# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Rozšíření překladače relační algebry |
| **Student:** | Bc. Martin Kubiš |
| **Vedoucí:** | Ing. Jiří Hunka |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Webové a softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | Do konce letního semestru 2019/20 |

## Pokyny pro vypracování

Cílem této práce je navrhnout a vytvořit nový překladač z Relační algebry do SQL, který bude realizovat bezchybné překlady do všech níže uvedených databázových systémů.
Tento překladač bude integrován místo současného překladače do komplexního systému pro podporu výuky dbs.fit.cvut.cz v předmětu Databázové systémy (BI-DBS).

1. Analyzujte stávající řešení překladače a proveďte rešerši konkurence.
2. Na základě analýzy navrhněte a implementujte překladač, který bude realizovat překlady z Relační algebry do SQL pro databáze Oracle, PostgreSQL, MariaDB (MySQL).
3. Zabývejte se analýzou a řešením problému použití tečkové notace. Dále uvažujte další možnosti zlepšení.
4. Ve své práci vhodně aplikujte své nabyté znalosti ohledně návrhových vzorů.
5. Vytvořte API, přes které bude aplikace přístupná dalším komponentám portálu do kterého ji integrujete.
6. Překladač řádně otestujte a zdokumentujte.
7. Klaďte důraz na rozšiřitelnost překladače s ohledem na možné využití i jinými univerzitami.

## Seznam odborné literatury

Dodá vedoucí práce.

| | |
|---|---|
| Ing. Michal Valenta, Ph.D.<br>vedoucí katedry | doc. RNDr. Ing. Marcel Jiřina, Ph.D.<br>děkan |

V Praze dne 13. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

# Rozšíření překladače relační algebry

***Bc. Martin Kubiš***

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Hunka

30. dubna 2019

# Poděkování

Tímto chci poděkovat Ing. Jiřímu Hunkovi za poskytnutí možnosti pokračovat ve své bakalářské práci, stejně tak i za příkladné vedení a poskytování cenných rad při tvorbě této diplomové práce. Dále děkuji Ing. Michalu Valentovi, Ph.D. za poskytnutí písemných materiálů, jejichž je sám autorem. V neposlední řadě patří mé díky i Bc. Oldřichu Malcovi, za technickou a projektovou podporu při integraci praktické části do webového portálu Databázové systémy, a stejně tak i Ing. Tomáši Nováčkovi za jazykovou korekturu.

# 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. § 46 odst. 6 tohoto zákona 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). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 30. dubna 2019 . . . . . . . . . . . . . . . . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Martin Kubiš. 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 a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Kubiš, Martin. *Rozšíření překladače relační algebry.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

# Abstrakt

Cílem této diplomové práce bylo rozšířit existující překladač z relační algebry do SQL a odstranit jeho nedostatky, které vyvstaly při jeho několikaletém používání. Tato práce volně navazuje na mou bakalářskou práci Překladač z relační algebry do SQL [1], kterou jsem úspěšně obhájil v květnu 2016.

V rámci diplomové práce jsem se zabýval rešerší konkurenčních řešení, které v době psaní předchozí práce ještě nebyly k dispozici. Dále jsem provedl analýzu současného mnou vyvinutého překladače, který se aktuálně v době psaní této práce používá v rámci podpory výuky v předmětu Databázové systémy, zde na Fakultě informačních technologií Českého vysokého učení technického v Praze.

Posléze jsem navrhl a implementoval řešení nové, které již zmíněné nedostatky původního překladače odstraňuje, a jež přidává další užitečné funkcionality. Výsledek této práce zahrnuje mimo jiné i funkcionality, které byly specifikované pro původní bakalářskou práci, a u kterých se následně dospělo k závěru, že byly nad rámec rozsahu bakalářské práce.

V závěru této práce zmiňuji další možnosti, a případně i postupy, jak překladač obohatit o další užitečné funkce, zejména o rozpoznávání původní syntaxe zápisu relační algebry.

**Klíčová slova** překladač, relační algebra, SQL, Oracle, PostgreSQL, MariaDB, softwarové inženýrství, návrhové vzory, Python, PLY, Flask

# Abstract

The objective of this diploma thesis was to the extend existing Relational Algebra translator and to mitigate its flaws. This thesis loosely follows my Translator from Relational Algebra to SQL [1] bachelor thesis which I published back in May 2016.

An analysis of existing solutions, which were not present while I was writing the bachelor thesis, was made. Subsequently, I made an analysis of the existing solution which is currently being used as a supportive educational tool within the Database Systems course, here, at Faculty of Information Technology at the Czech Technical University in Prague. The previous solution was the outcome of the bachelor thesis mentioned above.

Additionally, I proposed and implemented a new solution that does not only mitigate the origin translator's flaws but also adds additional features and functionalities. The list of additional features includes requirements that were previously specified within the bachelor thesis. However, those requirements have not been met because they have exceeded the bachelor thesis' scope.

Finally, at the end of this thesis, I have described and partially analyzed a few potential concepts for future development, such as an origin syntax recognition.

**Keywords** translator, relational algebra, SQL, Oracle, PostgreSQL, MariaDB, software engineering, design patterns, Python, PLY, Flask

# Obsah

**Úvod** **1**

**1 Cíl práce** **3**

**2 Analýza** **5**
2.1 Rešerše konkurence . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Analýza současného překladače . . . . . . . . . . . . . . . . . . . 9

**3 Návrh** **17**
3.1 Definice požadavků . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 Návrhové vzory . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**4 Realizace** **23**
4.1 Koncepty jazyka Python . . . . . . . . . . . . . . . . . . . . . . 24
4.2 Praktické využití návrhových vzorů . . . . . . . . . . . . . . . . 26
4.3 Definice databázového schématu . . . . . . . . . . . . . . . . . . 29
4.4 Definice souhrnného příkladu . . . . . . . . . . . . . . . . . . . . 30
4.5 Konfigurace překladače . . . . . . . . . . . . . . . . . . . . . . . 31
4.6 Logování . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.7 Lexikální analýza . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.8 Syntaktická analýza . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.9 Transformace AST . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.10 Tvorba překladu . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.11 Shrnutí . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.12 Testování . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.13 Měření výkonu . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.14 Rozhraní překladače – API . . . . . . . . . . . . . . . . . . . . 64
4.15 Dokumentace překladače . . . . . . . . . . . . . . . . . . . . . . 65
4.16 Nasazení překladače . . . . . . . . . . . . . . . . . . . . . . . . 66

**5 Další funkcionality** **69**
5.1 Rozšíření chybových hlášení . . . . . . . . . . . . . . . . . . . . . 69
5.2 Optimalizace SQL výstupu . . . . . . . . . . . . . . . . . . . . . 70
5.3 Podpora dalších operací relační algebry . . . . . . . . . . . . . 73
5.4 Deklarace relací . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.5 Vyhodnocovací stromy dotazu v relační algebře . . . . . . . . . 76

**6 Možnosti budoucího rozvoje** **79**
6.1 Psaní komentářů do dotazu v relační algebře . . . . . . . . . . 79
6.2 Rozšíření selekce o porovnání IS (NOT) NULL . . . . . . . . . 80
6.3 Rozpoznávání další synaxe relační algebry . . . . . . . . . . . . 80

**Závěr** **85**

**Literatura** **87**

**A Seznam použitých zkratek** **91**

**B Obsah přiloženého CD** **93**

**C Obsah GIT repozitáře** **95**

**D Seznam rozpoznatelných tokenů** **97**

**E Kompletní definice gramatických pravidel** **101**

# Seznam obrázků

2.1 Příklad výsledku pro zadaný dotaz v rámci nástroje RelaX . . . . 7

3.1 Class diagram návrhového vzoru Singleton . . . . . . . . . . . . . . . 19
3.2 Class diagram návrhového vzoru Visitor . . . . . . . . . . . . . . . . 20
3.3 Class diagram návrhového vzoru FactoryMethod . . . . . . . . . . . 21

4.1 Příklad vícenásobné dědičnosti . . . . . . . . . . . . . . . . . . . . . 24
4.2 Aplikace návrhového vzoru Visitor . . . . . . . . . . . . . . . . . . . 27
4.3 Class diagram lexikálního analyzátoru . . . . . . . . . . . . . . . . . 32
4.4 Class diagram syntaktického analyzátoru . . . . . . . . . . . . . . . 35
4.5 Odpovídající AST sestrojený na základě dotazu D4 . . . . . . . . . 37
4.6 Class diagram pro implementaci TransformationVisitora . . . . . . 38
4.7 Class diagram zachycující veškeré typy sloupců . . . . . . . . . . . 39
4.8 Návrh tříd pro modelování dotazu `SELECT` jazyka SQL . . . . . . . 45
4.9 Graf průměrných hodnot pro vyřízení daného požadavku překladačem v závislosti na cílovém prostředí . . . . . . . . . . . . . . . . . . . . 63

# Seznam tabulek

4.1 Výsledek po spuštění dotazu SELECT z překladu P10 . . . . . . . 59
4.2 Průměrné časy pro vyřízení požadavku (# – č. testu, R – počet relací, D – počet deklarovaných relací, Z – změna priority, $T^{loc}_{avg}$ – prům. čas na lokálním prostředí, $T^{prod}_{avg}$ – prům. čas na produkčním prostředí) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.3 Maximální a minimální absolutní časy pro vyřízení požadavku . . 64

6.1 Priority vyhodnocování operací relační algebry převzaté z dokumentace nástroje RelaX . . . . . . . . . . . . . . . . . . . . . . . . 83

# Úvod

Úkolem překladače z relační algebry do SQL (Relational Algebra Translator, zkráceně RAT) je přijmout vstupní větu zapsanou v relační algebře, tuto větu zanalyzovat a převést ji do odpovídajícího vyjádření pomocí jazyka SQL.

Relační algebra je mocným prostředkem relačního modelu dat pro práci s daty a stala se teoretickým základem pro návrh jazyka SQL [2]. Vazba mezi relační algebrou a SQL je tedy zřejmá, převádět jeden jazyk na druhý proto dává smysl. Nicméně jazyk SQL je daleko mocnější než relační algebra. Ne všechny jeho konstrukty jsou totiž relační algebrou vyjádřitelné. Dále pak v souvislosti s jazykem SQL nemluvíme o relacích, nýbrž o tabulkách. O relacích lze analogicky mluvit jako o tabulkách, nicméně je potřeba si uvědomit, že mezi těmito dvěma pojmy jsou drobné nuance. Relace, na rozdíl od tabulky, je množina. V relaci tudíž nezáleží na pořadí n-tic a žádná n-tice se v relaci nevyskytuje vícekrát.

Problematikou překladu z relační algebry do SQL jsem se zabýval v rámci své bakalářské práce, ze které vzešel překladač, který se aktuálně používá jako nástroj (služba) pro podporu výuky v rámci předmětu Databázové systémy, zde na Fakultě informačních technologií Českého vysokého učení technického v Praze [1].

Při zpětném vyhodnocení překladače jsme s Ing. Jiřím Hunkou definovali upravující a rozšiřující požadavky, které se posléze staly podkladem této diplomové práce.

Kapitola 1

# Cíl práce

Cílem této diplomové práce je provést analýzu současného řešení překladače, který vzešel z mé závěrečné práce před třemi lety. Na základě požadavků definovaných níže buď současné řešení upravit, či zhotovit řešení nové, které těmto požadavkům vyhoví.

Dále je potřeba provést analýzu konkurenčních řešení, které se zabývají stejnou problematikou, a zhodnotit, v čem jsou, oproti současnému překladači používaném v rámci portálu Databázové systémy, lepší, a v čem se naopak odlišují.

Řešení, které bude výsledkem této práce, musí zachovat rozsah, který na něj byl kladen již v rámci bakalářské práce:

- Překladač musí podporovat korektní překlad do SQL spustitelného v RDBMS Oracle.
- Součástí implementace překladače musí být i sada testů pro ověření funkčnosti a korektnosti překladu.
- Překladač musí být řádně zdokumentován.
- Jeho součástí musí být i rozhraní (API), přes které bude překladač přístupný dalším aplikacím – toto rozhraní musí opět být řádně zdokumentováno.
- Řešení musí být opět integrováno do portálu pro podporu výuky v rámci předmětu Databázové systémy (BI-DBS) (dále jen Portál).

V rámci diplomové práce byly pro překladač z relační algebry do SQL definovány další nutné požadavky:

- Překladač musí být rozšířen o korektní překlady do SQL spustitelného v RDBMS PostgreSQL a MariaDB (MySQL).
- Překladač musí správně pracovat s tzv. tečkovou notací (viz dále).

- V práci musí být aplikovány nabyté znalosti ohledně návrhových vzorů.
- Výsledné řešení musí být připravené tak, aby jej mohly využít případně i další univerzity.
- Volitelně uvažujte a případně implementujte další rozšíření, které práci s překladačem usnadní.

Podpora samotných překladů do RDBMS PostgreSQL a MariaDB se stala stěžejní úlohou této práce, neboť tyto požadavky se mi v předchozí práci z důvodu vysoké náročnosti naplnit nepodařilo.

Kapitola 2

# Analýza

Než se pustíme do analýzy současného řešení překladače je vhodné udělat rešerši konkurenčních řešení. Níže se zmíním pouze o řešeních, která v době psaní bakalářské práce nebyla veřejná, nebo je moje původní práce nezmiňuje.

V této části se poprvé objevují formulace dotazů relační algebry, překlady těchto dotazů do SQL a ukázky zdrojového kódu překladače. Pro jednoduchost, a zejména přehlednost, budu jednotlivé ukázky nebo formulace v dalším textu patřičně odlišovat. Formulace dotazů relační algebry, překlady do SQL, úryvky ze zdrojového kódu překladače a výstupy z příkazové řádky budou dále v textu označovány písmeny „D“, „P“, „U“ respektive „V“, a číslem určující jejich pořadí (například D1, P4, U7, V10 apod.).

## 2.1 Rešerše konkurence

Pro rešeršní část této práce byly na konkurenční řešení kladeny následující minimální nároky:

- Musí se jednat o webovou aplikaci či aplikaci přístupnou skrze API.
- Daná aplikace musí definovat operace relační algebry alespoň v obdobném rozsahu, jaký je prezentován v předmětu Databázové systémy [3].

Při hledání podkladů pro vypracování této části jsem narazil na několik dalších konkurenčních nástrojů (vyjma těch, které jsem již popsal ve své bakalářské práci [1]) pracujících s relační algebrou.

### 2.1.1 RAT

Jedním z nich je nástroj se shodným názvem (RAT), který zaštiťuje Národní univerzita v Kostarice [4]. Dle webové prezentace se jedná o desktopovou aplikaci, která je dostupná pouze uživatelům používající MS Windows. Na základě velmi omezených zdrojů (oficiální manuál je napsaný ve španělštině), zejména

s přispěním podpůrných obrázků, bylo zjištěno, že právě rozebíraná aplikace RAT například neimplementuje přímo operace relační dělení, ani operaci anti join [5]. Tyto operace a například operace obecného spojení jsou ovšem vyjádřitelné pouze pomocí jiných, již implementovaných, operací. Například operace spojení je v aplikaci RAT vyjádřitelná pomocí operace kartézského součinu a selekce. Zejména z důvodu porušení prvního kritéria jsem aplikaci RAT nepodroboval samostatnému a podrobnějšímu zkoumání.

### 2.1.2 Pireal

Dalším konkurenčním řešením je Pireal. V tomto případě se jedná o multiplatformní, avšak stále desktopovou, aplikaci napsanou v jazyce Python. Dle oficiálního repozitáře aplikace Pireal nepodporuje operaci přejmenování a ani přímé použití operace relačního dělení. Při detailnějším zkoumání repozitáře se ukázalo, že aplikace, dle gramatiky, nejspíše nepodporuje ani negace logických podmínek, neprovádí překlad do SQL a gramatika použitá pro parsování relační algebry je nejednoznačná [6]. Na základě uvedených zjištění tuto aplikaci dále nebudu podrobněji rozebírat.

Jedinou aplikací, která splňuje daná kritéria je aplikace Relational Algebra Executor (zkr. RelaX). Na vlastnosti a specifika této aplikace se podíváme v následující podsekci.

### 2.1.3 RelaX

RelaX je označován jako kalkulátor relační algebry, jež vznikl v rámci bakalářské práce Johannese Kesslera BSc. na Univerzitě v Innsbrucku. Tento kalkulátor je zakomponován do stejnojmenného webového portálu. Vznik této práce se datuje již k roku 2013. Samotná práce pak byla dokončena až o rok později, v roce 2014 [7]. Nicméně při bližším ohledání veřejného repozitáře jsem dospěl k závěru, že aplikace RelaX musela být původně čistě interní záležitostí záležitostí Univerzity v Innsbrucku, neboť první commit ve veřejném repozitáři se objevuje až ke dni 3. listopadu 2015. Bohužel první verzí, která se v repozitáři objevuje, je až verze 0.18, první známá verze je však verze 0.8 [8]. Navíc se jedná až o upravenou verzi původní aplikace, která byla napsána v jazyce ES5 a jQuery. Verze, která je dnes veřejně přístupná, byla později dalšími studenty přepsána pomocí technologií React a TypeScript. Historicky nelze tedy dohledat, čemu všemu se výše zmíněnému studentovi podařilo dosáhnout, a co už je práce dalších studentů.

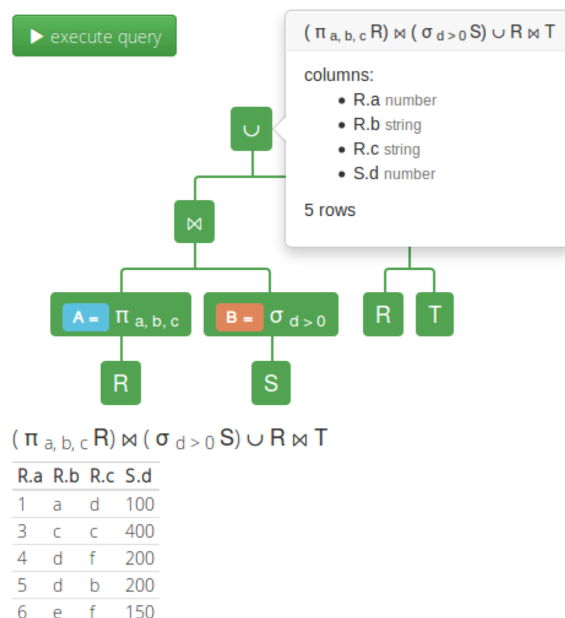
Samotný RelaX portál je ve své podstatě první veřejný edukativní webový portál, na němž si zájemci mohou osvojit syntaxi a koncepci relační algebry velmi příjemnou a nenásilnou formou.

Hned na úvodní stránce je připravený demo příklad zapsaný v relační algebře, který lze spustit a obratem zobrazí výsledek daného dotazu. Dále je zde předpřipravený průvodce, který provede uživatele krok za krokem a

ukáže mu, jak s daným nástrojem pracovat, a odhalí jeho veškeré stěžejní funkcionality. Mezi tyto funkcionality zejména patří:

- Definice dotazů v relační algebře, jejíž syntaxe vychází z definici použitých v publikacích Datenbanksysteme – Eine Einführung a Database Systems – The Complete Book [9].
- Možnost využití alternativní syntaxe pro zápis relační algebry.
- Využití stávajících / import nových datasetů, jež slouží pro vyhodnocování relační algebry.
- Zobrazení výsledku dotazu formou prováděcího stromu.
- Podpora překladu jednoduchých SQL dotazů do relační algebry a zobrazení výsledku formou zmíněnou výše.

Na obrázku 2.1 níže je pak zobrazený dotaz vytvořený v relační algebře, prováděcí strom samotného dotazu a v neposlední řadě i jeho výsledek:




Obrázek 2.1: Příklad výsledku pro zadaný dotaz v rámci nástroje RelaX

#### 2.1.3.1 Výhody

Mezi hlavní přednosti portálu RelaX patří zejména připravený průvodce, který seznámí uživatele s chováním a fungováním portálu a jeho kalkulátoru, dále pak automatická detekce lexikálních a syntaktických chyb v reálném čase a také pěkná vizualizace dotazu relační algebry pomocí vyhodnocovacího stromu.

Samotný kalkulátor pak implementuje některé další užitečné operace, jako například přejmenování relace, definice vlastních relací (proměnných) a také (přirozená) vnější spojení. Na druhou stranu ale kalkulátor implementuje také jisté rysy jazyka SQL, například používání typických SQL funkcí typu `upper()`, `coalesce()` a dalších, nebo například operátor `LIKE` či podmíněný výraz `CASE`. Dle mého osobního názoru dané operace do relační algebry nepatří.

Oceňuji též možnost použití alternativní syntaxe, čili přepis řeckých a dalších symbolů pomocí jejich alfanumerické reprezentace, například projekci označenou řeckým písmenem $\pi$ lze alternativně zapsat pouze jako `pi` a znak $\times$ vyjádřit alternativně pomocí slov `cross join`. Toto chování je velmi přínosné, neboť syntaxe, se kterou kalkulátor pracuje, se opírá o spoustu symbolů, nejen řecké abecedy, ale i dalších, které se nedají snadno napsat z klávesnice.

#### 2.1.3.2 Nevýhody

Mezi drobné nedostatky bych zařadil senzitivitu na velká a malá písmena, kdy je uživatel nucen dodržet předdefinovanou konvenci, tj. názvy relací velkými písmeny a názvy sloupců malými písmeny. Dále pak kalkulátor nerozeznává operaci pravého přirozeného a ani obecných anti joinů.

#### 2.1.3.3 Srovnání

Pokud srovnáme současný překladač (RAT), který jsem vytvořil v rámci své bakalářské práce, a kalkulátor RelaX, jsou zde patrné rozdíly. Za prvé, nejznatelnějším rozdílem je použití odlišné syntaxe pro zápis relační algebry. Záměrně jsem tento rozdíl nepsal do sekce 2.1.3.2, neboť se nejedná o chybu, ale pouze o odlišný pohled na danou problematiku. Neexistuje totiž standard, který by předepisoval zápis (syntaxi) relační algebry obdobně tak, jak je tomu v případě jazyka SQL. Tudíž způsoby zápisu relační algebry a i množina jejích operací se mohou lišit publikace od publikace a autor od autora. Stejně tomu je i v případě relační algebry vyučované na FIT ČVUT. Ta přebírá syntaktickou definici použitou v knize Databázové systémy, jejímž spoluautorem je garant stejnojmenného předmětu pan Ing. Michal Valenta, Ph.D. [2].

Druhým zásadním rozdílem je chování kalkulátoru RelaX. Nejedná se totiž o překladač relační algebry v tradičním slova smyslu, neboť při zadání dotazu v relační algebře se neprovádí překlad do žádného cílového jazyka, pouze se daný dotaz vizualizuje pomocí prováděcího / vyhodnocovacího stromu dotazu.

Jinak je tomu v případě zápisu dotazu v SQL. V tomto případě se překlad již provádí, protože výsledkem dotazu zapsaného v SQL je pak odpovídající dotaz v relační algebře a jeho prováděcí / vyhodnocovací strom. Tudíž, pokud nazveme RelaX překladačem, pak se jedná o překladač, který překládá směrem opačným, tedy z jazyka SQL do relační algebry.

Posledním rozdílem je odlišná koncepce RelaXu a RATu. RelaX je kompletní webový portál, který v sobě implementuje stejnojmenný kalkulátor, a je dále nedělitelný. Kdežto RAT vznikl jako samostatná komponenta, která je integrovatelná do Portálu a která je přístupná skrze API. Z čehož vyplývá, že RAT je komponentou, která je potencionálně využitelná vícero aplikacemi.

Nicméně frontend aplikace RelaX implementuje řadu zajímavých funkcí, u kterých by se vývojáři našeho interního Portálu (s přispěním nové verze RATu) mohli inspirovat a následně Portál těmito funkcionalitami obohatit. O jedné takové možnosti se podrobněji zmiňuji téměř v samotném závěru práce v sekci 5.5.

#### 2.1.3.4 Výhled do budoucna

Jak se zdá, stejně tak, jako já navazuji na svou bakalářskou práci touto diplomovou prací, tak i na Univerzitě Innsbruck mají s nástrojem RelaX další plány do budoucna. Aktuálně, dle dostupných informací, na nástroji RelaX dále pracuje student Thomas Blaas, který daný nástroj rozvíjí v rámci své bakalářské práce „*Improving the RelaX-tool with Automatic Optimization Functionality*“ [10].

## 2.2 Analýza současného překladače

Překladač, který vznikl v rámci mé bakalářské práce v akademickém roce 2015/2016, se již tři roky plně využívá při výuce v rámci Portálu. Tento překladač poskytuje studentům praktickou možnost osahat si aspekty relační algebry a následně je demonstrovat ekvivalentním způsobem pomocí jazyka SQL.

Zmiňovaný překladač je napsaný v jazyce Python verze 3.4.2 a jeho základní komponenty se opírají o další knihovny třetích stran. Mezi takové knihovny patří zejména knihovna PLY[1], pomocí které jsou implementovány stěžejní části překladače a to jak lexikální, tak i syntaktický analyzátor. Kromě nativních knihoven jazyka Python, aplikace RAT využívá ještě knihovny Flask, cx_Oracle nebo třeba sqlparse. Knihovna Flask umožnila zpřístupnění překladače skrze API. Knihovnu cx_Oracle tato implementace překladače využívá při testech, neboť součástí překladače je i pár základních testů překladu, jejichž výsledek se spouští v RDBMS Oracle. Poslední zmiňovaná knihovna, sqlparse, pak slouží k formátování řetězce obsahujícího SQL výstup do čitelné podoby.

Takto vyvinutý překladač implementuje překlad kompletní syntaxe relační algebry, která je v předmětu Databázové systémy na FIT ČVUT vyučována, a jejíž znalost je od studentů vyžadována. Součástí je samozřejmě i doku-

[1] PLY je zkratka pro Python Lex-Yacc a jedná se o implementaci lexikálního analyzátoru lex a syntaktického analyzátoru yacc v jazyce Python.

mentace, která částečně posloužila i pro vývoj této diplomové práce. V dokumentaci k aplikaci (či službě) RAT, nechybí ani dokumentace rozhraní (API), která specifikuje formát požadavků a odpovědí tak, aby tuto službu mohly využívat jiné aplikace. V dokumentaci je zahrnuta i programátorská příručka, která mi velmi usnadnila nasazení původní aplikace na lokální prostředí a která slouží i vývojářům Portálu, pro jejich automatické sestavení vývojového a produkčního prostředí.

I přesto, že se jedná o nejzdařilejší implementaci překladače, který byl v rámci fakulty vyvinut, tak i tento překladač, jako každý software, trpí jistými neduhy, o kterých se zmiňuji v podsekcích níže.

### 2.2.1 Lexikální analýza operace přejmenování

První chyba, nikoliv však fatální, se vyskytuje v lexikální analýze. Jedná se ale o chybu, která běžnými uživateli nejspíše ani nebyla detekována. Přechodový automat lexikálního analyzátoru umožnil následovný zápis dotazu pomocí relační algebry:

```
ALBUMS [album_id → some_id → other_id]
```
(D1)

Počet takto použitých operací přejmenování jednoho atributu může být teoreticky nekonečný. Příklad výše pouze demonstruje, kde se chyba nachází. Na druhou stranu tento chybný zápis je detekován syntaktickým analyzátorem, který jej dle očekávání reportuje jako chybu v syntaxi. Nicméně toto chování muselo být v rámci praktické části této diplomové práce opraveno, neboť chování lexikálního analyzátoru přímo reflektuje jedno rozšíření, o kterém se zmiňuji v sekci 5.1.

### 2.2.2 Nepřesné vyhodnocování operace selekce

O něco větším nedostatkem překladače je vyhodnocování operace selekce[2], pokud tato operace následuje po projekci[3]. Tento problém souvisí s definicí priorit vyhodnocování jednotlivých operací.

Prioritu při vyhodnocování operací relační algebry jsme společně s vedoucím Ing. Jiřím Hunkou a garantem předmětu Databázové systémy (BI-DBS) Ing. Michalem Valentou Ph.D., pro jednoduchost, definovali takto: „*Operace relační algebry se vyhodnocují zleva doprava, unární operace (selekce a projekce) mají vyšší prioritu před binárními operacemi. Samotnou prioritu při vyhodnocování binárních operací lze změnit použitím složených závorek*“ [3].

[2] Operace selekce značená jako `R`($\varphi$) vybírá z relace `R` n-tice takové, které splňují definovanou logickou podmínku $\varphi$.

[3] Operace projekce značená jako `R`[$\pi$] vybírá z relace `R` pouze hodnoty takových atributů, jež jsou uvedeny ve výrazu $\pi$.

Následující dotaz demonstruje, že tato definice, za výše uvedených podmínek, není úplně přesně dodržena. Mějme dotaz

```
ALBUMS[album_id](name = 'Viva la Vida')                (D2)
```

a k němu odpovídající překlad

```sql
SELECT DISTINCT album_id
FROM ALBUMS
WHERE name = 'Viva la Vida';
```

Překlad P1: Výsledek překladu dotazu D2

Dotaz D2 je logicky nesprávný. Pokud by překladač exaktně dodržoval definici priorit při vyhodnocování operací, pak by správný překlad neměl být spustitelný v žádném RDBMS, neboť po aplikaci projekce výsledná relace už žádný atribut `name` neobsahuje.

Na výstupu překladače (P1), si ale lze všimnout, že výsledek je obsažen v rámci jediného `SELECT` dotazu, který ovšem například v RDBMS Oracle spustitelný je. Pokud si tento výsledek rozebereme podrobněji, pak díky vlastnostem jazyka SQL se nejprve na n-tice tabulky `ALBUMS` aplikuje podmínka uvedená v klauzuli `WHERE`, a až následně se provede restrikce dané relace na zadanou množinu atributů, která je uvedená v klauzuli `SELECT` [11].

Tento překlad by správně měl být výsledkem dotazu, ve kterém selekce předchází projekci.

Korektní, avšak nespustitelný, překlad by vypadal následovně:

```sql
SELECT DISTINCT *
FROM
  (SELECT DISTINCT album_id
   FROM ALBUMS)
WHERE name = 'Viva la Vida';
```

Překlad P2: Výsledek překladu dotazu D2, pokud by překladač striktně dodržoval prioritu při vyhodnocování

### 2.2.3 Problém tečkové notace

Nejzávažnějším problémem, který vyvstal až při pozdějším reálném využívání překladače, je problém tečkové notace[4]. Problém tečkové notace v rámci původní práce nebyl specifikován a bohužel ani odhalen během následné implementace.

Nejprve si zadefinujme dotaz, na němž si ukážeme a posléze vysvětlíme problematiku tečkové notace

```
ALBUMS[genre_id][ALBUMS.genre_id =
        GENRES.id_genre>GENRES
```
(D3)

Takto formulovaný dotaz je sémanticky i syntakticky korektní a odpovídá následující slovní formulaci: „Vyber všechny hudební žánry takové, ke kterým je evidováno alespoň jedno album daného žánru“. Problém se však ukrývá v použití tečkové notace při definici podmínky spojení dvou relací. Vztaženo k dotazu D3 a jeho následovném překladu:

```
SELECT DISTINCT *
FROM GENRES
WHERE EXISTS
    (SELECT 1
     FROM
       (SELECT DISTINCT genre_id
        FROM ALBUMS)
     WHERE ALBUMS.genre_id = GENRES.id_genre);
```

Překlad P3: Demonstrace problému tečkové notace na překladu dotazu D3

Použití `ALBUMS.genre_id` v definici spojovací podmínky působí chybu při spuštění tohoto překladu v kterémkoliv RDBMS. Jak si lze na překladu P3 povšimnout, podmínka spojení tabulek `ALBUMS` a `GENRES` se beze změny vložila do klauzule `WHERE` korelovaného[5] poddotazu. Samotná tabulka `ALBUMS` se sice vyskytuje v rámci tohoto korelovaného poddotazu, avšak je součástí dalšího vnořeného poddotazu v klauzuli `FROM`. Tudíž přímý přístup k této tabulce z klauzule `WHERE` již není možný.

Problém tečkové notace vzniká pouze za určitých okolností, a to tehdy a jen tehdy, vede-li překlad alespoň jedné ze stran obecného spojení na vytvoření SQL poddotazu. V dotazu D3 je záměrně použita projekce atributu `genre_id` nad relací (tabulkou) `ALBUMS`. Operace projekce je jednou z možných operací,

[4]Problémem tečkové notace rozumíme použití výrazu tabulka.sloupec v situacích, kdy výsledný překlad vyústí v chybu při jeho spuštění.

[5]„*Termín vnořený korelovaný poddotaz znamená, že tento dotaz se odkazuje na řádky dotazu vnějšího*“[12]

jejichž překlad při použití s jakoukoliv další binární operací ústí ve vytvoření potřebného poddotazu.

V případě, že bychom se pokusili výsledek překladu P3 spustit v rámci RDBMS Oracle, pak by takové spuštění dotazu skončilo vyhozením výjimky `ORA-00904: "ALBUMS"."GENRE_ID": invalid identifier`.

Na druhou stranu lze v tomto případě dotaz D3 zapsat bez použití tečkové notace a následný překlad, respektive jeho spuštění, vrací korektní výsledek.

### 2.2.4 Porušení programátorských principů

Stejně tak, jako se postupem času vyvíjí programovací jazyky a software samotný, rozvíjí se i zkušenosti samotných programátorů. Nejinak je tomu i v mém případě. Za ty tři roky, co je překladač nasazený, jsem nabyl mnoho dalších zkušeností, vědomostí a programátorských návyků, které původní kód samozřejmě nemůže reflektovat. Jedná se zejména o programátorské principy a návrhové vzory, které se podrobně vysvětlují až v magisterském předmětu Architektonické a návrhové vzory (MI-ADP.16). Na ukázkách níže si ukážeme, jaké principy a jakým způsobem je zdrojový kód implementace stávajícího překladače porušuje.

#### 2.2.4.1 Single Responsibility Principle (SRP)

Prvním takovým principem, který je porušen, je princip jedné zodpovědnosti. Tento princip propaguje myšlenku: „*A class/interface should have only one reason to change*“ [13]. Čili jedna třída, či rozhraní, by měla mít pouze jednu zodpovědnost (důvod ke změně). Pod zodpovědností si lze představit nějakou jednoduchou a oddělenou funkcionalitu.

```
class Node(metaclass=ABCMeta):
    ##
    # @brief Propagates warning messages
    #
    def propagate_warning_messages(self, l_warnings,
                                   r_warnings = None):
        # Method implementation

    ##
    # @brief Translates an SQL query or it's part
    #
    def translate(self):
        # This method picks the appropriate translation method based
        # on the value of node_type property
```

Ukázka zdrojového kódu U1: Porušení principu SRP

V implementaci překladače existuje třída `Node`, která například provádí překlad do SQL (každý uzel ví, jak se má přeložit), ale také ví, že má vzniklé varovné zprávy dále propagovat, více viz ukázka U1.

V této ukázce vznikají hned dva důvody pro změnu, jednak to může být změna týkající se překladu daného typu uzlu, ale také například změna v chování (propagaci) varovných zpráv z podřízených uzlů. Instance této třídy by pak neměly vědět, jak se mají přeložit do SQL, neboť pouze slouží jako uzly abstraktního syntaktického stromu. Tato informace by se naopak měla předat jiné instanci, například instanci typu `Visitor`. Mimo jiné takto definovaná třída porušuje i princip Open-Closed Principle, o kterém píši v následující podsekci.

#### 2.2.4.2 Open-Closed Principle (OCP)

Dalším z takových principů, který kód překladače porušuje, je princip Open-Closed Principle. Tento princip říká: „*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*“ [13]. Volně přeloženo, každá softwarová entita (třída, modul, funkce apod.) by měla být otevřená pro rozšíření, ale uzavřená pro modifikaci. Jinými slovy, přidáním další entity nebo entit by se chování dotčené entity mělo změnit bez toho, aniž by se muselo zasahovat do jejího zdrojového kódu. Ukázka zdrojového kódu U2 původního překladače níže přesně tento princip porušuje.

```
##
# @brief Translates an SQL query or it's part
#
def translate(self):
    if self.is_leaf() and self.value is None:
        raise RATException("Leaf node '{0}' without any value"
            .format(self.node_type))

    if self.node_type == "table":
        self.translate_table()
    elif self.node_type == "projection":
        self.translate_projection()
    elif self.node_type in ("attribute", "rename"):
        self.translate_attribute()
    elif self.node_type == "selection":
        self.translate_selection()
    elif ...
```

Ukázka zdrojového kódu U2: Porušení principu OCP

Tato zkrácená ukázka jasně zachycuje zásadní problém v návrhu abstraktního syntaktického stromu (AST). Přidání dalšího uzlu by znamenalo

zásah do metody `translate()` a přidání další `elif` větve, což přesně porušuje princip definovaný výše. Odstranění tohoto problému tkví v použití dědičnosti.

#### 2.2.4.3 Law of Demeter

Deméteřin zákon je definován následovně: „*Each unit should have only limited knowledge about other units: only units 'closely' related to the current unit*“ [13]. Jinými slovy, každá programová jednotka (objekt – `O`) volá metody (`m`):

- pouze své vlastní – `O`,
- metody objektů poskytnutých skrze parametry metody `m`,
- metody objektů vytvořených v rámci metody `m`,
- metody objektů, ze kterých je objekt `O` složen.

Jaké to přináší výhody? Dodržování Deméteřina zákonu snižuje závislost dané třídy na dalších třídách, které pro její fungování nejsou nezbytně nutné. Pokud daný objekt závisí přímo na objektech, které pro něj nejsou přímo viditelné, významně se tím zvyšuje provázanost jednotlivých komponent v rámci programu. Ukázka kódu U3 níže znázorňuje porušení tohoto principu.

```
##
# @brief Translates @b projection node
#
def translate_projection(self):
    # Some code

    if self.children[0].sql_query.attr_part is None:
        ...

    # Some other code
```

Ukázka zdrojového kódu U3: Porušení principu Law of Demeter

Tento výňatek ze zdrojového kódu překladače názorně demonstruje závislost uzlu AST. Tento uzel typu `projection` přímo závisí na hodnotě atributu `attr_part` objektu `sql_query`, jež je jiného uzlu. Teprve tento uzel je přímo přístupný z kolekce synovských uzlů.

### 2.2.5 Shrnutí

Ačkoliv jsem se tenkrát snažil řešení překladače co nejvíce uzpůsobit pro snadné rozšíření (viz citace z mé bakalářské práce: „*Avšak je třeba mít na paměti, že se v budoucnu může vyskytnout požadavek na překlad nejen do Oracle SQL, z tohoto důvodu je žádoucí připravit překladač pro snadné rozšíření*“ [1]), rozhodl jsem se po zjištění výše uvedených nedostatků pro **zcela novou implementaci překladače**.

Toto rozhodnutí však neznamená, že by původní implementace překladače byla zcela nedostačující. Některé její součásti, zejména lexikální a syntaktickou analýzu (včetně definice gramatiky relační algebry), bylo možné s určitými změnami využít jako podklad při implementaci nového překladače v rámci této práce.

Na druhou stranu, zejména pro řešení problému tečkové notace není současná implementace překladače uzpůsobena. Dodělání této funkcionality a odstranění porušení programátorských principů by znamenalo provést nemalý zásah do téměř celé aplikace.

Kapitola 3

# Návrh

Předtím, než se pustíme do samotného návrhu překladače, povíme si obecně, co to překladač vlastně je a z jakých částí se vlastně skládá. Překladačem rozumíme v informatice nástroj, který akceptuje věty z jazyka vstupního jež jsou následně analyzovány pomocí lexikálního analyzátoru (zkráceně lexer). Pokud je analýza úspěšná, převede se vstupní věta na posloupnost tokenů, která reprezentuje danou větu, a předá se k analýze syntaktickému analyzátoru. Syntaktický analyzátor (zkráceně parser) pak čte tuto posloupnost a na základě gramatických pravidel, zpravidla určených pomocí bezkontextové gramatiky, určuje, zda daná věta do daného jazyka patří, či nikoliv. Syntaktická analýza je úspěšná, pokud po přečtení celé posloupnosti tokenů je vrchol zásobníku parseru prázdný a na vstupu není žádný další token [14]. Výstupem obecného parseru je pak derivační strom, který se předává sémantickému analyzátoru, který produkuje odpovídající abstraktní syntaktický strom [15]. Tento strom se posléze posléze vyhodnocuje, a případně se ještě převádí do mezikódu, než je finálně přeložen do cílové podoby (do cílového jazyka).

Od této chvíle a dále budu hovořit o překladači, který vznikl v rámci mé bakalářské práce, jako o původním překladači. Naopak o překladači, který vznikl v rámci praktické části této diplomové práce budu dále mluvit jako o současném či novém překladači.

Než přejdeme k samotné implementaci nového překladače, je nutné nejprve specifikovat veškeré funkční a nefunkční požadavky, které na nový překladač budou kladeny. Jakmile budeme mít tyto požadavky specifikovány, obecně si představíme některé návrhové vzory, které jsem při návrhu překladače aplikoval. V kapitole Realizace se pak věnuji jejich reálnému použití.

## 3.1 Definice požadavků

Překladač, který vznikl v rámci této práce, musí bezpodmínečně odpovídat požadavkům, které byly kladeny na překladač původní. Z tohoto důvodu se tyto požadavky budou v následujících sekcích do jisté míry opakovat.

### 3.1.1 Funkční požadavky

- **FP01**: Překladač bude realizovat překlad z RA do SQL.
- **FP02**: Překladač musí rozeznávat veškerou syntaxi a značení operací, které jsou v rámci předmětu Databázové systémy (BI-DBS) vyučovány:
  - značení operace projekce,
  - značení operace selekce,
  - značení operace kartézského součinu,
  - značení operace přirozeného spojení, včetně levých, pravých přirozených polospojení a anti joinů,
  - značení operace obecného spojení, včetně levých, pravých obecných polospojení a anti joinů,
  - značení operací množinového rozdílu, sjednocení a průniku,
  - značení operace relačního dělení,
  - značení operace přejmenování.
- **FP03**: Překladač bude dodržovat definovanou prioritu při vyhodnocování operací, a to i v případě projekce a selekce (viz 2.2.2).
- **FP04**: Překladač musí umožnit práci i s duplicitními názvy sloupců, které mohou vzniknout při operacích kartézského součinu a obecného spojení.
- **FP05**: Překladač musí implementovat řešení pro problém tečkové notace.
- **FP06**: Překladač bude realizovat překlad pro RDBMS Oracle, PostgreSQL a MariaDB.

### 3.1.2 Nefunkční požadavky

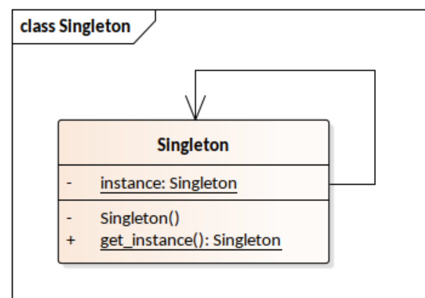
- **NP01**: Překladač bude dostupný skrze API.
- **NP02**: Vyřízení 95 % požadavků nesmí přesáhnout 1 vteřinu a vyřízení jakéhokoliv požadavku nesmí přesáhnout 3 sekundy.
- **NP03**: Funkčnost překladače musí být ověřena pomocí testů.
- **NP04**: V práci musí být vhodně využity návrhové vzory.
- **NP05**: Překladač musí být rozšiřitelný.
- **NP06**: Zdrojový kód musí být řádně zdokumentovaný.
- **NP07**: Součástí dokumentace musí být i dokumentace API.
- **NP08**: Překladač musí být integrován do Portálu.

## 3.2 Návrhové vzory

S termínem návrhový vzor nebo vzory se nejčastěji setkáváme v souvislosti s objektově orientovaným programováním. Mluvíme-li o objektově orientovaném programování, rozumíme tím programovací paradigma, které nám umožňuje modelovat prvky reálného světa při řešení dané programovací úlohy. Návrhové vzory lze chápat jako souhrn široce rozšířených postupů pro řešení obecných problémů v konkrétním kontextu. Použitím návrhových vzorů lze minimalizovat riziko špatného návrhu při vývoji software [16][17].

Jednou z nejznámějších publikací, která se zabývala návrhovými vzory, se stala kniha Design Patterns: Elements of Reusable Object-Oriented Software vydaná již v roce 1994. Za touto knihou stojí skupina GoF (Gang of Four), jejímiž členy byli Erich Gamma, John Vlissides, Ralph Johnson a Richard Helm [16]. V rámci této knihy bylo publikováno na třiaadvacet návrhových vzorů, z nichž pouze některé jsem využil při návrhu nového překladače. V následujících podsekcích popisuji vybrané vzory na obecné úrovni, v sekci 4.2 pak zmiňuji jejich konkrétní využití.

### 3.2.1 Singleton




Obrázek 3.1: Class diagram návrhového vzoru Singleton

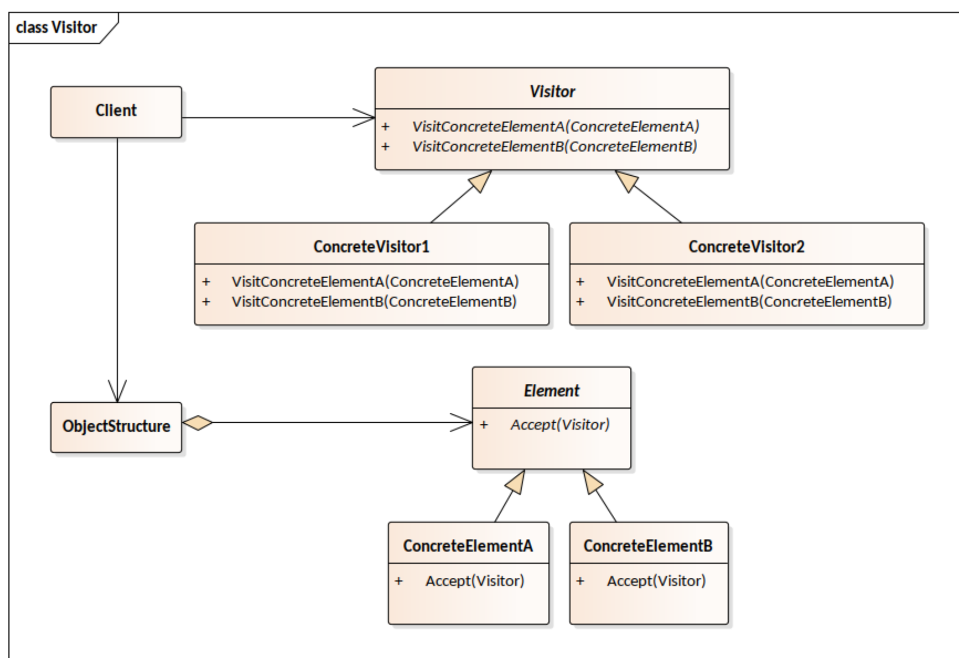
Prvním vzorem je vzor Singleton neboli jedináček. Aplikací vzoru Singleton na konkrétní třídu zajistíme, že daná třída bude mít vždy pouze jednu jedinou instanci, ke které lze globálně přistupovat. Vztaženo k obrázku 3.1, přímé vytvoření Singleton instance není možné, neboť tato třída obsahuje privátní konstruktor. Jediným veřejným rozhraním je metoda `get_instance()`. Když tuto metodu poprvé zavoláme, uloží se do proměnné `instance` nově vytvořená instance třídy `Singleton`. Při dalších následných voláních funkce `get_instance()` se vždy vrací reference na tuto vytvořenou instanci. Tím je zajištěno, že veškeré objekty, které si danou instanci vyžádají, pracují s jednou a tou samou instancí.

Toto chování může být v některých situacích velmi užitečné. Návrhový vzor Singleton například implementuje `logging` balíček ze standardní knihov-

ny jazyka Python [18]. Vícenásobné volání funkce `get_logger()` z uvedeného balíčku pokaždé vyústí v získání reference na jednu globální instanci loggeru. Neznamená to však, že by v rámci dané aplikace mohl existovat pouze jeden jediný logger. Konkrétní instanci loggeru, jehož reference je vyžadována, lze specifikovat hodnotou parametru výše zmíněné funkce, která reprezentuje jeho název. Tímto lze získat přístup k jedné a té samé instanci loggeru z vícero míst v rámci aplikace bez toho, aniž by se reference k danému loggeru musela předávat pomocí parametrů.

Na druhou stranu je potřeba zmínit, že použití vzoru Singleton má i svá úskalí. Největším úskalím je použití návrhového vzoru Singleton ve vícevláknových aplikacích, u nichž je nutné serializovat přístup k atributu `instance`. Dalším problémem je pak nemožnost použití dědičnosti [19].

### 3.2.2 Visitor




Obrázek 3.2: Class diagram návrhového vzoru Visitor

Dalším návrhovým vzorem je návrhový vzor Visitor. Návrhový vzor Visitor se řadí mezi behaviorální vzory a umožňuje nám přidat nové operace do hierarchické struktury tříd bez toho, aniž bychom museli zasahovat do jejich zdrojového kódu [16].

Na diagramu 3.2 je zobrazena hierarchická struktura elementů, které generalizují třídu `Element`. Každý konkrétní element musí pak implementovat

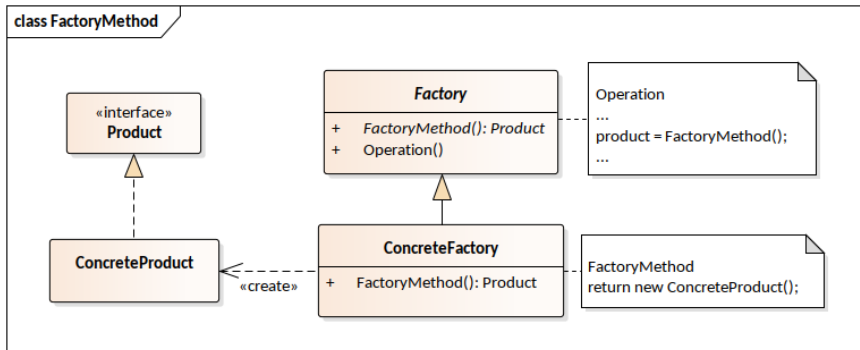
metodu, zde `accept()`, která danou instanci Visitora přijme a zavolá jeho odpovídající metodu.

Dále je pak definována abstraktní třída `Visitor`, která obsahuje abstraktní metody pro navštívení jednotlivých konkrétních elementů. Tyto metody pak každá konkrétní instance, která dědí od třídy `Visitor`, musí implementovat. Každý konkrétní Visitor pak může implementovat zcela odlišné chování metod `VisitConcreteElement()`. Pro jednoduchost si lze představit, že třída `ConcreteVisitor1` vypíše typ elementu do konzole, naproti tomu třída `ConcreteVisitor2` naopak zapíše typ elementu do souboru. Další specifické chování lze přidat velice snadno, a to přidáním další konkrétní implementace třídy `Visitor`.

Naproti tomu přidání další konkrétní implementace třídy `Element` vyžaduje přidání a implementaci patřičné metody do všech podtříd třídy `Visitor`, které s danou hierarchickou strukturou pracují.

Samotný průchod touto strukturou může definovat například daná struktura (například voláním metody `accept()` svých potomků), ale i Visitor samotný.

### 3.2.3 Factory method




Obrázek 3.3: Class diagram návrhového vzoru FactoryMethod

Posledním návrhovým vzorem, který zde uvedu, je návrhový vzor FactoryMethod, česky tovární metoda. Tovární metoda se řadí mezi vytvářecí návrhové vzory a jejím jediným úkolem vytvořit instanci nějakého typu a tu vrátit volajícímu.

Produkt definuje společné rozhraní pro objekty, které tovární metoda vytváří a třída `ConcreteProduct` je konkrétní realizací tohoto rozhraní. Třída `Factory` deklaruje tovární metodu, která vrací instanci typu `Product`. `ConcreteFactory` je pak konkrétní implementací třídy `Factory`, která předefinovává tovární metodu tak, aby vracela `ConcreteProduct` instanci.

Tovární metoda má hned několik podob, jednou z nich může být i statická tovární metoda. „*Takové tovární metody lze sdružovat do logických celků a pro každý takový celek vytvořit tzv. tovární třídu*“ [20].

### 3.2.4 Další návrhové vzory

V souvislosti s návrhem nového překladače byly uvažovány i další návrhové vzory mezi nimiž byly vzory Composite a Observer.

Návrhový vzor Composite byl uvažován v souvislosti s uzly abstraktního syntaktického stromu. Dle dostupných zdrojů, implementace tohoto vzoru vyžaduje i implementaci metod pro přidávání a odebírání komponent [21][22]. Toto není zcela žádoucí chování uzlů AST, neboť uzlům AST jsou odkazy na jejich případné potomky předány již při jejich vytváření a uzly jako takové už musí být dále neměnné.

Pro návrhový vzor Observer jsem nenalezl při implementaci vhodný případ pro jeho užití, byť jsem se domníval, že například při automatickém přejmenovávání relací by mohly relace informovat své navázané sloupce o změně. Tato domněnka se mi nepotvrdila a daný problém byl tedy řešen jiným způsobem.

Kapitola 4

# Realizace

Nyní se dostáváme k samotné realizaci nového překladače. Na základě zkušeností s úspěšnou implementací předchozího překladače nedává úplný smysl jakkoliv drasticky měnit použité technologie či knihovny. Přesto jistých změn nová implementace doznala.

Za prvé, překladač je implementován opět v jazyce Python. Nicméně pro usnadnění nasazení překladače do Portálu byl proveden upgrade verze 3.4.2 na verzi 3.5.3. Tento upgrade nemá žádný signifikantní dopad na samotnou implementaci, neboť nepřináší žádné nové funkcionality, které bych nutně při vývoji potřeboval. Na druhou stranu poslední zmiňovaná verze jazyka Python (3.5.3) je předinstalována na samotném Portálu a pro eliminaci potenciálních rozdílů mezi lokálním a produkčním prostředím (Portálem) byly tyto verze sjednoceny.

Za druhé, pro implementaci lexikální a syntaktické analýzy byla opět zvolena knihovna PLY, jejímž původním autorem je David Beazley. U této knihovny byl též proveden upgrade z verze 3.8 na verzi 3.11, která vyšla v roce 2018. Jedná se pravděpodobně o jednu z posledních verzí této knihovny, neboť se sám autor pustil do kompletního přepracování této knihovny, která nově vzniká pod názvem SLY. Autor se sice zavázal, že v případě objevení kritických chyb budou tyto chyby řešeny, nicméně veškerý vývoj nových funkcionalit probíhá pouze v rámci onoho nového projektu [14]. Hloubavý čtenář by se mohl ptát, proč nová implementace nevyužívá novou knihovnu? Odpověď je nasnadě. Prvním důvodem, méně závažným, je kompatibilita s verzemi jazyka Python. Knihovna SLY vyžaduje verzi 3.6 nebo novější. Druhým a závažnějším důvodem je absence jakékoliv oficiálně vydané verze této knihovny. V době implementace a psaní této diplomové práce je na portálu této knihovny umístěno varování: „*THIS IS A WORK IN PROGRESS. NO OFFICIAL RELEASE HAS BEEN MADE. USE AT YOUR OWN RISK*“ [23].

Za třetí, zpřístupnění překladače pomocí API je realizováno opět pomocí knihovny Flask, u níž proběhl upgrade z verze 0.10.1 na verzi 1.0.2.

V neposlední řadě je nutné zmínit, že implementace nového překladače

oproti původnímu již nadále nezávisí na knihovnách cx_Oracle a sqlparse.

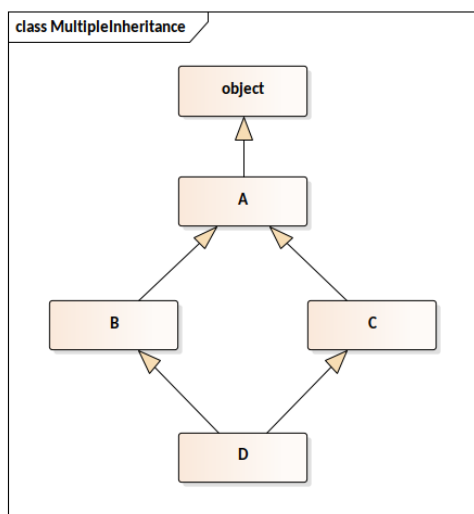
Původní překladač obsahoval jen velmi malé množství testů (28), které se spouštěly v RDBMS Oracle. Implementace původního překladače tak navíc byla závislá nejen na knihovně cx_Oracle, která spuštění testů v RDBMS Oracle umožňuje, ale také na ovladačích, které na daném prostředí musí být nainstalovány. Takto malá skupina testů dokázala pokrýt jenom základní a typické dotazy v RA a jejich překlady. Upuštění od těchto testů neznamená, že by se překlady v novém překladači netestovaly. Více se o testování lze dočíst v sekci 4.12.

Knihovna sqlparse se využívala pro formátování řetězce, který reprezentoval výsledný překlad do SQL. Velmi výrazně se tím zvýšila čitelnost výstupu, avšak v některých případech ani toto nebylo zcela ideální. O problémech a změně ve formátování SQL výstupu se podrobněji rozepisuji v podsekci 4.10.2.

## 4.1 Koncepty jazyka Python

Kromě výše zmíněných knihoven stojí implementace nového překladače na některých velmi důležitých konceptech jazyka Python, mezi nimiž je například používání properties místo getterů a setterů, ale hlavně koncept vícenásobné dědičnosti. K použití properties místo getterů a setterů na internetu existuje spousta zdrojů, například na serverech www.programiz.com a www.python-course.eu, včetně praktických ukázek použití této techniky, nicméně pro účely diplomové práce je tento koncept příliš jednoduchý a méně zajímavý. Naopak je tomu v případě vícenásobné dědičnosti.

### 4.1.1 Vícenásobná dědičnost




Obrázek 4.1: Příklad vícenásobné dědičnosti

Prvně je potřeba říci, že jazyk Python neimplementuje rozhraní tak, jak je známe z ostatních programovacích jazyků, jako je Java a C++. Syntaxe podporující tvorbu a používání rozhraní byla navržena v rámci PEP-245, avšak toto rozšíření bylo zamítnuto [24]. Python je totiž programovací jazyk, který korektně implementuje vícenásobnou dědičnost, podporu dynamického typování[6] (tzv. ducktyping) i podporu abstraktních tříd včetně abstraktních metod. Z tohoto důvodu podpora rozhraní není v jazyce Python potřeba.

Právě vícenásobná dědičnost je jedním z konceptů jazyka Python, na nichž je implementace nového překladače postavena. V jazyce Python, technicky vzato, je každá třída a každý objekt potomkem třídy `object`.

Pokud se podíváme na obrázek 4.1, všimneme si, že třída `A` dědí od třídy `object`. Od třídy `A` pak dědí třídy `B` a `C`. Konečně třída `D` dědí od tříd `B` a `C` přesně v tomto pořadí. Díky vlastnostem dědičnosti pak třída `D` tranzitivně dědí i od třídy `A`, respektive `object`.

V případě vícenásobné dědičnosti vyvstává problém při volání libovolného atributu či metody třídy `D`. Tento problém je též referovaný jako tzv. „diamond problem“. Nejprve se zjišťuje, zda se každý takový atribut či metoda nachází v aktuální třídě (`D`). Pokud tomu tak není, pokračuje se v hledání u přímých rodičů třídy `D` a to vždy zleva doprava, tj. nejprve v třídě `B` a až poté ve třídě `C`. Zároveň se žádné třídě nevyhledává více než jednou. Pokud ani v těchto třídách není nalezen patřičný atribut či metoda, stejný postup je aplikován i na rodiče těchto tříd. Tomuto chování se též říká linearizace třídy `D` a způsobu této linearizace se říká Method Resolution Order (MRO) [26].

Tato metoda musí zajistit, že třída, v našem případě `D`, bude vždy po linearizaci předcházet třídám svých rodičů, a že žádná třída se výstupu po linearizaci nebude vyskytovat vícekrát. V případě, že třída dědí od vícero rodičovských tříd, jak je tomu v případě třídy `D`, pak linearizace musí zohlednit pořadí těchto tříd přesně tak, jak jsou specifikovány třídou `D`. Názorněji si linearizaci ukážeme na následujícím příkladu V1, který reflektuje diagram výše. Výsledek linearizace dané třídy lze v jazyce Python zjistit voláním metody `mro()`.

```
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>]
```

Výstup z příkazové řádky V1: Ukázka linearizace třídy `D` pomocí MRO

A skutečně, výsledek volání metody `mro()` přesně odpovídá požadavkům

[6] U dynamicky typovaných programovacích jazyků probíhá kontrola datových typů až při samotném běhu programu [25]

uvedeným výše. Třída `D` předchází svým rodičovským třídám, jejichž pořadí je zachováno a třída `A` se ve výstupu objevuje pouze jednou. Výstup V1 ukazuje přesné pořadí tříd, v nichž by se daná metoda či atribut, vyhledávala.

## 4.2 Praktické využití návrhových vzorů

V této podkapitole si popíšeme a případně i názorně ukážeme, jakým způsobem a v jakých situacích byly návrhové vzory diskutované v sekci 3.2 aplikovány, čímž dojde ke splnění prvního nefunkčního požadavku **NP04**.

### 4.2.1 Singleton

Návrhový vzor Singleton byl využit na místech, kde bylo potřeba zajistit existenci pouze jedné jediné instance v rámci programu. Konkrétně například třída `DBScheme`, která objektově reprezentuje předávané databázové schéma, je Singleton. Je žádoucí, aby toto schéma existovalo v programu pouze jednou, nechceme totiž vytvářet případné další kopie, čímž je ušetřeno místo v paměti. Nutno zmínit, že reprezentace databázového schématu je neměnná a používá se pouze pro čtení, zejména pro dotazování na existenci tabulky (relace) a následné získání jejího seznamu dostupných sloupců, se kterými se dále pracuje.

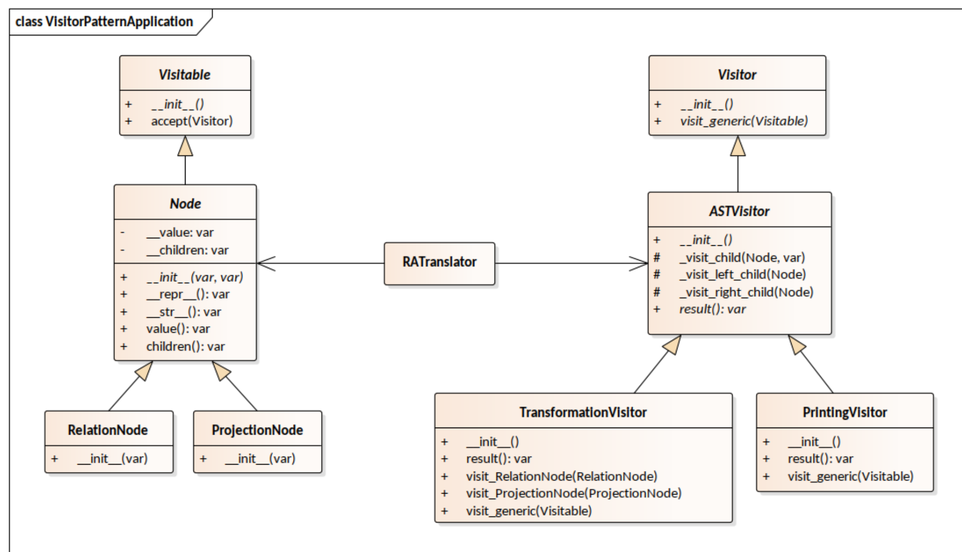
Další implementací návrhového vzoru Singleton je třída `RATConfig`. Jak název napovídá, tato třída načítá konfigurační soubor překladače a jeho hodnoty pak ukládá do paměti, ke kterým pak poskytuje přístup. Třída `RATConfig` implementuje vzor Singleton proto, aby přístup k samotnému konfiguračnímu souboru umístěnému na disku byl proveden pouze jednou. Tento přístup se provede pouze při volání konstruktoru této třídy a jak již víme, konstruktor takové třídy se volá pouze v případě, je-li instanční atribut prázdný (nenastavený). V opačném případě se předává hodnota tohoto atributu, která obsahuje referenci na objekt, v našem případě referenci na jednu jedinou instanci typu `RATConfig`.

### 4.2.2 Visitor

Asi vůbec nejdůležitějším vzorem použitým v implementaci nového překladače je vzor Visitor. Tento návrhový vzor byl v aplikaci použit hned několikrát. Obrázek 4.2 pak názorně, byť zjednodušeně, ukazuje konkrétní jeho použití v souvislosti s abstraktním syntaktickým stromem. O dalších případech užití tohoto návrhového vzoru pak budu mluvit v podkapitolách 4.10 a 5.5.

Pokud přejdeme k diagramu 4.2, můžeme si všimnout, že je zde vyobrazena abstraktní třída `Visitable`. Tato třída definuje metodu `accept()`, jež je společná všem odvozeným instancím, které jsou navštívitelné jakoukoli instancí třídy `Visitor`. Konkrétní implementace této metody je znázorněna na ukázce U4. Implementaci metody `accept()` automaticky přejímají veškeré

třídy dědící od třídy `Visitable`. Pokud bychom tedy volali metodu `accept()` na uzlu `RelationNode`, pak se nejprve díky dědičnosti zjistí, zda daný `Visitor` implementuje metodu `visit_RelationNode()`. Pokud tomu tak není, zavolá se metoda `visit_generic()`, která slouží jako tzv. „fallback“ metoda.




Obrázek 4.2: Aplikace návrhového vzoru Visitor

Na ukázce níže je znázorněna implementace již zmíněné metody `accept()` v rámci třídy `Visitable`. Chování této metody už pak žádná odvozená třída nepředefinovává.

```
import abc

class Visitable(metaclass=abc.ABCMeta):

    # ...

    def accept(self, visitor) -> None:
        visitor_method = getattr(visitor, 'visit_' +
                                 self.__class__.__name__, None)
        if visitor_method is None:
            visitor_method = getattr(visitor, 'visit_generic')

        visitor_method(self)
```

Ukázka zdrojového kódu U4: Generická metoda `accept()` pro všechny instance třídy `Visitable`

Dále je pak na diagramu 4.2 uvedená další abstraktní třída `Node`. Třída `Node` je společná rodičovská třída veškerým možným uzlům AST. Na diagramu jsou pro jednoduchost uvedeny pouze dva typy uzlů, a to `RelationNode` a `ProjectionNode`. Celkový počet typů uzlů se zastavil na čísle třicet sedm. Každý takový uzel je implementačně velmi jednoduchý a obsahuje pouze svůj vlastní specifický konstruktor. Otázka, která je nasnadě, proč existuje tolik tříd, když nemají žádnou specifickou funkci? Odpověď je prostá, jejich specifickou funkci dodefinovává patřičná instance třídy `Visitor`.

V původní implementaci překladače existovaly pouze dva typy uzlů, respektive pouze jeden. AST syntaktický strom se skládal pouze z uzlů `OracleNode`, což byla třída odvozená od třídy `Node`. Překlad se pak řídil na základě interního atributu `node_type`. Ale jak jsme si ukázali v sekci 2.2.4.2, tento přístup porušoval princip OCP!

Dostáváme se k třídě `Visitor`. Tato třída definuje minimální rozhraní, které odvozené třídy, jež jsou instanciovány, musí implementovat. Abstraktní třída pak obsahuje abstraktní konstruktor a dále předpis pro generickou metodu `visit_generic()`. Tato metoda se volá, jestliže instance podtřídy `Visitor` neimplementuje patřičnou metodu `visit`, jejíž suffix odpovídá právě navštívené instanci, viz ukázka U4.

Třída `ASTVisitor` je další abstraktní třídou rozšiřující třídu `Visitor` o další užitečné metody. V drtivé většině mají uzly AST právě dva syny, proto zde přibyly implementace pro navštívení jak levého, tak i pravého potomka daného uzlu. Z toho vyplývá, že průchod AST si sama řídí konkrétní instance třídy `ASTVisitor`. Tato třída pak ještě definuje abstraktní property pro získání kumulovaného výsledku po průchodu AST.

A konečně třídy `TransformationVisitor` a `PrintingVisitor` dodefinovávají dodatečné funkcionality pro uzly AST, které navštěvují. Třída `PrintingVisitor` pak definuje pouze generickou metodu `visit_generic()`, která se volá pro jakýkoliv uzel AST. Je to pochopitelné, úkol tohoto návštěvníka je získat řetězcovou reprezentaci AST pro účely ladění a opravy chyb. Metoda `visit_generic()` tedy jednotně definuje formát a způsob, jakým se jednotlivé uzly serializují do výsledného řetězce. Naproti tomu třída `TransformationVisitor`, která má na starosti transformaci AST pak dodefinovává metody, které jsou specifické právě pro daný konkrétní typ uzlu, který je navštíven. K uzlům `RelationNode`, `ProjectionNode` a dalším je potřeba přistupovat a tedy je i transformovat rozdílným způsobem.

Samotný překladač, reprezentovaný třídou `RATranslator`, pak disponuje jednou jedinou referencí na uzel typu `Node`, který reprezentuje kořen AST. Tento kořen pak akceptuje danou instanci návštěvníka, a zahájí se tím tak rekurzivní sestup v rámci daného stromu.

### 4.2.3 FactoryMethod

V rámci překladače existuje několik továrních tříd, které na požádání vyrábí nové objekty, případně je i konfigurují, nebo vytváří objekty skládáním z hodnot objektů, které byly předány pomocí parametrů. Ve většině případů se nejedná o implementaci návrhového vzoru FactoryMethod, implementace se spíše podobá návrhovému vzoru AbstractFactory, nicméně i v tomto případě nejde o přesnou implementaci tohoto vzoru, neboť zde chybí element společného předka.

Například továrna `ComponentFactory` dokáže vytvořit a předkonfigurovat lexikální či syntaktický analyzátor, či poskytnout referenci na komponentu, se kterou překladač dále pracuje. Takovou komponentou může být například daný AST Visitor. Naproti tomu například továrna `ASTFactory` centralizuje veškeré vytváření uzlů do jediného místa a klient (syntaktický analyzátor) si během parsování pouze žádá o vytvoření konkrétního daného uzlu a dále se nestará o to, jak je uzel vytvořen. Tato kompetence je plně přenechána třídě `ASTFactory`.

Ovšem existuje zde třída `MessageFactory`, která implementuje variantu vzoru FactoryMethod, a to variantu statické tovární metody, více viz [20]. Cílem této továrny je vyrobit zprávu, která se posléze posílá zpět volajícímu, a která je před samotným odesláním serializována do formátu JSON.

## 4.3 Definice databázového schématu

V analytické části práce byly v rámci dotazů relační algebry (D1, D2 či D3) použity názvy relací včetně jejich atributů, se kterými budu v této práci i nadále pracovat a na nichž budu demonstrovat chování nového překladače.

Tyto relace byly přeneseny do databáze a následně naplněny formou databázových CREATE a INSERT skriptů. Odpovídající skripty jsou samozřejmě součástí GIT repozitáře tak, aby si dané schéma mohl kdokoliv vytvořit a zprovoznit, ať už na svém lokálním či vzdáleném prostředí.

Schéma, evidující informace o hudebních albech, skladbách, interpretech, žánrech, ale i drobných obchodech bylo vybudováno za účelem ladění překladače samotného. Takto definované schéma nám v následujících sekcích poslouží i pro demonstraci dalších ukázkových dotazů, některých specifických problémů, ale i překladů samotných. Předně je potřeba říci, že dané schéma je smyšlené a data v něm použitá se nemusí zakládat na pravdě.

#### Evidence hudebních alb

Tabulka `ALBUMS` eviduje základní údaje o hudebních albech. Každé album je jednoznačně identifikováno svým umělým klíčem `album_id` a ke každému takovému albu je vždy přiřazený umělec, který je jeho autorem. Hudební album musí mít vyplněné jméno, které nemusí být v rámci evidence unikátní. Dále

evidujeme cenu alba, rok vydání, případnou poznámku a nepovinně i žánr daného alba.

### Evidence umělců

Tabulka ARTISTS uchovává jednoduché n-tice o hudebních skladatelích (umělcích). Každý umělec je identifikován svým uměle vytvořeným číslelným klíčem artist_id. Umělec pak má vždy své umělecké jméno a případně je k němu možné uložit i dodatečnou poznámku.

### Evidence skladeb

Tabulka TRACKS nám umožňuje evidenci skladeb. U skladby evidujeme její název, nepovinně i délku skladby. Skladba je identifikovatelná skrze umělý identifikátor track_id. Abychom mohli přiřazovat skladby jednotlivým albům, existuje zde i vazební tabulka ALBUMS_TRACKS, která je tvořená umělými klíči album_id, track_id a číslem určujícím pořadí skladby na daném albu. Počet skladeb takto přiřazených není nijak omezen, hudební album může mít skladbu jednu, deset, nebo také žádnou.

### Evidence žánrů

Naše databáze eviduje několik málo hudebních žánrů v tabulce GENRES. Jedná se o velmi jednoduchou evidenci skládající se z povinných atributů obsahujících název žánru a hodnotu jednoznačného umělého identifikátoru žánru (id_genre).

### Evidence prodejen

Poslední databázovou entitou je tabulka prodejen pojmenována jako STORES. Každá prodejna je též identifikována svým umělým identifikátorem store_id a každá prodejna má své jméno. K evidenci alb, které jsou na prodejnách dostupné slouží vazební tabulka ALBUMS_STORES, která se skládá z třech povinných sloupců. Jsou jimi album_id, id_store a pcs reprezentující id alba, obchodu a počet kusů na dané prodejně. V databázi se mohou vyskytovat prodejny, které neprodávájí jediné album, ale také prodejny, které prodávají všechna alba vedená v evidenci.

## 4.4 Definice souhrnného příkladu

Na základě výše uvedené definice databázového schématu konečně můžeme formulovat dotaz v RA, který nás bude provázet následujícími podkapitolami.

```
{ARTISTS[ARTISTS.artist_id = ALBUMS.artist_id]{ALBUMS
!<* ALBUMS_TRACKS}}[artist_id, artist_name, ALBUMS.name]
```
(D4)

Tento dotaz je komplexní, nebo chcete-li souhrnný, jeho transformace do SQL totiž zahrnuje řešení veškerých problémů, které při překladu mohou nastat. Mezi tyto problémy patří zejména již zmiňovaný problém tečkové notace, dále pak problém automatického přejmenování sloupců a též se v něm mění priorita při jeho vyhodnocování.

Slovně bychom tento dotaz mohli popsat následovně: „Vyber všechny hudební umělce (interprety), u nichž evidujeme hudební alba, na kterých se nevyskytují žádné skladby (chybí v naší evidenci). Zobraz umělcovo id, jméno a název takového alba."

Na takto definovaném dotazu D4 si ukážeme, jak je tento dotaz lexikálním analyzátorem převeden na posloupnost tokenů a jaký se k němu konstruuje odpovídající abstraktní syntaktický strom. Posléze si přiblížíme transformaci tohoto stromu do objektově orientované reprezentace modelující dotaz SQL a na závěr si ukážeme samotnou tvorbu překladu, která vzniká průchodem této objektové struktury.

Než se do toho pustíme, povíme si něco o změnách v logování a nové možnosti konfigurace překladače.

## 4.5 Konfigurace překladače

Významným posunem oproti předchozímu překladači je možnost jeho částečné konfigurace, která u původního překladače nebyla možná. Existuje zde totiž již zmiňovaná třída `RATConfig`, která načítá informace ze souboru `default.ini` umístěného v adresáři `rat/conf/`. V rámci tohoto souboru lze například specifikovat v jakém módu se má překladač spustit, jakou syntaxi má použít (o této předpřipravené funkcionalitě se více rozepisuji v sekci 6.3), nebo maximální délku RA dotazu.

V současné chvíli překladač podporuje dva módy, `prod` a `dev`, ve kterých je možné jej spustit. Mód `dev` slouží pro vývoj a lokální testování překladače a oproti módu `prod` se loguje daleko více informací, které pomáhají k debugování a ladění překladače. Naproti tomu mód `prod` je určen pro běh překladače v produkčním prostředí, například v rámci Portálu.

Maximální délka řetězce nesoucí dotaz v RA je nastavena na tisíc znaků. Tuto hodnotu lze kdykoliv v budoucnu změnit bez nutnosti zásahu do implementace překladače. Nicméně jak se během používání původního překladače ukázalo, tato hodnota je více než dostatečná.

Výčet hodnot zde uvedený není úplný, ale zahrnuje asi nejzajímavější možnosti konfigurace.

## 4.6 Logování

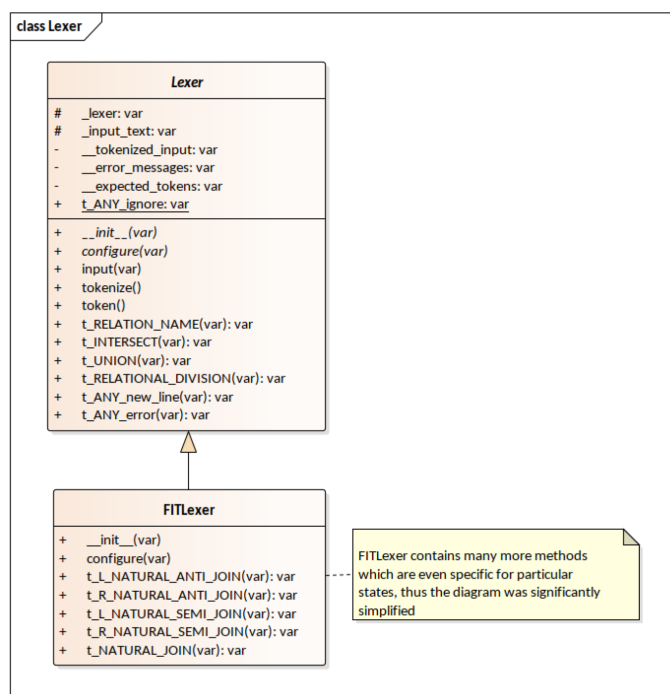
Další změna nastala v logování. Kromě zmiňovaného souboru `default.ini` je ve stejném adresáři přítomen i další konfigurační soubor `logging.ini`.

V tomto souboru je uvedena konfigurace celkem devíti loggerů. U každé specifikace takového loggeru je pak uvedena i cesta a název souboru, do kterého daný logger provádí zápis, je-li o to překladačem požádán.

Každý log soubor je navíc rotovaný. Ve většině případů probíhá rotace logových souborů po dosažení určité velikosti v MB, v jednom případě se rotují soubory vždy každý den o půlnoci, nejpozději před prvním zápisem do daného souboru. Samotnou rotaci zajišťuje pak modul `logging` jazyka Python. V původní implementaci překladač samotný logové soubory nerotoval, tato úloha byla přenechána komponentě operačního systému, komponentě `logrotate`.

## 4.7 Lexikální analýza

Lexikální analýza oproti původnímu řešení nedoznala zásadních změn. Během několikaletého používání původního překladače se neukázalo, že by překladač nebyl schopný rozeznat veškeré značení operací relační algebry, nebo že by vykazoval jakékoli potíže při jejich identifikaci.




Obrázek 4.3: Class diagram lexikálního analyzátoru

Na základě těchto zjištění posloužila původní implementace lexikálního analyzátoru jako dobrý základ pro vývoj nové a upravené verze, která se stala součástí nového překladače. První větší změna se odehrála v rámci samotného

návrhu tříd lexeru. Oproti předchozímu řešení byla nově zavedena abstraktní třída `Lexer`. Tato třída definuje společné rozhraní, které musí implementovat veškeré odvozené třídy implementující lexikální analyzátor, viz diagram 4.3. Tento krok byl učiněn zejména s ohledem na možnou rozšiřitelnost, o které se podrobněji bavím v sekci 6.3.

Na již zmíněném diagramu jsou zobrazeny nejdůležitější detaily z návrhu tříd lexikálního analyzátoru. Abstraktní třída `Lexer` je zde téměř kompletní a definuje rozpoznávání společných operací, jejichž značení je univerzální napříč různými zdroji [2][8]. Mezi tyto operace je zařazeno rozpoznávání názvů relací, operace množinového průniku, sjednocení a relačního dělení.

Odvozená třída `FITLexer` doimplementovává další metody, které slouží k rozpoznávání operací relační algebry, jejichž značení je specifické v rámci výuky Databázových systémů (BI-DBS) na FIT ČVUT. Pro zvýšení čitelnosti diagram 4.3 neobsahuje kompletní výčet veškerých implementovaných operací. Z tohoto důvodu návrh třídy `FITLexer` obsahuje jen velmi malý zlomek metod, které jsou ve skutečnosti implementovány.

Další méně důležitou změnou bylo oddělení definic regulárních výrazů do samostatných souborů, které tak nejsou součástí implementace třídy `Lexer`, `FITLexer` respektive. Tyto regulární výrazy se používají pro rozpoznávání konkrétních tokenů. Daná metoda, například `t_RELATION_NAME`, je pak odekorována dekorátorem obsahující proměnnou nesoucí definici regulárního výrazu, viz ukázka U5:

```
##
# @brief Definition for @c RELATION_NAME token in @c initial state
#
@lex.TOKEN(regex_common.RELATION_NAME)
def t_RELATION_NAME(self, t):
    return t
```

Ukázka zdrojového kódu U5: Ukázka použití dekorátoru při definici lexikálního pravidla

Takto definovaný a použitý regulární výraz přináší další výhodu, zejména pokud je použit na vícero místech. Ideálním příkladem je regulární výraz pro detekci názvu sloupce. Tento typ tokenu je rozpoznáván pouze pokud se lexikální analyzátor nachází ve specifickém stavu, například v rámci selekce, či projekce.

Spolu s dalšími obdobně definovanými pravidly je tak lexikální analyzátor schopen rozpoznat validní zápis operací relační algebry a ten převést na posloupnost tokenů, která se vzápětí předává syntaktickému analyzátoru.

Posledním vylepšením pak byla oprava již zmiňované chyby (2.2.1) umož-

ňující zřetězení použití operátoru přejmenování. Řešením se ukázalo přidání dalšího stavu lexikálního analyzátoru, v němž se neumožňuje opětovné použití operace přejmenování. Pokud by student přeci jen použil zřetězené přejmenování sloupce obdobně tak, jak je tomu u dotazu D1, pak by studentovi byla vrácena chybová hláška a jeho vstup by nebyl předán k syntaktické analýze, dokud by tato chyba nebyla opravena.

### 4.7.1 Příklad

Odpovídající výstup z lexikální analýzy pro souhrnný dotaz D4 je uveden na výstupu V2 níže.

```
[LexToken(L_CURLY_BRACKET,'{'), LexToken(RELATION_NAME,'ARTISTS'),
LexToken(JOIN_BEGIN,'['), LexToken(COLUMN_NAME,'ARTISTS.artist_id'),
LexToken(COMPARATOR,'='), LexToken(COLUMN_NAME,'ALBUMS.artist_id'),
LexToken(JOIN_END,']'), LexToken(L_CURLY_BRACKET,'{'),
LexToken(RELATION_NAME,'ALBUMS'), LexToken(L_NATURAL_ANTI_JOIN,'!<*'),
LexToken(RELATION_NAME,'ALBUMS_TRACKS'), LexToken(R_CURLY_BRACKET,'}'),
LexToken(R_CURLY_BRACKET,'}'), LexToken(PROJECTION_BEGIN,'['),
LexToken(COLUMN_NAME,'artist_id'), LexToken(COMMA,','),
LexToken(COLUMN_NAME,'artist_name'), LexToken(COMMA,','),
LexToken(COLUMN_NAME,'ALBUMS.name'), LexToken(PROJECTION_END,']')]
```

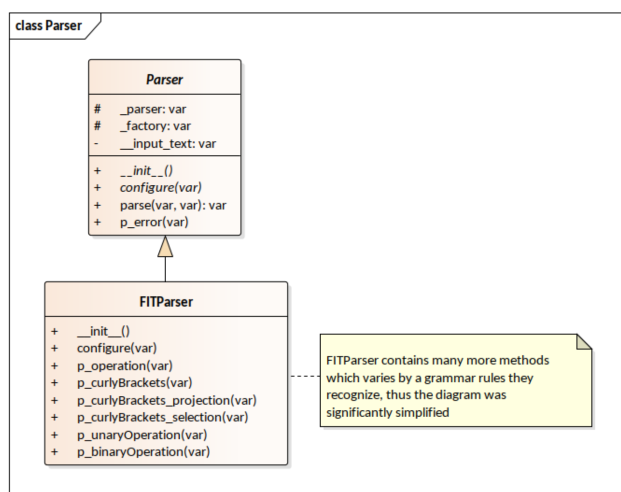
Výstup V2: Seznam tokenů po lexikální analýze dotazu D4

Daný dotaz, který byl předán lexikálnímu analyzátoru formou řetězce, byl úspěšně převeden na posloupnost tokenů definovanou výše. Token je instance třídy `LexToken` a za vytváření těchto instancí zodpovídá samotná knihovna PLY. Každá taková instance si v sobě nese svůj typ (název tokenu, například `RELATION_NAME`), hodnotu (část řetězce, která byla identifikována jako token, například `ARTISTS`), ale i číslo řádky a pozice, kde se tento token nachází ve vstupním řetězci.

## 4.8 Syntaktická analýza

Úkolem syntaktického analyzátoru (parseru) je ověřit, že takto vytvořená posloupnost tokenů zastupuje větu jazyka vstupního, v našem případě relační algebry. Lexikální analyzátor totiž není schopen určit, zda je daný zápis dotazu syntakticky správný. Tato kompetence plně spadá pod syntaktický analyzátor, který na základě předem definovaných gramatických pravidel, s využitím zásobníku, dokáže určit, zda tato posloupnost tokenů do daného jazyka patří, či nikoliv. Výsledkem úspěšné syntaktické analýzy může být téměř cokoli, například řetězec, nebo abstraktní syntaktický strom.

Pro podrobnější popis syntaktické analýzy odkáži čtenáře na svou bakalářskou práci [1]. V sekci 3.3 totiž popisuji, jaká metoda a jaké techniky se používají při parsování dané věty, a též se tam zaměřuji na formu definice gramatických pravidel. Pro účely této práce nejsou tyto základní informace až tak podstatné, neboť se vytvořením nové verze překladače nezměnily. Naopak zajímavější a přínosnější pro nás bude nástin toho, co se oproti původnímu řešení změnilo.




Obrázek 4.4: Class diagram syntaktického analyzátoru

Opět, citelnou změnou oproti původní implementaci bylo vytvoření společné abstraktní třídy `Parser`, která slouží jako minimální rozhraní implementující, či předepisující implementaci společných metod. Tato změna byla učiněna opět na základě požadavku na rozšiřitelnost, jemuž se věnuji podrobněji v sekci 6.3. Na diagramu 4.4 je pak dále přítomna třída `FITParser`, jejíž znázornění bylo oproti realitě velmi zjednodušeno. Tato třída ve skutečnosti obsahuje implementaci veškerých potřebných metod sloužících k rozpoznávání kompletní syntaxe relační algebry vyučované na FIT ČVUT. Každá tato metoda musí být definována přesně v souladu s dokumentací knihovny PLY a v rámci svého docstringu[7] musí obsahovat alespoň jedno gramatické pravidlo, jehož chování je následně implementováno.

Změny se dočkala i samotná struktura gramatiky relační algebry, jež byla notně zoptimalizována. Původní počet 121 přechodových pravidel byl zredukován na celkových 114. Nutno ovšem dodat, že úplná gramatika tak, jak je prezentována v příloze E, byla rozšířena o dalších 21 gramatických pravidel, které umožňují implementaci nových funkcí. K těmto funkcionalitám se ještě v průběhu této práce dostaneme.

[7]Docstring je dokumentační řetězec, který se vyskytuje na začátku definice modulu, třídy, funkce, metody, apod.

Takto definovaná gramatika je opět jednoznačná a umožňuje tak bezproblémovou (bezkonfliktní) syntaktickou analýzu. Pokud by gramatika byla víceznačná, pak bychom do jazyka relační algebry zavedli jistou míru neurčitosti či víceznačnosti, což by bylo zdrojem konfliktů („shift-reduce“ / „reduce-reduce“). Konflikt „shift-reduce“ vzniká v situacích, kdy parser neví, zda má vrchol svého zásobníku redukovat pomocí daného pravidla, či na něj přesunout další symbol. Naproti tomu konflikt „reduce-reduce“ vzniká tehdy, pokud parser disponuje vícero pravidly, podle kterých lze vrchol zásobníku redukovat. V obou případech by se parser musel rozhodnout, buď na základě svého výchozího nastavení, nebo na základě dodatečných informací (priorit), jak s daným konfliktem naložit. Ať tak či onak, oba případy jsou možným zdrojem chyb a nepřesností, které nechceme. Proto je jednoznačnost gramatiky důležitá a i v budoucnu musí být zachována!

Poslední, avšak nutnou, změnou prošlo vytváření samotných uzlů AST. Během parsování posloupnosti tokenů žádá třída `FITParser` tovární třídu `ASTFactory` o vytvoření konkrétního typu uzlu. Uzly, které jsou listy AST, obsahují pouze hodnotu, například název relace (tabulky) či atributu (sloupce). Uzly které jsou vnitřními uzly AST naopak obsahují reference na své synovské uzly a v ojedinělých případech obsahují i svou hodnotu. Takto početná množina implementačně jednoduchých uzlů nám umožňuje perfektně definovat chování Visitora při návštěvě konkrétního typu uzlu bez toho, aniž bychom nadále porušovali OCP.

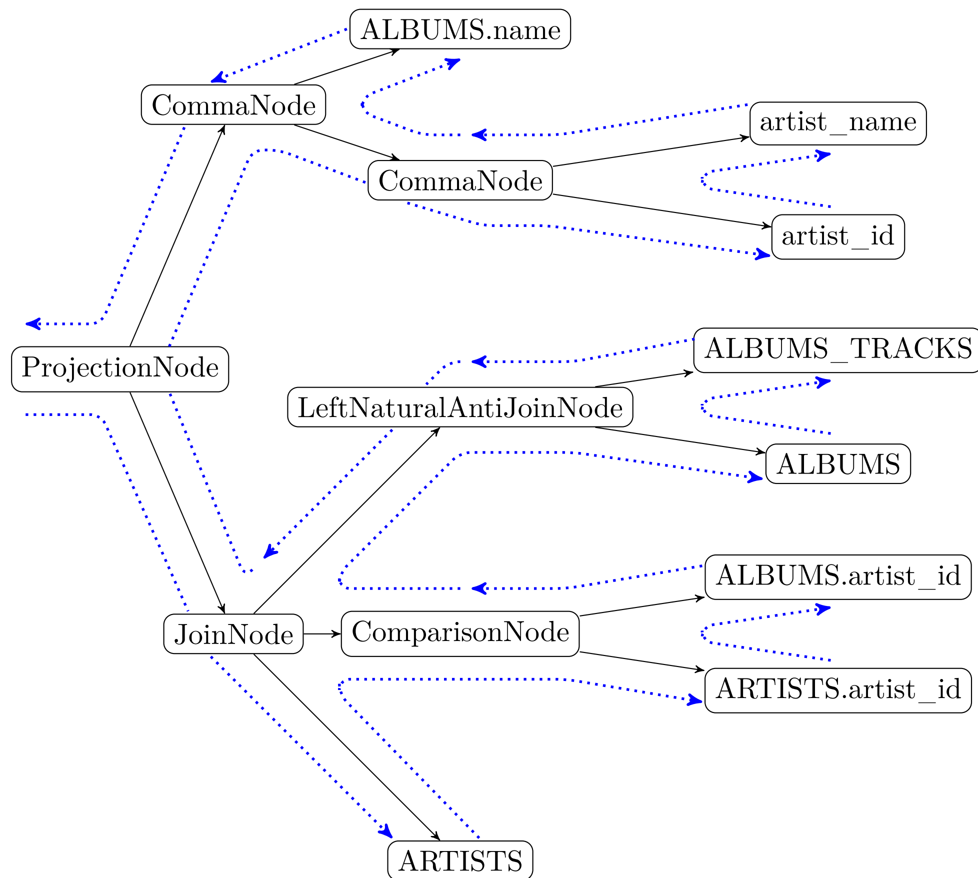
### 4.8.1 Příklad

Následující obrázek 4.5 znázorňuje výsledný abstraktní syntaktický strom, který vznikne parsováním posloupnosti tokenů z ukázky výstupu V2 lexikálního analyzátoru. Výsledný strom je příliš široký na to, aby se vešel na stránku v klasické podobě, tj. kořen nejvýše, vnitřní a koncové uzly pod ním. Z tohoto důvodu byl abstraktní strom otočen o devadesát stupňů proti směru hodinových ručiček.

Lze si všimnout, že výsledný AST opravdu obsahuje různé typy uzlů. Listy jsou ve skutečnosti uzly typu `RelationNode`, respektive `ColumnNode`, nicméně pro účely ukázky je pro nás přínosnější ukázat si skutečnou hodnotu daného uzlu. Samotné vnitřní uzly mívají zpravidla dva potomky, výjimečně tři. Tři potomky (synovské uzly) mají uzly reprezentující určitý druh obecného spojení, kde kromě operandů dané operace je potřeba znát i podmínku spojení.

Grafická reprezentace AST je na obrázku 4.5 doplněna modrými tečkovanými čarami, které znázorňují průchod Visitora tímto stromem. Instance třídy `TransformationVisitor` prochází tuto strukturu metodou POST-ORDER, tj. nejprve se navštíví levý podstrom daného uzlu, pak pravý podstrom daného uzlu a na základě získaných mezivýsledků se provede patřičná akce. Naproti tomu instance třídy `PrintingVisitor` prochází strom metodou PRE-ORDER.

Modré šipky na obrázku 4.5 naznačují navštívení a finální zpracování uzlu metodou POST-ORDER.




Obrázek 4.5: Odpovídající AST sestrojený na základě dotazu D4

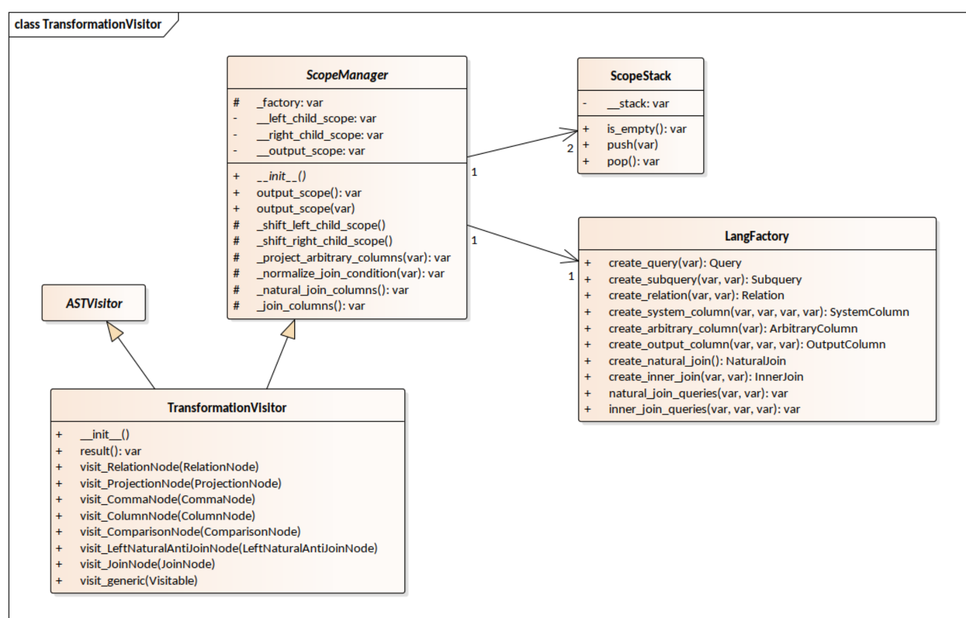
## 4.9 Transformace AST

Jakýkoliv obdobný abstraktní syntaktický strom, jako je znázorněn na obrázku 4.5, ani zdaleka nepřipomíná strukturu `SELECT` dotazu jazyka SQL. Tato struktura, byť jakkoliv je dobře definovaná, nám pouze naznačuje, které tabulky, sloupce a operace s nimi by se ve výsledném SQL výstupu měly objevit. Rozložení tohoto stromu nám definuje i pořadí operací při jeho vyhodnocování. Pohledem na obrázek 4.5 snadno zjistíme, že nejprve musíme vyhodnotit operaci levého přirozeného anti joinu a až teprve výsledek této operace se připojuje pomocí operace obecného spojení k tabulce `ARTISTS`. Otázkou tedy je, jak takovouto strukturu převést do SQL dotazu? Odpovědí je transformace do jiné, vhodnější struktury!

V původním řešení překladače se tato struktura překládala rovnou do řetězců reprezentující části SQL dotazu, které se při průchodu stromem různě

skládaly dohromady. Řešení to bylo dostatečné, ale každý uzel musel vědět, jak se má přeložit. Práce s řetězci by pro řešení například tečkové notace byla příliš pracná a ne zcela ideální, což už jistým způsobem odrážela samotná implementace původního překladače. Navíc s tím, jak je nově struktura AST definována, by při zachování původního způsobu musel každý uzel vědět, jak se má přeložit do SQL spustitelného v rámci RDMBS Oracle, PostgreSQL a MariaDB, což už opět porušuje SRP.

Na základě těchto zjištění jsem se rozhodl vzít tuto strukturu AST a přetransformovat ji do nové objektové struktury modelující právě daný SQL dotaz, což je přesně úkol pro instanci třídy `TransformationVisitor`. Transformace AST je tedy úplně nový mezikrok, který byl do nového překladače implementován, a který nám pomáhá řešit například problém tečkové notace. Hlavním úkolem instance třídy `TransformationVisitor` je danou strukturu AST projít, a na základě metod definovaných v rámci zmíněné třídy sestavit generickou objektovou strukturu modelující výsledný dotaz `SELECT`.



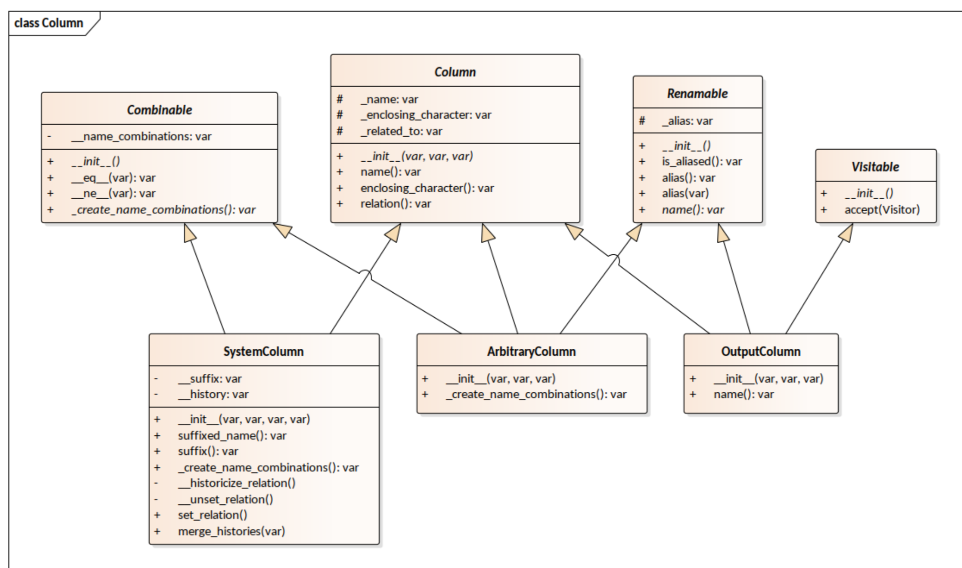

Obrázek 4.6: Class diagram pro implementaci TransformationVisitora

Třída `TransformationVisitor` je prvním případem z mnoha, ve kterém je využito vícenásobné dědičnosti. Na obrázku 4.6 je záměrně vynechána implementace rodičovské abstraktní třídy `ASTVisitor`, neboť již byla naznačena dříve. Na tomto diagramu si lze všimnout, že instance třídy `TransformationVisitor` je zároveň instancí třídy `ScopeManager`. Druhá jmenovaná třída je opět abstraktní a jejím úkolem je řídit veškerou manipulaci se sloupci tak, aby byl řešen a hlavně vyřešen problém tečkové notace, či automatického přejmenování sloupců. O tom ještě později.

Visitor si během průchodu AST musí odkládat seznam aktuálních sloupců na zásobník příslušný patřičnému synovskému uzlu. V opačném případě by totiž mohlo dojít k přepsání tohoto seznamu seznamem novým, například při návštěvě levého podstromu pravého syna, což není žádoucí. Ke zpracování těchto seznamů sloupců uložených na zásobníku dochází vždy až po navštívení obou (případně všech tří) synovských uzlů. Samotný zásobník je na diagramu reprezentován třídou `ScopeStack`. Poslední třídou je třída `LangFactory`. Tuto tovární třídu žádá Visitor pokaždé, potřebuje-li vytvořit nějaký element, který se stane součástí hierarchické struktury modelující výsledný dotaz `SELECT`.

### 4.9.1 Práce se sloupci

Proto, aby překladač mohl jakkoliv pracovat se sloupci a řešit problémy spojené s jejich manipulací, je nutná znalost databázového schématu, nad kterým má překladač operovat. Z tohoto důvodu překladač, v souladu s původní implementací, očekává kromě samotného RA dotazu i informaci o cílovém databázovém schématu.




Obrázek 4.7: Class diagram zachycující veškeré typy sloupců

Samotná manipulace se sloupci během transformace není vůbec triviální. Až na třetí pokus se mi podařilo navrhnout struktury takové, s nimiž lze řešit problémy s manipulací spojené. Výsledný návrh je zobrazen na diagramu 4.7. Na diagramu jsou znázorněny tři konkrétní třídy, které modelují sloupec (atribut). Tyto sloupce se liší v účelu svého použití. Vzápětí si tyto tři třídy podrobněji rozebereme.

### SystemColumn

Instance této třídy slouží čistě pro interní reprezentaci sloupců, které jsou k dispozici v rámci dané úrovně transformace (zanoření Visitora). Jsou to jediné instance, které si třída `ScopeManager` udržuje a s jejichž pomocí řeší problematiku tečkové notace, včetně problematiky automatického přejmenování. Každý sloupec (nejen systémový) má své jméno. Toto jméno může být uzavřeno mezi uvozovky. Každý sloupec si může dále uchovávat referenci objektu, ke kterému se váže, typicky k tabulce. K udržování těchto společných atributů slouží abstraktní třída `Column`, která je společná veškerým konkrétním podtřídám reprezentující jakýkoliv typ sloupce.

Třída `SystemColumn` je navíc podtřídou třídy `Combinable`. Třída `Combinable` vyžaduje implementaci metody, jejímž úkolem je vytvořit množinu ekvivalentních vyjádření názvu sloupce. Na základě této množiny lze pak provést porovnání s dalšími objekty. Tyto kombinace zahrnují variantu obsahující pouze samotný název sloupce, případně název sloupce doplněný o název tabulky oddělený tečkou (tečková notace). Pokud navíc byla tabulka přejmenována[8] (byl ji dán automaticky vygenerovaný alias), pak kombinace zahrnují i tuto možnost – název sloupce doplněný o alias tabulky oddělený tečkou. Těch kombinací může být ještě daleko více, nicméně pro nástin principu vytváření kombinací je tento výčet plně dostačující.

### ArbitraryColumn

Instance třídy `ArbitraryColumn` mají odlišnou úlohu a případy, ve kterých se vyskytují, jsou zcela odlišné. O vytvoření těchto instancí žádá `TransformationVisitor` tovární třídu `LangFactory` pokaždé, navštíví-li uzel `ColumnNode` v rámci průchodu AST. Tento uzel se vyskytuje v AST tehdy a jen tehdy, pokud uživatel explicitně použije název sloupce v rámci svého RA dotazu, například v projekci, selekci, nebo v podmínce spojení.

Objekt třídy `ArbitraryColumn` opět obsahuje stejné položky (název, ...), nicméně tyto položky vznikají parsováním názvu tak, jak jej uživatel zadal. Důsledkem je zachování formátování řetězce nesoucí název sloupce – ponechávají se velká a malá písmena beze změny. Tato třída je opět `Combinable`, tudíž opět dochází ke generování kombinací názvů pro následné porovnání, avšak zcela odlišným způsobem. Pakliže student použil tečkovou notaci, tj. název sloupce je ve tvaru „tabulka tečka sloupec“, pak veškeré kombinace, které tato třída vytváří, bezpodmínečně musí obsahovat i název dané tabulky.

Třída `ArbitraryColumn` je navíc `Renamable`. Co to znamená? V relační algebře je definován operátor přejmenování sloupce. Požadovaný název po přejmenování se tedy ukládá jako hodnota atributu `_alias` třídy `Renamable`.

[8]Tabulka se automaticky přejmenovává v případě, že se v dotazu vyskytuje víc než jednou.

Z diagramu 4.7 je patrné, že instance třídy `ArbitraryColumn` není jediná, které lze přiřadit alias. To samé lze provést i pro instanci třídy `OutputColumn`.

**OutputColumn**

Třída `OutputColumn` je implementačně nejjednodušší a jejím jediným úkolem je nést veškeré potřebné informace, které se posléze objeví v samotném SQL dotazu. Vzhledem k tomu, že s instancemi třídy `SystemColumn` se v průběhu průchodu AST stále pracuje (mění se), bylo potřeba vytvořit nový typ sloupce, jehož hodnoty budou i přese všechno konstantní. Hodnoty, které si instance třídy `OutputColumn` uchovává, typicky vznikají skládáním hodnot instancí tříd `SystemColumn` a `ArbitraryColumn` mezi nimiž byla pomocí operátoru porovnání (metoda `__eq__`) nalezena shoda. Typicky název sloupce se přejímá z instance třídy `SystemColumn`, neboť tento název je normalizovaný (vždy malými písmeny). Naopak proto, aby se ve výsledném SQL dotazu objevila operace přejmenování, je potřeba vzít odpovídající hodnotu aliasu z instance `ArbitraryColumn`, neboť to je jediná instance, která si tuto informaci nese.

#### 4.9.1.1 Problém tečkové notace

S takto výše definovanou objektovou strukturou je problém tečkové notace řešitelný. Instance třídy `AbstractColumn` si nesou informaci o tom, co uživatel chce, naproti tomu instance třídy `SystemColumn` nesou nejaktuálnější informace o sloupcích, které aktuálně použít lze.

Pro shrnutí, problém tečkové notace se týká zápisu názvu sloupce ve formátu „tabulka tečka sloupec“ v situacích, kde daná tabulka (relace) již není přímo viditelná. Tato situace nastává tehdy a jen tehdy, když pro zajištění korektnosti překladu musel být vytvořen v rámci klauzule `FROM`, `JOIN`, případně `WHERE`, poddotaz zahrnující onu tabulku.

Proto, aby tečková notace mohla fungovat, musely být pro instance třídy `SystemColumn` definovány následující pravidla:

1. Každý systémový sloupec (instance třídy `SystemColumn`) musí ve svém výchozím stavu znát tabulku, ke které se váže.
    - Řečí diagramu 4.7 musí hodnota atributu `_related_to` obsahovat referenci na objekt třídy `Relation`, který reprezentuje zdrojovou tabulku.
2. Použil-li student operaci přejmenování sloupce, pak se informace o původní vazbě na tabulku musí zapomenout – student definuje nový název sloupce, který v dané relaci není obsažen.
3. Převádí-li se daná část výsledného dotazu na poddotaz, pak se u všech sloupců viditelných v rámci daného rozsahu musí změnit vazba (hodnota atributu `_related_to`) na instanci nově vytvořeného poddotazu.

- Má-li být i po převodu na poddotaz zachována možnost použití tečkové notace, musí být splněny následující tři podmínky:
  - Původní vazba na tabulku se musí archivovat (viz atribut `__history` třídy `SystemColumn`).
  - Tato historie se musí zohlednit při vytváření jmenných kombinací – i přesto, že tabulka již není viditelná v rámci nového rozsahu, musíme umožnit použití tečkové notace zahrnující původní název tabulky.
  - Vytvořený poddotaz je opět potřeba pojmenovat – přiřadit mu alias.

4. Předchozí dvě pravidla musí být neustále opakována do doby, než je transformace dokončena

Z takto definovaných pravidel plynou následující dva důsledky. Za prvé, každá objektová reprezentace poddotazu modelující poddotaz v klauzuli `FROM` a `JOIN` musí být pojmenována, tj. musí mít přiřazený vlastní (automaticky generovaný) alias. Za druhé, je-li sloupec aktuálně vázán k poddotazu a nikoliv k tabulce, tato vazba se již nearchivuje.

Už víme, že projekce, selekce, či podmínka spojení obsahuje instance třídy `ArbitraryColumn`. Tyto instance se pak porovnávají proti seznamu aktuálně dostupných systémových sloupců. Dojde-li k nalezení shody mezi instancemi tříd `SystemColumn` a `ArbitraryColumn` na základě vygenerovaných kombinací, pak se pro výstup (vytvoření instance třídy `OutputColum`) používají nejaktuálnější informace získané z instance třídy `SystemColumn`. Příklad: instance třídy `SystemColumn` totiž ví, že se už přímo váže nikoliv k tabulce `ALBUMS`, ale k poddotazu `R1`. Proto se ve výstupu název tabulky, například `ALBUMS`, nahrazuje nejaktuálnější informací, názvem poddotazu, například `R1`.

Věci se o něco více komplikují ve spojení s přirozenými spojeními a RDBMS Oracle. Je-li použita projekce na dotazem, který obsahuje přirozené spojení, pak sloupce, které se objeví v klauzuli `SELECT` výsledného dotazu a přes něž došlo ke spojení dvou tabulek, nesmí obsahovat kvalifikátor. Jinými slovy, v tomto případě nelze tečkovou notaci použít! V opačném případě by spuštění daného `SELECT` dotazu vyústilo v chybu `ORA-25155: column used in NATURAL join cannot have qualifier`. Toto chování bylo podrobeno testování i v RDBMS PostgreSQL verze 10.3 a v něm použití tečkové notace nepůsobí žádný problém. Jedná se tedy o specifikum RDBMS Oracle – testováno na verzích 12c a 19c.

Jak v tomto případě překladač, respektive Visitor postupuje? Jednoduše. Mezi oběma stranami přirozeného spojení se prvně najdou všechny dvojice sloupců se shodným názvem, přes něž se tabulky spojí. Každá taková dvojice je tvořena právě dvěma instancemi třídy `SystemColumn`. V druhém kroku se u každého sloupce ve dvojici archivuje jeho vazba na tabulku (je-li nějaká) a aktuální vazba se nastaví na prázdnou hodnotu (`None`). Ve třetím kroku

se sloučí dvojice vždy v jednoho zástupce třídy `SystemColumn` sloučením jejich historií (archivované vazby k původním tabulkám). Vznikne tak instance, která má ve své historii vazby na obě původní tabulky. Díky tomu, že v této sloučené instanci je aktuální vazba nastavena na prázdnou hodnotu, pak se kvalifikátor nikdy neobjeví v klauzuli `SELECT`, byť o to student může jakkoliv požádat. Tím je chyba eliminována. Jakmile se stane výsledek tohoto přirozeného spojení součástí poddotazu (viz pravidlo 3), je možné tečkovou notaci opět použít, neboť vazba byla přenastavena z prázdné hodnoty referencí daného poddotazu.

Jelikož transformace ctí výše uvedená pravidla, lze označit požadavek **FP05** za splněný, a problém tečkové notace za vyřešený.

#### 4.9.1.2 Problém automatického přejmenování sloupců

Problematice automatického přejmenování se názorně i s příklady věnuji ve své bakalářské práci [1]. V této podkapitole si tuto problematiku velmi letmo osvěžíme a řekneme si, v čem se implementace nového překladače liší od té původní.

Problém automatického přejmenování sloupců se váže k operacím kartézského součinu a obecného spojení (vnějšího i vnitřního). Mějme dvě tabulky `ALBUMS` a `ARTISTS`. Obě tyto tabulky mají společný atribut `artist_id`. Pokud bychom na tyto dvě tabulky použili výše zmíněné operace, pak vznikne relace, jejíž atributy jsou tvořeny sloupci tabulky `ALBUMS` následovanými sloupci tabulky `ARTISTS`. Důsledkem je duplicitní výskyt sloupce `artist_id`, který je ve výsledné relaci obsažen dvakrát. I přes to je to relace opět validní, RDBMS Oracle i PostgreSQL zobrazí výsledek.

Potíže nastávají v případě, stane-li se takový dotaz poddotazem, a my nad tímto poddotazem budeme chtít vybrat hodnotu sloupce `artist_id`, viz následující ukázka možného překladu:

```
SELECT DISTINCT artist_id
FROM (
    SELECT DISTINCT *
    FROM ALBUMS
    CROSS JOIN ARTISTS
) R1;
```

Překlad P4: Ukázka dotazu SQL působící problémy při jeho spuštění

Pokud tento dotaz budeme chtít spustit v RDBMS Oracle, či PostgreSQL, pak se přístup jednotlivých systémů k danému problému liší. Spuštění tohoto dotazu v RDBMS Oracle končí chybou `ORA-00904: "ARTIST_ID": invalid`

identifier, což není úplně vypovídající hláška, neboť víme, že tento identifikátor existuje, dokonce dvakrát. Naproti tomu přístup PostgreSQL a MySQL je o něco lepší, chybová hláška vesměs sděluje, že sloupec artist_id je definovaný vícekrát, což je nezpochybnitelné.

Pro řešení této problematiky už v implementaci původního překladače byla implementována funkcionalita, která tento problém dokáže řešit tak, že duplicitní sloupce na pravé straně (v našem případě sloupce tabulky ARTISTS) v následné implicitní projekci přejmenuje. Tato funkcionalita vychází z chování RDBMS Oracle a k názvu duplicitního sloupce je přidán číselný suffix oddělený podtržítkem, viz další ukázka P5. Avšak toto chování bylo tenkrát implementováno pouze pro operaci kartézského součinu, nikoliv pro obecná spojení. V rámci této práce tedy logicky došlo k zahrnutí i operací obecného spojení, čímž je problematika automatického přejmenování sloupců kompletně vyřešena.

```sql
SELECT DISTINCT ALBUMS.album_id,
                ALBUMS.name,
                ALBUMS.note,
                ALBUMS.price,
                ALBUMS.year,
                ALBUMS.artist_id,
                ALBUMS.genre_id,
                ARTISTS.artist_id AS artist_id_1,
                ARTISTS.artist_name,
                ARTISTS.description
FROM ALBUMS
CROSS JOIN ARTISTS;
```

Překlad P5: Ukázka automatického přejmenování sloupce artist_id

Pozorný čtenář jistě zpozorněl při pohledu na ukázku P5. Nabízí se otázka, jak se k automatickému přejmenování sloupce postaví překladač s ohledem na problém tečkové notace. V tomto případě sice dochází k přejmenování, ale implicitnímu. Z tohoto důvodu u instance SystemColumn odpovídající sloupci artist_id z tabulky ARTISTS nedochází k zahození vazby na tuto tabulku. Naopak, u instance SystemColumn je pouze inkrementována číselná hodnota suffixu (viz atribut __suffix třídy SystemColumn na diagramu 4.7).

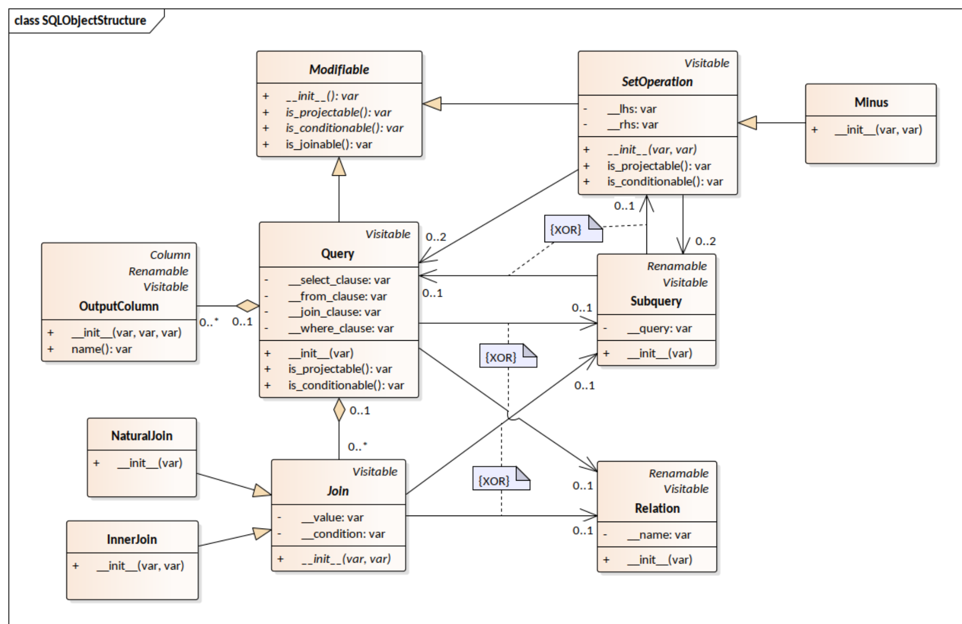
Kombinace, které se pak vytváří pro účely porovnání, obsahují navíc i název sloupce doplněný o právě přidaný suffix. Jinými slovy, pokud bychom následnou projekcí chtěli zobrazit hodnoty atributu artist_id z tabulky ARTISTS, můžeme toho docílit dvěma způsoby. Zápis ARTISTS.artist_id a artist_id_1 vedou ke stejnému výsledku a to k výběru požadovaného atributu.

Na základě výše uvedených sdělení můžeme požadavek **FP04** označit za splněný. Překlad P5 demonstruje toto chování v praxi.

### 4.9.2 Generování mezikódu

Úděl instance `TransformationVisitor` není přeložit strukturu AST do SQL, ale, jak již bylo zmiňováno, převést ji na novou, vhodnější strukturu, vzdáleně připomínající strukturu dotazu `SELECT` jazyka SQL. Nicméně ani tato struktura neví, jak se má do výsledného SQL přeložit. Z tohoto důvodu bychom mohli návštěvníka výše připodobnit ke generátoru mezikódu, neboť výsledná struktura není finální.

Na následujícím diagramu tříd 4.8 je znázorněna část tříd, na základě jejichž instancí se skládá hierarchická reprezentace, která je výsledkem transformace AST. Tento diagram zachycuje minimální strukturu tříd, na kterou se budu odkazovat během vysvětlování transformace na souhrnném příkladu. Z diagramu byly záměrně vynechány další typy spojení (podtřídy třídy `Join`), typy množinových operací, ale také getter a setter metody, které nejsou implementačně zajímavé a pro náš příklad důležité. Než přejdu k samotnému diagramu, tak na závěr ještě malá poznámka. Pokud třída dědí od některé z tříd, které již byly modelovány na předchozích diagramech, pak vazba na tyto třídy je naznačena v pravém horním rohu příslušné podtřídy.




Obrázek 4.8: Návrh tříd pro modelování dotazu `SELECT` jazyka SQL

Základní třídou je třída `Query`. Instanci této třídy si lze představit jako klasický jednoduchý `SELECT` ... `FROM` dotaz jazyka SQL. Instance třídy `Query`

může, ale nemusí obsahovat kolekci instancí třídy OutputColumn, která je uložena v klauzuli SELECT.

Každá instance třídy Query musí mít vyplněnou svou klauzuli FROM. Hodnota odpovídajícího atributu se musí předat při vytváření objektu. V klauzuli FROM pak může být uložena reference na instanci třídy Relation, či Subquery. Rozdíl mezi těmito dvěma instancemi by měl být zřejmý.

Objekt třídy Query může mít ještě nepovinně vyplněnou klauzuli JOIN a / nebo klauzuli WHERE. Pokud je vyplněna, pak se jedná o kolekci dalších objektů, například instancí třídy NaturalJoin v případě klauzule Join.

Množinová operace, reprezentována třídou SetOperation, Minus respektive, musí vždy znát referenci na levý a pravý operand. Operandy mohou být dvě instance třídy Query (srovnejme se syntaxí množinových operací jazyka SQL), Subquery, či kombinace předchozích. Operandy množinové operace mohou být dokonce další množinové operace, nicméně tyto množinové operace musí být obaleny instancí třídy Subquery pro jednoznačné určení priorit vyhodnocování.

Třída Subquery je pouze obalující třídou, která ve výsledném výstupu zajistí uzavření části dotazu mezi kulaté závorky. Je-li navíc dotaz pojmenovaný, je následně zobrazen i přiřazený alias. Vzpomeňme na problém tečkové notace v sekci 4.9.1.1, kde pro správné fungování je vyžadováno pojmenování veškerých poddotazů v klauzulích FROM a JOIN.

Implementačně nejjednoduššími třídami jsou třídy Join a Relation. Instance podtříd třídy Join obsahují hodnotu, jež může být reference na objekt třídy Subquery, či Relation, a případnou podmínku spojení. Rozdíl mezi případy, kdy je v atributu __value uložena ta, či ona instance by měl být zřejmý. Chování vychází z jazyka SQL, kdy lze k dotazu připojit poddotaz nebo tabulku samotnou. Naproti tomu instance třídy Relation nemá žádný speciální úkol. Nese v sobě pouze název tabulky převedený na velká písmena.

Veškeré třídy, které mohou být instanciovány, jsou ať už přímo, či nepřímo, potomky třídy Visitable. S touto hierarchickou strukturou si v dalším kroku dokáže poradit jiná instance třídy Visitor, která ji dokáže finálně přeložit do SQL.

#### 4.9.2.1 Oprava vyhodnocování operace selekce

S takto definovaným rozložením tříd, viz diagram 4.8, lze řešit požadavek **FP03**, který vyžaduje urovnání nepřesnosti při vyhodnocování operace selekce, která následuje po projekci, viz podkapitola 2.2.2. Při průchodu AST a před závěrečném zpracováním uzlu se TransformationVisitor dívá na mezivýsledky získané návštěvou podstromů daného uzlu. V případě selekce se dívá na výsledek pouze levého podstromu, neboť pravý podstrom reprezentuje podmínku selekce.

Výsledkem průchodu levého podstromu operace selekce je vždy instance třídy Query nebo SetOperation. Následně zde hraje úlohu třída Modifiable

z diagramu 4.8, která u tříd `Query` a `SetOperation` vynutila implementaci metod `is_projectable()` a `is_conditionable()`. Tyto metody říkají, zda na daný objekt lze aplikovat operaci projekce, selekce respektive.

V případě instance `Query` se pomocí metod zmíněných výše zjišťuje, zda instance již neobsahuje projekci nebo selekci. Pokud alespoň v jednom případě odpovíme „ano“, pak je potřeba:

1. Obalit instanci třídy `Query` instancí `Subquery`, čímž vznikne poddotaz. Tomuto poddotazu musí být následně přiřazen automaticky generovaný alias.

2. Vytvořit novou instanci třídy `Query` do jejíž klauzule `FROM` se vloží pojmenovaný poddotaz vytvořený o krok dříve.

3. Na takto nově vytvořenou instanci lze teprve v tomto okamžiku aplikovat selekci.

U množinových operací není selekce a projekce možná bez toho, aniž by byl poddotaz vytvořen. Teprve po jeho vytvoření teprve lze projekci a selekci aplikovat. Postup převodu na poddotaz zůstává úplně stejný, jako je uvedeno výše, s tím rozdílem, že se provádí vždy.

### 4.9.3 Příklad

Veškeré teoretické aspekty vysvětlované v předchozích sekcích budou za moment shrnuty a aplikovány na náš souhrnný příklad definovaný dříve. Následující popis kroků vychází ze struktury AST, která byla graficky znázorněna na obrázku 4.5. Na tomto obrázku je zobrazen i průchod tímto stromem, jehož šipky nám budou definovat pořadí návštěv jednotlivých uzlů v rámci transformace AST.

#### Krok 1 – Návštěva uzlu ARTISTS

Prvním navštíveným a zpracovaným uzlem je uzel `ARTISTS`, který je instancí třídy `RelationNode`. V tomto kroku vznikne první instance třídy `Query`, do jejíž klauzule `FROM` se vloží instance třídy `Relation` nesoucí název `ARTISTS`. Okamžitý překlad nově vytvořené instance třídy `Query` by vypadal následovně:

```sql
SELECT DISTINCT *
FROM ARTISTS;
```

Překlad P6: Překlad mezivýsledku po navštívení uzlu `ARTISTS`

Pro správné fungování tečkové notace je též potřeba znát seznam výstupních sloupců této operace. Výstupními sloupci jsou veškeré sloupce tabulky `ARTISTS` převedené na instance třídy `SystemColumn`, viz následující výstup příkazové řádky V3:

```
[SystemColumn(ARTISTS.artist_id), SystemColumn(ARTISTS.artist_name),
SystemColumn(ARTISTS.description)]
```

Výstup z příkazové řádky V3: Seznam výstupních sloupců po navštívení uzlu `ARTISTS`

Při návratu zpět do uzlu `JoinNode` je tento mezivýsledek reprezentovaný instancí třídy `Query` lokálně uložen a seznam sloupců, viz V3, je přesunut na levý zásobník. Levý zásobník slouží pro ukládání seznamu výstupních sloupců, které vznikly navštívením levého podstromu daného uzlu. Tento seznam se musí uchovat pro pozdější zpracování.

#### Krok 2, 3 a 4 – Sestavení podmínky spojení

V dalších krocích se sestavuje podmínka spojení. Tato podmínka je sestavena až na závěr po navštívení uzlů `ARTISTS.artist_id` a `ALBUMS.artist_id`. Tyto uzly jsou instancemi třídy `ColumnNode` a při jejich návštěvě se vytváří instance třídy `ArbitraryColumn` nesoucí patřičné hodnoty. Po návratu do uzlu `ComparisonNode` se sestaví tříčlenný seznam, který je tvořen právě instancemi třídy `ArbitraryColumn`. Mezi tyto instance je ještě vložen objekt nesoucí informaci o použitém operátoru porovnání. Tento seznam je pak opět lokálně uložen v uzlu `JoinNode` pro pozdější použití.

#### Krok 5 – Navštívení uzlu ALBUMS

Po zpracování podmínky spojení se Visitor dostane až do uzlu `ALBUMS`. Jeho zpracování je shodné jako v kroku prvním, čili je vytvořen objekt třídy `Query` v jehož klauzuli `FROM` se nachází objekt `Relation` nesoucí název ALBUMS. Opět si objekt `Query` lze obdobným způsobem připodobnit k jednoduchému dotazu `SELECT` jako v případě P6. Seznam sloupců propagovaných do vyšší úrovně je odlišný a odpovídá seznamu veškerých sloupců tabulky `ALBUMS`, viz ukázka výstupu V4.

Při návratu zpět do uzlu `LeftNaturalAntiJoinNode` se opět uloží tento seznam na levý zásobník, který nyní obsahuje dva seznamy sloupců. Pokud by se tyto seznamy výstupních sloupců na zásobník neukládaly, přišli bychom v tomto momentu definitivně o znalost výstupních sloupců z kroku prvního,

```
[SystemColumn(ALBUMS.album_id), SystemColumn(ALBUMS.name),
SystemColumn(ALBUMS.note), SystemColumn(ALBUMS.price),
SystemColumn(ALBUMS.year), SystemColumn(ALBUMS.artist_id),
SystemColumn(ALBUMS.genre_id)]
```

Výstup z příkazové řádky V4: Seznam výstupních sloupců po navštívení uzlu `ALBUMS`

jelikož by došlo k jejich přepsání. Zároveň je lokálně uložena reference na objekt třídy `Query`, který byl v rámci návštěvy uzlu `ALBUMS` vytvořen.

### Krok 6 – Navštívení uzlu ALBUMS_TRACKS

Postup je totožný jako v předchozím kroku. Rozdíl si jistě každý dokáže představit. Seznam sloupců je pochopitelně odlišný, ale pro účely další ukázky není až tak podstatný, jak si ukážeme za okamžik. Tento seznam sloupců se při návratu zpět do rodičovského uzlu uloží na pravý zásobník, kde bude čekat na finální zpracování.

### Krok 7 – Závěrečné zpracování uzlu LeftNaturalAntiJoinNode

Než se pustíme do samotného rozboru zpracování tohoto uzlu, osvěžme si nejprve, jak je operace přirozeného anti joinu definována. Mějme relace `R` s atributy `A` a relaci `S` s atributy `B`, operaci levého přirozeného anti joinu definujeme takto (vzpomeňme na definici priorit):

$$\texttt{R !<* S} :=_{def} \texttt{R} \setminus \{\texttt{R * S}\}\texttt{[A]}$$

Tuto definici operace musí následovat i samotná transformace. Po návštěvě svých synovských uzlů disponuje Visitor dvěma instancemi třídy `Query`. Tyto dvě instance jsou spojitelné bez nutnosti převodu na poddotaz, neboť neobsahují selekci ani projekci. Označme instanci třídy `Query` levého potomka jako `R` a instanci třídy pravého potomka jako `S`, pak transformaci těchto objektů provedeme následovně:

1. Vytvoříme si hlubokou kopii instance `R` a označíme ji jako `R'`.
2. Instance `R` a `S` jsou spojitelné, proto požádáme o přirozené spojení těchto instancí tovární třídu `LangFactory`, která
   a) získá obsah klauzule `FROM` instance `S` – objekt třídy `Relation`, který označíme jako `r`,

   b) vytvoří nový objekt `n` třídy `NaturalJoin` jehož hodnotou bude objekt `r`,
   c) vytvořený objekt `n` vloží do klauzule `JOIN` instance `R`,
   d) takto zmodifikovanou instanci `R` vrátí jako výsledek aplikace přirozeného spojení.

3. Na zmodifikované instanci `R` provedeme implicitní projekci veškerých výstupních sloupců levého potomka, které jsou uloženy na vrcholu levého zásobníku – na základě instancí `SystemColumn` vytvoříme instance `OutputColumn` a ty vložíme do klauzule `SELECT` instance `R`.

4. Na závěr opět požádáme tovární třídu o vytvoření instance `M` třídy `Minus`, jehož levým operandem bude původní nezměněná instance `R'` a pravým operandem modifikovaná instance `R`.

Instance `M` reprezentuje výsledek operace levého přirozeného anti joinu podle výše uvedené definice. Na závěr ještě vybereme seznam sloupců z pravého zásobníku a ten zahodíme, jelikož dále nebude potřeba. Naopak seznam sloupců na vrcholu levého zásobníku teprve teď ze zásobníku vyjmeme, a tento seznam použijeme jako seznam výstupních sloupců, který budeme propagovat do vyšší úrovně. Pokud bychom měli v tuto chvíli objekt `M` přeložit do SQL, pak bychom získali následující překlad P7:

```
SELECT DISTINCT *
FROM ALBUMS
MINUS
SELECT DISTINCT album_id,
                name,
                note,
                price,
                year,
                artist_id,
                genre_id
FROM ALBUMS
NATURAL JOIN ALBUMS_TRACKS
```

Překlad P7: Odpovídající překlad objektu `M` do SQL

#### Krok 8 – Návrat a zpracování uzlu JoinNode

V tento okamžik má Visitor provádějící transformaci k dispozici:

1. objekt $Q_1$ třídy `Query`, který je výsledkem kroku 1,

2. seznam objektů C reprezentujících podmínku spojení,
3. objekt M třídy Minus, který je výsledkem kroku 7.

Transformace v tomto kroku je v rámci celého průchodu AST nejnáročnější. Objekt M se pro další použití musí ještě upravit, což má dopad i na výstupní sloupce, jak si ukážeme za chvíli. Dále je potřeba znormalizovat podmínku spojení, tj. veškeré instance třídy ArbitraryColumn převést na odpovídající OutputColumn instance tak, aby jejich překlad do SQL byl korektní. Na závěr je potřeba automaticky přejmenovat duplicitní názvy sloupců, které vzniknou spojením instancí $Q_1$ a M. Z tohoto důvodu celkovou transformaci rozdělíme do několika dílčích kroků, které na sebe budou vzájemně navazovat.

**Dílčí krok 8.1** – Transformace objektu M – Instanci třídy Minus tak, jak ji máme navrženou, nelze snadno spojit s instancí $Q_1$. Je to logické. Proto, abychom s množinovou operací mohli dále pracovat (připojit ji k jinému výsledku), je potřeba tento objekt převést na poddotaz v klauzuli FROM nové instance třídy Query. Dvě instance třídy Query již totiž spojit umíme, viz krok 7. Proto:

1. Požádáme tovární třídu o vytvoření instance S třídy Subquery, jejíž hodnotou bude objekt M.
2. Instanci S přiřadíme alias (pojmenování poddotazu je potřeba kvůli tečkové notaci) – v tomto případě je automaticky přiřazen alias R1.
3. Jelikož se objekt M stal součástí poddotazu S s aliasem R1, je potřeba zaktualizovat vazby výstupních sloupců získaných z kroku 7 – vazba sloupců na původní tabulku ALBUMS je archivována a aktuální vazba je nastavena na poddotaz S s aliasem R1, viz výstup V5 (porovnej rozdíl oproti V4).
4. Na závěr požádáme tovární třídu o vytvoření instance $Q_2$ třídy Query, do jejíž klauzule FROM umístíme poddotaz S (instance $Q_1$ a $Q_2$ zatím nespojujeme).

```
[SystemColumn(R1.album_id), SystemColumn(R1.name),
SystemColumn(R1.note), SystemColumn(R1.price),
SystemColumn(R1.year), SystemColumn(R1.artist_id),
SystemColumn(R1.genre_id)]
```

Výstup z příkazové řádky V5: Upravený seznam výstupních sloupců z kroku 7 po převodu na poddotaz

Takto upravený seznam sloupců se teprve v tento okamžik dostává na vrchol pravého zásobníku. Na vrcholu levého zásobníku máme nezměněný seznam sloupců V3 a na vrcholu pravého zásobníku upravený seznam V5.

**Dílčí krok 8.2** – Normalizace podmínky `C` – V rámci kroků 2–4 vznikla podmínka, která je tvořena dvěma objekty `ArbitraryColumn(ARTISTS.artist_id)` a `ArbitraryColumn(ALBUMS.artist_id)` přesně v tomto pořadí. Tuto podmínku nyní musíme normalizovat, tj. převést na odpovídající instance třídy `OutputColumn`.

Normalizace podmínky tedy probíhá oproti sloupcům uloženým na vrcholech levého i pravého zásobníku. Zde hraje významnou roli generování kombinací názvů sloupců tak, jak ji popisuji v podkapitole 4.9.1. Jelikož tuto funkcionalitu obě třídy `SystemColumn` a `ArbitraryColumn` implementují, nalezení shody pro sloupec `ARTISTS.artist_id` je přímočaré, jelikož takový systémový sloupec se nachází v seznamu V3, který je umístěný na vrcholu levého zásobníku.

Ve druhém případě nalezení shody tak přímočaré není. I přes to shoda nalezena je. Odpovídající systémový sloupec je nyní nalezen v seznamu V5 na vrcholu pravého zásobníku. Označme sloupec `ArbitraryColumn(ALBUMS.artist_id)` z podmínky `C` jako `a` a sloupec `SystemColumn(R1.artist_id)` z vrcholu pravého zásobníku jako `s`. Porovnání se provádí porovnáním množin kombinací, během něhož se zjišťuje, zda jsou tyto množiny disjunktní. Pro sloupec `a` byla vygenerována následující jednoprvková množina kombinací {`ALBUMS.artist_id`} a pro sloupec `s` dvou prvková množina {`artist_id, ALBUMS.artist_id`}. Mezi těmito množinami je nalezen průnik a proto je nalezena i shoda mezi sloupci `a` a `s`.

Pozorný čtenář by se mohl ptát, proč množina kombinací sloupce `s` neobsahuje možnost `R1.artist_id`? Odpověď je prostá. Relace (poddotazy) s aliasy `R1, ..., Rx` jsou generovány interně. Počet těchto relací a jejich číslování se mění s ohledem na počet poddotazů, které je během transformace potřeba vytvořit. Jinými slovy pokud by student, nebo kdokoliv jiný, upravil souhrnný dotaz D4 a to tak, že by přidal například projekci nad tabulkou `ALBUMS_ARTISTS`, pak by se v kombinacích objevila relace `R2` a nikoliv `R1`, jelikož první pojmenovaný poddotaz by musel být vytvořen již v kroku 7. Z tohoto důvodu není podporováno použití těchto prefixů v rámci projekce, selekce a spojení, neboť jejich výskyt není „stabilní“.

Při shodě pak tovární třída zajistí vytvoření odpovídající instance `OutputColumn`, jejíž hodnoty vychází z hodnot sloupců, mezi kterými shoda byla nalezena.

Tímto je podmínka znormalizována. Do výstupu se vždy dostane pouze nejaktuálnější vazba, která převzata ze sloupce `s`, tj. místo `ALBUMS.artist_id` se v podmínce spojení objeví `R1.artist_id`.

**Dílčí krok 8.3** – Spojení dvou instancí třídy `Query` – V tento okamžik má Visitor k dispozici instanci $Q_1$ třídy `Query`, která vznikla v kroku 1 a instanci té samé třidy $Q_2$, která vznikla v dílčím kroku 8.1. Spojení těchto instancí probíhá podobně jako v kroku 7. Nejprve je požádána tovární třída o obecné spojení instancí $Q_1$ a $Q_2$ na základě již znormalizované podmínky, kterou označíme jako `C'`. Výsledkem je zmodifikovaná instance $Q_1$, do jejíž klauzule `JOIN` přibyla instance třídy `InnerJoin`. Hodnotou instance třídy `InnerJoin` je pak poddotaz `S`, který byl vytvořen v dílčím kroku 8.1 a který byl uložen v klauzuli `FROM` instance $Q_2$.

**Dílčí krok 8.4** – Implicitní projekce – Bohužel v tomto kroku se nám, jak na levé, tak na pravé straně objevil duplicitní název sloupce `artist_id`. Aby bylo možné s tímto sloupcem dále pracovat, je potřeba jeho jeden výskyt přejmenovat v rámci implicitní projekce. Jinými slovy, Visitor při sestavování seznamu výstupních sloupců zjistí onu duplicitu. Vrchol levého zásobníku vyprázdní a tento seznam sloupců se stane součástí výstupního seznamu sloupců bez jakékoliv změny. Naproti tomu u duplicity v seznamu na pravé straně je potřeba zvýšit číslo nesoucí hodnotu suffixu z nuly na jedničku. Tuto změnu pak reflektuje samotný překlad v němž se v klauzuli `SELECT` objeví následující formule: `R1.artist_id AS artist_id_1`. Výsledný výstupní seznam sloupců je tak rozšířen hodnotami z pravého zásobníku, z nichž jeden sloupec byl dodatečně modifikován. Po této operaci jsou oba zásobníky prázdné.

#### Krok 9 – Projekce

Finálním krokem v celé transformaci je projekce. Výsledkem průchodu levého podstromu je instance třídy `Query`, která byla modifikována v dílčím kroku 8.3. Nad touto instancí není možné provést projekci, neboť zde v kroku 8.4 byla provedena implicitní projekce. V tomto případě se musí opět tato instance třídy `Query` převést na pojmenovaný poddotaz a upravit aktuální vazby dostupných sloupců.

Výsledkem průchodu pravého podstromu je seznam obsahující tři instance třídy `ArbitraryColumn`. Následně opět probíhá normalizace tohoto seznamu oproti seznamu všech dostupných (a aktualizovaných) sloupců. Ve všech případech je shoda na základě kombinací nalezena. V klauzuli `SELECT` se tedy dané sloupce objeví ve své nejaktuálnější podobě.

## 4.10 Tvorba překladu

Poslední fází z celého procesu je samotná tvorba překladu. Jak jsem již naznačoval, výstupem transformace AST je jiná hierarchická objektová struktura, která vzdáleně modeluje jediný dotaz SELECT jazyka SQL. Každý prvek (objekt) této hierarchické struktury opět dědí od již několikrát zmiňované třídy

`Visitable`. Jinými slovy, každý element této struktury je zase navštívitelný instancí třídy `Visitor`.

Jelikož cílem této diplomové práce je realizovat překlad do třech různých dialektů jazyka SQL, intuice mě vedla k tomu, aby pro jednotlivé dialekty existovala podtřída třídy `Visitor`.

V drtivé většině je SQL výstup velmi podobný mezi jednotlivými SQL dialekty (RDBMS), proto vznikla abstraktní třída `SQLVisitor`, která implementuje společné funkcionality. Od této třídy pak teprve dědí třídy `OracleVisitor`, `PostgreSQLVisitor` a `MariaDBVisitor`, které dodefinovávají chování v situacích, kdy se jednotlivé SQL dialekty odlišují.

Výsledkem průchodu touto hierarchickou strukturou je pak řetězec, který obsahuje ekvivalentní vyjádření dotazu RA formou SQL příkazu SELECT.

Použitím návrhového vzoru Vizitor můžeme tuto hierarchickou strukturu procházet i několikrát za sebou různými Visitory, a jejich výsledky přitom nejsou navzájem ovlivňovány. Další výhodou je připravenost pro další rozšíření. Pokud by se v budoucnu například objevil požadavek na rozšíření o překlad do MS-SQL, pak by mělo stačit pouze rozšířit překladač o novou podtřídu třídy `Visitor` za předpokladu, že struktura dotazu a množina operací v MS-SQL se diametrálně neliší od operací definovaných ostatními RDBMS. V opačném případě by přidání podpory překladu pro tento dialekt mělo dopad i na samotnou transformaci AST.

### 4.10.1 Specifika jednotlivých RDBMS

V této podkapitole se budu věnovat rozdílům jednotlivých RDBMS, jejich dopadům na tvorbu překladu a způsobům, jakými se překladač s těmito rozdíly vyrovnal.

#### 4.10.1.1 Oracle

Jako výchozí (základní) překlad byl zvolen překlad do SQL spustitelného v RDBMS Oracle. Jeho korektnost je bezpodmínečná obdobně tak, jako tomu bylo v případě mé bakalářské práce. Výhradním RDBMS, se kterým studenti během výuky předmětu Databázové systémy (BI-DBS) přichází do styku a na němž je koncipovaná celá výuka, je právě RDBMS Oracle. Z tohoto důvodu byla tvorba překladu právě pro tento systém upřednostněna před ostatními. V kontrastu RDBMS Oracle pak budou porovnávány specifika ostatních RDBMS (PostgreSQL a MariaDB).

#### 4.10.1.2 PostgreSQL

Chování RDBMS PostgreSQL oproti RDBMS Oracle je velice podobné až na pár odlišností. Množina operací podporovaných tímto RDBMS je, s ohledem na definované operace RA, naprosto totožná, jako je tomu v případě RDBMS Oracle.

První objevený rozdíl se nachází v klíčovém slově pro vyjádření operace množinového rozdílu – místo klíčového slova `MINUS` se musí použít klíčové slovo `EXCEPT` [27].

O něco zásadnějším rozdílem oproti RDBMS Oracle je nutnost pojmenovávat veškeré poddotazy v klauzuli `FROM` [28]. Během testování se však ukázalo, že PostgreSQL kromě pojmenovávání poddotazů v klauzuli `FROM` vyžaduje také pojmenovávání poddotazů v klauzuli `JOIN`. V obou případech by to byl velice zásadní problém, pokud by překladač neřešil problém tečkové notace. Problém tečkové notace pojmenování poddotazů v klauzulích `FROM` a `JOIN` vyžaduje, čímž je rozdíl těchto dvou RDBMS odstraněn.

Největší rozdíl dvou jmenovaných RDBMS se ukrývá ve vyhodnocování množinových operací v rámci dotazu SELECT. Oracle definuje vyhodnocování množinových operací následovně: „*All set operators have equal precedence. If a SQL statement contains multiple set operators, then Oracle Database evaluates them from the left to right unless parentheses explicitly specify another order*“ [29]. Naproti tomu PostgreSQL definuje vyhodnocování operace množinového průniku následovně: „*Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. INTERSECT binds more tightly than UNION. That is, A UNION B INTERSECT C will be read as A UNION (B INTERSECT C)*“ [27]. Jinými slovy, při vyhodnocování množinových operací má operace množinového průniku v PostgreSQL vždy vyšší prioritu před operací množinového sjednocení. Aby tento rozdíl byl eliminován, tj. aby byla dodržena stanovená definice priorit vyhodnocování operací RA (viz 2.2.2), určuje překladač prioritu vyhodnocování množinových operací explicitním přidáním kulatých závorek v situacích, kdy je detekováno použití vícero množinových operací za sebou. Pro názornost si to ukážeme na příkladu. Mějme dotaz D5 a překlad tohoto dotazu P8:

$$\texttt{ALBUMS[album_id]} \cup \texttt{ALBUMS_TRACKS[album_id]} \cap \texttt{ALBUMS_STORES[album_id]} \quad \text{(D5)}$$

```
(
    SELECT DISTINCT album_id
    FROM ALBUMS
    UNION
    SELECT DISTINCT album_id
    FROM ALBUMS_TRACKS
)
INTERSECT
SELECT DISTINCT album_id
FROM ALBUMS_STORES;
```

Překlad P8: Překlad dotazu D5 pro PostgreSQL

Na překladu P8 vidíme, že priorita, tak jak jsme ji definovali, je dodržena. Operace projekce má prioritu nejvyšší a binární operace jsou vyhodnocovány zleva doprava. Přesného určení priorit vyhodnocování množinových operací docílil překladač přidáním závorek obalující operaci `UNION`.

#### 4.10.1.3 MariaDB

Z historického hlediska se RDBMS MySQL, respektive MariaDB nejvíce odlišovala od předchozích dvou. S vydáním MariaDB verze 10.3, jejíž první stabilní verze byla vydána v květnu 2018, přibyly dvě zásadní operace, které v předchozích verzích chyběly. Těmito operacemi jsou množinový průnik a množinový rozdíl. Tyto dvě operace doplnily již existující operaci množinového sjednocení [30]. Bez operací množinového průniku a rozdílu by tvorba překladu, respektive transformace AST, musela být značně pozměněna. Například operace množinového průniku by byla vyjádřitelná pomocí operací spojení.

Bohužel se však ukázalo, že MariaDB ve verzi 10.3 neumožňuje použití kulatých závorek v kombinaci s množinovými operacemi. Jejich použití ústí v chybu v syntaxi. V předchozí podkapitole jsme si vysvětlili, že použití závorek je nutností pro dodržení definované priority, neboť RDBMS PostgreSQL určuje prioritu množinových operací odlišným způsobem. V případě MariaDB má i tento problém řešení! Od verze 10.4, která je momentálně ve vývoji, a jejíž vydání se v brzké době očekává, přibyde podpora použití závorek pro explicitní určení priority vyhodnocování množinových operací. V době psaní této diplomové práce je poslední vydanou verzí verze 10.4.4, která byla vydána 7. dubna 2019, a která je označena jako „Release Candidate“ [31].

Dalším drobným rozdílem v SQL mezi RDBMS MariaDB a Oracle, respektive PostgreSQL, je rozdíl ve funkcích pro konverzi řetězců na datum a jejich formátovacích řetězcích. Zatímco RDBMS Oracle a PostgreSQL umožňují použití funkce `TO_DATE()` se shodným formátovacím řetězcem (`dd.mm.yyyy`), pak v MariaDB se musí použít funkce `STR_TO_DATE()` s odlišným formátovacím řetězcem. Další rozdíly jsem oproti předchozím dvěma RDBMS již nenalezl.

S ohledem na předpokládané brzké vydání nové verze, která by měla odstranit nejzávažnější nedostatky, které byly zmíněny v této podkapitole, jsem se rozhodl pro ponechání současného stavu překladů.

Drtivá většina operací, které se v překladu mohou vyskytnout, je RDBMS MariaDB již plně podporována. **K úplné kompatibilitě překladače s RDBMS MariaDB dojde až s vydáním verze 10.4**. Tato specifická vlastnost překladače byla vedoucím diplomové práce akceptována, neboť podpora překladu do RDBMS MariaDB není natolik kritická, jak je tomu v případě RDBMS Oracle.

### 4.10.2 Změna formátování

Příjemným zlepšením nového překladače je změna ve formátování výsledného SQL dotazu. Původní překladač spoléhal na knihovnu třetí strany, na knihovně sqlparse jazyka Python. Tato knihovna dokázala zformátovat řetězec obsahující SQL dotaz tak, aby byl lépe čitelný a nebyl součástí jediného řádku. Ne vždy byl výstup úplně optimální jako v případě překladu dotazu D5. Odpovídající překlad původního překladače je zachycen ukázkou překladu P9

```
(
   (SELECT DISTINCT album_id
    FROM ALBUMS)
 UNION
   (SELECT DISTINCT album_id
    FROM ALBUMS_TRACKS)) INTERSECT
  (SELECT DISTINCT album_id
   FROM ALBUMS_STORES);
```

Překlad P9: Výstup původního překladače zobrazený na Portálu pro dotaz D5

Ve výsledku překladu P9 není úplně na první pohled zřejmé, kde přesně se nachází uzavírací závorka k závorce na první řádce. Dále zde chybí konzistence v odsazení a odřádkování klíčových slov množinových operací, klíčové slovo `UNION` je zcela jinak odsazeno než klíčové slovo `INTERSECT`. Letmým pohledem není ani zřejmé, která ze dvou množinových operací má vyšší prioritu, byť je specifikována pomocí kulatých závorek.

Díky použitému návrhovému vzoru Visitor bylo možné definovat vlastní způsob formátování, který by měl odstranit nedostatky, které byly zmíněny v předchozím odstavci. Třída `SQLVisitor` si tedy sama řídí formátování – přidává zalomení řádků na patřičná místa a hlavně jednotně upravuje odsazení začátků řádků veškerých poddotazů. Výsledek překladu dotazu D5 v novým překladačem je zachycen na ukázce překladu P8. Porovnání obou výstupů nechávám plně na čtenáři. Věřím, že vylepšené formátování, viz překlad P8, ocení jak studenti, tak i učitelé.

### 4.10.3 Příklad

Nyní se konečně dostáváme k samotnému výsledku celého procesu překladu. Tento výsledek je reprezentován překladem P10 pro RDBMS Oracle na následující straně.

Pokud by zvídavého čtenáře zajímalo, jak by vypadal překlad do RDBMS PostgreSQL, respektive MariaDB, pak by mělo stačit nahradit pouze klíčové slovo `MINUS` klíčovým slovem `EXCEPT` a dotaz by se měl stát rázem spusti-

telným v obou RDBMS. Nutno připomenout, že podpora operace `EXCEPT` v MariaDB byla přidána až ve verzi 10.3, tudíž na starších verzích by tento dotaz spustitelný být neměl.

```
SELECT DISTINCT artist_id,
                artist_name,
                R2.name
FROM (
    SELECT DISTINCT ARTISTS.artist_id,
                    ARTISTS.artist_name,
                    ARTISTS.description,
                    R1.album_id,
                    R1.name,
                    R1.note,
                    R1.price,
                    R1.year,
                    R1.artist_id AS artist_id_1,
                    R1.genre_id
    FROM ARTISTS
    JOIN (
        SELECT DISTINCT *
        FROM ALBUMS
        MINUS
        SELECT DISTINCT album_id,
                        name,
                        note,
                        price,
                        year,
                        artist_id,
                        genre_id
        FROM ALBUMS
        NATURAL JOIN ALBUMS_TRACKS
    ) R1 ON ARTISTS.artist_id = R1.artist_id
) R2;
```

Překlad P10: Výsledek překladu dotazu D4 do SQL pro RDBMS Oracle

Z překladu P10 by nyní měli být úplně zřejmé některé aspekty, které jsem se snažil vysvětlit v rámci transformace AST v podkapitole 4.9. Předně je to problém tečkové notace, který výsledek překladu plně respektuje:

1. Veškeré poddotazy klauzulích `FROM` a `JOIN` jsou pojmenovány.

2. Podmínka spojení tak, jak byla definována v dotazu D4 byla upravena, aby reflektovala aktuální rozsah, ve kterém se tabulka `ALBUMS` již nenachází – tabulka se nachází v klauzuli `FROM` poddotazu `R1`.

3. Překlad tečkové notace použité v projekci dotazu D4 opět odráží aktuální rozsah, ve kterém už opět není tabulka `ALBUMS`, ale poddotaz `R2`.

Jediná operace přejmenování, která je na výstupu patrná, je důsledkem řešení problému nutnosti automatického přejmenování duplicitních názvů sloupců. Pro získání hodnot tohoto přejmenovaného sloupce je v rámci projekce v dotazu D4 možné použití dvojitého zápisu: `ALBUMS.artist_id`, respektive `artist_id_1`.

Pokud bychom výsledek překladu P10 pustili nad námi definovaným schématem, pak na základě dat v něm uložených bychom dostali následující výsledek:

| artist_id | artist_name | name |
|---:|---|---|
| 10 | Robbie Williams | Intensive Care |
| 14 | Justin Timberlake | Justified |
| 7 | U2 | The Best of 1990-2000 |
| 10 | Robbie Williams | Greatest Hits |

Tabulka 4.1: Výsledek po spuštění dotazu SELECT z překladu P10

## 4.11 Shrnutí

Od začátku kapitoly Realizace až doposud jsme se zabývali jednotlivými částmi překladače a jejich vzájemnými vztahy. Nyní si celý proces shrneme a podíváme se na požadavky, které byly v rámci předchozích kapitol splněny.

Než-li se překladač pustí do zpracování dotazu RA, je potřeba jej prvně nakonfigurovat. V rámci konfigurace se překladači musí předat databázové schéma, nad nímž se daný dotaz vyhodnocuje, cílový SQL dialekt ovlivňující výslednou podobu SQL dotazu a i dotaz RA samotný. Překladač si načítá též svou konfiguraci z konfiguračních souborů a inicializuje si veškeré své potřebné komponenty.

Po úspěšné inicializaci se spouští kompletní proces překladu. Dotaz RA je nejprve předán lexikálnímu analyzátoru, který se jej pokusí převést na posloupnost tokenů. Pokud se mu to povede, tato posloupnost se předá syntaktickému analyzátoru, který určí, zda je daná posloupnost tokenů validní (syntakticky správná). Výsledkem syntaktické analýzy je abstraktní syntaktický strom. Tento strom se pak prochází instancí třídy `TransformationVisitor`, která transformuje jeho strukturu do nové hierarchické podoby. Během průchodu pak dochází k interní manipulaci se sloupci, aby bylo možné pokaždé určit rozsah sloupců, které se v dané úrovni nachází, čímž se řeší problém tečkové notace. Tato hierarchická struktura, která modeluje strukturu dotazu SELECT, se následně prochází další instancí třídy `Visitor`, která danou strukturu serializuje do řetězce obsahující výsledný SQL dotaz. Výsledek

překladu je pak spustitelný v rámci RDBMS, jehož typ byl specifikován pomocí předaného SQL dialektu.

V případě, že překladač narazí v jakékoliv části na chybu, tuto chybu zapíše do patřičného log souboru a problém oznámí uživateli.

Zaměříme-li se na specifikované funkční požadavky, pak všechny z nich byly splněny. Jednotlivě:

- Funkční požadavek **FP01** na realizaci překladu z RA do SQL **byl splněn** – výsledkem této diplomové práce je nový překladač z relační algebry do SQL – jednotlivé kroky tvorby překladu byly v předchozích kapitolách podrobně rozebrány.

- Funkční požadavek **FP02** na rozpoznávání veškeré syntaxe vyučované v rámci předmětu Databázové systémy (BI-DBS) **byl splněn**, byť to nebylo explicitně dokázáno – překladač vychází z předchozí verze, která tento požadavek splňovala, nová verze množinu rozpoznatelných operací nezúžila, ba naopak ji rozšířila, jak si ukážeme v kapitole Další funkcionality.

- Funkční požadavek **FP03** vyžadující striktní dodržení priorit při vyhodnocování operací **byl splněn**, viz 4.9.2.1.

- Funkční požadavek **FP04** na automatické přejmenování duplicitních názvů sloupců **byl splněn**, viz sekce 4.9.1.2 a výsledek překladu P10.

- Funkční požadavek **FP05** na řešení problému tečkové notace **byl splněn**, viz sekce 4.9.1.1 a výsledek překladu P10.

- Funkční požadavek **FP06** pro překlad do RDBMS Oracle, PostgreSQL a MariaDB **byl splněn s akceptovaným omezením** pro překlad do SQL RDBMS MariaDB – tento požadavek bude kompletně splněn s oficiálním vydáním RDBMS MariaDB verze 10.4.

Z nefunkčních požadavků byl zatím splněn pouze požadavek **NP04** na využití návrhových vzorů. Použitím návrhových vzorů se věnuji v sekci 4.2, z nichž návrhový vzor Visitor nás pak provázel téměř celou realizací. Zbylým, doposud nesplněným nefunkčním požadavkům, se budu věnovat v následujících podkapitolách.

## 4.12 Testování

Dalším formulovaným nefunkčním požadavkem byl požadavek **NP03**. Pro splnění tohoto požadavku bylo nutné ověřit funkčnost a korektnost překladů pomocí testů.

V původní implementaci překladače byla sada testů již přítomna. Pomocí regresních testů se oproti databázi testovala korektnost překladů, tj. byla vytvořena množina základních SQL dotazů, které se spolu s výslednými překlady pouštěly v databázi a porovnávaly se výsledky těchto dvojic dotazů. Další skupinou testů byly jednotkové testy, které ověřovaly funkcionalitu klíčových komponent. Jako poslední bylo přidáno testování API. Toto testování nebylo řešeno šťastným způsobem, neboť bylo prováděno pomocí shellového skriptu a vyžadovalo uživatelovu interakci – vyhodnocení těchto testů se provádělo vizuálně [1].

V rámci implementace nového překladače byly tyto předchozí skupiny sloučeny do jedné. O veškeré testování korektnosti nového překladače se starají pouze jednotkové testy, jejichž definici umožňuje nativní knihovna jazyka Python, knihovna `unittest`.

Regresní testy nebyly přenášeny z původní implementace hned ze dvou důvodů. Prvně z důvodu bezpečnosti. Překladač si nyní nemusí uchovávat přihlašovací údaje do testovací databáze, tudíž se v žádném souboru nevyskytují citlivé údaje jako je přihlašovací jméno a heslo. Dále pak odpadla závislost na další knihovně a ovladači, který umožňoval překladači připojení do databáze.

Samotné testování API pomocí shellového skriptu bylo kompletně nahrazeno testováním pomocí jednotkových testů.

Veškeré sady jednotkových testů jsou pak nalezitelné v adresáři `tests/unit` v obsahu GIT repozitáře nového překladače. Část těchto testů, zejména ověřující korektnost lexikální a syntaktické analýzy, mohla být převzata z původní implementace téměř beze změny. Tato sada testů byla pak rozšířena o další skupiny ověřující zejména korektnost překladu.

Typický test ověřující korektnost překladu definuje dotaz RA, který se předává překladači a výsledek, který je očekáván. Samotný výsledek je pak porovnán na rovnost s očekávaným výstupem. Tímto způsobem bylo definováno celkem **1147** jednotkových testů, které ověřují základní kombinace RA dotazů, respektive jejich překladů. Každý takový test pak obsahuje tři dílčí testy, které ověřují překlad pro RDBMS Oracle, PostgreSQL a MariaDB.

Testování API se podařilo též zahrnout do jednotkových testů. V dokumentaci knihovny Flask, na níž je API překladače postaveno, jsem nalezl dostatek informací, které mi pomohly v převodu původního způsobu testování na nový. V rámci testů rozhraní se testuje forma jednotlivých požadavků a odpovědí, včetně jejich obsahu. Vyhodnocení pak probíhá strojově pomocí knihovny `unittest`, čímž bylo eliminováno riziko přehlédnutí chyby které bylo s původním způsobem testování úzce spjato.

Celkem se tedy o ověření korektnosti překladače stará **1419** jednotkových testů, které pokrývají testy rozhraní (část předcházející překladu), proces překladu a zejména ověřují jeho výsledek. Nefunkční požadavek **NP03** můžeme tedy označit za splněný.

## 4.13 Měření výkonu

Aby bylo možné splnit nefunkční požadavek **NP02**, který definuje nároky na odezvu překladače, bylo potřeba provést měření jeho výkonu. Samotná odezva byla měřena na lokálním i produkčním prostředí (serveru dbs.fit.cvut.cz). Jednotlivé hardwarové specifikace prostředí jsou uvedené níže.

Na lokálním prostředí probíhalo měření výkonu v rámci 64-bitového operačního systému Ubuntu Budgie verze 18.04 s touto konfigurací:

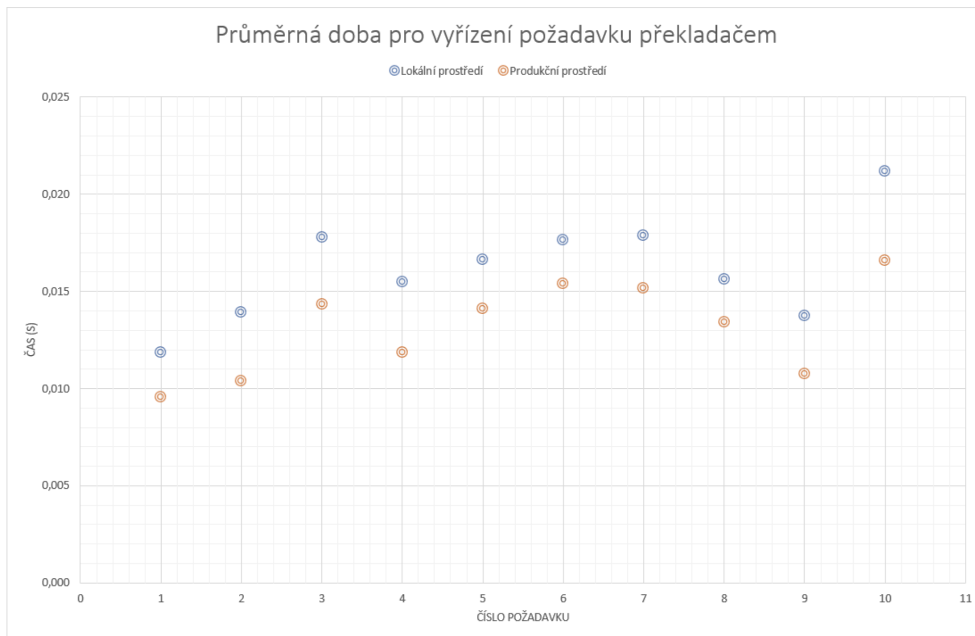
- Procesor: Intel(R) Core(TM) i5–6200U CPU @ 2.30GHz (4 CPUs)
- Operační paměť: 8192 MB
- Pevný disk: SSD KINGSTON RBU - SNS8152S3256GG2

O produkčním prostředí se mi nepodařilo zjistit veškeré informace, zejména o použitém pevném disku. Víme, že celé prostředí serveru dbs.fit.cvut.cz běží ve virtuálním prostředí pod 64-bitovým operačním systémem Debian 9.5 Stretch. Hardwarová konfigurace je pak následující:

- Procesor: Intel(R) Xeon(R) E5–2630 CPU @ 2.30GHz (4 CPUs)
- Operační paměť: 8192 MB
- Pevný disk: Neznámý, pravděpodobně klasický plotnový pevný disk

V obou případech se testovala poslední vydaná verze překladače a to verze 2.2.0. Pro účely testování byl zvolen mód `prod`, který neprodukuje tolik výstupu do logových souborů a reprezentuje jediný mód, ve kterém by měl překladač v rámci Portálu běžet. Pro obě prostředí byl použit stejný testovací skript, napsaný v jazyce Python verze 3.5.3. Samotné měření časů mezi odesláním požadavku a přijetím odpovědi bylo provedeno pomocí standardní knihovny time. Na obou prostředích probíhalo testování překladače skrze webový server Nginx a aplikační server Phusion Passenger.

Pro účely testování jsem připravil sadu deseti různorodých, typických, dotazů v RA, které zahrnovaly různé počty relací, operací, změnu priority, apod. Každý z těchto deseti dotazů byl na obou prostředích poslán tisíckrát překladači k vyhodnocení. Samotné měření doby běhu začínalo před odesláním požadavku a končilo po přijetí odpovědi. Na základě tisíce výsledků byl stanovený průměrný čas pro vyřízení daného požadavku, tyto průměrné časy jsou pak zobrazeny na grafu 4.9.




Obrázek 4.9: Graf průměrných hodnot pro vyřízení daného požadavku překladačem v závislosti na cílovém prostředí

Z výše uvedeného grafu vyplývá, že průměrná doba pro vyřízení daného požadavku nepřesahuje **tři setiny sekundy** na lokálním prostředí, na produkčním prostředí nedosahuje průměrná doba **dvěma setinám sekundy**. Zdrojová data tohoto grafu jsou zobrazena v tabulce 4.2. Veškeré časy zde uvedené jsou v souladu s grafem uvedené v sekundách:

| # | R | D | Z | $T_{avg}^{loc}$ | $T_{avg}^{prod}$ | Poznámka |
|---|---|---|---|---|---|---|
| 1 | 2 | 0 | Ne | 0.011893 | 0.009545 | |
| 2 | 6 | 4 | Ne | 0.013928 | 0.010378 | |
| 3 | 3 | 0 | Ne | 0.017786 | 0.014318 | |
| 4 | 4 | 0 | Ano (1x) | 0.015517 | 0.011870 | |
| 5 | 3 | 0 | Ano (1x) | 0.016655 | 0.014081 | Dotaz D4 – komplexní |
| 6 | 3 | 0 | Ne | 0.017647 | 0.015401 | Dotaz D5 |
| 7 | 5 | 0 | Ano (2x) | 0.017854 | 0.015186 | |
| 8 | 2 | 0 | Ne | 0.015658 | 0.013415 | Relační dělení – ÷ |
| 9 | 6 | 0 | Ano (2x) | 0.013756 | 0.010730 | |
| 10 | 8 | 5 | Ne | 0.021188 | 0.016620 | Relační dělení – deklarace |

Tabulka 4.2: Průměrné časy pro vyřízení požadavku (# – č. testu, R – počet relací, D – počet deklarovaných relací, Z – změna priority, $T_{avg}^{loc}$ – prům. čas na lokálním prostředí, $T_{avg}^{prod}$ – prům. čas na produkčním prostředí)

Dále v rámci všech měření byl zaznamenáván i maximální a minimální absolutní čas pro vyřízení jednoho konkrétního požadavku. Tyto hodnoty jsou pak zachyceny v tabulce 4.3. Na jednotlivé časy uvedené v sekundách v této tabulce má zásadní vliv aktuální zatížení daného operačního systému v rámci testovaného prostředí. Z tohoto důvodu je maximální čas pro zpracování požadavku na produkčním prostředí o něco vyšší oproti lokálnímu prostředí, než jak by člověk na základě dat z tabulky 4.2 čekal.

| | $T_{max}$ | $T_{min}$ |
|---|---|---|
| Lokální prostředí | 0.037458 | 0.010015 |
| Produkční prostředí | 0.039344 | 0.006502 |

Tabulka 4.3: Maximální a minimální absolutní časy pro vyřízení požadavku

Měřením se ukázalo, že v průměrném případě dosahuje překladač lepší doby odezvy na produkčním prostředí. Rozdíl oproti lokálnímu prostředí je však zanedbatelný.

Na základě výše uvedených měření lze uzavřít požadavek **NP02** jako splněný, neboť se neukázalo, že by vyřízení jediného požadavku trvalo déle než jednu, natož tři vteřiny.

## 4.14 Rozhraní překladače – API

Implementace rozhraní překladače doznala pouze drobných změn. Prvně je potřeba říci, že stejně jako tomu bylo u původní implementace překladače, je i rozhraní nového překladače implementováno s využitím knihovny Flask. Veškerá komunikace s překladačem je možná pouze ve formátu JSON.

Struktura samotných požadavků se oproti původní implementaci rozhraní vůbec nezměnila. Metoda, přijímající požadavky pro překlad, očekává JSON objekt nesoucí tři základní informace. Požadovanými informacemi v takovém požadavku jsou:

1. Dotaz relační algebry zadaný uživetelem Portálu (v případě víceřádkového dotazu musí každá řádka být oddělena znakem `\n`).
2. SQL dialekt – název RDBMS – do kterého má být překlad proveden, možné hodnoty jsou `oracle`, `postgresql`, `mariadb`.
3. Databázové schéma – vnořený JSON objekt jehož klíči jsou názvy tabulek a hodnotami jsou seznamy sloupců daných tabulek.

Struktura odpovědí byla lehce pozměněna. Za prvé, byla rozšířena o další položku, více viz 5.5. Za druhé, struktura chybových zpráv byla ještě podrobněji rozčleněna. Struktura typické odpovědi překladače se skládá z následujících částí:

1. status odpovědi – výsledek překladu – možné hodnoty `success`, `warning`, `error`,
2. výsledný řetězec nesoucí předformátovaný SQL dotaz (pouze v případě, kdy je status odpovědi `success` nebo `warning`),
3. seznam objektů, kde každý objekt je zpráva, kterou je potřeba zobrazit uživateli – tyto zprávy informují uživatele o chybách, či varováních, které se při překladu objevily,
4. seznam hirerachických objektů reprezentující prováděcí stromy RA dotazu, více viz 5.5.

Dalším drobným rozdílem je číslo portu, na kterém překladač implicitně naslouchá. Aby byla umožněna koexistence dvou implementací překladače v rámci Portálu – původního překladače a i toho nového – muselo být novému překladači přiděleno jiné číslo portu, aby se předešlo kolizím. Z tohoto důvodu nový překladač implicitně naslouchá na portu číslo `5002`, zatímco původní překladač naslouchá na portu `5000`.

Poslední změnou byla úprava metody požadavku, která se pro volání metody rozhraní překladače `http://127.0.0.1:5002/translate` očekává. Tato adresa je jedinou možnou adresou, pod kterou lze překladači předávat požadavky k překladu. Na rozdíl od původní implementace se očekává použití metody `PUT` místo `POST`. Pokud by byla použita jiná metoda, pak je vrácena chybová odpověď s kódem `400 BAD REQUEST`.

Oproti původnímu řešení byla dokumentace API přesunuta ze souboru ve formátu LaTeX na webový portál Apiary. Apiary je původně český projekt, který později koupila společnost Oracle. Tento portál umožňuje velmi jednoduché modelování aplikačního rozhraní. Veškerý zápis jednotlivých metod, požadavků a odpovědí se zapisuje ve formátu Markdown. Kompletní dokumentace API, včetně popisu ukázkových požadavků a odpovědí, je veřejně přístupná na adrese https://rat2.docs.apiary.io/.

Na základě informací uvedených výše lze považovat nefunkční požadavek **NP01** (přístupnost skrze API) a **NP07** (dokumentace API), za splněný.

## 4.15 Dokumentace překladače

V této podkapitole si jen ve stručnosti shrneme, jak byl řešen a hlavně vyřešen nefunkční požadavek **NP06**.

Veškerá dokumentace k překladači je součástí média přiloženého k této práci, ale také oficiálního GIT repozitáře projektu. Tento repozitář je umístěn v rámci školního serveru GitLab, kde je možné jej nalézt pod hlavním projektem předmětu Databázové systémy. Pro podrobný popis struktury repozitáře odkážu zaujatého čtenáře do přílohy C. Samotná dokumentace je pak umístěna v repozitářovém adresáři `doc`.

V podadresáři `api` lze nalézt zdrojový soubor dokumentace rozhraní ve formátu Markdown, který byl stažen z Apiary, viz výše.

V podadresáři `state_machine_diagram` se pak nachází doprovodná dokumentace k lexikální analýze. V tomto podadresáři se nachází dva soubory, jedním z nich je obrázek ve formátu `.jpg`, na němž je zobrazen stavový automat lexikálního analyzátoru. Tento automat přívětivě vizualizuje dění během lexikální analýzy, které ze zdrojového kódu nemusí být na první pohled zřejmé. Dále je zde umístěn i zdrojový soubor ve formátu `.xml`, který je možné nahrát do webového nástroje https://www.draw.io/ v němž byl přechodový automat modelován.

Posledním podadresářem adresáře `doc` je adresář `source_code`. Celý zdrojový kód byl během vývoje řádně dokumentován proto, aby z něj bylo možné vygenerovat dokumentaci pomocí nástroje Doxygen. V tomto podadresáři je umístěn zdrojový soubor Doxyfile v němž je umístěna aktuální konfigurace nástroje Doxygen. Do podadresáře `source_code/output` se pak generuje veškerá dokumentace ze zdrojového kódu ve formátu `.html`. Pro zobrazení domovské stránky této dokumentace je potřeba otevřít soubor `index.html`.

## 4.16 Nasazení překladače

Přesný postup nasazení překladače do produkčního prostředí, tj. do Portálu, je uveden v souboru `README.md`, který je součástí nejen repozitáře GIT, ale i adresáře s implementací překladače na přiloženém médiu. Uvedený soubor obsahuje souhrn příkazů, krok za krokem, jak překladač úspěšně nasadit do produkčního prostředí, například do Portálu. V této podkapitole se tedy pouze omezím na prerekvizity, které musí být splněny proto, aby bylo možné překladač nasadit hladce na produkční prostředí. Na závěr této podkapitoly si shrneme aktuální stav nasazení a integrace překladače do Portálu.

Základním předpokladem pro úspěšné nasazení překladače do produkčního prostředí je administrátorský přístup. Bez administrátorského přístupu není možné překladač úspěšně nasadit! Sekvence kroků popsaných v souboru `README.md` totiž vyžaduje práva uživatele `root`, nebo uživatele s oprávněním ke spouštění příkazu `sudo`. V souladu s aktuální konfigurací Portálu se předpokládá přístup k překladači pomocí webového serveru Nginx a aplikačního serveru Passenger. Pro snadné a úspěšné nasazení překladače se také předpokládá, že na daném cílovém prostředí je instalován jazyk Python ve verzi 3.5.3 nebo vyšší, je zde umožněno vytváření virtuálních prostředí pomocí nástroje `virtualenv` a v ideálním případě i možnost práce s verzovacím systémem GIT. Jako produkční operační systém je pak předpokládán operační systém Debian, případně jakákoliv distribuce založená na tomto operačním systému.

Tyto předpoklady Portál bez výjimky splňuje, proto nasazení nového překladače není nikterak složité. Dokonce bylo ve srovnání s původním překladačem mírně zjednodušeno. Ubyly totiž některé závislosti, které pro nasa-

zení a běh nového překladače již nejsou nadále potřeba. V tuto chvíli je na Portálu, tj. na serveru dbs.fit.cvut.cz nasazena poslední, nejaktuálnější, verze a to verze 2.2.0. Aby došlo k jednoznačnému odlišení původní a nové implementace překladače, byla první verze nového překladače označena jako verze 2.0.0, zatímco původní překladač je na Portále nasazen ve verzi 1.0.1.

Z pohledu překladače je kompletní implementace nasazena do Portálu a tím je tak splněn i nefunkční požadavek **NP08**. Bohužel z důvodu velké časové vytíženosti klíčových členů vývojářského týmu Portálu zatím nedošlo ze strany webové aplikace ke kompletní integraci. Z tohoto důvodu není zatím možné nový překladač využít v rámci výuky. Pro úplnou integraci je potřeba upravit webovou aplikaci Portálu tak, aby reflektovala veškeré změny v rozhraní, které nová implementace přinesla. Tyto změny zahrnují změnu formátu odpovědí, změnu metody požadavku z `POST` na `PUT` a přidání další položky do odpovědi, ve které je uložen prováděcí strom RA dotazu.

Obecně se předpokládá, že k úplné integraci nového překladače dojde až po obhajobě této diplomové práce, tj. během léta 2019. Studenti předmětu Databázové systémy (BI-DBS) by měli tedy začít využívat nový překladač v rámci výuky až během zimního, případně letního, semestru následujícího akademického roku (2019/2020).

Kapitola **5**

# Další funkcionality

Nyní se dostáváme k funkcionalitám, které byly do překladače přidány nad rámec zadání. V závěru implementace zbyl ještě prostor pro dodělání dalších funkcí, které práci s překladačem mohou velmi zpříjemnit a usnadnit. Zajisté by se daly vymyslet další užitečné funkcionality, o kterých se zmiňuji v kapitole 6. V podkapitolách níže se zmiňuji o veškerých vylepšeních (rozšířeních), které se podařilo ještě do překladače doimplementovat a nasadit společně s překladačem do Portálu.

## 5.1 Rozšíření chybových hlášení

Prvním vylepšením je rozšíření chybových hlášení v případě, je-li chyba objevena během lexikální analýzy. Překladač nyní ve své chybové zprávě uživateli dokáže poradit, které výrazy či operace, v daný moment očekával, ale nenašel. Tato serializovaná zpráva do formátu JSON je pak obohacena i o samostatné informace o výskytu chyby. Zpráva tak obsahuje položky čísla řádku a sloupce, na kterém se chyba vyskytuje. Tyto přidané operace by webová aplikace nasazená na Portálu mohla využít například pro označení (podtržení) chyby v editoru, ve kterém student píše daný RA dotaz. Mějme následující RA dotaz:

```
ALBUMS[; album_id name]
```
(D6)

Tento dotaz obsahuje navíc nepovolený znak `';'` a mezi názvy sloupců v projekci chybí čárka sloužící jako oddělovač. Pokud by byl takový dotaz předán překladači, pak by uživateli byly v rámci odpovědi zaslány chybová hlášení viz ukázka výstupu V6.

Obsah odpovědi čítá za prvé informaci o chybě samotné – znak nebo výraz, který na daném místě nebyl očekáván – i informaci o znacích / výrazech, které v daném okamžiku naopak očekávány byly. Lexikální analyzátor se pouze snaží

napovědět, které výrazy v daném momentě je možné použít pro eliminaci lexikální chyby. V silách lexikálního analyzátoru už ale není schopnost určit, zda s použitím nápovědy bude dotaz za každých okolností syntakticky správný. Tato kompetence spadá pod syntaktický analyzátor. Jinými slovy mohou nastat situace, kdy lexikální analyzátor napoví výraz, který je možné použít, jeho použití v následném pokusu o překlad povede ke korektnímu převodu na token, avšak posloupnost těchto tokenů i přes opravu nemusí být syntakticky korektní.

```
{
    "line": 1,
    "column": 8,
    "message": "Illegal character ';' at line: 1, column: 8,
                expected: 'COLUMN_NAME', '¬', 'NOT', 'not', '('"
},
{
    "line": 1,
    "column": 19,
    "message": "Illegal expression 'name' at line: 1, column: 19,
                expected: '->', ',', ']'"
}
```

Výstup z příkazové řádky V6: Ukázka rozšířených chybových hlášení překladače

## 5.2 Optimalizace SQL výstupu

Další ze změn, které byly implementovány, je optimalizace SQL výstupu. Tato optimalizace se týká optimalizace gramatiky relační algebry a silně souvisí se strukturou AST. V původní implementaci znamenalo použití složených závorek vytvoření speciálního typu uzlu, jehož překlad vedl v naprosté většině na vytvoření poddotazu v rámci SQL. Výstup překladače pak tedy obsahoval poddotazy, které v daný moment byly nadbytečné. Pokud student napsal dotaz:

$$\{\{\{\texttt{ALBUMS}\}\}\} \tag{D7}$$

pak tento dotaz původní překladač vyhodnotil způsobem, který je uveden na ukázce překladu P11.

V nové implementaci toto chování bylo odstraněno a zodpovědnost za vytváření poddotazů náleží rodičovskému uzlu AST, respektive instanci třídy

`TransformationVisitor`, která díky průchodu POST-ORDER rodičovský uzel zpracovává až po navštívení všech potomků.

```
SELECT DISTINCT *
FROM
  (SELECT DISTINCT *
   FROM
     (SELECT DISTINCT *
      FROM ALBUMS);
```

Překlad P11: Výsledek překladu dotazu D7 pomocí původního překladače

Pokud bychom dotaz D7 překládali pomocí nového překladače, pak bychom dostali následující opdověď:

```
SELECT DISTINCT *
FROM ALBUMS;
```

Překlad P12: Výsledek překladu dotazu D7 pomocí nového překladače

Zajímavější případ nastává v situacích, kdy jsou použity binární operace. Ke změně priority stále slouží složené závorky, nicméně jejich použití se ne vždy ve výsledném výstupu musí projevit. Překladač, respektive instance třídy `TransformationVisitor`, si tvorbu poddotazů řídí sama. Pokud budeme mít dotaz:

$$\{\texttt{ALBUMS * ARTISTS}\} \times \texttt{GENRES} \tag{D8}$$

pak výsledek překladu je naprosto shodný s překladem dotazu, který by složené závorky neobsahoval. V tomto případě totiž složené závorky nemění prioritu při vyhodnocování, neboť operace se vyhodnocují zleva doprava, všechny binární operace mají stejnou prioritu a struktura AST v obou případech zůstává stejná.

V dotazu SELECT jazyka SQL se stejným způsobem vyhodnocují definovaná spojení tak, jak jdou po sobě. Z tohoto důvodu tvorba poddotazu v klauzuli `FROM` je nadbytečná a překlad P13 dotazu D8 je tak naprosto korektní. V případě přidání projekce nebo selekce za složené závory bychom se tvorbě poddotazu v klauzuli `FROM` již nevyhnuli.

Pokud bychom v dotazu D8 operaci přirozeného spojení nahradili množinovou operací, tj. znak `*` nahradili znakem $\cap$, $\cup$ či $\setminus$, pak by došlo k vy-

```
SELECT DISTINCT *
FROM ALBUMS
NATURAL JOIN ARTISTS
CROSS JOIN GENRES;
```

Překlad P13: Výsledek překladu dotazu D8 pomocí nového překladače

tvoření poddotazu v klauzuli `FROM` vždy, nehledě na složené závorky. Výsledek množinové operace není možné spojit s tabulkou `GENRES` bez toho, aniž by se tento výsledek převedl na poddotaz.

Naprosto jiná situace nastává, pokud bychom v dotazu D8 složenými závorkami místo operace přirozeného spojení obalili operaci kartézského součinu, viz dotaz D9:

$$\texttt{ALBUMS} * \{\texttt{ARTISTS} \times \texttt{GENRES}\} \tag{D9}$$

V tomto dotazu se děje hned několik zajímavých věcí. Za prvé, uživatel použitím složených závorek upravuje prioritu vyhodnocování RA dotazu. V tomto případě chce, aby se operace kartézského součinu vyhodnotila dříve, než operace přirozeného spojení. Za druhé, na pozadí dochází ke změně struktury AST. Pokud se vrátíme k dotazu D8, pak kořenovým uzlem odpovídajícího AST je uzel `CrossJoinNode`. U dotazu D9 už tomu tak není, kořenovým uzlem AST se stal uzel `NaturalJoinNode`, jehož pravým synem je právě uzel `CrossJoinNode`.

Transformace takto změněné struktury AST, respektive výsledek jejího následného překladu je zobrazen níže. Překlad P14 přesně kopíruje změnu priority tak, jak ji uživatel specifikoval v dotazu D9.

```
SELECT DISTINCT *
FROM ALBUMS
NATURAL JOIN (
    SELECT DISTINCT *
    FROM ARTISTS
    CROSS JOIN GENRES
) R1;
```

Překlad P14: Výsledek překladu dotazu D9 pomocí nového překladače

Díky tomu, jak jsou definována gramatická pravidla v příloze E a jak se mění struktura AST s použitím složených závorek, není nutné vytvářet

speciální typ uzlu, který by určoval potřebu vytvoření poddotazu. Tato potřeba se totiž řídí samotnou strukturou AST a průběžnými mezivýsledky transformace. Z toho vyplývá, že vícenásobné použití složených závorek v RA dotazu D9 na výsledný překlad nemá žádný vliv.

## 5.3 Podpora dalších operací relační algebry

Pro maximální usnadnění přechodu na jinou syntaxi (více viz 6.3) byla do překladače přidána podpora vnějších spojení. Tyto operace jsou nad rámec výuky předmětu Databázové systémy (BI-DBS}, nicméně nástroj RelaX tyto operace implementuje a ve své knize Databázové systémy tyto operace definuje i Ing. Michal Valenta, Ph.D. [2].

Značení, jakým jsou tyto operace definovány ve výše uvedené knize, není snadno zapsatelné z klávesnice. Například značení levého vnějšího přirozeného spojení je definováno pomocí znaků $*_L$. Takové značení je sice možné v systému LaTeX, nicméně v klasickém editoru tento zápis není většinou možný. Nejvíce se tomuto zápisu podobá následující dvojice znaků `*L`. V tomto případě však písmeno `L` může být počátečním písmenem názvu relace v případě, že student chtěl použít operaci přirozeného spojení (ne vnějšího).

Jedním způsobem, jak tento problém vyřešit, by bylo v rámci lexikální analýzy zjišťovat, zda po `*L` následuje bílý znak. Pokud ano, dala by se tato posloupnost znaků v tomto případě považovat za značení operace vnějšího levého spojení. Každopádně tato mezera by musela být přítomna vždy, aby došlo ke korektnímu rozpoznání levého přirozeného vnějšího spojení.

Z tohoto důvodu jsem definoval značení jiné, avšak velmi podobné. Mezi znaky `*` a `L` musí být vepsán znak `^`, aby byl jednoznačně indikován úmysl použití operace levého vnějšího přirozeného spojení. Tento symbol je z amerického rozložení klávesnice snadno zapsatelný a v matematickém zápisu se tím standardně značí horní index (mocnina). Tento způsob značení je pak snadno aplikovatelný i na operaci obecného spojení, čímž získáme operace obecného vnějšího spojení, jak si ukážeme dále.

### 5.3.1 Přirozená vnější spojení

Operaci levého, pravého, respektive plného vnějšího přirozeného spojení tedy značíme `*^L`, `*^R`, respektive `*^F`. Definice jednotlivých typů vnějších přirozených spojení jsou uvedeny v totožném pořadí níže:

$$\texttt{R *\^{}L S :=}_{def} \{\texttt{R * S}\} \cup \{\{\texttt{R !<* S}\} \times \texttt{(NULL, ..., NULL)}\}$$
$$\texttt{R *\^{}R S :=}_{def} \{\texttt{R * S}\} \cup \{\texttt{(NULL, ..., NULL)} \times \{\texttt{S !<* R}\}\}$$
$$\texttt{R *\^{}F S :=}_{def} \{\texttt{R *\^{}L S}\} \cup \{\texttt{R *\^{}R S}\}$$

### 5.3.2 Obecná vnější spojení

Značení operace obecného spojení v RA vyučované na FIT ČVUT je definováno pomocí hranatých závorek. Výše uvedené značení směrů vnějších spojení je pak snadno aplikovatelné i na toto obecné spojení, čímž získáme levé, pravé, respektive plné vnější obecné spojení. Definice těchto operací jsou obdobné a jsou uvedeny dále, znak $\Theta$ zastupuje podmínku spojení:

$$\texttt{R [}\Theta\texttt{]\^{}L S} :=_{def} \{\texttt{R [}\Theta\texttt{] S}\} \cup \{\{\texttt{R !<}\Theta\texttt{] S}\} \times \texttt{(NULL, ..., NULL)}\}$$
$$\texttt{R [}\Theta\texttt{]\^{}R S} :=_{def} \{\texttt{R [}\Theta\texttt{] S}\} \cup \{\texttt{(NULL, ..., NULL)} \times \{\texttt{S !<}\Theta\texttt{] R}\}\}$$
$$\texttt{R [}\Theta\texttt{]\^{}F S} :=_{def} \{\texttt{R [}\Theta\texttt{]\^{}L S}\} \cup \{\texttt{R [}\Theta\texttt{]\^{}R S}\}$$

### 5.3.3 Příklad

Pro ukázku překladu vnějších spojení nejprve definujme dotaz:

```
ALBUMS *^L ARTISTS *^R ALBUMS_TRACKS
```
(D10)

Překladač tento dotaz přeloží následovně:

```sql
SELECT DISTINCT *
FROM ALBUMS
NATURAL LEFT OUTER JOIN ARTISTS
NATURAL RIGHT OUTER JOIN ALBUMS_TRACKS;
```

Překlad P15: Výsledek překladu dotazu D10 pomocí nového překladače

## 5.4 Deklarace relací

Asi nejzajímavějším rozšířením celého překladače je umožnění deklarací vlastních relací. K deklaraci vlastní relace slouží operátor `:=`, jemuž musí předcházet název nové relace a po němž musí následovat klasický dotaz RA. S deklaracemi však souvisí některá pravidla, která svým způsobem vychází z chování SQL dotazu zahrnující příkaz WITH. Za prvé, rekurze v deklaracích není možná. Za druhé, aby bylo možné vlastní relaci použít v dalších deklaracích je potřeba ji nejprve deklarovat. Konečně, po bloku deklarací musí následovat klasický dotaz RA, který produkuje finální výstup a který může obsahovat jednu, či více deklarovaných relací. Ukážeme si na příkladu. Náš komplexní dotaz D4 bychom pomocí deklarací vlastních relací mohli zapsat například takto:

```
A := ARTISTS
B := ALBUMS !<* ALBUMS_TRACKS
C := A [A.artist_id = B.artist_id] B
C[artist_id, artist_name, C.name]
```
(D11)

Dotaz D4 byl v rámci dotazu D11 rozfázován na několik kroků a jednotlivé výsledky byly uloženy do deklarovaných relací `A`, `B`, respektive `C`. Nejdůležitější je poslední řádka dotazu D11, která produkuje výstupní hodnoty celého RA dotazu.

```
WITH A AS (
    SELECT DISTINCT *
    FROM ARTISTS
),
B AS (
    SELECT DISTINCT *
    FROM ALBUMS
    MINUS
    SELECT DISTINCT album_id,
                    name,
                    note,
                    price,
                    year,
                    artist_id,
                    genre_id
    FROM ALBUMS
    NATURAL JOIN ALBUMS_TRACKS
),
C AS (
    SELECT DISTINCT A.artist_id,
                    A.artist_name,
                    A.description,
                    B.album_id,
                    B.name,
                    B.note,
                    B.price,
                    B.year,
                    B.artist_id AS artist_id_1,
                    B.genre_id
    FROM A
    JOIN B ON A.artist_id = B.artist_id
)
SELECT DISTINCT artist_id,
                artist_name,
                C.name
FROM C;
```

Překlad P16: Výsledek překladu dotazu D11 pomocí nového překladače

Z pohledu jazyka SQL se závěrečná řádka dotazu D11 přeloží na dotaz SELECT, který vždy musí následovat po deklaracích v klauzuli WITH, viz překlad P16. Výsledek po spuštění tohoto dotazu je shodný s výsledkem uvedeným v tabulce 4.1.

Obdobně lze pomocí deklarací vlastních relací vyjádřit i operaci relačního dělení, která v žádném RDBMS není přímo implementována. Postup, jak takové relace deklarovat a jaké operace použít pro dosažení relačního dělení, je uvedený v přednáškách předmětu Databázové systémy (BI-DBS) [3].

## 5.5 Vyhodnocovací stromy dotazu v relační algebře

Posledním větším rozšířením překladače oproti základnímu zadání je tvorba vyhodnocovacích stromů RA dotazu. I v tomto případě se ukázalo, že použití návrhového vzoru Visitor má své opodstatnění. Do implementace překladače přibyla další třída `EvaluationVisitor`, která dědí od třídy `ASTVisitor`, a jejímž úkolem je na základě definovaného AST sestavit strom vyhodnocování RA dotazu. Tento strom se pak posílá v rámci odpovědi překladače ve formátu JSON. Portál by pak mohl zobrazovat tento vyhodnocovací strom obdobně tak, jak je tomu v případě portálu RelaX, viz obrázek 2.1.

Jednotlivé uzly tohoto stromu pak mají vždy nějakou hodnotu, například název relace či celé znění operace, která byla použita. Součástí uzlu je pak i seznam sloupců, které jsou z daného uzlu výstupní (výsledné po aplikaci dané operace – v případě samotné relace je to pak seznam veškerých sloupců dané relace). Každý vnitřní uzel stromu má alespoň jeden synovský uzel, pro nějž platí ty samé podmínky uvedené výše. Příklad uzlu serializovaného do formátu JSON je uveden ve výstupu V7.

```
{
    "value": "[artist_id, artist_name, C.name]",
    "columns": [
        ["C.artist_id", "artist_id"],
        ["C.artist_name", "artist_name"],
        ["C.name", "name"]
    ],
    "children": [
        ...
    ]
}
```

Výstup z příkazové řádky V7: Ukázka typického uzlu vyhodnocovacího stromu RA dotazu

Zobrazený serializovaný uzel odpovídá kořenovému uzlu dotazu D11, jehož hodnotou je právě závěrečná projekce, a jehož synovským uzlem je uzel reprezentující deklarovanou relaci `C`. Seznam sloupců tak, jak je výše uvedený, obsahuje tři sloupce. V tomto případě je každý sloupec reprezentován dvojicí řetězců, které definují veškeré možné formy zápisu daného názvu sloupce, a to s nebo bez použití tečkové notace.

V případě deklarací relací bývá těchto prováděcích stromů více, v případě dotazu D11 by pak tyto stromy byly čtyři. První tři stromy by ve svém kořenovém uzlu uloženy vždy název deklarované relace, čtvrtý strom by pak měl kořenový uzel shodný s uzlem zobrazeným ve výstupu V7. Zůstává otázkou, zda z pohledu uživatelského rozhraní je zobrazování většího počtu prováděcích stromů stále uživatelsky přívětivé či nikoliv. V tomto ohledu se nový překladač odlišuje od chování webového nástroje RelaX, který i v případě deklarací generuje pouze jeden prováděcí strom, jehož podstromy jsou kopírovány s každým použitím deklarované relace.

Případná pozdější úprava na straně překladače samozřejmě možná je. Na druhou stranu i Portál samotný by měl být schopen, na základě veškerých stromů jemu předaných, sestavit strom jediný obdobně tak, jak to dělá webový nástroj RelaX.

Kapitola 6

# Možnosti budoucího rozvoje

V této kapitole si představíme tři možná rozšíření, kterými by se dal překladač v budoucnu ještě obohatit. Možností pro rozšíření by se dalo vymyslet hned několik, výčet uvedený níže představuje pouze drobné, avšak zajímavé nápady, plus se zabývá již několikrát zmiňovanou podporou jiné syntaxe relační algebry.

## 6.1 Psaní komentářů do dotazu v relační algebře

Prvním, a asi i nejtriviálnějším, rozšířením by mohlo být zavedení možnosti psaní komentářů v dotazu RA. Obdobně, jako to například implementuje nástroj RelaX, by mohlo být povoleno psaní blokových i jednořádkových komentářů. Jejich značení by mohlo vyjít například z jazyka SQL.

Na překladači, respektive lexikálním analyzátoru by pak zbyla úloha tyto komentáře detekovat a jejich obsah ignorovat. Do definic společných regulárních výrazů, které se používají v třídě `Lexer`, by pak přibyly definice regulárního výrazu pro blokový a jednořádkový komentář. S určitou obměnou by bylo možné využít regulární výrazy pro komentáře, které jsou uvedeny v dokumentaci samotné knihovně PLY [14]. Na základě těchto regulárních výrazů by pak bylo možné do abstraktní třídy `Lexer` přidat následující metodu:

```
##
# @brief Comments recognition rule in @c any state
#
@lex.TOKEN(regex_common.COMMENT)
def t_ANY_comment(self, t):
    pass
```

Ukázka zdrojového kódu U6: Návrh metody pro rozpoznávání komentářů

Z pohledu překladače by tedy měla stačit pouze tato drobná úprava lexikálního analyzátoru. Z pohledu Portálu by bylo uživatelsky přívětivé použité komentáře obarvovat, aby jejich použití bylo na první pohled zřejmé.

Vzhledem k přidání podpory deklarací vlastních relací se podpora komentářů přímo nabízí. Pro studenty i učitele by mohlo být přínosné přidat k deklarovaným relacím kratičký popisek.

## 6.2 Rozšíření selekce o porovnání IS (NOT) NULL

Dalším, avšak již náročnějším, rozšířením by bylo zavedení podpory porovnání na (ne)vyplněnost hodnot podmínce selekce. Toto rozšíření dává smysl poté, co byla do relační algebry zavedena podpora vnějších spojení. Naproti tomu, výše uvedený konstrukt je konstruktem jazyka SQL, nikoliv samotné relační algebry – v knize Databázové systémy je sice hodnota NULL zmiňována z pohledu tabulek, ale z pohledu relační algebry není definována. Zůstává tedy otázkou, zda toto rozšíření v budoucnu do relační algebry zahrnovat, či nikoliv. V případě kladné odpovědi je dopad na implementaci překladače poměrně větší, než tomu bylo v případě přidání podpory komentářů.

Z pohledu lexikálního analyzátoru by bylo potřeba definovat další stavy, do nichž lexikální analyzátor může přejít a které by detekovaly přesnou sekvenci výrazů `IS NULL` a `IS NOT NULL` v rámci selekce. Výsledkem by pak byly pravděpodobně nové tři typy tokenů, obzvlášť větší pozornost by se musela věnovat pozornost tokenu pro výraz `NOT`, neboť jeden takový typ tokenu už je v lexikálním analyzátoru implementován a po jeho detekci se očekává použití kulatých závorek, které v tomto případě nejsou možné!

Z pohledu syntaktického analyzátoru by bylo potřeba rozšířit selekci o další gramatická pravidla, která by při jejich rozpoznání generovala patřičné nový nebo nové typy uzlů abstraktního syntaktického strom.

V závislosti na počtu vytvořených pravidel by bylo třeba přidat patřičné `visit` metody do tříd `TransformationVisitor` a `EvaluationVisitor`. Samotná transformace těchto nově přidaných uzlů by pak pouze generovala instance třídy `Element`, jejíž překlad je již implementovaný. Tyto instance by se pak staly pouze součástí výsledného seznamu objektů, ze kterých se klauzule `WHERE` objektu `Query` skládá. Z tohoto důvodu by nemělo být třeba jakkoliv editovat třídu `SQLVisitor` a její podtřídy.

Samozřejmostí je pak na závěr přidání odpovídajících jednotkových testů pro ověření korektnosti překladu těchto nových konstruktů.

## 6.3 Rozpoznávání další synaxe relační algebry

Asi nejzajímavějším, ale implementačně nejnáročnějším, by bylo přidání podpory pro rozpoznávání jiné (matematické) syntaxe, která je definována na-

příklad v knize Database Systems: The Complete Book [32] a z níž čerpá i samotný nástroj RelaX [9].

Možnosti rozšíření o rozpoznávání jiné syntaxe byl vesměs uzpůsoben i samotný návrh nového překladače. Věřím, že přidání této funkcionality do překladače by za splnění určitých předpokladů nemuselo být extrémně náročné, nicméně je to dosti velký objem práce, který je potřeba udělat. Pro následující analýzu budeme předpokládat zavedení obdobné syntaxe, kterou používá nástroj RelaX.

Prvně je potřeba říci, že tato syntaxe je v drtivé většině odlišná od syntaxe vyučované na FIT ČVUT. Pokud bychom dotaz D4 chtěli zapsat pomocí nové syntaxe, pak bychom jej pravděpodobně zapsali následovně:

$$\pi\ \texttt{artist_id, artist_name, ALBUMS.name (ARTISTS}\ \bowtie \\ \texttt{ARTISTS.artist_id = ALBUMS.artist_id (ALBUMS}\ \rhd \\ \texttt{ALBUMS_TRACKS))} \tag{D12}$$

Způsob zápisu je odlišný, ale i přes to tam lze jistou podobnost vidět. Operace levého přirozeného anti-joinu je pouze definována jediným znakem $\rhd$, místo tří. Operace obecného spojení pak obsahuje „otevírací“ značku $\bowtie$ a definici podmínky spojení, nikoliv už zavírací značku, jako je tomu v případě dotazu D4. Konečně, projekce se značí též pouze pomocí otevírací značky $\pi$ a nepíše se za dotaz, nýbrž před. Pro určení priority vyhodnocování operací se v tomto případě používají kulaté závorky na rozdíl od složených.

### 6.3.1 Porovnání operací

Ve výše uvedeném případu lze jistou podobnost vypozorovat, ale jak je tomu z globálního hlediska? Pro selekci platí stejná pravidla, jako pro projekci. Způsob zápisu množinových operací je podobný, s drobnou výjimkou značení u množinového rozdílu. Veškerá přirozená spojení se značí jako obecná spojení s tím rozdílem, že po značce spojení nenásleduje podmínka spojení (platí i pro vnější).

Naopak nástroj RelaX implementuje některé operace, které v rámci nového překladače nejsou podporovány a ani nejsou vyučovány v rámci relační algebry na FIT ČVUT. Mezi tyto operace patří agregace, řazení a přejmenování relace. Přejmenování názvu sloupce je pak možné pomocí levé, či pravé šipky.

Nový překladač na rozdíl od nástroje RelaX podporuje například operace obecných polospojení a anti joinů. V případě anti joinů nástroj RelaX podporuje pouze levý přirozený anti join, zatímco nový překladač podporuje i pravý přirozený anti join, včetně obecných variant levého a pravého anti joinu.

Pokud bychom tedy předem vyřadili operace agregace a řazení, pak by měla být množina podporovaných operací překladačem dokonce větší, než je množina operací, které používá nástroj RelaX (s ohledem na tvrzení o polospojeních a anti joinech, viz výše). Z tohoto důvodu by nemuselo být nutné

přidávat další typy uzlů AST. Na zvážení pak zůstává, zda-li implementovat i operaci přejmenování relací, která je vhodná zejména v případě self joinů.

### 6.3.2 Implementace

Před samotnou implementací je potřeba nové syntaxi přiřadit nějaký souhrnný název, neboť tento název se pak vyskytuje, jak v samotném názvu některých tříd, tak i v konfiguračním souboru. Pro naše teoretické účely ji pojmenujeme anglickým slovem `general`. Přidání podpory další syntaxe s sebou přináší minimálně vytvoření nového lexikálního a syntaktického analyzátoru.

#### 6.3.2.1 Lexikální analyzátor

Z pohledu lexikálního analyzátoru je nejprve potřeba podrobně analyzovat značení následujících operací:

- projekce (včetně přejmenování sloupců),
- selekce (strukturu podmínek),
- přirozených a obecných spojení (včetně vnějších),
- přirozených polospojení,
- pevého přirozeného anti joinu,
- kartézského součinu,
- množinových operací (průniku, sjednocení, rozdílu),
- relačního dělení.

V případě deklarace relací může zůstat značení stejné. Na základě této analýzy je pak potřeba navrhnout podobu přechodového automatu (stavy a přechody), obdobně tak, jak je tomu v případě současného lexikálního analyzátoru. Inspirovat se lze u současného stavového automatu, jehož grafické znázornění je součástí dokumentace. Vzhledem k faktu, že značení používané nástrojem RelaX, na rozdíl od značení používaného na FIT ČVUT, obsahuje daleko více symbolů, které nelze snadno zapsat z klávesnice, by bylo záhodné, aby bylo definováno i alternativní značení. Například projekci vyjadřovat pomocí symbolu $\pi$, ale i pomocí alternativního zápisu například `\pi`.

S jasně definovanou množinou stavů, operací a jejich značením je možné definovat regulární výrazy, které se použijí pro vytváření tokenů. Patřičné definice by bylo vhodné uložit do souborů `general.py` a ty umístit do patřičných podadresářů adresáře `rat/definition/lexer`, kde již nějaké definice přítomny jsou.

Dalším krokem je implementace samotného lexikálního analyzátoru podle knihovny PLY. Pro značení, které není společné je potřeba definovat třídu, ideálně `GeneralLexer`, která rozšíří abstraktní třídu `Lexer`. Třída `Lexer` definuje jen minimum operací, jejichž značení je totožné (například průniku a sjednocení). Je možné, že v případě přidání alternativního zápisu bude potřeba předefinovat i chování těchto funkcí ve třídě `GeneralLexer`, pokud bychom nechtěli povolovat alternativní zápis i pro syntaxi používanou na FIT ČVUT.

Nový lexikální analyzátor pak na základě definovaných operací bude generovat zcela odlišné posloupnosti tokenů, které se musí předat novému syntaktickému analyzátoru.

#### 6.3.2.2 Syntaktická analýza

Pro účely syntaktické analýzy je potřeba definovat kompletní novou a hlavně jednoznačnou gramatiku relační algebry, která bude schopna rozpoznat nové posloupnosti tokenů. Gramatická pravidla musí být definována s ohledem na zcela odlišné vyhodnocování priority operací. V rámci nástroje RelaX mají například množinové operace nižší prioritu před operacemi spojení [33]. Pro kompletní definici priorit operací vizte tabulku 6.1. Naštěstí lze tyto priority určit úrovní jednotlivých gramatických pravidel. Příkladem může být současná gramatika relační algebry, která je uvedena v sekci E. Tato gramatika ctí prioritu vyhodnocování aritmetického součtu, která je nižší oproti operaci aritmetického součinu. Obě operace je možné použít v rámci selekce. Podobně tedy lze definovat potřebnou prioritu i mezi jinými operacemi, například již zmiňovanými množinovými operacemi a spojeními.

| | |
|---|---|
| 0 | projekce, selekce, přejmenování – nejvyšší |
| 1 | kartézský součin, přirozené spojení, obecné spojení, vnější přirozená spojení, vnější obecná spojení, přirozená polospojení, levý přirozený anti join, relační dělení |
| 2 | množinový průnik |
| 3 | množinové sjednocení a rozdíl |

Tabulka 6.1: Priority vyhodnocování operací relační algebry převzaté z dokumentace nástroje RelaX

Stejně jako v případě lexikálního analyzátoru, tak i v tomto případě bude potřeba vytvořit novou třídu, tentokrát například `GeneralParser`, která bude podtřídou třídy `Parser`. V této třídě je potřeba implementovat veškeré nutné metody pro rozpoznávání nových gramatických pravidel. Inspirovat se lze v třídě `FITParser`, případně v dokumentaci knihovny PLY. Třída `General-Parser` pak musí mít referenci na tovární třídu `ASTFactory`, která má na starosti vytváření veškerých uzlů AST. Je ale nutné brát zřetel na strukturu těchto uzlů, která musí být i nadále dodržena. Například uzel `ProjectionNode`

má ve svém pravém podstromu seznam sloupců, které mají být projektovány. Nicméně z dotazu D12 víme, že sloupce, které mají být projektovány, *předchází* výrazu, na nějž je operace projekce aplikována. Z tohoto důvodu by se seznam těchto sloupců měl nacházet v levém podstromu, zatímco relace, na níž je projekce aplikována v podstromu pravém. Abychom ale nemuseli jakkoliv modifikovat průchod AST instancí třídy `TransformationVisitor`, je potřeba prohodit pořadí potomků, které se předávají při žádosti o vytvoření daného uzlu (tím se projektované sloupce dostanou do pravého podstromu).

Pokud by došlo ke splnění veškerých předpokladů uvedených v této podkapitole, pak by nemělo být nutné jakkoliv zasahovat do implementace třídy `TransformationVisitor`, a tudíž ani následná tvorba překladu by neměla být ovlivněna. Množina potřebných uzlů by totiž díky vyřazení operací agregace a řazení měla zůstat stejná.

Na závěr je potřeba přidat reference na nově vytvořené třídy `GeneralLexer`, respektive `GeneralParser` do tovární třídy `ComponentFactory`, která řídí vytváření jednotlivých komponent. V konfiguračním souboru `default.ini` je pak potřeba změnit používanou syntaxi z `fit` na `general`, a pokud byla třída `ComponentFactory` správně upravena, měl by překladač automaticky začít používat novou syntaxi relační algebry.

Výše uvedené odstavce by měly poukazovat na připravenost překladače pro rozšíření, proto lze poslední, doposud nesplněný, nefunkční požadavek **NP05** označit za splněný.

# Závěr

Během vývoje původního překladače byly z důvodu přílišné náročnosti ze zadání vyjmuty požadavky pro překlad do dalších RDBMS. Jmenovitě RDBMS PostgreSQL a MariaDB (MySQL). Až při reálném používání překladače v rámci výuky předmětu Databázové systémy (BI-DBS) na FIT ČVUT se ukázaly problémy, které měly negativní dopad na jeho používání (problém tečkové notace, automatického přejmenování, priority vyhodnocování operace selekce a projekce).

Cílem této práce bylo buď upravit stávající řešení překladače, nebo navrhnout překladač nový. V obou případech bylo potřeba přidat podporu překladů do dalších dialektů jazyka SQL a řešit veškeré výše zmíněné problémy. Na základě nedostatků zjištěných během analýzy jsem se rozhodl pro implementaci překladače nového.

Nový překladač byl navržen a implementován s ohledem na rozšiřitelnost, jíž se podrobněji věnuji v sekci 6.3. Samotná implementace odráží použití návrhových vzorů diskutovaných v sekci 3.2. Proces samotného překladu byl oproti původní implementaci rozšířen o krok transformace AST do nové, vhodnější, objektové struktury. Během této transformace jsou řešeny a vyřešeny veškeré výše vzpomínané problémy. Řešení problému tečkové notace bylo vysvětlováno v kapitole 4.9.1.1, rozšíření automatického přejmenování sloupců i pro operace obecného spojení v kapitole 4.9.1.2, a konečně řešení nepřesnosti při vyhodnocování operace projekce a selekce bylo shrnuto v kapitole 4.9.2.1. S ohledem na použití návrhového vzoru Visitor nebylo přidání podpory překladů do dalších RDBMS nijak obzvlášť náročné. Nový překladač podporuje překlady relační algebry do SQL spustitelného v RDBMS Oracle, PostgreSQL a bude kompletně podporovat překlady pro RDBMS MariaDB po oficiálním vydání verze 10.4, více viz 4.10.1.

Po zhotovení základní kostry překladače byly dále implementovány další funkcionality, které již byly nad rámec samotného zadání této práce. Překladač nově umožňuje deklarace vlastních relací, včetně jejich následného dotazování v rámci toho samého dotazu relační algebry. K tomuto dotazu překladač navíc

sestavuje i prováděcí strom, pro jehož opětovné sestavení v rámci externích aplikací poskytuje potřebná data ve formátu JSON.

Po vzoru původního překladače byl i nový překladač podroben testování a zpřístupněn skrze API. Veškerá dokumentace, která je nahraná na přiložené médium a je také součástí oficiálního repozitáře projektu, zahrnuje dokumentaci vygenerovanou ze zdrojového kódu, dokumentaci API i grafické znázornění stavového automatu lexikálního analyzátoru.

Na závěr byl překladač integrován do Portálu a je lokálně přístupný pro další aplikace skrze webový server Nginx a aplikační server Phusion Passenger. Po integraci překladače do Portálu jsem provedl měření výkonu, abych potvrdil splnění nefunkčního požadavku, který kladl nároky na jeho odezvu. Tyto výsledky jsem pak porovnal s výsledky z testování na lokálním prostředí, více viz 4.13. V obou případech se ukázalo, že veškeré odezvy překladače jsou v řádu setin sekundy, což se s velkou rezervou vejde do požadované doby odezvy.

Byť se z počátku mohlo zdát, že práce na překladači relační algebry do SQL již nemusí být mnoho, opak se stal pravdou. Vývoj nového překladače mě i přes svou časovou náročnost velice bavil a naplňoval.

Věřím, že se mi touto prací podařilo vytvořit překladač, který, až dojde k jeho plné integraci ze strany Portálu, posune práci s relační algebrou na novou, ještě lepší úroveň. Přál bych si, aby nový překladač byl kladně hodnocen učiteli i studenty předmětu Databázové systémy (BI-DBS), a aby sloužil plně k jejich spokojenosti.

# Literatura

[1] KUBIŠ, M.: *Překladač z relační algebry do SQL.* Bakalářská práce [online], České vysoké učení technické v Praze, Fakulta informačních technologií, Katedra softwarového inženýrství, 2016, [cit. 2019-03-29]. Dostupné z: https://alfresco.fit.cvut.cz/share/proxy/alfresco/api/node/content/workspace/SpacesStore/88049e6b-362a-47c9-8f05-056d2dc46174

[2] POKORNÝ, J.; VALENTA, M.: *Databázové systémy.* Praha : České vysoké učení technické v Praze, první vydání, 2013, ISBN 978-80-01-05212-9, 32–42 s.

[3] VALENTA, M.: Relační model, relační algebra [online]. [cit. 2019-04-01]. Dostupné z: https://courses.fit.cvut.cz/BI-DBS/materials/slides/hand-les03-relacni-model-a-ra.pdf

[4] UNIVERSIDAD NACIONAL DE COSTA RICA - ESCUELA DE INFORMATICA: Relational Algebra Translator [online]. [cit. 2019-04-03]. Dostupné z: http://www.slinfo.una.ac.cr/rat/rat.html

[5] UNIVERSIDAD NACIONAL DE COSTA RICA - ESCUELA DE INFORMATICA: Programación para todos - Cap. 6- RAT [online]. [cit. 2019-04-03]. Dostupné z: http://www.slinfo.una.ac.cr/rat/documentation/Cap.%206-%20RAT.pdf

[6] ACOSTA, G.: Pireal official GitHub repository [online]. [cit. 2019-04-04]. Dostupné z: https://github.com/centaurialpha/pireal

[7] UNIVERSITÄT INNSBRUCK: Übungstool für Relationale Algebra, Relationen-Tupel-Kalkül und SQL [online]. [cit. 2019-03-30]. Dostupné z: https://dbis-informatik.uibk.ac.at/ubungstool-fur-relationale-algebra-relationen-tupel-kalkul-und-sql

[8] KESSLER, J.: RelaX – relational algebra calculator [online]. [cit. 2019-03-30]. Dostupné z: https://dbis.uibk.ac.at/sites/default/files/2019-03/2019-BTW-Relax.pdf

[9] KESSLER, J.; TSCHUGGNALL, M.; SPRECHT, G.: RelaX: A Webbased Execution and Learning Tool forRelational Algebra [online]. [cit. 2019-03-30]. Dostupné z: https://dbis.uibk.ac.at/sites/default/files/2019-03/2019-BTW-Relax.pdf

[10] UNIVERSITÄT INNSBRUCK: Improving the RelaX-tool with Automatic Optimization Functionality [online]. [cit. 2019-03-30]. Dostupné z: https://dbis-informatik.uibk.ac.at/improving-relax-tool-automatic-optimization-functionality

[11] COOK, S.: SQL Query Order of Execution [online]. [cit. 2019-04-01]. Dostupné z: https://www.periscopedata.com/blog/sql-query-order-of-operations

[12] MOLINARO, A.: *SQL Kuchařka programátora*. Computer Press, a.s., první vydání, 2009, ISBN 978-80-251-2617-2, 58 s.

[13] ŠPAČEK, P.: Lecture 02 - Characteristics of good and bad design & Principles of good OO design [online]. [cit. 2019-03-31]. Dostupné z: https://moodle.fit.cvut.cz/pluginfile.php/197/course/section/10597/miadp-2018-lecture02-principles.pdf

[14] BEAZLEY, D.: PLY (Python Lex-Yacc) [online]. [cit. 2019-03-30]. Dostupné z: http://www.dabeaz.com/ply/index.html

[15] JANOUŠEK, J.: Programovací jazyky a překladače - úvod, struktura překladače [online]. [cit. 2019-03-30]. Dostupné z: https://courses.fit.cvut.cz/BI-PJP/media/lectures/01/prednaska1.pdf

[16] ŠPAČEK, P.: Lecture 03-04-05a - Architecture vs. Design patterns & Classification of OO design patterns & Patterns for semestral work [online]. [cit. 2019-04-05]. Dostupné z: https://moodle.fit.cvut.cz/course/format/wiki/mediafile.php?id=77&path=%2flectures%2fmiadp-2017-lecture03-04-05a-mvcgame-patterns.pdf

[17] ZAVORAL, F.: Návrhové vzory [online]. [cit. 2019-04-05]. Dostupné z: http://www.ksi.mff.cuni.cz/lectures/NPRG024/DesignPatterns.ppt

[18] SAJIP, V.; MICK, T.: PEP 282 - A logging system [online]. [cit. 2019-04-05]. Dostupné z: https://legacy.python.org/dev/peps/pep-0282/

[19] ŠPAČEK, P.: Lecture 05b - Creational patterns(Prototype, Singleton, Builder) [online]. [cit. 2019-04-05]. Dostupné z: https://moodle.fit.cvut.cz/course/format/wiki/mediafile.php?id=

77&path=%2flectures%2fmiadp-2017-lecture05b-creational-patterns.pdf

[20] HORDĚJČUK, V.: GoF: Tovární metoda (Factory Method) [online]. [cit. 2019-04-05]. Dostupné z: http://voho.eu/wiki/factory-method/

[21] ŠPAČEK, P.: Lecture 05 - Creational patterns(Prototype, Singleton, Builder) and Structural patterns(Facade, Adapter, Bridge, Flyweight, Composite) [online]. [cit. 2019-04-05]. Dostupné z: https://moodle.fit.cvut.cz/pluginfile.php/197/course/section/10597/miadp-2018-lecture05-creational-and-structural-patterns.pdf

[22] OODESIGN.COM: Composite Pattern [online]. [cit. 2019-04-05]. Dostupné z: https://www.oodesign.com/composite-pattern.html

[23] BEAZLEY, D.: SLY (Sly Lex-Yacc) [online]. [cit. 2019-04-06]. Dostupné z: https://sly.readthedocs.io/en/latest/

[24] PYTHON SOFTWARE FOUNDATION: PEP 245 – Python Interface Syntax [online]. [cit. 2019-04-06]. Dostupné z: https://www.python.org/dev/peps/pep-0245/

[25] HACKERNOON.COM: I Finally Understand Static vs. Dynamic Typing and You Will Too! [online]. [cit. 2019-04-26]. Dostupné z: https://hackernoon.com/i-finally-understand-static-vs-dynamic-typing-and-you-will-too-ad0c2bd0acc7

[26] PAREWA LABS PVT. LTD.: Python Multiple Inheritance [online]. [cit. 2019-04-06]. Dostupné z: https://www.programiz.com/python-programming/multiple-inheritance

[27] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP: SELECT SQL Commands [online]. [cit. 2019-04-16]. Dostupné z: https://www.postgresql.org/docs/current/sql-select.html

[28] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP: Oracle to Postgres Conversion [online]. [cit. 2019-04-16]. Dostupné z: https://wiki.postgresql.org/wiki/Oracle_to_Postgres_Conversion

[29] ORACLE: The UNION [ALL], INTERSECT, MINUS Operators [online]. [cit. 2019-04-16]. Dostupné z: https://docs.oracle.com/database/121/SQLRF/queries004.htm#SQLRF52341

[30] MARIADB: Changes & Improvements in MariaDB 10.3 [online]. [cit. 2019-04-17]. Dostupné z: https://mariadb.com/kb/en/library/changes-improvements-in-mariadb-103/

[31] MARIADB: Changes & Improvements in MariaDB 10.4 [online]. [cit. 2019-04-17]. Dostupné z: https://mariadb.com/kb/en/library/changes-improvements-in-mariadb-104/

[32] GARCIA-MOLINA, H.; ULMANN, J. D.; WIDOM, J.: *Database Systems: The Complete Book*. Pearson, druhé vydání, 2009, ISBN 978-0131873254.

[33] KESSLER, J.: RelaX – Help [online]. [cit. 2019-04-22]. Dostupné z: https://dbis-uibk.github.io/relax/help.htm#relalg-operator-precedence

Příloha **A**

# Seznam použitých zkratek

**RAT** Relational Algebra Translator

**RA** Relační algebra / Relational algebra

**SQL** Structured Query Language

**API** Application Programming Interface

**MS** Microsoft

**RelaX** Relational Algebra Executor

**FIT** Fakulta informačních technologií

**ČVUT** České vysoké učení technické

**RDBMS** Relational Database Management Systems

**SRP** Single Responsibility Principle

**OCP** Open-Closed Principle

**GoF** Gang of Four

**PEP** Python Enhancement Proposals

**MRO** Method Resolution Order

**MB** Megabyte

**JSON** JavaScript Object Notation

Příloha B

# Obsah přiloženého CD

```
readme.txt ................................ stručný popis obsahu CD
src
  impl ...................................... zdrojové kódy implementace
  thesis .................................... zdrojová forma práce ve formátu LaTeX
DP_Kubis_Martin_2019.pdf ..................... text práce ve formátu PDF
```

Příloha C

# Obsah GIT repozitáře

```
RAT
├── doc
│   ├── api ...................... Dokumentace rozhraní stažená z Apiary
│   ├── source_code ................... Dokumentace ze zdrojového kódu
│   │   ├── output
│   │   │   └── index.html .................. Výchozí soubor dokumentace
│   │   └── Doxyfile .. Konfigurační soubor pro generování dokumentace ze
│   │       zdroj. kódu
│   └── state_machine_diagram
│       ├── fit_lexer_state_machine.jpg ........ Vizualizace stavového
│       │   automatu lexikálního analyzátoru
│       └── fit_lexer_state_machine.xml ... Zdrojový soubor stavového
│           automatu pro draw.io
├── rat ............. Adresář obsahující zdrojové kódy nového překladače
├── src
│   ├── db ..... Adresář obsahující databázové CREATE a INSERT scripty
│   ├── grammar .......... Adresář obsahující pomocný soubor pro ověření
│   │   korektnosti gramatiky
│   ├── nginx
│   │   └── sites_available ........ Konfigurace webového serveru nginx
│   └── passenger_wsgi.py ........ Soubor pro přístup skrze API v rámci
│       webového serveru
├── CHANGELOG.md ....... Log změn mezi jednotlivými verzemi překladače
├── LICENSE.md ....................................... Soubor s licencí
├── README.md ........... Programátorská příručka, informace o projektu
├── generate_documentation.sh ...... Skript pro generaci dokumentace
├── setup.py ............................ Soubor pro sestavení aplikace
└── unit_test.sh ...... Skript pro spuštění veškerých jednotkových testů
```

Příloha D

# Seznam rozpoznatelných tokenů

`RELATION_NAME` – Název relace (tabulky)

`DECLARATION` – `':='` – Deklarace relace

`PROJECTION_BEGIN` – `'['` – Začátek projekce

`PROJECTION_END` – `']'` – Konec projekce

`SELECTION_BEGIN` – `'('` – Začátek selekce

`SELECTION_END` – `')'` – Konec selekce

`NATURAL_JOIN` – `'*'` – Přirozené spojení

`L_NATURAL_SEMI_JOIN` – `'<*'` – Levé přirozené polospojení

`R_NATURAL_SEMI_JOIN` – `'*>'` – Pravé přirozené polospojení

`L_NATURAL_ANTI_JOIN` – `'!<*'` – Levý přirozený antijoin

`R_NATURAL_ANTI_JOIN` – `'!*>'` – Pravý přirozený antijoin

`L_NATURAL_OUTER_JOIN` – `'*^L'` – Levé přirozené vnější spojení

`R_NATURAL_OUTER_JOIN` – `'*^R'` – Pravé přirozené vnější spojení

`F_NATURAL_OUTER_JOIN` – `'*^F'` – Plné přirozené vnější spojení

`JOIN_BEGIN` – `'['` – Počáteční značka obecného spojení

`JOIN_END` – `']'` – Ukončující značka obecného spojení

`L_SEMI_JOIN_BEGIN` – `'<'` – Počáteční značka levého obecného spojení

`L_SEMI_JOIN_END` – `']'` – Ukončující značka levého obecného spojení

`R_SEMI_JOIN_BEGIN` – `'['` – Počáteční značka pravého obecného spojení

R_SEMI_JOIN_END – '>' – Ukončující značka pravého obecného spojení

L_ANTI_JOIN_BEGIN – '!<' – Počáteční značka levého obecného antijoinu

L_ANTI_JOIN_END – ']' – Ukončující značka levého obecného antijoinu

R_ANTI_JOIN_BEGIN – '![' – Počáteční značka pravého obecného antijoinu

R_ANTI_JOIN_END – '>' – Ukončující značka pravého obecného antijoinu

L_OUTER_JOIN_BEGIN – '[' – Počáteční značka levého vnějšího spojení

L_OUTER_JOIN_END – ']^L' – Ukončující značka levého vnějšího spojení

R_OUTER_JOIN_BEGIN – '[' – Počáteční značka pravého vnějšího spojení

R_OUTER_JOIN_END – ']^R' – Ukončující značka pravého vnějšího spojení

F_OUTER_JOIN_BEGIN – '[' – Počáteční značka plného vnějšího spojení

F_OUTER_JOIN_END – ']^F' – Ukončující značka plného vnějšího spojení

CROSS_JOIN – '×' – Kartézský součin

RELATIONAL_DIVISION – '÷' – Relační dělení

INTERSECT – '∩' – Množinový průnik

UNION – '∪' – Množinové sjednocení

MINUS – '\' – Množinový rozdíl

COLUMN_NAME – Název sloupce vč. příp. tabulky

COLUMN_ALIAS – př. id nebo "nejake id"– Název sloupce po přejmenování

DATE – 'dd.mm.yyyy' – Datum

NUMBER – Číslo, může být i desetinné

STRING – Řetězec uzavřený mezi apostrofy '

COMMA – ',' – Čárka (mezi sloupci)

RENAME – '->', '→' – Znak pro přejmenování

COMPARATOR – '!=', '<>', ... – Operátory pro porovnání

ARITHMETIC_PLUS – '+' – Znaménko plus

ARITHMETIC_MINUS – '-' – Znaménko mínus

ARITHMETIC_TIMES – '*' – Znaménko krát

ARITHMETIC_DIVISION – '/' – Znaménko děleno

LOGICAL_OPERATOR – '∨', 'OR','∧', 'AND' – Logické operátory, lze i malými písmeny

LOGICAL_NOT – '¬', 'NOT', 'not' – Negace

L_CURLY_BRACKET – '{' – Levá složená závorka

R_CURLY_BRACKET – '}' – Pravá složená závorka

L_ROUND_BRACKET – '(' – Levá kulatá závorka

R_ROUND_BRACKET – ')' – Pravá kulatá závorka

Příloha **E**

# Kompletní definice gramatických pravidel

Níže je uvedený kompletní seznam gramatických pravidel pro rozpoznávání veškeré syntaxe relační algebry vyučované v předmětu Databázové systémy (BI-DBS) na FIT ČVUT. Tato gramatická pravidla zahrnují i pravidla pro rozšíření diskutovaná v kapitole 5. Startovacím symbolem gramatiky je symbol nejvýše uvedený.

⟨*declaration*⟩ ::= ‘`RELATION_NAME`’ ‘`DECLARATION`’ ⟨*operation*⟩ ⟨*declaration*⟩
  | ::= ⟨*operation*⟩

⟨*operation*⟩ ::= ⟨*curlyBrackets*⟩
  | ⟨*unaryOperation*⟩
  | ⟨*binaryOperation*⟩

⟨*curlyBrackets*⟩ ::= ‘`L_CURLY_BRACKET`’ ⟨*operation*⟩ ‘`R_CURLY_BRACKET`’
  | ⟨*curlyBrackets*⟩ ‘`PROJECTION_BEGIN`’ ⟨*projection*⟩ ‘`PROJECTION_END`’
  | ⟨*curlyBrackets*⟩ ‘`SELECTION_BEGIN`’ ⟨*selection*⟩ ‘`SELECTION_END`’

⟨*unaryOperation*⟩ ::= ‘`TABLE_NAME`’
  | ⟨*unaryOperation*⟩ ‘`PROJECTION_BEGIN`’ ⟨*projection*⟩ ‘`PROJECTION_END`’
  | ⟨*unaryOperation*⟩ ‘`SELECTION_BEGIN`’ ⟨*selection*⟩ ‘`SELECTION_END`’

⟨*projection*⟩ ::= ⟨*multipleColumns*⟩
  | ⟨*renameColumn*⟩
  | ⟨*column*⟩

⟨*multipleColumns*⟩ ::= ⟨*projection*⟩ ‘`COMMA`’ ⟨*renameColumn*⟩
  | ⟨*projection*⟩ ‘`COMMA`’ ⟨*column*⟩

⟨*renameColumn*⟩ ::= ⟨*column*⟩ ‘`RENAME`’ ⟨*columnAlias*⟩

⟨*column*⟩ ::= ‘COLUMN_NAME’

⟨*columnAlias*⟩ ::= ‘COLUMN_ALIAS’

⟨*selection*⟩ ::= ⟨*roundBrackets*⟩
| ⟨*compoundCondition*⟩
| ⟨*conditionNegation*⟩
| ⟨*condition*⟩

⟨*roundBrackets*⟩ ::= ‘L_ROUND_BRACKET’ ⟨*selection*⟩ ‘R_ROUND_BRACKET’

⟨*compoundCondition*⟩ ::= ⟨*selection*⟩ ‘LOGICAL_OPERATOR’ ⟨*roundBrackets*⟩
| ⟨*selection*⟩ ‘LOGICAL_OPERATOR’ ⟨*conditionNegation*⟩
| ⟨*selection*⟩ ‘LOGICAL_OPERATOR’ ⟨*condition*⟩

⟨*conditionNegation*⟩ ::= ‘LOGICAL_NOT’ ⟨*roundBrackets*⟩

⟨*condition*⟩ ::= ⟨*expression*⟩ ‘COMPARATOR’ ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*expression*⟩ ‘ARITHMETIC_PLUS’ ⟨*term*⟩
| ⟨*expression*⟩ ‘ARITHMETIC_MINUS’ ⟨*term*⟩
| ⟨*term*⟩

⟨*term*⟩ ::= ⟨*term*⟩ ‘ARITHMETIC_TIMES’ ⟨*factor*⟩
| ⟨*term*⟩ ‘ARITHMETIC_DIVISION’ ⟨*factor*⟩
| ⟨*factor*⟩

⟨*factor*⟩ ::= ‘COLUMN_NAME’
| ‘DATE’
| ‘NUMBER’
| ‘STRING’
| ‘L_ROUND_BRACKET’ ⟨*expression*⟩ ‘R_ROUND_BRACKET’

⟨*binaryOperation*⟩ ::= ⟨*crossJoin*⟩
| ⟨*naturalJoin*⟩
| ⟨*leftNaturalSemiJoin*⟩
| ⟨*rightNaturalSemiJoin*⟩
| ⟨*leftNaturalAntiJoin*⟩
| ⟨*rightNaturalAntiJoin*⟩
| ⟨*leftNaturalOuterJoin*⟩
| ⟨*rightNaturalOuterJoin*⟩
| ⟨*fullNaturalOuterJoin*⟩
| ⟨*join*⟩
| ⟨*leftSemiJoin*⟩
| ⟨*rightSemiJoin*⟩
| ⟨*leftAntiJoin*⟩

| ⟨*rightAntiJoin*⟩
| ⟨*leftOuterJoin*⟩
| ⟨*rightOuterJoin*⟩
| ⟨*fullOuterJoin*⟩
| ⟨*union*⟩
| ⟨*intersect*⟩
| ⟨*minus*⟩
| ⟨*relationalDivision*⟩

⟨*crossJoin*⟩ ::= ⟨*operation*⟩ ‘CROSS_JOIN’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘CROSS_JOIN’ ⟨*unaryOperation*⟩

⟨*naturalJoin*⟩ ::= ⟨*operation*⟩ ‘NATURAL_JOIN’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘NATURAL_JOIN’ ⟨*unaryOperation*⟩

⟨*leftNaturalSemiJoin*⟩ ::= ⟨*operation*⟩ ‘L_NATURAL_SEMI_JOIN’
⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘L_NATURAL_SEMI_JOIN’ ⟨*unaryOperation*⟩

⟨*rightNaturalSemiJoin*⟩ ::= ⟨*operation*⟩ ‘R_NATURAL_SEMI_JOIN’
⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘R_NATURAL_SEMI_JOIN’ ⟨*unaryOperation*⟩

⟨*leftNaturalAntiJoin*⟩ ::= ⟨*operation*⟩ ‘L_NATURAL_ANTI_JOIN’
⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘L_NATURAL_ANTI_JOIN’ ⟨*unaryOperation*⟩

⟨*rightNaturalAntiJoin*⟩ ::= ⟨*operation*⟩ ‘R_NATURAL_ANTI_JOIN’
⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘R_NATURAL_ANTI_JOIN’ ⟨*unaryOperation*⟩

⟨*leftNaturalOuterJoin*⟩ ::= ⟨*operation*⟩ ‘L_NATURAL_OUTER_JOIN’
⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘L_NATURAL_OUTER_JOIN’ ⟨*unaryOperation*⟩

⟨*rightNaturalOuterJoin*⟩ ::= ⟨*operation*⟩ ‘R_NATURAL_OUTER_JOIN’
⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘R_NATURAL_OUTER_JOIN’ ⟨*unaryOperation*⟩

⟨*fullNaturalOuterJoin*⟩ ::= ⟨*operation*⟩ ‘F_NATURAL_OUTER_JOIN ’
⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘F_NATURAL_OUTER_JOIN’ ⟨*unaryOperation*⟩

⟨*join*⟩ ::= ⟨*operation*⟩ ‘JOIN_BEGIN’ ⟨*joinClause*⟩ ‘JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘JOIN_BEGIN’ ⟨*joinClause*⟩ ‘JOIN_END’ ⟨*unaryOperation*⟩

⟨*leftSemiJoin*⟩ ::= ⟨*operation*⟩ ‘L_SEMI_JOIN_BEGIN’ ⟨*joinClause*⟩
‘L_SEMI_JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘L_SEMI_JOIN_BEGIN’ ⟨*joinClause*⟩ ‘L_SEMI_JOIN_END’
⟨*unaryOperation*⟩

⟨*rightSemiJoin*⟩ ::= ⟨*operation*⟩ ‘R_SEMI_JOIN_BEGIN’ ⟨*joinClause*⟩
‘R_SEMI_JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘R_SEMI_JOIN_BEGIN’ ⟨*joinClause*⟩ ‘R_SEMI_JOIN_END’
⟨*unaryOperation*⟩

⟨*leftAntiJoin*⟩ ::= ⟨*operation*⟩ ‘L_ANTI_JOIN_BEGIN’ ⟨*joinClause*⟩
‘L_ANTI_JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘L_ANTI_JOIN_BEGIN’ ⟨*joinClause*⟩ ‘L_ANTI_JOIN_END’
⟨*unaryOperation*⟩

⟨*rightAntiJoin*⟩ ::= ⟨*operation*⟩ ‘R_ANTI_JOIN_BEGIN’ ⟨*joinClause*⟩
‘R_ANTI_JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘R_ANTI_JOIN_BEGIN’ ⟨*joinClause*⟩ ‘R_ANTI_JOIN_END’
⟨*unaryOperation*⟩

⟨*leftOuterJoin*⟩ ::= ⟨*operation*⟩ ‘L_OUTER_JOIN_BEGIN’ ⟨*joinClause*⟩
‘L_OUTER_JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘L_OUTER_JOIN_BEGIN’ ⟨*joinClause*⟩ ‘L_OUTER_JOIN_END’
⟨*unaryOperation*⟩

⟨*rightOuterJoin*⟩ ::= ⟨*operation*⟩ ‘R_OUTER_JOIN_BEGIN’ ⟨*joinClause*⟩
‘R_OUTER_JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘R_OUTER_JOIN_BEGIN’ ⟨*joinClause*⟩ ‘R_OUTER_JOIN_END’
⟨*unaryOperation*⟩

⟨*fullOuterJoin*⟩ ::= ⟨*operation*⟩ ‘F_OUTER_JOIN_BEGIN’ ⟨*joinClause*⟩
‘F_OUTER_JOIN_END’ ⟨*curlyBrackets*⟩
| ⟨*operation*⟩ ‘F_OUTER_JOIN_BEGIN’ ⟨*joinClause*⟩ ‘F_OUTER_JOIN_END’
⟨*unaryOperation*⟩

⟨*joinClause*⟩ ::= ⟨*joinRoundBrackets*⟩
| ⟨*joinCompoundCondition*⟩
| ⟨*joinConditionNegation*⟩
| ⟨*joinCondition*⟩

⟨*joinRoundBrackets*⟩ ::= ‘L_ROUND_BRACKET’ ⟨*joinClause*⟩ ‘R_ROUND_BRACKET’

⟨*joinCompoundCondition*⟩ ::= ⟨*joinClause*⟩ ‘LOGICAL_OPERATOR’
⟨*joinRoundBrackets*⟩
| ⟨*joinClause*⟩ ‘LOGICAL_OPERATOR’ ⟨*joinConditionNegation*⟩
| ⟨*joinClause*⟩ ‘LOGICAL_OPERATOR’ ⟨*joinCondition*⟩

⟨*joinConditionNegation*⟩ ::= ‘`LOGICAL_NOT`’ ⟨*joinRoundBrackets*⟩

⟨*joinCondition*⟩ ::= ⟨*joinColumn*⟩ ‘`COMPARATOR`’ ⟨*joinColumn*⟩

⟨*joinColumn*⟩ ::= ‘`COLUMN_NAME`’

⟨*union*⟩ ::= ⟨*operation*⟩ ‘`UNION`’ ⟨*curlyBrackets*⟩
  | ⟨*operation*⟩ ‘`UNION`’ ⟨*unaryOperation*⟩

⟨*intersect*⟩ ::= ⟨*operation*⟩ ‘`INTERSECT`’ ⟨*curlyBrackets*⟩
  | ⟨*operation*⟩ ‘`INTERSECT`’ ⟨*unaryOperation*⟩

⟨*minus*⟩ ::= ⟨*operation*⟩ ‘`MINUS`’ ⟨*curlyBrackets*⟩
  | ⟨*operation*⟩ ‘`MINUS`’ ⟨*unaryOperation*⟩

⟨*relationalDivision*⟩ ::= ⟨*operation*⟩ ‘`RELATIONAL_DIVISION`’ ⟨*curlyBrackets*⟩
  | ⟨*operation*⟩ ‘`RELATIONAL_DIVISION`’ ⟨*unaryOperation*⟩