VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

OVĚŘENÍ VLASTNOSTÍ SQL KÓDU

CHECKING SQL CODE PROPERTIES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. FILIP BALI**
AUTHOR

VEDOUCÍ PRÁCE **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

143979

Ústav: Ústav informačních systémů (UIFS)
Student: **Bali Filip, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Informační systémy a databáze
Název: **Ověření vlastností SQL kódu**
Kategorie: Databáze
Akademický rok: 2022/23

Zadání:

1. Seznamte se se standardem jazyka SQL a jeho nejběžněji používanými verzemi. Prozkoumejte případná rozšíření jazyka v různých databázových systémech (např. Oracle, PostgreSQL, MySQL) a možnosti parsování výrazů jazyka SQL v jeho standardní i rozšířené podobě (např. v rámci parserů nástroje Tree-sitter).
2. Prozkoumejte možnosti a existující řešení pro statickou analýzu zdrojového kódu za účelem automatického ověření jeho kvality (tzv. "linting"). Zaměřte se zejména na typy, sílu a zápis pravidel pro ověření kódu. Prozkoumejte také požadované vlastnosti kvalitního SQL kódu.
3. Navrhněte způsob, jak uživatelsky snadno zapsat a později ověřit pravidla popisující vlastnosti SQL skriptu, jako je výskyt určitých konstrukcí (např. agregační funkce a seskupování, spojení přes více tabulek, použití určitých predikátů, propojení poddotazů s hlavním dotazem, atp.) či splnění výkonnostních a bezpečnostních požadavků (např. oprávnění pouze na některé sloupce, existence indexu nad cizím klíčem, atp.).
4. Implementujte nový, či rozšiřte existující, nástroj pro statickou analýzu SQL skriptů a ověření jejich vlastností dle pravidel zadaných v době spuštění. Vytvořte také vhodné testovací příklady s SQL pro databázový server Oracle.
5. Řešení otestujte, vyhodnoťte a diskutujte výsledky. Výsledný software publikujte jako open-source.

Literatura:

- MUSE, Biruk Asmare, et al. On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020. p. 327-338. ISBN 978-1-4503-7517-7
- LINK, Sebastian; MEMARI, Mozhgan. Static analysis of partial referential integrity for better quality SQL data. In Proceedings of the Nineteenth Americas Conference on Information Systems, Chicago, Illinois, 2013. ISBN 978-0-615-55907-0
- *Sqlfluff/sqlfluff: A SQL linter and auto-formatter for Humans* [online]. GitHub, 2022 [cit. 2022-05-02]. Dostupné z: https://github.com/sqlfluff/sqlfluff

Při obhajobě semestrální části projektu je požadováno:
Body 1, 2 a 3.

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

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 17.5.2023
Datum schválení: 18.10.2022

Abstrakt

Táto diplomová práca sa zaoberá kontrolou vlastností SQL kódu na základe statickej analýzy s využitím preddefinovaných pravidiel. Pravidlo predstavuje základný element kontroly. Užívateľ si môže definovať vlastné pravidlo a začleniť ho do kontroly. Pravidlo typicky obsahuje súbor podmienok obsiahnutých v algoritme, ktorý kontroluje uzly v abstraktnom syntaktickom strome. Ten je vytvorený zo vstupného SQL príkazu a prispôsobený tak, aby bolo možné nad jeho uzlami aplikovať pravidlá. Ak pravidlo zistí nezrovnalosť, potom môže vytvoriť hlásenie. Potom sú tieto hlásenia zobrazené na zvolený výstup. Na základe práce bol implementovaný program s otvoreným kódom v programovacom jazyku Python3. Tento program je verejne dostupný.

Abstract

This thesis focuses on checking the properties of SQL code based on static analysis using predefined rules. The rule represents a basic element of the check. The user can define their own rule and include it in the check. A rule usually contains a set of conditions that are contained in an algorithm that checks nodes in an abstract syntactic tree. Abstract syntactic tree is created from an input SQL statement and customized so that rules can be applied over its nodes. If the rule detects an error, then it can generate a report. These reports are then displayed on the selected output. Based on the thesis, an open source program in the Python3 programming language was implemented. This program is publicly available.

Kľúčové slová

Statická analýza kódu, SQL kód, Kontextová analýza, Pravidlá, Program, Otvorený kód

Keywords

Static code analysis, SQL code, Context analysis, Rules, Program, Open source

Citácia

BALI, Filip. *Ověření vlastností SQL kódu.* Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Ověření vlastností SQL kódu

Prehlásenie

Prehlasujem, že túto diplomovú prácu som vypracoval samostatne pod vedením pána RNDr. Mareka Rychlého Ph.D. a uviedol som všetky literárne pramene z ktorých som čerpal.

.....
Filip Bali
16. mája 2023

Poďakovanie

Chcel by som sa poďakovať svojmu vedúcemu práce, ktorý bol vždy k dispozícii, za jeho milý prístup a poskytnutie odbornej pomoci počas riešenia diplomovej práce. Rovnako by som sa chcel poďakovať svojej rodine, ktorá mi bola vždy oporou a stála pri mne počas celého štúdia. Poďakovanie patrí aj mojím kamarátom, za ich podporu a spoločne strávený čas.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

1.1 Motivácia

Vývoj softvéru obsahuje niekoľko etáp. Niektoré sa môžu pravidelne opakovať, pokým nie je implementovaný súbor funkcionalít podľa dohodnutých kritérií. Súčasťou vývoja je taktiež ladenie softvéru a hľadanie programových chýb [19]. K tomu, aby bolo v softvéri menší počet chýb môže pomôcť statický analyzátor kódu [2].

Statický analyzátor kódu je program, ktorého cieľom je kontrola kódu bez jeho prekladu alebo interpretácie. Princíp jeho fungovania môže byť založený na vopred definovaných pravidlách. Tieto pravidlá majú za cieľ kontrolovať vopred definované časti kódu a popísať jeho očakávanú formu. Pravidlami tak vzniká súbor podmienok, ktoré musí kód spĺňať na to, aby vyhovel kontrole. V prípade, že kód kontrole pravidla nevyhovie, musí o tom pravidlo informovať. Pravidlo sa tak dostáva do neočakávaného stavu, kde ma definovaný spôsob akým bude o nezrovnalosti informovať. Typicky sa vytvorí hlásenie, ktoré definoval autor pravidla a ktoré prináleží danej nezrovnalosti. O týchto hláseniach bude následne program informovať užívateľa.

V súčastnosti pre mnoho programovacích jazyk existuje aj niekoľko riešení statického analyzátora, ktoré je možné integrovať aj do vývojového prostredia [23][29]. Výnimkou je jazyk SQL, kde daná ponuka mierne absentuje, špeciálne tých s otvoreným kódom. Nájdené existujúce riešenia častokrát ponúkali len obmedzený rozsah kontroly, ktorý sa špecializoval na určitý typ kontroly, napríklad na štýl zápisu SQL príkazov.

1.2 Cieľ

Cieľom diplomovej práce je vytvoriť statický analyzátor kódu pre jazyk SQL, ktorý overí daný kód na syntaktickej a aj kontextovej úrovni. Vytvorený program bude dostupný s otvoreným kódom voľne k použitiu, upraveniu alebo k inšpirácií. Užívateľ bude môcť definovať vlastné pravidlá, ktoré budú môcť byť jednoduchým spôsobom začlenené do programu a tým aj do kontroly kódu. Program poskytne užívateľovi vo formátovanej podobe výstup obsahujúci získane hlásenia z jednotlivých pravidiel o stave kódu.

Kapitola 2

Jazyk SQL a jeho analýza

Jazyk SQL je štandardizovaný jazyk pre manipuláciu s relačným databázovým systémom. Počas svojho historického vývoja bol adaptovaný viacerými poskytovateľmi databázových systémov. Tie môžu obsahovať nadstavby pre jazyk SQL, ktoré nemusia byť s inými databázovými systémami kompatibilné [28].

Je štandardizovaný organizáciami ANSI/ISO, ktoré priebežne vydávajú novšie verzie štandardu jazyka SQL. V dobe písania textu je najnovší štandard ISO/IEC 9075-1:2016 s technickými zmenami uvedenými prostredníctvom ISO/IEC 9075-1:2016/Cor 1:2022[1]. Tento dokument opisuje koncepčný rámec používaný v iných častiach normy ISO/IEC 9075 na špecifikáciu gramatiky jazyka SQL a výsledok spracovania príkazov v tomto jazyku pri implementácií jazyka SQL [7].

Nasledujú kapitoly, ktoré sa venujú rôznym oblastiam a špecifikám jazyka SQL. Cieľom je priblížiť čitateľovi oblasti jazyka SQL, ktoré súvisia s jeho kontrolou a majú tak priamy dopad na návrh programu.

2.1 Dialekty jazyka SQL

Ako už bolo spomenuté, poskytovateľov databázových systémom existuje viacero. Tí zvyčajne rozširujú štandard jazyka SQL o ich špecifické vlastnosti a funkcionality. Tým vznikajú SQL dialekty, ktoré nemusia byť vzájomne plne kompatibilné. Z hľadiska diplomovej práce je dôležitý poznatok, že vstupný jazyk SQL môže byť špecifický v závislosti na databázovom systéme, ktorý užívateľ programu využíva. Táto podkapitola približuje túto problematiku na porovnaní dvoch existujúcich relačných databázových systémoch: *MySQL* a *Oracle Database*.

Významnou odlišnosťou sú rôzne definované dátové typy. Používa sa rozličné názvoslovie a taktiež iná implementácia, ktorá spočíva v rôznej veľkosti, ktorú môže dátový typ nadobudnúť. Databázové systémy zároveň rozširujú názvoslovie aj o nové typy, ktoré väčšinou rozširujú iný (existujúci) dátový typ napríklad väčším rozsahom hodnôt, ktoré možno nadobúdať. U niektorých implementácií nie je naopak možné pridať znak kladnej hodnoty(+) pred číslicu. Preto ak chceme tvoriť kompatibilný SQL kód, potom je vhodné sa vyhnúť podobným zápisom. V nasledujúcej tabuľke 2.1 je možné vidieť porovnanie dátových typov jazyka SQL, pričom sú vybrané práve typy, ktoré sa líšia v tom ako ich definuje štandard a ako boli prijaté autormi databázových systémov [30].

[1] https://www.iso.org/standard/84485.html

ANSI/ISO	MySQL	Oracle Database
SMALLINT	SMALLINT (n)	NUMBER (5)
INT	INT (n)	NUMBER (10)
-	BIGINT (n)	NUMBER (38)
NUMERIC(p, s) p = presnosť s = rozsah	DECIMAL (n, d) n = maximum číslic d = desatinne čísla	NUMBER (p, s) p ∈ <1, 38> s ∈ <84,127>
-	CHAR(n) n ∈ <0, 1>	BOOLEAN <0, 1>
CHAR(n) n je nešpecifikované	CHAR(n) n ∈ <0, 255>	CHAR(n) n ∈ <0, 2000>
-	NCHAR(n) n ∈ <0, 65,535>	NCHAR(n) n ∈ <0, 2000>
VARCHAR(n) n je nešpecifikované	VARCHAR(n) n ∈ <0, 255>	VARCHAR2(n) n ∈ <0, 4,000>
-	NVARCHAR(n) n ∈ <0, 65,535>	NVARCHAR2(n) n ∈ <0, 4,000>

Tabuľka 2.1: Porovnanie dátových typov jazyka SQL [30, Kapitola 2].

Ďalšou ukážkou je zápis príkazu *SELECT* s maximálnym počtom záznamov (počet vyjadrený s *number_of_rows*), ktoré majú byť na výstupe. V tomto prípade je možné vidieť rozličný názov kľúčového slova. V *Oracle Database* je táto funkcionalita vyjadrená pomocou kľúčového slova *ROWNUM*, ktorý predstavuje virtuálny stĺpec a je ho možné zobraziť na výstupe príkazu. V prípade *MySQL*, je pre rovnakú funkcionalitu možné využiť kľúčového slova *LIMIT*, ktoré však nie je prístupné rovnakým spôsobom ako u *Oracle Database*. Ak chceme vytvoriť u *MySQL* virtuálny stĺpec s poradím, musíme využiť databázovú funkciu *ROW_NUMBER()* [30]. Nasleduje znázornenie príkladu pomocou SQL kódu:

```
1  /* MySQL Database */                    /* --Oracle Database */
2  SELECT column_name(s)                   SELECT column_name(s)
3  FROM table_name                         FROM table_name
4  LIMIT number_of_rows;                   WHERE ROWNUM <= number_of_rows;
```

2.2 Vlastnosti dobrého SQL kódu

Táto podkapitola popisuje niektoré vlastnosti, ktoré rozumieme pod dobrým SQL kódom. Prístupy k vytváraniu SQL kódu môžu byť odlišné podla preferencií a cieľov, preto sú tieto vlastnosti popísané vo forme odporúčaní s vysvetlením výhod, ktoré môžu vzniknúť ich použitím.

2.2.1 Štruktúra a funkcionalita

Tieto vlastnosti SQL kódu popisujú aký spôsob zápisu zvoliť pre lepšiu prehľadnosť a zároveň ako vhodne využívať poskytnutú funkcionalitu jazyka SQL.

Štýl zápisu príkazu

Pri písaní kódu je vhodné dodržiavať dohodnutý štýl zápisu, aby bol tak ľahšie čitateľnejší. Štýl zápisu môže byť zvolený podľa preferencií, avšak odporúča sa správne odsadzovanie logických celkov daného príkazu. Je bežné, že príkaz jazyka SQL pozostáva z niekoľkých klauzúl, ktoré tvoria samostatné logické celky a preto je ich vhodné štylisticky oddeliť na zvlášť riadky zarovnané vľavo. Tým získame, že jednotlivé logické celky príkazu začínajú na zvlášť riadku s vlastným kľúčovým slovom. V prípade, že je príkaz logického celku príliš dlhý na jeden riadok, potom ho je vhodne rozdeliť do viac riadkov, avšak tak, aby bol odsadený od začiatku riadku aspoň o dĺžku kľúčového slova, ktoré daný riadok popisuje. Ak využívame vnorené príkazy, potom je ich vhodné odsadiť podobným spôsobom, aby vyniklo, že sa jedná o samostatný príkaz [6]. Pre zvýraznenie kľúčových slov v texte je zároveň vhodné ich písať veľkými písmenami. Pri voľbe nového názvu prvku SQL databázy je vhodné sa vyhnúť názvom, ktoré by obsahovali kľúčové slová SQL jazyka. Obzvlášť vhodné je názvy nepísať do úvodzoviek(napríklad názov tabuľky). Ide o netypickú variantu zápisu, kde daný názov potom už musí stále obsahovať úvodzovky [6].

Alias tabuľky

Veľmi častým javom v SQL príkazoch je prítomnosť viacerých tabuliek. Jazyk SQL preto dovoľuje, aby mala tabuľka v rámci príkazu vlastný alias. Ten je vhodný využívať v prípade, keď je výhodné skrátiť pomenovanie tabuľky. Výhodou môže byť ľahšia čitateľnosť a orientácia v príkaze alebo zmena na názov, ktorý lepšie vystihuje kontext v danom príkaze. V prípade, že pracujeme so stĺpcom, ktorý je unikátny(nachádza sa iba v jednej tabuľke), potom jazyk SQL nevyžaduje definovania tabuľky, z ktorej pochádza. Avšak, pre lepšiu orientáciu iného čitateľa kódu je vhodné definovanie tabuľky vždy uviesť. Získame tak istotu, že vieme z akej tabuľky dáta pochádzajú [22]. Aliasy tabuliek by mali byť vždy unikátne v rámci celého príkazu a nemali by byť rovnaké ani ako názvy schém, ktoré sa môžu legálne objaviť v dotaze [14].

Preferovanie štandardných funkcií jazyka SQL

Vývojári databázových systémov nadväzujú na štandard jazyka SQL. Zároveň v závislosti na databázovom systéme býva jazyk rozšírený o nové kľúčové slova, mimo štandard. Tým vzniká špecifický dialekt jazyka SQL. Z toho následne môžu vyplývať problémy s kompatibilitou naprieč databázovými systémami. Ak chce tvorca SQL kódu dosiahnuť čo najväčšiu kompatibilitu, tak je vhodné preferovať prvky zo štandardizovaného jazyka SQL. Tým je možné získať opätovnú použiteľnosť kódu na inom databázovom systéme bez väčších zmien [6].

Formát dátumu

Dátum môže mať rôzne formy zápisu. Spôsob zápisu sa môže odlišovať podľa geografického pôvodu ale zároveň existujú aj rôzne dátové typy, ktoré s dátumom pracujú odlišne. Aby sme túto informáciu uložili v kompatibilnej a jasne čitateľnej podobe pre všetkých, tak je vhodné využiť medzinárodného štandardu ISO 8601 [6].

Spájanie tabuliek

V jazyku SQL je možné v niektorých prípadoch spojenie tabuliek dvoma rôznymi spôsobmi:

- S pomocou klauzule *FROM* s kľúčovým slovom *ON*.
- S pomocou filtrovacej podmienky v klauzuly *WHERE*.

V rámci daného príkazu, obe spomenuté klauzule majú iný kontextový význam. Kvôli tomu je vhodné vykonávať operáciu spájania tabuliek výhradne v klauzuly *FROM* a klauzulu *WHERE* ponechať ako filtrovaciu.

Preferovanie kľúčového slova BETWEEN/IN oproti AND/OR

Jazyk SQL podporuje funkcionalitu testu na rozsah. Tú je vhodné realizovať pomocou kľúčového slova *BETWEEN*. Zároveň podporuje aj funkcionalitu na kontrolu prítomnosti hodnoty a to s kľúčovým slovom *IN*. Obe tieto funkcionality je možné dosiahnuť aj s pomocou viac násobného použitia kľúčových slov *AND* a *OR*. Tým však vzniká zbytočne komplikovanejší kód [6].

2.2.2 Výkon a optimálnosť

Popisuje vlastnosti a princípy pre optimálnejšie písanie SQL kódu.

Výstup príkazu SELECT

Pri príkaze *SELECT* musíme definovať stĺpce, ktoré budú na výstupe. Motiváciou pre použitie môže byť:

- Získanie určitých dát z tabuľky.
- Získanie informácie aký typ dát tabuľka obsahuje.
- Overenie prítomnosti dát v tabuľke.

Získanie určitých dát z tabuľky je štandardná funkcionalita, kedy vyberieme stĺpce, ktoré majú predstavovať výstup. K optimálnosti tohoto spôsobu použitia príkazu *SELECT* sa budeme venovať až v ďalších bodoch.

Občas sa môže stať, že nevieme akú štruktúru a aký typ dát tabuľka obsahuje a chceli by sme sa na to opýtať databázového systému. Pri tomto úkone je nutné definovať výstup pomocou hviezdičky, ktorá symbolizuje výstup pre všetky stĺpce tabuľky. Aby to však nebolo výpočtovo náročné v prípade, že tabuľka obsahuje veľa záznamov, odporúča sa využiť kľúčové slovo *LIMIT*. To zaručí obmedzenie počtu záznamov tabuľky na výstupe.

Ak chceme len overiť prítomnosť určitých dát v tabuľke, je vhodné využiť kľúčového slova *EXITS* namiesto *IN*. Dôvodom je, že *EXITS* považuje príkaz za splnený/ukončený hneď pri nájdení prvého záznamu, zatiaľ čo *IN* prehľadáva ďalej. Zbytočne tak prechádza celú tabuľku i keď je už možné kladne rozhodnúť o existencií záznamu. Rovnaké pravidlo platí aj v prípade *NOT EXITS* a *NOT IN*. Avšak, ak je to možné, tak je vhodné sa negatívnym operátorom vyhnúť, nakoľko vyžadujú viac času na spracovanie [22].

Filtrovanie výsledkov v klauzule WHERE

Klauzula *WHERE* slúži k obmedzeniu/filtrovaniu výsledkov príkazu *SELECT* a vďaka tomu môže byť príkaz vykonaný rýchlejšie. Výrazné filtrovanie výsledkov je obzvlášť vhodne pri použití agregačných funkcií a pred zoskupovaním záznamov. Jedná sa o veľmi náročné úkony a preto je vhodné, aby sa vykonávali s čo najmenej záznamami.

Ďalším spôsobom ako sa vyhnúť spomaleniu je nepoužívanie funkcií v klauzuly *WHERE*. Prítomnosť funkcie, ktorá vykonáva operáciu nad stĺpcom zabráni databázovému systému využiť indexy, čím sa nedostaví požadované zrýchlenie. Preto je vhodné, ak je to možné, klauzulu *WHERE* vhodne upraviť a dosiahnuť rovnakú funkcionalitu bez využitia funkcie [22].

Ak využívame zástupne znaky v hľadanom reťazci, potom je dobré zamyslieť sa nad tým ako ich používať. Pri normálnom indexovaní sa tieto dotazy môžu občas rozšíriť do veľkých zoznamov slov čo vedie k zníženiu výkonu. Toto sa zlepší v prípade, ak predpony alebo podreťazce sú zaznamenané v indexe [15].

Práca s tabuľkou

Hoci to nie je povinnosťou, každá tabuľka by mala mať primárny kľúč. Prináša to niekoľko výhod ako je indexovanie, unikátne označenie záznamu a predchádzanie duplicitám, dátová konzistencia [16]. Narozdiel od primárneho kľúča, pri vytváraní cudzieho kľúča nie je automatický vytvorený index. Typicky je ale tento index vytvorený manuálne, pretože môže priniesť zlepšenie výkonu [21].

Pri využívaní tabuliek treba byť opatrný. Využívanie privysokého počtu tabuliek v rámci dotazu môže byť neefektívne. Preto je vhodné, aby nevyužívané tabuľky boli z dotazu odstránené [17].

2.3 Možnosti statickej analýzy SQL kódu

Než bude môcť program analyzovať príkaz SQL, tak je vhodné, aby bol spracovaný z jeho pôvodnej textovej podoby do vhodnej dátovej štruktúry. Táto dátová štruktúra musí obsahovať všetky informácie, ktoré sa týkajú sémantiky daného príkazu. Musí byť vhodne usporiadaná, aby bolo možné ňou systematicky prechádzať a vykonávať nad ňou aktivity, ktoré budú predstavovať analýzu. K tomuto bola zvolená dátová štruktúra abstraktného syntaktického stromu.

Nasledovať budú podkapitoly, ktoré sa budú detailnejšie venovať oblasti abstraktného syntaktického stromu. V každom novom úseku textu bude táto dátová štruktúra pomenovaná jeho plným názvom(abstraktný syntaktický strom), pričom v prípade že je jasný kontext textu, tak nasledujúce názvy môžu byť už len v skrátenej podobe(strom). Cieľom je spríjemniť čitateľovi jeho čítanie textu a vyhýbanie sa duplicitným dlhým názvom.

Zároveň v nasledujúcej kapitole bude porovnaných niekoľko programových knižníc, ktoré sú v tomto momente dostupné pre vytvorenie abstraktného syntaktického stromu pre jazyk SQL. Najvhodnejšiu z týchto knižníc potom bude využívať program, ktorý je výstupom diplomovej práce.

2.3.1 Lexikálna a syntaktická analýza

Lexikálna analýza je realizovaná pomocou konečného automatu a predstavuje počiatočnú analýzu súboru so zdrojovým kódom. Pozostáva zo skenera, ktorého vstupom je programový kód, kde kód je čítaný po znakoch a výstupom je množina lexém [10]. Následne je z daných lexém vy-

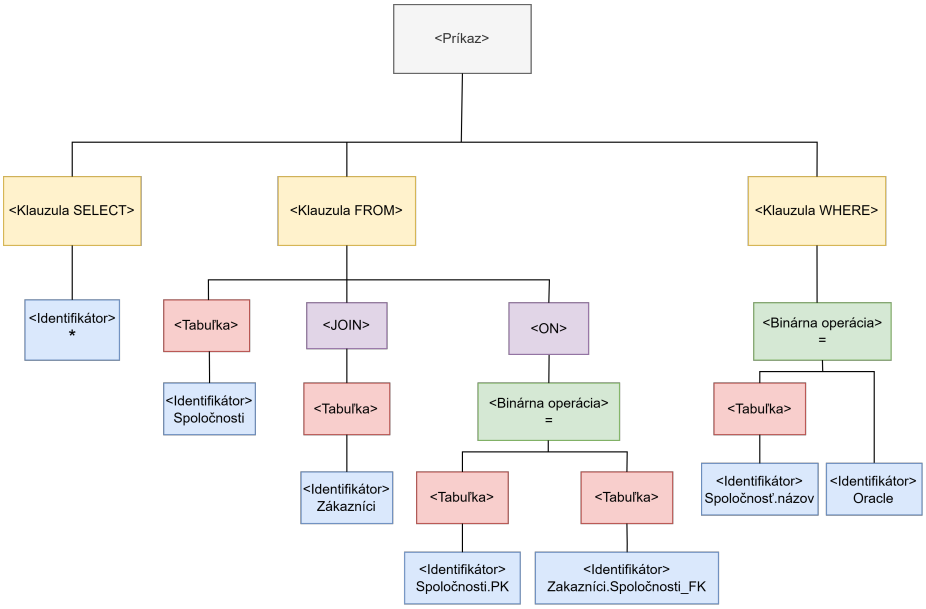
tvorená sekvencia tokenov pomocou procesu tokenizácie[2], ktorá predstavuje roztriedenie lexém do kategórií podľa funkcie [12]. Kategórie môžu pozostávať napríklad z kľúčových slov, identifikátorov, literálov, operátorov, oddeľovačov [18]. Samotné tokeny zároveň obsahujú aj umiestnenie lexémov v kóde. Následné sa vykonáva syntaktická analýza, ktorá prijme vstup vo forme sekvencie tokenov a analyzuje ich správnosť s pravidlami formálnej gramatiky jazyka [8]. Kompilátor vytvára derivačný strom a abstraktný syntaktický strom, pričom ich rozdiel spočíva v množstve informácií uložených v strome . Derivačný strom obsahuje presnú reprezentáciu vstupného textu, zatiaľ čo abstraktný syntaktický strom popisuje vstupný text vo vyššej abstrakcií a neobsahuje informácie, ktoré nie sú potrebné pre spracovanie kontextu. Inak povedané, výstupom syntaktického analyzátora je abstraktný syntaktický strom, ktorý obsahuje len sémanticky významne informácie narozdiel od derivačného stromu, ktorý obsahuje kompletný súbor informácií získaného zo vstupu [1][9].

2.3.2 Abstraktný syntaktický strom

Abstraktný syntaktický strom je stromová dátová štruktúra, ktorá je závislá od hierarchie zápisu. Z toho vyplýva, že pre tvorbu stromu je dôležité poradie príkazov ale nezáleží na štýle zápisu syntaxe, napríklad na koľkých riadkoch je príkaz zapísaný. Abstraktný syntaktický strom pozostáva z uzlov, kde každý z nich nadobúda určitého typu a tie sa odvíjajú od popisovaného zdrojového jazyka. Pre jazyk SQL môže byť príkladom strom príkazu *SELECT*, ktorý má koreňový uzol s názvom *Príkaz*, kde jeho typ je špecifický pre reprezentáciu koreňového uzla príkazu *SELECT*. Z tohoto uzla sa dá následne prechádzať do nižších uzlov. Tie sú typicky iného typu a predstavujú podstromy, ktoré podrobnejšie popisujú SQL kód. Prechodom stromu do hĺbky tak získavame uzly, ktoré predstavujú ďalšie podstromy daného stromu a ktoré tak popisujú konkrétnejšiu časť daného kódu. Tým pádom platí, že so zväčšujúcou sa hĺbkou uzla získavame podrobnejšie informácie o danom kóde.

Tento príklad je graficky znázornený na obrázku 2.1. Z koreňového uzla je možné pristúpiť do uzlov, ktoré už reprezentujú podstrom pre konkrétnu klauzulu príklazu: *SELECT*, *FROM* a *WHERE*. Tieto klauzule sú podrobnejšie definované ich obsahom - podstromom. Na obrázku je typ uzla vyznačený medzi dvoma šípkami, pričom ak má uzol hodnotu, tak tá je znázornená pod typom uzla.

[2] Proces vytvárania tokenov



Obr. 2.1: Abstraktný syntaktický strom jazyka SQL.

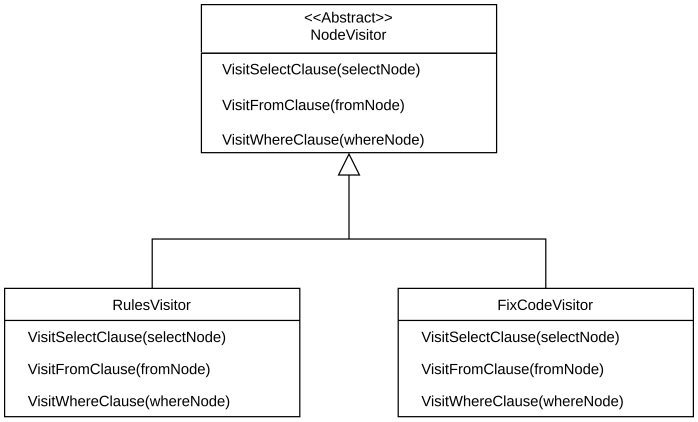
Štruktúra abstraktného syntaktického stromu a typy uzlov sú závislé od implementácie programu, ktorý ho vytvára a môžu sa navzájom líšiť. Avšak, strom musí vždy obsahovať všetky informácie pre správne pochopenie kontextu kódu. Pre analýzu abstraktného syntaktické stromu sa väčšinou používajú dva prístupy: prechod stromu do hĺbky alebo prechod stromu do šírky. Záleží od návrhu a použitia. Prechod stromu do hĺbky sa rovná sekvenčnému spracovaniu kódu, kde prechádzame riadok po riadku tak, že získame všetky informácie o aktuálnom riadku a až potom prechádzame na ďalší. Zatiaľ čo pri prechode do šírky spracovávame kód podľa logických celkov, kde opakovane prechádzame po jednotlivých riadkoch a pri každej iterácií sa informácia o daných riadkoch stáva podrobnejšou.

Abstraktný syntaktický strom je tak vhodná dátová štruktúra pre kontrolu kódu, keďže obsahuje len podstatné informácie ohľadom kontextu programového kódu. Aby sme mohli daný kód kontrolovať, musíme prechádzať uzlami stromu a analyzovať informácie, ktoré sú v nich uložené. Jednotlivé uzly môžu byť rôzneho typu a tým pádom mať o niečo odlišnejšiu štruktúru vzhľadom na iné uzly daného stromu. Pomocou typu uzla a v ňom uložených informácií je možné zistiť, aký úsek kódu uzol popisuje. Prechodom cez strom získavame podrobnejší obraz o kóde, kde cieľom je volať nad jednotlivými uzlami vopred definované pravidlá, ktoré budú špecifiky cieliť na daný typ uzla a zároveň budú volané iba vtedy, keď sa bude prechádzať práve daným typom uzla. K tomuto je možné využiť návrhový vzor Návštevník.

2.3.3 Návrhový vzor Návštevník

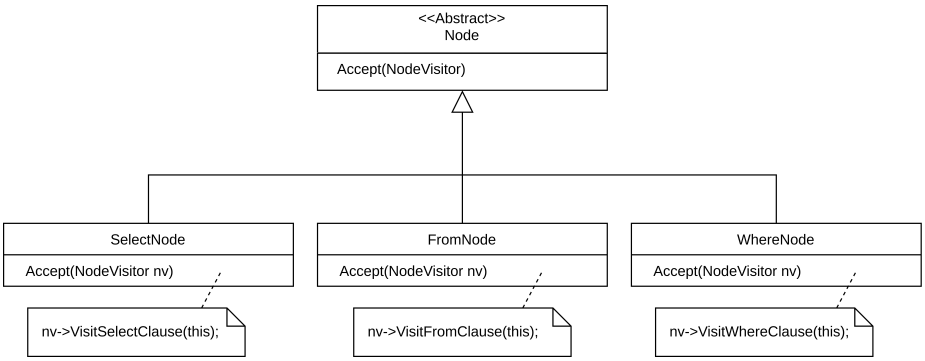
Návrhový vzor Návštevník patrí medzi návrhové vzory správania, ktoré popisujú algoritmy, priradené zodpovednosti medzi objektami a zároveň opisujú aj spôsob komunikácie medzi nimi. Návrhové vzory správania sú vhodné v prípade, že tok riadenia programu je komplexný, čo robí sledovanie programu za behu zložitejším. Uľahčujú vývoj programu tým, že presúvajú pozornosť od zložitého toku riadenia na spôsob ako sú objekty navzájom prepojené. Vzor využíva dedičnosť pre distribúciu správania medzi objektami. V tejto podkapitole boli informácie čerpané z [4, Kapitola 5].

Pre vytvorenie zjednoteného rozhrania všetkých návštevníkov je vytvorená abstraktná trieda návštevníka, ktorá popisuje dané rozhranie. Rozhranie obsahuje abstraktné metódy pre implementáciu vlastnej logiky návštevníka pre každý typ navštevovaného uzlu. Toto rozhranie potom musí implementovať každá trieda, ktorá dedí z abstraktnej triedy návštevník. Každý návštevník bude špecializovaný pre konkrétnu funkcionalitu, ktorá pracuje s uzlami abstraktného syntaktického stromu. Príklad je zobrazený na diagrame tried 2.2. Tento spôsob umožňuje dekompozíciu rôznych funkcionalít nad totožnými uzlami abstraktného syntaktického stromu do rôznych tried. Z toho vyplýva, že môže vzniknúť viac tried návštevníkov, ktoré budú zapudrovať rôznu funkcionalitu, ktorá bude vykonávaná nad uzlami stromu. Pre vytvorenie konkrétneho návštevníka vytvoríme objekt z príslušnej triedy návštevníka.



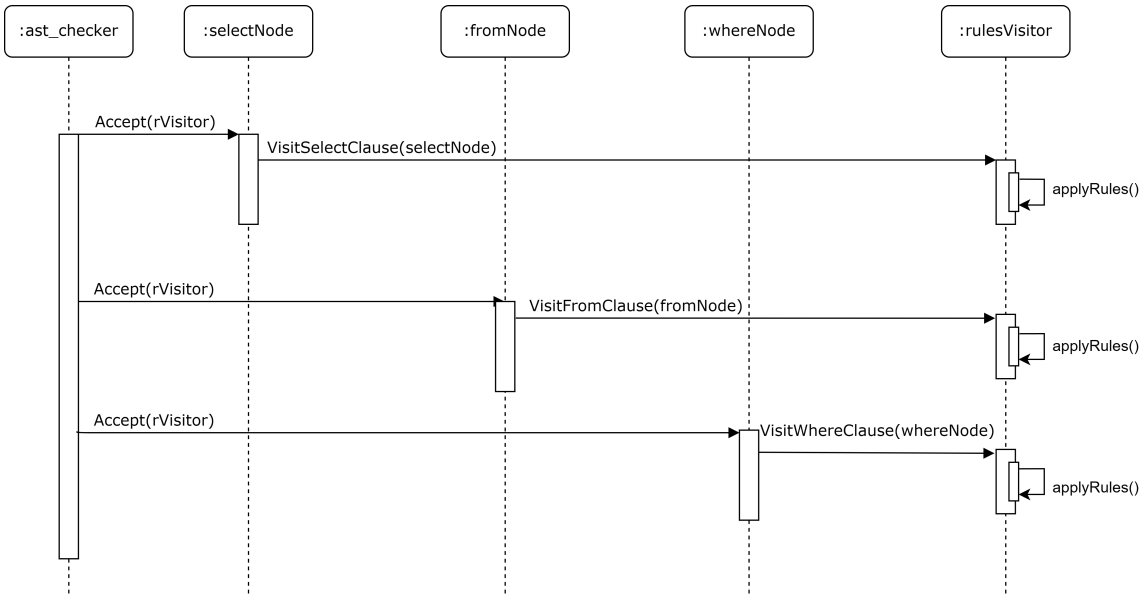
Obr. 2.2: Diagram tried: Návštevník uzlov

Aby objekt návštevníka mohol pracovať s informáciami uloženými v uzloch, musí mať k nim prístup. Pod týmto rozumieme, že návštevník musí byť prijatý daným uzlom. Pre tento účel bude vytvorená abstraktná trieda *Node*, ktorú bude dediť každý uzol abstraktného syntaktického stromu. Táto abstraktná trieda bude obsahovať metódu *Accept*, ktorá bude mať ako parameter návštevníka(z abstraktnej triedy návštevníka). Príklad je zobrazený na diagrame tried 2.3. Tento postup zaručí pri typovej kontrole, že metóda *Accept* bude prijímať každý objekt vytvorený potomkom abstraktnej triedy *Visitor*. Následne návštevník v metóde *Accept* nad sebou zavolá vlastnú metódu pre príslušný uzol, ktorá daný uzol spracuje, kde parametrom metódy je objekt navštíveného uzla.



Obr. 2.3: Diagram tried: Uzol

Na základe toho postupu je možné vytvoriť algoritmus, ktorý bude postupne prechádzať jednotlivé uzly stromu a volať nad nimi metódy. Príklad je znázornený na sekvenčnom diagrame 2.4 . Zároveň, získame tak prehľadnú dekompozíciu problému, kde abstraktný syntaktický strom musí implementovať iba prijatie návštevníka, ktorý má implementáciu spracovania a analýzy jednotlivých uzlov zapuzdrenú v sebe na jednom mieste. Zároveň je tiež možné vytvoriť niekoľko návštevníkov, ktoré budú mať odlišné úlohy nad uzlami stromu. Avšak v rámci programu dochádza k minimálnej zmene a týmto spôsobom sa je možné vyhnúť tomu, aby spracovanie a analýza uzla bola implementovaná priamo v každom uzle stromu. To by pri viacerých funkcionalitách nad uzlami viedlo k ťažko udržateľnému a neprehľadnému kódu.



Obr. 2.4: Sekvenčný diagram: Návrhový vzor Návštevník

- Linter (Program) - Vytvára abstrakciu nad prechádzaním uzlov stromu a vykonávaním operácií nad nimi.
- Select/From/WhereNode (Node) - Predstavujú konkrétne typy uzlov, ktoré sú zaradené do rôznych kategórií.
- RulesVisitor (NodeVisitor) - Objekt vytvorený z triedy *RulesVisitor*, ktorý kontroluje vopred definované pravidla.

2.3.4 Návrhový vzor Adaptér

Ako už bolo spomenuté, program tejto práce bude využívať knižnicu pre tvorbu abstraktného syntaktického stromu. Toto rozhranie stromu z knižnice, nemusí byť kompatibilné s kladenými požiadavkami programu, aby napríklad jednotlivé uzly stromu umožňovali prijatie objektu ľubovolného návštevníka. Cieľom je prispôsobiť rozhranie stromu tak, aby nebol upravený zdrojový kód knižnice. Rozhranie stromu spočíva v rozhraní jednotlivých uzlov, ktorými je strom tvorený. Tie sú pri zostavovaní stromu vhodne prepojené ich referenciami do stromovej štruktúry. V tejto podkapitole boli informácie čerpané z [4, Kapitola 4].

Návrhový vzor Adaptér je štrukturálny návrhový vzor, vďaka ktorému je možné previesť rozhranie triedy na požadované rozhranie. Adaptér umožňuje spoluprácu viacerých tried, ktoré

by inak nemohli spolupracovať kvôli svojim navzájom nekompatibilnými rozhraniam. Realizácia je možná viacerými spôsobmi: pomocou viac násobnej dedičnosti alebo objektovej kompozície. V rámci terminológie sa ustálili nasledujúce pomenovania. Trieda klient, ktorý predstavuje rozhranie klientského programu. Adaptovaná trieda alebo objekt, ktorý predstavuje rozhranie tretej strany, ktoré nie je kompatibilné s rozhraním klienta. Trieda adaptér, ktorá vzniká ako produkt návrhového vzoru adaptér, ktorý je schopný pracovať s adaptovanou triedou a triedou klienta zároveň.

Adaptér vytvorený s pomocou viacnásobnej dedičnosti

Tento prístup pri realizácií využíva vlastnosť, že programovací jazyk podporuje viacnásobnú dedičnosť tried. Adaptér následne zdedí rozhrania z adaptovanej triedy a triedy klienta. Výsledná trieda adaptéra nám tak poskytne prepojenie rozhraní do rozhrania, ktoré je možne ďalej použiť.

Adaptér vytvorený s pomocou objektovej kompozície

V prípade využitia objektovej kompozície, adaptér zdedí len rozhranie klienta a požiadavky voči adaptovanému objektu na neho vhodne deleguje. Výsledkom je opäť rozhranie adaptéra, ktorý dané rozhrania zjednotil pomocou objektovej kompozície.

Kapitola 3

Knižnice pre vytvorenie abstraktného syntaktického stromu

Nasleduje popis knižníc pre tvorbu abstraktného syntaktického stromu, ktoré sú schopné aj kontroly syntaktickej správnosti SQL kódu. Cieľom je sprostredkovať prehľad a porovnanie knižníc, ktoré je vhodné zvážiť pre použitie v programe. Zámerom bolo nájsť čo najvhodnejšie knižnice nehľadiac na programovací jazyk.

3.1 Tree-sitter

Jedná sa o knižnicu, ktorá poskytuje tvorbu syntaktického analyzátora na mnoho programovacích jazykov. Avšak, použitie knižnice je dostupné len na niekoľko z nich(Java, Kotlin, Javascript, Python3, Rust, Ruby a ďalšie) [26]. Ide o projekt s otvoreným kódom a tvorbu syntaktického analyzátora ako aj implementáciu knižnice spravuje komunita prispievateľov[1].

Tvoria ju dve hlavné komponenty: knižnica *libtree-sitter* napísaná v programovacom jazyku C a nástroj pre príkazový riadok *tree-sitter (CLI)* napísaný v programovacom jazyku Rust. *CLI* je nástroj slúžiaci na zostavenie syntaktického analyzátora pre konkrétny jazyk. Syntaktický analyzátor je vytvorený s pomocou vopred definovanej bezkontextovej gramatiky na vstupe, ktorá jazyk popisuje. Výstupom *CLI* je knižnica, ktorá predstavuje vytvorený syntaktický analyzátor. Knižnica *libtree-sitter* využíva syntaktický analyzátor vytvorený pomocou *CLI* k vytvoreniu syntaktického stromu zo zdrojového kódu na vstupe. Zároveň má na starosti zmenu syntaktického stromu ak sa zdrojový kód pozmení. Je navrhnutý tak, aby sa mohol stať súčasťou iných programov a sprostredkováva definované rozhranie nad funkcionalitami [25].

Výhodou je, že knižnica Tree-sitter vytvára generickú vrstvu pre vytvorenie statického analyzátora pre určitý programovací jazyk. Tým vzniká súbor statických analyzátorov s rovnakým rozhraním, ktoré podporujú rôzne jazyky [27].

Momentálnou nevýhodou pre túto prácu je slabšia podpora tvorby syntaktického analyzátoru pre jazyk SQL. Knižnica podporuje tri dialekty jazyka SQL(BigQuery, SQLLite, PostgreSQL), kde každý z nich je samostatným projektom. V dobe písania textu je otázka ich vyspelosti(málo prispievateľov, krátky popis/dokumentácia, slabá aktivita v repozitáry).

[1] https://github.com/tree-sitter/tree-sitter

3.2 Pglast

Ide o knižnicu napísanú v programovacom jazyku Python3, ktorá obsahuje statický analyzátor z databázového systému PostgreSQL. Výstupom knižnice je abstraktný syntaktický strom. Výhodou knižnice je veľké pokrytie SQL jazyka v prípade jeho dialektu PostgreSQL [3]. Avšak, táto knižnica nie je multiplatformová, keďže nie je preložiteľná na operačnom systéme Windows[2,3].

3.3 SQLGlot

Je knižnica napísaná v jazyku Python3 s dôrazom na minimum závislostí. Informácie boli čerpané zdrojov [24]. V dobe písania podporuje až 18 rôznych dialektov jazyka SQL. Okrem syntaktickej analýzy a vytvorenia abstraktného syntaktického stromu má knižnica ďalšie funkcionality:

- Formátovanie a preklad SQL kódu zo zdrojového dialektu na zvolený, ak je podporovaný.
- Poskytnutie API nad metadátami SQL kódu, vďaka čomu je možné získať podrobnosti o príkaze (napríklad výpis všetkých tabuliek, stĺpcov tabuľky).
- Syntaktická analýza, identifikácia chýb v SQL kóde a ich následné zobrazenie na výstupe vhodným spôsobom(zobrazenie úseku chybného kódu).
- Poskytuje API pre vytvorenie vlastného SQL kódu v jazyku Python3.
- Transformovať existujúci abstraktný syntaktický strom, kde výstupom môže byť pozmenený SQL kód s iným významom.
- Optimalizácia SQL kódu.
- Porovnanie dvoch abstraktných syntaktických stromov, kde výstupom sú zmeny, ktoré je nutné vykonať na transformáciu jedného stromu na druhý.

Knižnica je používaná vo viacerých projektoch, ktoré sú uvedené na ich domovskej stránke. Vývoj projektu je stále v aktívnej fáze a autori poskytujú podporu ostatným vývojárom pomocou komunikačného kanálu Slack. Kód je komentovaný iba v jeho hlavných celkoch, z ktorých je generovaná dokumentácia. To môže byť nevýhodou, podrobnejšie komentáre ku kódu absentujú.

3.4 Sqlparser-rs

Je to syntakticky analyzátor navrhnutý priamo pre jazyk SQL, ktorý je v súlade so štandardom ISO/ANSI a napísaný v programovacom jazyku Rust. V dobe písania textu sa autori projektu snažili syntakticky analyzátor priblížiť čo najbližšie k najnovšiemu štandardu SQL:2016. Zároveň je rozšíriteľný a dovoľuje prispôsobenie na iné dialekty [5]. Samotný preklad kódu do abstraktného syntaktického stromu je veľmi rýchly, čo motivovalo vytvorenie iných nástrojov, ktoré knižnicu integrujú do iných programovacích jazykov, napríklad knižnica Sqloxide[4] pre Python3.

[2] https://github.com/lelit/pglast/issues/7
[3] https://github.com/pganalyze/libpg_query/issues/44
[4] https://github.com/wseaton/sqloxide

3.5 Zhodnotenie

Jednotlivé knižnice sa od seba podstatne odlišujú svojimi cieľmi a možnosťami. Tento výstup bude neskôr zohľadnený pri výbere knižnice, ktorú bude program využívať. Nasleduje stručné zhrnutie a porovnanie knižníc.

V prípade pokrytia jazyka SQL sú knižnice Tree-Sitter a Pglast zamerané iba na konkrétne dialekty a Sqlparser-rs na štandard. Knižnica SQLGlot ako jediná podporuje viacero SQL dialektov zároveň. Keďže je cieľom programu pokryť čo najväčší rozsah jazyka SQL, prípadne jeho štandard, tak ako prijateľne riešenia obstáli knižnice SQLGlot a Sqlparser-rs.

Pri porovnaní statusu vývoja a aktivity autorov sú knižnice takmer identicky dobré, okrem knižnice Tree-Sitter. Hodnotila sa hlavne aktivita na repozitáry, prípadne na komunikačnom kanály, ak ho daný projekt má vytvorený.

Naopak knižnice Tree-Sitter a Pglast sú vhodnejšie z hľadiska použitia pri porovnaní, ktoré sa zaoberá podporou informácie o pozícií kódu v abstraktnom syntaktickom strome. Tým je myslené, či jednotlivé uzly stromu obsahujú aj informáciu, z akého úseku kódu boli vytvorené. Táto informácia je potrebná pri vytváraní hlásení o kóde SQL, aby bolo možné sa odkazovať na úsek a polohu kódu vo výpise programu. Knižnica Sqlparser-rs sa snaží túto funkcionalitu pridať[5], avšak v čase riešenia diplomovej práce, dané riešenie nebolo plnohodnotne dokončené[6,7]. V prípade knižnice SQLGlot bola vedená diskusia s autormi knižnice ohľadom riešenia podpory/implementácie pre danú funkcionalitu cez oficiálny Slack kanál, avšak bez úspechu.

Nakoľko po určitom zhodnotení sa knižnice SQLGlot a Sqlparser-rs ukazovali ako najlepší kandidáti na použitie, tak bola u tých dvoch knižníc, v rámci diplomovej práce, vykonaná analýza a experimenty pre doplnenie informácie o pozícií do uzlov abstraktných syntaktických stromov. V prípade Sqlparser-rs boli experimenty neúspešné, vzhľadom na zložitejšiu implementáciu v programovacom jazyku Rust. V prípade knižnice SQLGlot sa ukázal tento postup ako realizovateľný.

[5] https://github.com/sqlparser-rs/sqlparser-rs/issues/757
[6] https://github.com/sqlparser-rs/sqlparser-rs/pull/790
[7] https://github.com/sqlparser-rs/sqlparser-rs/pull/839

Kapitola 4

Rozbor existujúcich riešení pre statickú analýzu programového kódu

Táto kapitola popisuje existujúce riešenia/nástroje pre statickú analýzu programového kódu. Všetky vybrané nástroje majú verejne dostupný kód a vďaka tomu je ich možné analyzovať aj z pohľadu programového kódu.

Nasledujúce podkapitoly popisujú jednotlivé riešenia zvlášť, kde sú popísané ich slabé/silné stránky a princípy fungovania. Posledná podkapitola je zameraná na ich porovnanie a zhrnutie.

4.1 SQLFluff

Je to statický analyzátor kódu určený pre SQL kód, pričom podporuje mnoho dialektov a je napísaný v programovacom jazyku Python3. V čase písania ide o aktívne vyvíjaný projekt, ktorý spravuje komunita ľudí. Je ho možné využiť v prostredí príkazového riadku. Vstup očakáva v podobe SQL kódu, ktorý môže byť umiestnený v samostatnom súbore alebo zadaný po spustení cez štandardný vstup. SQLFluff je určený k detekcií štylistických chýb a čiastočne kontextu v rámci daného SQL príkazu. Užívateľ môže definovať vlastné pravidlá pre požadovanú kontrolu SQL kódu. Pri analyzovaní kódu využíva vlastnú implementáciu tvorby abstraktného syntaktického stromu. Výstupom je prehľad nájdených chýb, ktorý je umiestnený na štandardnom výstupe. Ten je prehľadne formátovaný a obsahuje informácie o polohe chyby v kóde, identifikátor chyby a popis chyby. Pri niektorých chybách poskytuje aj ich opravu a výstupom tak môže byť aj upravený SQL kód. Chovanie programu je možné špecifikovať nastavením programových parametrov. Informácie o tomto programe boli získavané zo zdroja [20] a priloženej dokumentácie na tomto zdroji[1].

Možnosti spustenia a funkcionality

Možnosti spustenia je možné rozdeliť do niekoľkých funkčných kategórií. Každý z nich je špecifikovaný pomocou kľúčového slova umiestneného ako prvý argument pri spustení programu. Jednotlivé funkcionality nástroja majú dodatočne ďalšie možnosti spustenia pomocou ďalších parametrov. Cieľom nasledujúceho textu je popísať iba hlavné body funkcionalít, ktoré nástroj ponúka. To má ponúknuť čitateľovi predstavu akými hlavnými možnosťami nástroj disponuje, pričom detailnejšie možnosti konfigurácie je možné nájsť v dokumentácií.

[1] https://docs.sqlfluff.com/en/stable/

Základnou skupinou parametrov je možné získať detailnejšie informácie o nástroji. Môžeme získať jeho verziu, výpis dialektov, s ktorými vie daná verzia pracovať a skupinu pravidiel, ktoré nástroj registruje. Tieto informácie sú schopné pomôcť pri riešení prvotných chýb v prípade, ak bolo pridané nové pravidlo a neprejavilo sa pri jeho testovaní.

Medzi jednoduché funkcionality patrí tiež výpis vstupu, ktorý je nástroju poslaný. Pre ukážku ďalších funkcií bol vytvorený SQL kód, ktorý má zámerný zlý štýl zápisu, ktorý SQLFluff vie rozpoznať. Vstupný kód vyzerá nasledovne:

```
1 SELECT table1.col5,
2 col6, col7 AS c7
3 from table1, table2, table3 as t3 WHERE table1.col1=table2.col2;
```

Medzi hlavné funkcionality patrí príkaz pre začatie kontrolu kódu - *lint*. Používanie je jednoduché a priamočiare, v minimálnej konfigurácií stačí špecifikovať vstupný súbor s SQL kódom a nastavenie cieľového dialektu parametrom *-d*, ktorý je v našom prípade SQL štandard ANSI. Okrem iného je možné špecifikovať, aké pravidlá sa majú kontroly účastniť, v prípade ak nie je úmyslom zahrnúť do kontroly všetky. Tiež je dovolené zmeniť vnútornú programovú konfiguráciu priložením vlastného konfiguračného súboru. Výstup je v základe formátovaný, aby bol dobre čitateľný pre človeka, avšak je ho možné špecifikovať aj do serializovaných formátov: *json*, *yaml*, *github-annotation*, *github-annotation-native*. Pre spomenutý SQL kód vyššie, podáva SQLFluff nasledujúci výstup po kontrole kódu:

```
$ sqlfluff lint testfile.sql -d ansi
== [testfile.sql] FAIL
L:   1 | P:  1 | L036 | Select targets should be on a new line unless there is
                       | only one select target.
L:   1 | P:  7 | L003 | Expected line break and indent of 4 spaces before
                       | 'table1'.
L:   2 | P:  1 | L003 | Expected indent of 4 spaces.
L:   2 | P:  8 | L027 | Unqualified reference 'col6' found in select with more
                       | than one referenced table/view.
L:   2 | P: 14 | L027 | Unqualified reference 'col7' found in select with more
                       | than one referenced table/view.
L:   3 | P:  1 | L010 | Keywords must be consistently upper case.
L:   3 | P: 29 | L010 | Keywords must be consistently upper case.
L:   3 | P: 32 | L025 | Alias 't3' is never used in SELECT statement.
L:   3 | P: 32 | L031 | Avoid aliases in from clauses and join conditions.
L:   3 | P: 52 | L006 | Expected single whitespace between naked identifier
    and
                       | raw comparison operator '='.
L:   3 | P: 53 | L006 | Expected single whitespace between raw comparison
                       | operator '=' and naked identifier.
All Finished!
```

Ako bolo zmienené, ak vstupný kód nevyhovie nejakým pravidlám, potom niektoré problémy v kóde dokáže nástroj opraviť. Táto funkcionalita má iné kľúčové slovo pre spustenie - *fix*. Najskôr zobrazí nájdené chyby rovnako ako pri predchádzajúcej kontrole chýb a potom sa opýta, či užívateľ súhlasí s opravou. V prípade, že vstup je súbor s SQL kódom, potom v základnej konfigurácií nástroj upraví daný kód v súbore čo môže viesť k strate pôvodného kódu. Avšak nástroj podporuje cez dodatočný parameter vytvorenie nového súboru s doplneným, užívateľom definovaným, sufixom k pôvodnému názvu. Výsledný kód po úprave:

```
1 1 SELECT
2 2     table1.col5,
3 3     col6,
4 4     col7 AS c7
5 5 FROM table1, table2, table3 WHERE table1.col1 = table2.col2;
```

Nástroj bol testovaný s pravidlami, ktoré sú dodané v rámci daného nástroja. Ako je možné vidieť, obsahuje v základe aj pravidlá, ktoré nemusia byť na očakávané, napríklad pravidlo s kódom *L031*[2], ktoré označuje využitie alias tabuliek za chybný zápis. Toto pravidlo je označené v dokumentácií ako kontroverzné. Popis pravidla vysvetľuje, že alias tabuľky môže viesť k zmäteniu čitateľa a nemusia byť tak užitočné, skôr naopak. Zároveň pripúšťajú, že pre väčšie databázy by malo byť toto pravidlo vypnuté, nakoľko vyhýbanie sa aliasom nie je realistické ani žiadúce. Rovnako stojí za pozornosť, že nástroj ponechal logické celky *FROM* a *WHERE* na jednom riadku a nepovažuje to za štylistickú chybu, aby ich oddelil na samostatné riadky.

Pre viac detailov k detekcií chýb kódu je možné pridať ďalšie parametre. Avšak, väčší výpis skôr len kombinuje už existujúce informácie z iných funkcionalít(zobrazenie syntaktického stromu a konfiguráciu nástroja).

Tvorba pravidiel

Každé pravidlo predstavuje vlastnú triedu, ktorá dedí funkcionalitu z triedy *BaseRule*. Táto trieda musí implementovať metódu *_eval*, kde sa predpokladá logika daného pravidla. Trieda často obsahuje aj autorom vytvorené metódy, pre lepšiu dekompozíciu kódu. Vstupom metódy *_eval* je spracovaný úsek kódu, nad ktorým sa vykonáva dané pravidlo. Návratová hodnota metódy je voliteľná, ak je chyba nájdená, potom je vytvorené hlásenie o chybe.

4.2 Pylint

Ide o aktívne vyvíjaný staticky analyzátor pre kód programovacieho jazyka Python2 a Python3. Je možné ho využívať cez príkazový riadok a zároveň podporuje aj integráciu do vývojových nástrojov. Nástroj je publikovaný s otvoreným kódom a umožňuje pridať vlastné pravidlá pre kontrolu kódu. Výstupom je zoznam nájdených chýb so špecifikovaným riadkom a krátkou správou. Pre jednoduchú orientáciu kvality kódu je dostupné skóre kódu, pričom po zmene kódu a opätovnej analýze nástroj upozorní na zlepšenie/zhoršenie. Informácie o tomto programe boli získavané zo zdroja [23] a priloženej dokumentácií v tomto zdroji[3].

Možnosti spustenia a funkcionality

Nástroj poskytuje možnosť výpisu textu alebo plnej dokumentácie, ktorý slúži ako manuál k použitiu a popisuje jednotlivé parametre nástroja.

Použitie je priamočiare, na vstupe nástroja sa predpokladá programový kód Python2 alebo Python3 uložený v súbore. Detailnosť generovaného výstupu je možné špecifikovať, pričom ďalšie informácie sa týkajú štatistických informácií o kóde. Nakoľko jazyk Python3 má oficiálnu dokumentáciu pre správny Python3 kód, tak nie je potrebné nič extra konfigurovať. Nástroj predpokladá, že užívateľ sa zameriava práve na tento typ kontroly.

Pre ukážku bol vytvorený jednoduchý kód v Python3:

[2] https://docs.sqlfluff.com/en/stable/rules.html#rule-aliasing.forbid
[3] https://pylint.readthedocs.io/en/latest/

```
1 1 #!/usr/bin/env python3
2 2
3 3 def Fun_1(parameter1):
4 4     print("Something")
5 5
6 6 if __name__ == '__main__':
7 7     Fun_1("Not used")
```

Výstupom je jednoduchý prehľad neželaných chýb. Nakoľko nástroj vie kontrolovať väčšie množstvo súborov naraz, tak prvotnou informáciou je v akom súbore sa daná chyba nachádza. Následne na akom riadku a pozícií so sprievodným textom ku chybe. Na záver je doplnený názov typu chyby, podľa ktorého je možné identifikovať zodpovedné pravidlo. Na záver je výsledok ohodnotenia kódu vo forme skóre.

```
1  pylint testfile.py
2  ************* Module testfile
3  testfile.py:4:0: W0311: Bad indentation. Found 1 spaces, expected 4
       (bad-indentation)
4  testfile.py:5:0: C0303: Trailing whitespace (trailing-whitespace)
5  testfile.py:7:0: W0311: Bad indentation. Found 1 spaces, expected 4
       (bad-indentation)
6  testfile.py:1:0: C0114: Missing module docstring (missing-module-docstring)
7  testfile.py:3:0: C0103: Function name "Fun_1" doesn't conform to snake_case
       naming style (invalid-name)
8  testfile.py:3:0: C0116: Missing function or method docstring
       (missing-function-docstring)
9  testfile.py:3:10: W0613: Unused argument 'parameter1' (unused-argument)
10
11 -----------------------------------
12 Your code has been rated at -7.50/10
```

Tvorba pravidiel

Pravidlá sú rozdelené do troch kategórií:

- Kontrola na neupravenom prúde dát, kde zdrojový kód nie je spracovaný do štruktúry.
- Zdrojový kód je spracovaný a výstupom je zoznam tokenov, ktoré jednotlivo reprezentujú určité časti kódu. Jednotlivé tokeny sú následne analyzované pravidlami kontroly kódu.
- Zdrojový kód je spracovaný, kde výstupom je abstraktný syntaktický strom, ktorý reprezentuje celý kód. Následne sa daným stromom prechádza a na jednotlivé uzly stromu sa volajú pravidlá pre kontrolu danej časti kódu. K vytvoreniu abstraktného syntaktického stromu je použitá knižnica *astroid*, ktorá využíva a rozširuje vstavaný modul jazyka Python3 pre generovanie stromu[4,5].

Nasledujúci text sa bude venovať tvorbe pravidiel využívajúcich abstraktného syntaktického stromu. Každé pravidlo predstavuje triedu, pričom táto trieda dedí z abstraktnej triedy *BaseChecker*, ktorá je spoločná pre všetky pravidlá a ktorá definuje rozhranie pravidla. Aby bolo pra-

[4]https://docs.python.org/3/library/ast.html
[5]https://github.com/pylint-dev/astroid#whats-this

vidlo zapojené do kontroly tak musí byť nutne registrované pomocou metódy *register*, ktorá sa nachádza v rovnakom súbore s triedou pravidla a na globálnej úrovni.

Trieda pravidla tiež obsahuje názov pravidla a slovník chybových správ, kde správa obsahuje svoje číselné pomenovanie(krátky identifikátor), textové pomenovanie(dlhý identifikátor) a popis chyby. Pomocou textového pomenovania je môže danú správu pridať do výsledku kontroly. Voliteľnou možnosťou je definovanie ďalších parametrov pravidla. Podstatnou časťou triedy je definovanie metód, ktoré sa starajú o kontrolu kódu - programovanie funkcionality pravidiel. Názov každej metódy, ktorá obsahuje kontrolu kódu je pevne daný(nemôže byť ľubovolný) a skladá sa z dvoch komponent. Prvým z nich je kľúčové slovo *visit* alebo *leave*, ktoré definuje moment kedy bude dané pravidlo zavolané. Keď nástroj prechádza cez abstraktný syntaktický strom, prechádza cez uzly stromu, ktoré sú pomenované a definujú konkrétne časti kódu. Týmto spôsobom je možné špecifikovať či pravidlo bude zavolané pri navštívení uzla alebo pri jeho opustení. Ďalšou časťou názvu metódy je špecifikovanie typu uzla. Nástroj vie vďaka pomenovaným uzlom akú časť kódu prechádza a preto je možné rozdeliť uzly do typov napríklad cyklu, podmienky, výrazu a podobne, kde tieto typy uzlov zodpovedajú analyzovanému programovaciemu jazyku Python. Kombináciou týchto dvoch komponent získavame celý názov metódy, ktorá môže byť napríklad *visit_return* alebo *leave_if*. Trieda pravidla môže obsahovať týchto metód viac a môže tak pokryť kontrolu viacerých typov uzlov. Parametrom týchto metód je daný typ uzla stromu. Nakoľko jazyk Python je dynamicky typovaný, definícia štruktúry uzla v parametre je len napovedá aký typ parametru sa očakáva a môžu ju využiť aj staticky analyzátor pre kontrolu kódu ak je v dobe pred spustením zrejmé, aký typ môže vstupný parameter nadobúdať.

4.3 Zhodnotenie

Okrem rozdielneho cieľového jazyka pre kontrolu sa tieto nástroje pre statickú analýzu kódu príliš nelíšia. Obsahujú podobné typy funkcionality, ktoré sú potrebné pre kontrolu vstupného kódu. Oba nástroje dovoľujú užívateľovi vytvoriť si vlastné pravidlo a zapojiť ho do kontroly. Spôsob kontroly je značne konfigurovateľný. Pravidlá môžu obsahovať kontrolu pre rôzne typy uzlov stromu a definície výstupných správ. Tieto pravidlá môžu byť z kontroly odobrané pomocou nastavenia konfiguračných súborov. Výstupom je prehľadný zoznam chýb, ktorý je vhodne formátovaný. Obsahuje popis chyby, umiestnenie v kóde a informácie o použitom pravidle. Výhodou Pylint je pokročilejšia kontextová kontrola. Tá dovoľuje objaviť nový súbor chýb alebo neželaných konštrukcií v kóde. Oba nástroje zároveň poskytujú návrhy na opravu kódu. Výhodou SQLFluff môže byť funkcionalita, ktorá vie tieto zmeny do istej miery začleniť do kódu a vyprodukovať tak nový kód. Samozrejme, dané zmeny sa nemôžu vzájomne vylučovať a SQLFluff tento konflikt rieši nastavením maximálneho počtu iterácií, v ktorých sa kód môže zmeniť.

Kapitola 5

Návrh

Cieľom práce je vytvorenie nástroja vo forme programu, ktorý bude schopný analyzovať kód jazyka SQL po jeho syntaktickej a kontextovej stránke. Zároveň je cieľom nezávislosť implementácie od existujúcich riešení databázových systémov. Program by mal byť schopný fungovať aj bez prítomnosti databázových systémov v počítači, prípadne využívať ich služby len ako doplnok. Z tohoto vyplýva, že program bude schopný spracovať SQL kód samostatne, avšak s prihliadnutím na implementáciu existujúcich databázových systémov. Cieľom je, aby spracovanie SQL kódu bolo medzi programom a databázovým systém kompatibilné. Je to nutné v dôsledku, že program bude predstavovať medzikrok pred spustením kódu v databázovom systéme a preto je dôležité, aby program kontextovo spracovával príkazy rovnako ako cieľový databázový server.

5.1 Vstup programu

Vstupom programu bude kód SQL. Tento kód môže predstavovať jeden príkaz alebo viacero na seba nadväzujúcich príkazov jazyka SQL, ktoré môžu byť doplnené o komentáre. Komentáre kódu alebo akýkoľvek text, pri ktorom nie je cieľom jeho spracovanie je nutné odlíšiť od zvyšku kódu prostredníctvom znakov, ktoré identifikujú začiatok komentára, pripadne aj jeho koniec. Text je chápaný ako komentár, pokiaľ v rámci rovnakého riadku nasleduje za dvojicou bezprostredne idúcich pomlčiek (`--`) alebo sa nachádza na riadkoch medzi dvojicou znakov reprezentujúci blokový komentár(`/*` a `*/`). Nasleduje jednoduchý príklad vstupu:

```
1  /*
2   Comment block
3   Input example
4  */
5
6  -- Line comment between empty lines
7  CREATE TABLE Table1 (
8          PersonID INT NOT NULL PRIMARY KEY,
9          FirstName VARCHAR(255) NOT NULL,
10         LastName VARCHAR(255) NOT NULL,
11         Address VARCHAR(255), -- Line comment inside SQL statement
12         Age INT(3)
13 );
14
15 SELECT LastName
16 FROM Table1;
```

Pri zápise SQL príkazov bude program požadovať, aby koniec každého príkazu bol symbolizovaný bodkočiarkou(`;`). V prípade konvencií písania kódu v existujúcich databázových systémoch toto nemusí byť vždy pravidlom ako napríklad u dialektu T-SQL od Microsoftu. Microsoft zároveň ale deklaruje, že túto vlastnosť v budúcich verziách T-SQL odstráni [13]. Zároveň z hľadiska štandardu SQL je ukončenie príkazu bodkočiarkou vyžadované [11]. V rámci implementácie bude vyžadovanie symbolu ukončenia príkazu zároveň zjednodušením algoritmu, ktorý bude príkazy identifikovať.

Vo výsledku bude program podporovať viacero spôsobov ako vstup zadať. Vstupný SQL kód bude môcť byť uložený v súbore, na ktorý sa bude možné v programe odkazovať pomocou programových parametrov, podrobnejšie v podkapitole 5.2. Pokiaľ cesta ku vstupnému súbor nebude zadaná, program poskytne možnosť zadať vstupný SQL kód na je štandardný vstup po spustení. Poslednou možnosťou je pripojenie sa na spustený databázový server, odkiaľ môže byť získaný SQL kód pre inicializáciu pamäťovej reprezentácie. Táto možnosť je len doplnkom ku predchádzajúcim dvom, kde získaný SQL kód z databázy nebude kontrolovaný. Podrobnejší v podkapitole 5.3, ktorá popisuje reprezentáciu štruktúry v pamäti.

V momente keď program obdrží vstupný SQL kód, tak ho môže začať spracovávať. Výstupom spracovania bude zoznam, kde každý prvok zoznamu reprezentuje jeden SQL príkaz. Začiatok spracovania predstavuje odstránenie blokových komentárov, jednoriadkových komentárov a prázdnych riadkov. Oboje neovplyvňujú význam SQL príkazu a preto môžu byť bezpečne odstránené. Tieto kroky sú nutné, nakoľko v ďalšom kroku rozdelíme súbor podla symbolu ukončenia príkazu (`;`), ktorý sa môže nachádzať aj v komentároch čo by vyústilo v nesprávne rozdelenie. Týmto získavame zoznam s výstupnými SQL príkazmi.

5.2 Parametre programu

Programy môžu ponúkať aj viacero funkcionalít. Užívateľ, ktorý využíva program by mal mať možnosť výberu z týchto funkcionalít alebo možnosti nastavenia programu jednoduchou a intuitívnou cestou. Keďže sa jedná o program, ktorý nebude disponovať užívateľským grafickým rozhraním, tak všetky možnosti nastavenia programu budú dostupné prostredníctvom programových parametrov a konfiguračných súborov uložených na pamäťovom médiu. Užívateľ sa potom môže na tieto konfiguračné súbory odkázať prostredníctvom cesty, ktorú zadá v rámci príslušného parametru programu. Zároveň pre niektoré parametre bude program obsahovať predom definované cesty, kde bude v predvolenom nastavení očakávať daný konfiguračný súbor v prípade, že užívateľ nešpecifikuje inú cestu. V prípade, že užívateľ zadá namiesto absolútnej cesty iba relatívnu, potom to program vyhodnotí ako cestu, ktorá začína od koreňového adresára programu. Tieto a ďalšie parametre programu budú teraz podrobnejšie rozpísané v jednotlivých podkapitolách, ktoré budú predstavovať časť funkcionalít programu, ktoré je možné ovládať pomocou programových parametrov.

5.2.1 Základné parametre

Neskúsený užívateľ s programom môže potrebovať rýchly pomocník ako s programom pracovať. Pre tieto prípady bude program obsahovať parameter *{help}*, ktorý vypíše na štandardný výstup všetky dostupné parametre programu a popis k čomu dané parametre slúžia. Pre špecifikáciu cesty vstupného súboru s kódom SQL, ktorý má byť kontrolovaný, slúži parameter *{file}*. Ďalším parametrom je *{dialect}*, ktorým užívateľ môže ovplyvniť knižnicu, ktorá vytvára abstraktný syntaktický strom. Ak tento parameter nebude špecifikovaný, potom bude knižnica používaná bez zvoleného dialektu. Posledným základným parametrom je *{verbose}*, ktorý určuje

mieru podrobnosti hlásení programu o jeho behu na výstup. V prípade, že daný parameter je zadaný, program produkuje na výstup aj podrobné hlásenia o práve vykonávaných akciách, inak len základné správy ohľadom kontroly kódu. Miera podrobnosti záleží od úrovne zadaného parametra.

5.2.2 Pravidlá

Pravidlá sú základnou súčasťou kontroly SQL príkazu, pretože použité pravidla budú definovať rozsah alebo typ kontroly. Cieľom je, aby hierarchia uloženia pravidiel mohla byť ľubovolná a v prípade preferencií zmenená užívateľom. To znamená, že program bude obsahovať sadu pravidiel, ktoré budú uložené v rámci adresárov programu ale štruktúra týchto adresárov ani cesta k nim nesmie byť nemenná, podrobnejšie fungovanie v podkapitole 5.4. Z tohoto hľadiska program podporuje zmenu cesty k adresáru s pravidlami parametrom *{rules-path}*, kde hodnotou je cesta ku pravidlám, pričom budú registrované všetky pravidlá v danom adresári a jeho podadresároch. Zároveň je možnosť aplikovať iba niektoré pravidlá v danom adresári. Túto vlastnosť je možné vynútiť jedným z dvojice parametrov, pričom tieto parametre sa vzájomne vylučujú a nemôžu byť použité naraz. Prvý spôsob je výber podmnožiny adresárov, ktoré majú byť zahrnuté do kontroly. Toto je možné dosiahnuť parametrom *{include-folders}*, kde hodnota parametra sú názvy adresárov. Druhým spôsobom je vylúčenie podmnožiny adresárov z kontroly, kedy sa použijú ku kontrole všetky pravidla okrem tých, ktoré sa nachádzajú vo vylúčených adresároch. Toto je možné dosiahnuť s parametrom *{exclude-folders}*, kde hodnota parametra sú názvy vylúčených adresárov.

5.2.3 Výstupné hlásenia

Počas svojho behu môže program užívateľovi zobrazovať na výstup rôzne hlásenia. Tieto hlásenia sa môžu týkať rôznych informácií: procesu prihlásenia sa k databázovému serveru, upozorneniu o zmene pamäťovej reprezentácie, ktorá porušuje integritu alebo varovanie o neočakávanej kombinácií programových parametrov. Tieto a ďalšie hlásenia sú predmetom behu programu. Hlásenia, ktoré sú získané počas kontroly pravidiel, budú vypísané až na konci behu programu.

Užívateľ si môže pomocou programových parametrov zvoliť miesto výstupu. Ak neuvedie žiadny z nasledujúcich parametrov, potom bude automaticky zvolený predvolený výstup, ktorým je štandardný výstup na príkazový riadok. Výstup do súboru je možné aktivovať parametrom *{report-output-file}*, kde očakávaná hodnota parametru je cesta, kde sa bude súbor nachádzať. V prípade, že užívateľ si neželá žiadne hlásenia, môže využiť parameter *{report-output-nothing}*. Zároveň platí, že tieto parametre sa vzájomne vylučujú a nemôžu byť použité spoločné.

5.2.4 Dodatočné parametre

Nasledujúce parametre nepredstavujú ovládanie hlavnej funkcionality programu. Návrh programu bol rozšírený o ďalšie funkcionality s cieľom ponúknuť užívateľovi možnosť inicializovať alebo uložiť stav programu. Stavom programu sa myslia dátové štruktúry programu, ktoré majú vplyv na proces kontroly SQL kódu.

Pripojenie k databáze

Pre získanie skriptu SQL z existujúceho databázového systému je nutné poznať viacero parametrov pripojenia k databázovému systému. Tieto parametre sa musia nachádzať v súbore s INI formátom. Pre prípady, keď užívateľ nevie ako takýto súbor zostaviť, program obsahuje parameter *{connection-file-create}*, ktorý vytvorí súbor s názvom *db_connection.cfg* v výstupnom adresári programu. Tento súbor predstavuje šablónu s dodatočným popisom a príkladom ako vytvoriť vlastnú konfiguráciu. Užívateľ môže v tomto súbore vytvoriť viacero konfigurácií pomocou sekcií a môže sa tak pripájať k rôznym databázam podľa voľby. Tieto sekcie musia mať tým pádom odlišný názov a zároveň sú rozlišované aj na základe veľkosti písmen. Pre zvolenie konfigurácie, ktorú má program použiť je možné túto informáciu programu poskytnúť cez parameter *{connection-file-option}*, kde predpokladaná hodnota parametru je názov sekcie v konfiguračnom súbore. Ak je tento argument zadaný, tak zároveň to signalizuje programu, že túto funkcionalitu má použiť a pripojiť sa na databázový server. Užívateľ má zároveň možnosť použiť aj konfiguračný súbor, ktorý je umiestnený na inej ceste. Ku špecifikovaniu tejto cesty slúži parameter *{connection-file-path}*, ktorého hodnota reprezentuje cestu ku danému súboru. Ak tento parameter nie je zadaný ale funkcionalita je aktivovaná, potom sa použije predvolená cesta, ktorá je zhodná s cestou kde sa vytvára šablóna ku konfiguračnému súboru. Nasleduje spomenutého popísaného konfiguračného súboru:

```
1  # Example of INI file
2
3  # Oracle database configuration
4  [ORACLE]
5  DIALECT = oracle # database dialect
6  USERNAME = User1 # username
7  PASSWORD = uFeDJCUswu # password
8  HOST = localhost # host url
9  PORT = 1521 # port number
10 SERVICE = orcl.mshome.net # database service name
```

Uloženie a obnovenie pamäťovej štruktúry

Nesmierne dôležitý je stav pamäťovej reprezentácie v dobe kontroly SQL príkazu. Daná kontrola z nej môže využívať informácie o stave a prítomnosti reprezentácie databázových objektov. Pamäťová reprezentácia sa vyvíja v čase a pre užívateľa môže byť dôležité z času na čas zachovať daný stav vo forme súboru. Program túto funkcionalitu podporuje prostredníctvom programového parametra *{serialization-path}*, kde hodnotou parametru je cesta, kde sa má uložiť súbor s pamäťovou reprezentáciou. Pre jej opätovné načítanie je možné použiť parameter *{deserialization-path}*, kde hodnotou parametru je cesta, kde program nájde súbor s pamäťovou reprezentáciou. Pre oba parametre platí, že argument funguje zároveň aj ako aktivátor danej funkcionality a v prípade, že hodnota cesty nie je zadaná, tak za použije predvolená cesta do výstupného adresára programu.

5.3 Reprezentácia schémy v pamäti

Reprezentácia schémy v pamäti(skrátene pamäťová reprezentácia) predstavuje dátovú štruktúru, ktorá ako celok reprezentuje databázovú schému v programe. Je tak tvorená objektami, ktoré reprezentujú obsah databázovej schémy. Chovanie a štruktúra pamäťovej reprezentácie

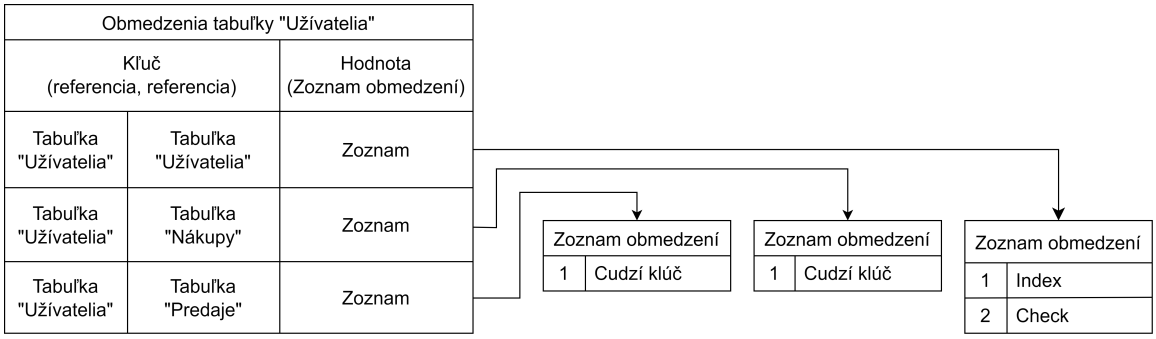
je ovplyvnené cieľovým databázovým systémom. Vo výsledku to znamená, že nakoľko sú implementácie a chovanie databázových systémov rôzne, tak aj pamäťová štruktúra môže v závislosti na cieľovom systéme povoliť iba určitú podmnožinu svojich funkcionalít, ktoré sú kompatibilné s daným databázovým systémom.

Cieľom pamäťovej reprezentácie je zaznamenávanie a uchovávanie zmien z príkazov SQL, ktoré majú za účel pozmeniť štruktúru databázovej schémy. Tie príkazy, ktoré majú túto vlastnosť, sú dodatočne spracované programom tak, aby mohla byť vykonaná taká akcia nad pamäťovou reprezentáciou, ktorá je ekvivalentná voči akcií databázového systému, ktorý spracuje daný SQL príkaz. Výsledkom je štruktúra, ktorá obsahuje také informácie o existencií objektov, ich dátach a vzájomných vzťahoch, že môžu byť použité pre kontrolu kontextu medzi jednotlivými spracovávanými príkazmi SQL.

V hierarchií pamäťovej reprezentácie je koreňovým objektom databáza, vytvorená z triedy *Database*. Obsahuje svoj názov, predvolený názov schémy, množinu schém, ktoré obsahuje a objektový index. Objektový index je dátová štruktúra typu kľúč-hodnota, kde kľúčom je usporiadaná n-tica pozostávajúca z názvov objektov, cez ktoré je nutné hierarchicky prejsť, aby sme sa dostali k výslednému objektu a hodnota je referencia na daný objekt. Týmto získavame dátovú štruktúru, cez ktorú vieme rýchlo získať konkrétny objekt bez toho, aby sme prechádzali celou štruktúrou pamäťovej reprezentácie. Uložený objekt v databáze reprezentujúci databázovú schému je objekt vytvorený z triedy *Schema*. Každý vytvorený objekt schémy obsahuje svoj názov a množinu tabuliek. Databázová tabuľka je reprezentovaná objektom vytvoreným z triedy *Table*. Tabuľka obsahuje svoj názov, množinu stĺpcov, primárny kľúč, množinu obmedzení a množinu indexov. Primárny kľúč obsahuje referenciu na objekty stĺpca a ak existuje tak predstavuje primárny kľúč danej tabuľky. Množina obmedzení a indexov je uložená spolu v rámci dátovej štruktúry kľúč-hodnota, kde kľúč pozostáva z dvojice, kde obe hodnoty predstavujú referenciu na tabuľku. Prvá referencia je vždy na zdrojovú tabuľku a teda, v ktorej je daná informácia uložená. Druhá referencia je referencia cieľovej tabuľky, na ktorú vedie vzťah alebo závislosť daného obmedzenia. Týmto prístupom je možné uložiť informáciu, či daná tabuľka má väzbu aj na inú tabuľku. Hodnotou je potom zoznam objektov obmedzení alebo indexov. Tie potom obsahujú informácie v závislosti na ich type:

- PRIMARY KEY (množina stĺpcov)
- FOREIGN KEY (referencie na tabuľky a stĺpce)
- CHECK (názov, výraz)
- INDEX (názov, množina stĺpcov)

Z toho vyplýva, že v tejto dátovej štruktúre je vzťah/kľúč, ktorý obsahuje obe referencie na zdrojovú tabuľku, symbolom pre obmedzenia, ktoré sa týkajú zdrojovej tabuľky a zároveň sú tu uložené aj indexy. V ostatných prípadoch vedú tieto referencie na odlišné tabuľky a tým je symbolizovaný vzťah medzi týmito tabuľkami. Pre lepšiu predstavu je princíp ukladania obmedzení a indexov ako príklad pre tabuľku *Užívateľ* aj graficky zobrazený na obrázku 5.1. Táto tabuľka má index a aj obmedzenie *Check*, ktoré sú uložené ako vzťah tabuľky so seba samou. Zároveň, tabuľka obsahuje dva cudzie kľúče, ktoré sú zvlášť uložené ako vzťahy s tabuľkami *Predaje* a *Nákupy*.



Obr. 5.1: Dátová štruktúra pre obmedzenia tabuľky Užívatelia

Databázový index reprezentuje objekt z triedy *Index*, ktorý obsahuje vlastný názov a množinu referencií na stĺpce. Databázový stĺpec je reprezentovaný objektom vytvoreným z triedy *Column*, ktorý obsahuje vlastný názov, databázový dátový typ ako objekt z triedy *Datatype* a množinu databázových obmedzení ako objekty z triedy *Constraint*, ktoré sa avšak týkajú iba daného stĺpca. Objekty obmedzení obsahujú informácie v závislosti na type obmedzenia. Program bude podporovať nasledovné:

- NOT NULL (stĺpec)
- UNIQUE (stĺpec)
- DEFAULT (stĺpec, predvolená hodnota)

Každá trieda, ktorej objekty sa podieľajú na tvorbe pamäťovej reprezentácie je potomkom triedy *Base*. Trieda *Base* obsahuje implementácie metód, ktoré majú dynamický charakter a na základe typu vstupných parametrov metód overia existenciu dát v štruktúre alebo získajú daný objekt. Vzhľadom na to, že pri zmene pamäťovej reprezentácie bude nutné zakaždým overovať integritu, tak bude táto funkcionalita potrebná v každej triede, ktorá tvorí pamäťovú reprezentáciu. Triedy, ktoré reprezentujú databázové obmedzenia sú zároveň aj potomkom triedy *Constraint*, ktorá predstavuje všeobecné obmedzenie. Toto pomáha rozdeliť funkcionalitu jednotlivých typov obmedzení do zvlášť tried a zároveň je stále možné vykonať typovú kontrolu na základe spoločnej rodičovskej triedy.

Inicializácia pamäťovej štruktúry predstavuje vytvorenie objektu z triedy *Database*, ktorá bude predstavovať referenciu na pamäťovú štruktúru. Jednotlivé objekty pamäťovej štruktúry musia mať v rámci svojej triedy implementované metódy, ktoré budú predstavovať vhodné rozhranie pre prácu s daným objektom. Hlavnými prvkami rozhrania musia byť správne pridanie objektu v rámci hierarchie pamäťovej reprezentácie, overenie atribútov na ich prítomnosť a typ, zmena vlastných atribútov, poskytnúť možnosť získať informácie o väzbách medzi objektami a overenie väzby medzi jednotlivými objektami.

5.4 Štruktúra pravidla kontroly

Štruktúra pravidla je dôležitou súčasťou návrhu kontroly SQL príkazu, kde pravidlo predstavuje element, ktorý sa uplatňuje na určité typy uzlov abstraktného syntaktického stromu vytvoreného z príkazu SQL, podrobnejšie v podkapitole 2.3.2. Kód, ktorý spracúva informácie z uzla,

je umiestnený v jednotlivých pravidlách. Každé pravidlo môže byť uložené do samostatného súboru. Tým môže uloženie pravidiel tvoriť štruktúry. Môžu byť umiestnené podľa potreby v rôznych adresároch, ktoré reprezentujú určitý typ kontroly. V praxi je štruktúra umiestnenia pravidiel voliteľná užívateľom, avšak pomerne veľa existujúcich riešení využíva vyššie spomenuté postupy. Pred tým, než bude rozobraná štruktúra pravidla do detailu, tak je potrebné, aby štruktúra každého pravidla bola jednotná. K tomuto poslúži vytvorenie triedy *BaseRule*. Táto trieda bude pozostávať z upravenej metatriedy *BaseRuleMetaclass*. Význam metatriedy *BaseRuleMetaclass* je podrobnejšia definícia triedy *BaseRule* a kontrola použitia jedného z preddefinovaných dekorátorov na jednotlivé metódy pravidiel. Dekorátory a ich použitie bude vysvetlené neskôr pri podrobnejšom popise fungovania pravidiel, avšak je vhodné ich existenciu spomenúť už teraz, nakoľko sa táto funkcionalita viaže na definíciu samotnej triedy pravidla. Pre vytvorenie konkrétneho pravidla je nutné vytvoriť triedu, ktorá bude potomkom triedy *BaseRule*. Z toho vyplýva, že každé pravidlo je trieda. Aby bolo pravidlo programom rozpoznané, tak je nutné, aby bola vykonaná registrácia funkciou *register*, ktorá sa nachádza v súbore s pravidlom ako samostatná funkcia. Táto funkcia má parameter pre referenciu na objekt, ktorý vykonáva kontrolu nad stromom a obsahuje zoznam registrovaných pravidiel. Následne vo funkcií daný objekt volá nad sebou metódu *register_rule*, kde parametrom je trieda registrovaného pravidla.

Ako už bolo spomenuté, pravidlá obsahujú implementáciu kontroly pre jednotlivé typy uzlov abstraktného syntaktického stromu. Tieto pravidlá môžu obsahovať implementáciu kontroly iba pre niektoré typy uzlov. Na základe toho sa môžu pravidlá špecializovať na konkrétne príkazy alebo časti kódu. Analýza stromu sa vykonáva prechodom daného stromu a volaním pravidiel na jeho uzly. Na každý uzol sú volané len tie pravidlá, ktoré majú implementáciu kontroly pre daný typ uzla.

Implementácia kontroly konkrétneho typu uzla je v pravidle obsiahnutá pomocou špecializovanej metódy. Táto metóda bude preťažená na základe parametrov typu uzla a okamihu volania. Pri prechádzaní cez strom sú jednotlivé uzly rôzneho typu. Tým pádom je možné zavolať metódu, pre konkrétny uzol na základe typu uzla. Druhý parameter, okamih volania, predstavuje v akú dobu bude metóda nad uzlom volaná. Metóda môže byť volaná, keď je daný uzol navštívený a zároveň v prípade, keď proces prechádzania cez strom opúšťa podstrom, ktorému je daný uzol koreňovým uzlom. Tieto parametre budú môcť byť nastavené kľúčovým slovom *visit* pre navštívenie uzla alebo *leave* pre opustenie spomenutého podstromu. Zároveň bude mať pravidlo prístup k pamäťovej reprezentácií cez sprostredkovanú referenciu.

Výstupom každého pravidla môžu byť hlásenia. Hlásenie reprezentuje určitú získanú informáciu a vytvorí sa v prípade, ak k tomu dospeje algoritmus kontroly uzla v pravidle. Trieda *BaseRule* obsahuje funkcionalitu a rozhranie k vytváraniu/validácií/získavaniu hlásení, ktoré sú daným pravidlom vytvorené. Ak sú dáta nového hlásenia korektné, vzniká objekt hlásenia z triedy *BaseReport*. Ten tieto dáta uchováva a je uložený do zoznamu všetkých hlásení. Hlásenie obsahuje referenciu na uzol, ktorého sa dané hlásenie týka a správu. Správa je dátová štruktúra typu kľúč-hodnota, kde identifikátor správy je kľúč a hodnota je podrobný popis nájdenej nezrovnalosti v kóde. Táto správa sa pri dokončení kontroly SQL kódu zobrazí na výstupe užívateľovi vo formátovanej podobe, podrobnejšie rozobrané v podkapitole 5.7. Správy môžu byť definované v akomkoľvek súbore, avšak v rámci existujúcich riešení je preferované, aby pravidlo a s ním spojené správy boli definované v rámci jedného súboru. Nasleduje ukážka definovania správ pre neexistujúcu tabuľku, neexistujúci stĺpec v tabuľke a zlý dátový typ stĺpca:

```
messages = {
    "table-not-exists": {
        "message": "Table {table_name} not exists"
    },
    "column-not-exists": {
        "message": "Column {column_name} not exists"
    },
    "column-wrong-datatype": {
        "message": "Column {column_name} has wrong datatype"
    },
}
```

Správy sú s pravidlom previazané využitím jedného z preddefinovaných dekorátorov. Tieto dekorátory sa viažu na metódy pravidla, v ktorých sú hlásenia vytvárané. Pre vytvorenie hlásenia je nutné zavolať špecializovanú metódu. Tá má povinný parameter identifikátor(kľúč) správy a voliteľné parametre pre nahradenie textu v správe, ktorý je ohraničený kučeravými zátvorkami. Tým bude vytvorené hlásenie a dekorátor zaručí pridanie obsahu správy na základe zadaného identifikátora. Program podporuje dva typy dekorátorov. Tie sa od seba odlišujú spôsobom ako registrujú správy, ktoré môžu byť použité. Dekorátor *include_reports* má parameter *reports*, kde očakáva referenciu na štruktúru so správami. Naopak, dekorátor *include_class_reports* nemá žiadny parameter ale očakáva, že trieda pravidla, v ktorej sa dekorovaná metóda nachádza, bude obsahovať atribút *reports* s referenciou na štruktúru so správami. Cieľom je dať užívateľovi na výber: buď registrovať všetky správy v rámci triedy pravidla alebo rozdeliť si spravy do viac časti(napríklad podľa významu) a priraďovať ich jednotlivým metódam.

Nasleduje ukážka definície pravidla *Rule01*, ktoré definuje kontrolu pre uzly tabuľky a identifikátoru. Pravidlo využíva oboch dekorátorov.

```
class Rule01(BaseRule):
    messages = messages

    @include_reports(reports=reports)
    def table_visit(self):
        # Implementation

            # Report creation
            self.create_report("table-not-exists",
                               self.node,
                               table_name=self.node.name)

    @include_class_reports()
    def table_leave(self):
        # Implementation

    @include_class_reports()
    def identifier_visit(self):
        # Implementation

#Rule registration
def register(checker) -> None:
    checker.register_rule(Rule01)
```

Ďalšou vlastnosťou pravidiel je možnosť ovplyvniť určitými atribútmi ich správanie. Algoritmus, ktorý volá jednotlivé pravidlá, zároveň vie rozpoznať ak má pravidlo zadané atribúty *restrict* alebo *temporary*/*persistent*.

V rôznych SQL príkazoch sa môžu nachádzať zhodne pomenované uzly ako napríklad uzol pre identifikátor alebo definíciu stĺpca. Atribútom *restrict* je možné ovplyvniť, na ktoré SQL príkazy bude pravidlo vyhradené a uplatnené. Cieľom je pridať možnosť cieliť pravidlo na konkrétny typ SQL príkazu. Ak užívateľ atribút nezadá, potom pravidlo bude automaticky akceptovať všetky SQL príkazy. Očakávanou hodnotou je množina reťazcov, na ktoré sa má pravidlo uplatniť, kde reťazec je kľúčovým slovom príkazu. Množina *restrict* bude v ukážkových scenároch vyzerať nasledovne:

- Pravidlo všetky príkazy *CREATE* a *DROP* => {*CREATE*, *DROP*}.
- Pravidlo všetky príkazy *CREATE* a iba *DROP TABLE* => {*CREATE*, *DROP TABLE*}.
- Pravidlo iba pre príkaz *CREATE TABLE* => {*CREATE TABLE*}.
- Pravidlo iba pre príkazy *SELECT* a *CREATE INDEX* => {*SELECT*, *CREATE INDEX*}.

Pravidlá bez dodatočnej konfigurácie majú vlastnosť, že ich objekty si uchovávajú stav počas celého prechodu stromu. Tým sa myslí prechod stromu v rámci jedného príkazu SQL. Aby bolo možné uchovávať stav objektu naprieč kontrolou všetkých príkazov SQL, to jest prechodom všetkých stromov, tak vznikol atribút *persistent*. Ak tvorca pravidla uplatní tento atribút, tak bude toto pravidlo reprezentovať stále rovnaký objekt pri kontrole jednotlivých SQL príkazov. To môže byť výhodné, ak pravidlo pracuje s informáciami v kontexte naprieč SQL príkazmi. Naopak atribút *temporary* dovolí autorovi pravidla aby sa stav objektu neukladal. Tým dôjde k tomu, že pri kontrole uzla sa z pravidla vytvorí objekt a po kontrole daného uzla sa odstráni. Tým pádom každý uzol kontroluje nový objekt, hoci sú kontrolované uzly súčasťou jedného stromu. Nasleduje ukážka ako bude táto konfigurácia pravidiel definovaná v samotnom pravidle:

```
1 class Rule01(BaseRule):
2     messages = messages
3     persistent = True
4     restrict = {CREATE, DROP}
5 
6     # Methods implementation
```

Poslednou vlastnosťou je možnosť zavolať pravidlo na začiatku alebo konci kontroly. Tieto dve udalosti nastávajú mimo kontrolu stromu SQL príkazu. Z toho vyplýva, že udalosť "začiatok kontroly"nastáva pred kontrolou SQL príkazu a udalosť "koniec kontroly"nastáva po skončení kontroly SQL príkazu. Túto vlastnosť podporujú len pravidlá s parametrom *persistent* alebo normálne pravidlá, pričom ich význam sa odlišuje. Pri pravidle s parametrom *persistent* znamená "začiatok kontrolyüdalosť, kedy program začína celkovú kontrolu. Rovnako je to pri udalosti "koniec kontroly", ktorá nastane po ukončení kontroly posledného SQL príkazu. Naopak normálne pravidlá, udalosť "začiatok kontroly"rozumejú ako začiatok kontroly konkrétneho SQL príkazu a "koniec kontroly"pod jeho ukončením. Inými slovami sa dá povedať, že udalosti "začiatok kontrolyä "koniec kontroly"sú charakteristické v závislosti na život objektu pravidla, kde objekt normálnych pravidiel je vytváraný pri začiatku kontroly SQL príkazu a ukončený pri jeho konci, zatiaľ čo u pravidla s parametrom *persistent* je život objektu zachovaný naprieč všetkými SQL príkazmi. Tieto vlastnosti je možné použiť vytvorením metódy, ktorá je špecifická pre konkrétne pravidlo.

- Metóda *start_lint* - udalosť "začiatok kontroly"pre pravidlo s parametrom *persistent*
- Metóda *end_lint* - udalosť "koniec kontroly"pre pravidlo s parametrom *persistent*
- Metóda *start_statement_lint* - udalosť "začiatok kontroly"pre normálne pravidlo
- Metóda *end_statement_lint* - udalosť "koniec kontroly"pre normálne pravidlo

5.5 Spracovanie a analýza príkazu SQL

Predchádzajúce podkapitoly podrobne popísali spracovanie parametrov programu, pamäťovú štruktúru a štruktúru pravidiel. V tomto momente je možné prejsť k spracovaniu a analýze samotného SQL príkazu, ktorá bude nadväzovať a spájať funkcionalitu už spomenutých podkapitol do jedného celku. Pre obsluhu týchto funkcionalít programu poslúži trieda *Linter*, ktorá bude spravovať hlavný beh programu.

Predpokladom pre analýzu SQL príkazu je vykonanie syntaktickej analýzy a tým overenie korektnosti syntaxe daného príkazu. Následne je potom možné získať z príkazu abstraktný syntaktický strom. Obe tieto úlohy budú splnené pomocou využitia knižnice, ktorá daný SQL príkaz spracuje. Očakávaným výstupom z knižnice je referencia na koreňový uzol stromu alebo vyvolanie výnimky, ak sa pri syntaktickej kontrole zistilo, že príkaz nie je validný. Vyvolanie výnimky program zachytí ako neočakávaný stav a varuje užívateľa varovnou správou na výstup.

Pri kontrole príkazu SQL bude program prechádzať stromom do hĺbky a nad každým uzlom zavolá pravidla, ktoré vyhovujú podmienkam na základe typu uzla a atribútov volaného pravidla. Aby bolo možné uplatniť kontrolu uzla pomocou pravidla, tak je nutné, aby sa pravidlo dostalo ku dátam daného uzla. K tomuto využijeme návrhový vzor Návštevník, podrobnejšie v podkapitole 2.3.3. V rámci knižnice je typ uzla stromu reprezentovaný ako trieda, ktorú je nutné upraviť tak, aby mohla prijať návštevníkov.

Každá knižnica ako externý program alebo súbor funkcionalít ma vlastný vývojový cyklus. Typicky obsahuje vlastný návrh riešenia, ktoré je dostupne cez dostupné rozhranie. Cieľom j,e aby úprava tried nebola inváza voči knižnici a nepozmenila tak zdrojový kód knižnice. Tým bude dosiahnuté, že knižnica môže byť stále aktualizovaná bez toho, aby musel byť udržovaný dodatočne pridaný kód. Pre tento účel využijeme návrhový vzor Adaptér, popísaný v podkapitole 2.3.4.

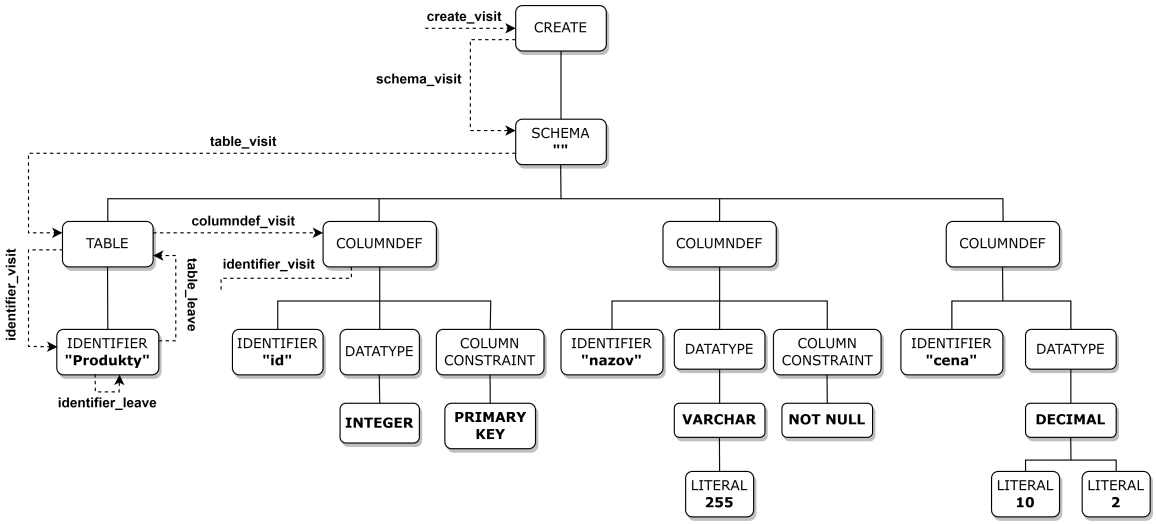
Pre vlastnú funkcionalitu vytvoríme triedu *AcceptVisitor*, ktorá obsahuje už spomínanú metódu *Accept*. Túto metódu je potrebné začleniť do uzlov stromu v rámci implementácie návrhového vzoru Návštevník. Pomocou adaptácie cez triedu, využijeme vlastnosť viacnásobnej dedičnosti pre zlúčenie rodičovských rozhraní do adaptačnej triedy, ktorá bude rozhrania dediť. Aby bolo takto možné dynamický prispôsobiť rozhranie knižnice, bude pre tento účel vytvorená funkcia *class_factory* z triedy *BaseClass*, ktorá predstavuje prázdnu triedu adaptéra. Zároveň bude vytvorená trieda *BaseCast*, ktorá bude obsahovať metódu pre zmenu typu triedy na inú triedu. Proces adaptácie triedy uzla bude nasledovný: Vo funkcií *class_factory* bude dynamicky vytvorená nová trieda adaptéru, ktorá bude dediť všetky vyššie spomenuté triedy (triedu uzla z knižnice, *AcceptVisitor*, *BaseCast*). Následne bude z triedy adaptéru vytvorený objekt, ktorému bude zmenený typ na pôvodnú triedu uzla, aby bolo vyhovené typovej kontrole. Tento proces adaptácie vykonáme nad každým uzlom stromu. Výsledkom bude pôvodný strom, rozšírený o metódu pre prijatie návštevníka.

Nasleduje už samotná analýza príkazu SQL 5.1, ktorá je znázornená na obrázku 5.2 aplikovaním pravidiel na abstraktný syntaktický strom daného SQL príkazu. Typ príkazu je rozpoznaný

podla koreňového uzla stromu, z ktorého je vytvorená akceptačná množina. Táto množina následne slúži pre výber vhodných pravidiel, nad ktorými bude príkaz kontrolovaný. Kontrola bude prebiehať ako už bolo spomenuté - prechádzaním jednotlivých uzlov, na ktoré budú volané vybrané pravidlá, kde pravidlo môže byť na uzol zavolané pri jeho navštívení alebo ak priechod stromom opúšťa podstrom, ktorého je daný uzol koreňovým uzlom. Počas analýzy stromu sú zhromažďované hlásenia z pravidiel.

```
1  CREATE TABLE Produkty (
2          id INTEGER PRIMARY KEY,
3          nazov VARCHAR(255) NOT NULL,
4          cena DECIMAL(10, 2),
5  );
```

Výpis 5.1: SQL kód, z ktorého je vytvorený abstraktný syntaktický strom na obrázku 5.2



Obr. 5.2: Analýza abstraktného syntaktického stromu.

5.6 Vytvorenie hlásení

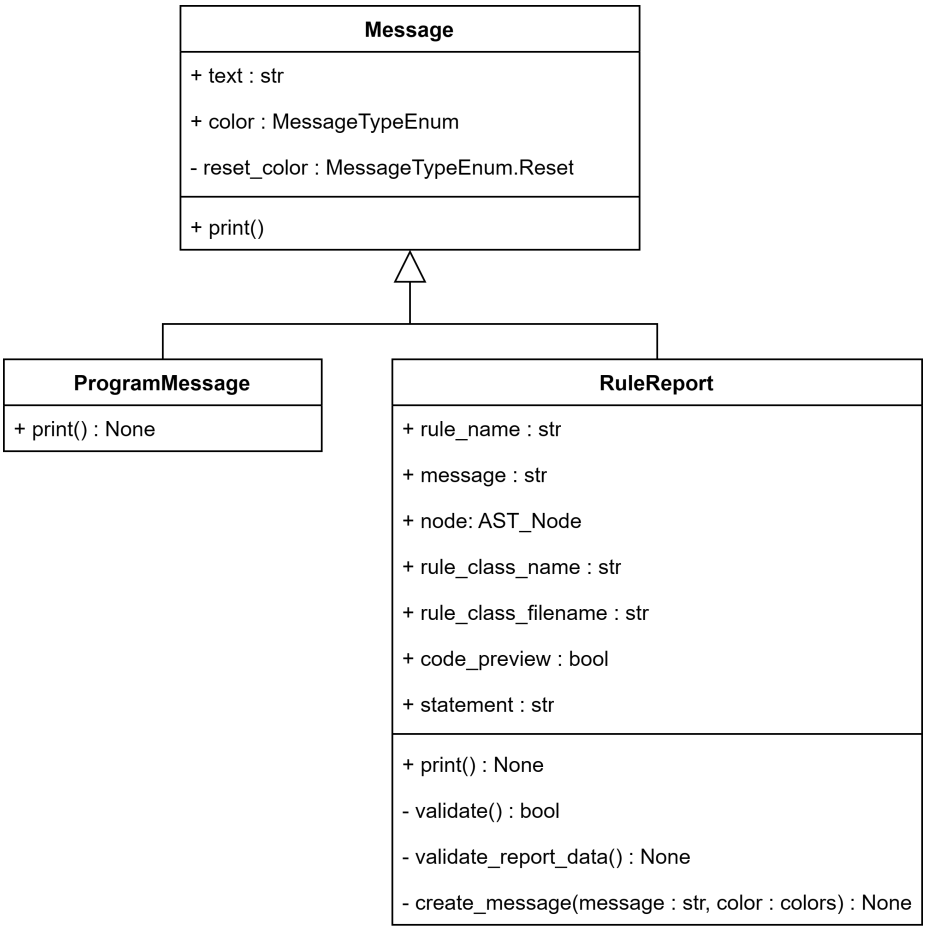
Ešte pred návrhom výstupu programu je nutné navrhnúť spôsob ako budú hlásenia pre užívateľa spracovávané a ukladané. Program musí z každého pravidla získavať vytvorené hlásenia a uložiť ich v kontexte s SQL príkazom, ktorého sa týkajú. Zároveň musí vedieť spracovať neočakávane a nechcené stavy programu, ktoré môžu nastať pri zlom vstupe programu alebo neočakávanej udalosti. Cieľom je vytvoriť štruktúru, ktorá bude mať za úlohu obsluhovať jednotlivé úlohy popísané vyššie. Za výstup programu sú považované ako hlásenia získanie z pravidiel tak správy týkajúce sa behu programu.

Program bude dekomponovať problém výstupu programu do tried, ktoré budú rozdelené podľa typu výstupu. Aby bolo rozhranie týchto tried zjednotené, tak sú vytvorené triedy *Reporter* a *Message*. Obe tieto triedy obsahujú základné rozhranie pre tvorbu potomkov. Trieda *Reporter* obsahuje rozhranie pre triedy, kde budú všetky hlásenia a správy ukladané a spravované. Zároveň bude obsahovať aj statickú premennú pre miesto výstupu, ktorá bude inicializovaná pri spracovaní vstupu programu. Trieda *Message* reprezentuje jedno konkrétne hlásenie so správou a rozhranie pre zobrazenie správy.

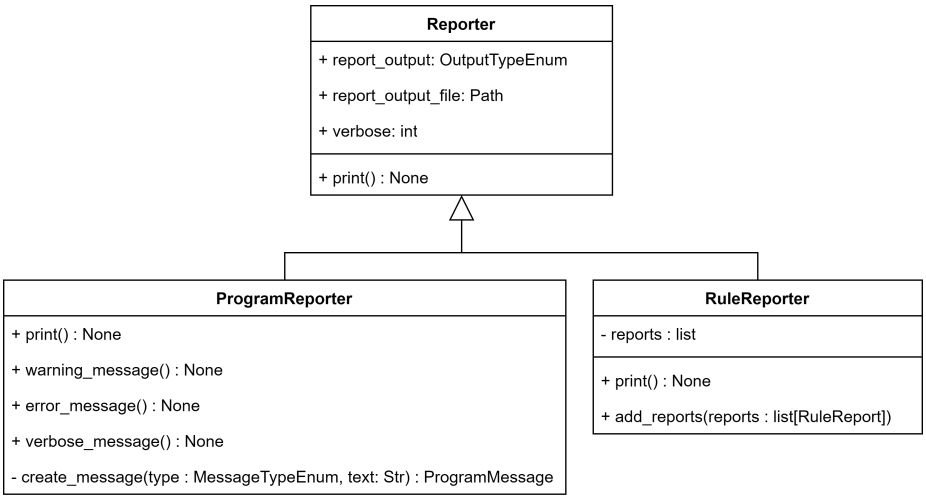
K obsluhovaniu správ z behu programu je vytvorená trieda *ProgramReporter*, z ktorej je následne vytvorený jediný objekt. Obsahuje funkcionalitu pre tri typy správ: chybové, varovné a informačné. Všetky typy správy sú vytvárané ako objekt pomocou triedy *ProgramMessage*, ktorý uchováva správu a obsluhuje formátovanie výpisu správy. Pri každom zobrazení správy sa najskôr zobrazí kľúčové slovo daného typu správy (*Error*, *Warning*, *Info*) a na nasledujúcom riadku sa nachádza obsah správy. V prípade, že miesto výstupu je na štandardný výstup, tak sú správy odlíšené aj farebne (červená, žltá, biela).

- Chybové správy (*Error*, červená farba), obsahujú text chyby a ukončujú program kódom, ktorý sa vzťahuje ku danému typu chyby. Tento typ správy je použitý pokiaľ sa program nevie zotaviť z výskytu fatálnej chyby.
- Varovné správy (*Warning*, žltá farba), obsahujú text správy, ktorý ma upozorniť užívateľa na nekorektné alebo nekonzistentné chovanie v programe. Tento stav je možné dosiahnuť napríklad neočakávaným SQL príkazom na vstupe alebo neočakávanou kombináciou SQL príkazov na vstupe. Program po upozornení správach pokračuje v behu.
- Informačné správy (*Info*, biela farba), obsahujú text správy, ktorý slúži pre informovanie užívateľa o vykonávaných akciách počas behu programu.

Pre obsluhu hlásení získaných z pravidiel je vytvorená trieda *RuleReporter*, z ktorej je vytvorený jediný objekt. Obsahuje jednoduchú funkcionalitu vo forme pridania nového hlásenia a výpisu všetkých hlásení. Samotné hlásenie predstavuje vytvorenie objektu z triedy *RuleReport*. Nakoľko sú hlásenia užívateľsky definované, tak v rámci inicializačnej metódy nového hlásenia je volaná metóda pre validáciu vstupných dát, ktorá kontroluje prítomnosť a typ všetkých potrebných informácií. Rovnako každé hlásenie obsahuje metódu pre vlastný formátovaný výpis, ktorý je podrobne popísaný v podkapitole 5.7, ktorá popisuje výstup programu. Hierarchia týchto tried je znázornená na nasledujúcej strane na diagramoch tried 5.3 a 5.4.



Obr. 5.3: Hierarchia tried pre tvorenie správ.



Obr. 5.4: Hierarchia tried pre spravovanie správ.

5.7 Výstup programu

Potreba intuitívneho výstupu je rovnako dôležitá ako aj ostatné časti programu. Užívateľ získava z programového výstupu informácie o výsledkoch, ktoré program počas svojho behu vytvoril. Tieto výsledky musia byť formátované tak, aby užívateľ intuitívne a na prvý pohľad výsledky pochopil. Aby program svojím výstupom zapadol medzi existujúce riešenia, tak bol výstupový formát inšpirovaný z časti kompilátorom jazyka C (GCC) a Haskell (GHCi). Cieľom pre inšpiráciu a odvodenie formátu je aby skúsenejší užívateľ, ktorý sa s danými výstupmi stretol u iných programov, mohol bez akýchkoľvek problémov chápať výstupný formát aj tohoto programu. Výstupný formát má dve úrovne: základný a rozšírený formát.

5.7.1 Základný formát

Základný formát má za cieľ minimalizovať výstup do najkompaktnejšej formy a informovať užívateľa len o najdôležitejších správach. Výstup obsahuje spracované hlásenia, ktoré boli vytvorené pri kontrole SQL príkazu. Obsahujú informácie o pozícií hlásenia (ak je dostupná) prostredníctvo riadku (`L`) a stĺpca (`C`) vo vstupom texte. Skratky riadky a stĺpca sú odvodené od ich anglického prekladu. Na rovnakom riadku nasleduje šípka, za ktorou sa nachádza identifikátor získanej správy. Na nasledujúcom riadku za nachádza plne znenie správy, z ktorého užívateľ má pochopiť zmysel daného hlásenia. Ak je dostupná pozícia, potom sa pod správou nachádza ukážka s kódom, kde užívateľ vidí na presnú časť kódu, v ktorej bolo hlásenie vytvorené. Nasleduje ukážka základného formátu:

```
1  L: 15 C: 23 --> [column-not-exists]
2  Report: Column CustomerID not exists
3
4     1 | SELECT Country, COUNT(CustomerID)
5                               ^~~~~~~~~~
6
7  L: 16 C: 6 --> [table-not-exists]
8  Report: Table Table1 not exists
9
10    2 | FROM Table1 JOIN Table2 ON Table1.id1=Table2.id2
11            ^~~~~~
```

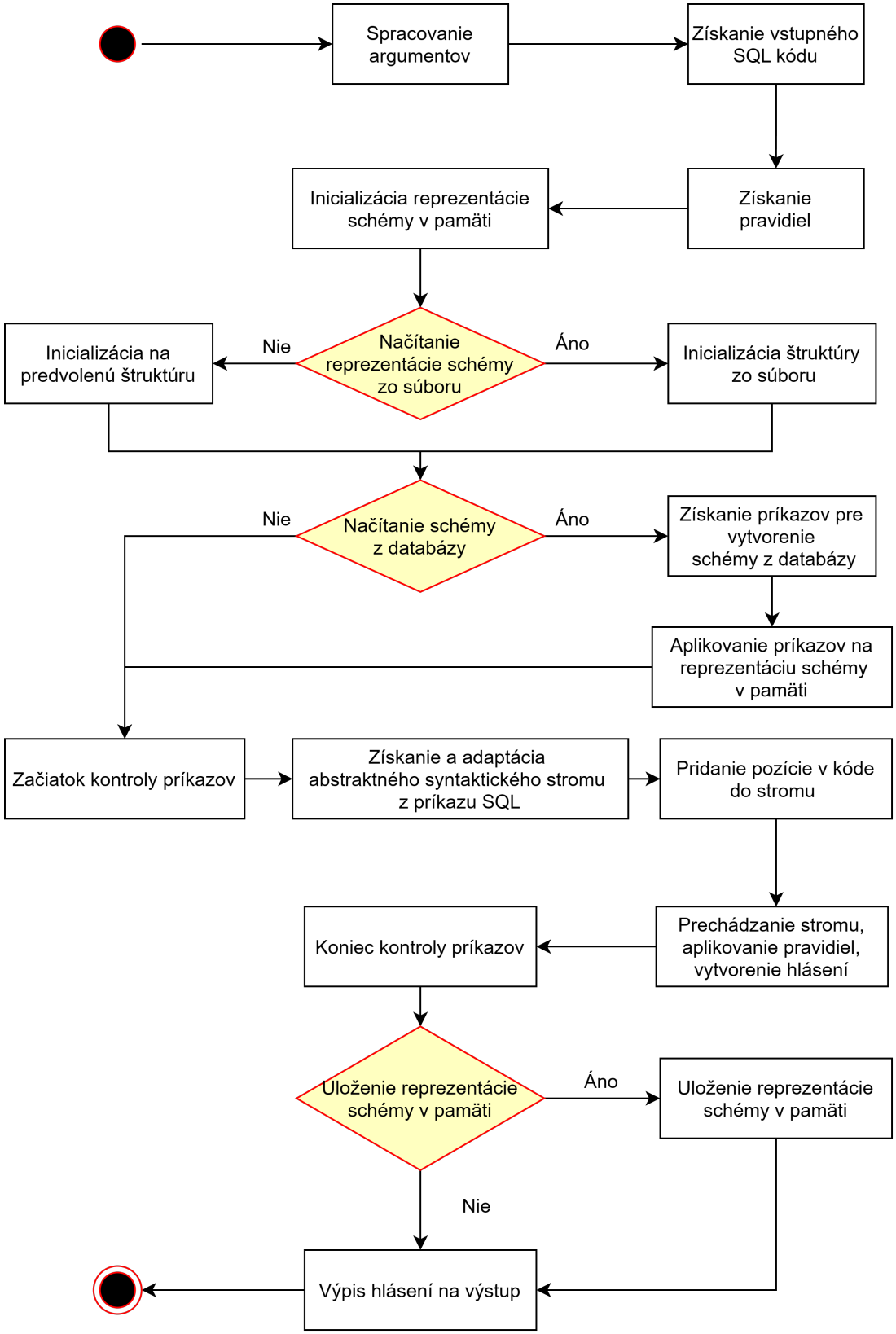
5.7.2 Rozšírený formát

Rozšírený formát rozširuje funkcionalitu základného formátu. Nakoľko sa každá správa vzťahuje k určitému SQL príkazu, tak v rozšírenom formáte pribúda výpis daného príkazu, ku ktorému sa hlásenia vzťahujú. Výsledkom je výpis, ktorý najskôr vypíše kontrolovaný SQL príkaz a k nemu sú následne vypísané hlásenia, ktoré sa ho týkajú. SQL príkaz je doplnený o riadky kódu vo forme komentárom za kódom. Cieľom je, aby užívateľ sa vedel ľahšie orientovať medzi hlásením a príkazom SQL prostredníctvom označenia riadku. Program zaisťuje, aby bol výpis SQL čo najkompaktnejší nezávisle od toho ako je napísaný vo vstupnom súbore. Program zachováva a neupravuje štýl zápisu kódu ale odstraňuje prázdne riadne riadky a komentáre z SQL príkazu. To je zároveň aj dôvod prečo sú čísla jednotlivých riadkov zapísané vo forme komentárov, pretože kód sa nemusí nachádzať hneď bezprostredne za sebou ale pre výstup programu je dôležité, aby sa odkazoval na správne čísla riadkov. Nasleduje ukážka rozšíreného formátu:

```
//============================\\
|| LINTING RESULT OF STATEMENT ||
\\============================//
SELECT Country, COUNT(CustomerID) -- 15
FROM Table1 JOIN Table2 ON Table1.id1=Table2.id2 -- 16
GROUP BY Country -- 17
HAVING COUNT(CustomerID) > 5 -- 18
ORDER BY COUNT(CustomerID) DESC; -- 19
\\============================//
L: 15 C: 23 --> [column-not-exists]
Report: Column CustomerID not exists

   1 | SELECT Country, COUNT(CustomerID)
                            ^~~~~~~~~~

L: 16 C: 6 --> [table-not-exists]
Report: Table Table1 not exists

   2 | FROM Table1 JOIN Table2 ON Table1.id1=Table2.id2
             ^~~~~~
```

5.8 Priebeh programu

Kapitola návrhu popisuje jednotlivé časti programu pre jeho implementáciu. Tieto časti sú popísané v samostatných celkoch, ktoré sa podrobnejšie venujú návrhu a fungovaniu. Koniec kapitoly návrhu je preto venovaný priebehu programu, ktorý je vyjadrený na diagrame aktivít 5.5 na nasledujúcej strane. Ten popisuje ako vyššie spomenuté časti návrhu na seba nadväzujú a vytvárajú priebeh programu.



Obr. 5.5: Priebeh behu programu.

Kapitola 6

Implementácia

Po vytvorení návrhu programu nasleduje jeho realizácia. Táto kapitola popisuje použité technológie a implementáciu podla vytvoreného návrhu. Cieľom je zoznámiť čitateľa aj s problémami, ktoré sa pri implementácií vyskytli a riešenie kľúčových problémov.

6.1 Použité technológie

Pri voľbe programovacieho jazyka boli zohľadnené mnohé požiadavky, medzi inými aj predchádzajúce skúsenosti a náročnosť implementácie v danom jazyku. Prioritou bolo zvolenie všeobecne známeho a rozšíreného jazyka, ktorý najlepšie vyhovie nižšie spomenutým požiadavkám.

Návrh programu bol vytvorený na základe princípoch objektovo orientovaného programovania, ktorý požadoval, aby užívateľ mohol jednoduchým spôsobom pridať vlastné pravidlá do programu. Nakoľko sú pravidlá tvorené ako programoví kód, ktorý je tiež súčasťou programu, tak boli považované interpretované jazyky za výhodnejšie. Dôvodom je, že program nemusí byť pri pridaní/odobraní pravidla opäť skompilovaný a tým je užívateľsky prívetivejší.

Mimo iné bola požiadavka pre nájdenie vhodnej knižnice pre vytvorenie abstraktného syntaktického stromu pre jazyk SQL, ktorý overí aj syntaktickú správnosť kódu. Pri výbere boli analyzované hlavne tieto vlastnosti: rozsah pokrytia jazyka SQL, podpora pre získanie pozície v kóde, podpora štandardu SQL, jednoduchý prechod stromom do hĺbky, návrh a spôsob implementácie knižnice a zostavenia stromu, náročnosť úpravy knižnice v prípade nevyhnutnej potreby, dokumentácia, status vývoja a aktivita autorov knižnice. Výber vhodnej knižnice bol ovplyvnený poznatkami zo podkapitoly 3.5. Žiadna zo spomenutých knižníc nespĺňala všetky potrebné požiadavky a preto bolo nutné pristúpiť pri implementácií ku kompromisom. Tie sú podrobnejšie popísané nižšie v podkapitole 6.3, ktorá popisuje vzniknuté problémy týkajúce sa abstraktného syntaktického stromu a ich riešenia.

Výsledkom zváženia predchádzajúcich poznatkov bol pre implementáciu zvolený programovací jazyk Python3 vo verzií 3.11, ktorá bola v čase počiatku implementácie najaktuálnejšou. Python3 je interpretovaný programovací jazyk, ktorý dovoľuje implementáciu programu pomocou objektovo orientovaného paradigmu. Ide o aktívne vyvíjaný jazyk, ktorý sa vyznačuje vysokou abstrakciou. Ďalšou výhodou jazyka je jeho vysoká popularita, veľké množstvo dostupných knižníc a podpora v mnohých vývojových prostrediach. Pri tvorbe programu využitý Jetbrains Pycharm.

Pre tvorbu abstraktného syntaktického stromu bola zvolená knižnica SQLGlot, ktorá je naprogramovaná v jazyku Python3. Ako už bolo spomenuté v podkapitole 3.3, knižnica obsahuje

mnoho funkcionalít a nešpecializuje sa iba na tieto úlohy. Pokrytie jazyka SQL bolo vyhodnotené ako dostatočné s istými výhradami, ktoré sú popísané v podkapitole 6.3.3. Pri výbere boľ uprednostnený aktívny vývoj a širšie pokrytie jazyka SQL a jeho dialektov pred prítomnosťou informácie o pozícií v kóde v abstraktnom syntaktickom strome. K tomuto rozhodnutiu pomohlo aj nájdenie riešenia pre minimalizáciu problému s nedostupnou informáciou o pozícií v kóde v knižnici SQLGlot.

6.2 Vstup

Program spracováva vstupný SQL kód, ktorý spĺňa požiadavky návrhu v podkapitole 5.1. Pri spracovaní kódu sú odstránené všetky typy komentárov. Avšak, odstránením komentárov nedochádza k odstráneniu celých riadkov. Tie sú zachované aj v prípade, že na danom riadku je iba komentár, kvôli ďalšiemu spracovaniu. Aby bolo možné lepšie pracovať s jednotlivými príkazmi SQL a odkazovať sa na ich umiestnenie vo vstupnom kóde, tak program si poznačí každé číslo riadku kde sa nachádzajú časti SQL kód. K tomu dochádza po odstránení komentárov, kedy program následne očísluje všetky riadky vo forme nových komentárov. Získaný vstupný SQL kód obohatený o čísla riadkov v komentároch je následne skrátený o riadky, na ktorých sa nenachádza SQL kód, to jest nachádza sa tam iba komentár s číslom riadku a prípadné medzery pred komentárom. Takto upravený SQL kód rozdelíme podľa znaku ukončenia SQL príkazu (`;`), ktorý je očakávaný za každým SQL príkazom. Výstupom je tak zoznam SQL príkazov obohatených o pôvodné čísla riadkov v komentároch, na ktorých sa nachádzali. Tie následne pomôžu pri ďalšom spracovaní SQL príkazu. Môžu pomôcť aj užívateľovi, pri rozšírenom výpise hlásení, kde budú riadky zobrazené pri výpise SQL príkazu.

6.3 Abstraktný syntaktický strom

Táto podkapitola popisuje problémy, ktoré sa vyskytli pri integrácií abstraktného syntaktického stromu z knižnice SQLGlot do programu. Tie vznikli tým, že strom neobsahuje všetku funkcionalitu, ktorú program potrebuje pre svoje fungovanie. Tá tak bola docielená prispôsobením stromu a čiastočne aj knižnice.

6.3.1 Pozícia kódu

Knižnica po spracovaní SQL príkazu vytvorí zoznam tokenov, kde jednotlivé tokeny obsahujú pozíciu v kóde reprezentovanú uvedením riadku a stĺpca. Tieto tokeny sú následne využívané pri tvorbe abstraktného syntaktického stromu, avšak informácia o pozícií kódu je knižnicou zahodená. Z toho vyplýva, že strom neobsahuje vo svojich uzloch pozíciu daného uzla v kóde. Tým pádom program pri vyhodnocovaní príkazu, ku ktorému je daný strom využitý, nie je schopný vytvoriť hlásenie, ktoré by odkazovalo na určitý úsek vstupného kódu. Cieľom preto bolo nájsť a implementovať riešenie, ktoré by do jednotlivých uzlov stromu pridalo pozíciu v kóde. Výsledkom je navrhnutý a implementovaný nasledujúci algoritmus.

Algoritmus sa nachádza v programe na mieste, ktoré bezprostredne nasleduje za spracovaním príkazu a vytvorením stromu knižnicou. Ten potrebuje na svojom vstupe zoznam tokenov vytvorených knižnicou pre spracovávaný príkaz SQL. Program získava tento zoznam tokenov priamo z knižnice, pomocou importovania a použitia konkrétneho modulu, ktorý má danú úlohu na starosti. Algoritmus využíva vlastnosti vytvárania abstraktného syntaktického stromu. Väčšina uzlov je vytváraných na základe tokenu/tokenov z daného zoznamu tokenov.

Tým sú určité dáta prenesené z tokenu do uzlu v rovnakej podobe. Pri vytváraní stromu sa cez zoznam tokenov prechádza postupne v rade od prvého po posledný, kde postup vytvárania stromu sa rovná jeho priechodu do hĺbky. Algoritmus preto prechádza rovnakým spôsobom cez strom a zoznam tokenov. Následne pre každý uzol stromu, algoritmus hľadá token, ktorý má spoločné dáta s uzlom. Zároveň sú tokeny rozlišované na preskúmané a nepreskúmané. Pri inicializácií algoritmu sú všetky tokeny považované za nepreskúmané. Za preskúmané sa považujú len tie, ktoré sa program neúspešne pokúsil spojiť s uzlom, až na výnimku, ak program nenašiel žiaden token, ktorý by vyhovoval. Z toho vyplýva, aby boli kandidáti na preskúmané tokeny naozaj označené za preskúmané, musí sa programu podariť nájsť spojenie token-uzol v nepreskúmaných tokenoch. Ďalšou výnimkou je, že ak token je spojený s uzlom, tak hoci bol daný token navštívený, nebude označený za preskúmaný. Testovanie ukázalo, že niektoré podstromy môžu byť mierne odlišne usporiadané ako tokeny. Preto ak nie je nájdená zhoda medzi nepreskúmanými tokenmi, tak nasleduje opätovný prechod cez preskúmané tokeny, od posledné navštíveného ku počiatočnému tokenu zo zoznamu. Podrobnejší popis algoritmu pomocou pseudokódu je možné nájsť v prílohe A.

6.3.2 Pozícia kódu vo výstupe

Knižnica SQGlot spracováva jednotlivé príkazy SQL samostatne po jednom. Tým je ovplyvnená pozícia kódu v strome, pretože knižnica predpokladá, že každý príkaz začína na začiatku súbora na jeho prvom riadku. Aby pozície kódu uložené v strome reflektovali reálne umiestnenie príkazu vo vstupnom súbore, tak sú po vytvorení stromu prepočítané. Program si udržuje informáciu o tom, z ktorého riadku získal daný príkaz a následne vykoná prechod celým stromom a k pozíciám umiestnených v uzloch pripočíta daný počet riadkov. Výsledkom čoho môže byť informácia o pozícií v kóde použitá vo výstupných hláseniach tak, aby reflektovala vstupný SQL kód.

6.3.3 Rozsah pokrytia SQL

Napriek tomu, že sa autori knižnice SQLGlot snažia pokrývať mnoho dialektov, tak tá momentálne nepokrýva celý jazyk SQL. Príkladom môže byť chýbajúce komplexnejšie pokrytie príkazov *ALTER*. V rámci práce prebehol pokus o dokončenie implementácie týchto príkazov, nakoľko existujúca implementácia sa týka iba databázovej tabuľky. Pokus sa ukázal ako realizovateľný. Knižnica tak predstavuje dobrú platformu pre doplnenie podpory pre ďalšie SQL príkazy. Nakoniec bolo ale rozhodnuté, že takto samostatne pozmenená knižnica nebude zaradená do finálneho riešenia práce, kvôli potencionálnym problémom s budúcou aktualizáciou knižnice. Výsledkom tak môže byť zaradenie zmien do repozitára knižnice v budúcnosti.

V prípade príkazov *ALTER*, bolo zároveň zistené, že knižnica pri tomto type príkazov produkuje mierne nekonzistentný strom oproti zvyšku príkazov. Typicky, štruktúra stromu z knižnice obsahuje koreňový uzol reprezentujúci akciu. Tento uzol potom obsahuje atribúty, ktoré špecifikujú objekt databázy, na ktorý sa akcia vzťahuje. Príkladom môže byť vytvorenie databázovej tabuľky, kde koreňový uzol by bol *CREATE* s atribútom *TABLE*. Ak by bol vytvorený databázový index, koreňový uzol by ostal rovnaký, *CREATE*, ale zmenil by sa jeho atribút na *INDEX*. V prípade príkazu *ALTER* je koreňový uzol nazvaný ako *AlterTable*, čím priamo špecifikuje akciu aj objekt databázy priamo v názve uzlu. Autori knižnice boli na túto problematiku upozornený prostredníctvom komunikačného kanála Slack, nakoľko to sťažuje spracovanie stromu v programe.

6.4 Pravidlá

Návrh pravidiel programu v podkapitole 5.4 bol založený na základe preťaženia metód. To malo zaručiť, aby pravidlá mohli obsahovať implementáciu kontroly pre viacero typov uzlov. Nakoľko súčasná implementácia jazyka Python3 nedovoľovala implementáciu preťaženie metód vhodným spôsobom, tak bol použitý iný prístup, ktorý ním bol inšpirovaný.

Funkcionalita bola dosiahnutá získaním a volaním metódy na základe špecifického názvu metódy. Daný názov sa skladá z dvoch celkov, ktoré sú spojene podčiarkovníkom. Prvý celok, prefix názvu metódy, tvorí názov typu uzla a druhý celok, sufix názvu metódy, je tvorený kľúčovým slovom okamihu volania metódy. Zároveň táto metóda nemá žiadne parametre a referencia na kontrolovaný uzol bude priradená do objektu pravidla pred zavolaním metódy pre kontrolu.

6.4.1 Správy

Správy musia byť autorom pravidla implementované ako dátový typ slovník v jazyku Python3. Ich hierarchia je zhodná s návrhom v podkapitole 5.4. Program zároveň nekontroluje ak sú medzi jednotlivými pravidlami duplicitne definované správy alebo ich identifikátory. Niektoré existujúce riešenia tento prvok kontroly obsahujú, avšak cieľom programu bolo ponechať užívateľovi slobodu v tvorení správ, nakoľko oba spôsoby majú svoje kladne a záporne stránky. V prípade, že užívateľ chce, aby boli správy unikátne, môže si všetky správy vytvoriť na centrálnom mieste, na ktoré sa bude v jednotlivých pravidlách následne odkazovať. Kontrolu duplicity môže byť potom zabezpečená statickým analyzátorom pre Python3 - PyLint[1].

6.5 Výstup

Pri výstupe programu bola zohľadnená užívateľská prívetivosť. Výstupné hlásenia sú usporiadané podla poradia kontrolovaných SQL príkazov a čísla riadku, na ktorý sa hlásenie odkazuje. V rámci parametra *{verbose}* je možné upravovať podrobnosť výstupu. Ak je výstup zobrazený na štandardnom výstupe, tak sú dôležité informácie v hláseniach aj farebne odlíšené, pre prehľadnejšie zobrazenie.

[1] https://github.com/pylint-dev/pylint

Kapitola 7

Testovanie

Pre overenie fungovania funkcionalít programu bolo vykonávané testovanie ako súčasť vývoja programu. Program bol vyvíjaný v iteráciách, kde na konci každej iterácie bolo vykonané testovanie. Testovanie sa zaoberalo hľadaním programových chýb, overením správnosti návrhu a spôsobu implementácie funkcionalít. Testovanie tak prispelo aj k miernej úprave funkcionalít, ktoré boli následne v ďalšej iterácií vývoja začlenené do návrhu. To zahrňovalo aj priebežnú údržbu kódu s cieľom zlepšiť štruktúru existujúceho kódu pri zachovaní jeho pôvodnej funkcionality. Realizácia testovania bola pomocou automatického testovania(*Unit test*) a používania programu na vopred definovaných scenároch. Tie sa v každej iterácií postupne dopĺňali o novo implementované funkcionality. Nasleduje popis použitých spôsobov testovaní na konkrétnych príkladoch. Testovanie zahŕňalo platformu Windows 11 od spoločnosti Microsoft Corporation a Ubuntu 22.04 od spoločnosti Canonical. Hoci je program vytvorený primárne na základných knižniciach jazyka Python3, ktoré podporujú obe platformy, tak bolo nutné otestovať spracovanie vstupných ciest, ktoré program prijíma zadaním od užívateľa a ktorých tvar sa môže medzi platformami líšiť.

7.1 Testovanie pozície v abstraktnom syntaktickom strome

Ako bolo už spomenuté v podkapitole 6.3.1, program implementuje pridanie pozície v kóde do uzlov abstraktného syntaktického stromu, ktorý je vytvorený z príkazu SQL. Tento algoritmus bol testovaný na sade SQL rozličných SQL príkazov, ktoré vedela knižnica SQLGlot rozpoznať a spracovať. Následná kontrola správnosti pozície bola vykonaná manuálne. Pri testovaní boli vyskúšané mierne modifikácie pre realizáciu algoritmu. Tie boli porovnávané z hľadiska ich spoľahlivosti a efektívnosti, kde hlavným cieľom bolo vždy nájsť korešpondujúci token k uzlu a tým získať pre tento uzol pozíciu v kóde.

7.2 Testovanie reprezentácie schémy v pamäti

V prípade štruktúry pre reprezentáciu databázových objektov v pamäti, bola vytvorená sada automatických testov. Tie kontrolujú vznikanie objektov, správnosť uloženia dát v objektoch a ich správne začlenenie do štruktúry, ktorá reprezentuje databázovú schému v programe. Sada testov je rozdelená do niekoľkých scenárov. Tieto scenáre na seba kontextovo nadväzujú a predstavujú postupne skladanie objektov, ktoré reprezentujú databázové objekty do popísanej hierarchie. Preto je veľká pravdepodobnosť, že ak nie sú splnené podmienky testov na počiatku testov, zároveň nebudú splnené ani ďalej. K takejto dekompozícií testovania bolo prístupné v z dôvodu

väčšej zložitosti kontrolovanej dátovej štruktúry. Testovanie pokračovalo v rámci kompletného testovania programu ako celku, ktorého časťou bola práve aj táto štruktúra.

7.3 Testovanie pravidiel a výstupu

Zvyšok programu bol testovaný manuálne počas vývoja jeho používaním. Ide hlavne o definovanie a spracovanie pravidiel. Výstup programu v podobe hlásení bol rovnako kontrolovaný manuálne. Tieto kontroly zahrňovali rôzne scenáre, pri ktorých sa prejavili chybové hlásenia behu programu alebo hlásenia získané z výstupu pravidiel. Návrh pravidiel bol vďaka testovaniu prototypov medzi iteráciami vývoja dodatočne rozšírený o nové vlastnosti. Týmito vlastnosťami sú myslené predovšetkým možnosť aplikovať pravidlo na špecifický SQL príkaz alebo rozdielny rozsah platnosti dát v objekte pravidla. Získané poznatky z testovania ukázali nutnosť doplnenia týchto vlastností, aby mohli byť kontrolované aj komplexnejšie požiadavky voči SQL kódu.

Kapitola 8

Záver

Táto diplomová práca riešila možnosti overenia vlastností SQL kódu. S využitím týchto poznatkov bol vytvorený program pre kontrolu vlastností SQL kódu. Cieľom bolo, aby program pri kontrole využíval vopred definované pravidlá. Tie predstavujú súbor podmienok, ktoré musia byť pri kontrole kódu splnené. Užívateľ si ich môže definovať aj sám a pridať ich do programu. Podmienkou je, aby boli vytvorené s prihliadnutím na rozhranie, ktoré program pre pravidlá poskytuje.

V počiatku práce bolo kľúčové sa zoznámiť so štandardom jazyka SQL, jeho dialektami a typickými vlastnosťami pre dobrý SQL kód. Tým bola získaná podrobnejšia predstava o možných vstupoch v podobe jazyka SQL a jeho vlastnostiach, ktoré môžu byť kontrolované. Aby mohol byť vstupný SQL kód analyzovaný, musí byť vhodne spracovaný a uložený do dátovej štruktúry. Nasledovalo zoznámenie sa so syntaktickou analýzou programovacie jazyka, ktorého výstupom je dátová štruktúra abstraktného syntaktického stromu. Ten reprezentuje vhodnú dátovú štruktúru, nakoľko obsahuje narozdiel od derivačného stromu len podstatné informácie pre analýzu, týkajúce sa sémantiky kódu. Jednotlivé uzly stromu sú určitého typu, v závislosti od kódu, ktorý popisujú. Aby tieto uzly mohli byť kontrolované pravidlami, bol využitý návrhový vzor Návštevník. Výsledkom čoho je návrh kontroly uzlov stromu.

Pre samotné získanie abstraktného syntaktického stromu bola použitá knižnica SQLGlot. Tá bola zvolená po rozbore existujúcich knižníc pre tento účel. Požadovaná vlastnosť bola kontrola syntaktickej správnosti kódu a ďalšie funkcionality. Keďže žiadna knižnica nebola ideálna, museli byť prijaté isté kompromisy. Zároveň, rozhranie stromu získaného výstupom z knižnice nebolo kompatibilné s rozhraním, ktoré požadoval program pre analýzu uzlov stromu. Riešením bolo využitie návrhového vzoru Adaptér, ktorým bolo docielené získanie kompatibilného rozhrania. V rámci spomenutých kompromisov, muselo byť implementované riešenie, ktoré doplnilo informáciu o pozícií v kóde do jednotlivých uzlov stromu. Zároveň knižnica zatiaľ nepokrýva niektoré príkazy SQL.

Pravidlá boli implementované tak, aby poskytovali flexibilitu pri implementácií kontroly. Základný koncept pravidiel bol inšpirovaný existujúcimi riešeniami, ktorý bol prispôsobený a rozšírený o potreby programu. Program poskytuje rozhranie pre vytvorenie a spravovanie správania pravidiel. Boli implementovanie funkcionality, ktoré autorovi pravidla dovoľujú zmeniť rozsah kódu, kde bude kontrola pravidla uplatnená. Pravidlo ma zároveň prístup ku štruktúre, ktorá zaznamenáva zmeny databázovej schémy. Tým môže pravidlo kontrolovať kontext z hľadiska už spracovaného SQL kódu.

Implementácia štruktúry pre uchovávanie zmien databázovej schémy sa skladá zo súboru tried, ktoré reprezentujú databázové objekty. Tie sú usporiadané do vhodnej hierarchie, ktorá bola inšpirovaná databázovými systémami. Príkazy SQL, ktoré majú za cieľ zmenu databázovej

štruktúry, sú rozpoznané a spracované. Následne program vykoná akcie nad vlastnými štruktúrami pre uloženie zmien, ktoré príkazy SQL reprezentujú. Z hľadiska pokrytia týchto príkazov, program implementuje rozpoznanie len základných príkazov pre zmenu schémy. Stav štruktúry je možné uložiť do súboru alebo inicializovať z predchádzajúceho použitia zo súboru. Zároveň bola implementovaná voliteľná funkcionalita pre inicializáciu štruktúry z existujúcej schémy uloženej v databázovom systéme. Tu bola implementovaná podpora len pre niektoré databázové systémy, avšak implementácia bola navrhnutá pre ľahké doplnenie ďalších. Pre použitie bolo vytvorené rozhranie vo forme konfiguračného súboru, kde užívateľ musí nastaviť potrebné informácie.

Výstupom programu sú hlásenia, ktoré sú zbierané počas behu programu. Tie sa môžu týkať behu programu alebo výstupov pravidiel. Bolo implementované rozhranie pre vytvorenie týchto hlásení, proces ich spracovania a ich vypísanie užívateľovi. Cieľom bolo poskytnúť viac možností pri výpise, preto bol implementované základný a rozšírený výstup, kde miesto výstupu môže byť tiež špecifikované: na štandardný výstup alebo vybraný súbor.

Testovanie programu prebiehalo už počas jeho vývoja. Mimo iné výsledkom bolo aj pridanie niektorých funkcionalít, ktoré sa ukázali ako nutné pri kontrole kódu SQL. Program bol v tomto štádiu testovaný ako prototyp. Pred vývojom boli definované scenáre, ktoré má program zvládnuť. Tieto scenáre boli testované postupne ako sa vyvíjal prototyp programu. Fáza testovania potom analyzovala nedostatky programu z hľadiska implementácie a návrhu. Tým návrh programu a jeho následná implementácia museli byť občas prepracované. Jednou z vlastností testovania bolo aj dbať na zachovanie vhodnej dekompozície programu a riešení, kde bolo cieľom vhodné zapuzdrenie funkcionalít do samostatných celkov v programe.

Výstupom je tak program, ktorý je schopný analýzy kódu SQL na základe pravidiel. Tento program má otvorený kód a je verejne dostupný[1]. Budúci vývoj programu sa môže uberať viacerými smermi. Môže byť doplnená implementácia, ktorá poskytne podporu pre začlenenie do vývojových nástrojov. Ďalej, fungovanie programu je priamo závislé na knižnici, ktorá vytvára abstraktný syntaktický strom. V ideálnom prípade by bolo vhodné, aby program nemusel tento strom dodatočne upravovať pre svoje použitie. Pre toto môže byť zaujímavá tvorba vlastného nástroja, ktorý daný strom vytvorí na mieru alebo použitie knižnice, ktorá v budúcnosti túto funkcionalitu implementuje. Program bol z tohoto hľadiska implementovaný tak, aby bol minimalizovaný počet zmien prí prípadnej zmene knižnice pre tvorbu stromu. Zároveň môže byť implementované rozpoznanie a spracovanie väčšieho rozsahu príkazov, ktoré majú za cieľ úpravu databázovej schémy.

[1] https://github.com/FilipBali/sql_code_analyzer

Literatúra

[1] ANDERSON, R. a ROBERTS, M. *CSE401: Introduction to Compiler Construction - Abstract syntax trees* [online]. Department of Computer Science and Engineering, Univerity of Washington, 2008 [cit. 2023-05-04]. Dostupné z: https://courses.cs.washington.edu/courses/cse401/08wi/lecture/AST.pdf.

[2] DELAITRE, A., STIVALET, B., FONG, E. a OKUN, V. Evaluating Bug Finders – Test and Measurement of Static Code Analyzers. In: *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*. IEEE, 2015, s. 14–20 [cit. 2023-05-04]. DOI: 10.1109/COUFLESS.2015.10. ISBN 978-1-4673-7034-9. Dostupné z: https://ieeexplore.ieee.org/abstract/document/7181477.

[3] GAIFAX, L. *PostgreSQL Languages AST and statements prettifier.* [online]. GitHub, 2023 [cit. 2023-05-07]. Dostupné z: https://github.com/lelit/pglast.

[4] GAMMA, E., HELM, R., JOHNSON, R. E. a VLISSIDES, J. *Design patterns: Elements of reusable object-oriented software.* Prvé vydanie. Addison-Wesley Professional, 1994. ISBN 0201633612.

[5] GROVE, A. *Sqlparser-rs: Extensible SQL lexer and parser for Rust* [online]. GitHub, 2023. Dostupné z: https://github.com/sqlparser-rs/sqlparser-rs.

[6] HOLYWELL, S. *SQL style guide by Simon Holywell* [online]. November 2022 [cit. 2023-05-03]. Dostupné z: https://www.sqlstyle.guide/.

[7] ISO CENTRAL SECRETARIAT. *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework).* Štandard ISO/IEC 9075-1:2016. Ženeva, Švajčiarsko: International Organization for Standardization, December 2016 [cit. 2023-05-13]. Dostupné z: https://www.iso.org/standard/63555.html.

[8] JIANG, T., RAVIKUMAR, B., LI, M. a REGAN, K. W. *Formal grammars and Languages* [online]. 2009 [cit. 2023-05-04]. Dostupné z: https://www.cs.ucr.edu/~jiang/cs215/tao-new.pdf.

[9] KLINE, R. M. *Csc520: Foundations of Computer Science - Parse Trees for CFLs* [online]. Computer Science Department West Chester University, 2018 [cit. 2023-05-04]. Dostupné z: https://www.cs.wcupa.edu/rkline/fcs/parse-trees.html.

[10] LUNDBERG, D. J. *Lexical analysis by finite automata - Compiler Construction* [online]. Linnaeus University, October 2014 [cit. 2023-05-04]. Dostupné z: https://homepage.lnu.se/staff/jlnmsi/cc1/lexical.pdf.

[11] MELTON, J. a SIMON, A. R. *Understanding the New SQL: A Complete Guide.* Prvé vydanie. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1558602453.

[12] MENG, N. a POURSARDAR, F. *CS 3304: Comparative Languages - Lexical and syntax analysis* [online]. Virginia Tech, 2019 [cit. 2023-05-04]. Dostupné z: https://people.cs.vt.edu/prsardar/classes/cs3304-Spr19/lectures/CS3304-9-LanguageSyntax-2.pdf.

[13] MICROSOFT. *Transact-SQL syntax conventions (transact-SQL) - SQL server* [online]. December 2022 [cit. 2023-05-16]. Dostupné z: https://learn.microsoft.com/en-us/sql/t-sql/language-elements/transact-sql-syntax-conventions-transact-sql?view=sql-server-ver16.

[14] ORACLE. *Oracle8 Concepts Release 8.0, Using User-Defined Datatypes - Table Aliases* [online]. 1997 [cit. 2023-05-15]. Dostupné z: https://docs.oracle.com/cd/A58617_01/server.804/a58227/ch_objs.htm.

[15] ORACLE. *Oracle Database Online Documentation 12c Release 1 (12.1) - Better Wildcard Query Performance* [online]. May 2015 [cit. 2023-05-03]. Dostupné z: https://docs.oracle.com/database/121/CCAPP/GUID-77B6DAF7-2945-4C6E-A81C-ED6C564FBE81.htm#CCAPP9072.

[16] PAPIERNIK, M. *How To Use Primary Keys in SQL* [online]. January 2023 [cit. 2023-05-15]. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-use-primary-keys-in-sql.

[17] POLLACK, E. *Query optimization techniques in SQL Server: tips and tricks* [online]. June 2018 [cit. 2023-05-15]. Dostupné z: https://www.sqlshack.com/query-optimization-techniques-in-sql-server-tips-and-tricks/.

[18] PYTHON SOFTWARE FOUNDATION. *The Python Language Reference - Lexical analysis* [online]. May 2023 [cit. 2023-05-04]. Dostupné z: https://docs.python.org/3/reference/lexical_analysis.html.

[19] RUPARELIA, N. B. Software Development Lifecycle Models. *SIGSOFT Softw. Eng. Notes.* New York, NY, USA: Association for Computing Machinery. May 2010, zv. 35, č. 3, s. 8–13. DOI: 10.1145/1764810.1764814. ISSN 0163-5948. Dostupné z: https://doi.org/10.1145/1764810.1764814.

[20] SQLFLUFF. *The SQL Linter for Humans* [online]. GitHub, 2023 [cit. 2023-05-16]. Dostupné z: https://github.com/sqlfluff/sqlfluff.

[21] STELLATO, E. *The Benefits of Indexing Foreign Keys* [online]. November 2012 [cit. 2023-05-15]. Dostupné z: https://sqlperformance.com/2012/11/t-sql-queries/benefits-indexing-foreign-keys.

[22] THE METABASE. *Best practices for writing SQL queries* [online]. 2023 [cit. 2023-05-03]. Dostupné z: https://www.metabase.com/learn/sql-questions/sql-best-practices.

[23] THÉNAULT, S. a POPA, C. *Pylint is a static code analyser for Python 2 or 3.* [online]. GitHub, 2023 [cit. 2023-04-22]. Dostupné z: https://github.com/pylint-dev/pylint.

[24] TOBY, M. *Python SQL parser and transpiler* [online]. GitHub, 2023 [cit. 2023-04-22]. Dostupné z: https://github.com/tobymao/sqlglot.

[25] TREE-SITTER. *Tree-Sitter - Implementation* [online]. GitHub, 2023 [cit. 2023-05-08]. Dostupné z: https://tree-sitter.github.io/tree-sitter/implementation.

[26] TREE-SITTER. *Tree-Sitter - Language Bindings* [online]. GitHub, 2023 [cit. 2023-05-08]. Dostupné z: https://tree-sitter.github.io/tree-sitter/#language-bindings.

[27] TREE-SITTER. *Tree-Sitter - Parsers* [online]. GitHub, 2023 [cit. 2023-05-08]. Dostupné z: https://tree-sitter.github.io/tree-sitter/#parsers.

[28] WEINBERG, P. N., GROFF, J. R. a OPPEL, A. J. *SQL, the complete reference*. Tretie vydanie. The McGraw-Hill Companies, 2009. ISBN 9780071592550.

[29] ZAKAS, N. C. *ESLint - Pluggable JavaScript linter* [online]. June 2013 [cit. 2023-04-22]. Dostupné z: https://eslint.org/docs/latest/about/.

[30] ZHANG, P. *Practical guide to Oracle SQL, T-SQL and MySQL*. Prvé vydanie. CRC Press, Taylor & Francis Group, 2017. ISBN 9781138105188.

Príloha A

Doplnenie pozície v kóde do uzlov abstraktného syntaktického stromu

V tejto prílohe je napísaný algoritmus pomocou pseudokódu, ktorý znázorňuje postup akým je do uzlov abstraktného syntaktického stromu pridaná pozícia v kóde. Táto pozícia je získavaná z tokenov, z ktorých je vytváraný strom a ktoré algoritmus spätne identifikuje ako tvorcu uzla na základe rovnakých dát medzi tokenom a uzlom. Dôvodom tohoto postupu je absencia tejto informácie v uzle stromu, ktorý je získavaný z knižnice SQLGlot. Túto informáciu program potrebuje, aby mohol byť identifikovaný úsek kódu, ktorý je potrebný pri výpise programu. Podrobnejšie sa tejto problematike venuje podkapitola implementácie 6.3.1.

```
tokens ← tokens_function_param
line_const ← line_const_function_param
seen_tokens ← [ ]
found ← False
for each node in tree do
    tokens_to_del ← [ ]
    for each token in tokens do
        if Ak sa zhodujú dáta medzi token a node then
            Pridaj lokáciu z token do node
            found ← True
            break
        end if
        tokens_to_del.append(token)
    end for
    if found then
        seen_tokens ← tokens_to_del + seen_tokens
        tokens_to_del.clear()
    else
        for each stoken in seen_tokens do
            if Ak sa zhodujú dáta medzi stoken a node then
                Pridaj lokáciu z stoken do node
                break
            end if
        end for
    end if
end for
```

Príloha B

Obsah priloženého pamäťového média

- **sql_code_analyzer.zip** - Zdrojové kódy programu.
- **sql_code_analyzer-requirements.zip** - Závislosti potrebné pre spustenie programu.
- **sql_code_analyzer-thesis.zip** - Zdrojové kódy diplomovej práce.
- **sql_code_analyzer.pdf** - Text diplomovej práce vo formáte PDF.