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

Obsah

[1 Úvod 1](#_Toc129516505)

[2 Analýza 2](#_Toc129516506)

[2.1 Architektúra JDBC 2](#_Toc129516507)

[2.2 Súbor ako úložisko údajov 5](#_Toc129516508)

[2.3 SQL Príkazy 5](#_Toc129516509)

[2.3.1 Regulárne výrazy 6](#_Toc129516510)

[2.3.2 Parsovanie pomocou bezkontextovej gramatiky 7](#_Toc129516511)

[2.4 Ľudsky čitateľný formát 8](#_Toc129516512)

[2.4.1 Serializácia a deserializácia 11](#_Toc129516513)

[2.4.2 Možné spôsoby ukladania údajov 14](#_Toc129516514)

[2.4.3 Obmedzenia 16](#_Toc129516515)

[2.5 Transakcie 17](#_Toc129516516)

[2.5.1 Úrovne izolácie 17](#_Toc129516517)

[2.5.2 Transakcie pri databázach, ktorý používajú súborový systém 19](#_Toc129516518)

[3 Návrh 21](#_Toc129516519)

[3.1 Parsovanie SQL príkazov 21](#_Toc129516520)

[3.2 Ľudsky čitateľný formát 22](#_Toc129516521)

[3.2.1 Formát tabuľky 22](#_Toc129516522)

[3.2.2 Formát schémy 22](#_Toc129516523)

[3.2.3 Formát databázového súboru 23](#_Toc129516524)

[3.2.4 Formát uloženia LOB-ov 23](#_Toc129516525)

[3.3 Transakcie 24](#_Toc129516526)

[3.3.1 Commit 25](#_Toc129516527)

[3.3.2 Rollback 26](#_Toc129516528)

[4 Implementácia 27](#_Toc129516529)

[4.1 SQL Príkazy 28](#_Toc129516530)

[4.1.1 Hierarchia príkazov 29](#_Toc129516531)

[4.1.2 Spracovanie príkazov 30](#_Toc129516532)

[4.1.3 Soft parsing 31](#_Toc129516533)

[4.1.4 Podporované príkazy a ich formáty 32](#_Toc129516534)

[4.2 Ľudsky čitateľný formát 34](#_Toc129516535)

[4.3 Transakcie 34](#_Toc129516536)

[4.4 Obmedzenia 35](#_Toc129516537)

[4.4.1 Syntax SQL 35](#_Toc129516538)

[4.4.2 Nepodporované funkcie v ovládači 35](#_Toc129516539)

[5 Overenie riešenia 37](#_Toc129516540)

[6 Návrh na zlepšenie 37](#_Toc129516541)

[Použitá literatúra 38](#_Toc129516542)

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)

# Úvod

# Analýza

## 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 súbory. [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 applet 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 nasledujúcom podkapitole sa budeme venovať dôležitejším triedam a rozhraniam.

**DriverManager** - zabezpečuje, aby sa na prístup ku každému zdroju údajov použil správny ovládač. DriverManager je schopný podporovať viacero súbežných ovládačov spojených k viacerým heterogénnym databázam. [3]

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

**Dôležité triedy a rozhrania**

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

**DriverManager -** Zodpovedá požiadavky na spojenie z Java aplikácie so správnou databázou ovládačom pomocou komunikačného podprotokolu. Prvý ovládač, ktorý rozpozná určitý podprotokol v rámci JDBC, sa použije na vytvorenie databázy Connection. [3]

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

**Statement –** Slúži na spúšťanie príkazov SQL 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 -** Predkompilovaný príkaz SQL je známy ako PreparedStatement. Keďže je predkompilovaný, 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.

## Súbor ako úložisko 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.

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

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 súborový systém je jednoduchší systém, ktorý je vhodnejší na organizovanie a ukladanie menšieho množstva údajov.

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

Okrem toho niektoré DBMS poskytujú nástroje na export údajov z databázy do HRF, napríklad CSV alebo JSON. To umožňuje používateľom pristupovať k údajom spôsobom, ktorý sa dá ľahko čítať a interpretovať ľuďmi, a umožňuje to používať údaje v iných aplikáciách alebo systémoch, ktoré nepodporujú binárny formát údajov používaný databázou.

## SQL Príkazy

SQL príkazy pozostávajú z viacerých vstupov, ako sú názvy tabuliek, názvy stĺpcov, rôzne funkcie a výrazy. Existujú dva možné spôsoby spracovania príkazov. Prvý, že sa nachádzajú vo vyrovnávacej pamäti (cache), a druhý, že sa nemusia nachádzať.

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.

### 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.

### 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. [6]

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.

## Ľudsky čitateľný formát

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:

|  |  |
| --- | --- |
| **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 1: Výhody a nevýhody HRF

Medzi najpopulárnejšie HRF patria CSV, JSON, XML a YAML. V tejto kapitole budeme analyzovať jednotlivé serializačné 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, boolean, 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.

**XML -** XML je serializačný protokol, ktorý je čitateľný pre človeka, hoci sa ľahko stratí medzi všetkými značkami, ktoré máme pred očami. 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.

**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.

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 2: Ukážka jedného záznamu

### 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 3: Veľkosti jednotlivých súborov

Tabuľka 4: Počet riadkov

Tabuľka 5: 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 najdlhší.

Tabuľka 6: 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.

### 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.

#### 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>

#### 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>

#### 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>

#### Uloženie LOB-ov

Ak tabuľka obsahuje stĺpec typu BLOB alebo CLOB, 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>

### Obmedzenia

#### 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 7: 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 viac ako dosť. 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.

#### Maximálny počet súčasne otvorených súborov

Vo všeobecnosti neexistujú žiadne prirodzené 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.

## 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]

Keď je transakcia commitovaná, účinky všetkých príkazov vykonaných od posledného commitu alebo rollbacku transakcie sa natrvalo zaznamenávajú v databáze.

Vykonaním rolbacku sa uvoľnia všetky zámky, ktoré databáza získala počas aktuálnej transakcie, ako aj všetky modifikácie údajov databázy vykonané počas aktuálnej transakcie. Pri používaní JDBC môžeme transakciu zrušiť zavolaním funkcie *rollback()* na objekte Connection. [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:

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

*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.

*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.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 8: Izolačné úrovne

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

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

*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.

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

### Transakcie pri databázach, ktorý používajú súborový systém

Ak databáza používa na ukladanie údajov súborový systém, transakcie sa zvyčajne implementujú pomocou techniky známej ako "zapisovanie pred zápisom". Ide o zápis všetkých zmien, ktoré sú súčasťou transakcie, do logovacieho súboru pred ich použitím v skutočných dátových súboroch. Tým sa zabezpečí, že zmeny sa dajú vrátiť späť, ak sa počas transakcie niečo pokazí, a tiež to umožňuje opätovné použitie zmien, ak sa databáza zrúti alebo sa inak stane nedostupnou. [7]

Uvažujme napríklad databázu SQLite, ktorá na implementáciu transakcií používa protokolovanie s predstihom zápisu. Keď sa spustí transakcia, DBMS zapíše do súboru protokolu záznam "begin transaction" a potom zapíše záznamy pre každú operáciu, ktorá je súčasťou transakcie. Tieto záznamy obsahujú podrobnosti o vykonávaných zmenách, napríklad o vkladaných alebo aktualizovaných hodnotách. [7]

Po zapísaní všetkých zmien do súboru protokolu ich databázový stroj aplikuje na skutočné dátové súbory. To sa uskutočňuje tak, že sa najprv vytvorí kópia pôvodných dátových súborov pre prípad, že by bolo potrebné transakciu neskôr vrátiť späť. Potom sa zmeny aplikujú na dátové súbory a databázový stroj zapíše do logovacieho súboru záznam "commit transaction", ktorý označuje, že transakcia bola úspešne dokončená. [7]

Ak sa počas transakcie niečo pokazí, napríklad chyba databázy alebo zlyhanie systému, databázový stroj môže použiť súbor protokolu na zrušenie vykonaných zmien. To sa vykoná tak, že sa zo súboru denníka prečítajú záznamy "begin transaction" a "commit transaction" a potom sa na dátové súbory použijú opačné operácie. Ak napríklad transakcia obsahovala operáciu vloženia, operácia zrušenia by bola operácia vymazania. [7]

# Návrh

## Parsovanie SQL príkazov

ANTLR (ANother Tool for Language Recognition) 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 všetko 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.

## Ľudsky čitateľný formát

### Formát tabuľky

Podľa analýzy sme sa rozhodli uložiť každú tabuľku ako jeden súbor, pretože nechceme byť obmedzení maximálnym počtom 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>37</age>  
 </Entry>  
</myTable>

### Formát schémy

Informácie o tom, akého dátového typu je hodnota, budeme ukladať do schém. XML a JSON poskytujú schémy na overenie štruktúry súboru, čím sa zabezpečí jeho súlad s určitým súborom pravidiel alebo usmernení.

*<?*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>

### 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>

### Formát uloženia LOB-ov

LOB-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>

## Transakcie

Transakcie budú riešené prostredníctvom kontroly verzií systému Git, ktorý poskytuje robustný systém verzií pre zmeny vykonané v databáze. Na realizáciu tohto procesu sa po vytvorení databázy inicializuje nový repozitár Git a každá operácia zápisu bude následne commitovaný .

Je však nevyhnutné poznamenať, že proces transakcií nebude úplne atomický, pretože commity sa vykonajú až po tom, ako sú zmeny k dispozícii v databáze a boli zapísané do súborov.

V prípade zlyhania operácie commit sa použije tvrdý reset (hard reset) na návrat do predchádzajúceho stavu. Na optimalizáciu výkonu bude program ukladať necommitované zmeny v dávkach a zapisovať ich do súborov len pri volaní metódy commit(), čím sa zníži počet potrebných diskových operácií. Tento prístup poskytuje efektívnu metódu na sledovanie a riešenie transakcií a zároveň minimalizuje vplyv na výkon systému.

Je však veľmi dôležité zabezpečiť, aby bol repozitár Git vhodne udržiavaný a aby sa operácie odovzdávania pozorne sledovali, aby sa predišlo strate údajov alebo nekonzistentnosti.

### Obrázok, na ktorom je diagram Automaticky generovaný popisCommit

### Obrázok, na ktorom je diagram Automaticky generovaný popisRollback

# Implementácia

## 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.

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 3: Príklad parsovacieho stromu

Z tohto sql reťazca sa vytvoril objekt typu *SelectStatement*, ktoré má nasledujúce atribúty:

tableName=myTable,

joinTableNames=[myTable2],

joinTypes=[INNER\_JOIN],

columns=[\*],

listOfJoinColumns=[[myTable.id, myTable2.id]],

whereColumns=[name],

whereValues=[Zsolti],

symbols=[LIKE],

binaryOperators=[]

### Obrázok, na ktorom je diagram Automaticky generovaný popisHierarchia príkazov

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.

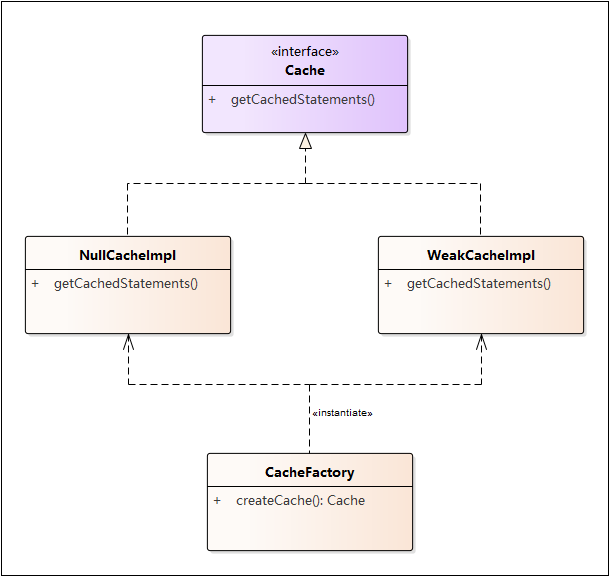
### Spracovanie príkazov

Obrázok, na ktorom je diagram

Automaticky generovaný popisANTLR4 zabezpečuje, aby sa príkazy spracovali len vtedy, ak neobsahujú gramatické chyby. Správnosť údajov sa však na úrovni parsera nekontroluje. Napríklad, či daná tabuľka existuje v databáze alebo či má daný stĺpec zodpovedajúci dátový typ. Ak má príkaz správnu syntax, odošle sa ovládaču na spracovanie.

Každý typ príkazu je spojený so servisom, ktorý ho spracúva. V servise sa kontroluje každý atribút na sémantickú validáciu, ktorú vykonáva komponent Semantic validator. Ak príkaz nie je platný, vyskočí výnimka SQLException a príkaz sa ďalej nespracúva. Pri operáciách, pri ktorých sa vykonáva zápis údajov, sa používa aj validácia podľa schémy. Ak je validácia neúspešná, automaticky sa zavolá metóda *rollback(),* ktorá vráti databázu do bodu, v ktorom boli údaje ešte platné. Tento prístup nám zabezpečí úplnú integritu údajov v databáze.

### Soft parsing

Na tento účel sme vytvorili rozhranie Cache, ktoré je implementované dvoma triedami, NullCacheImpl a WeakCacheImpl. NullCacheImpl nebude ukladať príkazy do vyrovnávacej pamäte, takže príkazy sa budú musieť vždy analyzovať. WeakCacheImpl používa na ukladanie príkazov do vyrovnávacej pamäte WeakHashMap, kde kľúče sú reťazce sql a hodnoty sú vytvorené objekty prenosu dát.

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.

### Podporované príkazy a ich formáty

* ALTER TABLE podporuje premenovanie tabuľky a stĺpca, pridanie nového stĺpca do tabuľky a odstránenie existujúceho stĺpca. Podporované formáty sú:

ALTER TABLE myTable RENAME TO myTableEdited;

ALTER TABLE myTable RENAME COLUMN id TO id\_edited;

ALTER TABLE myTable ADD COLUMN id INTEGER;

ALTER TABLE myTable DROP COLUMN id;

* CREATE TABLE podporuje dátové typy INTEGER, REAL, TEXT a BLOB. Podporovaný formáty je:

CREATE TABLE myTable (id INTEGER, name TEXT, age INTEGER);

* CREATE DATABASE sa používa na vytvorenie databázy. Tento príkaz je nepovinný a nie je podmienkou na pripojenie adresára. Podporovaný formát je:

CREATE DATABASE [C:\path\to\database];

* DELETE podporuje štandardný formát, v kombinácií s binárnymi operátormi.

DELETE FROM myTable;

DELETE FROM myTable WHERE id > 1;

* DROP TABLE podporuje štandardný formát.

DROP TABLE myTable;

* DROP DATABASE slúži na vymazanie databázového adresára. Táto operácia je ireverzibilná. Podporovaný formát je:

DROP DATABASE [C:\path\to\database];

* INSERT podporuje nasledujúce formáty:

INSERT INTO myTable (c1, c2, c3) VALUES (1, 2.5, 'abc');  
INSERT INTO myTable (c1, c2, c3) VALUES (1, 2.5, 'abc'), (2, 2.3, 'def');  
INSERT INTO myTable VALUES (1, 2.5, 'def');  
INSERT INTO myTable VALUES (1, 2.5, 'abc'), (2, 2.3, 'def');

* SELECT podporuje LEFT OUTER JOIN a INNER JOIN, v kombinácií s binárnymi operátormi. V príkazoch obsahujúci JOIN, prefixy pre stĺpce, ktoré sa vyskytnú vo viacerých tabuľkách je povinné.

SELECT *\** FROM myTable;

SELECT id FROM myTable;

SELECT c1, c2, c3 FROM myTable;

SELECT c1, c2, c3 FROM myTable WHERE value1 > 1;

SELECT *\** FROM myTable JOIN myTable2 ON myTable.id = myTable2.id;

SELECT *\** FROM myTable INNER JOIN myTable2 ON myTable.id = myTable2.id;

SELECT *\** FROM myTable LEFT JOIN myTable2 ON myTable.id = myTable2.id;

SELECT *\** FROM myTable LEFT OUTER JOIN myTable2 ON myTable.id = myTable2.id;

* UPDATE podporuje nasledujúce formáty, v kombinácií s binárnymi operátormi.

UPDATE myTable SET value1 = 26;

UPDATE myTable SET value1 = 26 WHERE value1 = 1;

UPDATE myTable SET value1 = 26, value2 = 'abc' WHERE value2 = 'def';

* Podporované binárne operátory sú AND a OR, ktoré možno reťaziť, podporované symboly porovnania sú LIKE, <, <=,=,>=,>.

## Ľudsky čitateľný formát

Obrázok, na ktorom je diagram

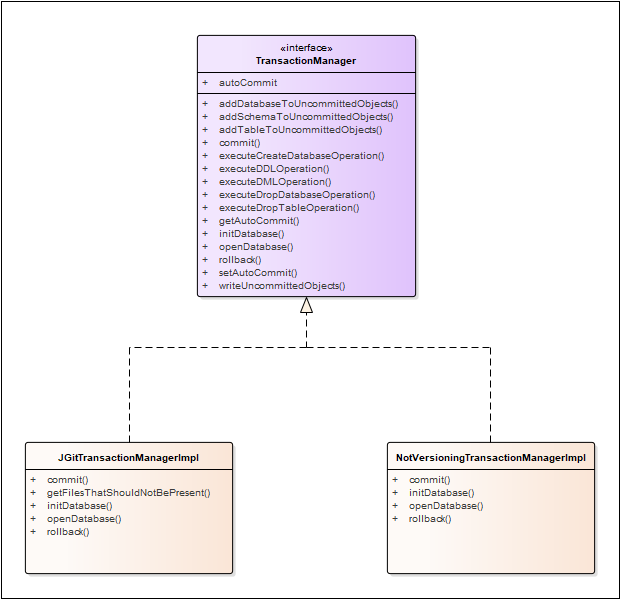
Automaticky generovaný popisNa základe analýzy sme sa rozhodli použiť formáty JSON a XML z dôvodu rýchlej serializácie, deserializácie a možnosti overiť integritu spracovávaných údajov prostredníctvom schém. Na čítanie a zápis zo súborov používame samotné triedy, ktoré implementujú rozhrania Reader a Writer. Tieto rozhrania poskytujú úplnú nezávislosť od serializačných formátov.

TO-DO: Write some BS here about schema validation

## Transakcie

Na transakcie sa budeme používať JGit, čo je open-source implementácia systému Git napísaná v jazyku Java. Poskytuje množinu rozhraní API na prístup k úložiskám Git a manipuláciu s nimi. JGit nevyžaduje, aby bolo Git nainštalované lokálne na zariadení, kde sa bude používať náš ovládač.

Jednou z kľúčových funkcií systému Git je podpora atomických commitov, čo znamená, že zmeny v úložisku sa stávajú trvalými až po ich commitovaní.

JGit umožňuje vytvárať a odovzdávať zmeny do úložiska, ale namiesto natívneho súborového systému ukladá údaje do pamäte. Zmeny vykonané pomocou JGit sa nezapisujú na disk okamžite, ale ukladajú sa do súborového úložiska objektov a odovzdávajú sa na disk ako súčasť transakcie.

Uvedený spôsob riešenia transakcií je implementovaný triedou JGitTransactionManagerImpl. Druhá trieda implementujúca rozhranie TransactionManager nepoužíva JGit pre transakcie, ale keď je táto trieda vybraná v ovládači, metóda *rollback()* nebude použiteľná. Hoci neexistuje mechanizmus na návrat databázy do predchádzajúceho (bezpečného) stavu, ak je operácia zápisu neúspešná a dôjde napríklad k poškodeniu údajov, čas potrebný na dokončenie každej operácie zápisu sa pri použití tejto triedy výrazne skráti.

Použitá literatúra

[1] K. Sharan, *Java APIs, Extensions and Libraries*. 2018. doi: 10.1007/978-1-4842-3546-1.

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

[3] TutorialsPoint, *JDBC Tutorial*. 2015. Accessed: Sep. 01, 2022. [Online]. Available: https://www.tutorialspoint.com/ebook/jdbc\_tutorial/index.asp

[4] “java.sql (Java Platform SE 7 ).” https://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html (accessed Sep. 02, 2022).

[5] Pratik. Patel and Karl. Moss, “Java database programming with JDBC,” p. 480, 1996.

[6] James H. Martin and Dan Jurafsky, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st edition., no. 4. Prentice Hall, 2000. Accessed: Feb. 26, 2023. [Online]. Available: http://www.mitpressjournals.org/doi/pdf/10.1162/089120100750105975

[7] “Write-Ahead Logging.” https://www.sqlite.org/wal.html (accessed Dec. 10, 2022).