FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Analýza dynamického SQL kódu v projektu Manta
Student:	Artur Nasyrov
Vedoucí:	doc. Ing. Jan Janoušek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

V datových skladech se běžně používá dynamický SQL kód. Používá se k různým účelům od jednoduchého dosazení jmen databázových objektů na základě hodnot konstant nebo proměnných až po konstrukce, kdy se kód skládá pomocí dynamických kurzorů a dotazů do databáze. Dynamický SQL kód nelze obecně zpracovat statickou analýzou. Cílem práce je navrhnout a implementovat základní sadu strategií, které alespoň v některých případech analýzu dynamického SQL kódu umožní.
1. Zanalyzujte typické případy použití dynamického SQL v datových skladech, vyjděte z dodaných skriptů. Zdrojem skriptů bude Internet a interní zdroje společnosti Manta. Výsledkem bude sada vzorů použití dynamického SQL.
2. Seznamte se s projektem Manta pro analýzu datových toků v datových skladech a způsobem jak se v projektu provádí statická analýza. Seznamte se s připravenou kostrou řešení dynamického SQL.
3. Pro vybrané vzory navhrněte a implementujte strategii, která umožní alespoň částečnou analýzu datových toků v daném vzoru.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 16. ledna 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Analýza dynamického SQL kódu v projektu Manta

Bc. Artur Nasyrov

Katedra teoretické informatiky

Vedoucí práce: doc. Ing. Jan Janoušek, Ph.D.

10. května 2018

Poděkování

Chtěl bych poděkovat týmu z Manty, především Jiřímu Touškovi a Lukáši Hermannovi, za jejich ochotu pomoc a cenné rady. Dále bych chtěl poděkovat Martinu Hnátkovi z týmu společnosti Profinit za pomoc s orientací v interních zdrojích a vedoucímu práce, doc. Ing. Janu Janouškovi, Ph.D. Závěrem bych chtěl poděkovat své rodině.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 občanského zákoníku tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům), vč. možnosti Dílo upravit či měnit, spojit jej s jiným dílem a/nebo zařadit jej do díla souborného. Toto oprávnění je časově, teritoriálně i množstevně neomezené a uděluji jej bezúplatně.

V Praze dne 10. května 2018 ……

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

NASYROV, Artur. *Analýza dynamického SQL kódu v projektu Manta.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem práce je analýza typických případů použití dynamického SQL v datových skladech a rozšíření projektu Manta sadou strategií, které umožní analýzu dynamického SQL kódu v některých případech. Předmětem práce je vyhledání typických případů užití dynamického SQL v datových skladech, návrh zlepšení existující sady strategií, následující implementace a testování výsledků.

Klíčová slova analýza dynamického SQL, návrh a implementace strategií analýzy, datové toky, databáze, Manta, Oracle, Java

Abstract

The aim of this work is to analyze the typical cases of dynamic SQL usage in the data warehouses and to extend the Manta project with a set of strategies which allow dynamic SQL analysis in some cases. The objective of this work is finding the typical cases of dynamic SQL usage in the data warehouses, the design of improvement of the existing set of strategies and subsequent implementation and testing.

Keywords dynamic SQL analysis, design and implementation of the analysis strategies, data flows, databases, Manta, Oracle, Java

Obsah

Seznam ukázek kódu

Seznam obrázků

Úvod

Typický datový sklad obsahuje desítky až stovky tabulek s velkým obsahem různých dat, která se navzájem ovlivňují. Zdrojem těchto dat je typicky velké množství různých systémů a vzniká mezi nimi velká provázanost. Metodologie Data Lineage [1] je užitečná v Business Inteligence a pomáhá zanalyzovat vztahy mezi tabulkami a co se děje s daty v průběhu různých procesů (datové transformace). Řešení Manta Flow (MF) [2] se zabývá analýzou datových transformací. Manta původně vznikla jako projekt na FIT ČVUT v rámci grantu TAČR a rozrostla se do samostatného produktu. Tento nástroj je určen pro Business analytiky, kteří pracují s datovými sklady.

V práci je provedena analýza typických případů použití dynamického SQL v datových skladech. Většina skriptů byla převzata z interních zdrojů firmy Profinit [3] bez práva na zveřejnění. Byla vybrána sada 12 skriptů v syntaxi PL/SQL [4], obsahující 50 dynamických výrazů. Další část práce pojednává o analýze třech již implementovaných strategií – *Value*, *Name* a *Identifier*, které zpracují odkazy na proměnné v dynamických výrazech (DV) a vrátí výsledky jejich vyhodnocení. DV jsou reprezentovány pomocí abstraktního syntaktického stromu (AST). Každá proměnná je uzlem tohoto stromu. Po zpracování vybraných skriptů bylo zjištěno, že není nutné implementovat novou strategii, ale je potřeba upravit existující Name strategii. Byla přidána funkcionalita detekce selhání strategií. Základem této funkcionality je použití regulárních výrazů pro reprezentaci obsahu dříve vyhodnocené části AST v celkovém procesu vyhodnocení.

V původní implementaci byla v případě selhání Value strategie použita Identifier strategie jako náhradní (*fallback*) strategie, která v části případu vrátila nepřesné vyhodnocení. V této implementaci byla za fallback strategii zvolena upravená Name strategie. Výsledkem bylo zmenšení počtu nepřesných vyhodnocení odkazu na proměnné.

Motivace

Zvolil jsem si toto téma, protože zlepšením současných strategií analýzy dynamického SQL se zlepší Manta jako produkt. To bude přínosné pro koncové zákazníky a pomůže to společnosti získat potenciálně více zákazníků.

Organizace textu

V kapitole 1 je uvedena teoretická část týkající se AST a stručný popis syntaktických konstrukcí jazyka PL/SQL – dialekt Oracle. Také zde jsou popsány části architektury systému MF, se kterými se setkáváme v následujících kapitolách této práce.

V kapitole 2 se zabýváme analýzou typických případů užití dynamického SQL v datových skladech a analýzou již existujících strategií zpracování dynamického SQL.

V kapitole 3 je popsán návrh zlepšení strategií pro vybrané vzory. Některé implementační detaily strategií jsou uvedeny v kapitole 4.

Poslední kapitola 5 se zabývá výsledky testů.

Kapitola 1

Základní pojmy

V této kapitole je uvedena teoretická část týkající se AST a jsou popsány použité technologie.

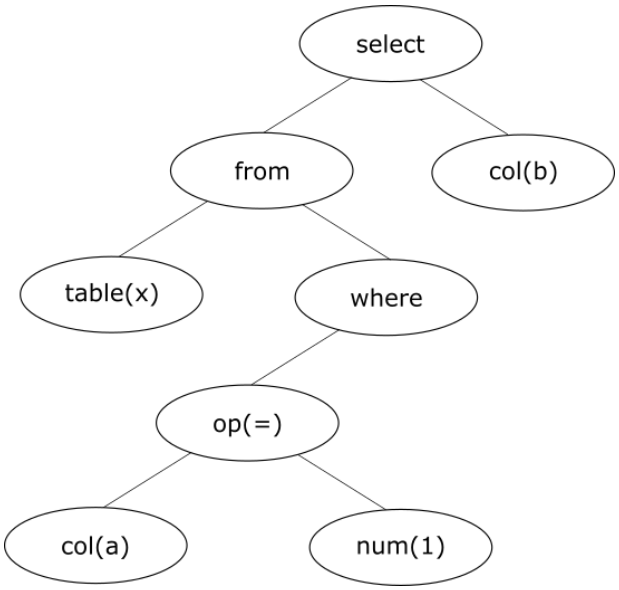
1.1 Konstrukce AST

Abstraktní syntaktický strom (AST) je stromová struktura, u které jsou v uzlech zachyceny programovací konstrukty [5, s. 69]. Příkladem programovacích konstruktů pro jazyk PL/SQL mohou být následující moduly:

- Procedury
- Funkce
- Balíčky (Packages)
- Triggery
- Typy objektů

Komentáře, proměnné, cykly, kurzory a pole jsou také příklady konstruktů.

ANTLR [6] je nástroj, který slouží pro generování jednoduchého překladače, řízeného LL syntaktickou analýzou [5, kapitoly Syntax Analisys a Syntax-Directed Translations]. Vygenerovaný překladač načte vstupní text a transformuje jej do AST. Příklad AST používaný v MF ukazuje obrázek 1.1.



Obrázek 1.1: AST pro SQL příkaz „select b from t where a = 1“.

1.2 Oracle PL/SQL

1.2.1 Dynamické SQL

Dynamické SQL je programovací metoda určená pro generování a spouštění SQL příkazů za běhu † [7]. Dynamické SQL je užitečné, když v době kompilace není známo plné znění SQL dotazů. Kód z ukázky 1.1 je příkladem vhodného použití dynamického SQL. Název tabulky, do které bude uložen řádek, je v době kompilace neznámý a může se měnit, taktéž je neznámá vkládaná hodnota.

```
CREATE PROCEDURE insert_table_code ( code VARCHAR2)
IS
  t_name varchar2(200);
BEGIN
  t_name := mnt.getParam('code_table_name');
  EXECUTE IMMEDIATE
    'INSERT INTO ' || t_name
      || ' (code) VALUES (:code)' USING code;
END;
```

Ukázka kódu 1.1: Příklad dynamického SQL

† v runtime

PL/SQL umožnuje dva způsoby zápisu dynamického SQL:

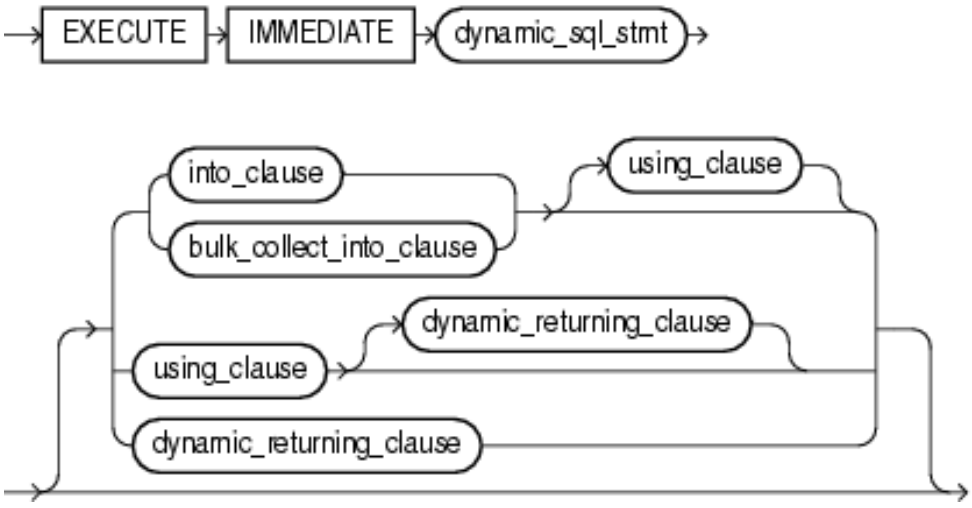
- Native Dynamic SQL (NDS) [7, sekce Native Dynamic SQL]
- DBMS_SQL [7, sekce DBMS_SQL]

NDS je součásti jazyka PL/SQL, která umožnuje jednodušší způsob zápisu. Dynamické výrazy jsou přímo uloženy do PL/SQL kódu. Pro spuštění DP ve se většině případů se používá příkaz `EXECUTE IMMEDIATE`. Dalšími možnostmi je použití příkazů `OPEN FOR`, `FETCH` a `CLOSE`.

DBMS_SQL je knihovna Oracle, která umožnuje pokročilejší práci s dynamickým SQL pomocí speciálního kurzoru, reprezentovaného číslem[†]. Porovnání NDS a DBMS_SQL je detailně popsáno v dokumentaci Oracle [8, kapitola Choosing Between Native Dynamic SQL and the DBMS_SQL Package]. Tato práce se bude zabývat NDS.

1.2.2 Struktura dynamického výrazu

Na diagramu 1.2 je zobrazena struktura `EXECUTE IMMEDIATE` příkazu [9]. V diagramu nás zajímá blok **dynamic_sql_stmt** - dynamický výraz.



Obrázek 1.2: Struktura EXECUTE IMMEDIATE příkazu [10].

Dynamický výraz může být literál, řetězcová proměnná nebo řetězcový výraz. DV reprezentuje jakýkoliv SQL výraz nebo PL/SQL blok.

[†]SQL cursor number

DV může být následujících typů:

- CHAR
- VARCHAR2
- CLOB

Řetězec může obsahovat tzv. placeholdery, které budou použity k provázání řetězce s argumenty. Provázat s řetězcem nelze názvy objektů schématu, jako např. názvy tabulek nebo názvy sloupců [11, s. 539].

Kód z ukázky 1.1 je příkladem dynamického výrazu. V této ukázce DV je přítomen řetězcový výraz, obsahující zřetězení literálů a proměnné **t_name**. Proměnné **t_name** bude za běhu programu přiřazena hodnota volání funkce **getParam** z balíčku **mnt**. Druhý literál obsahuje placeholder „**:code**“, který je provázán s parametrem **code** dané procedury.

1.3 Reprezentace AST v MF

Jak bylo popsáno v sekci „Syntaktická analýza“, AST reprezentuje vstupní skript. V této sekci bude uvedeno, jak AST vypadá. Pro vygenerování stromu slouží překladač `PLSQLMain`, který vrátí instanci třídy `OracleAstNode`, která dědí od `MantaAstNode`. Třída `MantaAstNode` je potomkem třídy `CommonTree`, která se nachází v balíčku `org.antlr.runtime.tree` nástroje ANTLR.

Třída `CommonTree` obsahuje metody `children` a `parent`, které slouží k prohledávání stromu, a další metody `addChild`, `deleteChild` atd., které slouží ke generování stromu [12, kapitola Tree Construction].

Důležitým atributem třídy `OracleAstNode` je privátní atribut `contextState` typu `OracleContextState`. Třída `OracleContextState` obsahuje privátní atribut `navigator` typu `MantaAstNavigator<OracleAstNode>`. Třída `MantaAstNavigator`, dle názvu, slouží k navigaci ve stromu. K navigaci lze využít XPath dotazů [13]. Některé metody třídy `OracleAstNode`, například `selectNodes` nebo `selectSingleNode`, využívají XPath dotaz jako svůj parametr. Dalšími užitečnými metody třídy `MantaAstNavigator` jsou `getLeftSibling`, která vrátí levého sourozence a `getParent`, která vrátí rodiče.

Třída `CommonTree` také obsahuje atribut `token` typu `Token`. Důležitou metodou třídy `OracleAstNode` je metoda `getTokenName`, která vrátí název tokenu. Token označuje typ uzlu.

Důležitými tokeny, které budou použity v dalších částech práce, jsou:

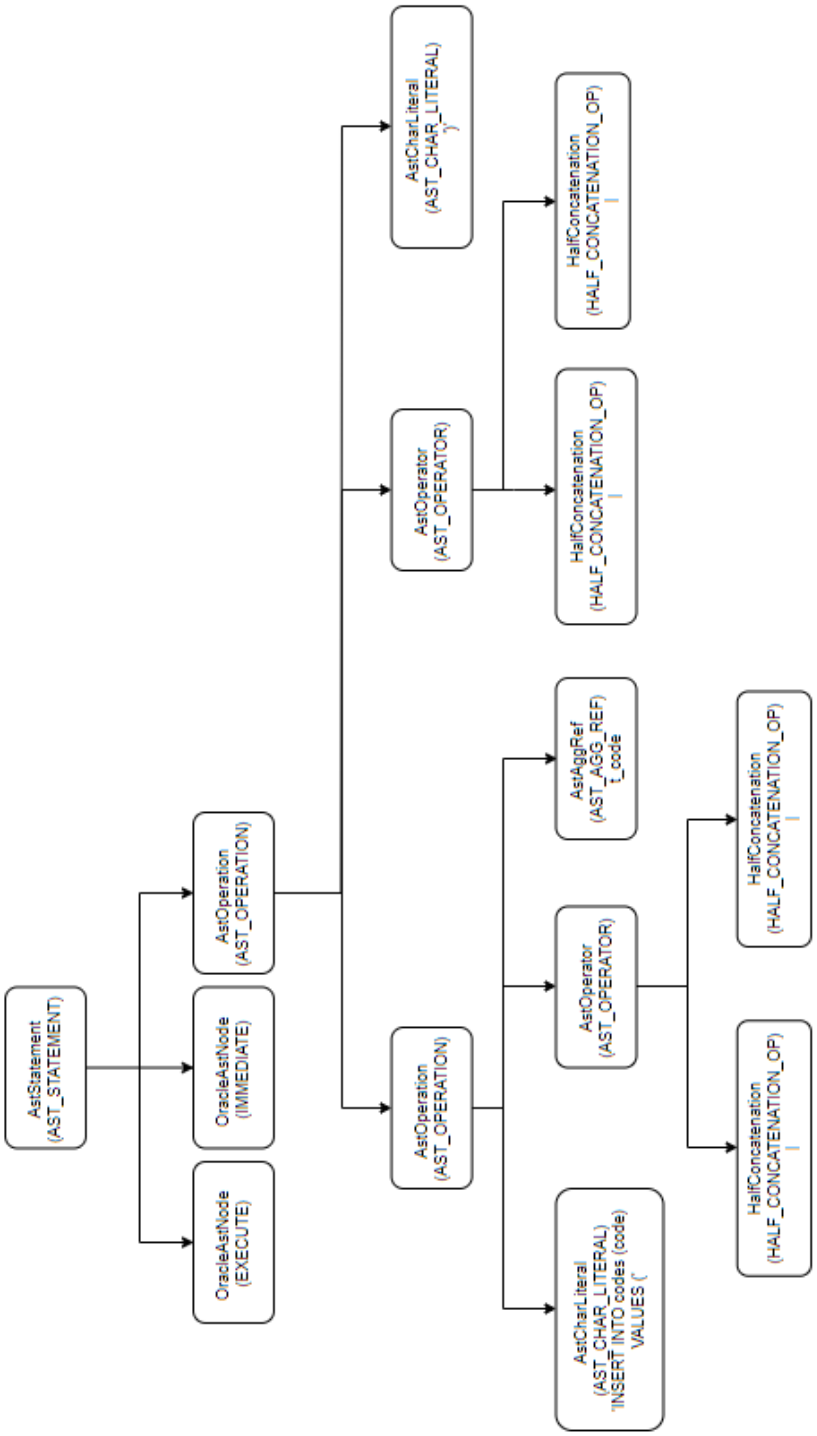
- `AST_STATEMENT` - token reprezentující příkaz.
- `AST_AGG_REF` - token reprezentující odkaz na proměnnou.
- `AST_CHAR_LITERAL` - token reprezentující literál.
- `AST_OPERATION` - token reprezentující operaci mezi dvěma uzly.
- `AST_OPERATOR` - token reprezentující operator.
- `HALF_CONCATENATION_OP` - token reprezentující symbol „|“.

Třída `MantaAstNode` obsahuje metodu `toNormalizedString` pro zjištení obsahu podstromu.

Ukážeme zjednodušené na obrázku 1.3 jak by mohl vypadat v MF následující výraz

```
EXECUTE IMMEDIATE 'INSERT INTO codes (code) VALUES (' || t_code || ')';
```

V prvním řádku je název třídy, která dědí od třídy `OracleAstNode`. Této třídy reprezentují některé důležité uzly. V druhém řádku je název tokenu. V třetím řádku, pokud je, je obsah uzlu.



Obrázek 1.3: Příklad reprezentace AST v MF.

1.4 Regulární výrazy

Regulární výraz (RV) je posloupnost znaků řetězce, která podle daných pravidel definuje vzor vyhledávání textu ve vstupním řetězci. V této sekci stručně popíšeme pravidla vytváření RV. Podrobnější návod ke tvorbě a použití RV může být nalezen v odborné literatuře, například [14].

V tabulkách této sekce se nacházejí příklady použití RV. V prvním sloupci je vstup, ve druhém sloupci je samotný RV a ve třetím sloupci podtrženě jsou označeny shody daného RV pro zadaný vstup.

Jednotlivé znaky

Nejjednodušší RV se skládají z jednotlivých znaků:

- konkrétní znak (písmeno, číslice apod.),
- **\t** (tabulátor),
- **\n** (konec řádky).

Pokud znak v RV má speciální význam, jedná se o **metaznak**. Příklady použití jednotlivých znaků v RV jsou uvedeny v tabulce 1.1.

Vstup	RV	Výstup
manta flow	a	m<u>a</u>nt<u>a</u> flow
manta flow	n	ma<u>n</u>ta flow

Tabulka 1.1: Příklady RV, které se skládají z jednotlivých znaku.

Zástupné znaky

Dalšími užitečnými znaky jsou tzv. zástupné znaky:

- . (tečka) – zastupuje libovolný znak s výjimkou konce řádky.
- **\s** – reprezentuje jeden z následujících znaků:
  - mezera,
  - tabulátor,
  - konec řádky.

- **\w** – reprezentuje jeden z následujících znaků:
  - písmeno,
  - číslice,
  - podtržítko,
  - tabulátor,
  - konec řádky.

Pokud chceme zapsat znak „.“ jako jednotlivý znak, nikoliv zástupný, musíme jej escapovat. To znamená, že k jednotlivému znaku přidáme na začátek únikový znak[†] „\“, nebo „\\“, pokud pracujeme s řetězcem v Javě. Escapování již bylo použito v zápisu tabulátoru a konce řádky. Protože znaky „\ ^ $. + * ? | [{ } ( )“ mají speciální význam, je nutně je escapovat. Příklady užití zástupných znaků jsou uvedeny v tabulce 1.2.

Vstup	RV	Výstup
manta&flow	.	manta&flow
manta&flow	\w	manta&flow

Tabulka 1.2: Příklady RV, které se skládají ze zástupných znaků.

Množiny znaků

Znaky lze seskládat do množin. V tom případu se ve vstupu vyhledává libovolný znak z množiny. Metaznakem „^“ označujeme doplněk této množiny a vyhledáváme ve vstupu libovolný znak, který není v dané množině. Metaznakem „-“ označujeme rozsah znaků.

Příklady množin znaků:

- **[abcd]** množina znaků,
- **[^abcd]** doplněk množiny,
- **[a-dA-D]** rozsah znaků (znaky a až d a znaky A-D),
- **[0-9]** rozsah číslic (číslice 0 až 9).

Příklady použití množin znaků jsou uvedeny v tabulce 1.3.

[†] escape symbol

Vstup	RV	Výstup
manta flow	[an]	manta flow
manta&flow	[a&]	manta&flow

Tabulka 1.3: Příklady RV, které se skládají z množiny znaků.

Skupiny znaků

V praxi RV často obsahuje posloupnost jednotlivých a zástupných znaků. V takovém případě pro úspěšnou shodu musí být zvolené znaky vedle sebe. Pokud je posloupnost uzavřena v závorkách, vzniká **skupina**. Skupina je část RV, která je uzavřena v závorkách a může být extrahována z výsledků úspěšné shody. Každé skupině je přiřazeno číslo nebo můžeme definovat název skupiny pomocí konstrukce `?<jméno skupiny>`. Čísla jsou skupinám přiražena automaticky od nuly a dále podle výskytu v RV. Příklady použití skupin znaků jsou uvedeny v tabulce 1.4. Buňky v posledním sloupci tabulky obsahují čísla a názvy nalezených skupin, pokud jsou definovány.

Vstup	RV	Výstup
manta flow	(a\sf)	0: manta flow
manta flow	t(?<s1>a\sf)(?<s2>l)	0: manta flow 1, s1: manta flow 2, s2: manta flow

Tabulka 1.4: Příklady RV s použitím skupin znaků.

Look-around

Look-around je specialní skupina znaků, která umožnuje ověřit, zda před/za aktuální pozicí kurzoru v textu (ne)následuje RV [15]. V této práci bylo použito negativní dopředné vyhledávání pomocí použití look-around znaku `?!`. Příklad použití tohoto druhu vyhledávání je uveden v tabulce 1.5.

Vstup	RV	Výstup
nonsens	ns(?!e)	nonsens

Tabulka 1.5: Příklad RV s použitím look-around znaků.

Metaznak |

Nechť máme na vstupu text „manta flow“ a rovněž následující RV, který obsahuje dvě skupiny **s1** a **s2**:

```
(?<s1>manta)\s(?<s2>flow)
```

Výstupem bude shoda, která obsahuje následující skupiny:

- 0: manta flow
- 1, s1: manta flow
- 2, s2: manta flow

Pokud nás zajímá libovolný výsledek, tj. výsledek skupiny s1 nebo výsledek skupiny s2, můžeme použit metaznak „|“, který znamená „X nebo Y“. Použití metaznaku „|“ tudíž znamená, že vznikne nová shoda, například pro RV

```
(?<s1>manta)|(?<s2>flow)
```

budou výstupem dvě následující shody:

1. shoda:
    - 0: manta flow
    - 1, s1: manta flow
2. shoda:
    - 0: manta flow
    - 1, s2: manta flow

Hranice vyhledávání

Regulární výrazy umožnují specifikovat, kde se musí nacházet výsledek vyhledávání ve vstupním textu:

- ^ – začátek vstupního řádku,
- $ – konec vstupního řádku,
- **\b** – hranice slova.

Příklady použití metaznaků hranic vyhledávání jsou uváděny v tabulce 1.6.

Vstup	RV	Výstup
manta flow	\bflow$	manta flow
manta flow	^man	manta flow
manta flow	^man\b	manta flow
manta flow	^manta.\b	manta flow

Tabulka 1.6: Příklady RV s použitím metaznaků hranic vyhledávání.

Kvantifikátory

Kvantifikátory slouží k určení počtu opakování předchozího znaku nebo předchozí skupiny ve shodě.

- **?** – znak (skupina) se opakuje jedenkrát nebo se nevyskytuje,
- **?*** – znak (skupina) se opakuje jedenkrát nebo vícekrát, nebo se nevyskytuje,
- **?+** – znak (skupina) se opakuje jedenkrát nebo vícekrát,
- **{n}** – znak (skupina) se opakuje c *n*-krát,
- **{n,}** – znak (skupina) se opakuje více nebo přesně *n*-krát,
- **{k,n}** – znak (skupina) se opakuje *k*-krát až *n*-krát,
- * – znak (skupina) se opakuje co nejvíce možně,
- *? – znak (skupina) se opakuje co nejméně možně.

Příklady použití kvantifikátorů jsou uváděny v tabulce 1.7.

Modifikátory

Názvy flagů modifikátorů se liší podle implementace. V této sekci jsou uvedeny názvy, které se používají v Javě verze 1.7 [16, Sekce Field Summary]. V regulárních výrazech se rozlišují velká a malá písmena. Flag modifikátoru `CASE_INSENSITIVE` slouží pro vypnutí tohoto rozlišení. Dalším užitečným flagem

Vstup	RV	Výstup
nonsens	(ns.{0,1}){2}	0: nonsens 1: nonsens
nonsens	(.?ns){2}	0: nonsens 1: nonsens
nonsens	(.*ns){2}	0: nonsens 1: nonsens

Tabulka 1.7: Příklady RV s použitím kvantifikátorů.

modifikátoru je flag `MULTILINE`. Pokud chceme hledat shody v řádcích, nikoliv v celém textu, použijeme tento flag.

Práce s RV v Javě

Balíček `java.util.regex` obsahuje třídy `Pattern` [16] a `Matcher` [17], které slouží pro práci s RV. Popíšeme základní možnosti těchto tříd.

Třída `Pattern` slouží pro validaci a přeložení vzoru s určenými flagy do vnitřní reprezentace pomocí statické metody `Pattern compile(String regex, int flags)`. Parametr `flags` je bitová maska, která obsahuje hodnoty flagů modifikátoru. Voláním metody `matcher(CharSequence input)` pro zadaný vstup dostaneme instanci třídy `Matcher`.

Třída `Matcher` má metody pro prohledávání podle RV:

- `boolean matches()` – porovnává celý řetězec s RV a vrátí `true`, pokud je přesná shoda.
- `boolean find()` – vyhledá ve vstupu shody, které odpovídají RV. Vyhledávání probíhá postupně - metoda hledá shodu od začátku vstupu, při dalším volání metoda hledá shodu od místa, kde byla nalezena předchozí shoda. Pokud shoda není nalezena, vrátí `false`.
- `String group(String name)` – vrátí hodnotu z předchozí shody pro skupinu se stanoveným jménem.
- `String group(int group)` – vrátí hodnotu z předchozí shody pro danou skupinu.

Kapitola 2

Analýza

V této kapitole je analyzována současná situace a na základě této analýzy vyvstává řada problémů.

2.1 Případy použití dynamického SQL

Nejdříve proběhlo seznámení s dodanými skripty. Většina skriptů patřila do testovací skupiny, tj. obsahovaly jednoduché příkazy, abychom zkontrolovali základní funkčnost strategií vyhodnocení odkazu na proměné, které budou popsány v další podkapitole 2.2. Zbytek byl dodán zákazníkem a obsahoval balíček 15 dynamických výrazů různé složitosti.

První možností rozšíření sady vzorů použití dynamického SQL bylo vyhledávání ve vnitřních projektových zdrojích. Další možností bylo vyhledávání vzorů v otevřených zdrojích – GitHub repozitářích [18, 19, 20].

Celkem bylo nalezeno 12 skriptů, obsahujících 50 dynamických výrazů, které jsou rozděleny do následujících skupin:

1. Vkládání a mazání dat
2. Audit (historizace)
3. Vytvoření a smazání představení a tabulky
4. Práce s uživateli

Nejzajímavější případy se nacházely v prvních třech skupinách, kde většina výrazů obsahovala odkazy na několik proměnných, identifikátorů, nebo zároveň proměnné a identifikátory. Příkladem takového případu je skript 2.1 výpisu počtu řádků v tabulkách uživatele do jiné tabulky se statistikou, obsahující tyto kombinace. V tomto skriptu se nacházejí dva dynamické výrazy. První výraz obsahuje odkaz na proměnnou **select_sql**. Proměnné **select_sql** je přiražena hodnota zřetězení řetězců, proměnné **table_name** a identifikátoru **param_owner**. Další přiřazení neproběhne. Druhý výraz obsahuje odkaz na identifikátor **param_stats** a proměnnou **sql_values**. Proměnné **sql_values** je přiražena hodnota zřetězení řetězců, atd.

```
CREATE OR REPLACE PROCEDURE collect_count
  (param_owner VARCHAR2, param_stats VARCHAR2)
AS
  TYPE CUR_TYPE IS REF CURSOR;
  cur   CUR_TYPE;
  table_name VARCHAR2(1000);
  select_sql VARCHAR2(1000);
  insert_sql VARCHAR2(1000);
  sql_values VARCHAR2(1000);
  count_res NUMBER(19,0);
BEGIN
  OPEN cur FOR
    SELECT DISTINCT OBJECT_NAME
      FROM ALL_OBJECTS
      WHERE OBJECT_TYPE = 'TABLE'
        AND OWNER = param_owner;
  LOOP
    FETCH cur INTO table_name;
    EXIT WHEN cur%NOTFOUND;

    select_sql := 'select count(*) from '
      || param_owner || '.' || table_name;
    EXECUTE IMMEDIATE select_sql INTO count_res;

    sql_values := '''' || param_owner || ''', '''
      || table_name || ''',' || count_res;
    insert_sql := 'insert into ' || param_stats
      || ' (owner, table_name, count) values ('
      || sql_values || ')';
    EXECUTE IMMEDIATE insert_sql;

    COMMIT;
  END LOOP;
  CLOSE cur;
END;
```

Ukázka kódu 2.1: Příklad skriptu výpisu počtu řádků v tabulkách uživatele do jiné tabulky.

2.2 Existující strategie zpracování dynamického SQL

Každý dynamický výraz může být zanalyzován jednou ze třech již implementovaných strategií. V následujících sekcích tyto strategie budou popsány detailně včetně jejich vhodných a nevhodných případů použití.

2.2.1 Value strategie

Value strategie vrací možné hodnoty nalezených odkazů. Strategie je vhodná ve všech případech mimo následujících:

1. Pokud hodnota odkazu nemůže být zjištěna (odkaz není proměnnou), pak je nutno spustit *fallback* (jinou) strategii.
2. Pokud hodnota odkazu nemůže být zjištěna vůbec, pak je nutno spustit *fallback* strategii na části, které nebyly vyhodnoceny.

Příkladem použití Value strategie může být následující skript 2.2 vkládání kódu do neznámé tabulky.

```
CREATE PROCEDURE insert_table_code (table_name VARCHAR2, code VARCHAR2)
IS
  stmt_str VARCHAR2(100);
BEGIN
  stmt_str := 'INSERT INTO ' || table_name
    || ' (code) values (:code)';

  EXECUTE IMMEDIATE stmt_str USING code;
END;
```

Ukázka kódu 2.2: Příklad výrazu ve skriptu, pro který je Value strategie částečně vhodná.

Skript obsahuje jeden dynamický výraz

```
EXECUTE IMMEDIATE stmt_str USING code
```

Proměnné **stmt_str** z ukázky 2.2 je přirazen výraz, který v sobě obsahuje dvě operace zřetězení:

```
'INSERT INTO ' || table_name || ' values (:code)'
```

Hodnota odkazu na identifikátor **table_name** nemůže být zjištěna, a proto na tento odkaz bude spuštěna jiná strategie.

Možným výsledkem vyhodnocení dynamického výrazu bude výraz

```
INSERT INTO table_name values (:code)
```

2.2.2 Name strategie

Name strategie vrací původní text odkazu. Strategie je vhodná, pokud proměnná je použita jako hodnota, ale není vhodná v následujících případech:

1. Pokud hodnota má být součástí SQL kódu (např. výsledek volání `TRIM` funkce apod.). Příkladem selhání Name strategie je výraz ve skriptu z ukázky 2.3. Výsledek vyhodnocení dynamického výrazu bude neplatný z hlediska gramatiky PL/SQL výrazů

   ```
   INSERT INTO codes (code) VALUES (trim(code)x)
   ```

   , protože obsahuje neplatný symbol „)x“ ve vkladané hodnotě „trim(code)x“.

2. Pokud hodnota má tvořit část jména entity. Příkladem selhání Name strategie je výraz ve skriptu z ukázky 2.4. Výsledek vyhodnocení dynamického výrazu bude neplatný z hlediska gramatiky PL/SQL výrazů

   ```
   INSERT INTO mnt.getParam('code_table_name')
     (code) VALUES (:code)
   ```

   , protože obsahuje neplatný symbol „(“ ve jménu tabulky.

```
CREATE PROCEDURE insert_code (code VARCHAR2)
IS
  t_code VARCHAR2;
BEGIN
  t_code := trim(code) || 'x';
  EXECUTE IMMEDIATE 'INSERT INTO codes (code) VALUES (' || t_code || ')';
END;
```

Ukázka kódu 2.3: 1. příklad výrazu ve skriptu, pro který Name strategie není vhodná.

```
CREATE PROCEDURE insert_code_trim (code VARCHAR2)
IS
BEGIN
   EXECUTE IMMEDIATE
    'INSERT INTO ' || mnt.getParam('code_table_name')
      || ' (code) VALUES (:code)' USING code;
END;
```

Ukázka kódu 2.4: 2. příklad výrazu ve skriptu, pro který Name strategie není vhodná.

Příkladem úspešného zpracování Name strategie je výraz ve skriptu z ukázky 2.5.

```
CREATE TABLE t1 (
  a INT,
  b INT,
  c VARCHAR2
);
/

CREATE TABLE t2 (
  a INT,
  b INT
);
/

CREATE OR REPLACE PROCEDURE insert_into_t2 (x VARCHAR2) AS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO t2' || ' SELECT a, b FROM t1'
    || ' WHERE TO_NUMBER(c) = ' || (TO_NUMBER(x) + 1);
END;
```

Ukázka kódu 2.5: Příklad výrazu ve skriptu, pro který je Name strategie vhodná.

Výsledkem vyhodnocení dynamického výrazu bude platný výraz

```
INSERT INTO t2 SELECT a, b FROM t1 WHERE TO_NUMBER(c) = TO_NUMBER(x) + 1
```

2.2.3 Identifier strategie

Identifier strategie vrací původní text odkazu, ale upraví jej tak, aby byl použitelný, když je částí identifikátoru. Identifier strategie je v takových případech použití vhodná. Při vyhodnocení se zachovají všechny abecední symboly a číslice z textu odkazu. Ostatní symboly budou vyměněny za symbol podtržítka „_“. Příkladem nevhodného použití strategie je výraz v ukázce 2.6.

```
CREATE OR REPLACE PROCEDURE insert_audit_delete (
    schema_name VARCHAR2,
    table_name VARCHAR2, entity_id NUMBER)
AS
    sql_values VARCHAR2 (1000);
BEGIN
    sql_values := entity_id || ',to_date('''
      || to_char(SYSDATE, 'DD.MM.YYYY HH24:MI:SS')
      || ''','''DD.MM.YYYY HH24:MI:SS'''),'''Delete''';

    EXECUTE IMMEDIATE 'INSERT INTO ' || schema_name
      || '.H' || table_name
      || ' (id, updated, operation_type) VALUES ( '
```

```
      || sql_values || ')';
END;
```

Ukázka kódu 2.6: Příklad výrazu ve skriptu, pro který Identifier strategie není vhodná.

Proměnné **sql_values** je přirazen výraz, obsahující operaci zřetězení a odkaz na volání funkce **to_char**:

```
to_char(SYSDATE, 'DD.MM.YYYY HH24:MI:SS')
```

Při vyhodnocení proběhne výměna symbolů a výsledkem bude výraz

```
to_char_SYSDATE___DD_MM_YYYY_HH24_MI_SS__
```

Jedná se o nepřesné vyhodnocení, ale na rozdíl od Identifier strategie, Name strategie by selhala, protože by vrátila neplatný výraz, ve kterém chybějí další dva apostrofy u podřetězce „'DD.MM.YYYY HH24:MI:SS'“ v parametru funkce **to_date**:

```
INSERT INTO schema_name.Htable_name
  (id, updated, operation_type) VALUES (
  entity_id,to_date('to_char(SYSDATE,
  'DD.MM.YYYY HH24:MI:SS')','DD.MM.YYYY HH24:MI:SS'),
  'Delete')
```

Pokud se podíváme na skript z ukázky 2.4, Identifier strategie bude znovu užitečná, protože vyhodnocením dynamického výrazu bude platný výraz

```
INSERT INTO mnt_getParam__code_table_name__
  (code) VALUES (:code)
```

2.2.4 Současné problémy

Každá z popsaných strategií má své omezení, proto při vyhodnocení výrazu je potřeba strategie kombinovat. První z možností kombinací je použiti *fallback* strategie, jak to již bylo implementováno ve Value strategii. Nevýhodou je to, že může být použita pouze jedna fallback strategie pro celkové vyhodnocení výrazů skriptu. Další alternativou, aby bylo možné použít kombinaci fallback strategií, je pro každý odkaz spustit všechny strategie. Tím pádem se exponenciálně zvětší počet variant vyhodnocení. Pokud máme n neohodnocených odkazů

a dvě fallback strategie, pak výsledkem bude 2^n variant. Z výkonnostních důvodů, maximální počet možností se rovná 128. Proto musíme omezit případy vyhodnocení odkazů takovým způsobem.

Pokud zjistíme pomocí heuristiky místa, kde je pravděpodobné selhání jedné fallback strategie, pouze tehdy spustíme jinou strategii. Tato práce se zabývá nativními dynamickými výrazy. Proto místy s pravděpodobným selháním jedné ze jmenovaných strategií bude obsah literalů. V případě `EXECUTE IMMEDIATE` příkazu je možné zřetězení odkazu na proměnnou nebo identifikátor s literály. Lze tedy použit sadu regulárních výrazů (RV) pro kontrolu obsahu [21]. Jejich výsledek bude signálem selhání. Podobným způsobem zkontrolujeme obsah literálů, použitých v dynamických výrazech v kurzorech. Algoritmus vyhodnocení s detekcí selhání bude popsán detailně v následující kapitole.

Kapitola 3

Návrh

V této kapitole v první sekci jsou probrány možné výskyty odkazů na proměnné a identifikátory v dynamických výrazech, jak se spojují s literály a jaký mají význam možná vyhodnocení odkazů z hlediska sémantické analýzy [22, s. 17–18]. V druhé sekci je popsán algoritmus kontroly obsahu spojených uzlů.

3.1 Varianty výskytů odkazů na proměnné

V této části práce je popsáno, kde se mohou objevit proměnné v dynamických výrazech. Představíme to na příkladu skriptu z úkazky 3.1

```
CREATE OR REPLACE PROCEDURE insert_audit_delete (
  schema_name VARCHAR2,
  table_name VARCHAR2, entity_id NUMBER)
AS
  audit_table_name VARCHAR2 (1000);
  sql_values VARCHAR2 (1000);
  sql_stmt VARCHAR2 (1000);
BEGIN
  audit_table_name := get_audit_table(schema_name, table_name);
  sql_values := entity_id || ',to_date('''
    || to_char(SYSDATE, 'DD.MM.YYYY HH24:MI:SS')
    || ''','''DD.MM.YYYY HH24:MI:SS'''),'''Delete''';
  sql_stmt := 'INSERT INTO ' || audit_table_name
    || ' (id, updated, operation_type) VALUES ( ' || sql_values || ')';
  EXECUTE IMMEDIATE sql_stmt;
END;
/

CREATE OR REPLACE PROCEDURE create_audit_table_copy (
  p_schema_name VARCHAR2,
```

```
  p_table_name VARCHAR2, p_condition VARCHAR2)
AS
  audit_table_name VARCHAR2 (1000);
BEGIN
  audit_table_name := get_audit_table(p_schema_name, p_table_name);
  EXECUTE IMMEDIATE 'INSERT INTO copy_' || audit_table_name
    || '_t SELECT * FROM ' || audit_table_name
    || ' WHERE ' || p_condition;
END;
```

Ukázka kódu 3.1: Příklad výskytu odkazů na proměnné v různých částech kódu.

Možné varianty výskytu odkazů na proměnné jsou:

- Výskyt v parametru funkce.
- Výskyt v názvu tabulky, schématu, nebo sloupce.
- Výskyt vedle klíčového slova.

3.1.1 Výskyt v parametru funkce

Podíváme se na část příkazu z ukázky 3.1 (metoda **insert_audit_delete**).

```
'to_date(''' || to_char(SYSDATE, 'DD.MM.YYYY HH24:MI:SS')
  || ''',''DD.MM.YYYY HH24:MI:SS'')'
```

Ukázka kódu 3.2: Příklad výskytu odkazu na proměnnou v parametru funkce.

Vlevo od odkazu na proměnnou **to_char** je literál, který obsahuje část volání funkce **to_date**. V tomto literálu má volání funkce pouze otevírací závorku, tj. začátek seznamu parametrů funkce. Vpravo od odkazu na proměnnou se nachází literál, který obsahuje zbytek seznamu parametrů funkce **to_date**.

Protože proměnná **to_char** je funkce, možným vyhodnocením odkazu na tuto proměnnou by mohl být název té funkce. Tím pádem prvním prvkem seznamu parametrů funkce **to_date** by byl řetězec „'to_char'“. V takovém případě bude celkovým vyhodnocením zřetězení literálů a odkazu na proměnnou z ukázky 3.2 výraz

```
'to_date(''to_char'',''DD.MM.YYYY HH24:MI:SS'')'
```

Jednoduchou úpravou by bylo možné zachovat plný text odkazu. Přidáním apostrofu ke konci literálu vlevo od volání funkce a apostrofu k začátku literálu vpravo od volání funkce by vznikl jiný parametr funkce **to_date**. Jeho hodnota by byla rovna výrazu zřetězení apostrofů a funkce **to_char**. V takovém případu

bude celkovým vyhodnocením zřetězení literálů a odkazu na proměnnou z ukázky 3.2 výraz

```
'to_date(''''||to_char||'''', ''DD.MM.YYYY HH24:MI:SS'')'
```

Algoritmus detekce selhání by měl správně detekovat, zda je možné zachování plného textu odkazu.

3.1.2 Výskyt v názvu tabulky, schématu, nebo sloupce

Podíváme se na část příkazu z ukázky 3.1 (metoda **insert_audit_delete**).

```
audit_table_name := get_audit_table(schema_name, table_name);
-- ...
sql_stmt := 'INSERT INTO ' || audit_table_name  -- ...
```

Ukázka kódu 3.3: Příklad výskytu odkazu na proměnou v názvu tabulky.

V jazyce PL/SQL má příkaz `INSERT` složitou syntaxi [23]. `INSERT` příkaz z ukázky 3.1 má následující tvar:

```
INSERT INTO table (column_1, ... column_n)
  VALUES (value_1, ... value_n);
```

Odkaz na proměnnou **audit_table_name** se vyskytuje v názvu tabulky. Podle pravidel jazyka PL/SQL [24] hodnota odkazu na proměnnou **audit_table_name** může mít v obsahu pouze symboly abecedy, číslice a také řadu speciálních symbolů: _, #, $, '. V SQL příkazu má proměnná **audit_table_name** přiraženou hodnotu, která je výsledkem volání funkce **get_audit_table**. Do vyhodnocení odkazu na proměnnou **audit_table_name** lze dát název této funkce.

V takovém případě bude celkovým vyhodnocením zřetězení literálu „'INSERT INTO '“ a odkazu na proměnnou z ukázky 3.4 gramaticky platný PL/SQL výraz

```
'INSERT INTO get_audit_table'
```

Algoritmus detekce selhání by měl správně detekovat, zda se odkaz na proměnnou vyskytuje v názvu tabulky, schématu, nebo sloupce.

3.1.3 Výskyt vedle klíčového slova.

Podíváme se na část příkazu z ukázky 3.1 (metoda **create_audit_table_copy**).

```
-- ...
audit_table_name := get_audit_table(p_schema_name, p_table_name);
EXECUTE IMMEDIATE 'INSERT INTO copy_' || audit_table_name
  || '_t SELECT * FROM ' || audit_table_name
  || ' WHERE ' || p_condition;
```

Ukázka kódu 3.4: Příklad výskytu odkazu vedle klíčového slova.

Jak již bylo dříve zmíněno v předchozí sekci 3.1.2, příkaz `INSERT` má složitou syntaxi, a proto v ukázce 3.1 má příkaz `INSERT` jiný tvar:

```
INSERT INTO table_1 SELECT * FROM table_2 WHERE condition;
```

V dynamickém výrazu se vedle klíčových slov vyskytují druhý odkaz na proměnnou **audit_table_name** a odkaz na proměnnou **p_condition**. Zároveň se první odkaz na proměnnou **audit_table_name** a druhý odkaz na proměnnou **audit_table_name** vyskytují v názvech tabulek, což bylo popsáno v sekci 3.1.2.

V procesu vyhodnocení není hodnota odkazu na proměnnou **p_condition** známa. Lze ovšem říci, že hodnota výrazu, obsaženého v `condition` z předchozího výrazu má typ `Boolean` a zároveň klauzule `WHERE` může mít složitou strukturu [25]. Proto by při vyhodnocení odkazu na proměnnou strategie mohla vrátit původní text odkazu.

Algoritmus detekce selhání by měl správně detekovat, zda se odkaz na proměnnou vyskytuje vedle klíčového slova.

3.2 Popis algoritmu vyhodnocení odkazů s detekcí selhání

Celý algoritmus vyhodnocení odkazů s detekcí selhání se skládá ze dvou částí. V první části pomocí prohlížení stromu dostaneme informaci, o jaký druh výskytu se jedná.

V prvním kroku algoritmus zkusí najít výskyt odkazu na proměnnou v seznamu parametrů funkce. O tomto druhu výskytu je popsáno v sekci 3.1.1. Nechť potřebujeme vyhodnotit část DV, která je částí kódu z ukázky 3.1.

```
'to_date(''' || to_char(SYSDATE, 'DD.MM.YYYY HH24:MI:SS')
  || ''',''DD.MM.YYYY HH24:MI:SS'');
```

Zkusíme najít, jestli vlevo od uzlu je zřetězení s jiným uzlem, který má ve svém vyhodnocení na konci řetězec, který je začátkem seznamu parametrů funkce.

Pro případ předchozího výrazu řetězcem, který je začátkem seznamu parametrů funkce, je řetězec „'to_date(""“. Pokud takové zřetězení existuje, hlídáme vpravo od uzlu zřetězení s jiným uzlem, který má ve svém vyhodnocení na začátku řetězec, který je na konci seznamu parametrů funkce. Pro případ předchozího výrazu řetězcem, který je na konci seznamu parametrů funkce, je řetězec „"','DD.MM.YYYY HH24:MI:SS'')'“.

V následujícím kroku algoritmus zkusí najít výskyt odkazu na proměnnou v názvu tabulky, schématu, nebo sloupce. O tomto druhu výskytu je popsáno v sekci 3.1.2. Zároveň jde o nalezení výskytu odkazu vedle klíčového slova (KS). O tomto druhu výskytu je popsáno v sekci 3.1.3. Zkusíme najít, jestli vlevo nebo vpravo od uzlu je zřetězení s jiným uzlem, který má ve svém vyhodnocení *omezující klíčové slovo*. Za omezující KS považujeme takové KS, které vyžaduje, aby vlevo nebo vpravo od něj přes mezeru následoval běžný uživatelský identifikátor[†] [26, kapitola Lexical Units, sekce Identifiers]. Pro případ nalezení z levé strany omezující KS musí být na konci vyhodnocení. Pro případ nalezení z pravé strany omezující KS musí být na začátku vyhodnocení. Nechť potřebujeme vyhodnotit část DV, která je částí kódu z ukázky 3.1.

```
audit_table_name := get_audit_table(p_schema_name, p_table_name);
EXECUTE IMMEDIATE ' SELECT * FROM ' || audit_table_name ||
  ' WHERE ' || p_condition;
```

Jak bylo zmíněno v sekci 3.1.3, vyhodnocením odkazu na proměnnou **p_condition** může být původní text odkazu, protože pro KS `WHERE` není nutné, aby vpravo následoval běžný uživatelský identifikátor. Vyhodnocením odkazu na proměnnou **audit_table_name** nemůže být původní text odkazu, protože pro KS `FROM` je potřeba aby vpravo následoval běžný uživatelský identifikátor [27, Sekce FROM Clause]. Předpokládáme, že vyhodnocením odkazu na proměnnou **audit_table_name** musí být název tabulky. Proto tímto vyhodnocením bude název přiražené k proměnné funkce **get_audit_table**.

V posledním kroku algoritmus zkusí najít výskyt odkazu na proměnnou v názvu tabulky, schématu, nebo sloupce jiným způsobem. Nechť potřebujeme vyhodnotit část DV, která je částí kódu z ukázky 3.1.

[†]Ordinary User-Defined Identifier

```
'INSERT INTO copy_' || audit_table_name || '_t SELECT'
```

Vlevo a vpravo od odkazu na proměnnou **audit_table_name** jsou dva literály. Literál vlevo obsahuje KS `INTO`. Literál vpravo obsahuje KS `SELECT`. Za KS `INTO` musí nasledovat bežný užívatelský identifikátor. V našem případu identifikátor se skladá ze tří části. Konec literálu vlevo obsahuje řetězec „copy_“. Další je odkaz na proměnnou **audit_table_name**. Začátek literálu vpravo obsahuje řetězec „_t“.

Zkusíme najít, jestli se vlevo nebo vpravo od uzlu nachází zřetězení s jiným uzlem, který je problematický. Při nalezení takového zřetězení strategie selže. Za problematický uzel vlevo považujme literál, jehož obsah končí jedním z následujících symbolů:

- symbol abecedy,
- číslice,
- speciální symboly _, #, $, '.

Za problematický uzel z pravé strany považujeme literál, jehož obsah začíná jedním ze symbolů ze seznamu výše. Pro vyhodnocení odkazu na proměnnou **audit_table_name** literál „'INSERT INTO copy_'“ je problematickým uzlem vlevo a literál „'_t SELECT'“ je problematickým uzlem vpravo.

V druhé části probíhá rozhodnutí o navrácení původního textu odkazu, nebo vyhledání jména funkce, nebo spuštění fallback strategie. Tato část je popsána v algoritmu 1.

Vstup: zjištěná informace z části 1 algoritmu.
Výstup: vyhodnocení odkazu

```
1: inicializace
2: Set values = prázdný seznam vyhodnocení odkazu
3: if omezující klícové je nalezeno then
4:     go to vyhledání jména funkce
5: else
6:     if nalezen začátek a konec parametr then
7:         Add upravené vyhodnocení odkazu to values
8:         go to vyhledání jména funkce
9:     else
10:        if nalezen vadící literál then
11:            go to vyhledání jména funkce
12:        else
13:            Add vyhodnocení odkazu to values
14:        end if
15:    end if
16: end if
17: vyhledání jména funkce:
18: if jméno je nalezeno then
19:    Add jméno funkce to values
20: else
21:    spustit Identifier strategii jako fallback
22:    Add vyhodnocení spuštěné strategie to values
23: end if
24: return values
```

Algoritmus 1: Vracení výsledku vyhodnocení na základě rozhodnutí

Kapitola 4

Implementace

V této kapitole v první sekci jsou popsány použité nástroje při implementaci. V druhé sekci jsou uváděny její detaily.

4.1 Použité nástroje

Pro vývoj a testování strategií byly použity následující nástroje:

- Eclipse [28]. Vývojové prostředí.
- Java [29]. Byla použita verze 1.7.
- JUnit [30]. Knihovna pro testování. Byla použita verze 4.
- Apache Maven [31]. Nástroj pro sestavování projektu. Jednou z funkčností nástroje je řízení závislosti.
- Subversion (SVN) [32]. Verzovací systém.

Vzhledem k použitému nástroji pro sestavování projektu, celý projekt MF je rozdělen do logických bloků tzv. *artefaktů*. V této práci se všechny změny nacházejí v artefaktu `manta-connector-oracle-resolver` projektu MF. Seznam změn ve zveřejněné části zdrojového kódu je popsán v sekci 4.2.10.

4.2 Implementace strategií

4.2.1 Rozhraní EvaluationStrategy

Všechny třídy strategií implementují rozhraní[†] `EvaluationStrategy`. Tento interface obsahuje metodu `evaluateReference`, která slouží pro vyhodnocení odkazu a vrátí množinu reprezentací (částečně) vyhodnocené hodnoty proměnné. O reprezentaci vyhodnocení bude pojednávat 4.2.2. Podrobný význam argumentů této metody:

- `referenceNode` je uzel odkazu.
- `dynamicSqlService` je služba pro zpracování dynamického SQL.
- `variableAssignments` jsou hodnoty proměnných v místě reference.
- `prevReferenceNodes` je seznam uzlů odkazu, slouží pro zachování informaci o dřív procházených uzlech. Detailní pojednání o použití tohoto seznamu se nachází v sekci 4.2.7.

```
public interface EvaluationStrategy<N extends IMantaAstNode> {
  public Set<VariableValue<N>> evaluateReference(
    IAstCommonReference<N, ?> referenceNode,
    LinkedList<IAstCommonReference<N, ?>> prevReferenceNodes,
    AbstractDynamicSqlService<N> dynamicSqlService,
    Assignments<N> variableAssignments);
}
```

Ukázka kódu 4.1: Interface EvaluationStrategy

4.2.2 Třída VariableValue

Třída `VariableValue` slouží k reprezentaci jedné (částečně) vyhodnocené hodnoty proměnné. Důležitým atributem této třídy je atribut `String valueText`. Atribut reprezentuje buď samotnou hodnotu proměnné, nebo částečně vyhodnocenou hodnotu s některými částmi ponechanými jako výraz nad operandy. Množina operandů, jejichž hodnotu se nepodařilo přímo zakomponovat do hodnoty proměnné a musí být při použití hodnoty navázány jako vstupní parametry, je reprezentována atributem `Set<IResObject<N>> operands`.

Zjednodušená definice třídy `VariableValue` je v ukázce 4.2.

[†] interface

```
public class VariableValue<N extends IMantaAstNode> {
    final String valueText;
    final Set<IResObject<N>> operands;

    public VariableValue(String valueText, Set<IResObject<N>> operands) {
        this.valueText = valueText;
        this.operands = operands;
    }
}
```

Ukázka kódu 4.2: Třída VariableValue.

4.2.3 Třída ValueStrategy

Třída `ValueStrategy` implementuje rozhrání `EvaluationStrategy` následujícím způsobem. Na začátku pomocí volání metody `isTrueVariable` zjistí, zda jde o proměnnou nebo položku, které je přiražen nějaký výraz. Pokud ano, vratí vyhodnocení výrazu. Jinak se spustí Name strategie (atribut `failureFallback`) vyhodnocení výrazu.

```
public class ValueStrategy<N extends IMantaAstNode>
  implements EvaluationStrategy<N> {
  private EvaluationStrategy<N> failureFallback =
    new NameStrategy<>();

  public Set<VariableValue<N>> evaluateReference(
    IAstCommonReference<N, ?> referenceNode,
    LinkedList<IAstCommonReference<N, ?>> prevReferenceNodes,
    AbstractDynamicSqlService<N> dynamicSqlService,
    Assignments<N> variableAssignments) {

    if (dynamicSqlService.isTrueVariable(referenceNode)) {
      IResEntity<N> refObj = referenceNode.getReferencedObject();
      Set<AssignmentContext<N>> possibleExprs =
        variableAssignments.getAssigmentsFor(refObj);

      if (CollectionUtils.isNotEmpty(possibleExpressions)) {
        prevReferenceNodes.addFirst(referenceNode);
          Set<VariableValue<N>> values = new LinkedHashSet<>();
          for (AssignmentContext<N> expr : possibleExprs) {
            if (expr.getLhs() == refObj) {
              // ordinary assignment
              values.addAll(
                dynamicSqlService.evaluateExpression(
                  expr.getExpressionContext(),
                  prevReferenceNodes, this));
            } else {
              // non-atomic
              // vratit log
            }
```

```
          }
          prevReferenceNodes.removeFirst();
          return values;
        } else {
          // vratit log
        }
      } else {
        // vratit log
      }

      return failureFallback.evaluateReference(
        referenceNode,
        prevReferenceNodes,
        dynamicSqlService,
        variableAssignments);
    }
}
```

Ukázka kódu 4.3: Třída ValueStrategy.

4.2.4 Třída NameStrategy

Třída `NameStrategy` implementuje rozhraní `EvaluationStrategy` a implementuje algoritmus detekce selhání popsaný v kapitole 3. Labelu „vyhledávání jména funkce“ slouží proměnná `searchFunctionName` typu `Boolean`. První podmínka algoritmu „omezující klícové slovo je nalezeno“ je implementovana v metodě `findSiblingKeywordConcatenation` třídy `OracleDynamicSqlService`. Další podmínka algoritmu „nalezen začátek a konec parametr“ je implementována v metodě `findSiblingApostropheConcatenation`. Poslední podmínka „nalezen problémový literal“ je implementována v metodě `findSiblingNameConcatenation`. Metoda `getEscapedNormalizedText` navrátí upravené vyhodnocení odkazu. Třída `OracleDynamicSqlService` bude detailně popsána v sekci 4.2.7.

```
public class NameStrategy<N extends IMantaAstNode>
  implements EvaluationStrategy<N> {
  private EvaluationStrategy<N> failureFallback =
    new NameStrategy<>();

  public Set<VariableValue<N>> evaluateReference(
    IAstCommonReference<N, ?> referenceNode,
    LinkedList<IAstCommonReference<N, ?>> prevReferenceNodes,
    AbstractDynamicSqlService<N> dynamicSqlService,
    Assignments<N> variableAssignments) {

    HashSet<VariableValue<N>> variableValues =
      new HashSet<VariableValue<N>>();
    Set<IResObject<N>> refs =
      dynamicSqlService.collectExpressionReferences(
```

```
        referenceNode);

    boolean searchFunctionName = dynamicSqlService.
      findSiblingKeywordConcatenation(
        referenceNode,
        prevReferenceNodes);

    if (!searchFunctionName) {
      boolean escapeNodeValue = dynamicSqlService.
        findSiblingApostropheConcatenation(
          referenceNode,
          prevReferenceNodes);

      if (escapeNodeValue) {
        String escapedNodeValue = dynamicSqlService.
          getEscapedNormalizedText(referenceNode);
        variableValues.add(new VariableValue<>(
          escapedNodeValue, refs));

        searchFunctionName = true;
      } else {
        searchFunctionName = dynamicSqlService.
          findSiblingNameConcatenation(
            referenceNode,
            prevReferenceNodes);
      }
    }

    if (searchFunctionName) {
      String functionName = dynamicSqlService.
        getFunctionName(referenceNode);
      if (functionName != null) {
        variableValues.add(new VariableValue<>(
          functionName, refs));
      } else {
        Set<VariableValue<N>> vals = failureFallback.
          evaluateReference(
            referenceNode,
            prevReferenceNodes,
            dynamicSqlService,
            variableAssignments);
        variableValues.addAll(vals);
      }
      return variableValues;
    }

    return Collections.singleton(
      new VariableValue<>(
        referenceNode.toNormalizedString(), refs));
}
```

Ukázka kódu 4.4: Třída NameStrategy.

4.2.5 Třída IdentifierStrategy

Třída `IdentifierStrategy` implementuje rozhraní `EvaluationStrategy`. Původní text odkazu je upraven pomocí volání metody `replaceAll`. Tato třída v rámci práce nebyla upravena.

```
public class IdentifierStrategy<N extends IMantaAstNode>
  implements EvaluationStrategy<N> {
  private EvaluationStrategy<N> failureFallback =
    new NameStrategy<>();

  public Set<VariableValue<N>> evaluateReference(
    IAstCommonReference<N, ?> referenceNode,
    LinkedList<IAstCommonReference<N, ?>> prevReferenceNodes,
    AbstractDynamicSqlService<N> dynamicSqlService,
    Assignments<N> variableAssignments) {

    return Collections.singleton(
      new VariableValue<>(
        referenceNode.toNormalizedString()
          .replaceAll("[^a-zA-Z0-9]", "_"),
        dynamicSqlService.collectExpressionReferences(
          referenceNode)));
}
```

Ukázka kódu 4.5: Třída IdentifierStrategy.

4.2.6 Enumerator KeywordName

Enumerator obsahuje klíčová slova (KS) a atribut `ForbidGroup`, který definuje, jestli KS je omezující ve zvoleném směru vyhledávání. Element enumeratoru obsahuje název KS a skupinu, do které to KS patří:

```
Element_Name("Keyword", ForbidGroup.value)
```

Atribut `ForbidGroup` může mít následující hodnoty:

- `ForbidLeft`. Pokud vyhodnocovací proměnná je vlevo od literálu s KS, KS je omezující.
- `ForbidRight`. Pokud vyhodnocovací proměnná je vpravo od literálu s KS, KS je omezující.
- `ForbidBoth`. KS je vždy omezující.
- `ForbidNone`. KS není omezující.

Příkladem je prvek **WHERE** enumeratoru, který ma omezující KS „Where“, pokud vyhodnocovací proměnná je vlevo od literálu s tímto KS:

```
WHERE("Where", ForbidGroup.ForbidLeft).
```

Všechny prvky enumeratoru jsou popsány v tabulce A.1.

4.2.7 Třída OracleDynamicSqlService

Třída `OracleDynamicSqlService` je službou pro vyhodnocování dynamického SQL v Oracle. Tato třída dědí od abstraktní třídy `AbstractDynamicSqlService`, která je základem služby pro vyhodnocování dynamického SQL v nějakém jazyce. Popíšeme některé zajímavé části implementace této služby.

Metoda `isTrueVariable` byla stručně popsána v části 4.2.3.

Metoda `findSiblingConcatenation` z ukázky 4.6 slouží pro ověření:

1. zda uzel `referenceNode`, reprezentující odkaz na proměnnou, je v podstromu, který reprezentuje zřetězení odkazu na proměnnou s literálem;
2. zda obsah uzlu, reprezentující literál, splňuje zvolenou podmínku algoritmu 1 ze sekce 3.2.

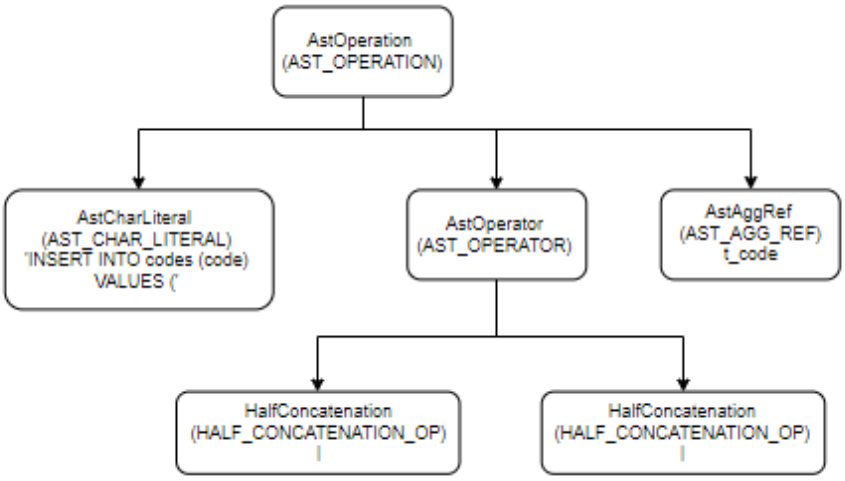
Příkladem reprezentace zřetězení odkazu na proměnnou s literálem vlevo je podstrom na obrázku 4.1 a jeho jiná varianta se nachází na obrázku 4.2.

Uzel `referenceNode` reprezentuje odkaz na proměnnou **t_code**. Podstrom reprezentuje zřetězení odkazu na proměnnou **t_code** s literálem, který má v obsahu řetězec „'INSERT INTO codes (code) VALUES ('“.

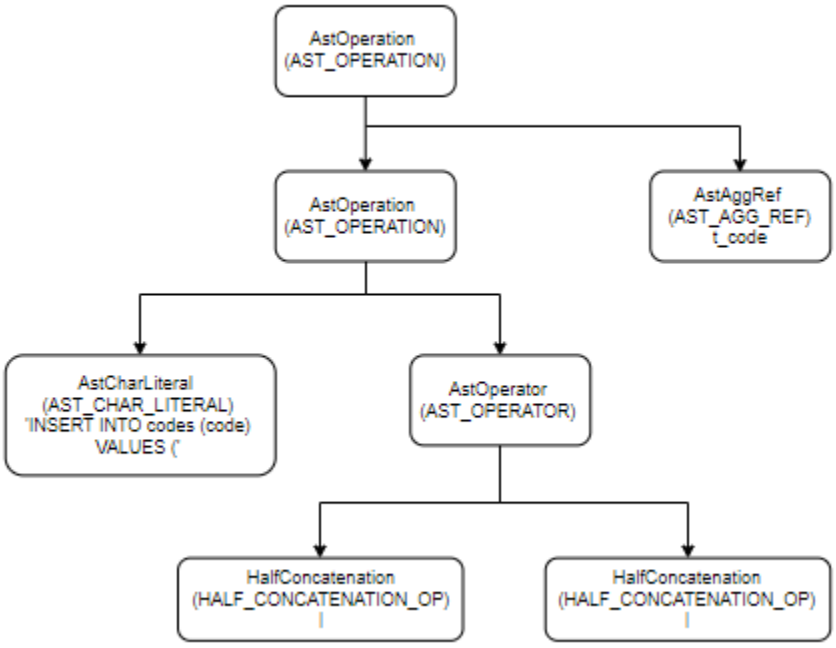
Příkladem reprezentace zřetězení odkazu na proměnnou s literálem vpravo je podstrom z obrázku 4.3

Uzel `referenceNode` reprezentuje odkaz na proměnnou **t_code**. Podstrom reprezentuje zřetězení odkazu na proměnnou **t_code** s literálem, který má v obsahu řetězec „')'“.

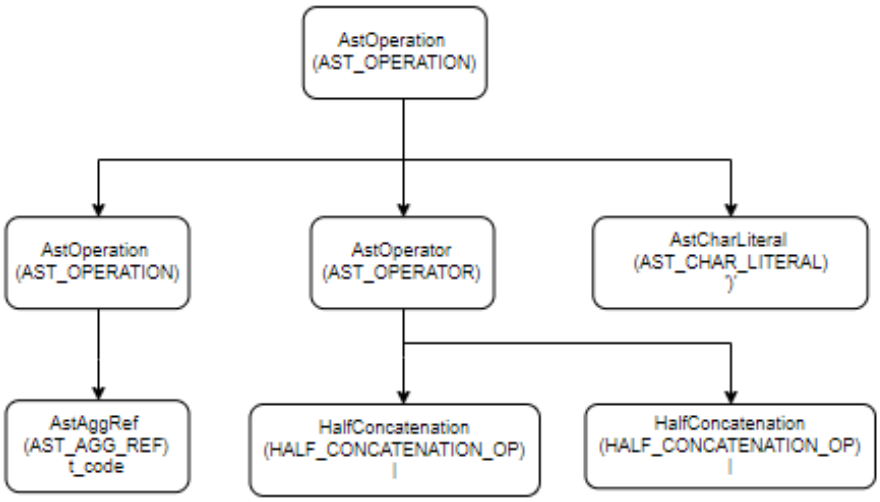
V této verzi metody stačí, aby podmínka byla splněna v nalezeném literálu zleva nebo v nalezeném literálu zprava. Takže existuje metoda `findSiblingConcatenationAnd`, kde podmínka musí být splněna zleva a zprava. Pokud zřetězení s literálem vlevo nebylo nalezeno, hodnota `lFound` je *null*. Respektive, pokud zřetězení s literálem vpravo nebylo nalezeno, hodnota `rFound` je *null*. V tomto



Obrázek 4.1: 1. příklad reprezentace zřetězení odkazu na proměnnou s literálem vlevo v MF.



Obrázek 4.2: 2. příklad reprezentace zřetězení odkazu na proměnnou s literálem vlevo v MF.



Obrázek 4.3: Příklad reprezentace zřetězení odkazu na proměnnou s literálem vpravo v MF.

případě se začíná prohlížet seznam `prevReferenceNodes`. Protože máme informace o dříve procházených uzlech, lze zjistit informaci, zda jde vůbec o zřetězení. Pokud ano, zkontrolujeme podmínku v prvním z prohlížených uzlů, kde je nějaké zřetězení.

```
private boolean findSiblingConcatenation(
  IAstCommonReference<IOracleAstNode, ?> referenceNode,
  LinkedList<IAstCommonReference<IOracleAstNode, ?>> prevReferenceNodes,
  IConcatenationProcessing concatenationProcessing) {
    Boolean lFound = null, rFound = null;
    lFound = concatenationProcessing.leftSibling(referenceNode);
    if (!Boolean.TRUE.equals(lFound)) {
      rFound = concatenationProcessing.rightSibling(referenceNode);
      if (!Boolean.TRUE.equals(rFound)) {
        for (IAstCommonReference<IOracleAstNode, ?> node :
          prevReferenceNodes) {
          lFound = concatenationProcessing.leftSibling(node);
            if (Boolean.TRUE.equals(lFound)) {
              break;
            }
          rFound = concatenationProcessing.rightSibling(node);
            if (Boolean.TRUE.equals(rFound)) {
              break;
            }
        }
      }
```

```
    }
    return Boolean.TRUE.equals(lFound) || Boolean.TRUE.equals(rFound);
}
```

Ukázka kódu 4.6: Metoda findSiblingConcatenation třídy OracleDynamicSqlService.

Ukázka vyhodnocení s použitím metody findSiblingConcatenation

Metoda `findSiblingConcatenation` je interně použita v implementaci dvou metod: `findSiblingKeywordConcatenation` a `findSiblingNameConcatenation` třídy `OracleDynamicSqlService`. Metoda `findSiblingConcatenationAnd` se zavolá z metody `findSiblingApostropheConcatenation`. O těchto třech metodách bylo stručně pojednáno v sekci 4.2.4. Ukážeme si na příkladu fragmentu kódu z ukázky 4.7, jak funguje prakticky vyhodnocení a prohlížení seznamu `prevReferenceNodes`.

1. Probíhá vyhodnocení proměnné **sql_stmt** pomocí Value stategie.
2. Přidáme uzel **sql_stmt** do seznamu `prevReferenceNodes`.
3. Probíhá vyhodnocení každé části přiřazeného výrazu k proměnné **sql_stmt** pomocí Value stategie.
4. Probíhá vyhodnocení proměnné **c_name** pomocí Value stategie.
5. Přidáme uzel **c_name** do seznamu uzlů.
6. Probíhá vyhodnocení každé části přiřazeného výrazu k proměnné **c_name** pomocí Value stategie.
7. Probíhá vyhodnocení proměnné **get_data_columns** pomocí Value stategie.
8. Protože volání metody `isTrueVariable` vrátí *false*, spustí se fallback Name strategie.
9. Probíhá vyhodnocení proměnné **get_data_columns** pomocí Name stategie.
10. Podle algoritmu detekce selhání z kapitoly 3.2 a její implementace se zavolá metoda `findSiblingKeywordConcatenation`.
11. Interně se zavolá metoda `findSiblingConcatenation`.

12. Hodnota proměnné **lFound** není *null*, protože máme zřetězení vlevo s literálem „'id, '“. Zároveň hodnota proměnné **lFound** není *true*, protože vlevo není žádné klíčové slovo. Proto proběhne kontrola pravé strany.
13. Hodnota proměnné **rFound** je *null*, protože nemáme zřetězení vpravo. Proto proběhne další cyklus pro seznam dříve prohlížených uzlů. Seznam `prevReferenceNodes` obsahuje následující uzly podle pořadí vkládání: **sql_stmt**, **c_name**.
14. Pro uzel **c_name** hodnota proměnné **rFound** není *null*, protože máme zřetězení vpravo s literálem „' from h_'“. Zároveň hodnota proměnné **rFound** je *true*, protože vpravo je klíčové slovo.
15. Analogicky proběhnou kontroly pomocí ostatních metod, implementujících algoritmus detekce selhání.
16. Když Name strategie vrátí výsledek vyhodnocení uzlu **get_data_columns**, proběhne smazání uzlu **c_name** ze seznamu.
17. Analogicky proběhne vyhodnocení **t_name** pomocí Value a fallback strategií.
18. Když Value strategie vrátí výsledek vyhodnocení uzlu **sql_stmt**, proběhne smazání uzlu **sql_stmt** ze seznamu.
19. Na konci posledního vyhodnocení dynamického výrazu seznam musí být prázdný, jinak se zavolá výjimka `IllegalArgumentException`.

```
c_name := 'id, ' || get_data_columns('cluster_4');
t_name := get_table_name('cluster_4');
sql_stmt := 'select ' || c_name || ' from h_' || t_name;
EXECUTE IMMEDIATE sql_stmt BULK COLLECT INTO res;
```

Ukázka kódu 4.7: Příklad použití seznamu uzlů odkazu prevReferenceNodes.

Rozhraní IConcatenationProcessing

Metoda `findSiblingConcatenation` interně volá jednu z implementací rozhraní `IConcatenationProcessing`, které se rozlišují v případě vyhledávání omezujícího klíčového slova, problematického uzlu, nebo řetězce, který je seznamem parametrů funkce – jedná se o případy algoritmu detekce selhání, který byl popsán v sekci 3.2.

Každá třída, která implementuje toto rozhraní, v implementaci metody `leftSibling` využívá následující logiku:

1. Zvolíme pattern regulárního výrazu, pomocí kterého bude zkontrolován obsah literálu, pokud je nalezen vlevo.

2. Zvolíme řetězec, který bude sloužit jako klíč pro ukládání výsledku vyhledávání v atributu uzlu. Důvodem je, že je možné při prohlížení seznamu `prevReferenceNodes` některé uzly nalézt několikrát.

3. Ověříme, že vlevo od uzlu je skutečně literál:

   a) Pokud sourozenec vlevo má token `AST_CHAR_LITERAL`,

   b) Pokud sourozenec vlevo má token `AST_OPERATION`, zkontrolujeme, zda se jedná o operaci zřetězení s literálem. Kontrola zřetězení s literálem je popsána v sekci 4.2.8.

4. Zkompilujeme RV s parametrem `Pattern.CASE_INSENSITIVE` a dostaneme `Matcher` [17]. Třída `Matcher` byla popsána v sekci 1.4.

5. Ověříme, že nalezené posloupnosti odpovídají dalším podmínkám:

   a) Vstupem pro instanci třídy `Matcher` je obsah literálu, opravený zvoleným způsobem.

   b) Pomocí volání metod třídy `Matcher`, které byly popsány v sekci 1.4, rozhodneme o správnosti podmínky algoritmu z levé strany.

   V sekci 4.2.9 je popsáno, které RV byly zvoleny, a jak byly ověřeny nalezené posloupnosti. Tyto RV jsou escapovány a jsou tudíž použitelné v Javě.

Každá třída, která implementuje rozhraní `IConcatenationProcessing` v implementaci metody `rightSibling`, využívá podobnou logiku:

1. Zvolíme pattern RV, pomocí kterého bude zkontrolován obsah literálu, pokud je nalezen vpravo.

2. Zvolíme řetězec, který bude sloužit jako klíč pro ukládání výsledku vyhledávání v atributu uzlu.

3. Ověříme, že prvek vpravo od uzlu je skutečné literál. Pokud vpravo od uzlu má prvek token `AST_OPERATION`, zkontrolujeme, zda jde o operaci zřetězení s literálem.

4. Zkompilujeme a ověříme, zda nalezené posloupnosti odpovídají dalším podmínkám. Rozdílem je, že rozhodneme o správností podmínky algoritmu z pravé strany.

4.2.8 Ověření operace zřetězení odkazu na proměnnou s literálem

V této části detailně popíšeme, jak probíhá ověření, zda skutečně jde o operaci zřetězení odkazu na proměnnou s literálem, v implementaci metod `leftSibling` a `rightSibling` v rozhraní `findSiblingConcatenation`.

Ověření operace zřetězení odkazu na proměnnou s literálem vlevo

Na obrázku 4.1 je zobrazen podstrom, který reprezentuje zřetězení odkazu na proměnnou s literálem vlevo.

Jak bylo zmíněno v sekci 1.3, v projektu MF pro navigaci v AST je možné použít volání metody `getLeftSibling` pro získání informací o levém sourozenci uzlu, reprezentující odkaz na proměnnou. Zkontrolujeme název tokenu pomocí volání metody `getTokenName`. Token reprezentující literál má název `AST_CHAR_LITERAL`.

Na obrázku 4.2 je zobrazen jiný podstrom, který také reprezentuje zřetězení odkazu na proměnnou s literálem vlevo.

Levým sourozencem uzlu, reprezentujícím odkaz na proměnnou, bude uzel s tokenem `AST_OPERATION`. Označíme nalezeného levého sourozence proměnnou **leftNode**. Zkontrolujeme, zda jde o operaci zřetězení odkazu na proměnnou s literálem pomocí XPATH dotazu:

```
leftNode.selectSingleNode(
  "AST_OPERATION/AST_OPERATOR/HALF_CONCATENATION_OP")
```

Výsledkem dotazu musí být nenulová hodnota. Dalším XPATH dotazem

```
leftNode.selectSingleNode("AST_CHAR_LITERAL")
```

dostaneme nenulovou hodnotu, pokud jde o operaci zřetězení s literálem.

Ověření operace zřetězení odkazu na proměnnou s literálem vpravo

Na obrázku 4.3 je zobrazen podstrom, který reprezentuje zřetězení odkazu na proměnnou s literálem vpravo.

Pomocí dvojnásobného volání metody `getParent` u uzlu reprezentujícího odkaz na proměnnou, se dostaneme k uzlu s tokenem `AST_OPERATION`. Označíme nalezený uzel proměnnou **gpNode**. Zkontrolujeme, zda jde o operaci zřetězení odkazu na proměnnou s literálem pomocí XPATH dotazu:

```
gpNode.selectSingleNode(
  "AST_OPERATION/AST_OPERATOR/HALF_CONCATENATION_OP")
```

Výsledkem dotazu musí být nenulová hodnota. Dalším XPATH dotazem

```
gpNode.selectSingleNode("AST_CHAR_LITERAL")
```

dostaneme nenulovou hodnotu, pokud jde o operaci zřetězení s literálem.

4.2.9 Použité RV a oveření nalezených výsledků

Vyhledávání problematického uzlu vlevo

Vstupem je původní obsah stromu bez bílých znaků před a za stromem. Byl použit následující RV:

```
.*[\\w\\$\\#\\.]{1}[\\']{1}$
```

Pokud celý vstup je shodou pro RV, výsledek je správný.

Vyhledávání problematického uzlu vpravo

Vstupem je původní obsah stromu bez bílých znaků před a za stromem. Byl použit následující RV:

```
^[\\']{1}[\\w\\$\\#\\.]{1}.*
```

Pokud začátek vstupu je shodou pro RV, výsledek je správný.

Vyhledávání klíčového slova vlevo

Vstupem je původní obsah stromu bez bílých znaků před a za stromem. Byl použit následující RV, klíčové slovo bude nalezeno ve skupině **kw**. Skupina

je část RV, která je uzavřená v závorkách, může být extrahována z výsledků úspěšné shody.

```
.*\\s{1,}(?<kw>\\w*?)\\s{1,}[\\']{1}$
```

Podmínkou správností výsledku je skutečnost, zda nalezené KS je elementem enumeratoru `KeywordName` a zároveň tento element má hodnotu atributu `ForbidGroup` rovnou `ForbidRight` nebo `ForbidBoth`.

Vyhledávání klíčového slova vpravo

Vstupem je analogicky upraveny obsah jak při vyhledávání KS vlevo. Byl použit následující RV, klíčové slovo bude nalezeno ve skupině **kw**:

```
^[\\']{1}\\s{1,}(?<kw>\\w*?)\\s{1,}.*?
```

Podmínkou správností výsledku je skutečnost, zda nalezené KS je elementem enumeratoru `KeywordName` a zároveň tento element má hodnotu atributu `ForbidGroup` rovnou `ForbidLeft` nebo `ForbidBoth`.

Vyhledávání seznamu parametrů vlevo

Vstupem je původní obsah stromu bez bílých znaků před a za stromem a bez symbolu přenosu řetězce a symbolu návratu do začátku. Byl použit následující RV:

```
.*(?<par>[(]\\s*?[\\']{2})
  (?!.*?[\\']{2}\\s*?[)])(?<any>.*?)[\\']{1}$
```

Pokud celý vstup je shodou pro RV, výsledek je správný.

Vyhledávání seznamu parametrů vpravo

Vstupem je analogicky upraveny obsah jak při vyhledávání seznamu parametrů vlevo. Byl použit následující RV:

```
^[\\']{1}(?<any>.*?)(?<par>[\\']{2}.*?[)]).*
```

Pokud symbol „(“ není v obsahu skupiny **any** ani v obsahu skupiny **par**, výsledek je správný.

4.2.10 Seznam změn ve zdrojovém kódu

Tato sekce obsahuje informace o změnách ve zveřejněné části zdrojového kódu produktu Manta Flow v průběhu psaní této práci.

1. Bylo změněno rozhraní `EvaluationStrategy` v metodě `evaluateReference`.
2. Byla opravena metoda `evaluateReference` třídy `NameStrategy` na řádcích 38–66.
3. Byla opravena třída `OracleDynamicSqlService`:
    - Byla přídána rozhraní `IConcatenationProcessing`, `ITestNode` a `ITestContent` na řádcích 39–68.
    - Byly přídány metody na řádcích 445–955.

Kapitola 5

Testování

V této kapitole je popsáno provedené testování skriptů. Testovací skripty jsou rozděleny do následujících skupin:

1. seznam parametrů funkce v DV je rozdělen proměnnou;
2. abecední symbol nebo číslice v DV je na konci, nebo začátku literálu, bez mezer;
3. klíčové slovo v DV je na konci, nebo začátku literálu;
4. nebylo nalezeno možné selhání.

Pro všechny skripty byla použita Value strategie s upravenou fallback Name strategií.

5.1 1. testovací skupina

Příklad z ukázky 2.6. Výsledkem jsou dvě vyhodnocení: Seznam parametrů funkce **to_date** je rozdělen, proto k původnímu textu odkazu byly přidány apostrofy.

```
INSERT INTO schema_name.Htable_name
  (id, updated, operation_type)
  VALUES ( entity_id,to_date(''
  ||to_char(SYSDATE, 'DD.MM.YYYY HH24:MI:SS')
  ||'','DD.MM.YYYY HH24:MI:SS'),'Delete')
```

Jako náhradní variantu vrátíme jméno funkce. V tomto případě to je méně příjemná varianta.

```
INSERT INTO schema_name.Htable_name
  (id, updated, operation_type)
  VALUES ( entity_id,to_date('TO_CHAR',
  'DD.MM.YYYY HH24:MI:SS'),'Delete')
```

5.2 2. testovací skupina

Příklad z ukázky 5.1.

```
CREATE OR REPLACE PACKAGE BODY name_demo IS
  FUNCTION RETURN_C RETURN VARCHAR2 AS
  BEGIN
    RETURN 'c';
  END;

  PROCEDURE CALL_SELECT(P VARCHAR2) AS
    S1 VARCHAR2(50);
    S2 VARCHAR2(50);
  BEGIN
    S2 := name_demo.RETURN_C();
    EXECUTE IMMEDIATE 'select c.name from manta.'
      || 'countries t_' || S2 || ' where t_'
      || S2 || '.code = code2 and rownum = 1'
    INTO S1 ;
  END;
END name_demo;
```

Ukázka kódu 5.1: Příklad skriptu z 2. testovací skupiny.

První a druhý odkaz na proměnnou **s2** mají zřetězení s literálem, který se ukončí písmenem "_", který patří do symbolů ze seznamu v kapitole Návrh. Proto detekce klíčového slova `WHERE` neproběhne. Přesto odkazy na stejnou proměnnou považujeme za dva různé.

Výsledkem je následující vyhodnocení:

```
select c.name from manta.countries t_RETURN_C
  where t_RETURN_C.code = code2 and rownum = 1
```

5.3 3. a 4. testovací skupiny

Příklad z ukázky 5.2.

V prvním literálu bylo nalezeno klíčové slovo `TABLESPACE`, proto vyhodnocením odkazu na proměnnou **v_tspace_name** byl název nalezené funkce **get_param**.

```
CREATE PROCEDURE RFR
IS
  v_start_time  DATE := SYSDATE;
  v_tspace_name VARCHAR2(100);
  v_day_count NUMBER;
  v_start_date_text VARCHAR2(30);
BEGIN
  v_start_date_text := pkg_std.get_param('P_START_DATE');
  v_tspace_name := pkg_std.get_param('P_TABLESPACE');
  v_day_count := TRUNC(SYSDATE) - TO_DATE(
  v_data_start_date_text,'DD-MON-YYYY') ;

  EXECUTE IMMEDIATE 'CREATE TABLE t_dates
    TABLESPACE ' || v_tspace_name || '
    AS
    SELECT cal_date,
      TRUNC(cal_date,''MM'') begin_date,
      LAST_DAY(cal_date) end_date
    FROM (SELECT TRUNC(SYSDATE - ROWNUM) cal_date
      FROM DUAL CONNECT BY
      ROWNUM < (1 + ' || TO_CHAR(v_day_count) || '))';
END;
```

Ukázka kódu 5.2: Příklad skriptu z 3. a 4. testovacích skupin.

Při vyhodnocení odkazu na proměnnou **TO_CHAR** v literálu zleva na konci ani na začátku literálu zprava nebyl nalezen symbol ze seznamu v kapitole Návrh. Proto byl vrácen původní text odkazu.

Výsledkem je následující vyhodnocení:

```
CREATE TABLE t_dates
  TABLESPACE GET_PARAM
  AS
  SELECT cal_date,
    TRUNC(cal_date,'MM') begin_date,
    LAST_DAY(cal_date) end_date
    FROM (SELECT TRUNC(SYSDATE - ROWNUM) cal_date
      FROM DUAL CONNECT BY
      ROWNUM < (1 + TO_CHAR(v_day_count)))
```

5.4 Další testovací skripty

Příklad z ukázky 5.3.

V prvním výrazu při vyhodnocení odkazu na proměnnou **TRIM** v literálu vpravo byl nalezen symbol ze seznamu v kapitole Návrh. Proto byl vrácen původní text odkazu.

```
CREATE PROCEDURE insert_code (code VARCHAR2)
IS
  t_code VARCHAR2;
BEGIN
  t_code := trim(code) || 'x';
  EXECUTE IMMEDIATE 'INSERT INTO codes (code) VALUES ('
    || t_code || ')';
  t_code := 'x' || trim(code) ;
  EXECUTE IMMEDIATE 'INSERT INTO codes (code) VALUES ('
    || t_code || ')';
END;
```

Ukázka kódu 5.3: 1. příklad testovacího skriptu.

Výsledkem je následující vyhodnocení:

```
INSERT INTO codes (code) VALUES (TRIMx)
```

V druhém výrazu při vyhodnocení odkazu na proměnnou **TRIM** v literálu vlevo byl nalezen symbol ze seznamu v kapitole Návrh. Proto byl vrácen původní text odkazu. Výsledkem je následující vyhodnocení:

```
INSERT INTO codes (code) VALUES (xTRIM)
```

Posledním příkladem z testů je příklad z ukázky 5.4. Odkaz na proměnnou **TO_NUMBER** je v seznamu parametrů.

```
CREATE OR REPLACE PROCEDURE insert_into_t2 (x VARCHAR2) AS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO t2' || ' SELECT a, b FROM t1'
    || ' WHERE TO_NUMBER(c) = dbms_random.value(TO_NUMBER(''1''),'
    || (TO_NUMBER(x) + 1) || ')';
END;
```

Ukázka kódu 5.4: 2. příklad testovacího skriptu.

Výsledkem je následující vyhodnocení:

```
INSERT INTO t2 SELECT a, b FROM t1 WHERE TO_NUMBER(c) =
  dbms_random.value(TO_NUMBER('1'),TO_NUMBER(x) + 1)
```

Závěr

Tato práce se zabývala analýzou dynamického SQL kódu. Pro tuto analýzu již byly implementovány tři strategie - Value, Name a Identity, každá z nich byla stručně popsána, včetně jejich výhod a nevýhod použití při vyhodnocení výrazu. Bylo zjištěno, že není potřeba implementovat novou strategii. Řešením bylo upravit Name strategii, aby byl redukován počet selhání a přidat Identifier strategii jako fallback podobně jako u fallbacku Value strategie.

Byla přidána sada regulárních výrazů, pomocí kterých probíhá rozhodnutí o vrácení původního textu odkazu, vrácení nalezeného jména funkce v odkazu, spuštění fallback strategie, nebo vrácení dvou alternativních hodnocení - vrácení nalezeného jména funkce v odkazu, nebo vyhodnocení pomocí fallback strategie.

Další vývoj

Algoritmus, použitý v Name strategii může být zlepšen podporou vyhodnocení v případě zřetězení dvou odkazů na proměnnou, přičemž proměnné vpravo je přiražen nějaký literál. V současné době bude vždy výsledkem hledání jména funkce u přiražení proměnné vlevo, nebo použití fallback strategie. Příkladem je skript z ukázky 5.5 a jde o hodnocení proměnné **sql_values**.

```
CREATE OR REPLACE PROCEDURE insert_audit_delete (schema_name VARCHAR2,
    table_name VARCHAR2, entity_id NUMBER)
AS
    sql_values VARCHAR2 (1000);
    sql_values_dt VARCHAR2 (1000);
BEGIN
    sql_values_dt := ',to_date('''
      || to_char(SYSDATE, 'DD.MM.YYYY HH24:MI:SS')
      || ''','''DD.MM.YYYY HH24:MI:SS'''),''Delete''';
    sql_values := entity_id || sql_values_dt;

    EXECUTE IMMEDIATE 'insert into ' || schema_name || '.H' ||table_name
      || ' (id, updated, operation_type) values ( ' || sql_values || ')';

END;
```

Ukázka kódu 5.5: Příklad skriptu pro možné zlepšení vyhodnocení

Bibliografie

1. TECHNOPEDIA. *Technopedia: Data Lineage* [online] [cit. 2018-03-22]. Dostupné z: https://www.techopedia.com/definition/28040/data-lineage.
2. *"Manta Flow"* [online] [cit. 2018-03-22]. Dostupné z: https://getmanta.com/.
3. *"BI & DWH | Profinit"* [online] [cit. 2018-05-01]. Dostupné z: https://profinit.eu/sluzby/bi-dwh/.
4. *Oracle PL/SQL* [online] [cit. 2018-03-23]. Dostupné z: http://www.oracle.com/technetwork/database/features/plsql/index.html.
5. AHO, A. V.; LAM, M. S.; AJ. *Compilers: Principles, Techniques, and Tools*. 2. vyd. Addison Wesley, 2007. ISBN 0-321-48681-1.
6. *ANTLR* [online] [cit. 2018-04-01]. Dostupné z: http://www.antlr.org/.
7. *PL/SQL Dynamic SQL* [online] [cit. 2018-04-11]. Dostupné z: https://docs.oracle.com/cloud/latest/db112/LNPLS/dynamic.htm.
8. *Coding Dynamic SQL Statements* [online] [cit. 2018-04-11]. Dostupné z: https://docs.oracle.com/cd/A97630_01/appdev.920/a96590/adg09dyn.htm#26586.
9. *PL/SQL Language Elements* [online] [cit. 2018-04-15]. Dostupné z: https://intellipaat.com/tutorial/oracle-plsql-tutorial/plsql-language-elements/.
10. *execute_immediate_statement.gif* [online] [cit. 2018-05-02]. Dostupné z: https://docs.oracle.com/cloud/latest/db112/LNPLS/img/execute_immediate_statement.gif.

11. FEUERSTEIN, S.; PRIBYL, B. *Oracle PL/SQL Programming*. 6. vyd. O'Reilly, 2014. ISBN 978-1-44932445-2.
12. *The Definitive ANTLR Reference*. 1. vyd. The Pragmatic Bookshelf, 2007. ISBN 978-09787392-4-9.
13. *XPath 2.0 Reference* [online] [cit. 2018-04-01]. Dostupné z: `http://zvon.org/comp/r/ref-XPath_2.html`.
14. FRIEDL, J. *Mastering Regular Expressions*. 3. vyd. O'Reilly, 2006. ISBN 0-596-52812-4.
15. *Regulární výrazy – pokročilé podskupiny* [online] [cit. 2018-05-04]. Dostupné z: `http://www.pepak.net/programovani/regularni-vyrazy-4-pokrocile-podskupiny/`.
16. *Class Pattern (Java Platform 7 SE)* [online] [cit. 2018-04-25]. Dostupné z: `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`.
17. *Class Matcher (Java Platform 7 SE)* [online] [cit. 2018-04-25]. Dostupné z: `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html`.
18. *oracle_oci_examples* [online] [cit. 2018-05-03]. Dostupné z: `https://github.com/alexeyvo/oracle_oci_examples`.
19. *Oracle Database 12.2.0.1 Sample Schemas* [online] [cit. 2018-05-03]. Dostupné z: `https://github.com/oracle/db-sample-schemas/releases/tag/v12.2.0.1`.
20. *dw-vldb-samples* [online] [cit. 2018-05-03]. Dostupné z: `https://github.com/oracle/dw-vldb-samples`.
21. *Using Regular Expressions in Java* [online] [cit. 2018-04-21]. Dostupné z: `https://www.regular-expressions.info/java.html`.
22. STRNAD, J. *Inkrementální analýza PL/SQL skriptů* [online]. České vysoké učení technické v Praze, Fakulta informačních technologií, 2015 [cit. 2018-04-14]. Dostupné z: `https://dspace.cvut.cz/bitstream/handle/10467/62752/F8-DP-2015-Strnad-Jan-thesis.pdf`.
23. *INSERT | Database SQL Reference* [online] [cit. 2018-04-29]. Dostupné z: `https://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_9014.htm`.

24. *Schema Object Names and Qualifiers* [online] [cit. 2018-05-06]. Dostupné z: https://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements008.htm.

25. *Conditions | Database SQL Reference* [online] [cit. 2018-04-29]. Dostupné z: https://docs.oracle.com/cd/B19306_01/server.102/b14200/conditions.htm.

26. *PL/SQL Data Types* [online] [cit. 2018-04-15]. Dostupné z: https://docs.oracle.com/cloud/latest/db112/LNPLS/fundamentals.htm.

27. *Select | Database SQL Reference* [online] [cit. 2018-04-29]. Dostupné z: https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_10002.htm.

28. *Eclipse desktop & web IDE* [online] [cit. 2018-05-05]. Dostupné z: https://www.eclipse.org/ide/.

29. *Java SE 7 Archive Downloads* [online] [cit. 2018-05-05]. Dostupné z: http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html.

30. *JUnit 4* [online] [cit. 2018-05-05]. Dostupné z: https://junit.org/junit4/.

31. *Maven - Welcome to Apache Maven* [online] [cit. 2018-05-05]. Dostupné z: https://maven.apache.org/.

32. *Apache Subversion* [online] [cit. 2018-05-05]. Dostupné z: https://subversion.apache.org/.

Dodatek A

Seznam omezujících klíčových slov

V následující tabulce jsou uvádená klíčová slova, které jsou elementy enumeratoru `KeywordName`. V prvním sloupci je název KS, v druhém sloupci je hodnota atributu `ForbidGroup`.

Klíčové slovo	Forbid Group
AS	Both
BY	Right
ELSEIF	None
FROM	Both
INTO	Right
JOIN	Both
SELECT	None
TABLE	Both
TABLESPACE	Both
THEN	Right
WHERE	Left
WITH	None

Tabulka A.1: Omezující klíčová slova.

Dodatek **B**

Seznam použitých zkratek

AST - Abstract Syntax Tree

DV - Dynamický výraz

KS - Klíčové slovo

MF - Manta Flow

NDS - Native Dynamic SQL

PL/SQL - Procedural Language/Structured Query Language

RV - Regulární výraz

SQL - Structured Query Language

Dodatek C

Obsah přiloženého CD

- `src` ..... zdrojové kódy práce
  - `thesis` ..... zdrojová forma práce ve formátu XeLaTeX
    - `pics` ..... obrázky použité v této práci
    - `pdfs` ..... zadání a logo ČVUT ve formátu pdf
  - `mf` ..... zveřejněné části zdrojového kódu produktu Manta Flow
    - `OracleDynamicSqlService.java` .... třída OracleDynamicSqlService
    - `EvaluationStrategy.java` ..... rozhraní EvaluationStrategy
    - `IdentifierStrategy.java` ..... třída IdentifierStrategy
    - `NameStrategy.java` ..... třída NameStrategy
    - `ValueStrategy.java` ..... třída ValueStrategy
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF