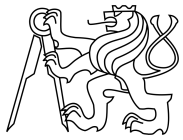
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Návrh a implementace analýzy datových toků pro jazyk Pig v projektu Manta
Student:	Andrej Taňkoš
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat funkční prototyp modulu, který provede syntaktickou a sémantickou analýzu skriptů v jazyku Pig a následně její výsledek využije pro analýzu datových toků. Modul bude zařazen do projektu Manta.

1. Seznamte se s projektem Manta a s nástrojem Apache Pig.
2. Analyzujte možnosti jak využít při analýze datových toků samotný zdrojový kód nástroje Pig a zvažte přínosy a rizika tohoto přístupu.
3.Navrhněte modul v projektu Manta pro zpracování jazyka Pig. Využijte existující infrastrukturu projektu.
4. Implementujte prototyp, řádně ho zdokumentujte a otestujte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 31. října 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Návrh a implementace analýzy datových toků pro jazyk Pig v projektu Manta

Andrej Taňkoš

Katedra softwarového inženýrství
Vedúci práce: Ing. Michal Valenta, Ph.D.

14. mája 2018

Poďakovanie

Rád by som poďakoval vedúcemu tejto práce Ing. Michalovi Valentovi Ph.D. za jeho pripomienky a cenné rady. Taktiež chcem poďakovať členom Manta projektu, hlavne Mgr. Jirkovi Touškovi a RNDr. Lukášovi Hermannovi za poskytnutie možnosti podieľať sa na projekte a ich pomoc. Nakoniec by som rád poďakoval Jánovi Burdeľovi za prečítanie a zhodnotenie tejto práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(a) všetky použité informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. V súlade s ust. § 2373 občianskeho zákonníka týmto udeľujem nevýhradné oprávnenie (licenciu) k použitiu tejto mojej práce, a to vrátane všetkých počítačových programov, ktoré sú jej súčasťou či prílohou a všetkej jej dokumentácie (ďalej súhrne len „Dielo“), a to všetkým osobám, ktoré si prajú Dielo použiť. Tieto osoby sú oprávnené Dielo použiť akýmkoľvek spôsobom, ktorý neznižuje hodnotu Diela a za akýmkoľvek účelom (vrátane užitia k zárobkovým účelom), vrátane možnosti Dielo upraviť či meniť, spojiť ho s iným dielom a/alebo zaradiť ho do diela súborného. Toto oprávnenie je časovo, teritoriálne a množstvom neobmedzené a udeľujem ho bezúplatne.

V Prahe 14. mája 2018 .....

Odkaz na túto prácu

Taňkoš, Andrej. *Návrh a implementace analýzy datových toků pro jazyk Pig v projektu Manta*. Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Získavanie dátových tokov z dat je veľmi obľúbený pojem v biznis sfére. Pre manažérov a pracovníkov veľkých korporácií, ktorí pracujú s veľkým množstvom informácií príde vhodné mať nástroj, ktorý im vizualizuje dátové zmeny a závislosti dát v nimi používaných technológiách. Nástroj takéhoto typu im umožní pracovať lepšie s dátami a zlepšiť efektivitu. Tvorbou takéhoto nástroja sa zaoberá projekt Manta, ktorý pre svoji zákazníkov poskytuje nástroj Manta Flow. Tento nástroj dokáže vizualizovať dátové toky pre množstvo známych technológií, avšak v momentálnej dobe neponúka podporu pre jazyk Pig Latin. Pig Latin je jazyk, ktorý sa používa v platforme Apache Pig a slúži na vytváranie programov, ktoré analyzujú a spracovávajú dáta.

Preto je cieľom tejto práce je navrhnúť a implementovať modul vykonávajúci analýzu a zisk dátových tokov pre Pig Latin programy, ktorý je možné integrovať do nástroja Manta Flow. Tento modul má za úlohu vykonať syntaktickú a sémantickú analýzu kódu, a z jej výsledku generovať dátové toky vo forme grafu. Modul je navrhnutý a implementovaný, tak aby spolupracoval s Manta systémom, čo umožňuje jednoduchú integráciu toho modulu do nástroja Manta Flow. Práca obsahuje popis jazyka Pig Latin, dokumentácie návrhu a implementácie výsledného modulu, a taktiež popis procesu testovania modulu.

Kľúčové slová návrh a implementácia Java modulu, analýza dátových tokov, Pig Latin, Manta, Big Data

Abstract

Getting data flows from data is a very popular concept in the business sphere. For managers and employees of large corporations who are working with a great deal of information, it is a good idea to have a tool that visualizes the data changes and data dependencies in the technologies used. A tool of this type will allow them to better work with data and improve efficiency. The creator of such a tool is Manta project, which provides the tool Manta Flow for its customers. This tool can visualize data flows for a number of well-known technologies, but currently does not offer Pig Latin support. Pig Latin is the language used in the Apache Pig platform to create programs that analyze and process data.

Therefore, the aim of this paper is to design and implement a module for analysis and acquisition of data flows from Pig Latin programs that can be integrated into Manta Flow. This module performs syntactic and semantic code analysis and generates graph of data flows from the result. The module is designed and implemented to work with the Manta system, allowing for easy integration of this module into Manta Flow. The work contains a description of Pig Latin, documentation for the design and implementation of the resulting module, as well as a description of the module testing process.

Keywords design and implementation of Java module, data lineage analysis, Pig Latin, Manta, Big Data

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

S narastajúcim počtom informačných technológií, ktoré sa denne používajú, narastá aj objem dát. V prípade veľkých spoločností, ktoré s dátami pracujú, sú tieto dáta veľmi objemné a nie je jednoduché ich spracovať. Jednou z možností ako tieto objemné dáta analyzovať a spracovať je použiť nástroj Apache Pig. Tento nástroj je postavený na Apache Hadoop ekosystéme, ktorý je známy tým, že sa zaoberá spracovaním veľkého množstva dat. Apache Pig programy môžu byť dlhé, komplexné, obsahovať veľké množstvo dátových transformácií a závislosti na vstupoch a výstupoch. Z toho dôvodu je priaznivé mať nástroj pre grafické zobrazenie významných informácií o dátových cykloch v kóde, ktoré sa nazývajú dátové toky. Takíto nástroj pomôže koncovému užívateľovi vylepšiť prácu s dátami, nájsť chyby a zlepšiť kvalitu kódu.

Práve týmto problémom sa zaoberá projekt Manta. Nástroje projektu Manta dokážu vizualizovať životné cykly dát pre množstvo používaných technológií, ale aktuálne neexistuje podpora pre Pig Latin, čo je aj dôvod výberu tejto práce. V tejto práci vytvorím modul, ktorý dokáže analyzovať a získať dátové toky z Pig Latin programov. Tento modul bude následne zadený do Manta systému. Táto práca sa delí na päť kapitol. V prvej kapitole uvádzam základné pojmy, s ktorými sa pracuje počas celej práce. V druhej kapitole nasleduje analýza požiadavkov na výsledný produkt a rozbor kľúčových technológií, ktoré sa budú používať v implementačnej časti. Rovnaká časť analyzuje jazyk Pig Latin a spracováva témy syntaktickej a sémantickej analýzu kódu. S využitím týchto poznatkov v tretej kapitole nasleduje časť návrhová, ktorá popisuje návrh modulu. Posledné dve kapitoly sa zaoberajú implementáciou a testovaním modulu.

Cieľ práce

Hlavným cieľom tejto práce je vytvoriť modul, ktorý dokáže analyzovať a získať dátové toky z kódu jazyka Pig Latin. Tento modul vykonáva syntaktickú a

sémantickú analýzu kódu, a jej výsledok následne používa pri generovaní grafu dátových tokov. Tento cieľ sa delí na časť analytickú a časť implementačnú. V analytickej časti je cieľom analýza štruktúry jazyka, výber jeho kľúčových častí, ktoré sú dôležité pre analýzu dátových tokov a spracovanie týchto častí spolu s návrhom riešenia. Taktiež je cieľom zoznámiť sa v priebehu návrhu so systémom Manta, s ktorým má výsledný modul komunikovať. Práca sa naviac zaoberá zdrojovým kódom nástroja Apache Pig, prínosmi a rizikami jeho použitia. S vytvoreným návrhom sa v časti implementačnej zaoberám tvorbou navrhnutého modulu a jeho testovaním.

Kapitola 1

Základné pojmy

1.1 Dátové toky

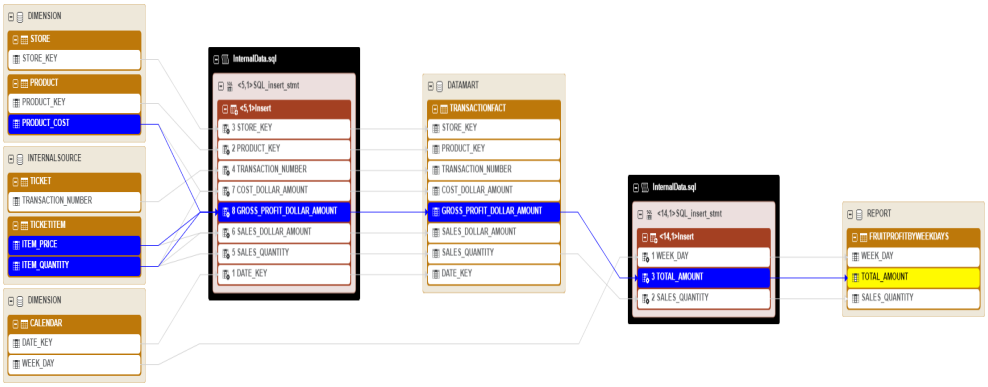
Podľa [1], dátové toky reprezentujú životný cyklus dat, ktorý zahŕňa zdroj dat a kam sa dáta pohybujú v čase. V tejto práci sa dátové toky reprezentujú vo forme grafu. Jednotlivé uzly grafu predstavujú dáta a hrany medzi nimi popisujú dátové toky. Tieto hrany sú vždy orientované.

Pre potreby tejto práce sa dátové toky rozdeľujú na dve kategórie. Sú to priame a filter (nepriame) toky. Priame toky popisujú priamy vzťah zdrojových dát k cieľovým dátam. To predstavuje vzťah, kedy sa zdrojové dáta podieľajú na tvorbe cieľových dát. V prípade nepriamych dátových tokov ide o nepriame ovplyvnenie cieľových dat zdrojovými dátami. Nepriame toky sa často vyskytujú v rôznych filtrovacích a agregačných operáciach, preto sa nazývajú aj filter toky. Nepriamé dátové toky sa nikdy priamo neúčastnia na tvorbe iných dat ale len ovplyvňujú vznik iných dat.

1.2 Manta

Manta je projekt zaoberajúci sa tvorbou BI nástrojov. BI nástroje označujú softvér, ktorý zhromažďuje a analyzuje dáta z firemných systémov a výsledky prezentuje užívateľovi. Tieto výsledky poskytujú zamestnancom a manažérom prínosy ako sú, urýchlenie a vylepšenie rozhodovania alebo zvýšenie efektivity. [2]

Manta pre tieto potreby poskytuje nástroj Manta Flow. Tento nástroj vytvára detailnú vizualizácia BI prostredia pre množstvo technológii, medzi ktoré patrí napr. Oracle, Microsoft SQL Server a Apache Hive. Jeho úlohou je extrahovať významné informácie z poskytnutých dat do formy grafu dátových tokov, a ten následne vizualizovať. Na obrázku 1.1 je možné príklad tejto vizualizácie vidieť. Jednotlivé uzly v obrázku reprezentujú nejakú formu dát a hrany medzi nimi reprezentujú dátové toky. Prínosom tohoto nástroja je



Obr. 1.1: Manta Flow vizualizácia [4]

zdokonalenie práce s dátami, zvýšenie efektivity a eliminovanie manuálnej práce. [3]

Spracovanie SQL kódu v nástroji Manta Flow je možné popísať podľa 3 krokov, ktoré sú:

1. Nástroj Manta Flow spracováva SQL kód.
2. Počas spracovania dokumentuje dátové toky.
3. Dátové toky vizualizuje alebo ich posunie tretej strane.

[3] Modul pre analýzu dátových tokov jazyka Pig Latin, ktorý je cieľom tejto práce, vykonáva rovnaké kroky.

1.3 Apache Pig

Apache Pig, ďalej už len Pig, je platforma pre analýzu a spracovanie veľkého množstva štrukturovaných aj neštrukturovaných dát, ktorá vyhodnocuje programy napísané vo vysoko-úrovňovom programovacom jazyku Pig Latin. Jedna z hlavných výhod použitia platformy Pig pre spracovanie dát je schopnosť paralelizácie v Pig Latin programoch, ktorá v konečnom dôsledku umožní spracovanie veľkého množstva dát efektívne. Ďalšia výhoda je v tom, že Apache Pig je vydaný pod voľnou licenciou, takže je jeho použitie bezplatné. Infraštruktúra nástroja Pig aktuálne pozostáva z prekladača, ktorý prevedie zdrojový kód Pig Latin programov do kódu MapReduce[1] programov, ktoré následne spracuje Apache Hadoop[2].[5]

Vlastnosti jazyka Pig Latin sú [5]:

[1] Predstavuje programovací model pre spracovanie veľkého množstva dat.

[2] Jedná sa o iný projekt na spracovanie veľkého množstva dát, na ktorom je Apache Pig postavený.

- Možnosť vyjadrenia komplexných úloh pomocou jednoduchých transformácií, ktoré predstavujú postupnosť dátových tokov.
- Automatická optimalizácia kódu.
- Možnosť rozšírenia jazyka o nové funkcie.

Z vlastnosti uvedených vyššie prichádza Pig Latin ako vhodný jazyk pre spracovanie veľkého množstva dát, ktoré je možné vykonať pomocou postupnosti jednoduchých transformácii. Do jazyka je taktiež možné pridať nové funkcie, čo obohatí jazyk o nové transformácie.

Pig ponúka dve možnosti vyhodnocovania programov. V prvom prípade ide o vyhodnocovanie skriptu, ktorý sa predá ako vstupný parameter. Druhá možnosť, ktorá sa používa v tejto práci, je vyhodnocovanie Pig Latin príkazov v interaktívnom móde. Tento mód dáva výhodu v tom, že je možné používať debugovacie príkazy za behu, a tak lepšie preskúmať štruktúru a správanie dát a operácií.

Kapitola 2

Analýza

Táto kapitola sa zaoberá analytickou stránkou tejto práce. Kapitola sa delí na niekoľko podkapitol, kde v prvej prezentujem analýzu požiadavkov na výsledný produkt. Následne prichádza analýza a popis technológii, ktoré sa používajú v praktickej časti práce. V tretej podkapitole sa nachádzajú dôležité informácie o jazyku Pig Latin, ktoré je nutné analyzovať, a posledných častiach kapitoly je popis syntaktickej a sémantickej analýzy kódu.

2.1 Analýza požiadavkov

Táto kapitola uvádza jednotlivé požiadavky na výsledný produkt, ktoré sú získane od zadávateľa. Tieto požiadavky sa delia na funkčne a nefunkčné požiadavky, a popisuje ich následujúci zoznam.

Funkčné požiadavky

F1 Modul analyzuje a získa dátové toky z Pig Latin kódu vo forme grafu.

F2 Modul dokáže získať priame aj nepriame dátové toky.

F3 Modul umožňuje spracovanie Pig Latin kódu vo forme reťazca a súboru.

F4 Modul dokáže oznámiť chybu v prípade, že kód nie je správny.

Nefunkčné požiadavky

NF1 Modul vie spracovať vstup v rozumnom čase.

NF2 Modul využíva triedy a rozhranie Manta ekosystému pre možnú integráciu modulu.

Všetky tieto požiadavky definujú výsledný produkt ako modul, ktorý dokáže spracovať programovací kód vo viacerých formách (súbor, reťazec), ten analyzuje, a vytvori z tejto analýzy graf dátových tokov.

Jedným z funkčných požiadavkov je spracovanie kódu v rozumnom čase. Pod pojmom rozumný čas sa myslí doba, ktorá pre užívateľa nepredstavuje značné čakanie pri práci s modulom. Aby modul dokázal spracovať a produkovať výstup v rozumnom čase je potrebné implementovať modul pomocou moderných technológií a efektívnych algoritmov.

Dosiahnuť funkčné prepojenie vytvoreného modulu s Manta systémom je možné v prípade použitia kompatibilných alebo priamo rovnakých technológií. V tejto práci používam pre implementáciu modulu rovnaké technológie ako používa Manta systém. Táto voľba dáva výhodu aj v tom, že nie je potrebné analyzovať a skúmať použitie iných technológií. Technológie použité v Manta systéme sú osvedčené ako efektívne, pretože sú v nich implementované iné moduly. Táto voľba zároveň pomáha splniť NF1.

Následujúca sekcia popisuje technológie, ktoré sa používajú v tejto práci. Ku každej technológií je priložený jednoduchý popis významu danej technológie pre prípad, že čitateľ nejakú z nich nepozná.

2.2 Použité technológie

Programovací jazyk Java. Java je programovací jazyk a výpočtová platforma. Existuje množstvo projektov, ktoré používajú Javu. Podľa [6], Java je všade. Rovnako je veľmi rozšírená aj v projekte Manta. Java je použitá ako programovací jazyk pre nástroj analýzy dátových tokov v projekte Manta a rovnako je použitá aj pre implementáciu tohoto modulu. Predpokladom je, že čitateľ pozná základné koncepty tohoto jazyka, ktoré sú hojne použité v praktickej časti práce. V prípade, že tomu tak nie je, je vhodné tieto znalosti nadobudnúť pred čítaním kapitoly o implementácií.

Apache Maven. Apache Maven je nástroj, ktorý sa používa pre zostavenie a správu Java projektov [7]. Tento nástroj ponúka schopnosti riešenia závislosti na iných moduloch, knižniciach alebo projektoch, a tak je možné jednoducho vytvoriť modulárny produkt. Jeho služby sa používajú vo výslednom module pre správu a získavanie závislostí na Manta systéme a externých knižniciach. Správa Maven modulu pozostáva v konfiguračnom súbore, ktorý sa označuje ako `pom.xml` a je umiestnený v koreňovej zložke modulu. Tento súbor obsahuje definície závislostí a informácií o module, a pre každý modul je jedinečný.

JUnit. JUnit je framework, ktorý sa používa na testovanie Java aplikácií. Jeho princíp pozostáva v použití Java anotácií, ktoré sa vkladajú pred testovacie triedy a metódy. V tejto práci je použitý pre testovanie častí a rovnako aj pre testovanie výsledného modulu. Vďaka jeho obľúbenosti sa stal súčasťou

najpoužívanejších Java IDE (Eclipse, Idea). Tieto IDE ponúkajú možnosti, kde si užívateľ môže zvoliť testovacie metódy, odštartovať ich v jednom kliku a po skončení prehľadne vidieť úspešnosť zvolených testov.

2.3 Programovací jazyk Pig Latin

Pig Latin je programovací jazyk určený k tvorbe programov, ktoré sa vyhodnocujú na platforme Pig. Prvky jazyka, ktoré sú významné pre analýzu dátových tokov sa delia do kategórií:

1. Identifikátory, dátové typy, relácie, schémy a konštanty
2. Relačné operátory
3. Aritmetické a iné operátory
4. UDF

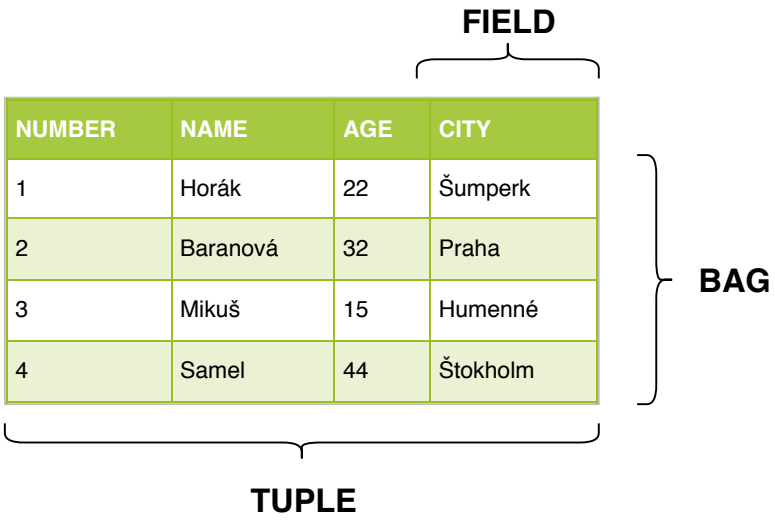
V tejto podkapitole prebieha analýza prvkov z týchto kategórii. Každá kategória obsahuje popis, čo dané prvky v jazyku predstavujú a taktiež čo predstavujú vo výslednom grafe dátových tokov.

Predtým ako rozoberiem každú kategóriu, uvádzam základné vlastnosti jazyka Pig Latin. Pig Latin, ako každý programovací jazyk, prichádza s množinou kľúčových slov, ktoré vytvárajú konštrukciu jazyka. Tie patria do množiny rezervovaných slov, majú určitú funkciu a nemôžu byť nimi označené užívateľom definované časti. Kľúčové slová sú „case insensitive“, čiže nezáleží na veľkosti písmen. Patria k nim vstavané funkcie, dátové typy, relačné operátory a taktiež aj iné prvky. Zoznam týchto slov je možné nájsť na manuálových stránkach Pig Latinu [8], kde sú taktiež uvedené informácie o vlastnostiach Pig Latin, ktoré popisujem v tejto podkapitole.

2.3.1 Identifikátory, dátové typy, relácie, schémy a konštanty

Identifikátory. Identifikátor predstavuje označenie pre užívateľom definovane prvky. Identifikátor môže pomenúvať relácie, vlastné funkcie, polia ale aj iné prvky. Identifikátor sa musí začínať písmenom a nasledovať ho môže postupnosť číslic, písmen alebo podčiarnikov. Všetky identifikátory sú „case sensitive“ čo znamená, že X(veľký znak) a x(malý znak) sú dva rozdielne identifikátory. Vo výslednom grafe dátových tokov, identifikátory označujúce dáta reprezentujú uzly.

Dátové typy. Všetky dáta v Pig Latin nadobúdajú nejaký dátový typ. V prípade, že dátový typ prvku nie je definovaný, tak je implicitný dátový typ bytearray. Dátové typy prichádzajú s jazykom a neexistuje možnosť ako pridať nové dátové typy.



Obr. 2.1: Príklad dátového modelu v Pig Latin

Základným stavebným kameňom reprezentácie dát v Pig Latin sú relácie. Relácie slúžia ako vstup a výstup pre relačné operátory, ktoré vykonávajú spracovanie dát. Dátový model, ktorý reprezentuje štruktúru relácie sa dá popísať podľa 4 pravidiel, ktoré sú:

1. Relácia je bag objekt.
2. Bag je kolekcia tuple objektov.
3. Tuple reprezentuje zoradenú[3] množinu polí[4].
4. Pole sú dáta.

Pre analógiu k relačným databázam, relácia v Pig Latin reprezentuje to čo tabuľka v databáze. Všetky tuple objekty v relácií reprezentujú to, čo riadky v tabuľke a jednotlivé ich polia sú rovné poliam, ktoré sú v tabuľke. Následujúci obrázok 2.1 ukazuje relačnú tabuľku s popisom čo dané časti reprezentujú v Pig Latin.

Polia, ktoré obsahujú dáta môžu nadobúdať ľubovolný dátový typ. Dátové typy sa podľa svojej reprezentácie delia na 4 kategórie:

- Jednoduché typy
- Tuple
- Bag

[3]Slovo zoradený reprezentuje vlastnosť, že záleží na poradí prvkov.
[4]V tomto kontexte, pole reprezentuje bunku.

- Map

Dáta, ktoré sú jednoduchého dátového typu reprezentujú jednu informáciu. V následujúcej tabuľke 2.1 je zhrnutie všetkých dátových typov s príkladom hodnoty, ktorú môžu nadobúdať.

Tabuľka 2.1: Tabuľka jednoduchých typov

Typ	Popis	Príklad
int	znamienkové 32-bitové celé číslo	10
long	znamienkové 64-bitové celé číslo	10L
float	32-bitové desatinné číslo	10.5F
double	64-bitové desatinné číslo	10.5
chararray	postupnosť čísel v UTF-8 kódovaní	Ahoj
bytearray	postupnosť bajtov	
boolean	boolean	true/false
datetime	dátum a čas	1970-01-01T00:00:00
biginteger	Java BigInteger	200000000000
bigdecimal	Java BigDecimal	33.456

Všetky ostatné dátové typy reprezentujú množinu dátových typov, ktoré sa nazývajú komplexné dátové typy. Komplexné dátové typy reprezentujú dáta, ktoré majú zložitejšiu štruktúru a reprezentuje ich viacero polí. Tieto typy popisujú následujúce odstavce a tabuľka 2.2 uvádza príklady ich použitia.

Bag. Bag je dátový typ, ktorý reprezentuje kolekciu tuple objektov. Narozdiel od ostatných komplexných typov, bag môže obsahovať len objekty tuple. Hlavnou vlastnosťou tohoto typu je, že bag reprezentuje kolekciu nezoradených dát. Táto vlastnosť sa využíva v relačných operátoroch, ktoré pracujú len s objektom bag (reláciou), pretože dáta v objekte bag je možné rozložiť na viacero skupín, a tak efektívne paralelne spracovať. Z konvencie sa dátový typ bag označuje pomocou zátvoriek typu `{ }`.

Tuple. Tuple je dátový typ, ktorý reprezentuje množinu zoradených dát. Ako už bolo spomenuté, tuple sa správa sa rovnako ako riadok v relačnej tabuľke. Tuple sa označuje pomocou zátvoriek typu `( )`. Narozdiel o relačných databáz, Pig Latin nepožaduje, aby všetky objekty tuple obsahovali rovnaký počet polí a ani aby polia, ktoré sú na rovnakej pozícií v objektoch tuple v relácií boli rovnakého typu. Polia v objekte tuple môžu obsahovať dáta ľubovolného jednoduchého aj komplexného typu.

Keďže tuple reprezentuje množinu zoradených polí, je možné sa na tieto polia odkazovať pomocou zadania pozície. Označenie pozície sa v Pig Latin reprezentuje pomocou znaku $, za ktorým nasleduje číselná pozícia. Tieto

číselné pozície začínajú od indexu nula, teda odkaz na prvé pole sa dá dosiahnuť pomocou výrazu „$0“.

Map. Map je dátový typ, ktorý reprezentuje dvojicu kľúč-hodnota. Dátový typ kľúča musí byť chararray ale dátový typ hodnoty môže byť ľubovolný dátový typ z Pig Latin. V ukážkových kódoch sa typ map označuje pomocou zátvoriek typu `[ ]`.

Tabuľka 2.2: Tabuľka komplexných typov

Typ	Popis	Príklad
tuple	zoradená množina dátových polí	(19,2)
bag	kolekcia objektov tuple	{(19,2), (18,1)}
map	pár kľúč-hodnota	[open#apache]

Relácie. Ako už bolo spomenuté, jediným prvkom, ktorý slúži ako vstup a výstup medzi operátormi je relácia. Relácia je pomenovaný objekt typu bag, presnejšie vonkajší bag[5]. Objekt typu bag vždy obsahuje len objekty typu tuple, čiže relácia reprezentuje kolekciu nezoradených dát. Relácia môže obsahovať aj objekty typu tuple, ktoré nemajú rovnaké dátové typy alebo majú iný počet položiek. V takomto prípade však nie je jasná štruktúra dát, ktorá sa v Pig Latin nazývaná schéma.

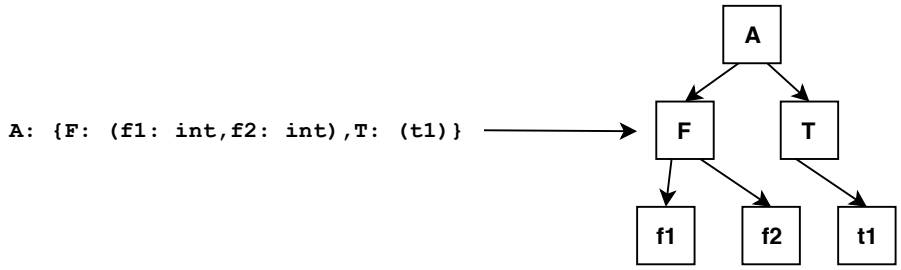
Schémy. Schéma predstavuje štruktúru dát relácie v Pig Latin. Jej úlohou je priradenie a popis dát, vďaka tomu, že priraďuje názvy a dátové typy poliam. Príklad použitia schémy je v následujúcom zdrojovom kóde 1.

```
A = LOAD '1.txt' AS (number:int, name:chararray,
                     age:int, city:chararray);
DESCRIBE A;
-- A: {number:int, name:chararray, age:int, city: chararray}
```

Zdrojový kód 1: Príklad použitia a priradenia schémy

Príkaz `DESCRIBE` slúži pre debugovacie účely. Pomocou neho je možné vypísať štruktúru relácie, ako je možné vidieť v kóde 1. V tejto práci budem vždy uvádzať výspis tohoto príkazu na následujúci riadok, kde tento výpis obsahuje prefix „–“, ktorý reprezentuje komentár v Pig Latin. Príkaz `LOAD` načítava dáta

[5]Vonkajší bag predstavuje bag, ktorý nereprezentuje žiadne dátové pole. Vonkajší bag sa v štruktúre dát nachádza vždy na najvyššej vrstve. Opak je vnútorný bag, ktorý nie je vonkajší.



Obr. 2.2: Prevod schémy do stromovej reprezentácie

zo súboru `1.txt` podľa schémy určenej za klauzulou `AS`. Táto schéma popisuje, že do relácie `A` sa načítavajú dáta ako štvorice. Po načítaní dát, schéma relácie `A` reprezentuje bag, ktorý obsahuje objekty tuple so štyrmi poliami. Vo výpise schémy relácie `A` sa neuvádza, že relácia obsahuje dátové položky tuple ale vyzerá to tak, že relácia obsahuje jednoduché dáta. V skutočnosti však relácia obsahuje objekty tuple a tieto polia patria do tuple. Dôvod prečo vo výpise schémy, Pig Latin vynecháva označenie tuple v relácií mi nie je známy ale najskôr to bude z dôvodu, že relácia môže obsahovať len objekty tuple, čiže výpis označenia tuple je redundantný.

Nástroj Pig dokáže spracovávať aj dáta, ktoré nemajú priradenú schému. V takomto prípade však je nie jasná štruktúra týchto dát, a preto sú dáta bez schémy často dôvodom chýb. Ďalšou nevýhodou nepoužitia schémy je neefektívnosť niektorých operácií. Z pohľadu analýzy dátových tokov to však nie je problém, pretože relácie, ktoré nemajú definovanú schému reprezentujú kolekciu objektov typu tuple s jedným nepomenovaným poľom typu bytearray.

Ako už bolo spomenuté, vo výslednom grafe dátových tokov uzly reprezentujú identifikátory a dáta. Každý identifikátor popisuje dáta a tieto dáta majú určitú schému. Pre účely analýzy dátových tokov je nutné túto schému reprezentovať aj vo výslednom grafe. Na nasledujúcom obrázku 2.2 je príklad prevodu reprezentácie schémy do stromovej reprezentácie, ktorá popisuje štruktúru dát. Jednotlivé uzly reprezentujú polia, kde každý uzol má odkaz na svojho rodiča a potomkov. V grafe je dôležité zachovať poradie uzlov, teda potomci sú reprezentovaný zoznamom.

Konštanty. V Pig Latin kóde je možné reprezentovať konštanty rôznych dátových typov. Výnimkami sú typy bytearray a datetime, pre ktoré neexistuje zápis konštanty. Jednoduchý príklad použitia konštanty je v ukážke 2.

V ukážke 2, operátor `FILTER` filtruje hodnoty podľa zadaného výrazu za klauzulou `BY`. Konštanty však nepredstavujú žiadne dáta, a preto vo výslednom grafe dátových tokov nemajú žiadnu rolu.

```
...
DESCRIBE A;
-- A: {a: bytearray}

X = FILTER A BY a == 0;
```

Zdrojový kód 2: Príklad použitia konštanty

2.3.2 Relačné operátory

V Pig Latin sa transformácie dát dosahujú pomocou relačných operátorov. Tie prijímajú na vstupe relácie a produkujú iné, nové relácie. Existuje množstvo relačných operátorov definovaných v Pig Latinu ale nie všetky vykonávajú dátové transformácie. Z pohľadu modulu pre analýzu dátových tokov je potrebné transformačné operátory určiť a spracovať vo výslednom module. Transformačné operátory sú hlavné časti, ktoré vytvárajú dátové toky, pretože pomocou týchto operátorov sa dáta menia. Vo výslednom grafe, tieto operátory reprezentujú uzly. Operátory sa delia podla účelu do kategórii a nasledujúci zoznamoch tieto kategórie popisuje.

Macro operátory

- DEFINE - Definuje macro, ktoré sa bude používať v kóde.
- IMPORT - Načíta definície makier zo súboru.

Vstupno-výstupné operátory

- LOAD - Načíta dáta zo súboru.
- NATIVE - Spustí skript z externého súboru.
- STORE - Uloží dáta do súboru.
- STREAM - Pošle dáta do externého programu alebo súboru.

Transformačné operátory

- CROSS - Vytvorí kartézsky súčin zo zadaných relácií.
- CUBE - Produkuje kombinácie zo zadanej relácie podľa kritérií.
- DISTINCT - Vyradí duplicitné dáta z relácie.
- FILTER - Filtruje reláciu podľa zadaného kritéria.

- FOREACH - Vytvorí reláciu podľa kritérií.
- GROUP - Zoskupí dáta z relácií.
- JOIN - Spojí dáta z relácií.
- LIMIT - Skráti reláciu na počet prvkov.
- ORDER BY - Zoradí reláciu podľa kritéria.
- RANK - Vráti reláciu s ohodnotením dát podľa kritéria.
- SAMPLE - Vyberie vzorku z relácie.
- SPLIT - Rozdelí reláciu na viacero relácií.
- UNION - Spojí vstupné relácie do jednej.

Skupina `macro` operátorov pracuje s makrami, ktoré predstavujú alternatívny názov pre príkazy. Z pohľadu dátových tokov sú tieto operátory nevýznamné ale predstavujú kľúčovú vlastnosť v jazyku. Táto práca sa nimi však nezaoberá. V tejto práci sa taktiež nezaoberám návrhom a ani implementáciou operátorov `FOREACH` a `NATIVE`, ktoré predstavujú komplexné operátory, pretože obsahujú zložité konštrukcie. Spracovanie ostatných operátorov je obsiahnuté v návrhovej a implementačnej časti.

2.3.3 Aritmetické a iné operátory

Všetky tieto skupiny operátorov majú špeciálny význam, pretože sa môžu vyskytovať v Pig Latin výrazoch. Výrazom sa rozumie postupnosť operandov a operátorov. Výrazy sa môžu vyskytovať na rôznych miestach v relačných operátoroch ale typicky sa vyskytujú za kľúčovým slovom BY. Operandy v týchto výrazoch, ktoré reprezentujú dáta, ovplyvňujú iné dáta len nepriamou formou, a preto reprezentujú nepriame toky.

Pig Latin podporuje základné aritmetické operátory +, -, *, / pre číselné operandy. Tieto operátory vykonávajú funkcie základných matematických operácií. V výslednom grafe, dátové operandy týchto operátorov reprezentujú zdroje pre nepriame dátové toky.

Boolovské (AND, OR), porovnávacie (==, >, <=) a null operátory[6] sú známe tým, že ich výsledok vyjadruje pravdivostnú hodnotu. Pri získavaní dátových tokov sú dátové operandy týchto operátorov zdroje nepriamych tokov.

Cast operátory a konštrukčné operátory sú operátory, ktoré produkujú výstup vo forme dát. Cast operátory dokážu zmeniť dátový typ operandu a konštrukčné operátory dokážu vytvoriť nové objekty. Táto vlastnosť ponúka

[6]Null operátor vráti TRUE hodnotu ak je daný výraz null. Opak je not null operátor, ktorý vráti opračnú pravdivostnú hodnotu.

rôznorodé mechaniky, ktoré sa dajú s týmto operátormi vykonať. Tieto vlastnosti sa však používajú len v operátore FOREACH, a preto nie sú v tejto práci ďalej analyzované.

Medzi ďalšie operátory patrí operátor nepriameho odkazu a operátor nejednoznačnosti. Operátor nepriameho odkazu je dôležitý pri referencovaní nejakého vnútorného objektu. Majme objekt A ktorý má pole B. V takomto prípade je možné sa na B odkázať pomocou A.B a to vďaka operátoru nepriameho odkazu, ktorý je reprezentovaný bodkou.

Operátor nejednoznačnosti je vlastnosť niektorých relačných operátorov, konkrétne JOIN, COGROUP, CROSS, kedy polia vo výstupnej relácií získajú prefix vo forme zdroja a „::“. Dôvod takéhoto správania je podstatný v prípadoch, kedy vstupné relácie majú rovnaké názvy polí. Tento prípad nastáva v ukážke kódu 3.

```
A = LOAD 'data1' as (x);
B = LOAD 'data2' as (x, y);
C = JOIN A by x, B by x;

DESCRIBE C;
-- C: {A::x : bytearray, B::x : bytearray, B::y : bytearray}
```

Zdrojový kód 3: Príklad použitia operátoru nejednoznačnosti

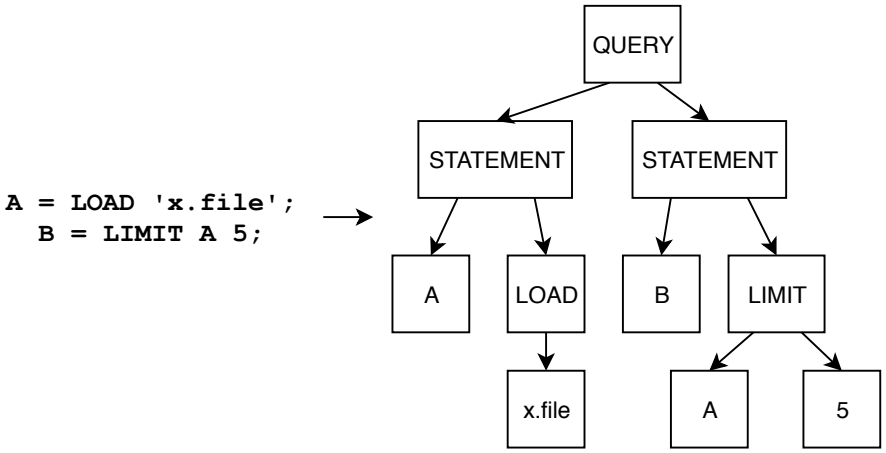
Obe relácie, `A` a `B`, majú pomenované prvé dve polia rovnako a z toho dôvodu je nutné rozlíšiť tieto polia vo výsledku. Zmena názvu polí nastáva aj v prípade, že sú všetky názvy jedinečné.

2.3.4 UDF

Pig poskytuje možnosť definovať vlastné príkazy, ktoré sa nazývajú UDF (Užívateľom definované funkcie). Táto možnosť ponúka užívateľovi definovať vlastné funkcie a vlastné transformácie, ktoré aktuálne platforma Pig neponúka. UDF je možné implementovať v rôznych programovacích jazykoch. Konkrétne sa jedná o Java, Jython, Python, JavaScript, Ruby a Groovy [9]. Pretože sa jedná o užívateľom definované funkcie tvorené v inom programovacom jazyku, táto práca sa analýzou dátových tokov pre UDF nezaoberá.

2.4 Syntaktická analýza

Pig Latin kód prichádza na vstup ako postupnosť znakov. To však nie je najlepšia podoba pre analýzu dátových tokov, pretože z čistého textu nie je jasná štruktúra a sémantika kódu. Pre jednoduchosť práce so zdrojovým kódom



Obr. 2.3: Príklad parsingu Pig Latin kódu do AST

a efektívne spracovanie príkazov je nutné kód previesť do vhodnej podoby. K tomuto výsledku pomôže lexikálna a syntaktická analýza kódu.

Úlohou syntaktickej analýzy nazývanej tiež parsovanie/parsing je analyzovať syntaktický význam textu. Typicky syntaktickú analýzu predchádza lexikálna analýza, ktorá vstupný text rozdelí na elementy. Výstupom syntaktickej analýzy je dátová štruktúra, ktorá sa nazýva abstraktný syntaktický strom alebo AST. AST obsahuje jednotlivé elementy kódu v uzloch a štruktúra stromu reprezentuje skutočnú štruktúru kódu. Všetky uzly majú identifikátor, ktorý dáva význam elementu v AST. Následujúci obrázok 2.3 ukazuje prevod kódu do AST.

Na obrázku 2.3 je znázornený prevod Pig Latin kódu do formy AST. Uzol `QUERY` reprezentuje koreň celého stromu. Tento uzol má potomkov, v tomto prípade dva `QUERY` uzly, ktoré označujú jednotlivé príkazy. Následne potomci týchto príkazov sú jednotlivé elementy a majú svoju štruktúru.

Pre potreby analýzy dátových tokov je AST vhodná dátová štruktúra s ktorou je možné efektívne pracovať, čo pomáha splniť požiadavok NF1. Pre prevod kódu do AST je potrebné mať nástroje, ktoré vykonajú lexikálnu a syntaktickú automaticky. Tieto nástroje sa nazývajú lexer (lexan) a parser. Lexer vykonáva lexikálnu analýzu, kde delí vstupný kód na postupnosť elementov nazývaných tokeny. Parser vykonáva syntaktickú analýzu nad postupnosťou týchto tokenov, z ktorých následne generuje výsledný AST. Tvorba lexeru a parseru však nepredstavuje jednoduchú úlohu. S týmto problémom pomôže zdrojový kód nástroja Pig, ktorý je vydaný pod voľnou licenciou, a teda je ho možné použiť bezplatne.

2.4.1 Zdrojový kód Apache Pig

Zdrojový kód nástroja Pig je možné nájsť voľne dostupný na GitHub repozitáry Apache Pig [10]. Ako už bolo spomenuté, tento kód je vydaný pod voľnou licenciou, takže je môžné ho používať a modifikovať. Nástroj Pig je naprogramovaný v jazyku Java, čo dáva možnosť využiť časti kódu vo výslednom module. Pre potreby vyvíjaného modulu je vhodné použiť existujúci parser a lexer pre jazyk Pig Latin, ktoré vykonajú syntaktickú analýzu. Existujúci parser a lexer však nie sú v Pig repozitáry reprezentované pomocou Java tried ale len popisom gramatiky jazyka v syntaxy nástroja ANTLR[7]. Jednou z možností ako tieto gramatiky transformovať do potrebných Java tried je zostavenie nástroja Pig zo zdrojového kódu. Táto práca sa týmto problémom nezaoberá a čitateľ, ktorý by si chcel nástroj Pig zostaviť sám je odkázaný na návod na Github repozitáry [10]. Pre potreby tejto práce sú triedy pre syntaktickú analýzu priložené v zdrojovom kóde modulu.

Tento prístup ma dostáva do pozície, kedy je nutné zhodnotiť prínosy a riziká, ktoré môžu nastať a zhodnotiť, či je použitie lexeru a parseru vhodné. Tieto prínosy a riziká sú:

Prínosy

- Existujúca implementácia.
- Novšia verzia prinesie aktualizovaný parser/lexer.
- Zdrojový kód priamo od tvorcov projektu.
- Jednoduché prepojenie tried s modulom (Java modul a Java parser/lexer)

Riziká

- Nevhodná reprezentácia AST.

Značná prevaha prínosov nad rizikami udáva jasný smer a to využiť tento zdrojový kód. Nevhodná reprezentácia AST je jediné riziko, ktoré môže nastať. Toto riziko sa dá potlačiť transformáciou získaného AST do AST, ktoré má vhodnú štruktúru pre analýzu dátových tokov.

Pri preskúmaní gramatiky som zistil, že toto riziko nastáva u niektorých častí stromu, ktoré majú nevhodnú reprezentáciu. Tieto časti sú pri spracovaní stromu transformované do vhodnej reprezentácie.

[7] Nástroj na generovanie parseru a lexeru z gramatiky.

2.5 Sémantická analýza

Sémantická analýza je proces, ktorý nasleduje po syntaktickej analýze. Jej úlohou je odhaliť sémantické chyby v kóde, ktorý je reprezentovaný pomocou AST. Presnejšie, jej záujmom je:

- Kontrola typov.
- Kontrola platnosti objektov.

Cieľom kontroly typov je odhaliť chyby, kedy príkazy obsahujú objekty nesprávneho typu. Jednoduchý príklad je použitie objektu tuple namiesto relácie ako vstupný parameter pre operátor. Táto chyba je chybou, pretože operátory pracujú len s reláciami. Kontrola typov je problém, ktorý sa dá vyriešiť pomocou získania skutočnej reprezentácie daného objektu, pretože je potom možné skontrolovať dátový typ s pravidlami jazyka. Takto pomocou uplatnenia pravidiel jazyka, ktoré dodáva sémantika jazyka Pig Latin, je možné vylúčiť sémantické chyby a splniť požiadavok F4. Získaním skutočnej reprezentácie objektu sa zaoberá nasledujúca časť.

2.5.1 Resolving

Resolving je proces, ktorého úlohou je získať referencujúci objekt pre uzol. Následujúci kód 4 ukazuje jeho použitie v príklade.

```
A = LOAD 'student' AS (name, age, gpa);
B = FILTER A BY name is not null;
```

Zdrojový kód 4: Príklad použitia resolvingu

Zdrojový kód 4 vyššie obsahuje dvakrát identifikátor `A`. Cieľom resolvingu je dosiahnuť, aby obe uzly v AST, ktoré reprezentujú identifikátor `A` odkazovali na rovnaký objekt.

Resolving má vyznám len pre uzly operátorov, funkcií a identifikátorov. V prípade resolvingu operátorov a funkcií je cieľom získať objekt, ktorý reprezentuje výstup z daného príkazu. Pre identifikátory sa získava objekt, ktorý referencuje daný identifikátor.

Pomocou resolvingu identifikátorov sa odhalia nielen nesprávame typy ale aj nedefinované objekty a prístupy k objektu, ktorý nepatrí do bloku. Pre riešenie týchto problémov však samotný resolving nestačí. Pre funkciu resolvingu a riešenie týchto problémov je potrebné pre všetky uzly určiť blok, do ktorého patria. Ďalšia časť sa zaoberá blokmi a mennými priestormi s použitím resolvingu.

2.5.2 Resolving, bloky a menné priestory

Ako každý moderný jazyk, rovnako aj Pig Latin prináša mechanizmus definovania objektov. Každý z týchto objektov má nejaký rozsah platnosti a patrí do nejakého bloku. V prípade, že objekt neplatí v danom bloku a odkazujeme sa neho v kóde v tomto bloku, môže byť tento program syntakticky správny ale sémanticky nie je. Takéto chyby sa dajú odhaliť pomocou resolvingu objektov v blokoch.

V Pig Latin rozlišujeme dva typy blokov, globálny a vnútorný blok. Globálny blok zahŕňa všetky globálne dáta skriptu, na ktoré je možne sa odkazovať pomocou identifikátorov. V niektorých operátoroch existuje aj vnútorný blok, kde je môžeme definovať objekty, ktoré sú len dočasné. Tieto objekty nie sú v nadradenom bloku viditeľné.

Ďalšia vlastnosť v Pig Latin sú menné priestory. Menné priestory vysvetľuje kód a komentár 5.

```
A = LOAD 'student' AS (name, age, gpa);
B = FILTER A BY name is not null;
```

Zdrojový kód 5: Príklad použitia menných priestorov

V ukážke kódu 5 sa na položku `name` relácie `A` v príkaze `FILTER` neodkazuje pomocou `A.name` ale len `name`, pretože príkaz nadobúda menný priestor `A`, podľa vstupnej relácie. Menné priestory nadobúdajú platnosť vo všetkých operátoroch, kde sa používajú dáta zo vstupnej relácie. Preto pre všetky polia, ktoré sa vzťahujú na vstupnú reláciu sa nemusia popisovať plne kvalifikovaným menom ale len častou. Pre všetky ostatné polia, ktoré sa nevzťahujú na vstupnú reláciu je potrebné dodať úplne meno objektu.

V Pig Latin je možné predefinovať reláciu inou reláciou s rovnakým menom. V takomto prípade stratí pôvodná relácia platnosť a nová ju nahradí.

Kapitola 3

Návrh

Táto kapitola sa zaoberá návrhom výsledného modulu. Pri návrhu sa využívajú poznatky z analytickej časti.

Výsledný modul je rozdelený na tri menšie moduly. Jedná sa o moduly:

- Resolver
- Model
- Generator

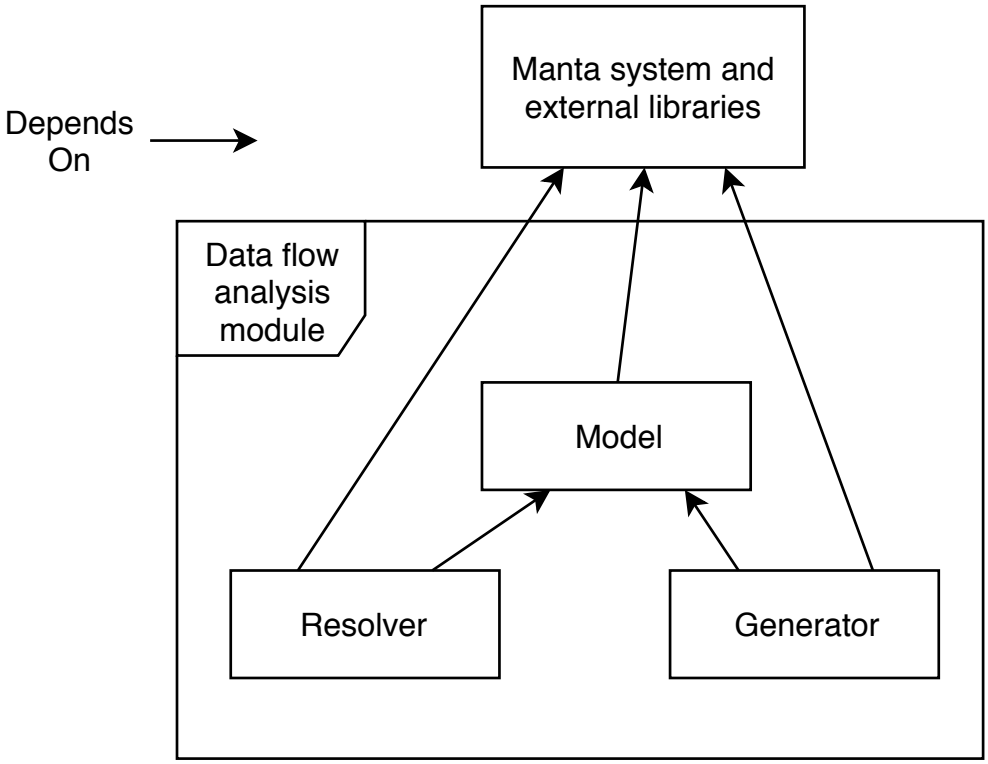
Na obrázku 3.1 je možné vidieť závislosti a komunikáciu medzi modulmi. Modul s názvom `Manta system and extern libraries` nepredstavuje v skutočnosti jeden modul ale skupinu Manta modulov. Tieto moduly obsahujú definície a implementácie generických Manta tried pre analýzu dátových tokov a iné nápomocné knižnice. Jedným z nefunkčných požiadavkov, konkrétne NF2, je integrácia modulu do Manta systému. Preto modul vytvorený v tejto práci silne závisí na týchto Manta triedach.

Skupina balíčkov a tried, ktoré patria do modulu s názvom `Model` obsahujú definície Java rozhraní. Úlohou týchto Java rozhraní je poskytnúť rozhranie pre komunikáciu medzi modulom `Resolver` a `Generator`. Tento modul neobsahuje žiadnu funkčnú logiku ale slúži len ako komunikačný prostriedok, a preto nebude v tejto práci ďalej rozoberaný.

V následujúcich častiach kapitoly sú uvedené návrhové detaily modulov `Resolver` a `Generator`.

3.1 Resolver

Triedy z balíčku `Resolver` majú na starosť spracovať kód do formy AST a dodatočné operácie nad týmto AST. Konkrétne triedy z toho balíčku vykonávajú lexikálnu, syntaktickú a sémantickú analýzu. Ďalšie významné úlohy týchto tried je resolving a transformácia stromu do vhodnej podoby.



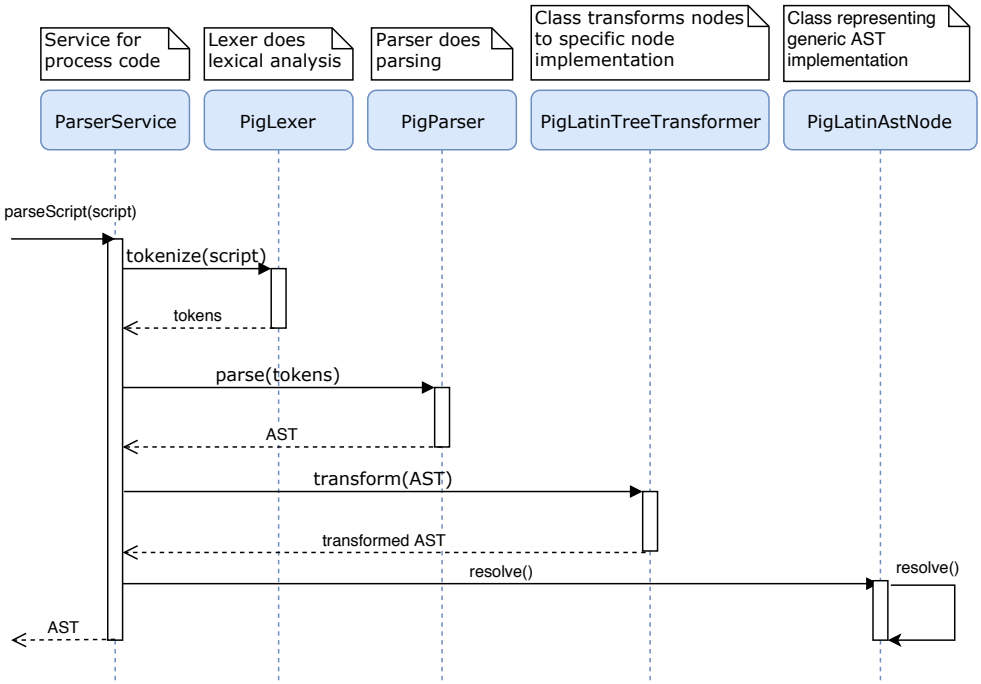
Obr. 3.1: Diagram Maven modulov

Hlavnou častou tohoto modulu je služba, ktorá dokáže spracovať vstup vo viacerých formách (súbor a reťazec), čo je jeden z funkčných požiadavkov F3. Jej priebeh je znázornený v sekvenčnom diagrame 3.2.

Pre potreby lexikálnej a syntaktickej analýzy sa používajú už existujúce implementácie tried `PigLexer` a `PigParser`, ktoré sú získané zo zdrojového kódu Apache Pig. Trieda `PigLexer` rozdeľuje Pig Latin skript získaný zo vstupu na elementy, ktoré následne putujú do triedy `PigParser` a tá ich spracuje. Tieto nástroje zaručia výstup vo forme AST, ktorý následne putuje do triedy `PigLatinTreeTransformer`, kde prebieha transformácia AST. Triedu `PigLatinTreeTransformer` popisuje následujúca sekcia.

3.1.1 Transformácia AST

Transformáciou AST získam AST, ktorý je špeciálne upravený pre potreby dátových tokov. Na vstupe do tejto triedy prichádza AST v podobe, kedy všetky uzly popisuje generická trieda `PigLatinAstNode`. Úlohou triedy `PigLatinTree Transformer` je nahradiť generické uzly v AST, ktoré reprezentujú identifikátor, relačný operátor, príkaz a dátové typy za špecificke uzly, ktorý reprezentuje ich trieda. Každý špecificky uzol je reprezentovaný vlastnou triedou s názvom `Ast*`, kde `*` reprezentuje vyznám daného uzlu. Nasledujúci diagram tried 3.3 popisuje



Obr. 3.2: Sekvenčný diagram služby modulu Resolver

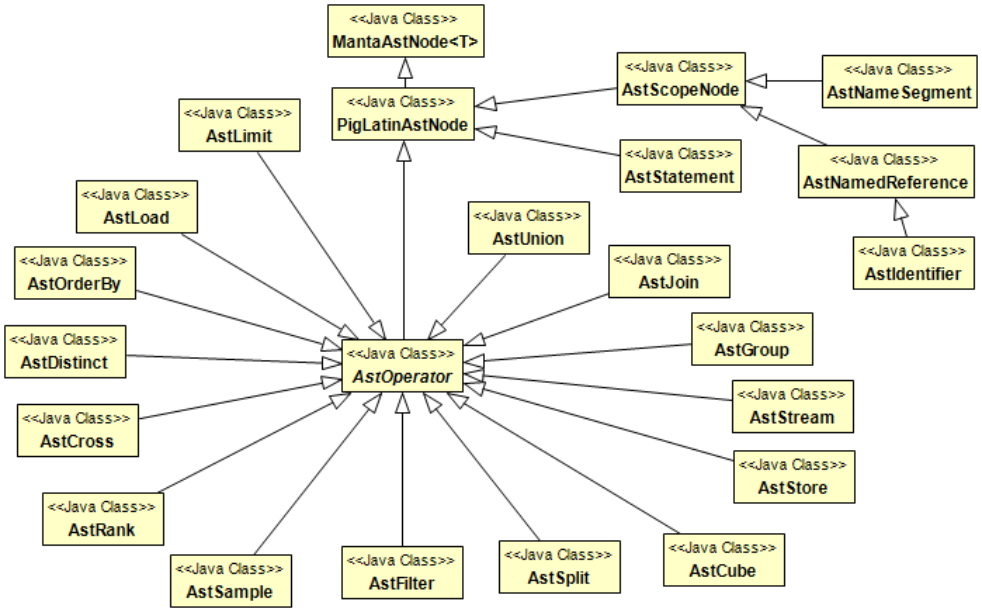
hierarchiu týchto tried. V diagrame sú z dôvodu prehľadnosti vynechané menej dôležité triedy, ktoré reprezentujú komplexné dátové typy a schému.

Ako je možné vidieť z diagramu 3.2, všetky vytvorené uzly v AST začínajú triedou `MantaAstNode`. Táto trieda pochádza z Manta systému a reprezentuje základný uzol AST. Medzi jej vlastnosti patria základné operácie s uzlom, ktoré sú napr. získavanie potomkov a rodiča, pridávanie a mazanie potomkov, zmena rodiča, zmena identifikátoru atď. Táto trieda taktiež poskytuje možnosť získavania uzlov stromu pomocou *XPath*.

XPath je dotazovací jazyk, ktorý bol pôvodne vytvorený pre získavanie uzlov z XML dokumentu. V tejto práci sa však používa pre získavanie uzlov v AST. Funguje na princípe volania metódy nad uzlom stromu s parametrom, ktorý udáva čo chcem z daného uzlu alebo stromu získať. Tento jazyk podporuje veľké množstvo možností ako je možné získavať uzly napr. podľa určenia potomka, podľa atribútu uzlov, podľa identifikátoru atď.

Všetky triedy, ktoré reprezentujú uzly v Pig Latin AST dedia od triedy `PigLatinAstNode`. táto tried reprezentuje základný uzol, ktorý popisuje všeobecnú logiku uzlu AST. Od nej sa ďalej rozlišujú tried pre operátor, príkaz a uzly reprezentujúce identifikátor.

Všetky uzly operátorov dedia vlastnosti od uzlu `AstOperator`. Táto trieda poskytuje možnosti pre resolving operátoru, ktorá vráti reprezentáciu výstupnej relácie z operátoru. Ďalšou úlohou tejto triedy je poskytnúť spoločné rozhranie



Obr. 3.3: Diagram tried pre uzly AST

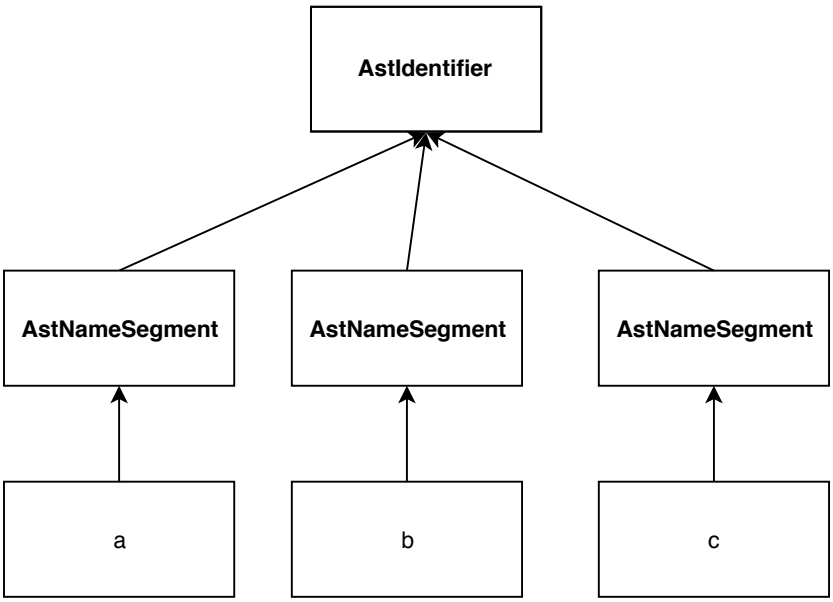
pre operátory.

`AstCopeNode` reprezentuje rozhranie, v ktorom je dôležité uchovávať informácie o bloku. To sa využíva pre dve triedy `AstNameSegment` a `AstIdentifier`. Obe tieto triedy maju úlohou v reprezentácií identifikátorov a kvalifikovaných mien. Trieda `AstNamedReference` popisuje správanie identifikátorov pre Manta systém, ktoré je dôležité pri resolvingu. Reprezentáciou identifikátorov sa zaoberá následujúci odstavec.

Reprezentácia identifikátorov. V Pig Latin kóde je možné reprezentovať identifikátory, ktoré sa skladajú z viacerých mien. Táto vlastnosť sa v Pig Latin volá operátor nepriameho odkazu a bola popísaná v analytickej časti. Príkladom takéhoto identifikátoru je `a.b.c`. Tento identifikátor sa skladá z 3 častí, ktoré sa nazývajú segmenty. Jeho stromovú reprezentáciu predstavuje následujúci obrázok 3.3. Rovnaká reprezentácia sa používa v tejto práci pre reprezentovanie uzlov v transformovanom AST.

Princíp transformácie pozostáva v triede `Transformer`, ktorá volá transformačnú metódu na všetky uzly stromu. Táto triedy sa správa ako stavový automat, ktorý sa rozhoduje ako transformovať uzol podľa jeho významu. Niektoré uzly sú pre analýzu dátových tokov nepodstatné, a preto sa v tejto transformácií nemenia. Ich triedou ostatne generický uzol `PigLatinAstNode`.

Z gramatiky, ktorú je možné nájsť v Pig Latin repozitáry je vidieť, že niektoré uzly majú nevhodnú reprezentáciu, ktorá je aj jedným z rizík použitia zdrojového kódu Apache Pig. Z toho dôvodu u niektorých uzlov prebieha zmena



Obr. 3.4: Príklad segmentov mena a.b.c

stromu. To môže zahŕňať tvorbu nových uzlov, zmenu potomkov alebo mazanie nepotrebných uzlov.

3.1.2 Resolving

Resolving je úloha, ktorá spočíva získaní referencujúceho objektu pre uzol. Pre správnu funkciu resolvingu identifikátoru je potrebné vedieť pre každý identifikátor a segment, v ktorom bloku sa nachádzajú. Bloky reprezentuje jedna trieda, ktorá sa volá `ResScope`. Objekt tejto triedy reprezentujú inštanciu konkrétneho bloku, ktorý má rozsah platnosti.

Vlastnosť každého bloku je tá, že objekt má prístup k aktuálnemu bloku ale aj jeho nadradeným blokom. Takto sa bloky rekurzívne spávajú až po najvyšší globálny blok, ktorý je konečný. Táto vlastnosť sa dosiahne pomocou zreťazenia blokov do jednosmerného spojového zoznamu. V prípade, že sa nájde novovytvorený identifikátor, vytvorí sa jeho inštancia a pridá sa do bloku. Cieľom resolvingu je nájsť referencujúci objekt vo svojom bloku alebo jeho nadradených.

3.2 Generator

Úlohou modulu `Generator` je generovanie grafu dátových tokov z AST, ktorý získa z modulu `Resolver`. Generovanie sa deje v triede `PigLatinAstVisitor`, ktorá používa návrhový vzor Visitor. V tejto triede sú implementované visitor metódy pre spracovanie všetkých dôležitých uzlov, ktoré sú znázornené v

diagrame tried 3.3. Táto trieda je závislá na pomocnej triede `PigLatinGraphHelper`, ktorá obsahuje metódy pre generovanie uzlov a hrán grafu dátových tokov.

Ďalšou úlohou v module `Generator` je vytvorenie úloh pre analýzu dátových tokov. Táto trieda reprezentuje úlohu spracovania vstupu a generovania výstupu nad nejakými zdrojmi, ktoré predstavujú napr. priečinok kde sa nachádza script, odkiaľ sa berú vstupné súbory atď.

Kapitola 4

Implementácia

Táto kapitola sa zaoberá implementáciou výsledného modulu. Z dôvodu veľkého rozsahu implementovaného kódu sú v tejto kapitole spracované len jeho dôležité časti. Prvá podkapitola slúži ako popis základných tried, ktoré sa používajú behom celého projektu. Dôvod vymedzenia týchto tried do samostatnej časti je ten, že všetky triedy na sebe zavia, a takto môžu byť zhrnuté jednej časti. Po tejto časti nasleduje dokumentácia modulov `Resolver` a `Generator`.

4.1 Základné triedy

4.1.1 PigLatinResObject

Táto trieda predstavuje konkrétny referencujúci objekt. Tento objekt je výsledkom resolvingu, teda volaním metódy `resolve()` na AST uzly. Resolvovany objekt predstavuje konkrétnu inštanciu identifikátoru, ktorý reprezentuje dáta. Pre kompatibilitu s Manta systémom, trieda implementuje Manta rozhranie `IResEntity`. Toto rozhranie popisuje množstvo metód, ktoré je potrebne pre správnu funkciu implementovať. Pre potrebu tvorby resolvovaných objektov však vystačím len s niektorými metódami. Následujúca tabuľka 4.1 uvádza atribúty s typom, ktoré obsahuje trieda `PigLatinResObject`.

Tabuľka 4.1: Atribúty triedy PigLatinResObject

`Atribút`	`Atribút`
EntityName entityName	Meno entity
Set<EntityProps> props	Vlastnosti entity
IMantaAstNode sourceNode	Zdrojový uzol, odkiaľ entita vznikla
DefinitionSourceType	Definícia zdroja
List<IResEntity> children	Zoznam odkazov na potomkov
IResEntity parentEntity	Odkaz na rodiča

V tejto triede meno entity označuje názov resolvovaného objektu. Vlastnosti entity sú reprezentované množinou vlastností. Z tejto množiny je ľahko rozoznať, čo daná entita reprezentuje. Zdrojový uzol dáva možnosť jednoduchého odkázania sa na rodiča, z ktorého vznikol. Táto možnosť je potrebná pri generovaní dátových tokov, kde je potrebné nájsť rodiča. Ďalší atribút predstavuje definíciu zdroja. Tento atribút je získaný z enumerátoru `DefinitionSource` a udáva, odkiaľ daný uzol vznikol. V tomto modul sa používa len jeden zdroj a to zdroj, ktorý reprezentuje skript - `DefinitionSource.SCRIPT`. Objekt `PigLatinResObject` má podobu uzlu v stromovej štruktúre, pretože každý identifikátor má svoju štruktúru (schému). K tejto reprezentácií slúži zoznam potomkov a odkaz na rodiča.

4.1.2 PigLatinEntityName

`EntityName` je abstraktná trieda z Manta systému, ktorá reprezentuje meno pre resolvovanú entitu. V tomto module ju rozširuje trieda `PigLatinEntity Name`, ktorá predstavuje jej konkrétnu implementáciu. Táto trieda obsahuje konštruktor prijímajúci parameter typu String. Týmto parametrom sa dodáva meno entity.

4.1.3 EntityProps

`EntityProps` je enumerátor, ktorým sa popisujú vlastnosti pre resolvovanú entitu. Trieda `EntityProps` obsahuje veľký výčet vlastností, ktoré môže nadobúdať ale v tomto module si vystačím len s pár vlastnosťami. V následujúcom zozname sú vlastnosti, ktoré sa používajú v module aj s ich vysvetlením.

- PRIMITIVE_TYPE - Reprezentuje jednoduchý dátový typ.
- TUPLE - Reprezentuje typ tuple.
- MAP - Reprezentuje typ map.
- BAG - Reprezentuje typ bag.

Použitie týchto vlastností v resolvovaných entitách je následovné. Entita, ktorá je komplexného dátového typu obsahuje jedinú vlasnosť, ktorá označuje jej konkrétny typ (tuple, map alebo bag). V prípade jednoduchého dátového typu sú to dve vlastnosti, kedy jedna z nich je `PRIMITIVE_TYPE` a druhá je konkrétny jednoduchý typ. Konkrétne jednoduché typy nie sú v zozname vypísané, pretože neurčujú aktuálne nič dôležité. Tieto typy sú taktiež obsiahnuté v enumerátore a sú pomenované je podľa názvu typu. Dôvod takejto reprezentácie je jednoduchosť práce s typmi, pretože pre všetky jednoduché typy je logika spracovania rovnaká a pre všetky komplexné je iná. Teda, v takejto reprezentácií stačí spracovávať len tieto štyri vlastnosti, ktoré sú zmienené vyššie.

4.1.4 ResScope

Trieda `ResScope` reprezentuje blok. Každá inštancia tejto triedy obsahuje názov bloku, odkaz na nadradený blok, zoznam objektov, ktoré existujú v danom bloku, názov a objekt menného priestoru. Úloha tejto triedy je významná pre resolving, pretože pri resolvingu sa vyhľadávajú objekty v blokoch. Entity, ktoré existujú v danom bloku sú reprezentované tabuľkou, kde kľúč je meno reprezentované triedou `PigLatinEntityName` a hodnota je entita `ResEntity`. Metódy ako nájsť entitu nad blokom sú `resolveObjectByName` a `resolveObjectInNamespace`. Obe metódy prijímajú objekt, ktorý reprezentuje meno entity a množinu vlastností, z ktorej musí hľadaná entita nejakú nadobúdať. Prvá metóda funguje na princípe rekurzívneho hľadania entity v aktuálnom bloku a postupuje až po globálny blok. U druhej metódy prebieha hľadanie len v aktuálnom bloku, ktorý musí mať menný priestor pre blok a v prípade existencie menného priestoru skúsi nájsť entitu v ňom. Táto trieda obsahuje taktiež metódy pre vkladanie entít do bloku.

Ďalšiu dôležitou častou tejto triedy je vnorená trieda `NewUnnamedObject Index`. Úlohou tejto triedy je získať meno pre nepomenované objekty v schéme. To je dôležité, pretože objekty v schéme nemusia obsahovať meno a tak je potrebné nejaké meno dodať. Princíp ako sa pomenúvajú objekty schémy bez mena som zistil z postupného debugovania programov platformy Pig. V podstate ide o pomenovanie nepomenovaného objektu v schéme menom, ktoré je `val` pre jednoduché typy, `tuple` pre tuple, `bag` pre bag a `map` pre map. Tieto mená sú nasledované reťazcom `_*`, kde číslo predstavuje index nepomenovaného prvku začínajúci od nuly. Takže tretí nepomenovaný tuple objekt bude mať meno `tuple_2`. Podotýkam, že tieto indexy sú pre každý typ odlišné teda nezávisia na sebe.

4.1.5 PigLatinContextState

Inštancia triedy predstavuje objekt, ktorý spája rôzne objekty do jedného. Tieto objekty sú predané konštruktorom. Ide o inštancie tried `MantaAstNavigator`, `PigLatinResolverEntitiesFactory`, `PigLatinReferenceResolver`, `AbsractPigLatinParser` a `PigLatinTreeAdaptor`. Tieto triedy slúžia pri parsingu, transformácií AST a resolvingu. Účelom tejto triedy je skrátenie predávania parametrov, pretože tieto triedy objekty sa používajú v mnoho metódach.

4.1.6 PigLatinResolverEntitiesFactory

`PigLatinResolverEntitiesFactory` je rozhranie, ktoré popisuje metódy pre tvorbu resolvovaných objektov. Rozhranie rozširuje generické Manta rozhranie `ResolverEntitiesFactory` a pridáva definície dôležitých metód. Jedná sa o dve metódy pre duplikáciu objektu a jednu pre získanie reprezentácie vytváraných objektov. V module `Resolver` existuje implementácia tohoto rozhrania s

názvom `PigLatinResolverEntitiesFactoryImpl`. Táto trieda neimplementuje všetky metódy, ktoré rozhranie dodáva ale len tri metódy. Dve z týchto metód sú v kóde 6.

```
IResObject createObject(EntityName entityName,
        Set<EntityProps> properties,
        IResDataType dataType,
        DefinitionSourceType definitionSourceType,
        IMantaAstNode definingAstNode);
IResEntity duplicateObject(IResEntity entity,
        IMantaAstNode node);
```

Zdrojový kód 6: Metódy triedy PigLatinResolverEntitiesFactory

Prvá metóda vytvára resolvovaný objekt. Jej parametre sú názov objektu, množina vlastností, dátový typ, zdrojový typ a uzol, v ktorom vzniká objekt.

V metóde duplikácie objektu na prvom parametru prijíma entitu pre duplikovanie a druhý parameter reprezentuje uzol, ktorý sa zvolí ako zdroj pre všetky novovytvorené entity. Telo tejto metódy pozostáva z rekurzívneho duplikovania objektu v prípade, že je objekt zloženého dátového typu, inak ide len o volanie konštruktoru s aktuálnymi dátovými položkami. Trieda obsahuje taktiež implementáciu duplikovania kolekcie objektov, ktoré duplikuje pomocou predchádzajúcej metódy a vráti v zoradenom zozname.

4.2 Resolver

Štruktúra modulu Resolver pozostáva z niekoľkých balíčkov. Sú to balíčky:

- `resolver/ast` - Obsahuje implementácie uzlov. (neobsahuje implementácie operátorov)
- `resolver/ast/operator` - Obsahuje implementácie uzlov, ktoré reprezentujú operátory.
- `resolver/data` - Obsahuje implementáciu resolvovaných objektov.
- `resolver/parser` - Obsahuje lexer, parser a ich pomocné triedy.
- `resolver/scope` - Obsahuje triedy slúžiace pre správu blokov a menných priestorov.
- `resolver` - Obsahuje riadiace štruktúry modulu a ich pomocné triedy.

Telo spracovania kódu do podoby AST obsahuje trieda `ParserServiceImpl`. `ParserServiceImpl` je existujúca implementácia služby `ParserService`, ktorá prijíma dáta v nejakej forme a produkuje AST. Jej metódy vyhodnocovania sú znázornené v kóde 7.

```
IPigLatinAstNode parseStringScript(String script,
        PigLatinResolverEntitiesFactory factory,
        ParserServiceParams params, String inputName);
IPigLatinAstNode parseFileScript(File file,
        PigLatinResolverEntitiesFactory factory,
        ParserServiceParams params, String encoding);
```

Zdrojový kód 7: Metódy spracovania vstupu rozhrania ParserService

V prvom prípade sa dodáva skript vo forme reťazca v prvom argumente a meno skriptu ako posledný argument. V druhom sa jedná o súbor a jeho kódovanie. Ďalšie parametere nutné k spracovanie vstupu sú `ParserServiceParams` a `PigLatinResolverEntitiesFactory`. V prípade parametru `ParserService Params`, ide o dodatočné parametre pre parsovaciu službu. Existencia tejto triedy nemá v aktuálnej chvíli žiadnu úlohu ale v budúcnosti môže obsahovať napr. debugovacie alebo konfiguračné metódy. Z toho dôvodu je táto trieda zahrnutá v module už teraz.

Logika služby pozostáva z princípu popísaného v návrhovej časti 3. V skratke ide o lexikálnu a syntaktickú analýzu kódu. Výstupom z týchto operácií je AST, nad ktorým následne prebieha transformácia, ktorej implementácia je popísaná v ďalšej časti.

4.2.1 Transfromácia AST

Transformácia AST prebieha v inštancii triedy `PigLatinTreeTransformer`. Princíp tejto transformácie pozostáva vo volaní metódy `transformTree`. Táto metóda prijíma ako parameter koreň stromu a odkaz na globálny blok v prvom volaní. Telo tejto metódy predstavuje rekurzívne spracovanie uzlu a následne jeho potomkov. Z týchto uzlov sa generujú nové uzly. Metóda pozostáva z konečného automatu, ktorý volá iné metódy podľa identifikátoru uzlu.

Ďalšie často používané operácie v tejto triede sú:

- Získanie pozície uzlu v zdrojovom kóde.
- XPath.
- Výpis stromu do reťazca.

XPath. XPath predstavuje jazyk pomocou, ktorého je možné prechádzať AST. Ako už bolo spomenuté v návrhu, tieto operácie sú implementované v triede `MantaAstNode`, čiže sa dajú používať nad každým AST uzlom. Tieto operácie používajú vstupný parameter typu String, v ktorom užívateľ predá XPath požiadavok podľa toho čo chce zo stromu získať. V tejto práci však nie je vysvetlenie tohoto jazyka. Pretože tieto metódy prijímajú typ String, tak je nutne tieto argumenty predávať „natvrdo“. V budúcnosti sa pre parser môže zmeniť názov identifikátoru uzlu a to bude predstavovať problém. Z toho dôvodu sú v tejto prací implementované metódy, ktoré prijímajú parametere vo forme čísel. Tieto čísla reprezentujú konštanty pre označenie identifikátoru uzlu a sú získané z tried `PigParser` a `PigLexer`. Tieto nové metódy sú implementované v triede `PigLatinAstNode`.

4.2.2 Resolving

Ako už bolo spomenuté, resolving je úloha pre získanie skutočnej reprezentácie určitých uzlov AST. Resolving prebieha pomocou volania metódy `resolve()`, ktorá vráti skutočnú reprezentáciu uzlu. Táto metóda je definovaná v rozhraní `IPigLatinAstNode`, čiže je možné ju volať nad každým uzlom. Objekt tejto skutočnej reprezentácie je v podobe rozhrania `IResEntity`, aby sa zaručila spätná kompatibilita s Manta systémom.

Resolving prebieha pomocou tried, ktoré už existujú v Manta systéme. Tieto triedy však predstavujú často len abstraktné rozhrania alebo logiku vyhodnocovania, a preto je nutné tieto triedy implementovať alebo rozšíriť.

V nasledujúcom texte je popis priebehu resolvingu pre identifikátor a následne pokračuje vysvetlenie resolvingu pre operátor. Ako už bolo spomenuté, u nie všetky uzlov ma volaní tejto metódy zmysel, pretože niektoré objekty nereprezentujú dáta. Resolving je užitočný pre uzly operátorov, schémy, identifikátorov, segmentov mena, príkazu a funkcie. U všetkých týchto uzlov sa však resolving líši.

Všetko sa záčina volaním metódy `resolve()` nad uzlom identifikátoru. Tento uzol, ako každý iný, si udržiava referenciu na `PigLatinContextState`, z ktorého získa objekt `PigLatinReferenceResolver` pre resolving. Nad týmto objektom sa zavolá metóda pre resolving, ktorej argument predstavuje referenciu na samého seba. Funkcia tejto triedy je resolvovanie objektov. Ten sa deje pomocou resolvingu všetkých segmentov mena, kedy objekt reprezentujúci posledný segment kvalifikovaného mena reprezentuje skutočnú reprezentáciu identifikátoru.

Resolving segmentu mena sa vykonáva následovne. Pri volaní resolvingu segmentu prijíma metóda objekty, ktoré sa získali z resolvingu predchádzajúcich segmentov. To znamená, že je nutné získať reprezentáciu objektu, ktorý reprezentuje predchádzajúci segment. Nad týmto objektom sa vyhodnocuje aktuálny segment a hľadá sa v nom objekt, ktorý reprezentuje tento segment. Takto vyhodnocujú segmenty až po posledný, ktorý sa vráti ako výsledok resolvingu

pre identifikátor. V prípade resolvingu prvého segmentu mena identifikátoru sa jedna o hľadanie objektu bloku. K tomuto prostriedku slúži trieda `ResScope`, ktorá reprezentuje blok. Každý identifikátor obsahuje nejakú inštanciu tejto triedy. V prípade, že sa nenájde skutočný objekt pre najaký segment mena, tak resolving skončí s chybou a to znamená, že kód je sémanticky nesprávny.

Resolving operátorov sa vykonáva pre každý operátor zvlásť. Získanie skutočnej reprezentácie výstupu operátoru sa deje pomocou volania metódy `resolve()` nad uzlom operátoru, rovnako ako u identifikátoru, a vracia objekt typu `PigLatinResObject`. Tento objekt reprezentuje štruktúru výstupu a jej koreňový objekt reprezentuje nepomenovaný objekt bag. tento objekt nemá určité meno, pretože výstup operátoru neobsahuje meno. Tento výstup sa typicky priraďuje do relácie s menom, kde meno nadobúda. Avšak v tento objekt obsahuje potomkov, ktoré reprezentujú štruktúru a obsahuj aj mená. Táto štruktúra vznik na základe vstupnej relácie a vo väčšine prípadov sa z nej kopíruje. Existujú taktiež operátory, ktoré túto štruktúru jemne menia. Všetky tieto zmeny sú jasne viditeľné v implementovanom kóde, ktorý je obsiahnutý v priloženom disku.

4.3 Generator

Táto podkapitola sa zaoberá dokumentáciou modulu `Generator`. Celú jeho implementovanú časť vytvárajú 4 triedy, ktoré sú:

- `PigLatinGraphHelper`
- `PigLatinDataFlowVisitor`
- `PigLatinDataFlowVisitorFactory`
- `PigLatinDataFlowTask`

Na začiatok uvádzam sekciu, ktorá priblíží čitateľovi reprezentáciu grafu dátových tokov, ktorá sa používa v tomto module. Po tejto sekcií nasledujú časti, ktoré popisujú implementované triedy.

4.3.1 Graf dátových tokov

Výsledný graf dátových tokov je reprezentovaný triedou `Graph`, uzlami typu `Node` a hranami `Edge`. Všetky tieto triedy pochádzajú z Manta systému a táto práca neuvádza ich skutočnú reprezentáciu. Avšak pre pochopenie funkcionality tohoto grafu, je potrebné popísať vlastnosti a atribúty, ktoré tieto triedy obsahujú. Týmto sa zaoberá práve táto kapitola, ktorá popisuje funkcionalitu grafu s jeho vlastnosťami.

Na začiatok prichádza trieda `Graph`, ktorá reprezentuje graf dátových tokov. Kód 8 popisuje túto triedu. Táto trieda obsahuje množinu všetkých hrán a

uzlov patriacich grafu. Jej metódy, ktoré sú významné pre tento modul sú metódy na pridanie uzlu a hrany. Parametry metódy pridania uzlu predstavujú meno vytváraného uzlu, typ uzlu, odkaz na rodiča uzlu a zdroj odkiaľ pochádza. U hrany sú to zdrojový a cieľový uzol, typ hrany a zdroj.

```
Set<Node> nodeSet;
Set<Edge> edgeSet;
Node addNode(String name, String type,
                    Node parent, Resource resource);
Edge addEdge(Node source, Node target,
                    Edge.Type type, Resource resource);
```

Zdrojový kód 8: Trieda Graph

Následujúce kódy 9 a 10 reprezentujú uzol a hranu.

```
String name;
Node parent;
Resource resource;
String type;
Set<Edge> outgoingEdges;
Set<Edge> incomingEdges;
Set<Node> children;
```

Zdrojový kód 9: Trieda Node

```
Node source;
Node target;
Edge.Type type;
Resource resource;
```

Zdrojový kód 10: Trieda Edge

V prípade oboch tried ide o klasickú implementáciu grafu, ktorá je doplnená o ďalšie vlastnosti. Každý uzol má stromovú štruktúru, kedy uzol obsahuje odkaz na svojho rodiča a množinu potomkov. Tento rodič reprezentuje nadradený uzol alebo uzol, z ktorého vznikol jeho potomok. Takáto funkcionalita

je dôležitá v prípade uzlov, ktoré reprezentujú komplexný objekt, napr. dátový typ tuple, operátor alebo funkciu. Všetky tieto komplexné objekty, vo väčšine prípadoch, obsahujú nejakých potomkov (jednoduché alebo komplexné objekty), ktoré je potrebné rovnakou štruktúrou reprezentovať v grafe. Na to slúži práve stromová štruktúra uzlu. Stromová reprezentácia uzlov v grafe popisuje aj hierarchiu uzlov. Príkladom môže byť to, že uzol, ktorý reprezentuje spracovávaný skript bude rodič pre všetky príkazy, identifikátory a ďalšie dôležité uzly, ktoré sa vyskytujú v globálnom bloku. Následne potomci operátorov sú výstupné uzly z operátoru. Rovnako súbor, ktorý je reprezentovaný uzlom má taktiež svojich rodičov, ktorý reprezentujú cestu k tomuto súboru.

Obe triedy obsahujú položku, ktorá udáva ich typ. V prípade uzlu ide o textovú reprezentáciu uzlu, ktorá informuje užívateľa, čo daný uzol reprezentuje. Všetky typy uzlov vo výslednom module sa začínajú slovom PigLatin, ktoré následuje významový pojem. Napr. PigLatin Command, PigLatin Tuple atď. Typ hrany určuje typ dátového toku. V tejto práci sa používajú dva typy, kedy priamé toky reprezentuje typ `Edge.Type.DIRECT` a filter tok typ `Edge.Type.FILTER`. Trieda taktiež umožnuje získať množinu hrán a uzlov pomocou get metód.

V predchádazjúcich kódoch je často spomínaná trieda `Resource`. Táto trieda reprezentuje zdroj odkiaľ dáta pochádzajú. V tejto práci sa služby tejto triedy nepoužívajú ale je nutné ju dotať z dôvodou kompatibility s Manta systémom.

4.3.2 PigLatinDataFlowVisitorFactory

Táto trieda, ako už jej názov vypovedá, implementuje návrhový vzor Factory. Jej úlohou je generovať objekty typu `PigLatinDataFlowVisitor`. Pre konštrukciu tejto triedy je potrebné dodať 3 parametre. Jedná sa o dátové zdroje typu `Resource` pre databázové objekty a objekty skriptu. Posledným parametrom je jeden objekt typu `NodeCreator`. Objekt `NodeCreator` má za úlohu vytvárať systémove uzly reprezentujúce súbory alebo tabuľky. V tejto práci slúži táto trieda pre tvorbu uzlov reprezentujúcich súbory. Pre potrebu tejto práce vytváram tieto uzly ako inštancie triedy `ResourceImpl` a `NodeCreatorImpl`. Príklad je uvedený v nasledujúcom kóde 11.

```
NodeCreatorImpl nodeCreator = new NodeCreatorImpl();
Resource scriptResource = new ResourceImpl("Script", "Script",
                    "Script");
Resource dbResource = new ResourceImpl("PigLatin", "PigLatin",
                    "PigLatin");
```

Zdrojový kód 11: Ukážka objektov tried NodeCreator a Resource

Jednotlivé parametry konštruktoru sú v tomto momente nepodstatné, pretože sa služby triedy `Resource` nepoužívajú.

Jediná významná metóda tejto triedy je *createFlowVisitor(Graph outputGraph, Node scriptNode)*. Jej parametre sú odkaz na graf (aktuálne prázdny graf) a odkaz na uzol, ktorý reprezentuje skript. Vo výslednom grafe tento uzol reprezentuje rodiča všetkých uzlov, ktoré sa vyskytujú v globálnom bloku Pig Latin kódu. Táto metóda vytvára inštanciu `PigLatinDataFlowVisitor`, ktorý je popísaný v nasledujúcich častiach.

4.3.3 PigLatinDataGraphHelper

Ako už z názvu vypovedá, táto trieda predstavuje pomocnú triedu pre vytváranie grafových uzlov a hrán. Jej hlavné metódy popisuje následujúci zoznam:

- buildOpNode - Metóda vytvorí grafový uzol pre uzol operátoru z AST.
- getScriptNode - Metóda získa uzol reprezentujúci skript pre daný uzol.
- buildSourceFileNode - Metóda vytvorí uzol pre súbor.
- buildNode - Metóda vytvorí uzol pre resolvovanú entitu.
- addDirectEdge - Metóda prepojí dve resolvované entity pomocou priamych tokov.

Popri metódach si táto trieda udržiava objekty, ktoré sú získané pomocou konštruktoru pri vytváraní inštancie tejto triedy. Jedná sa o odkaz grafu, do ktorého sa pridávajú nové uzly a hrany. Ďalej ide o zdroj pre databázové objekty, zdroj pre skriptové objekty, odkaz na uzol, ktorý reprezentuje skript a inštancia triedy `NodeCreator` pre tvorbu uzlov reprezentujúcich súbory.

Pre efektívnu prácu modulu, ktorá je jedným z funkčných požiadavkov NF2, sú v tejto triede kolekcie, ktoré si udržiavajú informácie o vytvorených objektoch. Jedná sa o hashovacie tabuľky typu Java HashMap s názvom `bindings` a `operationBindings`. Prvá tabuľka sa používa pri vytváraní uzlov pre revolvovanú entitu v metóde `buildeNode`. Táto metóda skontroluje, či sa daná entita už nevytvárala, a ak áno, tak vráti existujúci uzol, ktorý ju reprezentuje. V prípade, že sa nenachádza v mape, tak sa uzol vytvorí a pridá sa nový záznam do tabuľky. V prípade druhej tabuľky ide o obdobný postup, kedy si tabuľka udržiava záznamy o uzloch operátorov a tieto operátory spracováva metóda `buildOpNode`.

4.3.4 PigLatinDataFlowVisitor

Úlohou tejto triedy je generovanie grafu dátový tokov z AST. Táto trieda rozširuje triedu z modulu `Model`, ktorá obsahuje metódy `accept()` pre spracovanie každého dôležitého uzlu. Všetky tieto metódy sú v `PigLatinDataFlowVisitor` preddefinované.

V tejto časti sa nerozoberá logika spracovania každého uzlu ale len popis skupín uzlov. Tieto skupiny sú operátory, príkaz a identifikátor. Následne uvádzam popis jednotlivých skupín.

Spracovanie všetkých operátorov prebieha následne:

1. Vytvorí sa uzol, ktorý reprezentuje operáciu.
2. Vytvoria sa uzly, ktoré reprezentujú výstup z operátoru.
3. Získa sa vstupná relácia pre operátor.
4. Všetky listy zo vstupnej relácie sa prepoja s výstupom z operátoru.
5. V prípade, že existujú v operátore nepriamé ovplyvnenia (klauzuly BY, podmienky) vytvoria sa filter hrany medzi týmito dátami a výstupnými dátami.
6. Vráti sa zoznam listov z výstupu operátoru.

Pre upresnenie, jednou z podmienok, ktorá sa v Manta systéme dodržuje je, že dátové toky vždy prebiehajú v listoch. Listy predstavujú uzly, ktoré nemajú potomkov. Z tohoto dôvodu je v tejto triede implementovaná metóda `getAllLeaves`, ktorá vytvorí a vráti všetky uzly z resolvovanej entity, ktoré sú listami. Táto metóda sa používa často, pretože predstavuje efektívne získavanie a vytváranie uzlov v grafe. Výstup zo spracovania všetkých operátorov predstavuje zoznam listov.

Spracovanie identifikátoru prebieha pomocou dvoch krokov:

1. Vytvorí sa uzol pre resolvovaný objekt identifikátoru.
2. Získajú sa jeho listy, ktoré sa vrátia.

V následujúcej časti uvádzam kód 12, ktorý predstavuje poslednú skupinu, spracovanie príkazu.

```
List<Node> identifier = processAstNode(node.getIdentifier());
List<Node> command = processAstNode(node.getOperator());
graphHelper.addDirectFlow(command, identifier);
```

Zdrojový kód 12: Spracovanie uzlu príkazu

Telo metódy pre spracovanie príkazu pracuje v troch krokoch. V prvom spracuje identifikátor, ktorý získa z uzlu AST volaním metódy pre zisk identifikátoru. V druhom kroku nasleduje obdobné spracovanie operátoru. Z oboch týchto volaní sa získajú listy. Tieto listy následne prepojím pomocou priamych dátových tokov v pomocnej triede.

4.3.5 PigLatinDataFlowTask

Posledná trieda z balíčku `Generator` reprezentuje úlohu pre spracovanie kódu vo forme AST do formy grafu dátových tokov. Trieda rozširuje `AbstractGraphTask`, ktorá predstavuje generickú triedu tohoto spracovania pre projekt Manta. Konštruktor pre triedu pozostáva z parametrov typu `NodeCreator` pre tvorbu súborových uzlov, `PigLatinDataFlowVisitorFactory` pre vytvorenie Visitoru spracovania dátových tokov a zdroja `Resource`, ktorý označuje zdroj skriptu. Jedinou dôležitou metódou tejto triedy je `doExecute()`. Táto metóda prijíma na vstupe AST, ktoré chceme spracovať a jej druhý parameter predstavuje odkaz na graf. Odkaz na graf predstavuje prázdny alebo neprázdny graf, do ktorého sa pridajú uzly a hrany, ktoré sa získajú spracovaním vstupného AST.

Kapitola 5

Testovanie

Táto kapitola sa zaoberá testovaním, ktoré prebieha za pomoci frameworku `JUnit`. Testovanie prebieha v module `Resolver` a v module `Generator`.

5.1 Resolver

Testovanie modulu Resolver má za úlohu otestovať spracovanie kódu do podoby AST. Testovanie prebieha v triede s názvom `PigParserTest` z balíčku `eu.profinit.manta.connector.piglatin.resolver`. Táto trieda obsahuje 5 funkčných testovacích metód, ktorých cieľom je otestovať funkcionalitu. Týchto 5 testovacích metód sa delí na dve skupiny, kedy jedna skupina testuje parsovanie kódu do formy AST (prvé dva testy) a druhá sa zaujíma o funkčnosť resolvingu (posledné tri testy).

V triede existuje metóda `setup()`, ktorá sa vyhodnocuje pred spustením testovacích metód. Úlohou tejto metódy je inicializovať a nakonfigurovať objekty potrebné pre testovanie. V následujúcich odstavcoch sú popísané testovacie metódy a ich úlohy.

`testParse1`. Táto testovacia metóda testuje funkčnosť parsingu jednoduchého Pig Latin kódu. Dodaný kód v testovacej metóde je plne funkčný, takže parsing má skončiť úspechom. V prípade funkčnosti parsingu, parsing nevyhodí žiadnu výnimku a vtedy nastáva úspech testovacej metódy.

`testParse2`. Tento test pracuje na opačnom princípe ako `testParse1()`. Kód dodaný do testovacej metódy nie je syntakticky správny, takže parsing má skončiť chybou v podobe vyhodenia výnimky typu `MismatchedTokenException`. V prípade funkčnosti parsingu nastáva vyhodenie výnimky, kedy testovací framework očakáva táto výnimku a testovacia metóda následne zahlási úspech.

testResolve1. Táto metóda sa zaoberá testovaním resolvingu. V metóde sa predpokladá funkčnosť parsingu, ktorý testujú predchádzajúce dve metódy. Test pozostáva z volania metódy `resolve()` nad prvým identifikátorom z AST. V prípade funkčnosti resolvingu sa skutočná reprezentácia identifikátoru získa (nie je null) a test skončí úspechom.

testResolve2. Testovacia metóda testuje funkčnosť resolvingu relácie z bloku. V teste ide o dosiahnutie resolvingu identifikátoru `A`, ktorý sa vyskytuje na dvoch miestach. V prípade funkčnosti resolvingu sa skutočná entita z bloku získa a test skončí úspechom.

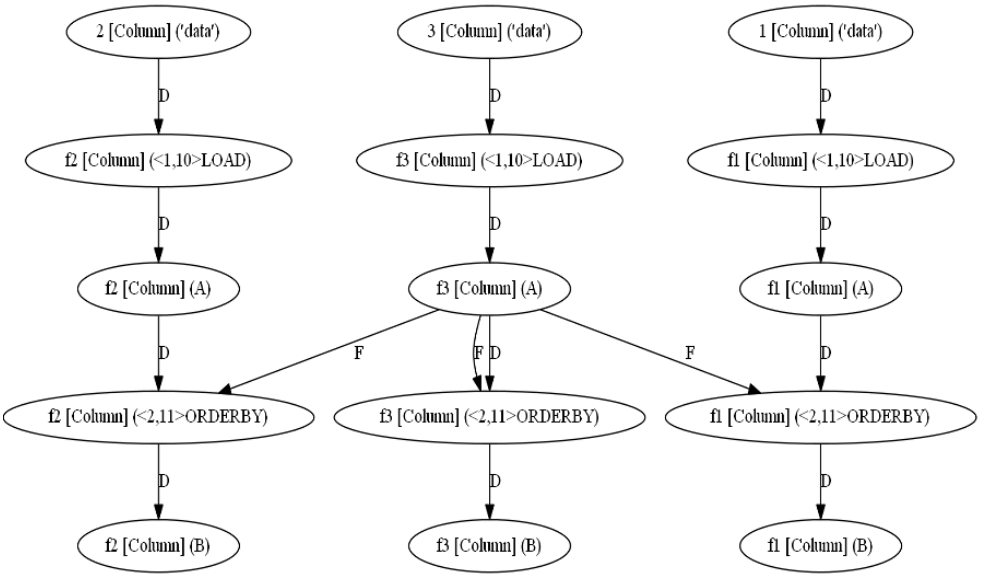
testResolve3. Posledná testovacia metóda testuje všetky funkčnosti. V teste nastáva parsing kódu, resolving objektov z bloku a menných priestorov. Testovacia podmienka testuje resolving poľa `name` z relácie `A`. V operátore `FILTER A BY name` sa toto pole nachádza v mennom priestore a týmto testom sa zistí funkčnosť resolvingu nad menným priestorom. V prípade nálezu, test končí úspechom.

5.2 Generator

Trieda `PigLatinDataFlowTest` z modulu `Generator` má za úlohu otestovať funkčnosť generovania grafu dátových tokov. Táto trieda je v tejto testovacej časti závislá na module `Resolver`, pretože je potrebné, pre testovacie účely, získať triedy pre spracovanie kódu. S týmto pomôže správca závislostí, Apache Maven. V `pom.xml` sa definuje závislosť na module `Resolver` s dôrazom, že jeho použitie je možné len v testovacích triedach. Takto sa dodajú potrebné triedy do testovacích metód.

Telo testovacej metódy `testDataFlow1()` pozostáva z inicializácie Manta zdrojov, grafu a načítania súboru so skriptom. Tento súbor obsahuje Pig Latin skript, ktorý obsahuje všetky spracované Pig Latin operátory. Následne nastáva parsing skriptu. Testovacou podmienkou je porovnanie grafu dátových tokov v textovej reprezentácií s testovacím výpisom grafu dátových tokov, ktorý je dodaný zo skriptom. Ak sa výstupy grafov zhodujú, tak analýza dátových tokov funguje správne a test skončí úspešne.

Pre tento modul bolo vykonaných mnoho manuálnych testov. Manuálne testy boli vykonávané za pomoci knižnice `GraphViz` a interných Manta tried, ktoré však nie sú poskytnuté v tejto práci. Ukážka tejto vizualizácie pomocou `GraphViz` a Manta tried je znázornená na obrázku 5.1.



Obr. 5.1: Vizualizácia grafu v Graphviz

Záver

Hlavným cieľom tejto práce bolo vytvoriť modul, ktorý dokáže analyzovať kód v jazyku Pig Latin a získať jeho dátové toky. Tento cieľ som rozdelil na niekoľko menších cieľov. Jednalo sa o analýzu jazyka Pig Latin a spracovanie jeho kľúčových častí pre modul analýzy dátových tokov. Tento cieľ som splnil a všetky jeho dôležité časti pre analýzu dátových tokov sú spracované. Ďalším cieľom bolo navrhnúť a implementovať modul, ktorý vykonáva syntaktickú a sémantickú analýzu, a ich výsledok používa pre generovanie grafu dátových tokov. Všetky tri časti modul vykonáva. Modul taktiež obsahuje patričnú dokumentáciu vo forme Javadoc a takisto obsahuje sériu testov, ktoré dokazujú funkčnosť modulu, čo predstavovalo ďalšie ciele tejto práce. Výsledný modul však nedokáže analyzovať dátové toky pre dva operátory. Tieto dva operátory, FOREACH a NATIVE, sú z práce vynechané pre svoju komplexnosť, a po následnej dohode s vedúcim práce.

Výstup z tejto práce, ktorý je vo forme Java modulu, je možné v blízkej dobe integrovať do Manta systému. Tento modul je implementovaný s dôrazom, že sa bude implementovať do Manta systému, a tak používa všetky potrebné prostriedky a rozhrania z Manta systému. Pre plnú funkčnosť analýzy dátových tokov jazyka Pig Latin je potrebné v module implementovať spracovanie operátorov FOREACH a NATIVE. Implementácia týchto operátorov nepredstavuje žiadne značné zásahy do implementovaného kódu ale len rozšírenie existujúcej implementácie o tieto operátory a ich potrebné mechaniky. Ďalšie možné vylepšenia pre modul pozostávajú vo vytvorení komunikácie s inými analyzátormi dátových tokov, ktoré dokážu spracovať UDF, a tak podporu pre rozšírenejšiu analýzu dátových tokov Pig Latin kódu.

Literatúra

[1] Techopedia: *Data Lineage [online]*. [cit. 2018-05-08]. Dostupné z: https://www.techopedia.com/definition/28040/data-lineage

[2] The Apache Software Foundation: *Co jsou nástroje business intelligence (BI)? [online]*. [cit. 2018-05-08]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/what-are-business-intelligence-tools/

[3] Manta Tools, s.r.o.: *MANTA - Get end-to-end data lineage including custom SQL code. [online]*. [cit. 2018-04-11]. Dostupné z: https://getmanta.com/

[4] How To Inspect Raw Data Lineage With Manta Flow. *Manta Blog [online]*, 7 2016, [cit. 2018-04-17]. Dostupné z: https://getmanta.com/how-to-inspect-raw-data-lineage-with-manta-flow

[5] The Apache Software Foundation: *Welcome to Apache Pig! [online]*. [cit. 2018-04-08]. Dostupné z: https://pig.apache.org/

[6] Oracle: *What is Java technology and why do I need it? [online]*. [cit. 2018-04-14]. Dostupné z: https://www.java.com/en/download/faq/whatis_java.xml

[7] The Apache Software Foundation: *What is Maven? [online]*. 04 2018, [cit. 2018-04-17]. Dostupné z: https://maven.apache.org/what-is-maven.html

[8] The Apache Software Foundation: *Pig Latin Basics [online]*. [cit. 2018-04-11]. Dostupné z: http://pig.apache.org/docs/r0.17.0/basic.html

[9] The Apache Software Foundation: *User Defined Functions [online]*. [cit. 2018-04-17]. Dostupné z: https://pig.apache.org/docs/r0.17.0/udf.html

[10] The Apache Software Foundation: *Apache Pig [online]*. [cit. 2018-04-30]. Dostupné z: `https://github.com/apache/pig`

Dodatok A

Zoznam použitých skratiek

AST Abstract Syntax Tree.

BI Bussines intelligence.

IDE Integrated Development Environment.

Dodatok B

Príklad použitia

Tento dodatok reprezentuje jednoduchú príklad použitia implementovaných tried pre analýzu a získanie dátových tokov z modulu. V následujúcom texte sú prezentované dve časti, jedna predstavuje inicializáciu potrebných tried modulu a druhá použitie týchto tried pre analýzu dátových tokov.

Na začiatok uvádzam kód 13, ktorý slúži ako ukážkový príklad inicializácií všetkých nutných tried pre analýzu dátových tokov.

```
ParserServiceParams params = new ParserServiceParams();

PigLatinResolverEntitiesFactory entitiesFactory =
        new PigLatinResolverEntitiesFactoryImpl();
ParserService parserService = new ParserServiceImpl();

NodeCreatorImpl nodeCreator = new NodeCreatorImpl();
Resource scriptResource = new ResourceImpl("Script", "Script",
        "Script");
Resource dbResource = new ResourceImpl("PigLatin", "PigLatin",
        "PigLatin");
Graph outputGraph = new GraphImpl(scriptResource);

PigLatinDataFlowVisitorFactory factory =
        new PigLatinDataFlowVisitorFactory
        (dbResource, scriptResource, nodeCreator);

PigLatinDataFlowTask dataFlowTask = new PigLatinDataFlowTask
        (nodeCreator, factory, scriptResource);
```

Zdrojový kód 13: Príklad inicializácie

Na začiatok sa vytvorí objekt pre dodatočné parametre. Tento objekt nasleduje tvorba inštancie továrne na resolvované objekty a parsovaciu službu. Ostatné objekty sa týkajú modulu pre generovanie dátových tokov, kde inicializujem potrebné zdroje a inštanciu grafu, ktorá je momentálne prázdna. Následne prichádza tvorba továrne pre `PigLatinDataFlowVisitor` a úlohy.

Následujúci kód 14 popisuje spracovanie kódu Pig Latin, ktorý je obsiahnutý v reťazci *script*.

```
IPigLatinAstNode result = parserService.parseStringScript(
                    script, factory, params, scriptName);

dataFlowTask.execute(tree, outputGraph);
```

Zdrojový kód 14: Príklad použitia

Spracovanie vstupu na graf dátových tokov je možné dosiahnuť volaním týchto dvoch metód, za predpokladu existencie nutných prostriedkov popísaných vyššie. Prvá metóda parsuje zadaný vstup vo forme reťazca, ktorý následne spracuje druhý príkaz. Druhý príkaz generuje uzly a hrany z AST, a postupe ich pridáva do grafu, ktorý je predaný ako vstupný parameter. Tento graf je však reprezentovaný len dátovou štruktúrou.

V prílohe sa tieto kódy nachádzajú v triede `PigLatinDataFlowTest`.

Dodatok C

Obsah priloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementácie
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF