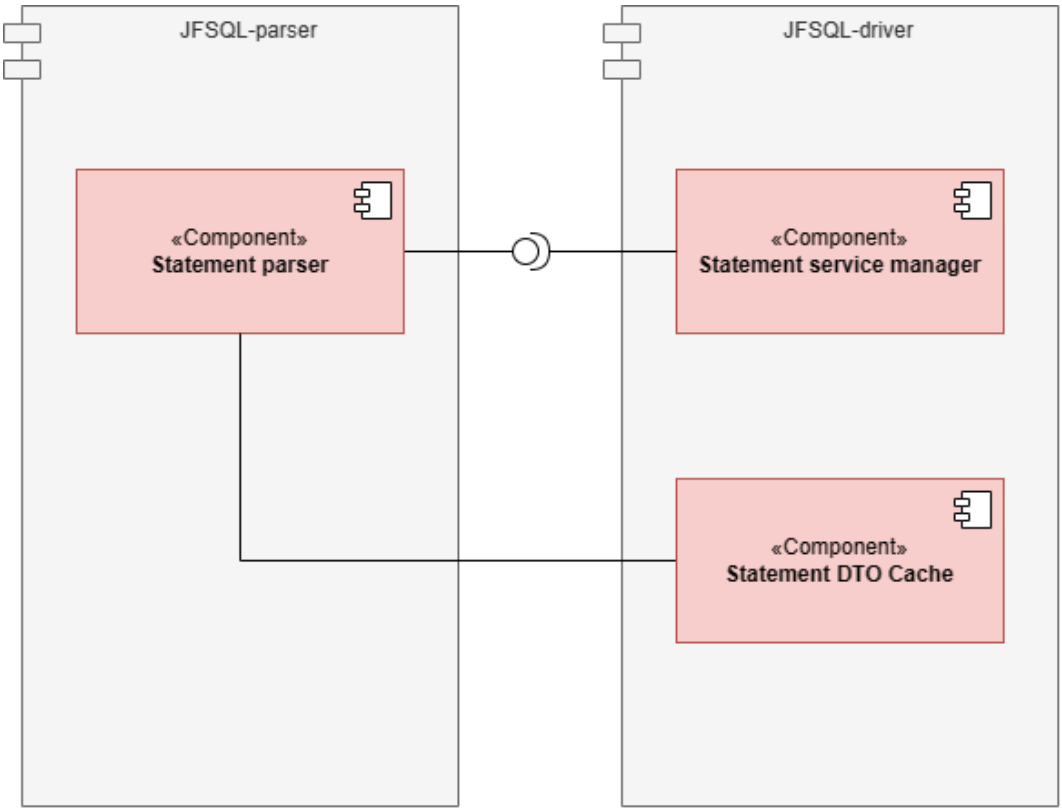


Obrázok 5: Diagram tried rozhrania Writer a SchemaValidator

Triedy typu Writer sú navrhnuté tak, aby zabezpečili integritu zapísaných údajov, a jedným z posledných opatrení je validácia schémy. Tento prístup zabezpečuje, že zapísané údaje zodpovedajú očakávanej štruktúre a spĺňajú potrebné normy. Tabuľky s formátmi JSON sa validujú pomocou schém JSON, zatiaľ čo tabuľky XML sa validujú podľa schém XSD. Schémy sa automaticky generujú pri vytváraní novej tabuľky a možno ich upravovať pomocou príkazov ALTER TABLE.

4.2 SQL Príkazy

SQL príkazy sa parsujú pomocou ANTLR4, ktorý pre každý príkaz vygeneruje parsovací strom. Na extrakciu údajov z parsovacieho stromu sa používa vzor návštevníka (Visitor pattern). Na prenos údajov medzi parserom a ovládačom sa používajú objekty prenosu údajov implementujúci rozhranie BaseStatement, ktoré poskytujú štandardizované rozhranie na výmenu údajov a pomáhajú oddeliť tieto dve moduly.

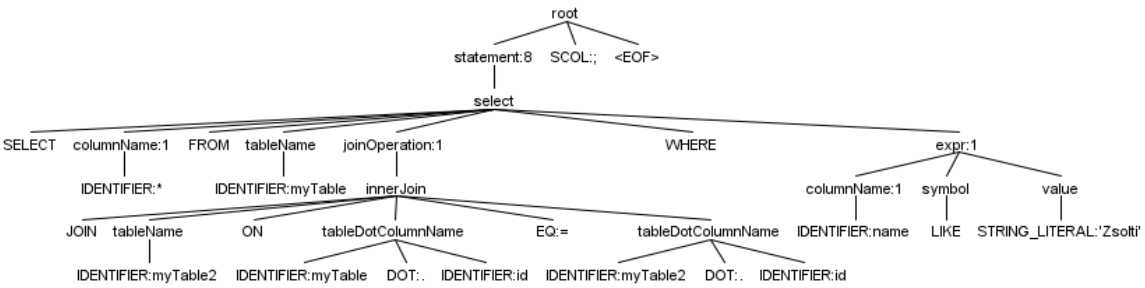


Obrázok 6: Spojenie medzi parserom a ovládačom

ANTLR4 poskytuje spôsob, ako zabezpečiť, aby boli príkazy spracované len vtedy, ak neobsahujú gramatické chyby. Treba však poznamenať, že na úrovni parsera sa nekontroluje správnosť údajov, z pohľadu sémantiky. Za predpokladu, že syntax príkazu je správna, sa potom odovzdá ovládaču na spracovanie.

Na dole uvedenom obrázku je znázornený parsovací strom vygenerovaný pomocou ANTLR4 pre reťazec:

*SELECT * FROM myTable JOIN myTable2 ON myTable.id = myTable2.id WHERE name LIKE 'Zsolti';*



Obrázok 7: Príklad parsovacieho stromu generované ANTLR4

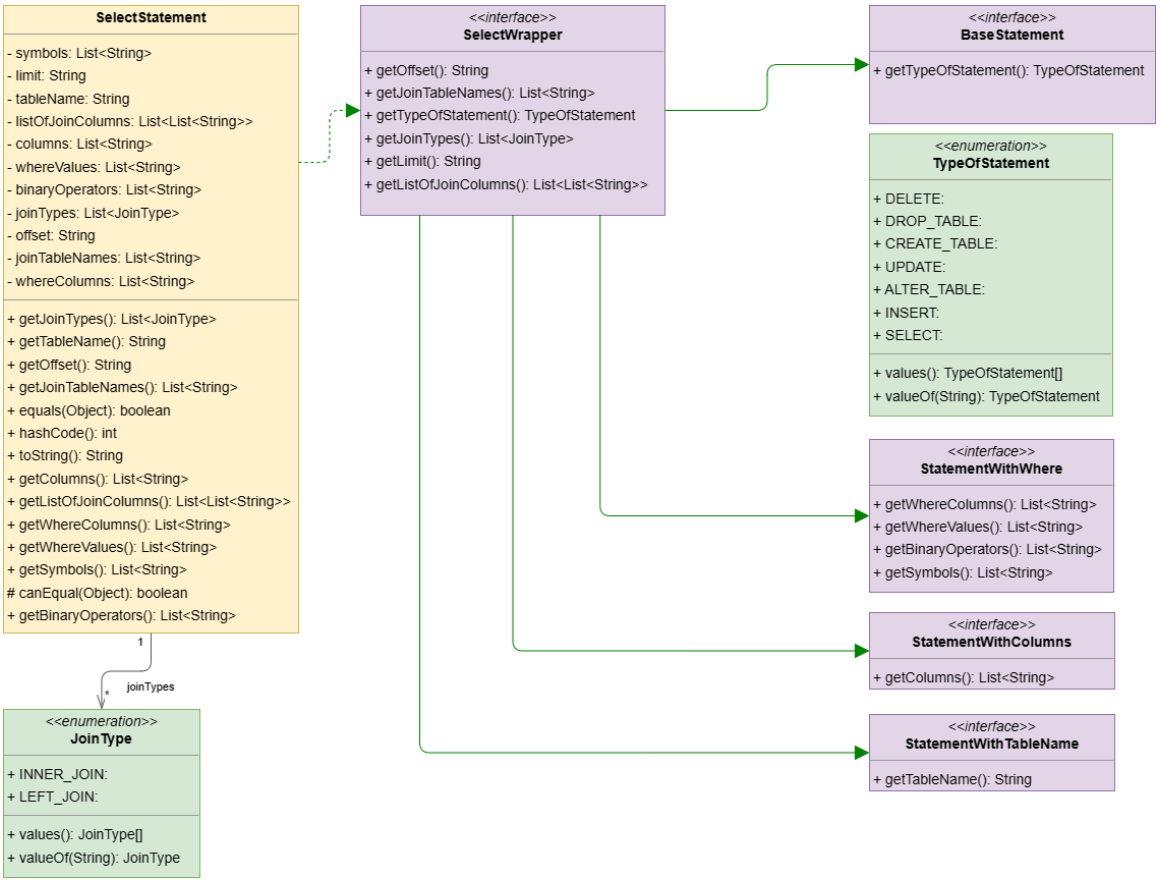
Z reťazca je následne vygenerovaný objekt na prenos údajov typu *SelectStatement*, ktorý má nasledujúce hodnoty:

tableName=myTable,
joinTableNames=[myTable2],
joinTypes=[INNER_JOIN],
columns=[*],
listOfJoinColumns=[[myTable.id, myTable2.id]],
whereColumns=[name],
whereValues=[Zsolti],
symbols=[LIKE],
binaryOperators=[]

4.2.1 Príkaz ako objekt na prenos údajov

Každý typ príkazu implementuje vlastné rozhranie Wrapper. Rozhrania Wrapper rozširujú rozhranie BaseStatement a ďalšie rozhrania, ktoré sa môžu vyskytovať alebo sa vyskytujú v danom príkaze. Rozhranie BaseStatement je spoločné rozhranie pre všetky typy príkazov a jeho metóda *getTypeOfStatement()* slúži na určenie presného typu príkazu.

Na dole uvedenom príklade si môžeme všimnúť, že trieda SelectStatement implementuje okrem rozhrania BaseStatement aj ďalšie rozhrania. Tieto rozhrania nám umožňujú obmedziť duplicity v kóde a tiež ich môžeme použiť v signatúrach metód, napríklad pri sémantickej validácii.

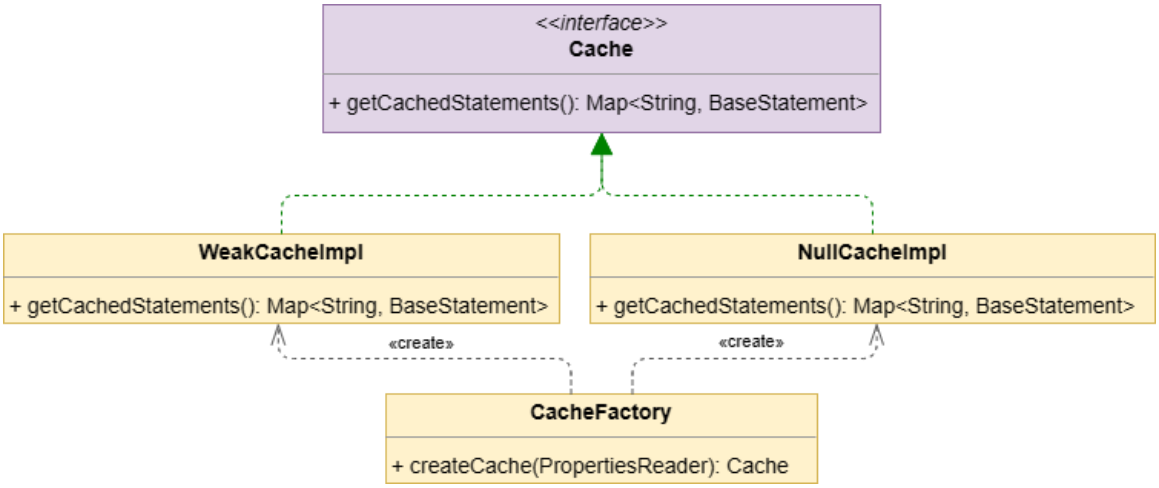


Obrázok 8: Hierarchia dedenia triedy SelectStatement

Úplný zoznam syntaxe podporovaných príkazov nájdete v prílohe v časti Podporované príkazy a ich formáty.

4.2.2 Soft parsing

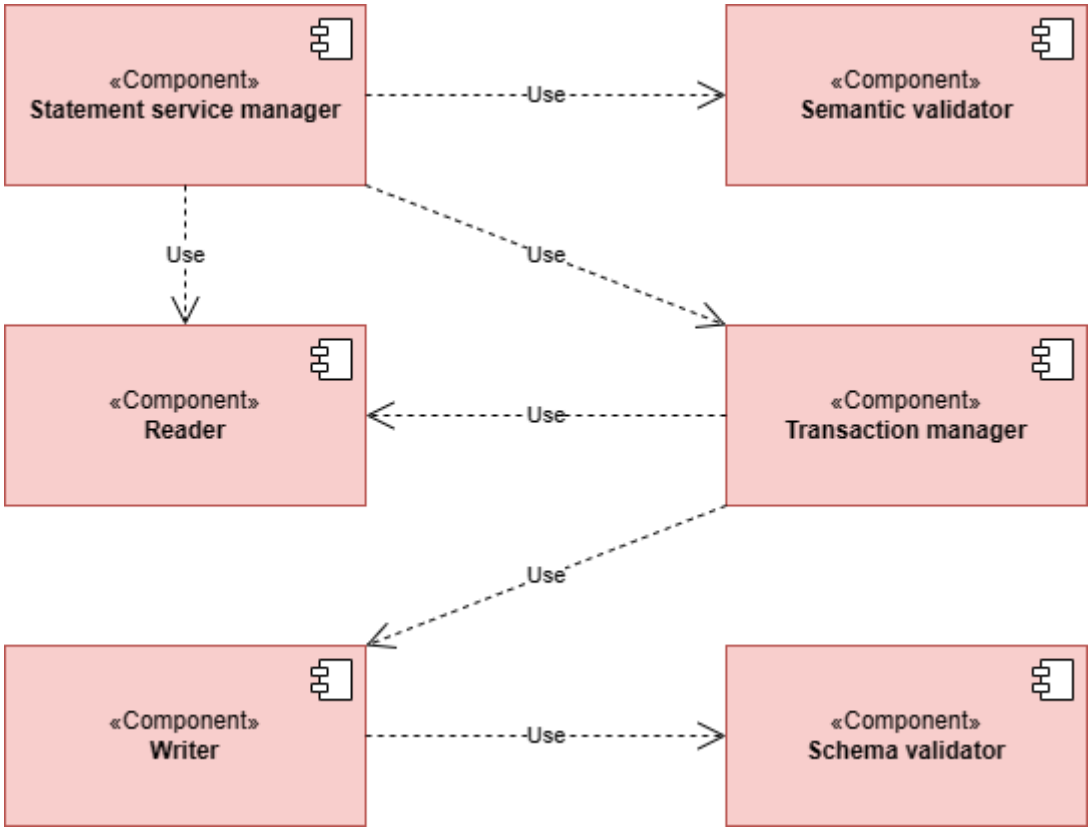
Na tento účel sme si vytvorili rozhranie Cache, ktoré je implementované dvoma triedami, NullCacheImpl a WeakCacheImpl. NullCacheImpl nebude ukladať príkazy do vyrovnávacej pamäte, takže príkazy sa budú musieť vždy parsovať. WeakCacheImpl používa na ukladanie príkazov do vyrovnávacej pamäte WeakHashMap, kde kľúče sú reťazce sql a hodnoty sú vytvorené objekty prenosu dát. V Jave je WeakHashMap trieda, ktorá implementuje rozhranie Map a poskytuje implementáciu založenú na hašovacej tabuľke so slabými kľúčmi. Je užitočná pre určité scenáre ukladania do vyrovnávacej pamäte, keď chceme ukladať dvojice kľúč-hodnota, ale nechceme, aby kľúče bránili zberu súvisiacich hodnôt do koša. [21]



Obrázok 9: Diagram tried rozhrania Cache

Vďaka ukladaniu SQL príkazov do vyrovnávacej pamäte môžeme zlepšiť celkový výkon nášho ovládača, najmä pri práci s veľkým počtom príkazov. Z hľadiska optimalizácie výkonu sme zistili, že najefektívnejšie je rozlišovať príkazy na základe citlivosti na veľkosť písmen namiesto vykonávania operácií transformácie reťazcov.

4.2.3 Spracovanie príkazov

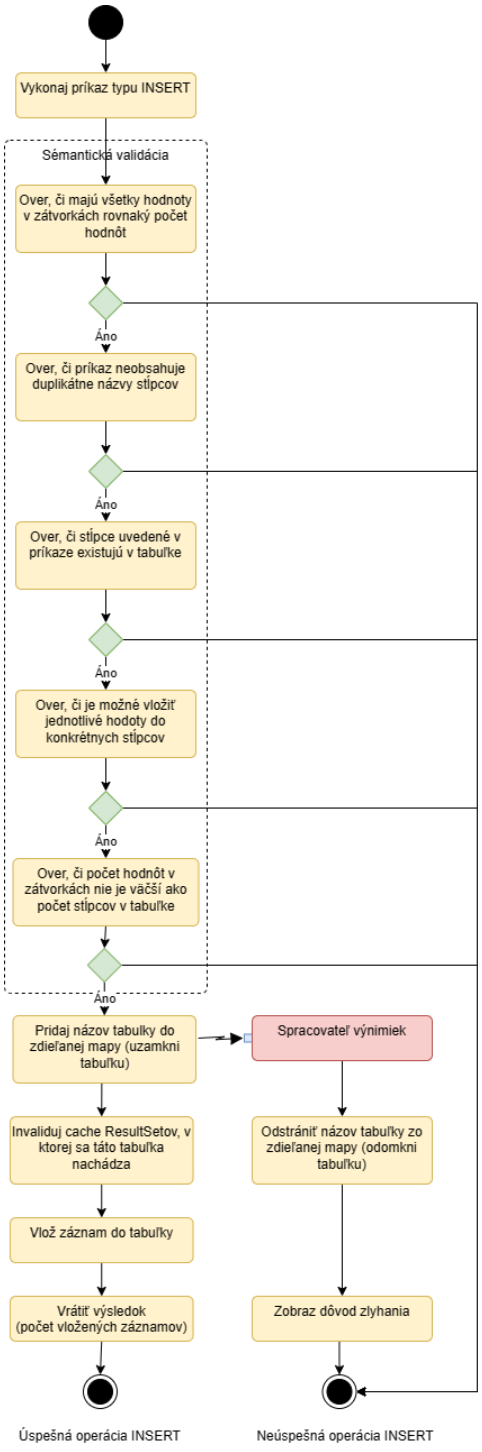


Obrázok 10: Diagram komponentov spracovania príkazu

Každý typ príkazu je spojený s príslušnou servisou, ktorá ho spracúva. Servis skúma každý atribút z hľadiska sémantickej validácie, ktorú vykonáva komponent Sémantický validátor. Ak sa príkaz považuje za neplatný, vyvolá sa výnimka *SQLException* a príkaz sa ďalej nespracúva.

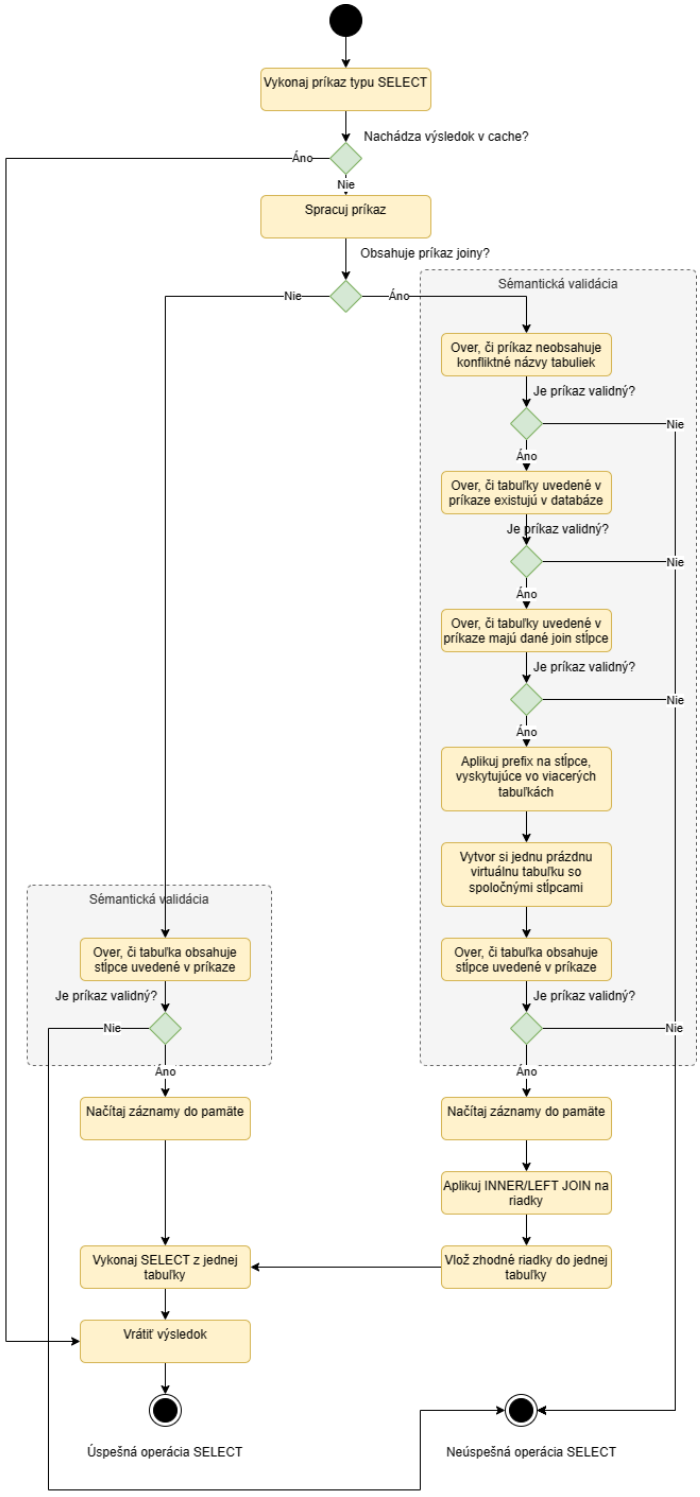
Ak ide o operáciu zápisu a sémantická validácia je úspešná, servis odovzdá objekt (tabuľku, schému alebo databázu) manažérovi transakcií, ktorý nakoniec odošle objekt na zápis. Pri zápise tabuliek sa používa aj validácia schémy. Ak je validácia neúspešná, automaticky sa zavolá metóda *rollback()*, ktorá vráti databázu do bodu, v ktorom boli údaje ešte platné. Tento prístup nám zabezpečuje úplnú integritu údajov v databáze.

4.2.3.1 Spracovanie príkazu INSERT



Obrázok 11: Diagram aktivít operácie INSERT

4.2.3.2 Spracovanie príkazu SELECT



Obrázok 12: Diagram aktivít operácie SELECT

Inner join

```
function innerJoin(t1, t2, t1JoinCol, t2JoinCol):
   commonEntries = new empty list
   hashTable = new empty hash table

   for t1e in t1.getEntries():
      key = value of column t1JoinCol in t1e
      if key is not in hashTable:
         hashTable[key] = new empty list
      add t1e to the list in hashTable[key]

   for t2e in t2.getEntries():
      key = value of column t2JoinCol in t2e
      if key is in hashTable:
         matchedEntries = list in hashTable[key]
         for t1e in matchedEntries:
            commonColumnsAndValues = new empty hash table
            copy columns and values from t1e to commonColumnsAndValues
            copy columns and values from t2e to commonColumnsAndValues
            add a new entry with commonColumnsAndValues to commonEntries

   return commonEntries
```

Vyššie uvedený pseudokód opisuje inner join medzi dvoma tabuľkami t1 a t2 na základe zadaného join stĺpca z každej tabuľky (t1JoinCol a t2JoinCol). Metóda najprv vytvorí hashovaciu tabuľku pomocou join stĺpca ľavej tabuľky (t1JoinCol).

Každý záznam v ľavej tabuľke sa pridá do hašovacej tabuľky s použitím hodnoty join stĺpca daného záznamu ako kľúča. V hašovacej tabuľke môže byť viac záznamov s rovnakou hodnotou stĺpca join, pretože hodnoty hašovacej tabuľky sú uložené v zozname.

Ďalej metóda skúma pravú tabuľku (t2) pomocou stĺpca join pravej tabuľky. Ak hodnota stĺpca join pravej tabuľky existuje v tabuľke hash ako kľúč, potom sa záznamy tabuľky hash s rovnakou hodnotou kľúča považujú za zhodné so záznamom pravej tabuľky. Spoločné stĺpce a hodnoty z dvoch zhodných záznamov (t1 a t2) sa zlúčia do jednej mapy a zlúčená mapa sa pridá do zoznamu spoločných záznamov.

Nakoniec metóda vráti zoznam spoločných záznamov nájdených v oboch tabuľkách.

Left join

```
function leftJoin(t1, t2, t1JoinCol, t2JoinCol):
    joinedEntries = new empty list
    hashTable = new empty hash table

    for t2e in t2.getEntries():
        key = value of column t2JoinCol in t2e
        if key is not in hashTable:
            hashTable[key] = new empty list
        add t2e to the list in hashTable[key]

    for t1e in t1.getEntries():
        key = value of column t1JoinCol in t1e
        if key is in hashTable:
            matchedEntries = list in hashTable[key]
            for t2e in matchedEntries:
                joinedColumnsAndValues = new empty hash table
                copy columns and values from t1e to joinedColumnsAndValues
                copy columns and values from t2e to joinedColumnsAndValues
                add a new entry with joinedColumnsAndValues to joinedEntries
        else:
            joinedColumnsAndValues = new empty hash table
            copy columns and values from t1e to joinedColumnsAndValues
            for columnName in t2.getColumnsAndTypes().keys():
                add a new column with the name columnName and a null value to joinedColumnsAndValues
            add a new entry with joinedColumnsAndValues to joinedEntries

    return joinedEntries
```

Vyššie uvedený pseudokód opisuje left join medzi dvoma tabuľkami t1 a t2 na základe zadaného join stĺpca z každej tabuľky (t1JoinCol a t2JoinCol).

Každý záznam v pravej tabuľke sa pridá do hašovacej tabuľky s použitím hodnoty join stĺpca daného záznamu ako kľúča.

Metóda potom iteruje cez každý záznam v t1 a skúma hash tabuľku, aby našla záznamy v t2, ktoré sa zhodujú v join stĺpci. Ak existujú zhodné záznamy v t2, metóda vytvorí nový objekt Entry, ktorý kombinuje stĺpce a hodnoty z oboch tabuliek, a pridá ho do zoznamu spojených záznamov. Ak v t2 nie sú žiadne zhodné záznamy, metóda vytvorí nový objekt Entry, ktorý obsahuje stĺpce a hodnoty z t1, ako aj nulové hodnoty stĺpcov z t2, a pridá ho do zoznamu spojených záznamov.

Nakoniec metóda vráti zoznam spojených záznamov.

Cachovanie výsledkov

Spracovaním údajov v pamäti môžeme výrazne urýchliť vykonávanie operácií SELECT. Na dosiahnutie tohto cieľa sme implementovali vláknovo bezpečnú štruktúru WeakHashMap, v ktorej objekty prenosu dát vytvorené z príkazov slúžia ako kľúče a príslušné súbory ResultSet, ktoré predstavujú výsledky vykonania príkazov, slúžia ako hodnoty. Tento prístup založený na vyrovnávacej pamäti viedol k pozoruhodnému zlepšeniu výkonu nášho ovládača.

Na zabezpečenie konzistentnosti údajov bolo nevyhnutné odstrániť z vyrovnávacej pamäte všetky záznamy, ktoré sú ovplyvnené operáciami zápisu do danej tabuľky. Tento prístup zabezpečuje, že údaje, ku ktorým sa pristupuje prostredníctvom operácií SELECT, sú vždy aktuálne, aj keď sa súčasne vykonávajú operácie zápisu.

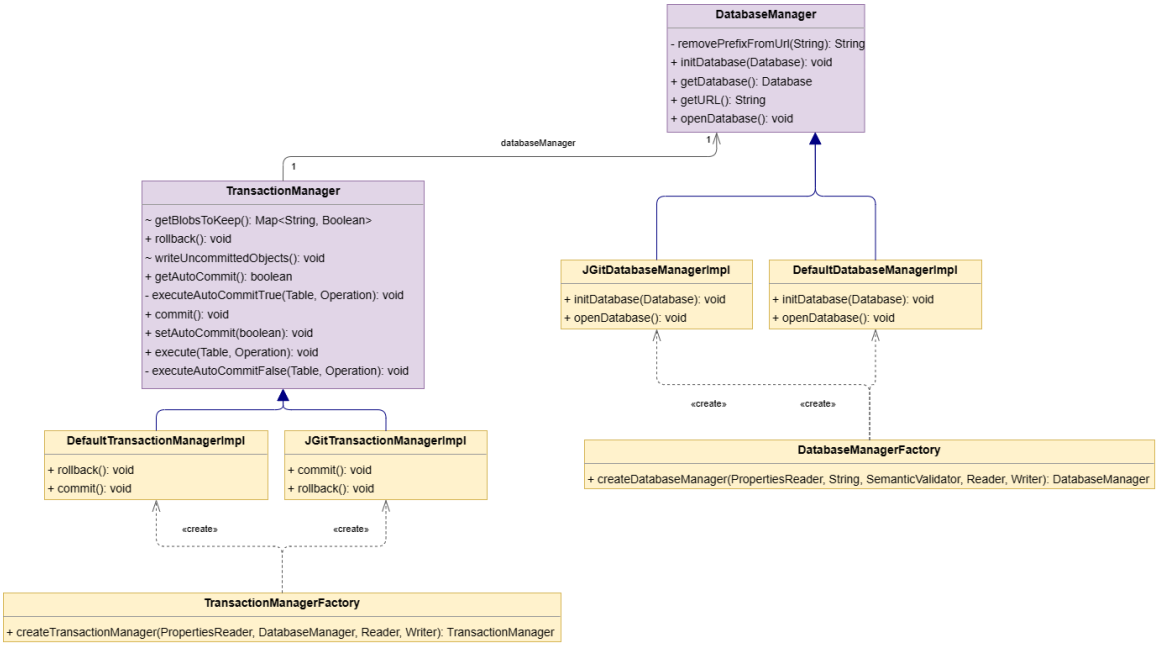
4.3 Transakcie

Na spracovanie transakcií ponúkame dve konfigurácie: predvolenú a JGit. JGit je implementácia systému Git založená na jazyku Java, ktorá dočasne ukladá zmeny v úložisku objektových súborov, kým sa neodovzdajú na disk ako súčasť transakcie. Počas nášho testovania rôznych scenárov sme zistili, že hoci je Git teoreticky vhodný na verzionovanie databáz, v praxi môže mať výrazný vplyv na výkon, najmä pri spracovaní veľkých tabuliek a častých odovzdávaniach s nastavením autoCommit na true.

Na riešenie tohto problému s výkonom sme vytvorili dve vzájomne závislé implementačné triedy, ktoré zodpovedajú našim rozhraniam TransactionManager a DatabaseManager. Ak sa používa JGit, trieda JGitDatabaseManagerImpl bude pri vytváraní databázy inicializovať úložisko pomocou príkazu "git init". Je dôležité poznamenať, že hoci naša predvolená

implementácia môže ponúknuť lepší výkon, neposkytuje rovnaké možnosti verziovania ako JGit.

Preto ak je verzovanie databázy kritickou požiadavkou, odporúčame pre optimálnu funkčnosť používať JGit. Výber medzi týmito dvoma konfiguráciami v konečnom dôsledku závisí od priorít a požiadaviek používateľa, pričom JGit poskytuje lepšie možnosti verzovania a predvolená konfigurácia ponúka lepší výkon.



Obrázok 13: Diagramy tried rozhraní TransactionManager a DatabaseManager

4.3.1 Pesimistické uzamykanie

Na implementáciu pesimistického uzamykania sme vyvinuli mechanizmus, ktorý kontroluje, či k danému databázovému súboru, tabuľke alebo schéme práve pristupuje iné vlákno. Pokiaľ áno, objekt sa považuje za uzamknutý a aktuálne vlákno ho nemôže modifikovať, kým sa zámok neuvoľní.

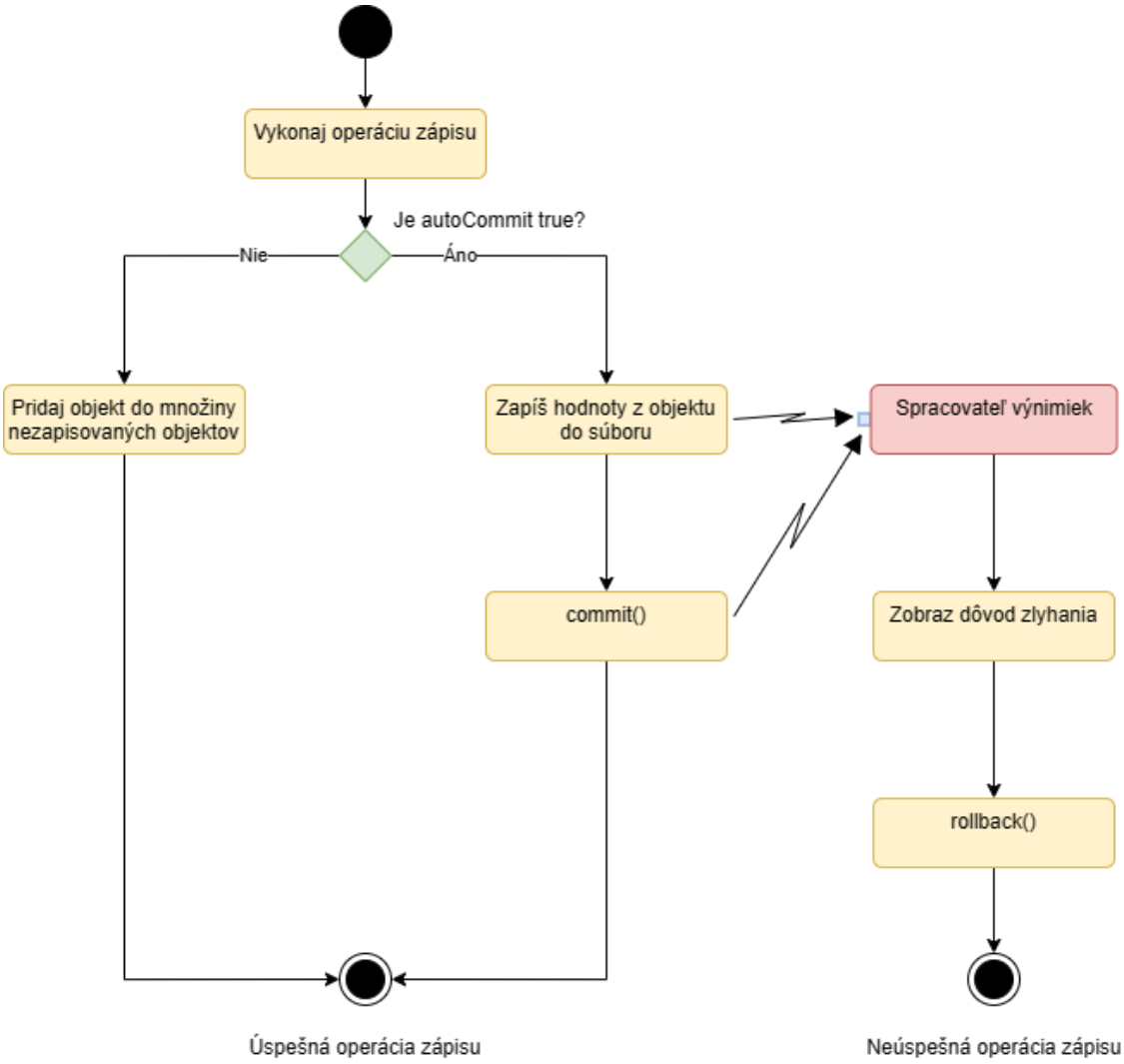
Atribút *OBJECT_NAME_TO_THREAD_ID_MAP* triedy *TransactionManager* predstavuje statickú mapu, zdieľané medzi vláknami, kde kľúče mená objektov a hodnoty sú id vlákien, ktoré daný objekt práve používajú.

Ak meno objektu sa v mape nenachádza, tak objekt sa pridá do množiny nezapisovaných objektov a jeho meno sa vloží do mapy, čím sa objekt uzamkne a stane sa neprístupným pre iné vlákna, kým sa transakcia neukončí.

Ak je objekt v zdieľanej mape, znamená to, že iné vlákno už zablokovalo zdroj. V tomto prípade sa záznamy v mape, ktoré boli vložené tým istým vláknom, ktoré uzamklo prostriedok, odstránia, aby sa objekt uvoľnil a umožnilo sa ostatným vláknam v prípade potreby ho upraviť.

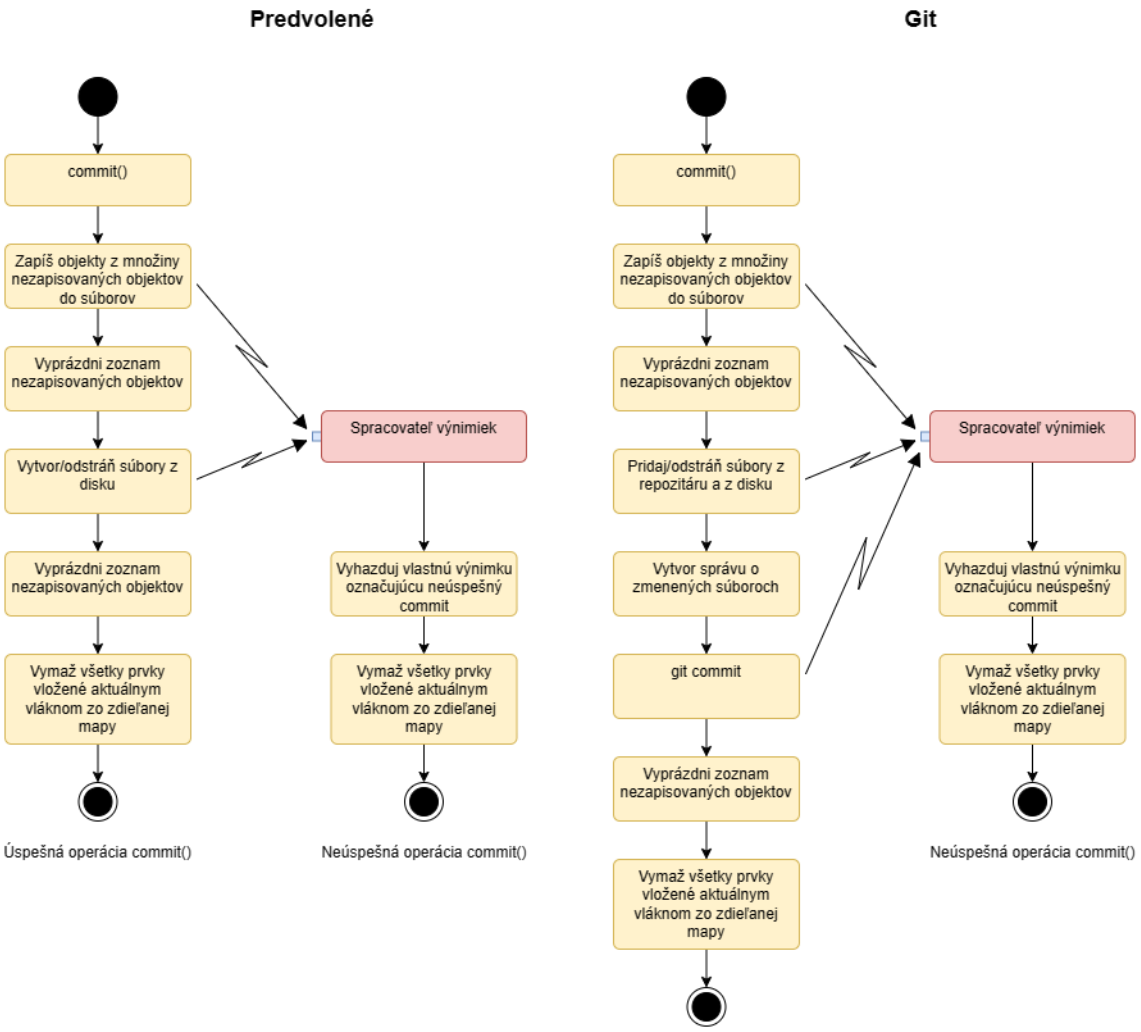
Aby sa zabránilo pretekom a zabezpečilo sa, že vlákno počká, kým sa prostriedok uvoľní, vyvolá sa výnimka *PessimisticLockException*, ktorá informuje aktuálne vlákno, že prostriedok je už uzamknutý a nemožno ho modifikovať, kým sa zámok neuvoľní. V tomto prípade bolo dôležité odstrániť zo zdieľanej mapy aj všetky záznamy, ktoré boli vložené týmto vláknom.

4.3.2 Priebeh operácie zápisu



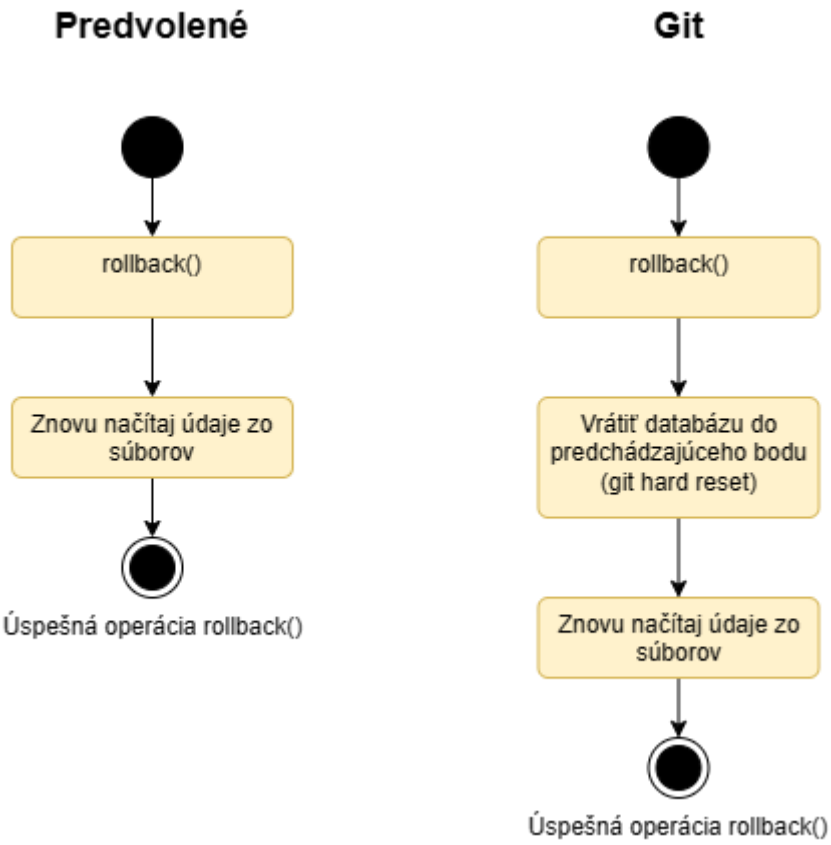
Obrázok 14: Aktivity diagram operácie zápisu

4.3.3 Commit



Obrázok 15: Aktivity diagram operácie commit

4.3.4 Rollback



Obrázok 16: Aktivity diagram operácie rollback

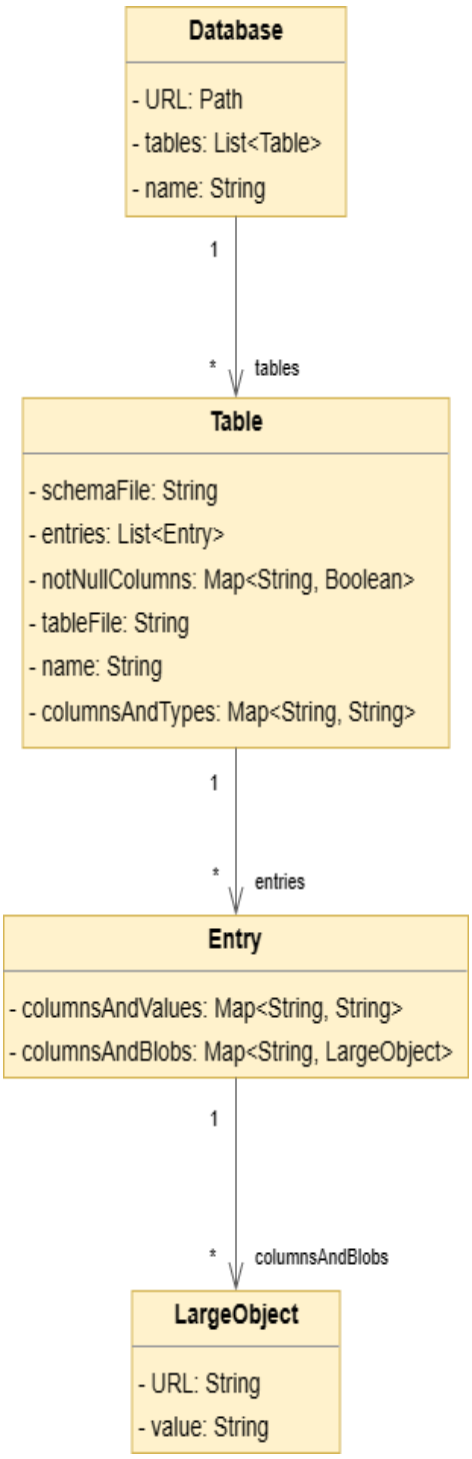
4.4 Uloženie BLOBov

Pri ukladaní BLOBov v ovládači používame formát Base64. Vzhľadom na ich potenciálne veľkú veľkosť ich do pamäte načítavame len v prípade potreby.

To znamená, že aj keď príkaz SELECT obsahuje stĺpec, ktorý má dátový typ BLOB, hodnota BLOBu nebude načítané do pamäte, iba ak používateľ ho explicitne nepožiada.

Tento prístup je užitočný najmä pri práci s veľkým množstvom údajov, pretože pomáha optimalizovať využitie zdrojov a zlepšiť výkon systému.

4.5 Vzťahy medzi hlavnými triedami



Obrázok 17: Vzťahy medzi triedami v ovládači

Trieda *Database* predstavuje databázu, ktorá môže mať 0 až n objektov Table. Atribút *URL* predstavuje absolútnu cestu k databázovému súboru.

Trieda *Table* predstavuje tabuľku, ktorá môže mať 0 až n objektov *Entry*. Atribút *name* predstavuje názov tabuľky. Atribút *tableFile* a *schemaFile* obsahujú absolútne cesty k tabuľke. Mapa *columnsAndTypes*, hovorí o tom, aký dátový typ má daný stĺpec. Mapa *notNullColumns* určí, ktorý stĺpec môže mať nulovú hodnotu. Atribút schemaFile obsahuje absolútnu cestu k

Trieda *Entry* predstavuje jeden riadok v tabuľke. Atribút *columnsAndValues* je mapa reprezentujúca hodnoty mapované na stĺpce v riadku.

Atribút *columnsAndBlobs* je mapa reprezentajúca BLOBy mapované na stĺpce v riadku.

Trieda *LargeObject* predstavuje BLOBy. Atribút *url* je absolútna cesta k súboru. Atribút *value* je hodnota BLOBu vo formáte Base64.

4.6 Konfigurácia ovládača

Ovládač ponúka celý rad konfigurovateľných možností, ktoré možno prispôsobiť konkrétnym požiadavkám.

Trieda *DriverManager* rozhrania JDBC API poskytuje niekoľko metód na pripojenie k databázam vrátane metódu, ktorá umožňuje pri pripojení k zdroju údajov vložiť objekt *Property*. Táto metóda volá metódu *connect()*, ktorá je implementovaná v našom programe, v rámci rozhrania Driver. Ak je objekt *Property* prítomný, náš ovládač ho potom spracuje a extrahuje z neho páry kľúč-hodnota. Ak je parsovanie úspešné, pomocou Factory pattern sú vytvorené rôzne triedy na základe vstupu. Ak objekt *Property* nebol prítomný alebo ho nebolo možné spracovať, použijeme vopred definované predvolené hodnoty.

V predvolenom nastavení je ovládač nakonfigurovaný tak, aby používal formát JSON na ukladanie a prístup k súborom, ukladal príkazy SQL do vyrovnávacej pamäte, overoval súbory tabuliek podľa schém a nevyužíval JGit ako správcu transakcií.

Ak chceme zmeniť predvolenú konfiguráciu, môžeme pri pripájaní k databáze priložiť objekt *Property*. Napríklad môžeme nastaviť vlastnosť "persistence" na "xml", aby sme sa prepli na formát XML. Medzi ďalšie konfigurovateľné možnosti patrí ukladanie príkazov do vyrovnávacej pamäte, overovanie schémy a verziovanie transakcií. Na ilustráciu je v nasledujúcom kóde znázornené, ako zmeniť formát perzistencie na XML:

```
final Properties properties = new Properties();
properties.setProperty("persistence", "xml");
final Connection connection =
DriverManager.getConnection("jdbc:jfsql:C:path/to/myDatabase, properties);
```

Nastavením "persistence" na "xml" bude teraz ovládač používať formát XML na čítanie a zápis súborov. Úplný zoznam vlastností konfigurácie nájdete v prílohe v časti Podporované konfigurácie.

5 Overenie riešenia

Na overenie funkčnosti ovládača sme použili niekoľko metód. Najprv sme pre každú zložku nášho ovládača vytvorili unit testy, ktoré testovali správnosť danej časti kódu. Testy boli napísané v testovacom frameworku JUnit5 v kombinácii s Mockito.

Ako integračný test sme vytvorili jednoduchú aplikáciu, do ktorej sme importovali náš ovládač. V aplikácii môžeme náš ovládač nahradiť ovládačom SQLite, pretože použitá syntax je kompatibilná s oboma ovládačmi. Potom sme porovnali náš ovládač z hľadiska výkonu s ovládačom SQLite, ktorý meral celkový čas behu operácií CRUD.

5.1 Unit testy

5.1.1 Testovanie parsera

Pri testovaní parsera sme testovali, či príkazy vytvárajú zodpovedajúce objekty a či sú atribúty nastavené na správne hodnoty.

Testy na overenie funkčnosti parsera sa nachádzajú v balíkoch *com.github.jfsql.parser.core* a *com.github.jfsql.parser.dto.*

5.1.2 Testovanie čitateľa a zapisovateľa

Testovanie tried typu Reader zahŕňalo vytvorenie súborov, ktoré napodobňujú súbory, ktoré by ovládač zvyčajne používal počas behu. Tým sa zabezpečilo, že trieda Reader dokáže efektívne extrahovať príslušné údaje z týchto súborov a produkovať presné a očakávané výsledky. Počas testovania sme vyvolali každú z príslušných metód triedy Reader a odovzdali sme vygenerované testovacie údaje. Potom sme porovnali výstup vygenerovaný každou metódou s očakávaným výstupom, ktorý sme definovali ručne.

Testovanie tried typu Writer prebiehalo podobným spôsobom. Najprv sme ručne vytvorili objekty prenosu údajov, ktoré sa majú zapisovať do súborov triedou Writer.

Následne sme použili triedu Writer na zápis týchto objektov do súborov a porovnali sme obsah vytvorených súborov s obsahom vopred vytvorených súborov.

Testy na overenie funkčnosti čitateľa a zapisovateľa sa nachádzajú v balíku *com.github.jfsql.driver.persistence.*

5.1.3 Testovanie príkazových servisov a spoločných metód

Vzhľadom na to, že každý servis, ktorý rieši nejaký príkaz môže závisieť od mnohých iných komponentov, bolo potrebné izolovať testovanú službu pomocou mockingu a injektovania jej závislostí zvonku. Na dosiahnutie tohto cieľa sme využili testovací rámec Mockito, ktorý nám umožnil vytvoriť mock objekty a overiť, ktoré metódy ktorých závislostí boli počas testu volané. To nám poskytlo komplexný prehľad o celom toku služby a pomohlo nám identifikovať prípadné problémy alebo chyby. Simulovaním rôznych scenárov, napríklad keď má servis uspieť alebo zlyhať a vyhodiť výnimky, sme mohli dôkladne otestovať funkčnosť každého servisu a zabezpečiť, aby spĺňala očakávané správanie. Tento prístup nám tiež umožnil ľahko pridávať nové testovacie prípady podľa potreby a refaktorovať kód bez toho, aby sme porušili existujúce testy.

Testy na overenie funkčnosti čitateľa a zapisovateľa sa nachádzajú v balíku *com.github.jfsql.driver.services.*

Napriek tomu, že vo servisoch spoločné metódy boli iba mockované, vytvorili sme tiež testy na overenie funkčnosti spoločných utility metód. Tie testy sa nachádzajú v balíku *com.github.jfsql.driver.util.*

5.1.4 Testovanie pesimistického uzamknutia a transakcií

Na testovania ovládača vo viacvláknovom prostredí a so simultánnymi transakciami sme pripravili viaceré testy, ktoré testovali rôzne situácie. Z každého testu je vytvorená verzia kde sú operácie vykonané v jednej transakcií, a verzia kde každá operácia je zapisovaná.

- Desať vlákien sa pokúšajú zmeniť databázový súbor súčasne vytvorením tabuliek tabuliek naraz. Príkaz iba jedného vlákna bude úspešný, druhé dostane výnimku *PessimisticLockException*.
- Desať vlákien sa pokúšajú vložiť do jednej tabuľky záznamov naraz. 9 z 10 vlákien bude zastavených kvôli *PessimisticLockException* a iba vloženie jedného vlákna bude perzistovaný a zapisovaný.
- Desať vlákien sa pokúšajú vložiť do rôznych tabuliek záznamov naraz. Neočakávajú sa žiadne výnimky. Každé vlákno vkladá do rôznych tabuliek, 10 vlákien - 10 tabuliek

Tieto sa nachádzajú v balíku *com.github.jfsql.driver.multithreading*.

5.1.5 Testovanie ovládača ako JDBC driver

Tieto testy sa zameriavali najmä na testovanie ovládača ako celok a jeho súladu s rozhraním JDBC API. Išlo o testovanie jednotlivých príkazov a funkcií poskytovaných rozhraním JDBC.

Testy pokrývali celý rad scenárov, napríklad situácie, ktoré by mohli viesť k vyvolaniu výnimky, kontrolu návratových hodnôt metód *execute*, *executeUpdate* a *executeQuery* a zabezpečenie perzistencie správnych údajov do súborov. Okrem toho sa testovali funkcie *commit()* a *rollback*(), hoci druhá z nich bola podporovaná len v konfiguráciách s JGit. Tieto testy boli navrhnuté tak, aby sa spustili v štyroch rôznych konfiguráciách: json bez jgit, json s jgitom, xml bez jgit, xml s jgitom.

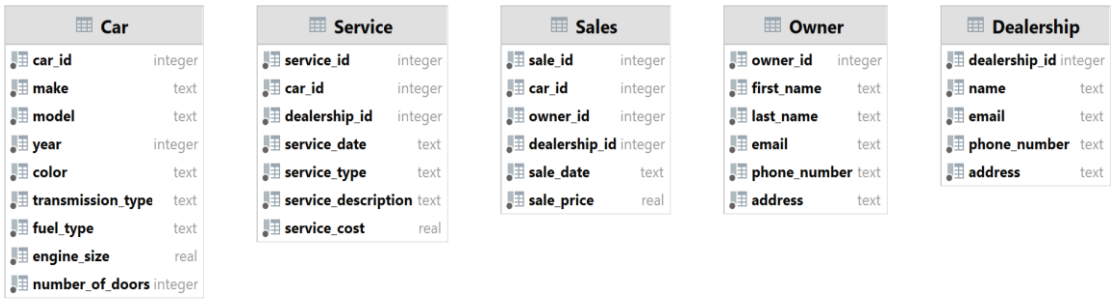
Testy na overenie funkčnosti ovládača cez JDBC API sa nachádzajú v balíku *com.github.jfsql.driver.jdbc.*

5.2 Integračný test

Na tento účel sme vytvorili samostatnú demo aplikáciu, v ktorej môžeme vykonávať jednoduché CRUD operácie. V aplikácii stačí zmeniť reťazec pripojenia, ak chceme zmeniť ovládač databázy na SQLite alebo naopak.

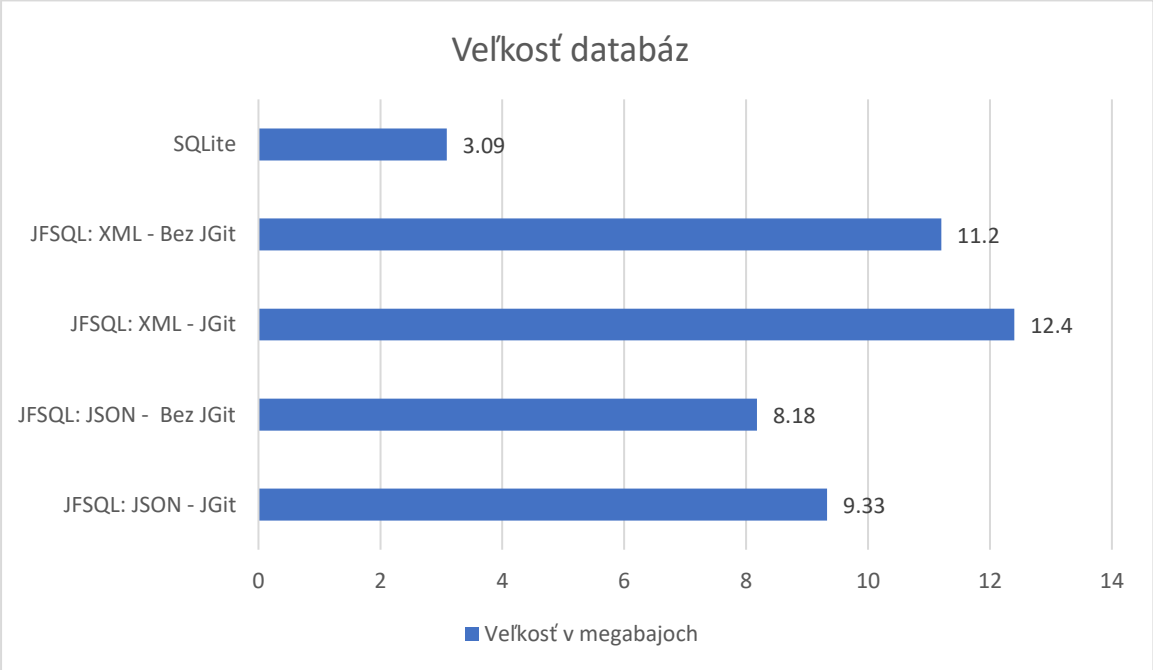
5.3 Porovnávanie ovládača s SQLite

Na tento účel sme vytvorili samostatný projekt, v ktorom sme spustili tie isté príkazy zo skriptov pre obe ovládače. Vygenerovali sme databázu, ktorá má 5 tabuliek a spolu tabuľky majú 33 020 riadkov:

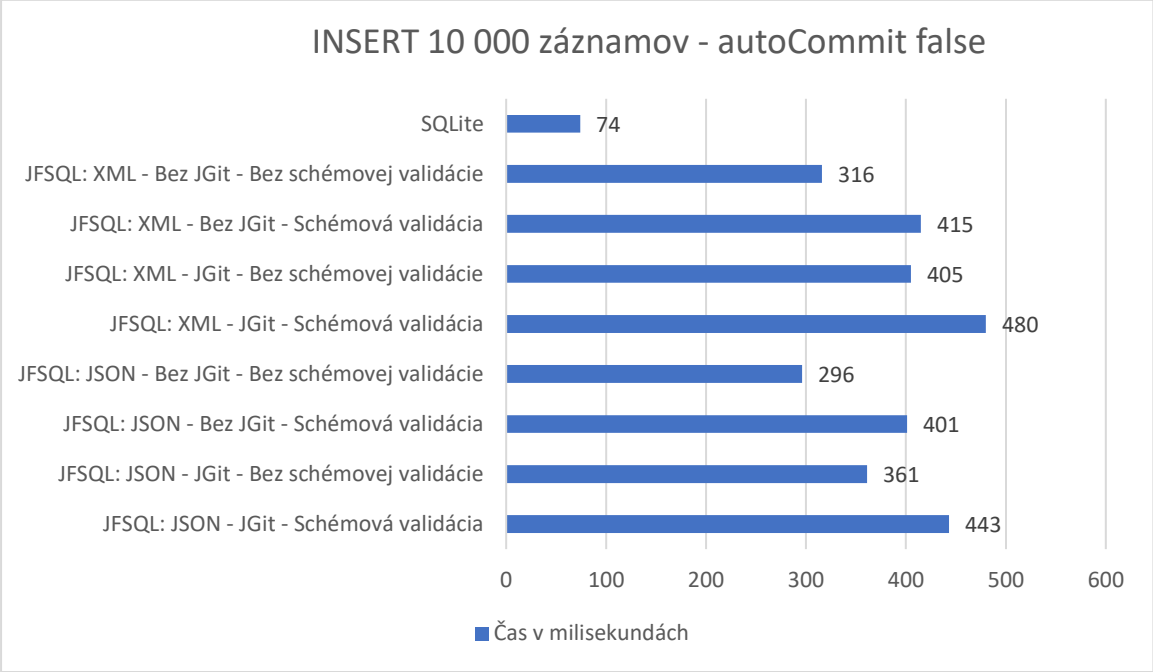


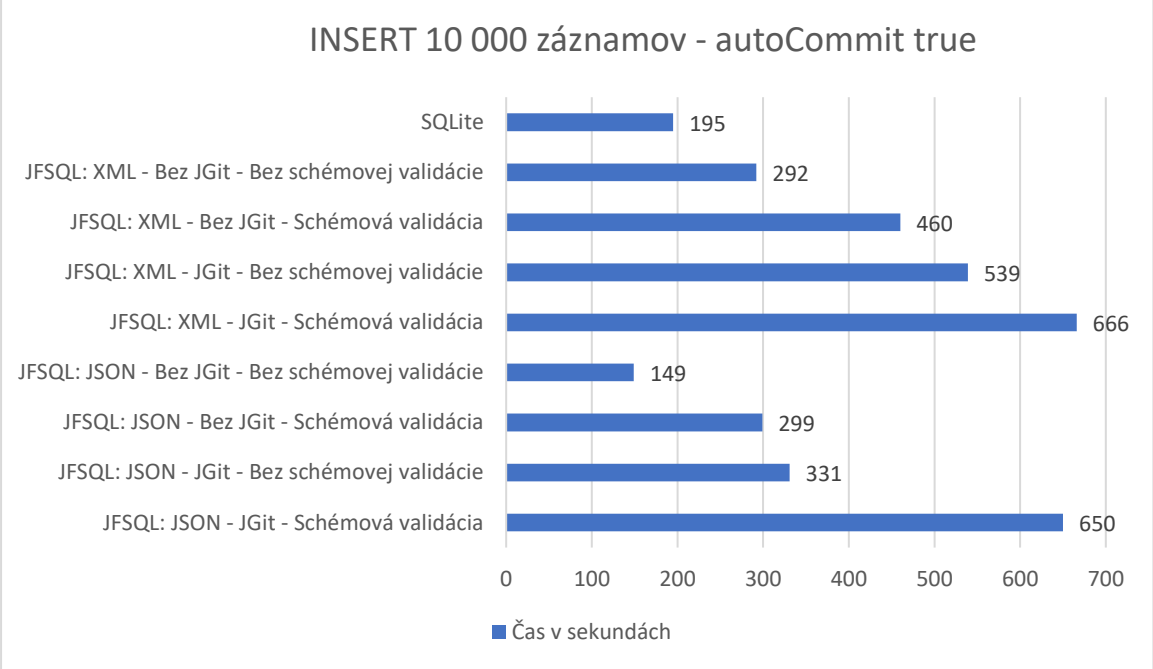
Obrázok 18: Schéma ukážkovej databázy

5.3.1 Veľkosť databáz

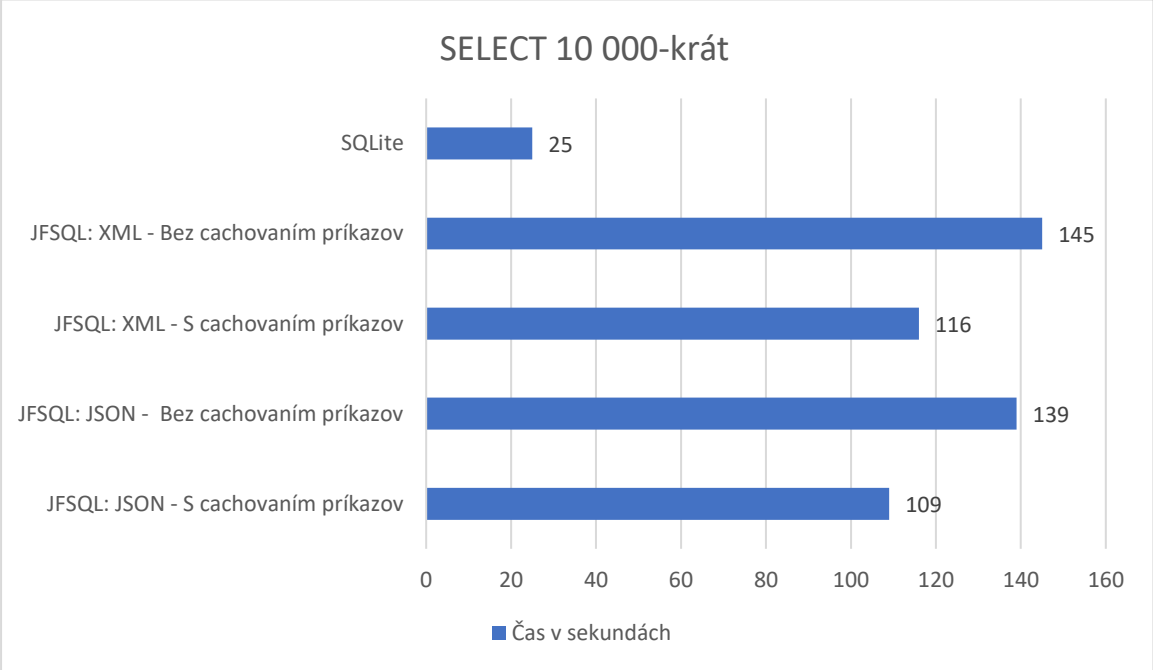


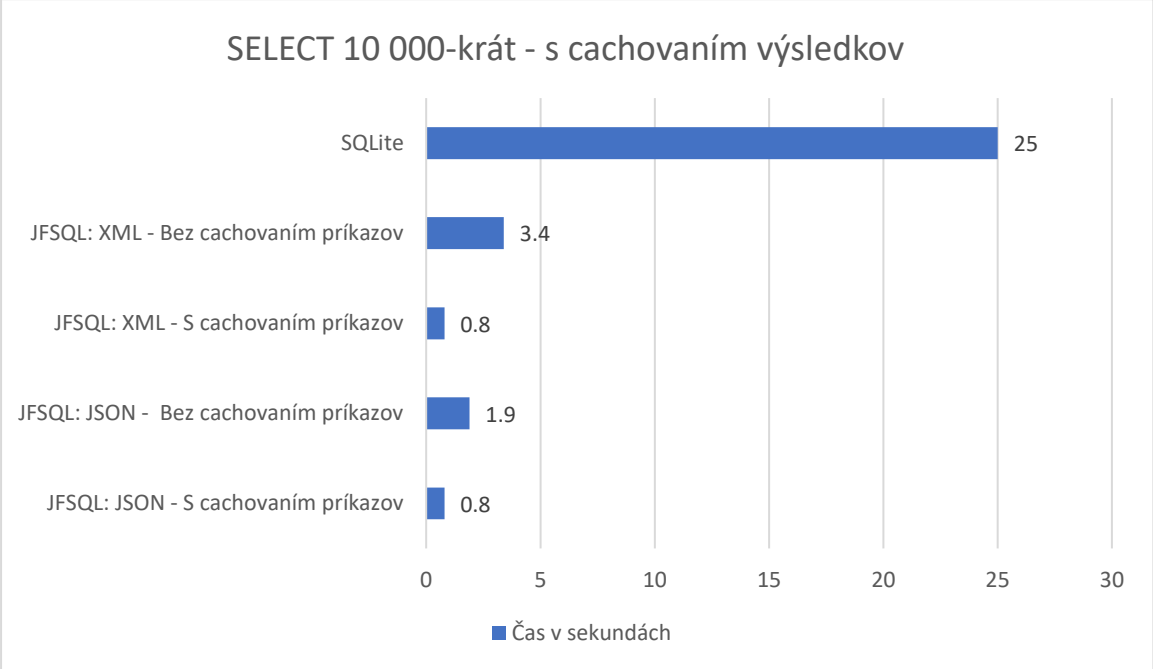
5.3.2 INSERT 10 000-krát



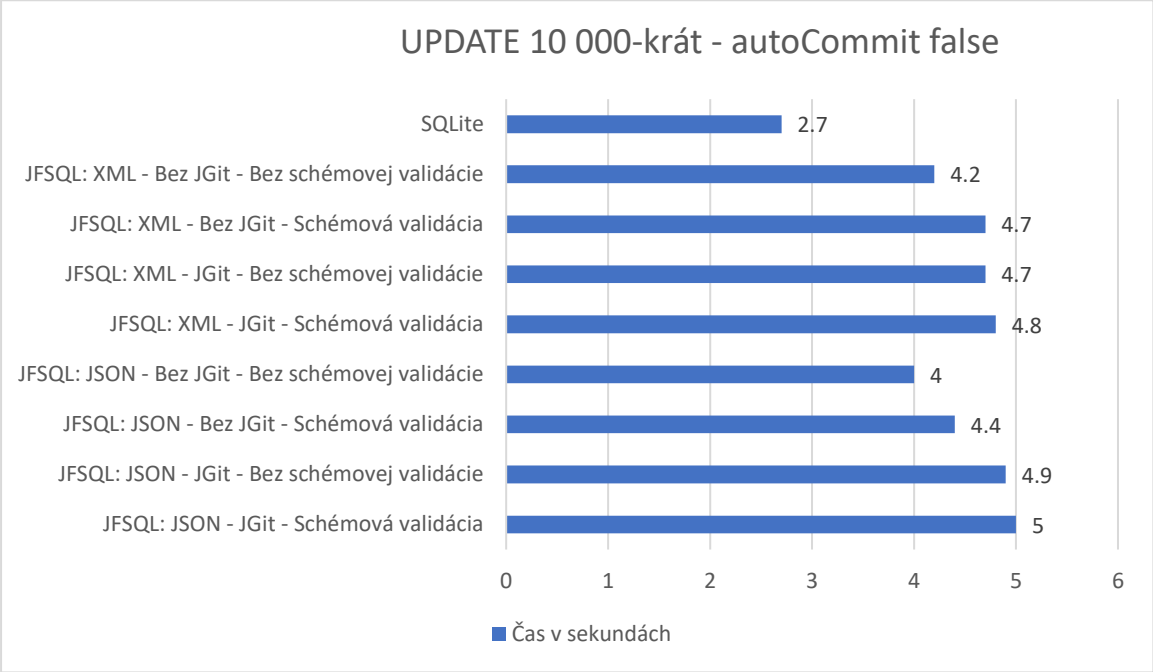


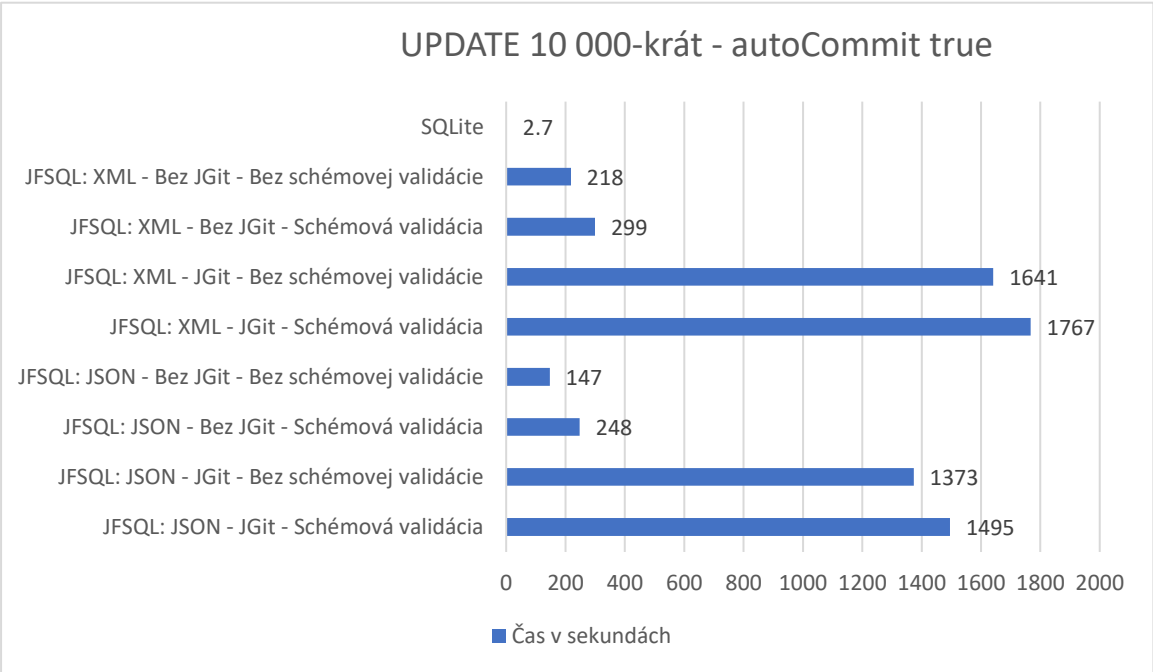
5.3.3 SELECT 10 000-krát



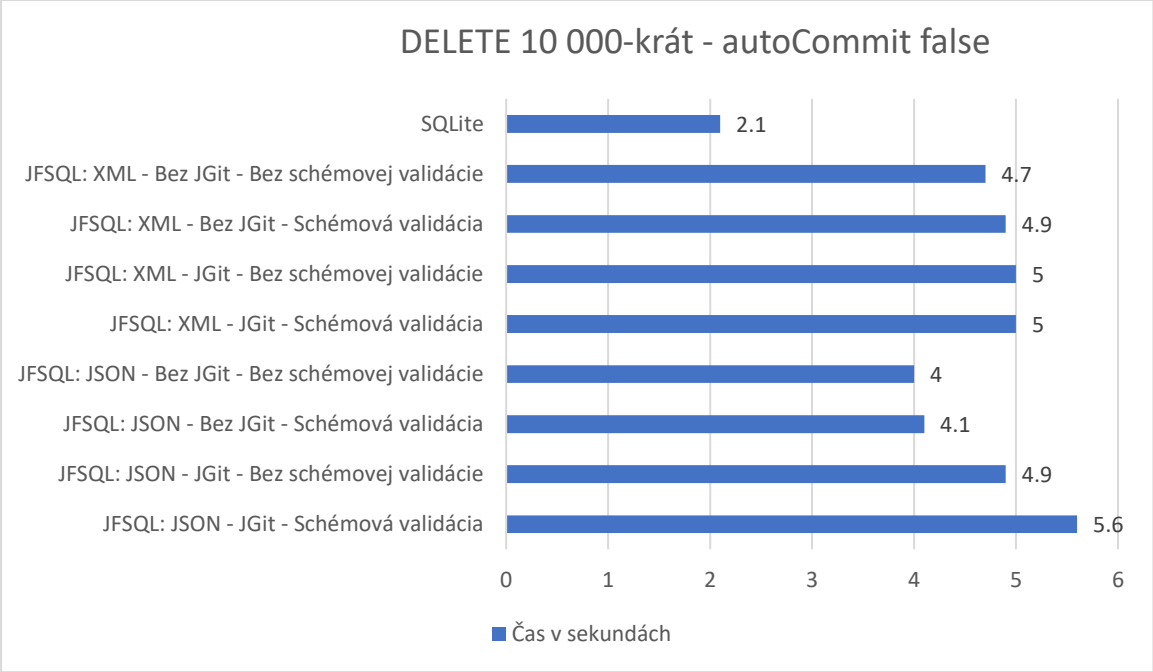


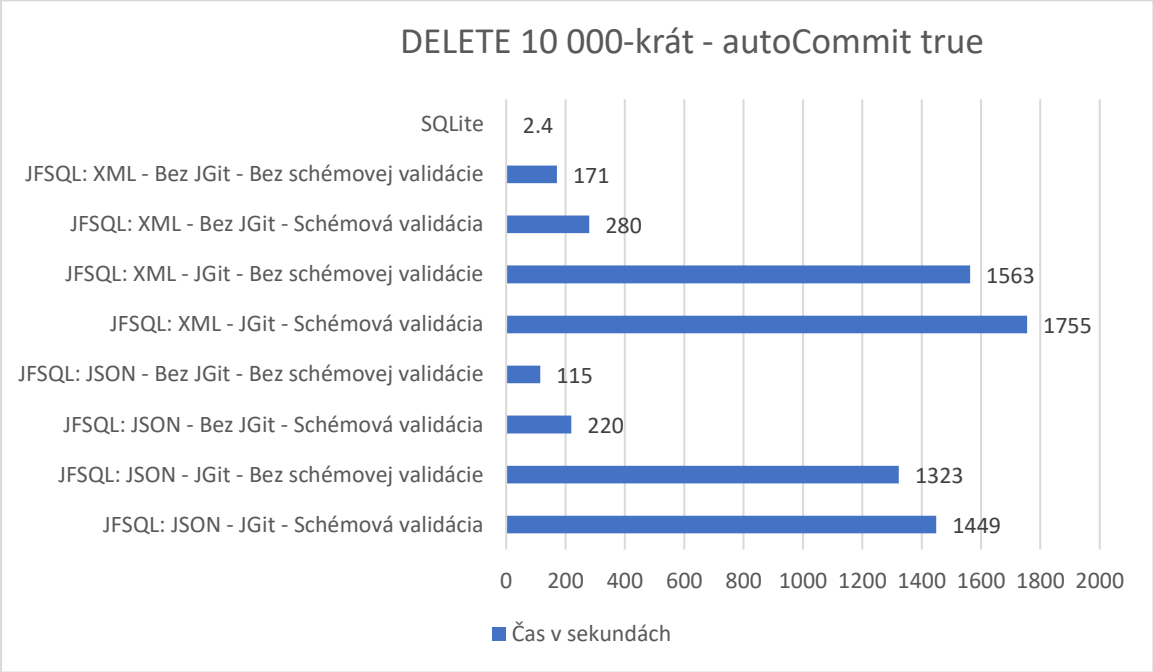
5.3.4 UPDATE 10 000-krát





5.3.5 DELETE 10 000-krát





6 Návrh na zlepšenie

6.1 Rozšírenie syntaxu

Syntax nášho ovládača bola vyvinutá prispôsobením gramatiky SQLite, v dôsledku čoho bola začlenená len podmnožina funkcií SQLite. Gramatiku ovládača však možno rozšíriť o ďalšie kľúčové slová, funkcie a iné funkcie. Napriek tomu je potrebné zodpovedajúcim spôsobom implementovať aj zodpovedajúce metódy na strane ovládača, aby ovládač mohol tieto novo pridané funkcie podporovať.

Je dôležité poznamenať, že takéto rozšírenia sa musia implementovať opatrne a dôkladne testovať, aby sa predišlo akýmkoľvek nepriaznivým účinkom na existujúce funkcie ovládača. Pri starostlivom plánovaní a realizácii môže rozšírenie gramatiky ovládača výrazne rozšíriť jeho možnosti a ponúknuť koncovým používateľom väčšiu flexibilitu.

6.2 Podpora iných ľudsky čitateľných formátov

Na základe analýzy sme sa rozhodli použiť JSON a XML kvôli výhodám vysvetleným v kapitole 3.1. Ovládač je však možné rozšíriť tak, aby podporoval aj iné formáty, napríklad CSV, YAML alebo TOML. Na to je potrebné vytvoriť nové triedy implementujúce rozhrania Reader a Writer a nakoniec upraviť metódy v triede FileNameCreator tak, aby podporovali nové formáty. Všetky ostatné triedy a komponenty sú nezávislé od serializačných formátov.

6.3 Implementácia ďalších metód z JDBC API

V našom ovládači boli implementované základné funkcie z tried, ktoré sú poskytované JDBC API. Každá metóda, ktorá nebola implementovaná do

nášho ovládača vyhadzuje *SQLFeatureNotSupportedException*. Tiež by bolo možné implementovať ďalšie triedy implementujúce rozhrania, aby sme rozšírili funkcie ovládača.

6.4 Alternatíva na verzovanie databázy

JGit má výrazný vplyv na výkon ovládača počas operácií zápisu, čo sa preukázalo v testoch. Na druhej strane je obrovskou výhodou možnosť vrátiť databázu do bezpečného bodu v prípade poškodenia údajov. Vytvorenie novej implementácie TransactionManager a DatabaseManager, ktorá serializuje java objekty do súborov, by bol skvelý nápad, ktorý by mohol slúžiť ako záloha v prípade poškodenia údajov.

7 Zhodnotenie

Cieľom tejto bakalárskej práce bolo vytvorenie ovládača, ktorý pracuje nad súborovým systémom, a údaje uloží vo formáte, ktoré sú čitateľné ľudom. Počas analýzy sme preskúmali rôzne serializačné formáty, možné spôsoby ukladania údajov. Preskúmali sme rôzne možnosti parsovania SQL príkazov a existujúce spôsoby riešenia transakcií v jednotlivých databázových systémoch.

V návrhu sme určili akým spôsobom budeme ukladať údaje na súborovom systéme, a aký format budeme používať. Určili sme, ako budeme parsovať SQL príkazy a záver ako budeme riešiť transakcie v našej implementácií.

V kapitole implementácia sme detailnejšie opísali najdôležitejšie komponenty nášho ovládača, ako jednotlivé komponenty a triedy interagujú a tiež sme si spomínali najväčšie výzvy pri implementácií.

V kapitole overenie riešenia sme detailne opísali ako sme testovali ovládač a parser, ako sme overili funkčnosť jednotlivých komponentov a záver sme porovnávali našu implementáciu s SQLiteom.

V poslednej kapitole návrh na zlepšenie sme uviedli niekoľko nápadov, čo by bolo ešte vylepšiť, aké funkcionality by bolo možné pridávať.

Použitá literatúra

[1] K. Sharan, "JDBC API", v *Java APIs, Extensions and Libraries*, Apress, 2018, s. 347–487. doi: 10.1007/978-1-4842-3546-1_5.

[2] R. M. Menon, *Expert Oracle JDBC Programming*. 2005. doi: 10.1007/978-1-4302-0029-1.

[3] TutorialsPoint, *JDBC Tutorial*. 2015. Cit: 01. september 2022. [Online]. Available at: https://www.tutorialspoint.com/ebook/jdbc_tutorial/index.asp

[4] Oracle Corporation, "Java Platform, Standard Edition 7 API Specification - Package java.sql". https://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html (cit 02. september 2022).

[5] Pratik. Patel a Karl. Moss, *Java database programming with JDBC*. Coriolis Group Books, 1996.

[6] Harvard EECS, "Introduction to File Systems". https://www.eecs.harvard.edu/~cs161/notes/intro-file-systems.pdf. (cit 08. máj 2023).

[7] C. Smith, "Re: What is the difference between a file system and a database?", *Quorra*. https://www.quora.com/What-is-the-difference-between-a-file-system-and-a-database/answer/Christian-Smith-2 (cit 08. máj 2023).

[8] A. Silberschatz, H. F. Korth, a S. Sudarshan, *Database system concepts*, Seventh edition.

[9] SQLite, "Architecture of SQLite". https://www.sqlite.org/arch.html (cit 08. máj 2023).

[10] F. Toussi, "'Chapter 12. Deployment Guide' - HSQLDB User Guide". https://hsqldb.org/doc/2.0/guide/deployment-chapt.html#dec_cache_mem_use (cit 08. máj 2023).

[11] N. Fortescue, "Re: Why Should I Use a Human Readable File Format?", *Stack Overflow*. https://stackoverflow.com/questions/568671/why-should-i-use-a-human-readable-file-format (cit 08. máj 2023).

[12] A. Bertram, "The state of config file formats: XML vs. YAML vs. JSON vs. HCL", *Octopus Deploy*. https://octopus.com/blog/state-of-config-file-formats (cit 08. máj 2023).

[13] J. Gray a A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st vyd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[14] P. A. Bernstein, V. Hadzilacos, a N. Goodman, *Concurrency Control and Recovery in Database Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[15] IBM, “Optimistic and pessimistic record locking - IBM Documentation”. https://www.ibm.com/docs/en/rational-clearquest/7.1.0?topic=clearquest-optimistic-pessimistic-record-locking (cit 27. marec 2023).
[16] H. Garcia-Molina, J. D. Ullman, a J. Widom, *Database systems : the complete book*. 2009.
[17] V. Mihalcea, “Optimistic vs. Pessimistic Locking”. https://vladmihalcea.com/optimistic-vs-pessimistic-locking/ (cit 08. máj 2023).
[18] H2 Database, “H2 Database Engine”. http://www.h2database.com/html/advanced.html#transactions (cit 27. marec 2023).
[19] F. Toussi, “‘Chapter 6. Sessions and Transactions’ - HSQLDB User Guide”. http://hsqldb.org/doc/2.0/guide/sessions-chapt.html#snc_tx_tx_cc (cit 27. marec 2023).
[20] SQLite, “Write-Ahead Logging”. https://www.sqlite.org/wal.html (cit 10. december 2022).
[21] baeldung, “Guide to WeakHashMap in Java | Baeldung”. https://www.baeldung.com/java-weakhashmap (cit 07. máj 2023).