České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

Inkrementální analýza PL/SQL skriptů

Bc. Jan Strnad

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

29. dubna 2015

Poděkování

Chtěl bych poděkovat týmu z Profinitu, především pak Mgr. Jiřímu Touškovi, za jejich ochotu a cenné rady. Dále bych chtěl poděkovat vedoucímu doc. Ing. Jan Janoušekovi, Ph.D. za přínosné rady při psaní textu této práce. V neposlední řadě bych rád poděkovat rodině a přítelkyni za jejich podporu.

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 29. dubna 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2015 Jan Strnad. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Strnad, Jan. *Inkrementální analýza PL/SQL skriptů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato práce se zabývá vylepšením mechanizmu transformací u systému Manta Tools. Transformace umožňuje automaticky upravit PL/SQL kód, reprezentovaný pomocí AST, tak, aby splňoval určitá pravidla. Typickým příkladem je převod jednoho syntaktického konstruktu na jiný, významově ekvivalentní. Sem lze zařadit např. funkci DECODE převedenou na CASE výraz.

V důsledku transformace dochází ke ztrátě informace obsažené v AST reprezentaci. Původní řešení doplnění této informace spočívalo v provedení úplné opětovné syntaktické a sémantické analýzy transformovaného skriptu. Tato práce analyzuje, navrhuje a implementuje inkrementální řešení. Implementované řešení nevyžaduje provedení jakýchkoliv změn na straně již existujících transformací. Efekt inkrementálního zpracování byl experimentálně změřen na několika typických transformacích. Bylo dosaženo velmi výrazného časového zlepšení, v některých případech až v řádu stonásobků.

Klíčová slova Procedural Language/Structured Query Language, syntaktická analýza, sémantická analýza, inkrementální, transformace

Abstract

This thesis deals with improvement of transformation mechanism in Manta Tools system. Transformation enables automatic modification of PL/SQL code, represented using an AST, such that it conforms to some rules. Typical example is conversion of one syntactical construct to another, semantically equal one. Conversion of DECODE function to CASE expression is a such example.

As a consequence of the transformation, there is a loss of information in the AST representation. Original solution used to recover this information consisted of complete syntactical and semantics reanalysis of the transformed script. This thesis analyses, designs and implements incremental solution. Implemented solution is completely transparent and does not require any additional changes of existing transformations. The effect of incremental processing was experimentally measured on several typical transformations. Very significant time improvement was achieved, in some cases up to hundreds of times.

Keywords Procedural Language/Structured Query Language, syntactic analysis, semantic analysis, incremental, transformation

Obsah

Úvod **1**

1 Základní pojmy **3**
1.1 Jazyk, gramatika 3
1.2 Syntaktická analýza 6
1.3 ANTLR 9
1.4 PL/SQL 12
1.5 Manta tools 16
1.6 Terminologie 22

2 Analýza **23**
2.1 Notace 23
2.2 Dekompozice problému 24
2.3 Detekce změn v AST 24
2.4 Určení stromů k opětovné analýze 29
2.5 Opětovná analýza podstromu 33
2.6 Oprava pozic a bílých znaků 34

3 Návrh **35**

4 Implementace a testování **47**
4.1 Použité nástroje 47
4.2 Organizační struktura 48
4.3 Implementační detaily 48
4.4 Testování korektnosti 53

5 Experimentální měření **55**
5.1 Granularita ukládaného stavu 56
5.2 Transformace 57
5.3 Paměťová složitost detekce změn 60

Závěr **63**

Literatura **65**

A Grafy **67**

B Seznam použitých zkratek **75**

C Obsah přiloženého CD **77**

Seznam obrázků

1.1 Derivační strom. 5
1.2 Abstraktní syntaktický strom. 6

2.1 Příklad opakované analýzy podstromu 31

Seznam diagramů

2.1 Vzor proxy. 28

3.1 Rozhraní IAstChangeProvider 36
3.2 Třídy TreeChange, TreeAction a TreeStructure 36
3.3 Třída OracleContextState, rozhraní IAstObserver 37
3.4 Rozhraní IAstChangeProviderFactory 37
3.5 Rozhraní IOracleParserState, výčet OracleIncrementalParsing-Granularity 38
3.6 Rozhraní IStatefulNode a jeho implementace 39
3.7 Modifikace rozhraní ParserService 40
3.8 Třída IncrementalParser 40
3.9 Návrh hierarchie pro práci s predikáty. 41
3.10 Modifikace třídy AbstractAstTransformTask 45

Seznam grafů

A.1 Měření vlivu granularity ukládaného stavu 69
A.2 Měření transformace spojení řetězcových literálů 70
A.3 Měření transformace převodu DECODE na CASE 71
A.4 Měření transformace ORDER BY klauzule 72
A.5 Měření transformace sjednocující popisek bloku 73
A.6 Měření transformací IF THEN EXIT, klíčového slova IN a velikosti klíčových slov 74

Seznam tabulek

5.1 Měření vlivu ukládaného stavu 56
5.2 Měření transformace spojení řetězcových literálů 57
5.3 Měření transformace převodu DECODE na CASE 58
5.4 Měření transformace ORDER BY klauzule 58
5.5 Měření transformace sjednocující popisek bloku 59
5.6 Měření transformací IF THEN EXIT, klíčového slova IN a velikosti klíčových slov 60
5.7 Paměťová náročnost pro transformaci spojující řetězcové literály. . 61
5.8 Paměťová náročnost pro transformaci převodu DECODE na CASE. 61
5.9 Paměťová náročnost pro transformaci ORDER BY klauzule. 62
5.10 Paměťová náročnost pro transformaci sjednocující popisek bloku. . 62
5.11 Paměťová náročnost transformace klíčového slova IN, změny velikostí klíčových slov a IF THEN EXIT příkazu. 62

Úvod

Tato práce se zabývá návrhem a implementací prototypu vylepšení u systému Manta Tools (MT) [1]. MT se zabývá analýzou, ale i automatickou úpravou Structured Query Language (SQL) skriptů [2]. Mezi podporované databázové systémy (DS) patří Oracle [3] či Teradata [4]. V této práci se zabýváme pouze systémem Oracle, odtud pochází část Procedural Language/Structured Query Language (PL/SQL) [5] z názvu této diplomové práce. V této práci se *nezabýváme* analytikou částí MT. Tato práce se týká pouze části transformační.

Mějme nějaký PL/SQL skript a nějakou normu (např. interní) předepisující formální požadavky na tento skript. Tímto požadavkem může být pro představu *nepoužívání* funkce **DECODE** ve prospěch použití výrazu **CASE**. MT umožňuje některé tyto požadavky splnit automaticky pomocí tzv. transformačních pravidel. Těchto transformací existuje celá řada, další příklady jsou uvedeny v podsekci 1.5.3. Systém MT je navíc navržen tak, že tyto transformace jsou uživatelsky definovatelné.

Transformace nepracuje přímo s řetězcem odpovídajícím PL/SQL skriptu, nýbrž s jistou vnitřní formou reprezentující tento skript. Tato vnitřní forma je obecně známa jako Abstraktní syntaktický strom (AST). Nad touto reprezentací se vždy provádí tzv. *sémantická analýza*. Pod tímto pojmem si prozatím představme např. analýzu toho, co daný identifikátor označuje: Je to sloupec, proměnná či volání funkce?

Drtivá většina transformací funguje poté tak, že v AST hledají výskyt definovaného vzoru. Při úspěšném nálezu uzlu stromu odpovídajícímu hledanému vzoru dojde k tomu, že u tohoto uzlu jsou pozměněni jeho potomci. Typicky se jedná o odebraní jistých uzlů, přidání uzlů zcela nových, a v neposlední řadě také přesun uzlů k novým rodičům.

Zde leží jádro této práce. Po provedení výše uvedené transformace je totiž AST v nekonzistentním stavu. Tyto inkonzistence mohou být několika druhů. Především se můžeme snadno dostat do stavu, kdy výsledek předchozí sémantické analýzy již nadále nebude platný. To je způsobeno zejména tím, že nově vytvořené uzly ani analyzovány nebyly a nelze tak ani jednoduše provést.

Další inkonzistence je způsobena tím, že každý *listový uzel* AST obsahuje informaci o své pozici (číslo řádku a sloupce, kde začíná jím reprezentovaná část skriptu). Tuto informaci může transformace snadno rozbít vzhledem k tomu, že může libovolně přidávat a odebírat uzly.

V této práci jde tedy o to, abychom zajistili konzistenci reprezentace po provedení transformace. Současné řešení zajištění konzistence je takové, že si AST s výsledkem transformace převedeme na *řetězec*, ze kterého vytvoříme zcela *nový* AST. To je postup korektní, avšak v mnoha případech velmi neefektivní. Cílem práce je toto provést *inkrementálně*, pakliže to bude aspoň v některých případech možné. To by mělo mít za výsledek ušetření poměrně značné části výpočetního času.

V této práci je navržena implementace založená na tzv. predikátech. Tyto predikáty rozhodují, které části AST je nutné opravit, případně jakým způsobem opravit. Toto rozhodnutí je založeno na skutečně provedených změnách v AST. Změnou se rozumí operace přidání, odebrání či nahrazení potomka uzlu. Pro detekci provedených změn v AST jsou implementovány dvě odlišné strategie.

Navržený systém predikátů je obecně použitelný jednak pro predikáty zajišťující inkrementální zpracování logického bloku (např. příkazu, PL/SQL bloku), jednak pro predikáty řešící speciální případ (přidání klíčového slova **IN** do parametrů procedury aj.). Konkrétní implementované predikáty jsou zvoleny tak, abychom pokryli co největší třídu změn prováděných typickými transformacemi.

Tato implementace inkrementální analýzy dosahuje pro zvolené transformace poměrně výrazného časového zlepšení. Pro některé transformace, např. pro výše zmíněné přidání implicitního klíčového slova, je dosažené vylepšení časové složitosti v řádu stonásobků. V typickém případě je toto vylepšení v řádu jednotek.

Organizace textu

V kapitole 1 je popsána teorie týkající se AST a jeho konstrukce pomocí nástroje ANTLR. Rovněž zde najdeme stručný popis základních syntaktických konstruktů jazyka PL/SQL. V neposlední řadě je zde popsána základní architektura systému MT nutná pro další části této práce.

Následující kapitola 2 analyzuje možnosti řešení. V této kapitole jsou i jednotlivá řešení *abstraktně* popsána. Na kapitolu Analýza navazuje kapitola 3, kde je zvoleno řešení, které je posléze rozpracováno do větších podrobností. Zde jsou již popsány použité datové struktury.

Předposlední kapitola 4 popisuje některé implementační detaily. Rovněž je zde uveden postup, jakým probíhalo testování korektnosti. Poslední kapitola č. 5 se zabývá experimentálním měřením dosaženého zlepšení.

Kapitola 1

Základní pojmy

Tato kapitola se věnuje popisu teorie a technologií používaných v další části této práce. První sekce 1.1 se věnuje konstrukci vnitřní reprezentace PL/SQL skriptu, tj. tvorbě AST.

1.1 Jazyk, gramatika

Jazyk Nechť Σ je množina nějakých znaků. Tuto množinu nazýváme *abeceda*. Jako *řetězec* označme libovolnou posloupnost těchto znaků. Množinu všech řetězců označíme jako Σ^*. Prázdný řetězec budeme značit jako ϵ. Množinu L takovou, že $L \subseteq \Sigma^*$, budeme označovat jako jazyk [6].

Gramatika Gramatika G může být formálně definována pomocí uspořádané čtveřice (N, Σ, P, S) [6], kde:

- N je množina neterminálních symbolů
- Σ je množina terminálních symbolů
- P je množina pravidel ve tvaru $(N \cup \Sigma)^* N (N \cup \Sigma)^* \rightarrow (N \cup \Sigma)^*$
- S je startovací symbol takový, že $S \in N$

Ve výše zmíněné definici předpokládáme, že $N \cap \Sigma = \emptyset$, tedy že množiny N a Σ jsou vzájemně disjunktní.

Jednotlivé gramatiky můžeme zařadit do jednotlivých tříd dle toho, jaká další omezení klademe na tvar pravidel. Tzv. Chomského hierarchie [6] je tvořena čtyřmi třídami. V této práci budeme používat bezkontextové gramatiky. Následuje příklad 1 s gramatikou popisující aritmetické výrazy.

Příklad 1 V tomto příkladu budeme chtít vytvořit gramatiku pro zápis aritmetických výrazů. Tato gramatika bude podporovat konstanty, proměnné,

operátory sčítání, odečítání, násobení a dělení, a konečně také závorky. Rovněž budeme chtít respektovat prioritu operátorů.

Tato gramatika je tedy tvořena uspořádanou čtveřicí (N, Σ, P, S), kde jsou jednotlivé symboly definovány následovně:

- $N = \{A, B, E, F, T\}$
- $\Sigma = \{(,), id, number, +, -, *, /\}$
- $S = E$
- P má následující pravidla:

$$
\begin{array}{lcl}
(1)\, E & \rightarrow & EAT \\
(2)\, E & \rightarrow & T \\
(3)\, A & \rightarrow & * \,|\, / \\
(4)\, T & \rightarrow & TBF \\
(5)\, T & \rightarrow & F \\
(6)\, B & \rightarrow & + \,|\, - \\
(7)\, F & \rightarrow & id \,|\, number \\
(8)\, F & \rightarrow & (E)
\end{array}
$$

Příklad obsahuje i pravidla ve tvaru $A \rightarrow B \,|\, C$. Je to pouze zkratka pro dvě pravidla $A \rightarrow B$ a $A \rightarrow C$. ♠

Derivace Nechť $G = (N, T, P, S)$ je gramatika a $x, y \in (N \cup T)^*$. Pak x derivuje y, značíme jako $x \Rightarrow y$, jestliže existuje $\alpha \rightarrow \beta \in P$ a $\gamma, \delta \in (N \cup T)^*$ takové, že $x = \gamma\alpha\delta$ a $y = \gamma\beta\delta$ [7, kapitola Simple Syntax-Directed Translator, s. 45].

Toto byla definice derivace pomocí jednoho kroku. Derivaci v k krocích, tj. $x \Rightarrow a_0 \Rightarrow a_1 \cdots \Rightarrow a_{k-1} \Rightarrow y$, značíme jako $x \Rightarrow^k y$. Pro označení derivace v 0 a více krocích používáme značení $x \Rightarrow^* y$ [7, kapitola Simple Syntax-Directed Translator, s. 45].

Jazyk generovaný gramatikou Jako jazyk L generovaný gramatikou G budeme značit následující množinu:

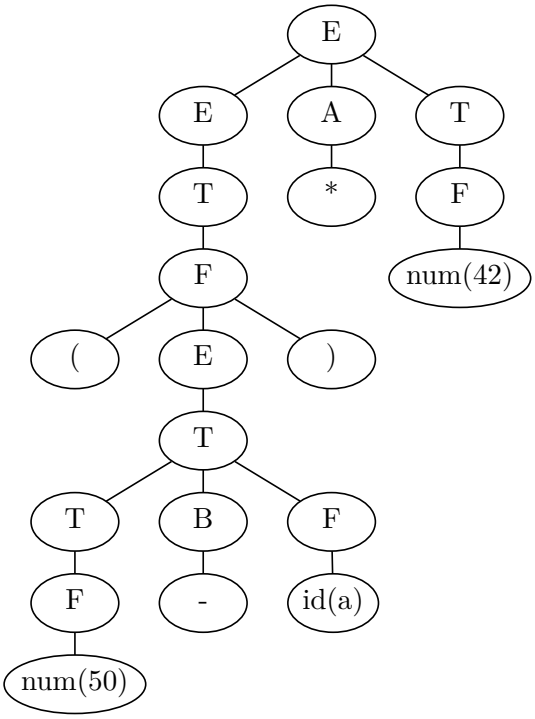
$$L(G) = \{w : w \in T^*, S \Rightarrow^* w\}$$

1.1.1 Derivační strom

Nechť máme gramatiku G a nějaký řetězec s, jenž náleží do jazyka generovaného gramatikou G. Jako *derivační strom* pro gramatiku G a vstup s označíme uspořádaný, ohodnocený, kořenový strom. Vnitřní uzly stromu jsou ohodnoceny *neterminálními* symboly, listy symboly *terminálními*. Kořen stromu je ohodnocen počátečním symbolem gramatiky G. Hodnoty listů v pořadí zleva doprava tvoří řetězec s.

Nechť máme nějaký uzel u, který má potomky $v_0, v_1, \ldots v_n$. Nechť $\$u$ je značení pro ohodnocení uzlu u. Pak uzel u vznikl z $\$u \rightarrow \$v_0\$v_1 \ldots \$v_n$, což je pravidlo v gramatice G [7, kapitola Simple Syntax-Directed Translator, s. 45].

Příklad 2 Mějme stejnou gramatiku jako v předchozím příkladu 1 a řetězec $s = (50-a)*42$. Derivační strom pro tuto gramatiku G a řetězec s je znázorněn na obrázku 1.1. ♠



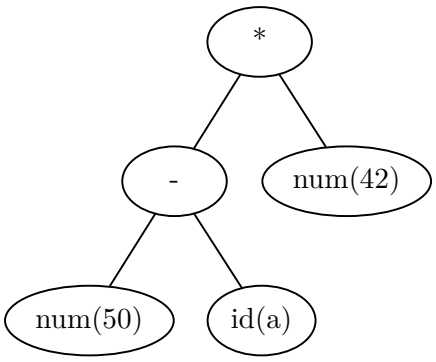
Obrázek 1.1: Derivační strom pro řetězec $(50 - a) * 42$.

1.1.2 Abstraktní syntaktický strom

V předchozí podsekci 1.1.1 jsme si ukázali, že derivační strom popisuje strukturu řetězce vůči nějaké gramatice. Pomocí gramatiky velmi často popisujeme programovací jazyky, v našem případě PL/SQL.

V praxi ovšem spíše oceníme popis *programovacích konstruktů*, nikoliv gramatiky sloužící k jejich formalizaci. Derivační strom obsahuje mnoho (pro nás) zbytečných informací. Typicky to jsou veškeré pomocné neterminální symboly. Stromová struktura zachycující programovací konstrukty se pak nazývá Abstraktní syntaktický strom (AST) [7, kapitola Simple Syntax-Directed Translator, s. 69].

Příklad 3 Máme stejnou gramatiku jako v příkladu 1, řetězec je stejný jako v předchozím příkladě 2. Na obrázku 1.2 pak vidíme, jak by mohl vypadat příslušný AST pro daný derivační strom. ♠



Obrázek 1.2: Abstraktní syntaktický strom pro řetězec $(50 - a) * 42$.

1.2 Syntaktická analýza

Cílem následujícího textu je stručné shrnutí, nikoliv kompletní popis včetně všech detailů. Ty lze najít např. v knize [8]. Syntaktická analýza je proces konstrukce derivačního stromu pro nějaký vstup s a gramatiku G. V zásadě existují dva odlišné přístupy: a) shora dolů, nebo b) zdola nahoru [8, kapitola Introduction to Parsing].

Syntaktická analýza shora dolů spočívá v konstrukci derivačního stromu směrem od kořene a postupně prochází strom směrem k listům. Analýza zdola nahoru postupuje opačně, tj. nejdříve konstruuje uzly reprezentující listy derivačního stromu a postupuje směrem ke kořeni. Zástupcem první kategorie je LL analýza, druhé kategorie LR, SLR či LALR analýza [9]. V následujícím textu se budeme věnovat pouze LL analýze. K tomu je nutné zavést formální výpočetní model.

1.2.1 Zásobníkový automat

Zásobníkový automat budeme formálně definovat jako uspořádanou sedmici $(Q, \Sigma, G, \delta, q_0, Z_0, F)$ [10], kde význam symbolů je následující:

- Q je konečná množina vnitřních stavů
- Σ je konečná vstupní abeceda
- G je konečná abeceda zásobníku
- δ je přechodová funkce, neboli zobrazení $Q \times (T \cup \epsilon) \times G \rightarrow Q \times G^*$
- q_0 je počáteční stav
- Z_0 je počáteční obsah zásobníku
- F je množina konečných stavů, $F \subset Q$

Zásobníkový automat je teoretický výpočetní model, jenž se používá pro syntaktické analyzátory bezkontextových gramatik. V našem případě si ovšem vystačíme s jednodušším modelem, který bude používat pouze jediný vnitřní stav.

1.2.2 LL(k) analýza

LL(k) je algoritmus deterministické syntaktické analýzy shora dolů využívající zásobníkový automat. Následující popis je velmi zjednodušený a nepopisuje *konstrukci* LL(k) analyzátoru, pouze postup konstrukce derivačního stromu. Pro kompletní popis včetně konstrukce analyzátoru odkazujeme na knihu [8, kapitola Deterministic Top-Down Parsing].

Algoritmus deterministické syntaktické analýzy provádí dva druhy operace: srovnání a expanze. K operaci srovnání může dojít pouze v případě, že se na vrcholu zásobníku nachází terminální symbol. Srovnáme tedy tento symbol s dalším dosud nezpracovaným symbolem na vstupu. Pokud jsou shodné, pokračujeme dále, v opačném případě nastala chyba.

Jestliže se na vrcholu zásobníku nachází symbol neterminální, dochází k provedení expanze. Expanze znamená, že se tento symbol X vyjme ze zásobníku a nahradí se symboly γ, jestliže existuje pravidlo $X \rightarrow \gamma$. Problém ovšem nastává, jestliže je takovýchto pravidel dostupných více. LL(k) analýza toto řeší pomocí tabulky, která nám říká pravidlo, jež máme použít. Klíčem této tabulky je symbol X, a dále k následujících, dosud nezpracovaných, vstupních symbolů (tzv. look-ahead).

Zbývá uvést, jaký je počáteční stav zásobníku. Tím je neterminální symbol S, který odpovídá startovacímu symbolu příslušné gramatiky. Rovněž musíme uvést, že k je *konečné* číslo udávající maximální délku dopředu prohlížených terminálních symbolů. Ty slouží k výběru správného pravidla pro provedení expanze.

1.2.3 LL(∗)

Nástroj ANTLR, jenž je využíván v systému MT, používá odvozenou variantu LL(k) analýzy zvanou LL(∗) [11, 12, kapitola LL(∗) parsing]. LL(∗) je ve své podstatě velmi obdobná LL(k) analýze, ovšem s tím rozdílem, že může využít potencionálně nekonečný look-ahead. To je dáno tím, že testujeme look-ahead vůči regulárnímu výrazu, který má konečný zápis, avšak potencionálně nekonečný vstup.

Příklad 4 Mějme následující gramatiku (v notaci, která je použita v nástroji ANTLR a bude popsána vzápětí):

```
method
    // napr, "int f(int x,int y);"
    : type ID '(' args ')' ';'
    | type ID '(' args ')' '{' body '}' // tj. "int f()
       {...}"
    ;
type: 'void' | 'int' ;
args: arg (',' arg)* ; // tj. "int x, int y, int z, ..."
arg : type ID ;
body: ... ;
```

Tato gramatika popisuje definici metody a byla převzata z knihy [13, část 11.2]. Jsou zde dvě pravidla pro definici metody. První je pro tzv. abstraktní metody, které nemají žádné tělo. Druhé pravidlo pokrývá „klasickou“ definici metody.

Uvedená gramatika je velmi dobře čitelná, ovšem není $LL(k)$ pro žádné k. To je dáno tím, že metoda může mít počet argumentů takový, že námi zvolené k nebude stačit. Klasickým způsobem, jak se s tímto problémem vyrovnat, je tzv. levá faktorizace. Jejím výsledkem je jediné pravidlo:

```
method
    :type ID '(' args ')' (';' | '{' body '}' )
    ;
```

Toto již je $LL(k)$, ovšem za cenu horší čitelnosti. Další nepříjemností je to, že musíme veškeré prováděné akce odložit, dokud nenarazíme na symbol `;` nebo `{`. Co to vlastně jsou akce je uvedeno v následující sekci 1.3.

V případě použití $LL(*)$ analyzátoru tento problém nenastane díky použití regulárních výrazů. Jelikož je takový $LL(*)$ analyzátor velmi složitý, tak se používá nástroj ANTLR, který tento analyzátor dokáže automaticky vygenerovat z příslušné gramatiky. ♠

1.3 ANTLR

V této části je stručně popsán nástroj ANTLR [12, 13]. Tento nástroj slouží k automatickému generování syntaktického analyzátoru na základě zadané gramatiky. Shrneme zde *základní* principy nutné pro další část práce. Budeme se zabývat verzí 3.5.

ANTLR se skládá ze dvou částí. První část můžeme chápat jako program, který má na vstupu rozšířenou gramatiku našeho jazyka (o tom, jak je gramatika rozšířená, později) a výstupem je zdrojový kód *parseru*. Pod pojmem parser se rozumí syntaktický analyzátor. Tento pojem je používán i v dalších částech textu. ANTLR ve verzi 3.5 podporuje více výstupních programovacích jazyků. V našem případě je použita Java.

Každý parser obsahuje část, která nikterak nezávisí na původní gramatice. Týká se to především různých procedur pro manipulaci se stromovými strukturami nebo použití vyrovnávacích pamětí při čtení vstupu. Proto jsou tyto části odděleny do tzv. *runtime* knihovny, jež tvoří druhou část nástroje ANTLR.

Následující ukázky jsou převzaty z knížky [12, kapitola 3]. Cílem je vytvořit gramatiku umožnující zadat aritmetické výrazy, se kterými budeme následně pracovat.

Základem je vstup v podobně gramatiky. V prvním kroku je nutné specifikovat jméno gramatiky.

```
grammar Expr;
```

Poté musíme nadefinovat pravidla pro lexikální analýzu. Ta využívá regulární výrazy a jejím výstupem jsou terminální symboly příslušné gramatiky jazyka.

```
ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t')+ {skip();} ;
```

Části `{skip();}` si prozatím nevšímejme. Nyní, když už máme terminální symboly, můžeme implementovat pravidla dané gramatiky.

```
prog:   stat+ ;

stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   multExpr (('+'|'-') multExpr)*
    ;

multExpr
    :   atom ('*' atom)*
    ;
```

```
atom:   INT
    |   ID
    |   '(' expr ')'
    ;
```

V této ukázce slouží znak | pro alternativu, znak + pro 1 a více opakování a znak * pro 0 a více opakování. Znak ?, který v tomto případě není použit, by sloužil k žádnému či jednomu výskytu daného elementu.

Nyní trochu pozměníme chování. Budeme chtít, aby byl daný výraz okamžitě vyhodnocen a vrácen jako výsledek.

```
grammar Expr;

@header {
import java.util.HashMap;
}

@members {
HashMap memory = new HashMap();
}

prog:   stat+ ;

stat:   expr NEWLINE {System.out.println($expr.value);}
    |   ID '=' expr NEWLINE
        { uloz $expr.value do pameti }
    |   NEWLINE
    ;

expr returns [int value]
    :   e=multExpr {$value = $e.value;}
        (   '+' e=multExpr {$value += $e.value;}
        |   '-' e=multExpr {$value -= $e.value;}
        )*
    ;

multExpr returns [int value]
    :   e=atom {$value = $e.value;} ('*' e=atom {$value *=
        $e.value;})*
    ;

atom returns [int value]
    :   INT {$value = Integer.parseInt($INT.text);}
    |   ID
        { nacti hodnotu z pameti a uloz do $value }
    |   '(' expr ')' {$value = $expr.value;}
    ;
```

Jak je z předchozí ukázky patrné, ANTLR umožňuje provádět tzv. *akce*, jež se mají provést. Jedná se o části kódu cílového jazyka, v našem případě Java. Další věcí, kterou zde vidíme poprvé, je syntaxe `e=atom`. Tato syntaxe umožňuje pojmenování si výsledku analýzy příslušného symbolu. To se bude hodit v zápětí při konstrukci AST.

Posledními novinkami jsou konstrukty `@header` a `@members`. Ty slouží k modifikaci generovaného kódu. Především k rozšíření funkčnosti generované třídy, které bude využito jednotlivými akcemi.

Do třetice budeme nyní chtít, aby výsledek naší syntaktické analýzy byl reprezentován pomocí AST zachycující pouze informace o výrazu. Upravíme proto gramatiku následovně:

```
options {
    output=AST;
    ASTLabelType=CommonTree; // Datovy typ uzlu
}

prog:   ( stat { zpracovat vystup;} )+ ;

stat:   expr NEWLINE        -> expr
    |   ID '=' expr NEWLINE -> ^('=' ID expr)
    |   NEWLINE             ->
    ;

expr:   (multExpr -> multExpr) (op=('+'|'-') multExpr ->
   ^($op $expr multExpr))*
    ;

multExpr
    :   (atom -> atom) (op='*' atom -> ^($op $multExpr
       atom))*
    ;

atom:   INT
    |   ID
    |   '(' e = expr ')' -> e
    ;
```

V předchozí ukázce byla opět nová syntaxe. Pokud neřekneme jinak, tak výsledek analýzy je tvořen plochou strukturou uzlů. Existuje ovšem modifikátor `->`. Ten umožňuje vytvořit libovolnou stromovou strukturu. Definice u pravidel `expr` a `multExpr` je poměrně divoká, blíže je vysvětleno v knize [13, kapitola 7, s. 187-188].

Na závěr této části zmíníme ještě tzv. *scopes*. Mějme nějaký druh informace, který chceme předávat z jednoho pravidla do druhého. Problém nastane, pokud tento řetězec předávání bude značně dlouhý. V tomto případě se může celý kód specifikující gramatiku stát značně nepřehledným. Typickým příkladem může být situace, kdy chceme určit, zda je nějaký identifikátor definován

v aktuálním bloku [12, kapitola 4, s. 114]. Odtud také pochází název scope. Nejdříve si tedy definujeme, jak má takový *globální* scope vypadat.

```
scope SymbolScope {
    List symbols ;
}
```

Poté upravíme i jednotlivé symboly, které pracují s našim scope.

```
classDefinition
scope SymbolScope;
: ...
;

methodDefinition
scope SymbolScope;
: ...
;
```

Zásadní rozdíl mezi definici `SymbolScope` jako scope a pomocí instanční proměnné je v tom, že v prvním případě se ANTLR postará o to, aby byla pro dotyčná pravidla umístěna *nová* instance `SymbolScope` na vrchol zásobníku [12, kapitola 4, s. 115]. Tato nově vytvořená instance může být navíc modifikována pomocí `@init` jak je naznačeno níže.

```
classDefinition
scope SymbolScope;
@init {
    $CScope::symbols = new ArrayList();
}
```

1.4 PL/SQL

V této části jsou představeny důležité konstrukty jazyka PL/SQL. V žádném případě se nejedná o kompletní popis jazyka. Současně se v této sekci předpokládá základní znalost Data manipulation language (DML) příkazů jako jsou **SELECT**, **UPDATE**, **INSERT** a **DELETE**. Vycházíme z knihy [5].

Nejdříve zavedeme notaci, jež bude nadále používána. Tvar [PRAVIDLO] znamená, že PRAVIDLO je v dané syntaktické konstrukci nepovinné. Obdobně PRAVIDLO∗(,) značí, že se PRAVIDLO může v syntaktické konstrukci opakovat libovolněkrát, jednotlivé výskyty jsou odděleny znakem ‘,’.

V následujícím textu se používá DATOVY_TYP pro označení datového typu. Tím může být např. **NUMBER**, **VARCHAR2**(20), **TIMESTAMP**, **INTERVAL**, **BLOB** ... Více o datových typech je v knize [5].

Základním kamenem je tzv. *blok*. Blok může mít tři části: deklaraci proměnných, deklaraci příkazů a ošetření výjimek. Syntaxe je následující:

```
[DECLARE
  <deklarace promennych>]
BEGIN
  <deklarace prikazu>
[EXCEPTION
  <osetreni vyjimek>]
END;
```

První a poslední část, tj. **DECLARE** a **EXCEPTION** jsou nepovinné. Bloky lze samozřejmě vzájemně vnořovat. Lze je rovněž opatřit tzv. popisky. Ty mohou být před blokem samotným ve tvaru <<popisek>>, nebo naopak na jeho konci ve formě **END** popisek;. Nyní se podíváme podrobněji na deklaraci proměnných. Ty lze deklarovat pomocí syntaxe:

```
jmeno DATOVY_TYP [:= HODNOTA];
```

Vidíme tedy, že inicializace na danou hodnotu je nepovinná. S obdobnou syntaxí můžeme definovat i konstanty:

```
jmeno CONSTANT DATOVY_TYP := HODNOTA;
```

Přeskočíme nyní část s deklarací příkazů a podíváme se na ošetření výjimek. To může obsahovat sérii výrazů v následujícím tvaru:

```
WHEN vyjimka THEN <prikazy>
```

Označení `vyjimka` v tomto případě odkazuje na jméno výjimky. Ty mohou být předdefinované, ale i uživatelsky definované. Protože se jedná o poměrně komplexní problém, tak pouze odkážeme opět na knihu [5, kapitola 6].

Doteď jsme pracovali s tzv. *anonymními* bloky. Praktičtější jsou *pojmenované* bloky, jež jsou tvořeny *procedurami* či *funkcemi*. Základní syntaxe pro proceduru je:

```
PROCEDURE jmeno [(PARAMETR *(,))]
IS
  <deklarace promennych>
BEGIN
  <deklarace prikazu>
[EXCEPTION
  <ostreni vyjimek>]
END;
```

Velmi podobné jsou funkce:

```
FUNCTION jmeno [(PARAMETR *( ,))]
RETURN DATOVY_TYP
IS
  <deklarace promennych>
BEGIN
  <deklarace prikazu>
[EXCEPTION
  <ostreni vyjimek>]
END;
```

Jediný rozdíl mezi procedurou a funkci je v tom, jakým způsobem je předáván výsledek. U funkce je datový typ výsledku explicitně vyjádřen pomocí **RETURN**. Zbývá ještě ukázat, jaká je syntaxe pro parametry. Hvězdička (*) napovídá, že počet parametrů může být libovolný. Parametry mohou být vstupní (**IN**), výstupní (**OUT**), případně obojí (**IN OUT**). V případě, že není určeno jinak, tak je parametr provažován za vstupní, tj. **IN**. Nyní již slíbená syntaxe:

```
jmeno_parametru [IN|OUT|IN OUT] DATOVY_TYP [:= HODNOTA]
```

Nyní se již konečně dostáváme k samotným příkazům. PL/SQL podporuje tradiční imperativní způsob řízení programu, tj. podmínky a cykly. Jednotlivé příkazy jsou od sebe odděleny středníkem ';'. Syntaxe pro podmínku je:

```
IF <podminka> THEN
  <prikazy>
[ELSIF <podminka THEN
  <prikazy>]*
[ELSE
  <prikazy>]
END IF;
```

Existuje i varianta příkazu switch nazvaná **CASE**:

```
CASE <vyraz>
  [WHEN <hodnota> THEN
    <prikazy>]*
  [ELSE
    <prikazy>]
```

Toto je tzv. *Simple CASE statement*. Existuje i zobecněná varianta zvaná *Searched CASE statement* [5, kapitola 4, s. 93].

PL/SQL obsahuje tři základní formy cyklů: **LOOP**, **FOR** a **WHILE**. **LOOP** je tzv. nekonečný cyklus, jehož syntaxe je:

```
LOOP
  <prikazy>
END LOOP;
```

Z tohoto cyklu lze vyskočit pomocí příkazu **EXIT**. Případně je možno použít podmíněného příkazu **EXIT WHEN** <podminka>. Velmi podobně jako **LOOP** je definován i **WHILE** cyklus:

```
WHILE <podminka>
LOOP
  <prikazy>
END LOOP;
```

Relativně nejsložitější je situace u **FOR** cyklu. Ten totiž obsahuje dvě formy [5, kapitola 5]. První je klasický čítač pro daný interval. Jeho syntaxe je následující:

```
FOR index IN [REVERSE] <od cisla> .. <do cisla>
LOOP
  <prikazy>
END LOOP;
```

Druhá varianta iteruje přes *kurzor*, nebo přes výsledek **SELECT** příkazu:

```
FOR zaznam IN { jmeno_kurzoru | (SELECT prikaz) }
LOOP
<prikazy>
END LOOP;
```

V případě, že je zadán explicitní **SELECT** příkaz, tak je chování očekávané, tj. proměnná `zaznam` bude postupně obsahovat jednotlivé řádky výsledku. Zbývá nám tedy vysvětlit, co to je kurzor.

Kurzor je ve své nejjednodušší podobě reference na výsledek **SELECT** dotazu [5, kapitola 15]. Můžeme ho deklarovat v **DECLARE** sekci bloku následovně:

```
CURSOR kurzor IS SELECT * FROM tabulka WHERE ...
```

Následně můžeme s tímto kurzorem pracovat, tj. otevřít pomocí `OPEN`, zavřít s `CLOSE` či načíst další záznam instrukcí `FETCH`. Kurzor navíc obsahuje sadu *atributů*, jež je možno použít pro zjišťování stavu. Pro představu uvedeme pouze atribut `jmeno_promenne%FOUND`, pomocí kterého můžeme zjistit výsledek posledního volání `FETCH`.

Tímto končí stručný přehled jazyka PL/SQL. Záměrně jsou vynechány části, které nejsou potřebné pro další část této práce. Především se jedná o různé objektové rozšíření a Data definition language (DDL) příkazy. V prvním případě bychom již byli mimo rozsah této práce, v druhém případě není podrobná znalost nezbytná. To z toho důvodu, že v případě změny DDL příkazu musíme postupovat konzervativně, tj. musíme vytvořit celý AST znovu. To se týká především u změn při definici schématu, datového typu apod.

1.5 Manta tools

V této sekci jsou popsané některé, pro nás zajímavé, technické aspekty systému MT, se kterými se budeme potkávat v dalších částech této práce.

1.5.1 Reprezentace AST

V předchozí části textu bylo zmíněno, že vstupní skript je reprezentován pomocí AST. Teď se tedy podíváme, jak *konkrétně* tento AST vypadá. Základem je třída `OracleAstNode`. Tato třída dědí z třídy `CommonTree`. Ta pochází z nástroje ANTLR, konkrétně z jeho *runtime* knihovny.

`CommonTree` obsahuje z našeho hlediska dvě zajímavé věci. Tou první je definice instančních atributů a metod mající za úkol „formování“ stromu. Tím jsou myšleny především atributy `children` a `parent`, a dále pak metody `addChild`, `deleteChild`, `setParent` ... [12].

Tato třída také obsahuje atribut `token` třídy `Token`. Často se používá i jméno `payload`. Pod pojmem *token* budeme v tomto textu nadále označovat typ uzlu. Pro představu takovýmto tokenem může být `AST_STATEMENT`, který je použit pro uzly reprezentující příkaz. Zde nutné zdůraznit, že token přímo *nesouvisí* s datovým typem, kterým je tento uzel fyzicky reprezentován.

Zpět k definici třídy `OracleAstNode`. Tato třída kromě jiného obsahuje následující definici:

```
OracleAstNode whiteTokens;
```

To je v podstatě speciální typ uzlu, který obsahuje bílé znaky, jež se mohou nacházet před či za daným uzlem. To je důležité ze dvou důvodů: a) Při převedení AST reprezentace zpět do řetězce chceme, aby výsledek byl stejný nejenom z hlediska syntaktického, ale i z hlediska vizuálního. Jinými slovy chceme zachovat i formátování. b) Mezi bílé znaky se řadí i komentáře. Ty jsou sice z hlediska sémantického zcela nepodstatné, ale jsou velmi důležité z hlediska uživatelského. Musíme je tedy zachovat.

Mezi další významné atributy třídy `OracleAstNode` patří `contextState`. Ten je typu `OracleContextState`, jehož stručnou definici naleznete na zdrojovém kódu 1. Tato definice obsahuje pro nás dva významné atributy.

Atribut `navigator` je typu `MantaAstNavigator<OracleAstNode>`. Třída `MantaAstNavigator` slouží k jednodušší navigaci v daném stromu. Nejzajímavější je zde asi to, že k navigaci můžeme použít i XPATH dotaz [14]. Toho využívají instanční metody třídy `OracleAstNode` jako `selectNodes` či `selectSingleNode`.

Druhým významným atributem je `resolverEntityFactory`. Ten má typ `ResolverEntitiesFactory`, který odpovídá tzv. *datovému slovníku*. O tom si povíme více v podsekci 1.5.2. Z důležitých členů třídy `OracleAstNode` nám zbývá už jen metoda `resolve`. Její význam si ukážeme opět až v podsekci 1.5.2.

```
1 public class OracleContextState {
2     MantaAstNavigator<IOracleAstNode> navigator;
3
4     ResolverEntitiesFactory resolverEntityFactory;
5
6     IResEntity defaultSchema;
7
8     Parser parser;
9 }
```

Zdrojový kód 1: Zjednodušená definice třídy `OracleContextState`.

Jelikož jazyk PL/SQL je syntakticky a sémanticky relativně bohatý, tak uzly AST nelze jednoduše popsat pouze třídou `OracleAstNode`. Především sémantická analýza jednoho obecného uzlu by byla značně komplikovaná. Proto jsou některé uzly významného typu reprezentovány vlastní třídou. Ta zajistí korektní provedení sémantické analýzy pro daný uzel. Typickým příkladem je uzel reprezentující PL/SQL blok, který je reprezentován třídou `AstBlock`, jež je podtřídou `OracleAstNode`.

1.5.2 Sémantická analýza

V této části se podíváme na to, co si v této práci budeme představovat pod pojmem sémantická analýza. Tento pojem budeme také často označovat jako *resolving*. Jedná se o dva názvy téže věci.

V systému MT se syntaktická a sémantická analýza provádí *současně*, respektive sémantická analýza se provádí již v průběhu analýzy syntaktické. Jelikož slovní definice by byla značně komplikovaná, tak uvádíme dva příklady.

Příklad 5 Mějme následující PL/SQL skript:

```
CREATE TABLE A(
  id INTEGER ,
  val VARCHAR (20)
);
DECLARE
  val2 VARCHAR (20) := 'test';
BEGIN
  val2 := (SELECT CONCAT(val, val2) FROM A);
END;
```

Zde by sémantická analýza měla určit „původ“ hodnot `val` a `val2` v příkazu **SELECT**. Mělo by tedy být rozlišeno, zda se jedná o proměnnou, či o sloupec

dané tabulky. V rámci resolvingu jsou rovněž plně kvalifikována jména. Tím je myšleno to, že místo jmen sloupců se uvažuje trojice (schéma, tabulka, sloupec). Obdobně pro tabulky, kde se uvažuje navíc schéma.

U tohoto skriptu by sémantická analýza měla určit, že první operand funkce CONCAT je jméno sloupce. U druhého operandu `val2` se naopak jedná o proměnnou. ♠

Příklad 6 Nyní trochu předchozí příklad 5 obměníme. Definice tabulky A zůstane stejná, přidáme ale i tabulku B.

```
CREATE TABLE B(
   id INTEGER,
   val2 VARCHAR2(20),
   a_id INTEGER,
   FOREIGN KEY (a_id) REFERENCES A(id)
);
DECLARE
   val2 VARCHAR (20) := 'test';
BEGIN
   val2 := (SELECT CONCAT(val, val2) FROM A
                                     NATURAL JOIN B);
END;
```

V tomto případě dojde k určení, že `val` pochází z tabulky A, zatímco hodnota `val2` z tabulky B. ♠

Předchozí dva příklady měly za úkol naznačit význam sémantické analýzy. Kromě výše zmíněných příkladů se určují i další atributy. Těmi může být např. typ výsledku **SELECT** dotazu.

Sémantická analýza je implementována pomocí metody `resolve` u třídy `OracleAstNode`. Odtud také pochází používaný název resolving. Základní chování této metody je jednoduché – daný uzel pouze deleguje práci na veškeré své potomky. Pokud má resolving na daném uzlu proběhnout jiným způsobem, pak je nutné, aby byl daný uzel reprezentován specializovanou podtřídou `OracleAstNode`.

Během resolvingu se využívá tzv. *datový slovník*. Ten slouží k vytváření instancí typu `IResEntity`, což je rozhraní s několika různými implementacemi. Existují implementace pro tabulku, kurzor, parametr funkce či procedury. V zásadě můžeme říct, ze smyslem `IResEntity` je poskytovat informace o daném objektu (tabulka, proměnná, kurzor aj.) Touto informací může být jméno, datový typ apod. Jako cíl sémantické analýzy pak můžeme označit získání těchto informací o daném objektu.

1.5.3 Transformace

V této části se konečně dostaneme k tomu, proč je v produktu MT potřebné dělat inkrementální analýzu. Jednou z funkcionalit je totiž automatická transformace kódu dle zadaných pravidel. Cílem těchto transformací je upravit daný PL/SQL skript tak, aby vyhovoval nějakým požadavkům. Typickým požadavek může být konformita vůči nějaké normě, nebo použití nových funkci namísto tzv. *deprecated* (zastaralých) variant. Nyní si tedy ukážeme několik konkrétních příkladů, a následně uvedeme seznam potencionálních problémů, které může transformační mechanizmus vyvolat.

Příklad 7 Mějme následující PL/SQL skript:

```
Select * FROM tabulka where id=42
```

Jednou z nejjednodušších transformací je konsolidace velikosti znaků u klíčových slov (**SELECT**, ...). Typicky územ je psát tato klíčová slova velkými písmeny. Po provedení transformace by tedy celý skript měl vypadat následovně:

```
SELECT * FROM tabulka WHERE id=42
```

♠

Příklad 8 Jazyk PL/SQL obsahuje funkci **DECODE**. Ta může být poměrně nečitelná, a proto je doporučeno požívat příkaz **CASE** [15]. Mějme následující PL/SQL skript:

```
DECODE(a, 1, 42, 2, -42, 24)
```

Neznalému člověku výše zmíněný úryvek PL/SQL moc neřekne. Proto existuje transformace taková, která výše zmíněný kód převede na následující:

```
CASE a
  WHEN 1 THEN 42
  WHEN 2 THEN -42
  ELSE 24
```

Tento kód je sice výrazně delší, ovšem je také mnohem čitelnější. ♠

Příklad 9 Spousta PL/SQL skriptů obsahuje konstrukty, které by mohly být vyhodnoceny ještě před samotným spuštěním. Mějme následující PL/SQL skript:

```
SELECT CONCAT('a', 'b', text) FROM tabulka
                              WHERE sloupec = NOT(TRUE)
```

Tento skript obsahuje dvě části, které by mohly být vyhodnoceny rovnou. Prvním jsou argumenty `'a'` a `'b'` u funkce CONCAT, které by mohly být spojeny do jednoho. Druhou částí je **NOT**(**TRUE**). Tu můžeme nahradit přímo za **FALSE**. Výsledek takovéto transformace je pak následující:

```
SELECT CONCAT('ab', text) FROM tabulka
                          WHERE sloupec=FALSE
```

♠

Příklad 10 Klauzule **ORDER BY** v jazyku SQL umožňuje zadat číslo sloupce z výsledné množiny, jenž má být použit pro řazení. Jelikož tato definice je poměrně zmatečná, tak si hned uvedeme příklad. Mějme následující tabulku a jednoduchý dotaz:

```
CREATE TABLE tabulka(
  id INTEGER,
  cislo INTEGER
);
```

```
SELECT * FROM tabulka ORDER BY 2;
```

V tomto dotazu budou výsledky řazeny dle sloupce č. 2. Tím je sloupec `cislo`. Takovýto dotaz je velmi nepřehledný, navíc se můžeme v případě změny struktury tabulky dostat snadno do problémů. Proto máme transformaci, které nám zajistí explicitní řazení dle sloupce `cislo`:

```
SELET * FROM tabulka ORDER BY cislo;
```

♠

Příklad 11 Obdobně jako v příkladu 10 si ukážeme transformaci, která napomáhá čitelnosti kódu a jeho robustnosti vůči změnám schématu. Mějme tabulku `tabulka`, která má schéma stejné jako v příkladu 10. Druhá tabulka a dotaz samotný jsou definované jako:

```
CREATE TABLE tabulka2(
  tid INTEGER,
  cislo2 INTEGER
);
```

```
SELECT cislo, cislo2 FROM tabulka JOIN tabulka2
                                  ON id=tid;
```

Tento dotaz je nepřehledný v tom smyslu, že se odkazujeme na sloupce, aniž bychom explicitně definovali tabulku. To může přinést problémy, především

pokud by obě tabulky sdílely jméno nějakého sloupce. Proto je lepší dotaz transformovat tak, aby tyto reference byly jednoznačné. To lze pomocí aliasů:

```
SELECT t1.cislo, t2.cislo2 FROM tabulka t1
                           JOIN tabulka2 t2
                           ON t1.id=t2.tid;
```

♠

Příklad 12 Jazyk PL/SQL umožňuje vynechat některé konstrukty, a tím použít výchozí hodnotu tohoto konstruktu. Mějme následující skript:

```
CREATE OR REPLACE PROCEDURE procedura (cislo NUMBER)
IS
  promenna NUMBER;
BEGIN
 ...
END;
```

DS při analýze toho skriptu použije některé výchozí hodnoty. Jsou jimi modifikátor **IN** u parametru, a dále inicializace proměnné na hodnotu **NULL**. Pomocí transformace můžeme toto vyjádřit explicitně.

```
CREATE OR REPLACE PROCEDURE procedura (cislo IN NUMBER)
IS
  promenna NUMBER := NULL;
BEGIN
 ...
END;
```

♠

Příklad 13 PL/SQL umožňuje blokům kódu přidat tzv. popisek. Mějme tedy skript:

```
<<jmeno_1>>
BEGIN
 ...
END;
```

Jelikož tento blok může být poměrně dlouhý, tak je možnost tento popisek uvést i za konec bloku. To může výrazně přispět k čitelnosti kódu. Výsledek transformace je tedy takovýto:

```
<<jmeno_1>>
BEGIN
 ...
END jmeno_1;
```

♠

Nyní něco málo k problémům, které mohou nastat v souvislosti s transformacemi. Jak bylo řečeno v podsekci 1.1.2, tak jednotlivé uzly AST mohou být reprezentovány různými třídami. Pro ilustraci by tedy uzel reprezentující PL/SQL blok měl být reprezentován třídou `AstBlock`. To ovšem nemusí být *po provedení* transformace pravda. Transformace pro konstrukci nového podstromu totiž využívá funkci `performConversion`, která toto nerespektuje.

Dále může nastat situace, že změna části podstromu AST způsobí, že výsledek sémantické analýzy některého z předků bude jiný. Viz podsekce 1.5.2.

Další situací, kterou bychom měli řešit, je neplatný výsledek transformace. Jelikož si transformaci může nadefinovat pomocí vzoru i sám uživatel, tak lze předpokládat, že se uživateli „povede“ definovat transformaci takovou, jejímž výsledkem bude neplatný PL/SQL kód. Proto bychom měli ověřit, že je výsledek transformace validní.

Rovněž je třeba řešit dopad transformace na platnost dat popisujících pozici (tj. číslo řádku a sloupce) v původním textu. Jelikož transformace může uzly libovolně přidávat či odebírat, tak se snadno dostaneme do situace, kdy tato data budou neplatná. Proto bychom měli provést konsolidaci.

1.6 Terminologie

V této sekci jsou definované některé termíny, jež se v této práci nadále používají.

Parser označení pro syntaktický analyzátor

Resolving synonymum pro sémantickou analýzu

Opětovné analyzování AST označuje opětovné provedení syntaktické a sémantické analýzy na části AST

Oprava pozic AST označuje provedení opravy pozic, tj. zajištění správnosti hodnot čísla řádku a sloupce, u všech dotčených uzlů AST

Znovupracování AST zahrnuje opětovné analyzování a opravu pozic AST

Kapitola 2

Analýza

V této části textu je analyzována současná situace. Z této analýzy vyplyne řada dílčích problémů. Tyto problémy jsou v této kapitole popsány, a dále jsou zde navržena různá řešení těchto dílčích problémů. Rovněž jsou zde zavedeny některé pojmy, jež se využívají i v následující kapitole 3. Nejdříve si ovšem zavedeme notaci, kterou budeme využívat v další části textu.

2.1 Notace

Nejdříve si zavedeme notaci, která nám umožní snadno určit, zda-li proměnná může nabývat hodnoty `null`. Nechť `Type` je nějaký datový typ. Ten definujeme tak, že proměnné toho typu nemohou obsahovat referenci na `null`. Potom z tohoto datového typu odvodíme `Type?`, který toho omezení nemá. Sekvence (seznam, pole) se značí jako `[A]`. Pro úplnost ještě uvádíme prázdný datový typ `()`, jenž je obdobou známého **void**.

Někdy bude užitečné, aby byl námi požadovaný datový typ generický. To budeme značit jako `Type<T>`. Dále se bude hodit mít zápis pro funkce. Funkce budeme zapisovat jako `[ T, U ]` ⇒ `R`, což v tomto případě reprezentuje funkci o dvou argumentech s typy `T` a `U`. Tato funkce vrací hodnotu typu `R`. Funkce bez argumentů se značí jako `[ ]` ⇒ `R`. Funkci bez návratové hodnoty (proceduru) značíme jako funkci vracející prázdný datový typ, tj. `[ T ]` ⇒ `()`.

`(A, B)` je typ popisující uspořádanou dvojici. První z této dvojice je typu `A`, druhý `B`. `(A, B)` je odlišného typu než `(B, A)`. Obdobně pro další n-tice.

Pro označení atributů a metod je použito pouze jednoduché slovní pojmenování, tj. tvar `metoda`. Pakliže ovšem z kontextu není zcela zřejmé, ke které třídě daný atribut či metoda patří, je použit zápis `Třída#jméno_členu`. Zbytek použitých konstruktů by měl silně připomínat jazyk Java, a proto zde nejsou uvedeny.

2.2 Dekompozice problému

Nechť máme nějakou transformaci (viz podsekce 1.5.3) a AST reprezentující nějaký skript. Chceme dosáhnout stavu, kdy po provedení této transformace bude výsledek validní, opět popsáno v podsekci 1.5.3. Budeme tedy muset znovuzpracovat podstrom AST, který byl transformací ovlivněn.

Pro samotné určení toho, které části AST je nutné znovuzpracovat, musíme nejdříve určit, které části tohoto stromu se vůbec změnily. Teprve poté lze určit podstromy, u kterých je nutné provést opětovnou syntaktickou a sémantickou analýzu, případně u kterých stačí opravit pozice. Dalším postupem je přirozeně provedení této dvojice analýz. Konečně v posledním kroku musíme opravit pozice. Co to znamená se dozvíme později v sekci 2.6. Když si celý postup shrneme do jednotlivých kroků:

1. Detekce provedených změn v AST.
2. Určení podstromů, které mají být opětovně analyzovány.
3. Samotná opětovná analýza podstromů AST.
4. Oprava pozic a bílých znaků.

Každá transformace kódu může transformovat více části AST. Proto je třeba rozhodnout, zda-li inkrementální analýzu provedeme po každé atomické operaci (tj. přidání či odebrání uzlu) či zda tuto analýzu provedeme hromadně až po ukončení celé transformace.

V průběhu transformace může nastat situace, kdy textová reprezentace AST bude *dočasně* syntakticky neplatná. Proto musíme zvolit postup takový, že zpracujeme vše hromadně až po provedení *celé* transformace. Navíc se díky tomuto postupu můžeme vyhnout opakované analýze stejného podstromu, jak je ukázáno na příkladu 16.

V následujících sekcích jsou analyzovány, někdy i včetně možností řešení, jednotlivé kroky.

2.3 Detekce změn v AST

Prvním krokem k inkrementálnímu zpracování je určení částí AST, které byly v průběhu transformace změněny. Pod pojmem *změna* uvažujeme operaci přidání, nahrazení či odebrání uzlu. Dále budeme navíc požadovat, aby změny nebyly zanořené, tj. jestliže byl uzel u změněn, pak *nebyl* změněn žádný jeho předek. Změna bude reprezentována typem `TreeChange`. Výsledkem této části implementace je tedy instance datového typu `[TreeChange]`.

Pro tento problém existují dva odlišné přístupy. První je založen na porovnání struktury AST před a po provedení nějaké transformace. Výhoda tohoto přístupu spočívá především v tom, že není třeba příliš upravovat jakýkoliv již

existující kód. Druhý přístup je založen na průběžném zaznamenávání provedených změn v AST. Tento přístup by, v porovnání s předchozím, měl být efektivnějším. V následujícím textu je popsáno několik variant využívajících jeden nebo druhý z těchto přístupů.

2.3.1 Porovnání dvou AST

Tato varianta využívá prvního přístupu, jak je uvedeno v sekci 2.3. Nejdříve musíme zajistit to, že vůbec budeme schopni porovnat strukturu AST před a po transformaci. To v důsledku znamená, že potřebujeme mít uloženou strukturu AST před transformací. Struktura po transformaci je reprezentována samotným výsledným AST.

Jedním z řešení by byla kompletní změna programovacího paradigmatu. To je ovšem spíše jen teoretická možnost, jelikož by to vyžadovalo změnu *velké*, patrně i celé, části existujícího kódu. V podstatě jde o to, že jednotlivé uzly budou neměnné. Operace přidání, odebrání či nahrazení uzlu tedy nezmění současný AST, ale vytvoří AST zcela nový. Samozřejmě se nejedná o kompletní kopii, jelikož původní i nový AST sdílí drtivou většinu (v průměrném případě) uzlů. Tento princip je znám jako *persistent data structure*, případně jako *immutable data structure* [16].

V systému MT je ovšem tento postup nepříliš dobře realizovatelný. Každý uzel totiž obsahuje referenci na svého rodiče, což je dáno tím, že uzel vychází z ANTLR třídy `CommonTree`. To by v důsledku znamenalo, že pro každou změnu musíme zkopírovat všechny uzly AST. To je nerealizovatelné. Pokud bychom netrvali na tom, že AST má vycházet z třídy `CommonTree`, tak bychom mohli navrhnout strukturu AST takovou, že uzel neobsahuje referenci na rodiče. To by ovšem způsobilo jiné výkonnostní problémy, např. při použití XPATH dotazu.

Praktičtějším řešením je zkopírovat strukturu AST před provedením transformace. Není třeba dělat kompletní kopii stromu, stačí nám zachytit, jaký uzel má jaké potomky. Následuje jednoduchý návrh:

```
1 public class TreeStructure {
2     OracleAstNode node;
3     TreeStructure[] children;
4 }
```

Třída `TreeStructure` je navržena tak, aby její paměťová reprezentace byla nejmenší možná. Z tohoto důvodu je pro reprezentaci potomků použito pole, jelikož počet potomků je známý a neměnný. Samotný algoritmus bude porovnávat dva stromy. Jeden je reprezentován pomocí instance `TreeStructure` a druhý pomocí `OracleAstNode`. Zde předem prozradíme, že tato varianta bude implementována. Popis porovnávacího algoritmu najdete v části zabývající se implementací, konkrétně v podsekci 4.3.1.

2.3.2 Zaznamenávání změn

Nyní se budeme zabývat variantou využívající průběžné zaznamenávání změn. K tomu budeme potřebovat datový typ `TreeAction`. Ten popisuje atomickou změnu, tj. přidaní či odebrání uzlu. Toto budeme nadále nazývat *akcemi*. V tuto chvíli nás nebude zajímat přesná implementace typu `TreeAction`.

Rozdíl proti `TreeChange` je v tom, že v případě akcí nepřipouštíme operaci nahrazení. Naopak povolujeme zanoření provedených akcí, tj. akce muže mít předka, nad kterým byla provedena jiná akce.

Předpokládejme, že máme způsob, jak dané akce zaznamenat, neboli máme definovaný mechanizmus vytváření instancí `TreeAction`. Nabízí se otázka, zda-li takto zaznamenané změny provedeme ihned, či si je pouze zaznamenáme a posléze provedeme všechny hromadně. První možnost si můžeme představit jako operaci známou pod pojmem logování. Druhou si můžeme představit jako plánování činnosti, která má být teprve provedena.

Nadále se budeme věnovat pouze první možnosti, tj. okamžitému provedení dané operace. Důvodem je obava z velkého množství potenciálních problémů.

Na tomto základu můžeme stavět dále. Samotné instance `TreeAction` je samozřejmě nutné někde ukládat. Od toho slouží modul `TreeChangeTracker`. Ten by měl být zodpovědný za ukládání provedených akcí (metoda `add`) i za celkovou detekci změn z provedených akcí (metoda `getChanges`). Opět zde prozradíme, že i tento postup bude implementován. Popis algoritmu je uveden v podsekci 4.3.2.

Zbývá ukázat, jakým způsobem budeme vytvářet instance `TreeAction` a jak je budeme ukládat. To lze provést několika způsoby. Společným jmenovatelem je to, že potřebujeme v okamžiku provedení akce spustit vlastní kód. V následujícím textu bude rozebráno několik různých variant postupu.

2.3.2.1 Přetížení metod OracleAstNode

Nejjednodušší a nejpřímočařejší je přetížení metod `addChild`, `insertChild`, a `deleteChild` [1] u třídy `OracleAstNode`. Tyto metody bychom přetížili tak, aby obsahovaly další argument typu `TreeChangeTracker`. V těchto přetížených metodách bychom vytvořili odpovídající záznam akce, a posléze zavolali původní implementace metody.

Tato varianta má několik problémů. Prvním problémem je to, že si danou instanci musíme předávat přes parametr. To je poměrně nešikovné a může to způsobit znepřehlednění celého kódu. Další problém je spíše problém „vkusu“: tyto přetížené metody logicky do `OracleAstNode` nepatří, jedná se o kód, který řeší poměrně speciální případ. Toto se pokusíme vyřešit v následující části.

[1] Ve skutečnosti existují i další obdobné metody, které by bylo třeba přetížit, ovšem z důvodu přehlednosti je zde neuvádíme.

2.3.2.2 OracleAstNode observer

Základní myšlenka je zde stejná, jako v případě podpodsekce 2.3.2.1. Opět budeme měnit metody `addChild`, `deleteChild` aj. u třídy `OracleAstNode`. Nyní ovšem tyto metody pouze přepíšeme, pro okolní svět se nic nezmění. Využijeme k tomu vzor *Observer*, česky známo též jako pozorovatel [17].

Ten si můžeme představit jako atribut třídy `OracleAstNode`, který má typ `([ TreeAction ] ⇒ ())?`. Metody modifikující potomky přepíšeme tak, aby zavolaly tuto funkci s příslušnou hodnotou akce. Samozřejmě pouze v případě, že hodnota tohoto atributu není `null`.

V principu je to velmi podobné předchozímu řešení. Pouze jsme se zbavili závislosti na `TreeChangeTracker`. Správné použití této varianty pak spočívá v tom, že před transformací vytvoříme novou instanci této funkce, která při zavolání přidá provedenou akci do příslušné instance `TreeChangeTracker`.

Tuto instanci musíme následně uložit do příslušného atributu rekurzivně u *všech* uzlů stromu, nehledě na to, zda-li k nějaké akci vůbec dojde. Navíc bychom po skončení transformace měli provést úklid. To je jediná, avšak poměrně zásadní, nevýhoda tohoto řešení.

2.3.2.3 Rozšíření OracleContextState

Předchozí přístup prezentovaný v podpodsekci 2.3.2.2 měl nevýhodu v nutném průchodu *celého* AST před i po provedení transformace. V této části tento nedostatek odstraníme. Pokud se podíváme na definici třídy `OracleAstNode`, tak vidíme, že obsahuje referenci na instanci `OracleContextState`. Tato instance je *společná* pro všechny uzly v jednom AST.

Využijeme tedy toho, že observer je stejný pro všechny uzly stromu. Tím dosáhneme stavu, kdy vytvoření i úklid tohoto pozorovatele nás stojí pouze konstantní čas.

2.3.2.4 Proxy objekty

Následující řešení je ze všech prezentovaných řešení asi to technicky nejsložitější. Využívá tzv. *proxy* objektů. Nejdříve si tedy popíšeme, jak proxy objekty fungují. Máme třídu s typem `T`. Tento typ má nějakou metodu `m`. Chceme dosáhnout toho, že kdykoliv se zavolá metoda `m`, tak se zavolá také nějaký náš kód, který označíme jako `c`.

Pomocí vzoru proxy [18] toho dosáhneme například tak, že vytvoříme nový typ `U`, který bude implementací rozhraní `I` a bude mít právě jeden atribut. Tímto atributem bude reference na instanci typu `T`, se kterou bude tato proxy svázána. Typ `T` rovněž implementuje rozhraní `I`. Z toho nám vyplývá, že typ `U` bude mít stejné metody jako typ `T`, přičemž jejich kód bude kromě metody `m` velmi jednoduchý: pouze se zavolá příslušná metoda na svázané instanci `T`. Pro metodu `m` navíc musíme zavolat kód `c`. Znázorněno v diagramu 2.1.

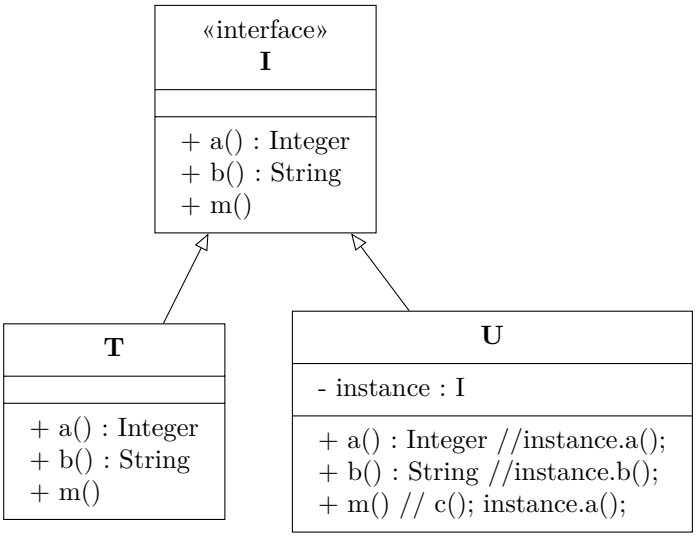


Diagram 2.1: Vzor proxy.

Výše zmíněný postup je sice poměrně jednoduchý, ale poměrně pracný z hlediska implementace. Podobného výsledku lze dosáhnout i pomocí techniky proxy objektů, jež jsou součástí Java Virtual Machine (JVM) [19]. Tento postup využívá tzv. *reflexe*, jejíž dopad na výkon může být znatelný [20].

V našem případě je situace u třídy `OracleAstNode` ještě složitější. Nejenom, že chceme zaznamenat volání metod `addChild`, `deleteChild` aj., ale navíc musíme zajistit, že i potomci daného uzlu budou proxy objekty. Museli bychom tedy ošetřit také veškeré metody, které jako návratovou hodnotu používají třídu `OracleAstNode`. To činí toto řešení mírně řečeno netriviálním.

2.3.2.5 Součást transformace

Poslední variantou je zaznamenávání prováděných akcí přímo *v místě volání* daných metod. To má tu výhodu, že nebudeme muset jakkoliv zasahovat do implementace třídy `OracleAstNode`.

Veškeré transformace jsou odvozeny od třídy `AbstractAstTransformTask`. V rámci této třídy lze transformacím poskytnout rozhraní pro modifikaci stromu, které by měly používat namísto přímého přístupu k uzlům stromu.

Problémem zde je ovšem to, že k modifikacím nemusí docházet pouze přímo v kódu transformace, ale často se tak děje i v jiných částech programu jako důsledek volání metody. Rovněž nejsme schopni garantovat použití našich metod pro modifikaci stromu. Stále totiž můžeme uzly modifikovat „přímo“. V tom případě nebude detekce fungovat.

2.4 Určení stromů k opětovné analýze

Nyní jsme v situaci, kdy daná transformace úspěšně proběhla a víme, které části AST byly touto transformací změněny. Dalším krokem je tedy určení, které části AST musíme opětovně syntakticky i sémanticky analyzovat. Z předešlé fáze máme výsledek datového typu `[TreeChange]`.

V podsekci 1.5.3 jsme naznačili, že výsledkem transformace obecně nemusí být validní AST. Jednak může obsahovat konstrukce, které neodpovídají syntaxi jazyka, jednak uzel s daným tokenem nemusí být reprezentován správnou třídou. Ukázáno na příkladu 14, respektive na příkladu 15.

Příklad 14 Předpokládejme, že máme transformaci definovanou uživatelem. Cílem této transformace je přidání implicitního klíčového slova **IN** do seznamu parametrů. Nyní nechť tato transformace obsahuje chybu. Jako vstup poslouží následující segment PL/SQL skriptu:

argument DATOVY_TYP

Výsledek transformace by měl následující:

argument **IN** DATOVY_TYP

Jelikož definovaná transformace obsahuje chybu, konkrétně špatné pořadí uzlů, tak dostaneme syntakticky neplatný kód:

argument DATOVY_TYP **IN**

To je samozřejmě nežádoucí chování, a proto je vhodné po provedení každé transformace opětovně provést syntaktickou analýzu (minimální možné části).

♠

Příklad 15 Mějme transformaci, která převede PL/SQL kód z následujícího tvaru:

IF <podminka> **THEN EXIT**; **END IF**;

na tvar

EXIT WHEN <podminka>;

Nyní srovnejme AST těchto dvou tvarů. Především se zaměřme na datové typy použité k reprezentaci jednotlivých uzlů. Notace pro zápis AST je následující: Obsah první špičaté závorky, tj. <>, je pro nás nezajímavý, obsahuje informace o pozici v původním textu. Poté následuje jméno tokenu, viz podsekce 1.5.1. Ve druhé špičaté závorce je uvedeno jméno třídy, kterou je daný uzel reprezentován.

Následuje zápis AST pro **IF** <podminka> **THEN EXIT**; **END IF**;:

```
<4,1> AST_IF <AstIf>
   <Uzel pro podminku>
    <4,10> THEN <OracleAstNode>
    <5,3> AST_STATEMENTS <OracleAstNode>
      <5,3> AST_STATEMENT <OracleAstNode>
        <5,3> AST_EXIT <AstExit>
          <5,3> EXIT <OracleAstNode>
        <5,7> ; <OracleAstNode>
  <6,1> END <OracleAstNode>
  <6,5> IF <OracleAstNode>
<6,7> ; <OracleAstNode>
```

A nyní pro **EXIT WHEN** <podminka>:

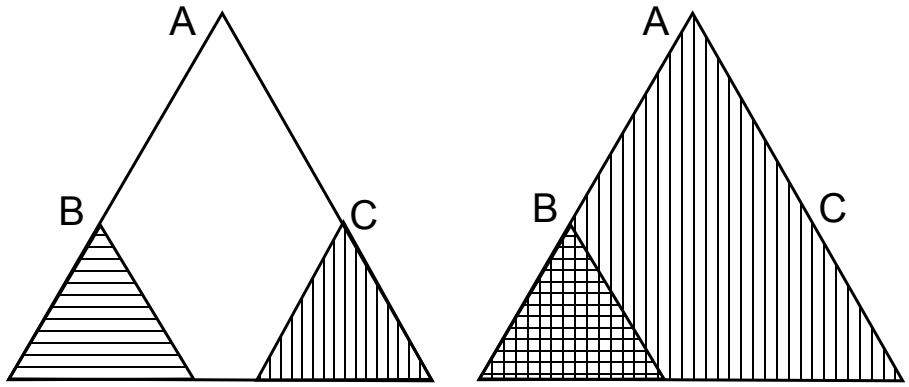
```
<8,1> AST_EXIT <OracleAstNode>
  <8,1> EXIT <OracleAstNode>
  <8,6> AST_JUMP_WHEN <OracleAstNode>
    <8,6> WHEN <OracleAstNode>
    <Uzel pro podminku>
<8,16> ; <OracleAstNode>
```

Srovnání ukazuje, že během transformace jsme ztratili datový typ u uzlu `AST_EXIT`. Původně byl uzel reprezentován pomocí třídy `AstExit`, což je specializovaná implementace přímo pro tento příkaz. Vy výsledku transformace je ovšem uzel `AST_EXIT` reprezentován pomocí třídy `OracleAstNode`, což je obecný typ uzlu. Toto chování porušuje konzistenci AST. ♠

Jak bylo naznačeno v úvodu této podsekce, cílem této fáze je určit, které části AST vyžadují opětovnou analýzu. Celý algoritmus by měl mít tyto vlastnosti:

- Sub-optimální řešení jsou přijatelná, ale nikdy nesmí být opětovně analyzováno méně informace, než je ve skutečnosti nutné.
- Opětovná analýza by měla být na daném pozměněném podstromu provedena *nejvýše jednou*. Toto je požadavek týkající se pouze efektivity, nikoliv korektnosti. Situace, kdy by mohlo dojít k opakované analýze, je ilustrována na příkladu 16.

Příklad 16 Mějme situaci takovou, že během transformace byly změny podstromy tak, jak je naznačeno na obrázku 2.1a. Vidíme, že došlo ke změně podstromů označených jako B a C. Poté spustíme algoritmus, který určí podstromy, jež se mají opětovně analyzovat. Výsledek je naznačen na obrázku 2.1b. Pokud bychom pouze naivně (tj. bez jakékoliv detekce překryvu) analyzovali všechny podstromy, které jsme získali jako výsledek, mohli bychom se dostat do situace, kde budeme analyzovat stejný podstrom vícekrát. V tomto konkrétním případě stačí provést opětovnou analýzu pouze podstromu označeného jako A. ♠



(a) Změny provedené během transformace.

(b) Podstromy které je nutné znovu analyzovat.

Obrázek 2.1: Příklad, kdy by mohlo dojít ke zbytečnému opakování analýzy podstromu.

Celý algoritmus se tedy bude skládat ze dvou částí: a) Určení podstromů, které je nutné opětovně analyzovat, pakliže je to vůbec nutné; b) Určení *minimální* množiny podstromů k analýze takové, abychom se vyhnuli opakovanému provádění téže operace nad stejnými segmenty AST. Až budeme mít hotovou část a), tak je část b) již triviální. Podstrom budeme analyzovat, právě když nebudeme analyzovat žádného jeho předka. Zbývá nám tedy navrhnout algoritmus pro část a).

Nechť výsledkem části a) bude typ `PredicateResult?`. Důvod, proč se toto nazývá `PredicateResult` bude objasněn vzápětí. Tento datový typ by měl mít dva podtypy: `ParseResult` a `NoParseResult`. První obsahuje informace nutné pro opětovnou analýzu podstromu, druhý informace nutné pro případ, kdy u podstromu stačí opravit pozice. Dále si ještě všimněme, že typ může nabývat hodnoty `null`. To pro případ, kdy inkrementální analýza není možná.

Celý návrh spočívá v použití *predikátů*. Predikát je funkce se signaturou `[ TreeChange ]` ⇒ `PredicateResult?`. V následujících řádcích bude popsána úloha predikátu.

Každý predikát v sobě obsahuje nějaký vzor změny. Vzor je zpravidla implicitní v podobě programového kódu. Vůči tomuto vzoru se predikát snaží spárovat provedenou změnu, kterou dostal jako argument. V případě úspěchu poté vrací instanci `PredicateResult`, což je popis toho, jak má být tato změna ošetřena.

Může se samozřejmě stát, že se mu toto spárování nepovede. Tento výsledek je vyjádřen návratovou hodnotou `null`. To nicméně ještě neznamená, že se inkrementální analýza jako celek nezdařila. Těchto predikátů může být použita celá řada a stačí, když uspěje aspoň jeden z nich.

V případě, že žádný z predikátů neuspěje, nám nezbývá nic jiného, než

naši snahu vzdát a okamžitě ukončit celou proceduru inkrementální analýzy.

Trochu zajímavější situace nastane, pokud nad jednou změnou uspěje více predikátů. V tomto případě vezmeme jako výsledek instanci `PredicateResult` takovou, která bude požadovat opětovnou analýzu nejmenší části AST.

Definujme si pomocné funkce, které se nám budou hodit při konstrukci ukázkového predikátu.

`ancestor : [ OracleAstNode, Token ]` ⇒ `OracleAstNode?` je funkce, která nám nalezne nejbližšího předka zadaného uzlu takového, že má požadovaný token. Viz příklad 17.

`selfOrAncestor` je definováno podobně jako `ancestor` s tím rozdílem, že výsledkem může být i uzel samotný, předaný jako argument.

Veškeré tyto funkce lze jednoduše definovat i pro ty uzly, které byly z AST odstraněny. To je zajištěno uloženými daty v instanci `TreeChange`.

Příklad 17 Máme následující PL/SQL příkaz:

```
SELECT id FROM tabulka WHERE a=b+c
```

Zjednodušená AST reprezentace pak vypadá takto (použitá notace zápisu AST je stejná jako v příkladu 15):

```
<7,1> AST_SELECT
  <7,1> SELECT <OracleAstNode>
  <7,8> AST_SELECT_LIST <OracleAstNode>
    ...
  <7,10> AST_FROM <OracleAstNode>
    ...
  <7,23> AST_WHERE <OracleAstNode>
    <7,23> WHERE <OracleAstNode>
      ...
      <7,30> = <OracleAstNode>
      ...
```

Nyní nechť hodnota `node` odpovídá uzlu na pozici `<7,30>`. Potom výraz `ancestor(node, AST_WHERE)` bude odpovídat hodnotě `whereNode`, kde `whereNode` je uzel na pozici `<7,23>`. Naopak `ancestor(node, AST_FROM)` bude mít hodnotu `null`, jelikož žádný uzel s tokenem `AST_FROM` není předkem uzlu `node`. ♠

2.4.1 Implementace predikátu

V této podsekci si ukážeme implementaci konkrétního predikátu. Tento predikát by měl zajistit korektní opětovnou analýzu v případě, kdy dojde ke změně ve **WHERE** podmínce příkazu (**SELECT**, **UPDATE** a **DELETE**).

Každý predikát musí v zásadě řešit tři situace: a) došlo k odstranění stromu; b) došlo k přidání stromu; c) došlo k nahrazení stromu. Náš predikát bude řešit pouze poslední situaci, ve zbylých selže, tj. vrátí `null`. Nechť `node` je změněný uzel. Jeho hodnotu získáme z instance `TreeChange`. Pseudokód je na algoritmu 1.

ancestor ← ancestorOrSelf(node, AST_WHERE);
if *canReparse(ancestor)* **then**
  return ParseResult(...);
end
else if *ancestor != null* **then**
  Rekurzivně zavolat na *ancestor*;
end

Algoritmus 1: Jednoduchý predikát pro případ nahrazeného podstromu.

Funkce `canReparse` ověřuje, že lze skutečně provést opětovnou analýzu. Jak se ukáže později v návrhu a implementaci, pro podporu opětovné analýzy je potřeba ukládat jistý stav, který je posléze použit pro analýzu samotnou. To ale poněkud předbíháme.

Prezentovaný predikát lze jednoduše zobecnit pro další typy uzlů. Mezi ně patří `AST_GROUP_BY`, `AST_ORDER_BY`, `AST_HAVING`. Musíme si ale dát pozor na uzly typu `AST_SELECT_LIST`, u nichž je vyžadováno opětovné analyzování na úrovni celého `AST_STATEMENT`, tedy příkazu.

Toto byla ukázka velmi jednoduchého predikátu. Během implementace je navíc možno sloučit více obdobných jednoduchých predikátů do jednoho většího a efektivnějšího. Tímto jsme ukončili část týkající se určování podstromů ke opětovné analýze. Přirozeně následuje samotná opětovná analýza.

2.5 Opětovná analýza podstromu

Nyní už víme, které části stromu musíme opětovně analyzovat. Musíme tedy ještě zajistit, že operace syntaktické a sémantické analýzy proběhne korektně. Druhou jmenovanou operaci uděláme korektní správným návrhem predikátů. Tento návrh by měl respektovat případné dopady jednotlivých PL/SQL výrazů na sémantickou analýzu. Později v této sekci jsou uvedeny i nějaké příklady.

Pro operaci *inkrementální* syntaktické analýzy musíme zajistit to, že parsování s počátečním symbolem `rule` proběhne stejně, jako bychom zpracovávali pravidlo `rule`, jež se se vyskytlo na zásobníku. To by samo o sobě nemělo být problém, ovšem náš parser je obohacen o určitý stav, pomocí něhož se rozhodujeme. Tento stav závisí na *předchozím* vstupu. Stav parseru si tedy

budeme muset v určitých zajímavých uzlech ukládat. Uložený stav bude posléze součástí uzlu AST.

Stav parseru je především reprezentován zásobníky pro jednotlivé scope. Co je to scope bylo popsáno v sekci 1.3. Tyto zásobníky je nutné zkopírovat.

Zbývá definovat, v jakých uzlech stromu musíme stav ukládat, abychom byli schopni opětovně analyzovat některé zajímavé části AST. Zcela jistě to bude příkaz (token `AST_STATEMENT`) a PL/SQL blok (`AST_BLOCK`). Nejspíše bude vhodné podporovat vybrané částí příkazů. Typickým příkladem bude podmínka (`AST_WHERE`). Obdobně pro části **GROUP BY**, **HAVING** nebo **ORDER BY**. Musíme zajistit, že veškeré tyto uzly budou reprezentovány datovým typem takovým, který umožní uložení stavu parseru.

Na závěr se sluší dodat, že některé transformace můžeme znovuzpracovat, aniž bychom potřebovali jakýkoliv stav. Je to kategorie transformací, která nevyžaduje opětovnou analýzu. Sem patří např. změna velikosti klíčových slov.

Nyní máme připraveno vše potřebné k opětovné analýze. Nejdříve si převedeme, pakliže to je nutné, změněnou část AST na řetězec. Nad tímto řetězcem provedeme analýzu a výsledek zařadíme na původní místo v AST.

2.6 Oprava pozic a bílých znaků

Poslední fází je oprava pozic a bílých znaků. Zde jde o to, že každý *listový* uzel obsahuje informace o své pozici v původním textu. Týká se to i tzv. bílých znaků, mezi něž patří i komentáře. Po provedení nějaké transformace mohou být tyto informace již neplatné. Jako následek transformace totiž dojde k posunutí o nějaký počet řádků, případně sloupců.

Pozice v textu je popsaná čtveřicí čísel: `line` (číslo řádku), `start`, `stop` a `charPositionInLine` (číslo sloupce). Atributy `start` a `stop` slouží jako indexy do vstupního bufferu. Ty opravovat nebudeme, jelikož je po provedení transformace nelze ani rozumně definovat.

Postup opravy bude zhruba následující: nejdříve seřadíme modifikované podstromy podle jejich pořadí ve stromu. Každý z těchto podstromů opravíme, tj. zajistíme že dané atributy odpovídají realitě. Během této opravy zjistíme, jaký je rozdíl mezi původní (před transformací) hodnotou řádku či sloupce a novou hodnotou. Tento rozdíl musíme přičíst ke všem následujícím uzlům, dokud nenarazíme na následující modifikovaný podstrom. Poté pokračujeme následujícím modifikovaným podstromem.

Jak je již patrně patrné, oprava pozic a bílých znaků je poměrně náročná. Teoreticky může vyžadovat úpravu všech listových uzlů.

Tímto jsme dokončili analýzu. Následující kapitola toto nadále rozpracovává do konkrétnější implementační podoby.

Kapitola 3

Návrh

V této kapitole je představen návrh architektury nutné pro inkrementální znovuzpracování PL/SQL skriptu. Tato architektura je navržena s ohledem na fakt, že jsme se rozhodli implementovat dva odlišné přístupy pro rozpoznávání provedených změn v AST. Konkrétně jsou jimi zaznamenávání provedených změn pomocí rozšíření třídy `OracleContextState` a porovnávání struktury AST. Podrobnosti a principy obou těchto přístupů byly popsány v sekci 2.3, konkrétně v podpodsekci 2.3.2.3, respektive v podsekci 2.3.1. Přístup založený na rozšíření třídy `OracleContextState` byl vybrán jakožto nejlepší možnost využívající zaznamenávání změn. Nevýhody ostatních přístupů založených na zaznamenávání změn byly popsány v podsekci 2.3.2.

V následujícím textu je velké množství diagramů tříd. U již existujících tříd jsou uvedeny pouze změněné atributy či metody. U jména typu se občas vyskytuje jako poslední znak otazník (?). Značí to, že daný atribut či metoda může nabývat hodnoty `null`. Stejně tak ne vždy jsou plně definovány konstruktory, getter či setter metody. Ty jsou pouze naznačeny pomocí `this(...)` nebo `...getters`.

Začneme s návrhem pro detekci změn. Ten je v diagramu 3.1. Základem je rozhraní `IAstChangeProvider` mající dvě virtuální metody. Metoda `getChanges` je samotné provedení detekce. Metoda `destroy` slouží, jak již název napovídá, jako destruktor. Toto rozhraní má dvě implementace. První je založena na srovnání struktury AST před a po transformaci, implementováno ve třídě `TreeCompare`. Druhá implementace využívá zaznamenávání prováděných operací pomocí vzoru observer, viz třída `ChangeTracker`. Obě tyto třídy mají konstruktor s parametrem, jenž je typu `OracleAstNode`. Pro úplnost přidáváme také v diagramu 3.2 definice souvisejících tříd `TreeChange`, `TreeAction` a `TreeStructure`.

V diagramu 3.3 si ukážeme, jakým způsobem bude implementován observer. Základem je rozhraní `IAstObserver<T>` s metodou `afterAdd`, respektive `beforeRemove`. Instance třídy `ChangeTracker` bude poté fungovat tak, že si vytvoří anonymní instanci tohoto observu, jež zajistí příslušné volání metody

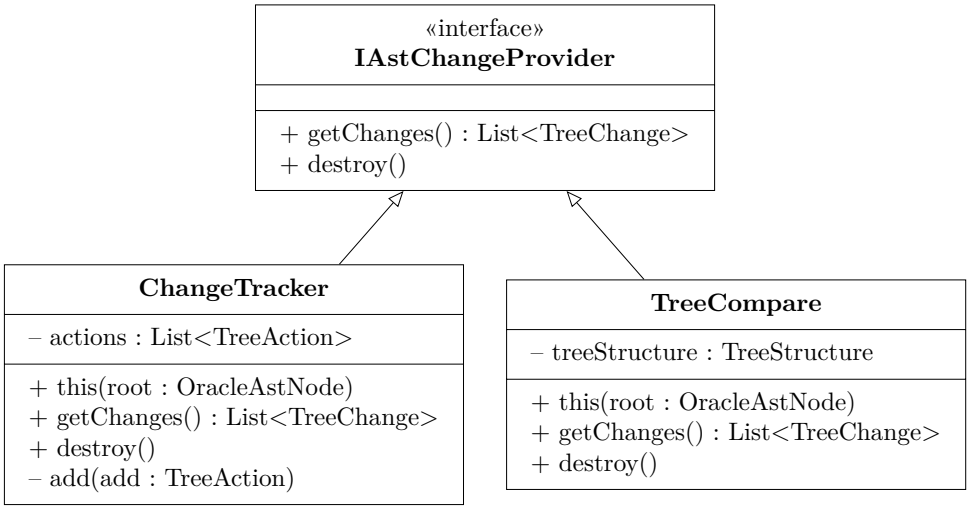


Diagram 3.1: Rozhraní `IAstChangeProvider` sloužící pro detekci změn ve stromu.

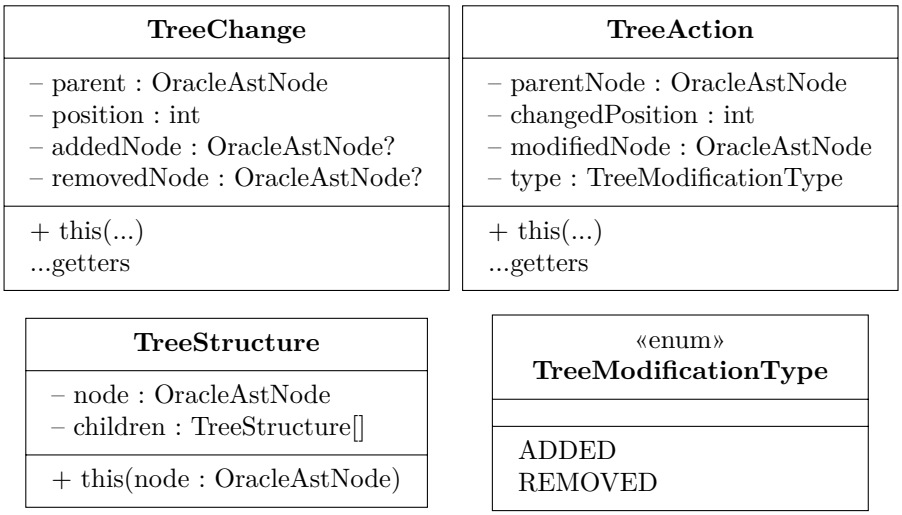


Diagram 3.2: Třídy `TreeChange`, `TreeAction` a `TreeStructure`.

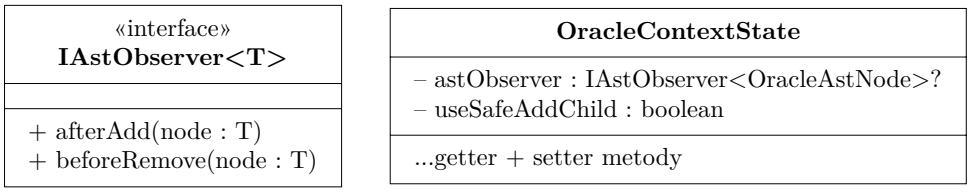


Diagram 3.3: Rozšíření třídy OracleContextState a rozhraní IAstObserver určené pro zaznamenání operací přidej/odstraň podstrom.

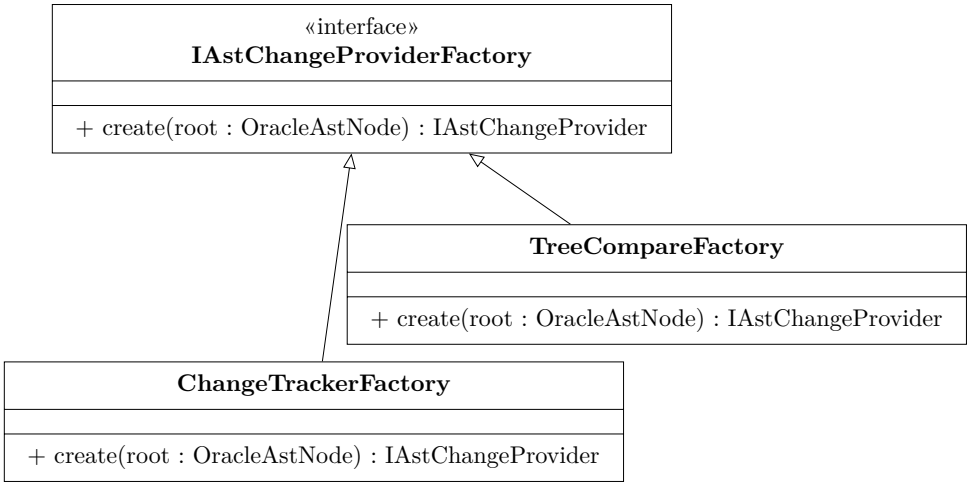


Diagram 3.4: Rozhraní `IAstChangeProviderFactory` pro vytváření instancí `IAstChangeProvider`.

`ChangeTracker#add`. Při zavolání `destroy` se tento observer opět odstraní. Na význam atributu `useSafeAddChild` uvedeného u třídy `OracleContextState` se podíváme až při popisu predikátů, jež jsou znázorněny na diagramu 3.9.

Na předchozí kroky navážeme tím, že si ukážeme, jak budeme vytvářet jednotlivé instance `IAstChangeProvider`. K tomu bude sloužit další rozhraní `IAstChangeProviderFactory`. Zobrazeno v diagramu 3.4. Instance toho rozhraní bude atributem třídy `AbstractAstTransformTask`. K tomu se ovšem dostaneme až později u diagramu 3.10.

Nejdříve se totiž podíváme na změny, které je nutné provést v gramatice. První věcí, co musíme zajistit, je to, že na některých místech v rámci AST budeme ukládat tzv. stav parseru. Tento stav bude reprezentován pomocí instancí rozhraní `IOracleParserState`, viz diagram 3.5. Implementace samotná bude skryta ve třídě `PLSQLMainParser`, což je vlastní parser. Instance stavu budeme vytvářet pomocí metody `PLSQLMainParser#makeParserState`, která má jediný argument typu `OracleIncrementalParsingGranularity`. Ten slouží k určení tzv. *granularity ukládání stavu*, která se bude hodit při pozdějším experimentálním měření. Popis jeho účelu najdete v podsekci 4.3.3.

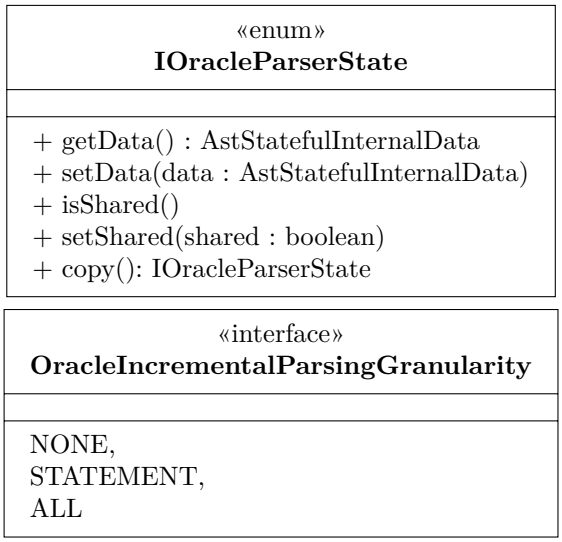


Diagram 3.5: Rozhraní `IOracleParserState` sloužící pro uchování stavu parseru. Výčet `OracleIncrementalParsingGranularity` definující granularitu ukládání stavu.

Rozhraní `IOracleParserState` definuje metody `getData` a `setData`. Ty slouží k uložení dodatečné informace nutné pro korektní provedení opětovné sémantické analýzy AST. Třídu `AstStatefulInternalData` zde neuvádíme. Její konkrétní použití bude uvedeno až na příkladu 18.

Dále toho rozhraní obsahuje metody `isShared` a `setShared`. Díky nim může několik uzlů AST sdílet stejnou instanci stavu parseru a tím něco málo ušetřit na paměťové i časové náročnosti. Týká se to především seznamu příkazů, kde není nutné vytvářet zvláštní kopii pro všechny příkazy. Stačí nám totiž pouze jedna společná, a teprve v případě potřeby vytvoříme kopii zvlášť pro daný příkaz.

Nyní musíme pouze definovat, ve kterých částech AST budeme tento stav ukládat. Stav uložíme tehdy, pakliže narazíme na uzel s jedním z následujících typů: a) `AST_STATEMENT`*, b) `AST_WHERE`, c) `AST_GROUPBY`, d) `AST_HAVING`, e) `AST_ORDER`, f) `AST_BLOCK`.

U uzlů s tokenem `AST_STATEMENT` ovšem nebudeme stav ukládat vždy, ale pouze tehdy pokud se bude jednat o příkaz, který je inkrementálně zpracovatelný. Tím může být pro představu definice bloku, DML operace, **COMMIT**, **ROLLBACK** aj. Stav tedy nebudeme ukládat pro příkazy vyžadující konzervativní chování, např. pro **ALTER TABLE**.

Jelikož si stav parseru potřebujeme nejenom zkopírovat, nýbrž i uložit, tak budeme dále požadovat, aby příslušné uzly AST byly reprezentovány instancemi rozhraní `IStatefulNode`. Toto rozhraní je definováno v diagramu 3.6. Má následující implementace: `AstStatefulNode`, `AstStatefulInternal`, případně již existující třídou `AstBlock`.

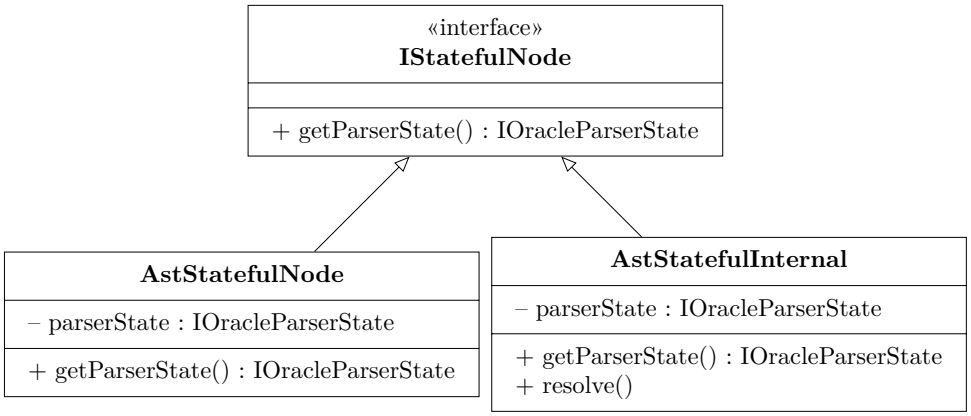


Diagram 3.6: Rozhraní `IStatefulNode`, jež musí být implementováno všemi uzly, u kterých chceme podporovat inkrementální zpracování.

Třída `AstStatefulNode` je velmi jednoduchá, pouze rozšiřuje základní třídu `OracleAstNode` o požadovanou funkcionalitu ukládání stavu. Další třída `AstStatefulInternal` je specializovaná pro podporu inkrementálního zpracování uzlů, jež jsou součástí většího celku. Týká se to především různých částí **SELECT** příkazu, jako jsou pro představu **WHERE** či **ORDER BY**. Tato implementace využívá dodatečné informace uložené v instanci rozhraní `IOracleParserState` pomocí metody `setData`. Ty jsou použity pro sémantickou analýzu.

Při použití klasického průběhu sémantické analýzy by potomci tohoto uzlu byli analyzováni některým ze svých předků, např. uzlem `AST_SELECT`. Uzel s instancí třídy `AstStatefulInternal` tak slouží jako jakási náhrada tohoto předka při opětovné sémantické analýze: drží si informace nutné k sémantické analýze takové, aby výsledek této analýzy byl stejný jako by ho provedl tento původní předek. Konkrétní způsob, jakým je této informace využito, bude ukázán v příkladu 18.

Následujícím krokem je úprava rozhraní `ParserService`, zobrazeno v diagramu 3.7. Záměr je takový, že umožníme syntaktickou analýzu s nastavitelným počátečním symbolem. Implementace je triviální, pouze pro každou možnost z výčtu symbolů gramatiky `OracleGrammarSymbol` přibude nová větev switch příkazu. Do této programové větve poté přidáme volání příslušné metody syntaktického analyzátoru. Opět uvedeno konkrétně v příkladu 18.

V diagramu 3.8 pokračujeme s definicí třídy `IncrementalParser`. Tato třída má metodu `parse` se dvěma argumenty: uzel s vrcholem AST a seznam provedených změn. Tato metoda vrací opět instanci `OracleAstNode`, která je již zpracována a korektně reprezentována. Tento uzel může být obecně *jiný*, než tato metoda dostala jako první argument.

V předchozí třídě `IncrementalParser` byl definován atribut `predicates`. Jedná se o pole instancí rozhraní `IChangePredicate`, jež je naznačeno v dia-

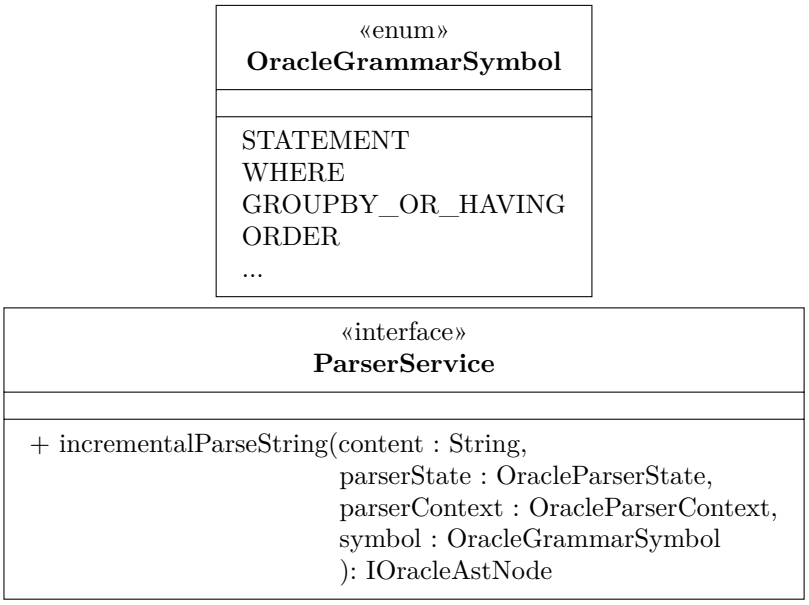


Diagram 3.7: Modifikace rozhraní `ParserService`.

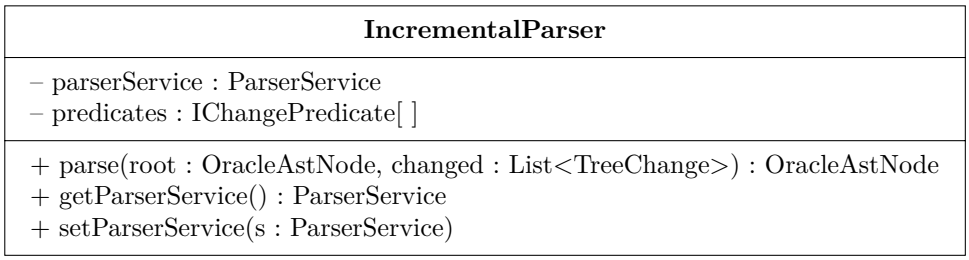


Diagram 3.8: Třída `IncrementalParser` zajišťující korektní zpracování.

gramu 3.9. Toto rozhraní má metodu `apply`, jež vrací typ `IPredicateResult`, případně `null`, pokud predikát neuspěl.

Kladný výsledek predikátu tedy může být dvojího typu. Prvním typem je `ParsePredicateResult`, jenž reprezentuje výsledek takový, který vyžaduje opětovnou analýzu AST daného uzlu, a obsahuje veškeré k tomu nezbytné informace. Naopak `NoParsePredicateResult` nám značí, že není třeba zpracovávat nic, pouze musíme opravit pozice AST (tj. číslo řádku a sloupce). K tomu slouží metoda `adjustNode`, jež vrací dvojici čísel udávající, o jaký počet řádků/sloupců musíme upravit následující uzly.

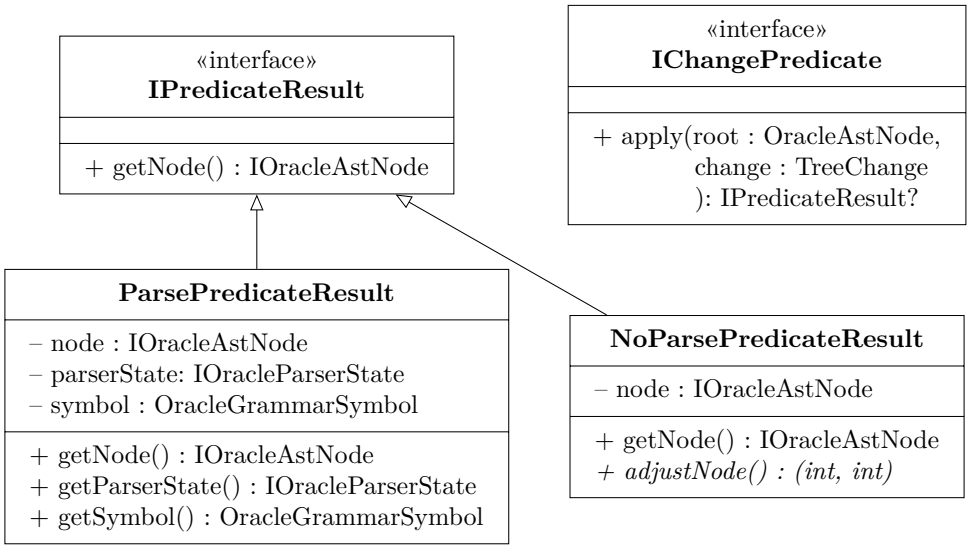


Diagram 3.9: Návrh hierarchie pro práci s predikáty.

Příklad 18 Cílem toho příkladu je konkrétně ukázat, co vše je nutné udělat pro podporu inkrementálního zpracování. Budeme chtít přidat podporu pro inkrementální zpracování podmínky (tj. **WHERE**) u příkazu **SELECT**. Prakticky totožně můžeme postupovat i u klauzulí **GROUP BY** a **HAVING**. Velmi podobný je i postup pro **ORDER BY**. Rovněž můžeme identický postup aplikovat i u příkazů **UPDATE** a **DELETE**.

Celá implementace se bude skládat z několika kroků:

1. Úprava gramatiky
2. Implementace nového predikátu detekujícího tuto možnost inkrementálního zpracování
3. (Úprava kódu sémantické analýzy, tj. metody resolve, nadřazeného uzlu)
4. Úprava implementace rozhraní `ParserService`

Krok č. 3 je pro některé případy nepovinný. Příkladem mohou být uzly typu `AST_STATEMENT`. V našem případě ovšem nutný je, nadřazeným uzlem se myslí uzel s tokenem `AST_SELECT`. Začneme s úpravou gramatiky.

Máme následující gramatiku (pro přehlednost zjednodušeno):

```
select:
    t001 = SELECT s_list = select_list
    t002 = FROM t_list = table_ref_list
    (t003 = WHERE cond = bool_value_expr)?
    -> ^(AST_SELECT<AstSelect>[contextState]
            $t001
            $s_list
```

```
                ^(AST_FROM $t002 $t_list)
                (AST_WHERE $t003 $cond)?
            )
        ;
```

Tuto gramatiku budeme chtít upravit tak, abychom před syntaktickou analýzou části **WHERE** uložili stav parseru. Současně budeme muset provést úpravy takové, aby byl příslušný podstrom AST reprezentován instancí rozhraní `IStatefulNode`. To z důvodu, abychom měli *kam* ukládat stav. Upravíme tedy gramatiku následovně:

```
select:
    t001 = SELECT s_list = select_list
    t002 = FROM t_list = table_ref_list
    (cond = where_clause)?
    -> ^(AST_SELECT<AstSelect>[contextState]
            $t001
            $s_list
            ^(AST_FROM $t002 $t_list)
            ($cond)?
        )
    ;

where_clause:
@init{
    // vytvorit kopii stavu parseru
    IOracleParserState parserState =
        makeParserState(ALL);
}
    : t001 = WHERE cond = bool_value_expr
    -> ^(AST_WHERE<AstStatefulInternal>
            [contextState, parserState] $t001 $cond)
    ;
```

Touto úpravou jsme dosáhli stavu, kdy: a) máme uložený stav na příslušném uzlu; b) uzel je reprezentován příslušným datovým typem. Tímto jsme vyřídili bod č. 1. Pokračujeme s implementací predikátu.

Nejdříve přidáme hodnotu `WHERE` do výčtu `OracleGrammarSymbol`. Samotný predikát budeme implementovat ve třídě `WhereChangePredicate`. Nechť byl modifikován uzel `node`. Modifikací se rozumí libovolná změna potomků. Tento predikát by měl uspět v případě, že se mu podaří najít předka s uzlem typu `AST_WHERE`, jenž je zároveň instancí `IStatefulNode`. V pseudokódu by to vypadalo zhruba takto:

```
1 while(node != null){
2     if(node.token == AST_WHERE &&
3        node instanceof IStatefulNode){
4         return ParsePredicateResult(
5             node,
```

```
6                 OracleGrammarSymbol.WHERE,
7                 node.getParseState());
8         }
9         node = node.getParent();
10    }
11    return null;
```

Pro úplnost implementace podpory inkrementálního zpracování musíme zajistit, že instance třídy `WhereChangePredicate` bude součástí pole `predicates`, jež je atributem třídy `IncrementalParser`.

Tímto jsme se přesunuli k implementaci bodu č. 3. Musíme upravit i sémantickou analýzu nad uzlem `AST_SELECT`, která je reprezentovaná třídou `AstSelect`. Konkrétně se jedná o metodu `resolveInternal`. Odtud taky pochází jméno `AstStatefulInternal`.

Důvodem, proč je tento krok na rozdíl od případu s `AST_STATEMENT` nutný, je to, že pro korektní sémantickou analýzu vnořených uzlů (a tedy i **WHERE** podmínky), musíme těmto uzlům předat dodatečné informace. V našem případě mohou v těchto podstromech být dva typy uzlů: poddotazy a uzly typu `AST_AGGR_REF`. Tyto uzly využívají k sémantické analýze dodatečné informace vycházející z nadřazeného dotazu. V našem případě to je kontext (označeno jako `context`) a výsledek dotazu (`sst`). Tyto dodatečné informace musíme uložit. K tomu slouží metoda `setData` u rozhraní `IOracleParserState` a instance `AstStatefulInternalData`. Nyní se podíváme na původní implementaci metody `resolveInternal` u třídy `AstSelect`:

```
1  // argument Stack<Map<String, IResObject>> context
2  IResObject sst =
3   factory.createResultset(DYNAMIC_SELECT_NAME, this);
4
5  // Vyhledat select klauzule v select podminkach.
6  List<AstMasterSelect> selectsInSelectList;
7  selectsInSelectList = findDependentSelects();
8  for (AstMasterSelect itm : selectsInSelectList) {
9      // Provest jejich resolving.
10     itm.resolveInternal(context, null);
11 }
12 // Provest resolving referenci v select, where,
13 // join, group by, order by polozkach
14 // Vyhledat reference
15 List<AstNamedReference> refs = findQueryReferences();
16 // Provest resolving techto referenci
17 for (AstNamedReference itm : refs) {
18     itm.resolveInternal(context, sst);
19 }
```

Rozšíření metody `resolve` bude tedy vypadat tak, že si uložíme hodnoty proměnných `context` a `sst`. V prvním případě navíc musíme použít metodu `clone`.

```
1  // argument Stack<Map<String, IResObject>> context
2  IResObject sst =
3   factory.createResultset(DYNAMIC_SELECT_NAME, this);
4  IStatefulNode node = selectSingleNode("AST_WHERE");
5  if(node != null
6    && node instanceof AstStatefulInternal){
7    // pokud existuje WHERE podminka
8    AstStatefulInternalData data;
9    data = new AstStatefulInternalData(context.clone()
10                                       , sst);
11   node.getParserState().setData(data);
12 }
```

Tímto jsme skončili s bodem č. 3. Zbývá nám upravit implementaci rozhraní `ParserService`, konkrétně metodu `incrementalParseString`. To už je poměrně triviální:

```
1  switch (grammarSymbol){
2    ...
3    case WHERE: {
4      PLSQLMainParser.where_clause_return
5          result_return = parser.where_clause();
6      result = result_return.getTree();
7      // musime zavolat i resolve
8      callResolve = true;
9      break;
10   }
11   ...
12 }
```

Za pozornost stojí řádek č.8. Ten definuje, že se má na výsledném uzlu zavolat metoda `resolve`, tj. sémantická analýza. Tato metoda je v některých případech volána přímo v definici gramatiky (např. `AST_STATEMENT`) a jindy, jako v tomto případě, ji musíme zavolat ručně.

Tímto je implementace podpory pro inkrementální zpracování uzlů typu `AST_WHERE` hotova. ♠

Z předchozí části textu máme ještě jeden dluh. Tím je implementace metody `resolve` u třídy `AstStatefulInternal`. Ta je na následujícím kódu:

```
1  AstStatefulInternalData
2      p = getParserState().getData();
3  // najit vnorene selecty a resolvovat je
4  for(AstMasterSelect node: findDependentSelects()){
5      node.resolveInternal(p.getContext(), null);
6  }
7
8  // najit vnorene reference
9  for(AstNamedReference ref: findReferences()){
10     ref.resolveInternal(p.getContext(),
11                         p.getReferencedObject());
12 }
13 return null;
```

Tímto jsme dosáhli stavu, že strom reprezentující podmínku, bude analyzován stejně, jako by byl analyzován v rámci příkazu **SELECT**.

Konečně dáme vše dohromady v třídě `AbstractAstTransformTask`. Ta je v diagramu 3.10.

AbstractAstTransformTask
+ getChangeProviderFactory() : IAstChangeProviderFactory + setChangeProviderFactory(f : IAstChangeProviderFactory) + setParserService(s : ParserService) + getParserService() : ParserService + doExecute(node : IMantaAstNode, output : ValidationReport)

Diagram 3.10: Třída `AbstractAstTransformTask`.

Na závěr této kapitoly popíšeme, jak bude vypadat metoda `doExexute` u třídy `AbstractAstTransformTask`. V prvním kroku vytvoříme instanci rozhraní `IAstChangeProvider` z příslušného uzlu a factory. Poté již provedeme transformaci samotnou. Po jejím ukončení voláme metodu `getChanges` a poté destruktor `destroy` u instance `IAstChangeProvider`. Inkrementální analýzu provedeme pomocí instance třídy `IncrementalParser` a její metody `parse`. Nakonec ještě musíme provést náhradu obsahu původního stromu. Nejlepší bude si opět uvést kód. Nejdříve uvádíme (zjednodušenou) původní verzi:

```
1  if(canTransform(node, report)){
2      doTransform(node, report);
3  }
```

Na zdrojovém kódu 2 uvádíme implementaci, jež zajistí korektní reprezentaci po provedení transformace, což ovšem dříve neplatilo. Tato modifikace je pro uživatele transformace naprosto *transparentní* (tj. programátor nemusí nic řešit, aby inkrementální analýza fungovala).

```
1  if (canTransform(node, output)) {
2    // vytvorit instanci IAstChangeProvider, tj.
3    // vytvoreni kopie ast nebo registrace observru
4    // v zavislosti na pouzite strategii
5    IAstChangeProvider changeProvider
6      = getChangeProviderFactory().create(node);
7    List<TreeChange> changes;
8    try {
9      // nepouzivat bezpecne pridani
10     node.getContext().setSafeAddChild(false);
11     // provest samotnou transformaci
12     doTransform(node, output);
13     // konsolidovat v dusledku pouziti safeChildAdd
14     // zjistit provedene zmeny
15     changes = changeProvider.getChanges();
16   }
17   finally {
18     // uvolnit zdroje, opet pouzivat bezpecne pridani
19     changeProvider.destroy();
20     node.getContext().setSafeAddChild(true);
21   }
22   // provest inkrementalni zpracovani
23   IncrementalParser parser =
24     new IncrementalParser(parserService);
25   OracleAstNode result = parser.parse(node, changes);
26   if(node != result){
27     // musime nahradit obsah puvodniho novym
28     node.removeChildren();
29     node.addChildren(result.getChildren());
30     ...
31   }
32 }
```

Zdrojový kód 2: Modifikovaná metoda `doExecute`.

Zbývá vysvětlit význam `safeChildAdd`. Jedná se zde o to, že standardní chování operace přidej potomka, kde potomek již má nějakého rodiče, je takové, že se odebere záznam o tomto potomkovi i v původním rodiči. Tím ovšem ztrácíme jistou část informace, kterou můžeme využít při implementaci predikátu. Atribut `safeChildAdd` toto chování vypíná. Abychom zajistili konzistentní strukturu AST po transformaci, musíme přidat průchod AST takový, který zajistí, že případné neplatné odkazy budou odstraněny.

Tímto jsme ukončili návrh. V následující kapitole jsou popsány konkrétní implementace některých navrhovaných metod a provedené změny v gramatice.

Kapitola 4

Implementace a testování

V první části této kapitoly je popsána organizační struktura implementace. Najdete zde popis souborové struktury či seznam použitých vývojových nástrojů.

V druhé části této kapitoly jsou popsány některé *zajímavé* implementační detaily. Především jsou zde uvedeny algoritmy pro detekci změn v rámci AST. Dále jsou zde uvedeny konkrétní změny v gramatice, jež byly provedeny. Zbylá část implementace přímo vychází z návrhu a není zde nadále rozebírána.

V poslední části této kapitoly se věnujeme testování korektnosti. Výsledky experimentálního měření jsou pak uvedeny a diskutovány až v další kapitole 5.

4.1 Použité nástroje

Pro vývoj a testování inkrementálního zpracování byly použity následující vývojové nástroje:

- Java, verze 1.6.
- Apache Maven. Nástroj sloužící ke správě závislostí, případně kompilaci.
- IntelliJ IDEA. Vývojové prostředí.
- Subversion (SVN). Verzovací systém.
- JUnit. Knihovna pro testování.

Tyto technologie jsou použity, jelikož již byly na projektu MT zavedené. Výjimku tvoří pouze vývojové prostředí, kde převládly osobní preference a zkušenosti autora.

4.2 Organizační struktura

Vzhledem k použitému systému sestavení (anglicky build system) Apache Maven je implementace MT rozdělena dle logických bloků do tzv. *artefaktů*. V této práci jsou pozměněny pouze tři z několika desítek artefaktů. Konkrétně jimi byly manta-connector-oracle-model, manta-connector-oracle-resolver a manta-transformer-ast-oracle.

Artefakt manta-connector-oracle-model obsahuje definice nejrůznějších rozhraní. Jmenovitě můžeme uvést třeba rozhraní `IAstObserver`, `IStatefulNode`, či `IOracleParserState`.

Druhý artefakt manta-connector-oracle-resolver obsahuje definice tříd uzlů AST. Najdeme zde i kód pro syntaktický analyzátor. Třídy `AstStatefulNode` a `AstStatefulInternal` najdeme rovněž zde.

Konečně v posledním artefaktu manta-transformer-ast-oracle najdeme implementace tříd týkajících se samotné *inkrementální* analýzy. Mezi ně patří např. třídy pro detekci provedených změn či implementace predikátů.

4.3 Implementační detaily

V této části je zmíněno několik implementačních detailů. Nejdříve je zmíněn postup u algoritmů pro detekci změn v AST. Poté uvedeme i konkrétní změny v gramatice.

4.3.1 Algoritmus porovnání stromů

V této části je popsán algoritmus detekující provedené změny pomocí porovnání struktury AST před a po provedení transformace. Připomínáme, že struktura původního stromu je reprezentována pomocí rekurzivní instance třídy `TreeStructure`. Její definici lze nalézt v diagramu č. 3.2.

Máme tedy instance dvou tříd: `TreeStructure` a `OracleAstNode`. Nejvyšší uzel AST se z definice transformace nikdy nezmění. Myšlenka tohoto algoritmu je pak založena na tom, že porovnáváme hodnoty referencí potomků. Nechť $reference_{old}$ je množina referencí potomků instance třídy `TreeStructure`. Obdobně je $reference_{new}$ množina pro instanci třídy `OracleAstNode`. Pak $reference_{old} \cap reference_{new}$ jsou potomci, kteří nebyli změněni. Množinu odstraněných potomků získáme jako $reference_{old} \setminus reference_{new}$ a přidaných potomků jako $reference_{new} \setminus reference_{old}$.

Výše zmíněný postup má jednu nevýhodu. Nedokáže totiž detekovat operaci nahrazení stromu. Proto je nutné tento postup modifikovat tak, abychom brali v potaz i pozici. Samotný algoritmus pak vypadá takto:

```
1  OracleAstNode node; // soucasna struktura stromu
2  TreeStructure origin; // puvodni struktura stromu
3  // vytvorit ze seznamu potomku mnoziny
4  Set<OracleAstNode> present = childrenSet(node);
5  Set<OracleAstNode> oldPresent = childrenSet(origin);
6
7  int oldIndex = 0;
8  int newIndex = 0;
9  while(oldIndex < origin.getChildrenCount() &&
10        newIndex < node.getChildCount()){
11     TreeStructure oldChild = origin.get[oldIndex];
12     OracleAstNode newChild = node.get(newIndex);
13     if(oldChild.getNode() == newChild) {
14         // shoda, zavolat funkci rekurzivne
15         oldIndex++;
16         newIndex++;
17     }
18     else if(present.contains(oldChild.getNode())) {
19         // vlozen novy uzel na pozici newIndex
20         newIndex++;
21     }
22     else if(oldPresent.contains(newChild)){
23         // smazan uzel
24         oldIndex++;
25     }
26     else {
27         // uzel nahrazen
28         oldIndex++;
29         newIndex++;
30     }
31 }
32 // vsechny nezpracovane uzly z origin jsou smazane
33 // vsechny nezpracovane uzly z node jsou pridane
```

Předchozí algoritmus funguje velmi obdobně jako algoritmus pro určení Levenshteinovy vzdálenosti (LV) [21]. Využíváme ovšem toho, že každá reference (tj. znak v terminologii LV) se může v každém seznamu vyskytnou nejvýše jednou. Díky tomu můžeme přímo určit, zda musíme uzel přidat, odebrat či nahradit.

4.3.2 Algoritmus sledování změn

Druhý algoritmus pro detekci provedených změn využívá zaznamenávání provedení dílčích změn. Tyto dílčí změny jsou reprezentovány pomocí instance List<TreeAction>, kde definici `TreeAction` najdeme v diagramu 3.2.

Nyní budeme uvažovat, že operace nahrazení podstromu se skládá z *po sobě jdoucí* dvojice akcí přidat a odebrat, případně v opačném pořadí. V tomto místě by bylo možno udělat algoritmus robustnější z toho pohledu, že nebudeme požadovat, aby tyto operace proběhly bezprostředně po sobě, nýbrž pouze po sobě v rámci daného rodiče. To by ovšem pravděpodobně zvýšilo časovou, případně paměťovou náročnost (byť ne asymptoticky). Nyní již samotný algoritmus:

```
1  // seznam provedenych zmen
2  List<TreeAction> actions;
3  // ignorovat zanorene zmeny
4  List<TreeAction> toProcess =
5      topLevelActions(actions);
6  //reference na predchozi zmenu
7  TreeAction last = null;
8  foreach (TreeAction action in toProcess) {
9      // dvojice (pridat, odstranit)
10     if(action.type == REMOVED) {
11         if(last != null && last.type == ADDED &&
12             action.parent == last.parent &&
13             action.position - 1 == last.position) {
14             //uzel byl nahrazen na pozici
15             //last.position u uzlu last.parent
16              last = null;
17              continue;
18         }
19     }
20     // dvojice (odstranit, pridat)
21     if(action.type == ADDED) {
22         if(last != null && last.type == REMOVED &&
23             action.parent == last.parent &&
24             action.position == last.position) {
25             //uzel byl nahrazen na pozici
26             //action.position u uzlu last.parent
27              last = null;
28              continue;
29         }
30     }
31     //jednoduchy prevod
32     if(last != null) {
33         // nejedna se o nahrazeni, jednoduchy prevod
34         // TreeAction na TreeChange
35     }
36     //ulozit referenci na predchozi
```

```
37        last = action;
38    }
39    if(last != null){
40        // nejedna se o nahrazeni, jednoduchy prevod
41        // TreeAction na TreeChange
42    }
43    return result;
```

Výše zmíněný algoritmus by se dal zcela jistě dále vylepšit. Nicméně tato varianta je poměrně jednoduchá a nenáročná na zdroje.

4.3.3 Změny v gramatice

První úpravou, která se musela provést v souboru s gramatikou, bylo přidání implementace `IOracleParserState`, což je rozhraní uvedené v diagramu 3.5. Nejdříve je třeba si uvědomit, co to vlastně znamená stav parseru. V našem případě to jsou jednotlivé zásobníky pro tzv. scope. Ty byly představeny v sekci 1.3. Dále je ještě nutné uchovat referenci na datový slovník. Jeho význam byl popsán v podsekci 1.5.2. Implementace rozhraní pro ukládání stavu parseru, tj. `IOracleParserState`, vypadá následovně:

```
 1  public static class OracleParserStateImpl
 2      implements IOracleParserState{
 3      Stack<PL_SQL_SCOPE_scope> $PL_SQL_SCOPE;
 4      Stack<SYMBOLTABLESCOPE_scope> $SYMBOLTABLESCOPE;
 5      Stack<ROUTINESCOPE_scope> $ROUTINESCOPE;
 6      //... apod pro ostatni scope
 7      ResScope dataDictionary;
 8      AstStatefulInternalData data = null;
 9      public OracleParserStateImpl(PLSQLMainParser i) {
10          this.dataDictionary = i.dataDictionary;
11          this.$BOOL_OR_EXPR =
12           instance.BOOL_OR_EXPR.clone();
13          // a obdobne pro dalsi scope
14      }
15      public void setData(AstStatefulInternalData arg) {
16          this.data = arg;
17      }
18      public AstStatefulInternalData getData() {
19          return this.data;
20      }
21      public IOracleParserState copy(){
22          // vytvoreni kopie
23      }
24  }
```

Na výše zmíněné implementaci je důležitá jedna věc. Tím je to, že dané zásobníky musíme *zkopírovat*, nestačí pouze zkopírovat referenci. To může činit operaci uložení stavu poměrně náročnou.

Zde připomínáme, že instance `IOracleParserState` budeme vytvářet pomocí metody `makeParserState`. Tato metoda má argument pro určení *granularity ukládání stavu* parseru. Tato granularita nám říká, jak moc se máme snažit při ukládání stavu. Pomocí hodnoty `NONE` (Nic) můžeme ukládání stavu parseru kompletně vypnout, a tím značně omezit možnosti případné inkrementální analýzy. Hodnota `STATEMENT` (Příkaz) ukládá stav pouze u příkazů. Konečně pomocí hodnoty `ALL` (Vše) budeme ukládat stav kdykoliv to bude možné. Smyslem toho argumentu bylo snadnější testování vlivu ukládání stavu na celkové zlepšení času běhu. Nyní přejdeme k úpravě gramatiky.

V příkladu 18 jsme si ukázali, jak musíme změnit gramatiku, abychom přidali podporu pro inkrementální zpracování **WHERE** podmínky. Takto je postupováno i ve skutečné implementaci, kde je přidáno pravidlo gramatiky `where_clause`. Podobně je upravena **WHERE** podmínka i u dalších příkazů **UPDATE** a **DELETE**, tj. u pravidla `update_statement`, respektive `delete_statement`. Zde se ovšem nepoužívá pravidlo `where_clause`, nicméně nově vytvořené pravidlo `where_clause_of`.

Další úpravou bylo pravidlo `group_by_or_having`. Tady ovšem stačilo přidat sekci `@init` s vytvořením kopie stavu. Poté již stačilo pouze specifikovat typ uzlu jako `AstStatefulInternal`.

Největší úpravy byly v oblasti pravidel generující uzly pro příkazy, tj. typu `AST_STATEMENT`. V původní implementaci byl tento uzel vytvářen z *výsledku* volání pravidla `statement`. V této implementaci je `AST_STATEMENT` již *součástí výsledku*. Důvodem je to, že v některých případech potřebujeme, aby byl tento uzel datového typu `OracleAstNode` a jindy datového typu `AstStatefulNode`. To v podstatě znamená, že některé uzly `AST_STATEMENT` podporují inkrementální zpracování, zatímco jiné ne.

Do gramatiky byla tedy přidána následující pravidla:

- `plsql_other_statement_incremental`
- `sql_statement_regular`
- `sql_statement_incremental`
- `dml_statement_incremental`

Tato pravidla vychází z původních pravidel a z jejich názvu je patrné, zda-li se jedná o příkazy s, nebo bez podpory inkrementálního zpracování.

Poslední změna se týkala uzlů `AST_BLOCK`. Tento uzel může, ale i nemusí, být součástí uzlu `AST_STATEMENT`. Situací, kdy tomu tak není, je např. definice procedury. Budeme tedy chtít, aby byl stav parseru uložen u uzlu `AST_BLOCK`, právě tehdy, když tento blok nebude současně příkazem. V opačném případě

bude uložen u odpovídajícího uzlu `AST_STATEMENT`. To proto, abychom neukládali stav parseru zbytečně dvakrát. Proto je pravidlo `plsql_block` rozšířeno o boolovský parametr `incremental`, který říká, zda se má provést uložení stavu. Pak již stačí upravit gramatiku tak, aby použití pravidla `plsql_block` obsahovala příslušné hodnoty toho parametru.

4.4 Testování korektnosti

Účelem testování korektnosti bylo ověřit skutečnost, že se inkrementální zpracování chová stejně jako původní řešení. K tomu byla použita sada mající celkem 31 PL/SQL skriptů. Tyto skripty měly velikost v rozmezí 612B až 80KB. Obsahovaly především nejrůznější DML příkazy, případně definice procedur či funkcí.

Dále bylo třeba vybrat sadu transformací pro testování vybraných skriptů. Celkem bylo vybráno 17 nejrůznějších transformací. Tyto transformace vycházely z již existujících transformací.

Samotné testování následně probíhalo tak, že se porovnal výsledek inkrementálního zpracování a výsledek zpracování pomocí původního postupu (tj. kompletního zpracování celého AST). Toto porovnání bylo provedeno pro všechny dvojice skript:transformace. Porovnání výsledků probíhalo na několika úrovních. Na první úrovni se ověřovalo to, zda oba výsledky neobsahují žádný chybový uzel (tj. uzel popisující chybu v syntaxi). Následovala druhá úroveň, kde se ověřovaly bílé znaky, respektive pozice. Tím je myšleno to, že oba výsledky mají totožné formátování. To se zjistí porovnáním hodnot v jednotlivých tokenech. Konečně v poslední úrovni se ověřoval výsledek sémantické analýzy.

Tato poslední úroveň byla nejsložitější. Porovnat výsledek sémantické analýzy totiž není jednoduchý úkol z několika důvodů. Jedním z nich je to, že výsledkem může být instance několika (řádově desítka) různých tříd. Navíc u testovaných skriptů nemáme kompletní datový slovník. To ve výsledku znamená, že je tento slovník budován průběžně. To má ale za následek to, že ani výsledek dvou po sobě jdoucích kompletních zpracování nemusí být stejný. Ukázáno na příkladu 19.

Příklad 19 Na tomto příkladu je ukázáno, jak se může lišit výsledek dvou kompletních zpracování v případě, kdy nemáme k dispozici kompletní datový slovník. Mějme následující skript:

```
1 SELECT * FROM tabulka WHERE a = 42;
2 SELECT * FROM tabulka WHERE b = 42;
```

Nechť nyní zpracováváme tento skript poprvé. Obsah slovníku je prázdný. Při zpracování prvního příkazu zjistíme, že tabulku `tabulka` nemáme ve slovníku, a proto vytvoříme nový záznam. Z klauzule `WHERE` dále zjistíme, že tato tabulka má sloupec `a`, a tedy že výsledek tohoto příkazu v podobě `*` obsahuje pouze sloupec `a`.

Při zpracování druhého `SELECT` příkazu navíc zjistíme, že tabulka se jménem `tabulka` navíc obsahuje i sloupec `b`. Výsledek vyjádřený pomocí `*` bude tedy obsahovat dva sloupce: `a` a `b`.

Teď začneme s novým kompletním zpracováním. V případě druhého příkazu `SELECT` bude výsledek stejný jako v předchozím případě, ovšem v prvním příkazu dostaneme jako výsledek dva sloupce (`a` a `b`), zatímco v prvním běhu jsme dostali pouze `a`. To je způsobeno tím, že při druhém běhu máme více informací. ♠

Na předchozím příkladu jsme si ukázali, jaký problém může nastat při opětovné analýze. Existuje zde ovšem i další varianta tohoto problému. Ta spočívá v tom, že během operace `resolve` se při nedostatku informací občas nepodaří určit, o co se vlastně jedná. Při opětovném zpracování se to již podařit může.

První problém je vyřešen tak, že nepožadujeme, aby dva výsledky byly stejné, ale aby jeden byl „podmnožinou“ druhého. Druhý problém je vyřešen tak, že dva výsledky porovnáváme právě tehdy, když byly určeny oba, nebo ani jeden z nich.

Během tohoto testování bylo objeveno několik chyb. Především se jednalo o nejrůznější nedostatečné ošetření případů, kdy daná proměnná může nabývat hodnoty `null`. Vyskytlo se i pár případů, kdy byla skupina funkcí volána ve špatném pořadí. Stejně tak byla objevena chyba v kódu samotné transformace, která již byla součástí původní implementace. Zde byla chyba v tom, že daný podstrom byl přesunut namísto toho, aby byl zkopírován.

Kapitola 5

Experimentální měření

V této kapitole je popsáno provedené experimentální měření, které mělo ověřit vliv inkrementální analýzy na čas běhu a paměťovou náročnost. Skládalo se ze třech částí. V první části je testováno, jaký vliv má granularita ukládaného stavu na celkový čas běhu prvotní syntaktické a sémantické analýzy. Jsou zde zvoleny tři kategorie tohoto ukládání: Vše, Příkaz a Nic. Ty odpovídají hodnotám popsaným v předchozí kapitole, konkrétně v podsekci 4.3.3. Druhou částí testování efektivity bylo ověření vlivu inkrementálního zpracování na čas strávený transformací. V poslední části byla věnována pozornost paměťové složitosti obou strategií detekce provedených změn.

Nejdříve byla vybrána vhodná sada transformací. Nakonec bylo použito těchto sedm transformací:

- převod z funkce **DECODE** na **CASE** výraz, viz příklad 8
- spojení sousedních literálů pro řetězec, viz příklad 9
- převod **ORDER BY** číslo sloupce na **ORDER BY** jméno sloupce, viz příklad 10
- sjednocení popisku PL/SQL bloku na jeho začátku a konci, viz příklad 13
- převod **IF** (...) **EXIT** na **EXIT WHEN** (...), viz příklad 15
- přidání implicitního klíčového slova **IN** u parametru procedury/funkce, viz příklad 12
- nastavení velikosti znaků klíčových slov, viz příklad 7

Pro každou z těchto transformací byla vybrána skupina několika skriptů, kde každý z těchto skriptů měl velikost v rozmezí 50 až 100 KB. Většina z těchto skriptů ovšem měla velikost přes 90KB.

Každé měření časové složitosti bylo opakováno 20×. Před samotným zahájením měření byla daná akce spuštěna „nanečisto“, rovněž 20×. To z toho

důvodu, abychom v rozumné míře zajistili, že veškeré inicializace jsou provedeny, případně že je daný bytecode zkompilován. Dále byla před zahájením každé měřené akce volána metoda `System.gc()`, čímž doporučíme garbage collectoru vhodný čas k úklidu. V případě testování transformací nebyla do času započítávána prvotní analýzu vstupního souboru.

Pro měření byla použita následující konfigurace:

- Procesor Intel Core i5 4670, 4 jádra, 3.4GHz, 6MB L3 cache
- 16GB RAM
- operační systém Linux, kernel verze 3.19.2
- OpenJDK, verze 1.7.u75_2.5.4

Odkazované tabulky vždy obsahují naměřené časy v milisekundách, případně velikost spotřebované paměti v KB. Dále za symbolem ± nalezneme rozptyl. Ve sloupcích je vždy výsledek pro jednu konfiguraci inkrementálního zpracování, popsáno konkrétněji později. V řádcích pak jsou výsledky pro jednotlivé vstupní soubory. Začneme s měřením dopadu granularity ukládání stavu parseru.

5.1 Granularita ukládaného stavu

Nejdříve si vysvětlíme, o co v tomto měření šlo. Cílem bylo zjistit, jak velký vliv má granularita ukládaného stavu na čas strávený analýzou. Bylo měřeno na šesti různých souborech se třemi hodnotami granularity. Hodnoty granularity byly popsány v předchozí kapitole, konkrétně v podsekci 4.3.3.

Výsledky jsou v tabulce 5.1, případně v grafu A.1. Vidíme zde, že mezi jednotlivými granularitami není žádný výrazný rozdíl. To je docela potěšující, jelikož budeme schopni inkrementálně zpracovat mnohem větší třídu změn, aniž bychom za to museli nějak výrazně platit.

Soubor	Nic	Příkaz	Vše
`1.sql`	686.865 ± 0.014	674.631 ± 0.019	700.556 ± 0.007
`2.sql`	909.683 ± 0.009	905.135 ± 0.009	911.761 ± 0.007
`3.sql`	816.954 ± 0.026	807.141 ± 0.024	815.759 ± 0.024
`4.sql`	1944.861 ± 0.019	1907.154 ± 0.016	1929.319 ± 0.017
`5.sql`	1464.860 ± 0.006	1466.836 ± 0.008	1452.356 ± 0.007
`6.sql`	1579.227 ± 0.010	1600.567 ± 0.014	1570.729 ± 0.016
`7.sql`	2034.477 ± 0.031	1979.560 ± 0.026	1956.149 ± 0.023

Tabulka 5.1: Časy v ms potřebné pro zpracování v závislosti na granularitě ukládaného stavu.

Výsledky měření vlivu granularity si lze vysvětlit tak, že velikosti jednotlivých zásobníků scope (viz sekce 1.3) nejsou příliš velké, a tedy operace zkopírování je poměrně levná.

5.2 Transformace

Tato část měření je zaměřena na vliv inkrementálního zpracování na vybrané transformace. Opět jsou měřeny všechny tři hodnoty granularity ukládání stavu. Navíc jsou změřeny i dvě různé implementace detekce změn: porovnávání (srovnání struktury AST před a po transformaci) a zaznamenávání (průběžné zaznamenávání provedených změn).

Výsledek pro transformaci spojující řetězcové literály je v tabulce 5.2, případně v grafu A.2. Vidíme, že jsme dosáhli poměrně výrazného zlepšení pro všechny testovací soubory. Velikost tohoto zrychlení ovšem vždy závisí na počtu výskytu provedených transformací. Čím více transformovaných částí AST, tím menší zrychlení. To ostatně platí pro všechny transformace.

Většina instancí vzorů, jež jsou touto transformací upravovány, způsobí nutnost zpracovat celý příkaz. Proto jsme zde neočekávali žádný výrazný vliv granularity. To se ve výsledku i potvrdilo, jelikož naměřené hodnoty jsou pro obě varianty velmi podobné.

Soubor	Nic	Příkaz (zaznam.)	Příkaz (porovnání)	Vše (zaznam.)	Vše (porovnání)
`1.sql`	1157.205 ± 0.019	28.690 ± 0.018	35.605 ± 0.010	28.804 ± 0.019	35.665 ± 0.015
`2.sql`	628.623 ± 0.019	54.113 ± 0.026	59.975 ± 0.023	54.223 ± 0.032	60.529 ± 0.024
`3.sql`	613.759 ± 0.010	151.214 ± 0.009	153.591 ± 0.016	152.390 ± 0.014	154.092 ± 0.017
`4.sql`	1371.057 ± 0.013	87.504 ± 0.029	97.509 ± 0.025	86.378 ± 0.007	97.121 ± 0.015
`5.sql`	1025.154 ± 0.015	22.970 ± 0.053	28.929 ± 0.026	22.760 ± 0.029	29.164 ± 0.020
`6.sql`	710.196 ± 0.011	235.113 ± 0.034	231.576 ± 0.015	224.254 ± 0.008	241.654 ± 0.021

Tabulka 5.2: Časy v ms potřebné pro transformace spojující řetězcové literály.

V tabulce 5.3, případně grafu A.3, je výsledek pro transformaci měnící **DECODE** na **CASE**. Obdobně jako v předchozím případě, ani zde nebyl očekáván výrazný vliv granularity ukládaného stavu. Co bylo překvapující, byl výsledek pro skript `2.sql`. Zde bylo dosažené zrychlení v porovnání k ostatním testovaným souborům relativně malé. Při bližší analýze bylo ovšem zjištěno, že je to zapříčiněno velkým počtem provedených transformací. Ač se tedy jedná

Soubor	Nic	Příkaz (zaznam.)	Příkaz (porovnání)	Vše (zaznam.)	Vše (porovnání)
`1.sql`	1651.830 ± 0.018	366.888 ± 0.009	361.777 ± 0.013	384.187 ± 0.064	360.635 ± 0.014
`2.sql`	2134.144 ± 0.011	1174.335 ± 0.023	1189.951 ± 0.029	1156.759 ± 0.034	1160.325 ± 0.033
`3.sql`	1405.194 ± 0.022	314.620 ± 0.013	325.592 ± 0.013	313.782 ± 0.014	333.734 ± 0.015
`4.sql`	1689.182 ± 0.020	284.551 ± 0.011	295.337 ± 0.012	194.057 ± 0.011	206.052 ± 0.067
`5.sql`	685.971 ± 0.009	76.921 ± 0.044	80.902 ± 0.018	77.840 ± 0.024	82.061 ± 0.019
`6.sql`	1281.213 ± 0.005	58.108 ± 0.040	65.323 ± 0.016	59.091 ± 0.033	65.391 ± 0.024
`7.sql`	628.721 ± 0.015	103.690 ± 0.016	108.878 ± 0.016	106.793 ± 0.015	109.380 ± 0.026

Tabulka 5.3: Časy v ms potřebné pro transformaci převodu funkce **DECODE** na **CASE** výraz.

Soubor	Nic	Příkaz (zaznam.)	Příkaz (porovnání)	Vše (zaznam.)	Vše (porovnání)
`1.sql`	1330.818 ± 0.012	157.184 ± 0.014	165.528 ± 0.019	17.671 ± 0.029	27.350 ± 0.019
`2.sql`	1977.640 ± 0.006	95.710 ± 0.014	105.263 ± 0.026	30.762 ± 0.032	40.482 ± 0.017
`3.sql`	1650.656 ± 0.011	195.195 ± 0.007	211.110 ± 0.013	23.325 ± 0.031	33.951 ± 0.077
`4.sql`	1568.294 ± 0.010	1634.571 ± 0.022	1606.600 ± 0.026	22.535 ± 0.024	33.366 ± 0.019
`5.sql`	1641.920 ± 0.007	12.904 ± 0.031	23.699 ± 0.016	12.835 ± 0.048	23.808 ± 0.016
`6.sql`	2154.523 ± 0.005	19.454 ± 0.040	34.570 ± 0.014	19.531 ± 0.022	34.230 ± 0.013

Tabulka 5.4: Časy v ms potřebné pro transformaci **ORDER BY** klauzule.

o inkrementální zpracování, musíme opětovně analyzovat více než třetinu původního skriptu.

Výsledek pro transformaci **ORDER BY** klauzule je v tabulce 5.4, popřípadě v grafu A.4. Zde se konečně projevil vliv granularity ukládaného stavu. Během této transformace totiž dochází ke změnám, které dokážeme efektivně inkrementálně zpracovat v případě, kdy máme uložený příslušný stav parseru. Ve výsledcích se to projevilo tak, že varianty označené jako Vše jsou zpravidla rychlejší, než varianty označené jako Příkaz.

Nejvýrazněji je to vidět na souboru `4.sql`. Zde je dokonce inkrementální zpracování označené jako Příkaz na úrovni původního algoritmu. Vysvětlení je jednoduché. V tomto souboru dojde k transformaci, která způsobí změnu definice kurzoru (tj. datového typu). Jelikož ve variantě Příkaz neumíme zpracovat změnu klauzule **ORDER BY** (nemáme totiž potřebný uložený stav), tak musíme postupovat konzervativně. To v tomto případě znamená *kompletní* analýzu skriptu. Naopak pokud máme příslušný stav uložen, je zpracování velmi rychlé.

Tabulka 5.5 a graf A.5 ukazují výsledky pro transformaci sjednocující popisek PL/SQL bloku. Dosažené výsledky jsou velmi podobné jako v prvních dvou případech. Vždy zde záleží na množství modifikací. Jedna modifikace zde znamená opětovnou analýzu celého bloku. Možná by zde stálo za úvahu, jak lze toto vylepšit. Pěkně je to vidět na souboru `5.sql`, kde byly tímto postupem opětovně analyzovány dva bloky, které ovšem co do počtu řádků původního skriptu představovaly pouhé jedno procento.

Soubor	Nic	Příkaz (zaznam.)	Příkaz (porovnání)	Vše (zaznam.)	Vše (porovnání)
`1.sql`	5840.586 ± 0.016	2196.452 ± 0.013	2318.970 ± 0.037	2211.974 ± 0.012	2187.801 ± 0.014
`2.sql`	8697.425 ± 0.008	2664.671 ± 0.018	2658.042 ± 0.011	2734.400 ± 0.015	2693.123 ± 0.015
`3.sql`	7341.820 ± 0.016	3144.655 ± 0.031	2897.243 ± 0.014	3020.748 ± 0.028	2814.953 ± 0.010
`4.sql`	7282.652 ± 0.012	4193.270 ± 0.013	4098.420 ± 0.012	4309.349 ± 0.011	4092.233 ± 0.014
`5.sql`	8784.765 ± 0.012	173.846 ± 0.041	234.487 ± 0.047	169.087 ± 0.048	231.467 ± 0.050

Tabulka 5.5: Časy v ms potřebné pro transformaci sjednocující popisek bloku.

Poslední výsledky pro transformace převádějící **IF THEN EXIT**, přidání klíčového slova **IN** a změna velikosti všech klíčových slov jsou v tabulce 5.6 a grafu A.6. Začneme se skriptem `if_exit_.sql`, na který byla použita transformace převádějící **IF THEN EXIT**. Už ze samotné definice transformace zde vychází, že případné inkrementální zpracování by mělo být velmi rychlé. To se ve výsledku také potvrdilo.

Zbylé dvě transformace jsou speciálního typu. Jejich výsledek totiž není nutné opětovně analyzovat. Musíme pouze opravit pozice a bílé znaky, viz sekce 2.6. Proto je zde patrné velmi výrazné zlepšení oproti původní implementaci.

Celkové zhodnocení je takové, že díky této implementaci inkrementálního zpracování, lze dosáhnout velmi výrazného ušetření výpočetního času. Jasně se ukázalo, že se vyplatí používat nejvyšší možnou granularitu ukládání stavu parseru.

Soubor	Nic	Příkaz (zaznam.)	Příkaz (porovnání)	Vše (zaznam.)	Vše (porovnání)
`if_exit_.sql`	1639.342 ± 0.016	56.946 ± 0.028	63.849 ± 0.032	53.704 ± 0.026	65.735 ± 0.046
`in_.sql`	1673.391 ± 0.010	48.411 ± 0.041	53.973 ± 0.047	47.518 ± 0.040	53.861 ± 0.047
`keyword_1.sql`	784.290 ± 0.013	28.109 ± 0.094	35.672 ± 0.099	28.368 ± 0.107	34.767 ± 0.101
`keyword_2.sql`	1037.684 ± 0.014	34.924 ± 0.076	39.758 ± 0.069	35.071 ± 0.075	41.003 ± 0.052
`keyword_3.sql`	921.265 ± 0.009	35.336 ± 0.085	41.207 ± 0.086	35.546 ± 0.097	41.669 ± 0.078
`keyword_4.sql`	2066.170 ± 0.011	66.222 ± 0.083	72.545 ± 0.081	65.073 ± 0.067	78.440 ± 0.010

Tabulka 5.6: Časy v ms potřebné pro transformace přidávající klíčové slovo **IN**, měnící velikost klíčových slov a transformace převádějící příkaz se vzorem **IF THEN EXIT**.

Při bližším prozkoumání naměřených výsledků zjistíme, že strategie zaznamenávání změn byla lepší ve všech provedených transformacích kromě jediné. Tou byla transformace sjednocující popisek PL/SQL bloku, především pro granularitu označenou jako *Vše*. V tomto případě vyšlo pro soubory `1.sql`, `2.sql`, `3.sql` a `4.sql` jako výhodnější použít strategii porovnání struktury. Rozdíly mezi oběma strategiemi ovšem nejsou nikterak výrazné, přesto se přikláníme spíše ke strategii sledování změn. Tato volba bude nadále podpořena i v následující sekci. Ta se bude zabývat experimentálním měřením paměťové složitosti.

5.3 Paměťová složitost detekce změn

Poslední sekce se věnuje paměťové složitosti obou implementovaných strategií pro detekci provedených změn. Těmi jsou porovnání struktury AST a zaznamenávání provedených změn pomocí rozšíření `OracleContextState`.

V rámci implementace byla krátce ověřena náročnost kopírování struktury AST pomocí třídy `TreeStructure`. Pro AST zabírající cca. 125 MB zabírala kopie AST pomocí `TreeStructure` zhruba 5 MB. U zaznamenávání změn je paměťová náročnost lineárně závislá se počtu provedených akcí, tj. přidání či odebrání uzlu.

Na základě toho lze tedy předpokládat, že strategie zaznamenávání změn by neměla mít prakticky žádný vliv na paměťovou náročnost, zatímco strategie porovnávající strukturu AST by měla mít vliv v řádu MB.

Měření probíhalo následovně. Nejdříve bylo určeno místo v programu, kde je nejvyšší očekávaná spotřeba paměti. Tím je řádek č.16 na zdrojovém kódu 2. V tomto místě byl pomocí nástroje `jmap` [22] proveden tzv. *heap*

Soubor	Původní implementace	Vše (zaznamenávání)	Vše (porovnání)
`1.sql`	71928	72009	73114
`2.sql`	63071	63269	64088
`3.sql`	61920	62630	63105
`4.sql`	79531	79746	81186
`5.sql`	70083	70254	71294
`6.sql`	70305	70768	71693

Tabulka 5.7: Paměťová náročnost v KB pro transformaci spojující řetězcové literály.

Soubor	Původní implementace	Vše (zaznamenávání)	Vše (porovnání)
`1.sql`	91791	92084	93773
`2.sql`	112867	113115	115668
`3.sql`	87000	87276	88816
`4.sql`	91165	91398	93094
`5.sql`	70650	70771	71697
`6.sql`	76997	77094	78276
`7.sql`	66406	66526	67341

Tabulka 5.8: Paměťová náročnost v KB pro transformaci převodu funkce **DECODE** na **CASE** výraz.

dump, neboli byl uložen obsah haldy na disk. Tento heap dump obsahoval *pouze živé* objekty (přepínač `-dump:live`). Velikost použité haldy byla poté určena nástrojem `jhat` [23]. Alternativě lze použít pouze nástroj `jmap` s přepínačem `-histo:live`. Výsledek měření není úplně exaktní. To je způsobeno dalšími běžícími vlákny procesu. Pozorované rozdíly ovšem byly zanedbatelné, tj. v řádu KB. Z tohoto důvodu bylo měření provedeno pouze jednou. Byla měřena pouze jedna hodnota granularity. Tou byla hodnota *Vše*. V tabulce jsou zaznamenány velikosti v KB.

Transformace sjednocující popisek bloku je složena z několika menších transformací. V rámci tohoto měření byla změřena pouze první z těchto malých transformací.

Výsledek pro transformaci spojující řetězcové literály je v tabulce 5.7. V tabulce 5.8 najdete výsledek pro transformaci převádějící funkci **DECODE** na **CASE** výraz. Výsledek pro transformaci **ORDER BY** klauzule je zachycen v tabulce 5.9. Částečný výsledek (viz předchozí odstavec) pro transformaci sjednocující popisek bloku je v tabulce 5.10. Konečně, v tabulce 5.11 jsou zachyceny výsledky pro transformaci **IF THEN EXIT**, klíčového slova **IN** v seznamu parametrů funkce/procedury a velikosti znaků klíčových slov.

Výsledky tohoto měření nejsou nikterak překvapivé. Strategie zaznamenávající provedené změny je prakticky stejně paměťově náročná jako původní

Soubor	Původní implementace	Vše (zaznamenávání)	Vše (porovnání)
`1.sql`	85018	85188	91731
`2.sql`	92211	92450	94103
`3.sql`	90977	91242	92924
`4.sql`	89814	90065	91710
`5.sql`	88636	88962	90710
`6.sql`	108959	109177	111758

Tabulka 5.9: Paměťová náročnost v KB pro transformaci **ORDER BY** klauzule.

Soubor	Původní implementace	Vše (zaznamenávání)	Vše (porovnání)
`1.sql`	100682	100887	102594
`2.sql`	103510	103713	105461
`3.sql`	105782	106047	107967
`4.sql`	106624	107000	108820
`5.sql`	124785	125066	127882

Tabulka 5.10: Paměťová náročnost v KB pro transformaci sjednocující popisek bloku.

Soubor	Původní implementace	Vše (zaznamenávání)	Vše (porovnání)
`in_.sql`	91163	91419	93105
`if_exit_.sql`	89818	90069	91735
`keyword_1.sql`	81079	81080	82325
`keyword_2.sql`	70830	70903	71974
`keyword_3.sql`	70185	70410	71424
`keyword_4.sql`	89359	89547	91209

Tabulka 5.11: Paměťová náročnost v KB pro transformaci přidávající klíčové slovo **IN**, transformaci měnící velikost klíčových slov a transformaci převádějící příkaz se vzorem **IF THEN EXIT**.

implementace. Rozdíly byly nejvýše v řadu malých stovek KB. Strategie porovnávající strukturu AST vyžadovala typicky o pár MB více než původní implementace.

Závěr

Tato práce se zabývala inkrementální analýzou PL/SQL skriptů. Každý z PL/SQL skriptů je reprezentován pomocí AST, který je výsledkem syntaktické analýzy, což bylo popsáno v sekci 1.2. Konkrétní forma AST byla naznačena v sekci 1.5.1. Nad AST se rovněž provádí sémantická analýza. Ta byla popsána v podsekci 1.5.2.

AST slouží k provádění tzv. transformací. Ty byly stručně popsány v podsekci 1.5.3. Jejich cílem je obecně *automatická* změna PL/SQL kódu na základě daných pravidel. Smyslem může být např. to, aby tento výsledný kód odpovídal příslušným normám. Jednoduchým příkladem je změna funkce **DECODE** na výraz typu **CASE**. Transformace si může definovat i sám koncový uživatel.

Tyto transformace ovšem přináší řadu problémů. Mezi jeden z nich patří neplatná reprezentace výsledného AST. Tímto jsou myšleny především tyto skutečnosti: a) Daná syntaktická konstrukce jazyka PL/SQL není reprezentována správným datovým typem. Platí totiž, že významný jazykový konstrukt by měl být reprezentován specializovaným datovým typem (pro ilustraci lze uvést např. **EXIT**, který by měl být reprezentován třídou `AstExit`). To po provedení transformace nemusí platit, jelikož tento konstrukt může být reprezentován pouze obecným typem uzlu. b) Během transformace může dojít k odstranění, či naopak přidání znaků. Vzhledem k tomu, že každý listový uzel AST obsahuje informaci o pozici v původním skriptu, může toto způsobit neplatnost těchto informací. c) Následkem transformace se může změnit sémantika (části) skriptu.

Původní řešení těchto problémů bylo takové, že se *celý* výsledný AST převedl na řetězec, a posléze se tento řetězec analyzoval celý znovu. To je zbytečné. V této práci bylo navrženo inkrementální řešení.

První kapitola práce se věnovala popisu teoretických základů a technologií. Kapitola 2 se věnovala analýze možného řešení. Ukázalo se, že řešení lze dekomponovat na čtyři postupné kroky: a) Detekce provedených změn; b) Určení, které části AST je nutné opětovně analyzovat; c) Opětovné analyzování určených částí AST; d) Oprava pozic listových uzlů.

V kroku označeném jako Detekce provedených změn bylo zvažováno několik variant, z nichž byly nadále implementovány dvě. Těmi byly a) porovnávání struktury AST před a po provedení transformace; b) průběžné zaznamenávání prováděných operací pomocí vzoru observer. Určení částí k opětovné analýze bylo navrženo pomocí systému predikátů. Ty lze chápat jako funkce, jež mají na vstupu provedenou změnu. Výstupem těchto funkcí je pak popis toho, jak má být tato změna ošetřena. Predikátů může být v systému několik a uvažujeme pouze nejlepší výsledek. Zbylé dva kroky implementace byly vesměs mechanické.

V kapitole 3 byly rozpracovány výsledky analýzy. Především zde byly navrhnuty konkrétní reprezentace datových typů. Rovněž zde najdeme příklad 18, na němž bylo ukázáno, jak by měla vypadat referenční implementace podpory inkrementálního zpracování. Předposlední kapitola č. 4 se věnovala implementačním detailům a testování korektnosti. Výsledky provedeného experimentální měření byly prezentovány v kapitole 5, případně v příloze A.

Z výsledků vyplývá, že dosažené zrychlení je velmi výrazné. V extrémních případech jsme pomocí inkrementálního zpracování mohli pouze opravit pozice u listových uzlů, zatímco původní implementace vyžadovala kompletní opětovnou analýzu skriptu. Stejně tak je patrné, že cena za časovou režii nutnou pro inkrementální zpracování není nikterak vysoká. Nebyl pozorován žádný výrazný vliv inkrementálního zpracování na paměťovou náročnost.

Na tuto práci lze navázat několika způsoby. Především implementací podpory inkrementálního zpracování dalších syntaktických konstruktů. V této implementaci jsme schopni inkrementálně zpracovat příkazy, PL/SQL bloky a některé dílčí části příkazů, např. **WHERE** podmínku. Zde by byl prostor pro mnohá vylepšení, především pro příkazy různé od příkazu **SELECT**. Pro tato vylepšení je veškerá infrastruktura připravena, jejich implementace by tak byla pouze otázkou času.

Velkou výzvou by byla podpora inkrementálního zpracování tzv. select listu. Tím je označována část **SELECT** příkazu, kde se definují výsledné sloupce. Zde nenastává problém při inkrementální syntaktické analýze, jež je jednoduchá, ale při analýze sémantické. Změna v této části AST se totiž během sémantické analýzy může projevit o několik předků *výše*, zatímco tato implementace je navržena tak, že je vždy potřeba analyzovat *celý* podstrom obsahující potenciálně ovlivněné uzly. Při úspěšné a efektivní implementaci by zde mohlo dojít k velkému zlepšení časové složitosti.

Stálo by také za zamyšlení, zda by se dalo současné rozhraní pro inkrementální zpracování upravit tak, aby podporovalo inkrementální zpracování pro skupinu transformací. Umožnili bychom tedy provést sérii transformací, aniž bychom zajišťovali konzistenci AST. Tu bychom zajistili až po provedení celé série. Transformace by si ovšem musely s případnou inkonzistencí poradit. Tímto bychom se mohli vyhnout situaci, kdy zajišťujeme konzistenci, aniž by to bylo nezbytně nutné.

Rovněž můžeme zobecnit řešení pro další databázové systémy.

Literatura

[1] *Manta Tools*. Profinit, citováno 14. 4. 2015. Dostupné z: `http://mantatools.com`

[2] Groff, J. R.; Weinberg, P. N.; Oppel, A. J.: *SQL: The Complete Reference*. McGraw-Hill, třetí vydání, 2009, ISBN 978-0071592550.

[3] *Oracle Database*. Oracle Corporation, citováno 14. 4. 2015. Dostupné z: `https://www.oracle.com/database/index.html`

[4] *Teradata Database*. Teradata Corporation, citováno 14. 4. 2015. Dostupné z: `http://www.teradata.com/products-and-services/Teradata-Database`

[5] Feuerstein, S.; Pribyl, B.: *Oracle PL/SQL Programming*. O'Reilly, 6 vydání, leden 2014, ISBN 978-1-449-32445-2.

[6] Chromsky, N.: Three models for the description of language. *Information Theory, IRE Transactions on*, ročník 2, č. 3, září 1956: s. 113–124, ISSN 0096-1000. Dostupné z: `http://www.chomsky.info/articles/195609--.pdf`

[7] Abo, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, druhé vydání, září 2006, ISBN 978-0321486813.

[8] Grune, D.; Jacobs, C. J.: *Parsing Techniques, A Practical Guide*. Springer, 2008, ISBN 978-0-387-20248-8.

[9] Cooper, K. D.; Torczon, L.: *Engineering a Compiler*, kapitola Parsing. Morgan Kaufmann, druhé vydání, 2012, ISBN 978-0-12-088478-0.

[10] Hopcroft, J. E.; Ullman, J. D.: *Introduction To Automata Theory, Languages, And Computation*, kapitola Pushdown Automata/110. Addison-Wesley, 1979, ISBN 0-201-02988-X.

[11] Parr, T.; Fisher, K. S.: LL(*): The Foundation of the ANTLR Parser Generator. In *PLDI*, 2011. Dostupné z: http://www.antlr.org/papers/LL-star-PLDI11.pdf

[12] Parr, T.: *Pragmatic Bookshelf - The Definitive ANTLR Reference: Building Domain-Specific Languages*. Dallas, Texas: The Pragmatic Bookshelf, první vydání, květen 2007.

[13] *ANTLR Parser Generator*. Citováno 30. 1. 2015. Dostupné z: http://www.antlr3.org/

[14] Robie, J.; Dyck, M.: XML Path Language (XPath) 3.1. Technická zpráva, W3C, 2014. Dostupné z: http://www.w3.org/TR/xpath-31/

[15] *Case and Decode and others*. Citováno 12. 3. 2015, naposledy upraveno: 1. 11. 2013. Dostupné z: https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:1243867216406

[16] Kaplan, H.: Persistent data structures. In *Handbook of Data Structures and Applications*, editace D. P. Mehta; S. Sahni, Chapman & Hall/CRC, první vydání, 2004, ISBN 1584884355. Dostupné z: http://www.cs.tau.ac.il/~haimk/papers/persistent-survey.ps

[17] Erich Gamma, R. J. J. V., Richard Helm: *Design Patterns: Elements of Reusable Object-Oriented Software*, kapitola Behavioral Patterns/Observer. Addison-Wesley, 39 vydání, březen 2009, ISBN 0-201-63361-2.

[18] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*, kapitola Structural Patterns/Proxy. Addison-Wesley, 39 vydání, březen 2009, ISBN 0-201-63361-2.

[19] Proxy. Technická zpráva, Oracle, 2014. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

[20] Trail: The Reflection API. Technická zpráva, Oracle, 2014. Dostupné z: http://docs.oracle.com/javase/tutorial/reflect/index.html

[21] Navarro, G.: A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, březen 2001: s. 31–88, ISSN 0360-0300. Dostupné z: http://swp.dcc.uchile.cl/TR/1999/TR_DCC-1999-005.pdf

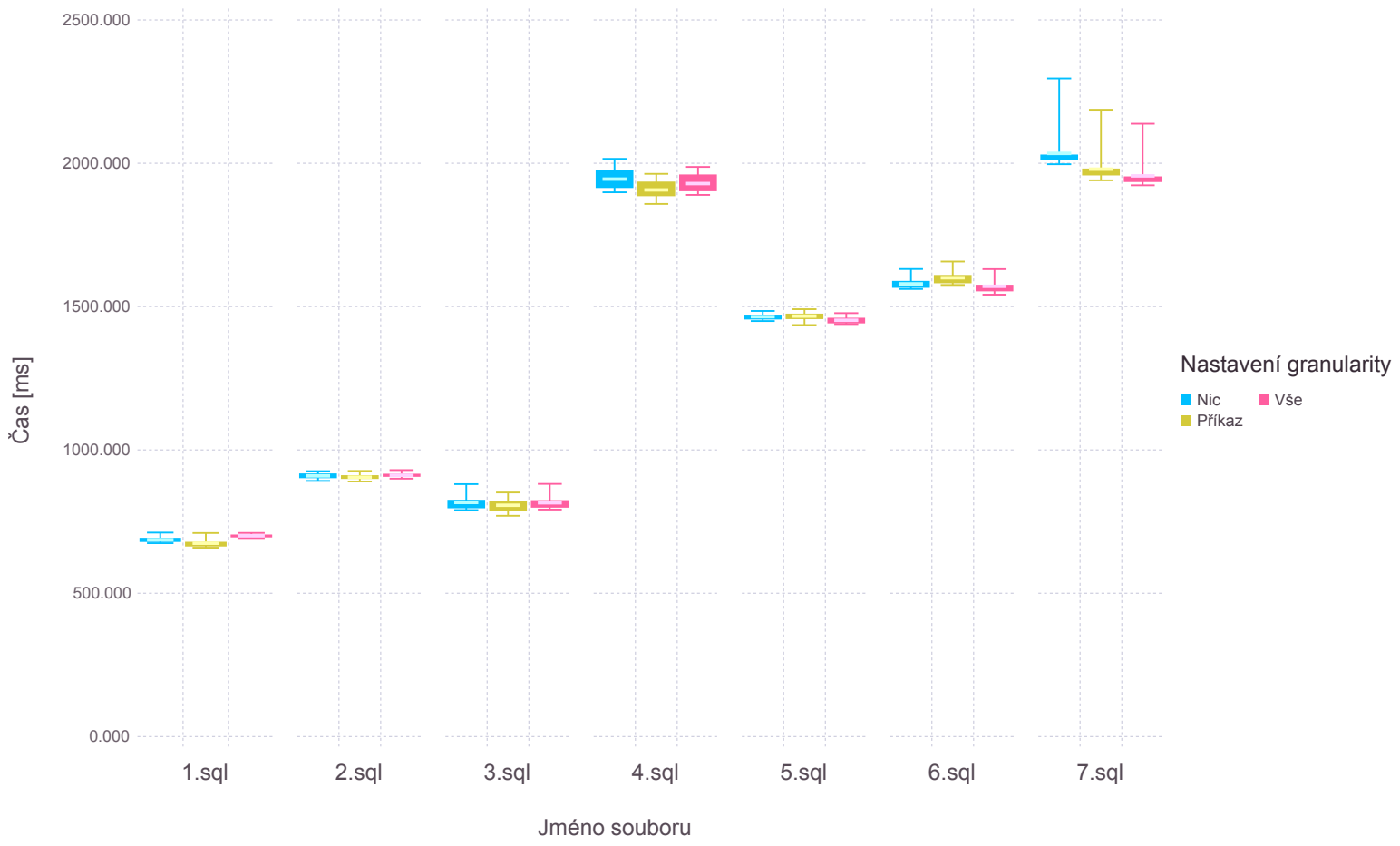
[22] *jmap - Memory Map*. Citováno 26. 4. 2015. Dostupné z: http://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html

[23] *jhat - Java Heap Analysis Tool*. Citováno 26. 4. 2015. Dostupné z: http://docs.oracle.com/javase/7/docs/technotes/tools/share/jhat.html

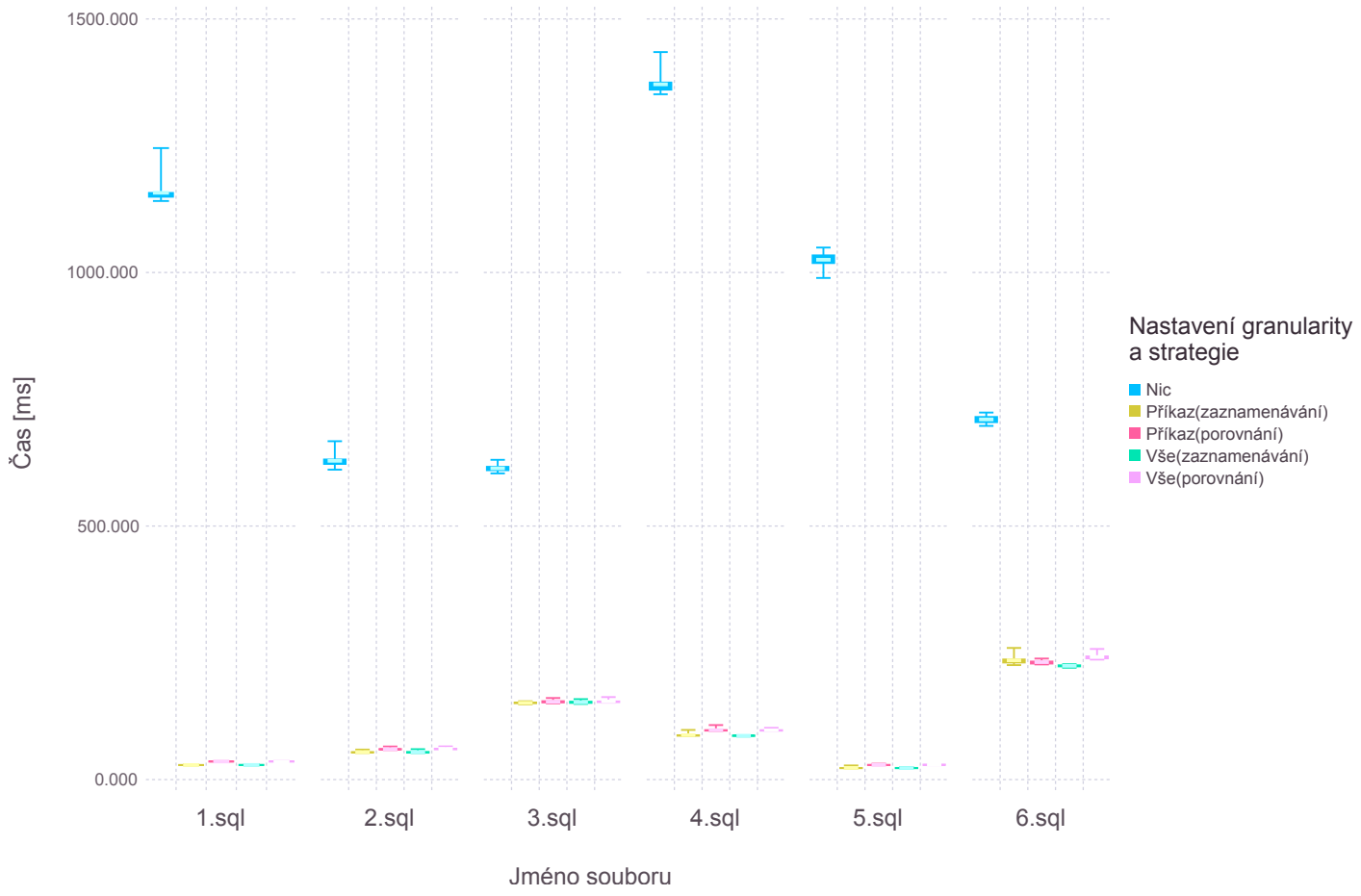
Příloha A

Grafy

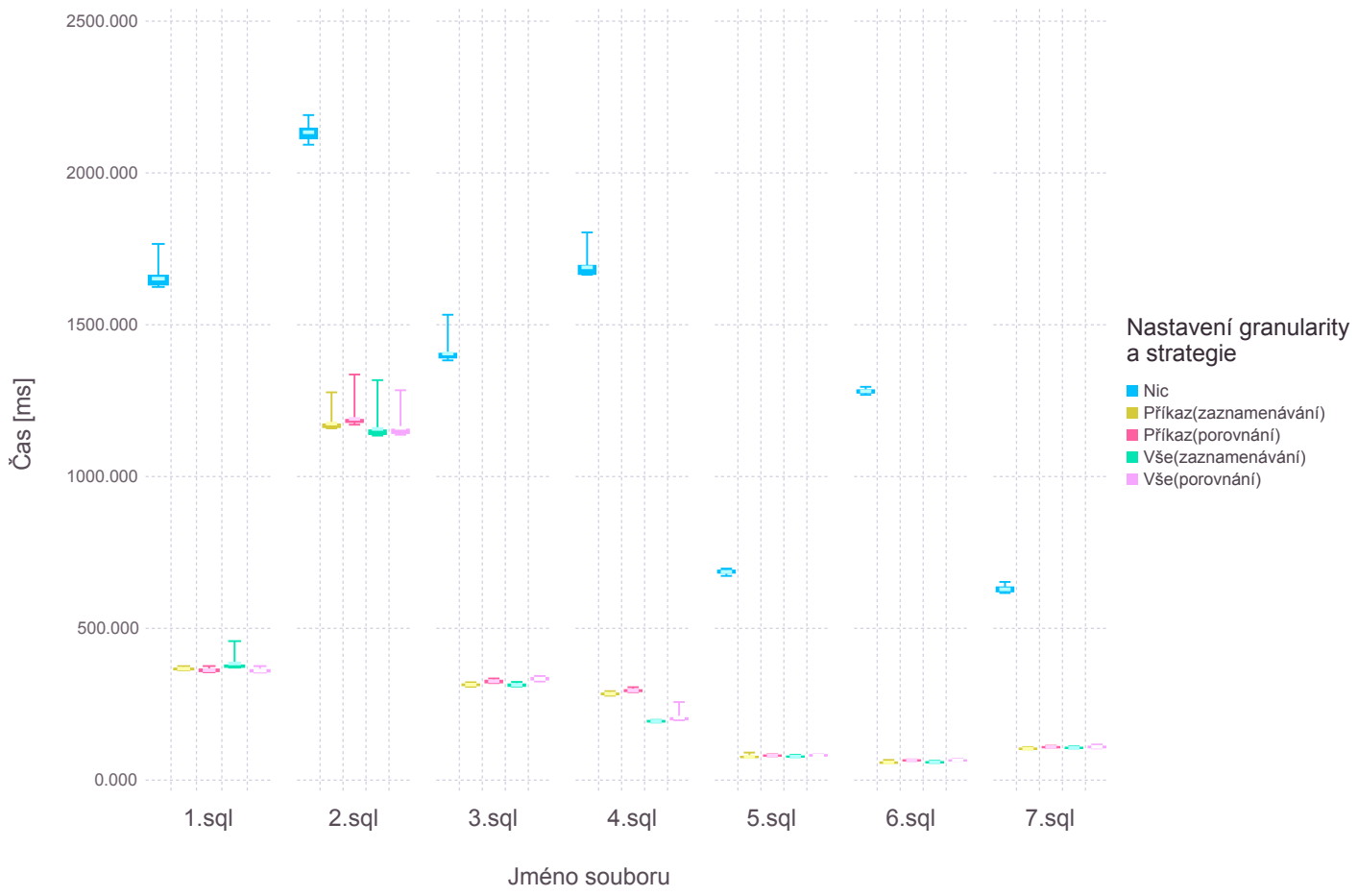
V této příloze jsou graficky znázorněny výsledky experimentálního měření, jak bylo diskutováno v kapitole 5.



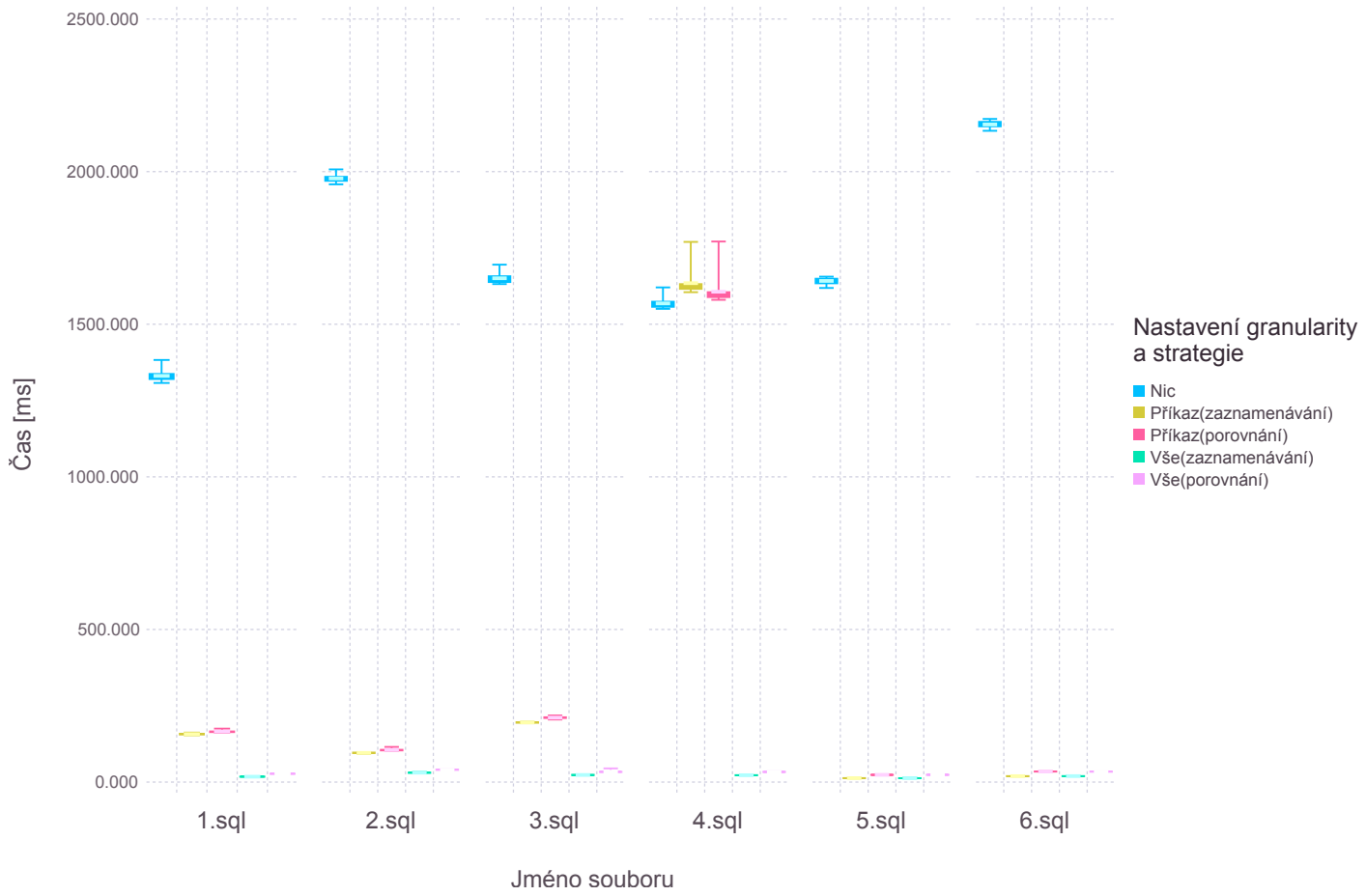
Graf A.1: Časy v ms potřebné pro zpracování PL/SQL souboru v závislosti na granularitě ukládaného stavu.



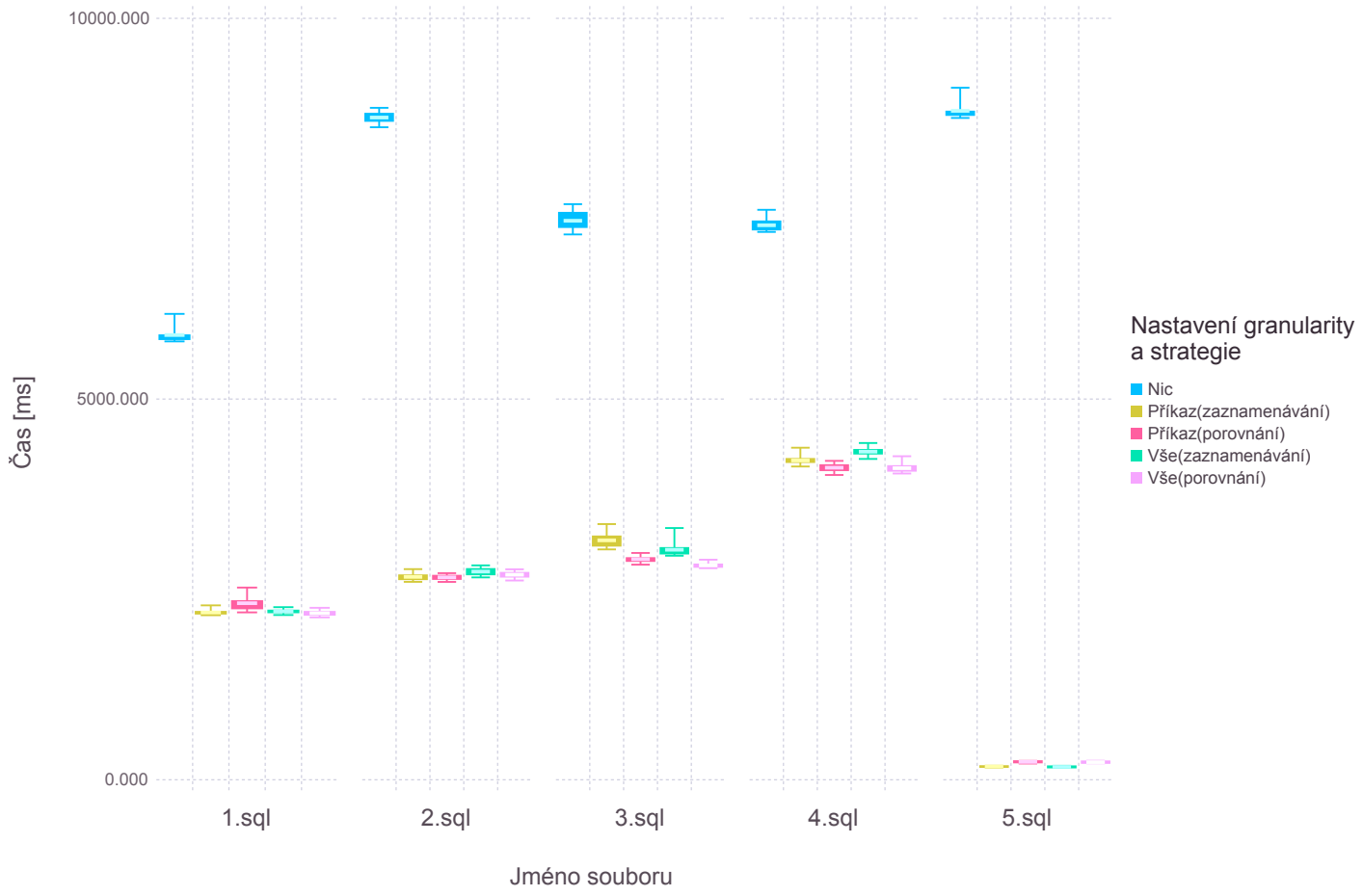
Graf A.2: Časy v ms potřebné pro transformaci spojující řetězcové literály.



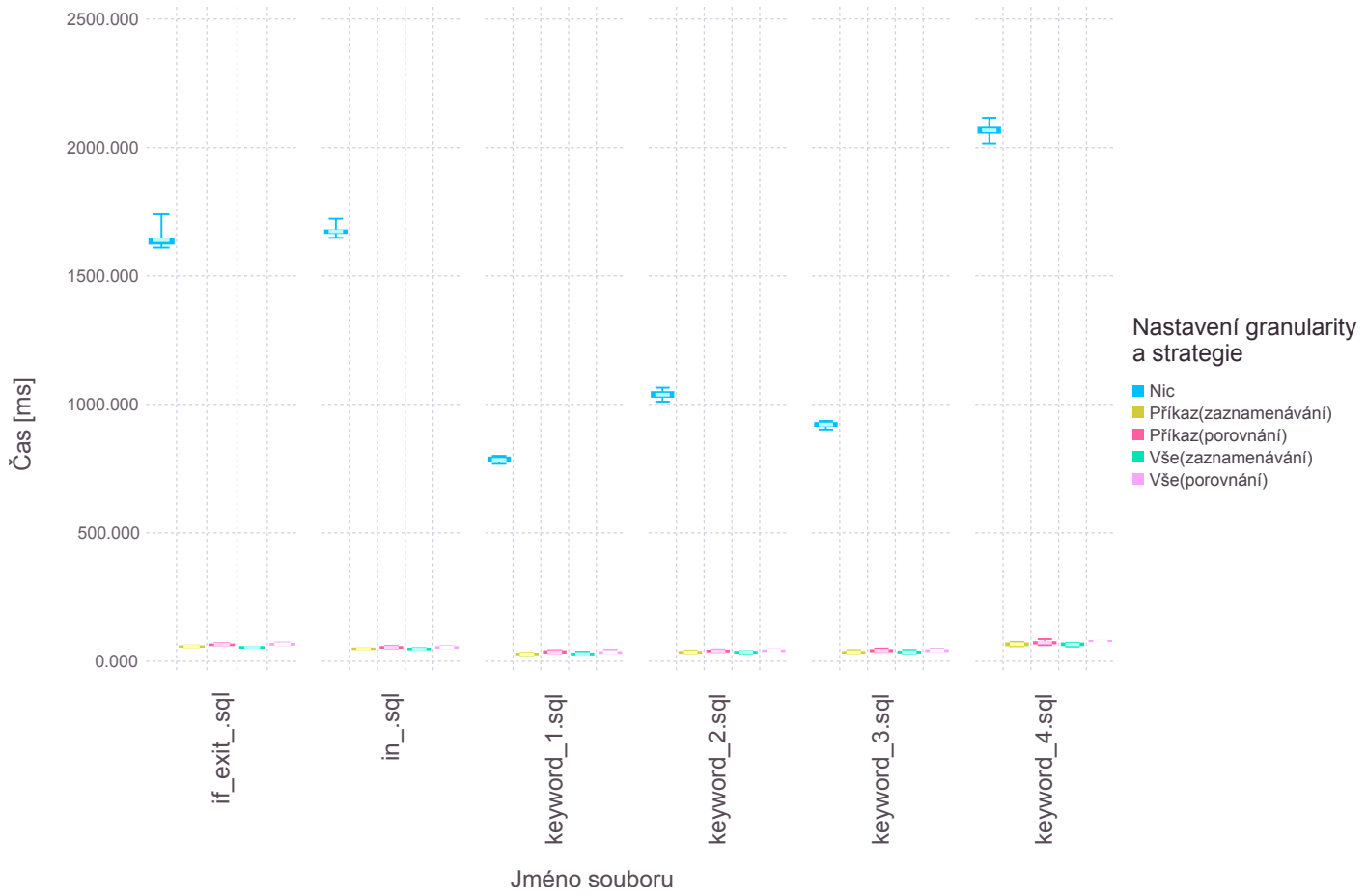
Graf A.3: Časy v ms potřebné pro transformaci převodu funkce **DECODE** na **CASE** výraz.



Graf A.4: Časy v ms potřebné pro transformaci **ORDER BY** klauzule.



Graf A.5: Časy v ms potřebné pro transformaci sjednocující popisek bloku.



Graf A.6: Časy v ms potřebné pro transformace přidávající klíčové slovo **IN**, měnící velikost klíčových slov a transformace převádějící vzor **IF THEN EXIT**.

Příloha **B**

Seznam použitých zkratek

AST Abstraktní syntaktický strom. ix, xi, xv, 1–3, 6, 11, 15–17, 22, 24, 25, 27, 29–32, 34, 35, 37–40, 42, 46–48, 53, 57, 60, 62–64

DDL Data definition language. 15

DML Data manipulation language. 12, 38, 53

DS Databázový systém. 1, 21

JVM Java Virtual Machine. 28

LV Levensteinova vzdálenost. 49

MT Manta Tools. ix, xi, 1, 2, 8, 16, 17, 19, 25, 47, 48

PL/SQL Procedural Language/Structured Query Language. ix, xi, 1–3, 6, 12, 14, 15, 17, 19, 21, 22, 29, 32–35, 53, 55, 59, 60, 63, 64, 69

SQL Structured Query Language. 1, 20

SVN Subversion. 47

Příloha C

Obsah přiloženého CD

```
graphs ..... PDF s grafy uvedenými v příloze A. Grafy
parse.csv ..... výsledky měření pro prvotní syntaktickou analýzu
transforms.csv ..... výsledky měření pro všechny transformace
memory.csv ..... výsledky měření paměťové složitosti
thesis ..... zdrojová forma práce ve formátu LaTeX
thesis.pdf ..... text práce ve formátu PDF
```