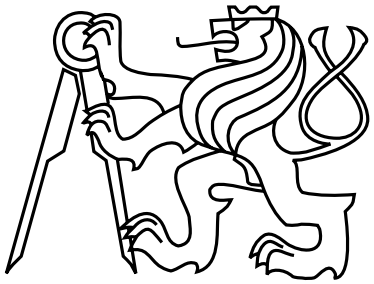
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práca

Lexikální analyzátor dialektu dotazovacího jazyka databáze MySQL pro zachytávání změn v databázi

Bc. Roman Kuchár

Vedúci práce: Ing. Jiří Pechanec

Študijný program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

14. mája 2018

Poďakovanie

Predovšetkým by som chcel poďakovať vedúcemu mojej diplomovej práce Ing. Jiřímu Pechancovi za jeho čas, skúsenosti a výborné rady, ktoré mi počas konzultácií poskytol. Zároveň by som rád poďakoval aj vedúcemu projektu Debezium p. Gunnarovi Morlingovi za jeho ochotu poradiť mi počas začleňovania novej implementácie do projektu. Poďakovanie taktiež patrí mojej rodine a priateľom, ktorí ma pri písaní diplomovej práce podporovali.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky použité informačné zdroje v súlade s Metodickým pokynom o dodržovaní etických princípov pri príprave vysokoškolských záverečných prací.

V Prahe dňa 20. 5. 2018 .....

Abstract

Project Debezium is distributed platform for change data capture in database systems. One of supported databases is MySQL, where Debezium needs meta-data for describing of the database structure. These meta-data are acquired by analysis of DDL statements, which has to be parsed. This thesis deals with the possibilities of generating the syntactic analyzers necessary to modify the in-memory model, which describes the database structure. One part of this thesis is the design and implementation of generated parser, which will replace the existing, hand written, solution in Debezium project.

Abstrakt

Projekt Debezium je distrubuovaná platforma na zachytávanie zmenených dát v databázových systémoch. Jednou z podporovaných databázi je MySQL, pri ktorej Debezium potrebuje metadáta popisujúce štruktúru databáze. Tieto metadáta sú získavané na základe analýzy DDL dotazov, ktoré je nutné sparsovať. Táto práca sa zaoberá možnosťami generovania syntaktických analyzátorov potrebných na úpravu modelu uloženého v pamäti, ktorý popisuje štruktúru databáze. Súčasťou práce je návrh a implementácia generovaného syntaktického analyzátoru, ktorý nahradí stávajúce, ručne napísané riešenie v projekte Debezium.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam ukážok

Kapitola 1

Úvod

Každý databázový systém má svoj dotazovací jazyk pomocou ktorého s ním užívateľ môže manipulovať. Tento jazyk je jasne definovaný svoju syntaxou, ktorá určuje súhrn pravidiel udávajúcich prípustné tvary čiastkových konštrukcií a celého dotazu. Na to, aby databázový systém vedel, akú akciu sa snaží užívateľ vykonať, musí zanalyzovať výraz napísaný pomocou jeho syntaxe.

Projekt Debezium si v pamäti udržiava štruktúru sledovanej databáze a snaží sa zachytávať zmeny nad touto databázou. Rovnako ako databázový systém musí zanalyzovať syntaxi spusteného výrazu, aby vedel, ako užívateľ mení štruktúru databáze a mohol rovnaké zmeny aplikovať na svoj model uložený v pamäti. Stávajúci syntaktický analyzátor je ručne napísaný, je veľmi jednoduchý a zďaleka nepostihuje všetky nuance SQL jazyka, čím sa stáva náchylným k chybám. Novo implementovaný strojovo generovaný syntaktický analyzátor nahradí aktuálne riešenie v projekte Debezium, čím sa zníži pravdepodobnosť vzniku chýb, a bude možné ho upraviť jednoduchou zmenou v gramatike jazyka nad ktorým bude pracovať.

1.1 Motivácia

Analýza MySQL dotazov alebo akejkoľvek inej dôležitej relačnej databázy sa môže javiť ako skľučujúca úloha. Zvyčajne databázový systém má vysoko prispôsobenú gramatiku SQL a hoci výrazy jazyka manipulácie s údajmi (DML) sú často pomerne blízke štandardom, výrazy jazyka pre definíciu dát (DDL) sú zvyčajne menej blízke a zahŕňajú viac špecifických funkcií databázového systému.

Mnoho aktuálne implementovaných a prístupných analyzátorov rieši iba analýzu základných DDL výrazov a nepodporuje špecifické možnosti jednotlivých databázových systémov. Existujú aj analyzátory napísané konkrétne pre MySQL, no často sú nekompletné alebo nepodporujú poslednú verziu tejto databáze. Použitie týchto dostupných, no nekompletných implementácií by mohlo pokryť väčšinu požiadavkov projektu Debezium, no zvyšok by musel byť implementovaný iným spôsobom, čo by bolo veľmi zmätočné a náchylné k chybám.

1.2 Cieľ práce

Cieľom práce je zanalyzovať projekt Debezim, jeho aktuálnu implementáciu MySQL DDL syntaktického analyzátoru a navrhnúť nové riešenie. Implementácia nového riešenia by mala byť intuitívne pochopiteľná, bez nutnosti študovania rozsiahlých dokumentačných materiálov. Projekt Debezium plánuje v budúcnosti rozširovať množstvo podporovaných databázových systémov, a preto by výsledok tejto práce mal byť implementovaný čo najprijateľnejšie voči jeho potencionálnemu prepoužitiu pri syntaktických analyzátoroch iných databázových systémov.

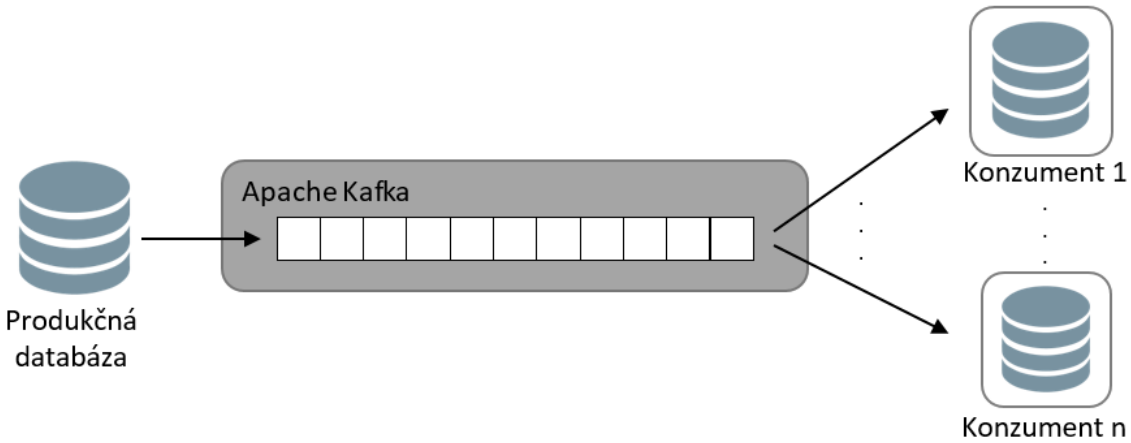
Kapitola 2

Debezium

Debezium[4] je projekt, ktorý slúži k zaznamenávaniu zmien v databáze analýzou udalostí v transakčnom logu. Jednou z podporovaných databáze je taktiež MySQL. Na správnu funkcionalitu Debezium potrebuje metadáta popisujúce štruktúru databáze v závislosti na čase. Pre MySQL je to možné dosiahnuť tak, že príkazy, ktoré vytvárajú alebo upravujú štruktúru databáze (DDL) sú zachytávané, parsované a na ich základe je upravený model v pamäti, ktorý popisuje štruktúru databáze.

2.1 Zachytávanie zmenených dát

Hlavnou myšlienkou zachytávania zmenených dát, anglicky Change Data Capture (CDC), je vytvárať sled udalostí, ktoré reprezentujú všetky zmeny v tabuľkách danej databáze. To znamená, že pre každý *insert*, každý *update*, každý *delete* dotaz sa vytvorí jedna odpovedajúca udalosť, ktorá bude odoslaná a následne dostupná pre konzumentov tohto sledu viď obrázok 2.1 [23]. V projekte Debezium sa na sprostredkovanie sledu udalosti využíva Apache Kafka [25] infraštruktúra, no myšlienka CDC nie je viazaná na Kafku.



Obr. 2.1: Koncept distribúcie zmenených dát

2.1.1 Dátové sklady

V dátových skladoch sa uchovávajú dáta a ich história z viacerých databáz za účelom analytických výpočtov. Aktuálne najpoužívanejším riešením na napĺňanie dátových skladov je ETL[1]. Tento proces funguje v zmysle periodického nahrávania veľkého množstva dát do dátových skladov. Tento proces sebou ale nesie nevýhody, ktoré v minulosti neboli tak podstatné, no v dnešnej dobe už sú. Použitím CDC miesto ETL je možné niektoré z týchto nevýhod obmedziť. Dátové sklady sa za pomoci CDC napĺňajú priebežne a nie periodicky, takže dáta nad ktorými sa vykonáva analýza budú skoro vždy aktuálne. Nakoľko sa každá zmena zapisuje jednotlivo a nie pomocou veľkého balíčku, neobmedzí to beh systémov, zamedzí nutnosti systémových prestojov a zároveň to redukuje cenu za túto operáciu. Premiestnením iba zmenených údajov CDC vyžaduje oveľa menej zdrojov na presun a transformáciu dát. To zníži náklady na hardvér, softvér a aj ľudské zdroje. [2]

2.1.2 Replikácia dát

Jedným z využití CDC je replikácia dát do iných databáz napríklad v zmysle vytvorenia zálohy dát, ale taktiež je možné CDC využiť pri implementácií zaujímavých analytických požiadavkov. Predstavme si, že máme produkčnú databázu a špecializovaný analytický systém na ktorom chceme spustiť analýzu. V tomto prípade je nutné dostať dáta z produkčnej databáze do analytického systému a CDC je možnosť, ktorá nám to umožní. Ďalším využitím môže byť prísun dát ostatným tímom, ktoré na základe nich môžu napríklad vypočítavať a smerovať svoju marketingovú kampaň, napríklad na užívateľov, ktorí si objednali istý konkrétny produkt. Nakoľko nechceme aby sa takýto výpočet vykonával nad produkčnou databázou ale skôr nad nejakou separovanou databázou, tak opäť je možné využiť CDC na propagáciu dát do separovanej databáze, kde si už marketingový tým môže vykonávať akokoľvek náročné výpočty.

2.1.3 Microservice Architecture

Ďalšie využitie CDC je vhodné pri použití Microservice architecture, kde je doména rozdelená na niekoľko služieb, ktoré potrebujú medzi sebou interagovať. Pre príklad máme tri mikro služby: objednávaciu aplikáciu na spracovávanie užívateľských objednávok, produktovú službu, ktorá sa stará o produktový katalóg, a nakoniec máme skladovú službu, ktorá kontroluje reálne množstvo produktových vecí na sklade. Je zreteľné, že na správne fungovanie bude objednávacia aplikácia vyžadovať dáta od produktovej a skladovej služby. Jednou z možností je, že objednávacia aplikácia bude priamo komunikovať s ostatnými službami napríklad pomocou REST API[2], čím ale bude úzko spojená a závislá na chode danej služby. Ak by takáto služba zlyhala/spadla tak nebude fungovať celá aplikácia. Druhou možnosťou je práve využiť CDC, ktoré odzrkadľuje použitie Event Sourcing paternu[3]. Produktová a skladová služba budú poskytovať sled zmenených dát a objednávacia aplikácia ich bude

[1] Export, transform, load

[2] <https://cs.wikipedia.org/wiki/Representational_State_Transfer>

[3] <http://microservices.io/patterns/data/event-sourcing.html>

zachytávať a udržiavať kópiu časti týchto dát, ktoré ju zaujímajú, vo vlastnej lokálnej databáze. Ak by v takomto prípade niektorá zo služieb zlyhala, tak objednávacia aplikácia môže naďalej fungovať.

2.1.4 Ostatné

Bežnou praxou vo väčších aplikáciách je používanie cache pre rýchly prístup k dátam na základe špecifických dotazov. V takýchto prípadoch je potrebné riešiť problémy updatu cache alebo jej zmazania, pokiaľ sa isté dáta zmenia.

Riešenie fulltextového vyhľadávania pomocou databáze nie je veľmi vhodné a namiesto toho sa používa SOLR[4] alebo Elasticsearch[5], čo sú systémy, ktoré potrebujú byť synchronizované z dátami v primárnej databáze.

2.2 Odchytávanie zmien v databáze

Každý databázový systém (DBMS) má svoj log súbor, ktorý používa na zotavenie sa po páde a odvolaní transakcií, ktoré ešte neboli potvrdené, alebo na replikáciu dát voči sekundárnym databázam alebo inej funkcionalite. Či už to sú transakčné, binárne alebo replikačné logy, vždy v sebe udržujú všetky transakcie, ktoré boli úspešne vykonané nad databázou, a preto sú vhodné na odchytávanie zmien v databázach pre projekt Debezium. Konkrétne v MySQL databáze sa volá **binlog** (3.1.1). Nakoľko sú tieto logy plne transparentné voči aplikácii, ktorá do databáze zapisuje, výkon aplikácie nebude nijako ovplyvnený čítaním týchto logov.

2.2.1 Apache Kafka

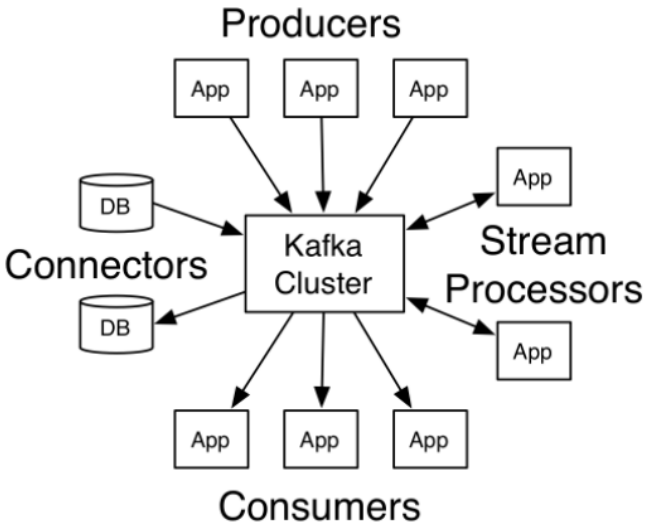
Apache Kafka je open-source distribuovateľná platforma na streamovanie správ vyvinutá firmou Apache Software Foundation. Umožňuje vytvárať a sledovať tok záznamov podobný fronte správ. Tento tok ukladá spôsobom odolným voči chybám. Hlavným použitím Kafky je vytváranie dátových potrubí v reálnom čase, ktoré spoľahlivo získavajú dáta medzi systémami alebo aplikáciami a budovanie aplikácií na streamovanie v reálnom čase, ktoré transformujú alebo reagujú na prúdy dát. Kafku je možné spustiť ako cluster na jednom, alebo viacerých serveroch, ktoré ukladajú toky záznamov v kategóriách nazvaných **topiky**. Každá správa v Kafke pozostáva z kľúča, hodnoty a časovej značky. Každej správe Kafka priradí sekvenčné identifikačné číslo nazývané **offset**, ktoré unikátne identifikuje každý záznam a jeho poradie v topiku.

Kafka pozostáva zo štyroch základný API:[25]

- **Producer API**, ktoré umožňuje aplikáciám publikovať sled záznamov do jedného alebo viacerých topikov.
- **Consumer API**, ktoré umožňuje konzumovať existujúce topiky a spracovať sled záznamov, ktorý obsahujú.

[4] <http://lucene.apache.org/solr/>
[5] <https://www.elastic.co/>

- **Streams API**, ktoré umožňuje aplikáciám chovať sa ako spracovávateľ sledu záznamov. Aplikácie pohlcujú prichádzajúci sled a produkujú transformovaný výstupný sled.
- **Connector API**, ktorý umožňuje vytvárať znovu použiteľné dátové spojenia na publikovanie a konzumovanie záznamov, ktoré pripoja topiky k existujúcim aplikáciám alebo dátovým systémom.



Obr. 2.2: Hierarchia Apache Kafka

Projekt Debezium využíva posledné zmienené *Connector API* použitím Kafka Connect frameworku (2.2.3), pomocou ktorého implementuje CDC pre jednotlivé databázové systémy.

2.2.2 Infraštruktúra správ pomocou Apache Kafka

Apache Kafka poskytuje sémantické pravidlá, ktoré dobre vyhovujú potrebám projektu Debezium. Prvým z nich je, že všetky správy v Kafke majú kľúč a hodnotu. Táto vlastnosť sa využíva na zjednotenie správ, ktoré spolu súvisia a to konkrétne tak, že na základe primárneho kľúča v tabuľke, ktorej zmena sa zmena týka je možné štruktúrovať kľúč správy a hodnota správy bude reprezentovať konkrétnu zmenu.

Kafka taktiež garantuje poradie správ metódou FIFO[6], čím sa zabezpečí správne poradie zmien, ktoré bude konzument prijímať. Táto vlastnosť je veľmi dôležitá, nakoľko ak by nastala situácia *insert* a následne *update* alebo dve *update* akcie za sebou, tak musí byť zabezpečené, aby sa ku konzumentovi dostali v správnom poradí inač by mohla nastať nekonzistencia voči dátam v primárnej databáze a dátam, ktoré si udržiava konzument.

Kafka je pull-based systém, čo znamená, že konzument je sám sebe pánom a drží si informáciu o tom, ktoré správy z konkrétneho topiku už prečítal resp. kde chce začať čítanie

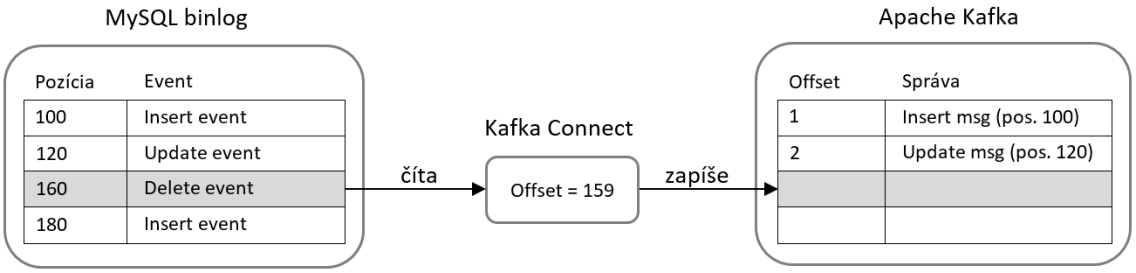
[6]First in First out

ďalších správ. Takto môže sledovať aktuálne pribúdajúce správy, ale taktiež sa môže zaujímať aj o správy z minulosti.

Zmien v databázach môže byť veľmi veľa, čo spôsobí veľké množstvo udalostí, a preto je nutné spomenúť ďalšiu výhodu Kafky a to jej škálovateľnosť. Kafka podporuje horizontálnu škálovateľnosť a jednotlivé topiky môžu byť rozdelené na viacero partícií. Je ale nutné si uvedomiť, že poradie zmien je garantované iba na konkrétnej partícii. Kafka zabezpečí, že všetky správy s rovnakým kľúčom budú na rovnakej partícii, čím sa garantuje ich správne poradie, ale môže nastať situácia, že udalosť s iným kľúčom, ktorá reálne nastala neskôr, môže byt konzumentom spracovávaná skôr, čo môže, ale aj nemusí vadiť v závislosti na konkrétnej funkcionalite konzumenta.

2.2.3 Kafka Connect

Kafka Connect je framework, ktorý umožňuje jednoduchú implementáciu dátových spojení (konektorov) s Kafkou. Tieto konektory majú na starosti dáta, ktoré vstupujú alebo vystupujú z Kafky. Nazývajú sa *source*(vstupujúce dáta resp. import) konektory alebo *sink*(vystupujúce dáta resp. export) konektory. Debeziové konektory majú na starosti naplňovanie Kafky, takže sa používa *source* konektor. Kafka Connect ponúka možnosť na riešenie offsetu. Na rozdiel od Kafka offsetu, ktorý je priradený každej správe v topiku, Kafka Connect offset si udržuje informáciu o pozícií poslednom prečítanom evente z binlogu. Môže nastať situácia, že konektor zhavaruje a bude musieť byť reštartovaný. V takomto prípade konektor potrebuje vedieť ako ďaleko v čítaní logu bol a kde má s čítaním pokračovať.

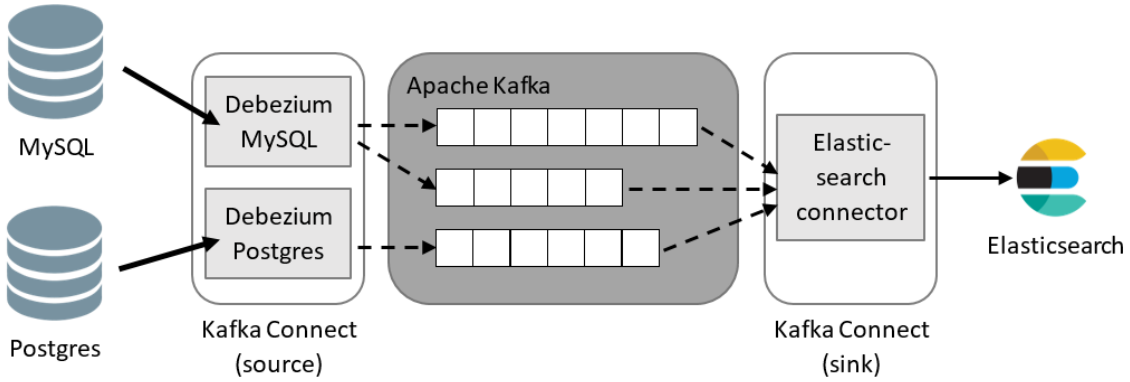


Obr. 2.3: Príklad nastavenia Kafka a Kafka Connect offsetov

Pre príklad si povedzme, že jedna udalosť v MySQL binlogu má veľkosť 20. Na obrázku 2.3 je možné vidieť situáciu, kedy sa konektoru podarilo prečítať a spracovať udalosti z MySQL binlogu nachádzajúce sa na pozíciách 100 a 120. Ich príslušné správy sú dostupné v Kafke s nastaveným Kafka offsetom 1 a 2. Kafka Connect má v tejto situácií offset rovný 159, nakoľko to je posledná prečítaná pozícia. Použitím Kafka Connect je zabezpečené, že po každom spracovaní udalosti konektor potvrdí svoj offset, a ak by konektor musel byť reštartovaný, tak môže zistiť posledný potvrdený offset a pokračovať v čítaní logu z nasledujúcej pozície.

Ďalším prínosom je možnosť konfigurácie schémy správ. Kafka Connect má svoj systém na definovanie typu dát, ktorým umožňuje popísať štruktúru kľúčov a hodnôt v správach. Bližšie popísané v kapitole 2.2.4.

Kafka Connect je clustrovateľná, takže je možné v závislosti na špecifikácií rozdeliť konektor a jeho úlohy medzi viacero uzlov. Taktiež ponúka bohatý eko-systém konektorov. Na stránkach Confluent[7] je možné si stiahnuť rôzne typy či už *sink* alebo *source* konektorov. Príklad CDC topológie s použitím Kafka Connect je na obrázku 2.4 [23]. Zámerom v danom obrázku je zdieľať dáta dvoch tabuliek z MySQL databáze a jednej tabuľky z Postgres databáze. Každá monitorovaná tabuľka je vyjadrená jedným topikom v Kafke. Prvým krokom je nastavenie clusterov v Apache Kafka, pričom je to možné spustiť na jednom alebo viacerých clusteroch. Ďalším krokom je nastavenie Kafka Connect, ktorá je oddelená od Apache Kafka a beží v separátnych procesoch alebo clustroch, a ktorá bude spravovať spojenie z Apache Kafka. Následne je nutné nasadiť inštancie Debezium konektorov do Kafka Connect a to konkrétne MySQL a Postgres konektory, nakoľko sú sledované dáta v týchto DBMS. Posledným krokom je konfigurácia aspoň jedného *sink* konektoru, ktorý bude spracovávať dané topiky v Apache Kafka a odosielať ich inému systému (konzumentovi). Na konkrétnom príklade je použitý Elasticsearch konektor nakoľko je konzumentom Elasticsearch.



Obr. 2.4: CDC topológia z Kafka Connect

2.2.4 Štruktúra správy

Ako už bolo spomenuté, správy v Kafke obsahujú kľúč, v prípade Debezia je to primárny kľúč v tabuľke, a hodnotu, ktorá má komplexnejšiu štruktúru skladajúcu sa z:

- **before** stavu, ktorý v sebe nesie predchádzajúci stav data, ktoré sa mení. V prípade, že nastane *insert* event, táto hodnota bude prázdna, nakoľko práve vzniká a nemá žiadny predchádzajúci stav.
- **after** stavu, ktorý v sebe nesie nový stav dát. Táto hodnota môže byť opäť prázdna a to v prípade *delete* udalosti.
- **source** informácie, ktoré obsahujú metadáta o pôvode danej zmeny. Tieto dáta sú závislé na type databáze, ktorá sa sleduje. Pre MySQL databázu sa napríklad skladá z informácií ako meno databázového serveru, názvu a pozície logovacieho súboru, z ktorého číta, názvu databáze a tabuľky, timestamp a pod.

[7] <https://www.confluent.io/product/connectors/>

Kafka dokáže spracovávať akýkoľvek druh textových a binárnych dát, takže jej na tejto logickej štruktúre nezáleží. Na odosielanie správ sa používajú konvertory, ktoré prevádzajú správu do formy v ktorej bude odosielaná. Použitie Kafka Connect prináša možnosť využiť dostupné konventory, ktoré pokytuje. Pre Debezium to sú:

- **JSON**, do ktorého je možnosť zahrnúť informácie o schéme dát, na základe ktorej môžu konzumenti správne interpretovať prijatú správu. Tento formát je výhodné používať počas vývoja aplikácie nakoľko je čitateľný pre človeka. Ukážku správy vo formáte JSON je možné zhliadnuť v prílohe B.1.
- **Avro**, ktorý má veľmi efektívnu a kompaktnú reprezentáciu vhodnú na produkčné účely. Takáto správa nie je vo forme, aby ju človek bez úprav mohol prečítať, nakoľko je to binárna reprezentácia správy. V týchto správach sa nenachádza informácia o schéme tabuľky, ale iba identifikátor na danú schému a jej verziu, ktorú je možné získať pomocou registru schémat, čo je ďalšia časť ekosystému Kafky. Konzument môže získať konkrétnu schému z registrov a na základe nej interpretovať binárne dáta, ktoré dostal.

Kapitola 3

Aktuálne riešenie MySQL konektoru

Projekt Debezium sa skladá z viacerých častí. Hlavnou časťou je modul systému, ktorý je spoločný pre všetky typy konektorov podporovaných Debeziom. Tento modul zaobstaráva základnú funkcionalitu spojenú s CDC, ktorú tento systém podporuje. Definuje model sledovaných dát, na základe ktorých si systém udržuje aktuálne schéma tabuliek a ich dátový stav v pamäti. Jedným z týchto podporovaných konektorov je aj konektor pre MySQL databázu.

3.1 MySQL konektor

Minimálnou podporovanou verziou MySQL je aktuálne verzia 5.6. MySQL konektor Debezia dokáže sledovať zmeny v databáze na úrovni jednotlivých riadkov v tabuľkách pomocou čítania databázového binlogu (3.1.1). Pri prvom pripojení na MySQL server si konektor vytvorí aktuálny obraz všetkých tabuliek (3.1.2) a následne sleduje všetky komitnuté zmeny, na základe ktorých vytvára jednotlivé *insert*, *update* a *delete* eventy. Pre každú tabuľku je vytvorený separátny topik v Kafke, v ktorom sa ukladajú eventy spojené z danou tabuľkou. Týmto spôsobom je zabezpečený štart s konzistentným obrazom všetkých dát.

Konektor je taktiež veľmi tolerantný voči chybám. Zároveň s čítaním udalostí z binlogu si konektor ukladá ich pozíciu. Ak by nastala akákoľvek situácia, pri ktorej by konektor prestal pracovať, a bol by nutný jeho reštart, tak jednoducho začne čítanie binlogu na pozícii na ktorej skončil pred pádom. Konektor sa bude rovnako správať aj keby chyba a jeho pád nastali počas prvotného vytvárania aktuálneho obrazu.

3.1.1 Binárny log

v MySQL je možné implementovať CDC na základe sledovania binárneho logu v skratke nazývaného binlog[26]. Binlog obsahuje všetky udalosti, ktoré popisujú zmeny vykonané nad MySQL databázou ako napríklad vytváranie tabuliek alebo zmena dát. Poradie týchto udalosti je zachované voči reálnemu poradiu, ako boli SQL dotazy vykonávané. Toto binárne logovanie sa využíva na dva základne účely:

- Pre **replikáciu**, kde binlog na hlavnom (master) replikačnom serveri sprostredkováva záznamy o zmenách, ktoré majú byť odoslané na vedľajší (slave) server. Master server

odošle udalosti nachádzajúce sa v binlogu slave serveri, ktorý tieto udalosti vykoná u seba za účelom udržania rovnakého dátového stavu ako je na master replikačnom serveri.

- Pre **obnovu systému z chybového stavu** anglicky nazývanú **recovery**, počas ktorej je potrebné nahrať zálohu databáze. Aby sa nestratili zmeny v databáze, ktoré nastali po vytvorení zálohy, sa po nahraní zálohy znovu spustia události, ktoré tieto zmeny v binlogu popisujú. Tým sa zabezpečí konzistentný stav databázových dát z dátami v dobe zlyhania databázového servera.

Binárny log neobsahuje udalosti, ktoré nemajú žiadny efekt na dáta ako napríklad SELECT alebo SHOW. Udalosti môžu byť do logu zapisované v rôznych formátoch, na základe ktorých sa mení aj spôsob replikácie dát. Tieto formáty logovania sú:

- **Statement-based** logovanie, v ktorom udalosti obsahujú SQL dotazy, ktoré produkujú zmeny v dátach (INSERT, UPDATE, DELETE). V rámci tohto logovanie môže taktiež obsahovať dotazy, ktoré môžu iba potencionálne meniť dáta napríklad DELETE dotaz, ktorý sa nespáruje so žiadnymi dátami. Pri replikácií slave server číta binlog a zaradom vykonáva SQL dotazy, ktoré obsahujú jednotlivé udalosti.

- **Row-based** logovanie, v ktorom udalosti popisujú zmeny pre jednotlivé riadky v tabuľkách. Pri replikácií sa kopírujú udalosti, ktoré reprezentujú zmeny riadkov v tabuľkách na slave serveri.

Pre účely CDC v Debeziu je používané row-based logovanie, nakoľko zalogované udalosti obsahujú zmeny pre konkrétne riadky v tabuľkách, a tým pádom nie je nutné dopočítavať dáta, ktoré by boli na základe daného dotazu zmenené. Master server sa snaží ukladať do binlogu iba kompletné a vykonané transakcie, no bohužiaľ prax ukázala, že existujú aj výnimky. To znamená, že výskyt syntakticky nesprávnych dotazov je možný, ale zároveň veľmi zriedkavý. V MySQL konektoru teda nie je nutné sledovať korektnosť parsovaných dotazov.

Pomocou SQL dotazu *SHOW BINARY LOGS* je možné zobraziť aktuálne existujúce binárne logy na serveri. Následne dotazom *SHOW BINLOG EVENTS [IN 'log_name'] [FROM pos] [LIMIT [offset,] row_count]* je možné sledovať informácie o všetkých udalostiach obsiahnutých v danom binlogu ako sú napríklad typ udalosti, jeho začiatočná a jeho konečná pozícia v logu. Na čítanie a spracovávanie binlogu ponúka MySQL nástroj *mysqlbinlog*, ktorý je možné spustiť príkazom *mysqlbinlog [options] log_file*. Prvý riadok udalosti vždy obsahuje prefix *# at* za ktorým nasleduje číslo reprezentujúce pozíciu udalosti v binlogu. Podľa základného nastavenia MySQL zobrazuje mysqlbinlog udalosti týkajúce sa zmien na úrovní riadkov zakódované ako base-64[1] použitím interného príkazu *BINLOG*. Aby bolo možné vidieť tento pseudokód je možné použiť prepínač *—verbose* alebo *-v*. Na výstupe bude možné vidieť tento pseudokód na riadkoch, ktoré budú začínať prefixom *###*. Použitím prepínača *—verbose* alebo *-v* dvakrát, môžeme nastaviť aj zobrazovanie dátových typov a iných metadát pre každý stĺpec. Aby sa v logu nezobrazoval interný príkaz *BINLOG* a zakódovaná hodnota udalosti, je možné použiť prepínač *—base64-output=DECODE-ROWS*. Kombináciou týchto

[1] Typ kódovania, ktorý prevádza binárne dáta na postupnost znakov

prepínačov získame možnosť pohodlne sledovať obsah udalosti týkajúcich sa zmien v dátach. [26]

Pre Debezium sú dôležité udalosti typu:

- **Query**, v ktorom sa objavujú dotazy na zmenu štruktúry databáze (DDL) ako je možné vidieť na poslednom riadku príkladu udalosti 3.1.
- **Table_map**, pomocou ktorého binlog mapuje konkrétne tabuľky na identifikátor, ktorým sa následne na tieto tabuľky odkazuje. Príklad takejto udalosti je možné vidieť v príklade 3.2 na pozícii 552 a jeho následné použitie pre udalosť na pozícii 621.
- **Update_rows**, ktorý obsahuje informácie o zmene dát na úrovni riadkov, ako je možné vidieť na udalosti v príklade 3.2 na pozícii 621.
- **Write_rows**, ktorý obsahuje informácie o novo vzniknutých dátach.
- **Delete_rows**, ktorý obsahuje informácie o zmazaných dátach.

Ukážka 3.1: Query udalosť z binárneho logu MySQL

```
1  # at 219
2  #180213 9:59:15 server id 223344 end_log_pos 408 CRC32 0x19237396 Query thread_id
       ↪ =15 exec_time=0 error_code=0
3  use ‘inventory‘/*!*/;
4  SET TIMESTAMP=1518515955/*!*/;
5  SET @@session.pseudo_thread_id=15/*!*/;
6  SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.
       ↪ unique_checks=1, @@session.autocommit=1/*!*/;
7  SET @@session.sql_mode=1436549152/*!*/;
8  SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
9  /*!\C utf8 *//*!*/;
10 SET @@session.character_set_client=33,@@session.collation_connection=33,@@session.
       ↪ collation_server=8/*!*/;
11 SET @@session.lc_time_names=0/*!*/;
12 SET @@session.collation_database=DEFAULT/*!*/;
13 /* ApplicationName=IntelliJ IDEA 2017.2 */ alter TABLE customers add column
       ↪ phone_number varchar(15) NULL
```

Ukážka 3.2: Table_map a Update_rows udalosti z binárneho logu MySQL

```
1  # at 552
2  #180219 11:51:22 server id 223344 end_log_pos 621 CRC32 0x622e3e17 Table_map: ‘
       ↪ inventory‘.‘customers‘ mapped to number 109
3  # at 621
4  #180219 11:51:22 server id 223344 end_log_pos 736 CRC32 0x8ec54ac4 Update_rows:
       ↪ table id 109 flags: STMT_END_F
5  ### UPDATE ‘inventory‘.‘customers‘
6  ### WHERE
```

```
### @1=1001 /* INT meta=0 nullable=0 is_null=0 */
### @2='Sally' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
### @3='Thomas' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
### @4='sally.thomas@acme.com' /* VARSTRING(255) meta=255 nullable=0 is_null
    ↪ =0 */
### @5=NULL /* VARSTRING(15) meta=15 nullable=1 is_null=1 */
### SET
### @1=1001 /* INT meta=0 nullable=0 is_null=0 */
### @2='John' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
### @3='Thomas' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
### @4='sally.thomas@acme.com' /* VARSTRING(255) meta=255 nullable=0 is_null
    ↪ =0 */
### @5=NULL /* VARSTRING(15) meta=15 nullable=1 is_null=1 */
```

3.1.2 Aktuálny obraz tabuliek

Po nakonfigurovaní a prvom spustení MySQL konektoru sa podľa základného nastavenia spustí tvorba aktuálneho obrazu tabuliek sledovanej databáze. Pri každom vytváraní aktuálneho obrazu, konektor postupuje podľa týchto krokov[5]:

1. Aktivuje globálny zámok čítania (read lock) aby zabránil ostatným databázovým klientom v zapisovaní.

2. Spustí transakciu s izoláciou na opakované čítanie (repeatable read)[2], aby všetky nasledujúce čítania v rámci tejto transakcie boli voči jednému konzistentnému obrazu.

3. Prečíta aktuálnu pozíciu binlogu.

4. Prečíta schéma databáz a tabuliek na základe konfigurácie konektoru.

5. Uvoľní globálny zámok, aby ostatný databázový klienti mohli znovu zapisovať do databáze.

6. Voliteľne zapíše zmeny DDL do Kafka topiku vrátane všetkých potrebných SQL dotazov.

7. Skontroluje všetky databázové tabuľky a vygeneruje príslušné *create* udalosti Kafka topiky pre jednotlivé riadky v tabuľkách.

8. Potvrdí transakciu.

9. Do konektorového offsetu zaznamená, že úspešne ukončil vytváranie obrazu.

[2]Stupeň izolácie založený na používaní *read* a *write* zámkoch, ktorý ale nezabráni prítomnosti fantómov vznikajúcich v stiuácii, keď v jednej transakcii podľa rovnakého dotazu čítame dáta 2x z rôznymi výsledkami, pretože v medzičase stihla iná transakcia vytvoriť alebo zmazať časť týchto dát.

Transakcia vytvorená v druhom kroku nezabráni ostatným klientom upravovať dáta, ale poskytne konektoru konzistentný a nemenný pohľad na dáta v tabuľkách. Nakoľko transakcia nezabráni klientom aplikovať DDL zmeny, ktoré by mohli vadiť konektoru pri čítaní pozície a schémat v binlogu, je nutné v prvom kroku použiť globálny zámok na čítanie k zamedzeniu tohto problému. Tento zámok je udržiavaný na veľmi krátku dobu potrebnú pre konektor na vykonanie krokov tri a štyri. V piatom kroku je tento zámok uvoľnený predtým, než konektor vykoná väčšinu práce pri kopírovaní údajov.

3.2 DDL parser

Pri čítaní binárneho logu MySQL konektor parsuje DDL dotazy na základe ktorých si v pamäti vytvára modely schémat každej tabuľky podľa toho ako sa vyvíjali v čase. Tento proces je veľmi dôležitý, pretože konektor generuje udalosti pre tabuľky, v ktorých definuje schéma tabuľky v čase, kedy daná udaloť vznikla. Aktuálne schéma sa nemôže použiť, nakoľko sa môže zmeniť v danom čase prípadne na danej pozícii v logu na ktorej konektor číta.

Konektor produkuje správy použitím Kafka Connect Schemas, ktoré definujú jednoduchú dátovú štruktúru obsahujúcu názvy, typy polí a spôsob organizácie týchto polí. Pri generovaní správy na udaloť týkajúcu sa dátovej zmeny je najprv nutné mať Kafka Connect Schema objekt, v ktorom definujeme všetky potrebné polia. Následne je nutné konvertovať usporiadané pole hodnôt stĺpcov do Kafka Connect Struct objektu na základe polí a ich hodnôt z odchytenej udalosti.

Ak Debezium konektor odchytí DDL udalost, stačí mu aktualizovať model, ktorý si drží v pamäti a ten následne použiť na generovanie Schema objektu. V rovnakom čase vytvorí komponentu, ktorá bude používať tento Schema objekt na vytváranie Struct objektu z hodnôt v odchytenej udalosti. Tento proces sa vykoná raz a použije sa na všetky DML údálosti až do doby pokiaľ sa neodchytí ďalší DDL dotaz, po ktorom bude opäť nutné aktualizovať model v pamäti.

Nato aby bolo možné túto akciu vykonať je nutné parsovať DDL dotazy, pričom pre potreby Debezia stačí vedieť rozpoznať iba malú časť z celej DDL gramatiky. Model, ktorý sa udržiava v pamäti a zbytok funkcionality spojený z generovaním Schema objektu a konvertoru hodnôt na Struct objekt je generické nakoľko nie je priamo spojené z MySQL.

3.2.1 Framework na parsovanie DDL

Keďže Debezium nenašlo žiadnu použiteľnú knižnicu na parsovanie DDL, rozhodlo sa implementovať vlastný framework podľa ich potrieb, ktoré sú[30]:

- Parovanie DDL dotazov a aktualizácia modelu v pamäti.
- Zameranie sa na podstatné dotazy ako sú *CREATE*, *UPDATE* a *DROP* tabuliek, pričom sa ostatné dotazy budú ignorovať bez nutnosti ich parsovať.
- Štruktúra kódu parsru, ktorá bude podobná dokumentácii MySQL DDL gramatiky a názvoslovie metód, ktoré bude odzrkadľovať pravidlá gramatiky. Takúto implementáciu je jednoduchšie udržiavať v priebehu času.

- Umožniť vytvorenie parserov pre PostreSQL, Oracle, SQLServer a všetkých ostatných DBMS, ktoré budú potrebné.
- Umožniť prispôsobenie pomocou dedičnosti a polymorfizmu.
- Uľahčiť vývoj, ladenie a testovanie parserov.

Výsledný framework pozostáva z tokenizeru, ktorý konvertuje DDL dotaz v jednom reťazci na sekvenciu tokenov. Každý token reprezentuje slová a symboly, citované reťazce, kľúčové slová, komentáre a ukončujúce znaky ako napríklad bodkočiarku pre MySQL. DDL parser prechádza sled tokenov a volá metódy na spracovanie variácii sady tokenov. Parser taktiež využíva interný DataTypeParser na spracovanie dátových typov SQL, ktoré si je možné pre jednotlivé DBMS ručne zaregistrovať.

MySqlDdlParser trieda dedí od základnej triedy DdlParser a sprostredkováva celú parsovaciu logiku špecifickú pre MySQL. Napríklad DDL dotaz 3.3 je možné zparsovať podľa ukážky 3.4.

Ukážka 3.3: DDL dotaz v MySQL

```
1  # Create and populate our products using a single insert with many rows
2  CREATE TABLE products (
3    id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
4    name VARCHAR(255) NOT NULL,
5    description VARCHAR(512),
6    weight FLOAT
7  );
8  ALTER TABLE products AUTO_INCREMENT = 101;
9
10 # Create and populate the products on hand using multiple inserts
11 CREATE TABLE products_on_hand (
12   product_id INTEGER NOT NULL PRIMARY KEY,
13   quantity INTEGER NOT NULL,
14   FOREIGN KEY (product_id) REFERENCES products(id)
15 );
```

Ukážka 3.4: Parsovanie dotazu pomocou MySqlDdlParseru

```
1 String ddlStatements = "..."
2 DdlParser parser = new MySqlDdlParser();
3 Tables tables = new Tables();
4 parser.parse(ddl, tables);
```

Tables objekt reprezentuje model uložený v pamäti konkrétnej databáze. Parser zprocesuje jednotlivé DDL dotazy a aplikuje ich na odpovedajúce definície tabuliek nachádzajúce sa v Tables objekte.

3.2.2 Implementácia MySQL DDL parsru

Každá implementácia DdlParser implementuje metódu, ktorá parsuje DDL dotazy poskytnuté v reťazci. Táto metóda vytvára nový TokenStream pomocou DdlTokenizer, ktorý

rozdelí znaky v reťazci do typovaných Token objektov. Následne volá ďalšiu parsovaciu metódu v ktorej nastaví lokálne premenné a snaží sa zaradom parsovať DDL dotazy do doby, kým žiadny ďalší nenájde. Ak by počas parsovania nastala chyba napríklad že by sa nenašla zhoda, parser vygeneruje ParsingException, ktorá obsahuje riadok, stĺpec a chybovú správu oznamujúcu aký token bol očakávaný a aký sa našiel. V prípade chyby sa TokenStream pretočí na začiatok, aby sa prípadne mohla použiť implementácie iného parseru.

Pri každom volaní metódy parseNextStatement je predávaný objekt Marker, ktorý ukazuje na začiatočnú pozíciu parsovaného dotazu. Vďaka polymorfizmu MySqlDdlParser prepisuje implementáciu parseNextStatement metódy (ukážka C.2), v ktorej kontroluje, či prvý token vyhovuje niektorému z typov MySQL DDL gramatiky. Po nájdení vyhovujúceho tokenu sa zavolá odpovedajúca metóda na ďalšie parsovanie.

Pre príklad si predstavme, že by parser chcel parsovať dotaz začínajúci na *CREATE TABLE ...*. Prvým parsovaným slovom je *CREATE*, čím sa podľa ukážky z kódu C.2 zavolá metóda parseCreate. V nej sa toto slovo skonzumuje a rovnakým spôsobom nastane kontrola druhého slova, kde sa po vyhodnotení hodnoty *TABLE* zavolá metóda parseCreateTable (ukážka C.3). Táto metóda odzrkadľuje nasledujúce pravidlá MySQL gramatiky pre *CREATE TABLE*:

CREATE [**TEMPORARY**] **TABLE** [**IF NOT EXISTS**] tbl_name
(create_definition,...)
[table_options]
[partition_options]

CREATE [**TEMPORARY**] **TABLE** [**IF NOT EXISTS**] tbl_name
[(create_definition,...)]
[table_options]
[partition_options]
select_statement

CREATE [**TEMPORARY**] **TABLE** [**IF NOT EXISTS**] tbl_name
{ **LIKE** old_tbl_name | (**LIKE** old_tbl_name) }

create_definition:
...

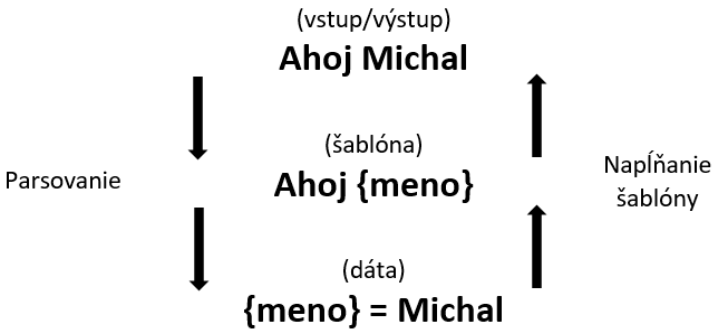
Metóda parseCreateTable sa snaží najskôr skonzumovať nepovinné slovo *TEMPORARY*, potom slovo *TABLE*, nepovinný fragment *IF NOT EXISTS* a následne konzumuje a parsuje názov tabuľky. Ak by dotaz obsahoval fragment *LIKE otherTable*, tak sa použije objekt Tables, z ktorého sa získa definícia odkazovanej tabuľky. V ostatných prípadoch sa na úpravu stávajúcej tabuľky použije TableEditor objekt. Takýmto spôsobom parser pokračuje vo svojej činnosti ďalej a snaží sa parsovať dotaz na základe pravidiel gramatiky.

Kapitola 4

Syntaktická analýza

Syntaktickou analýzou (slangovo z angličtiny tiež **parsovaním**) sa v teórii rozumie konštrukcia derivačného stromu vety bezkontextového jazyka[24] popísaného v kapitole 4.1.3. Program, ktorý vykonáva túto úlohu sa volá syntaktický analyzátor (slangovo **parser**). Počas konštrukcie derivačného stromu parser zachováva hierarchické usporiadanie symbolov, ktoré je vhodné pre ďalšie spracovanie.

Parsovanie je taktiež možné si predstaviť ako inverziu k napĺňaniu šablón. Šablóna definuje napríklad štruktúru textu s variabilnými premennými, ktoré je treba naplniť dátami a parsovanie identifikuje túto šablónu a extrahuje dáta, ktoré boli do nej vložené.



Obr. 4.1: Príklad parsovania a naplňovania šablóny

Podstata parsovania je veľmi dôležitá, pretože rôzne entity potrebujú dáta na spracovanie v rôznych formátoch. Parsovanie umožňuje transformovať získané data tak, aby im mohol porozumieť špecifický software.

4.1 Teória parsovania

Na správne pochopenie problému parsovania je nutné si najprv zadefinovať základné pojmy, ktoré sú s ním spojené. Teória parsovania je postavená na teórii jazykov, gramatík a automatov, z ktorej najdôležitejšie pojmi sú definované v rámci tejto sekcie.

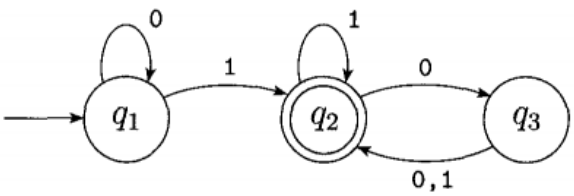
4.1.1 Deterministický konečný automat

Konečné automaty sa používajú v rôznych oboroch ako napríklad pri prekladačoch, spracovávaní prirodzeného jazyka, pri návrhu hardwaru a ďalších [6]. Predstavujú model systémov, ktoré rozpoznajú, či je vstupný reťazec patrí do jazyka. Deterministický konečný automat (DFA), taktiež aj *akceptor* je najpoužívanejší zo štyroch typov automatov.

Definícia 4.1.1. *Deterministický konečný automat* M je pätica $M = (Q, \Sigma, \delta, q_0, F)$, kde

- Q je konečná množina stavov
- Σ je konečná množina vstupných symbolov
- δ je prechodová funkcia $\delta : Q \times \Sigma \to Q$
- q_0 je počiatočný stav
- $F \subseteq Q$ je množina koncových stavov [6]

Konečný automat je možné prehľadne znázorniť formou stavového diagramu. *Stavový diagram* je orientovaný graf, v ktorom sú uzly ohodnotené stavmi automatu a hrany vstupnými symbolmi automatu. Z uzlu q vedie hrana ohodnotená symbolom a do uzlu p vtedy, ak $\delta(q, a) = p$. Počiatočný stav sa označuje šípkou, ktorá neprichádza zo žiadneho iného stavu a uzly ohodnotené koncovými stavmi označujeme dvojitým krúžkom. Príklad takto znázorneného DFA je na obrázku 4.2.



Obr. 4.2: Príklad DFA znázorneného pomocou stavového diagramu

Definícia 4.1.2 (Jazyk prijímaný konečným automatom). Je daný DFA $M = (Q, \Sigma, \delta, q_0, F)$. Slovo $u \in \Sigma^*$ je *prijímané* automatom M práve vtedy, keď

$$\delta^*(q_0, u) \in F.$$

Množina všetkých slov, ktoré automat prijíma sa nazýva *jazyk prijímaný* M a značíme ju $L(M)$. Platí teda

$$L(M) = \{\omega | \delta^*(q_0, u) \in F\}.[6]$$

Každý jazyk L, pre ktorý existuje deterministický konečný automat prijímajúci tento jazyk, sa nazýva **regulárny jazyk**.

4.1.2 Regulárny výraz

Regulárny výraz je ďalšia možnosť ako popísať regulárne jazyky, ktoré sú uzatvorené vzhľadom k operáciám zjednotenia, súčinu a iterácie. Regulárne výrazy sú postavené na Kleeneho operátore(*), ktorý sa používa na označenie, že určitý prvok môže byť prítomný nula alebo nekonečne veľa krát.

Definícia 4.1.3 (**Regulárne výrazy nad abecedou**). Je daná abeceda Σ. Množina všetkých regulárnych výrazov nad Σ je definovaná induktívne:

- $\emptyset$ je regulárny výraz,
- ϵ je regulárny výraz,
- $\mathbf{a}$ je regulárny výraz pre každé písmeno $a \in \Sigma$,
- pokiaľ sú $\mathbf{r}_1$ a $\mathbf{r}_2$ regulárne výrazy, tak $\mathbf{r}_1 + \mathbf{r}_2$, $\mathbf{r}_1\mathbf{r}_2$ a $\mathbf{r}_1^*$ sú regulárne výrazy. [6]

Podpora regulárnych výrazov je dostupná u väčšiny programovacích jazykov. Pre zjednodušenie zápisu sú definované viaceré znaky, ktoré vychádzajú zo spomenutých základných operácií. Ich najčastejšie využitie je na vyhľadávanie v texte.

4.1.3 Bezkontextový jazyk

Bezkontextový jazyk je jazyk nad abecedou, ktorý je prijímaný bezkontextovou gramatikou (CFG). Gramatikou sa rozumie súpis pravidiel, ktoré určujú ako vygenerovať všetky slová daného jazyka. CFG reprezentuje silnejšiu metódu popisovania jazykov, pomocou ktorej je možné opísať vlastnosti, ktoré majú rekurzívnu štruktúru.

Definícia 4.1.4. *Bezkontextová gramatika* je usporiadaná štvorica $\mathcal{G} = (N, \Sigma, S, P)$, kde

- N je konečná množina tzv. *neterminálov*[1]
- Σ je konečná neprázdna množina tzv. *terminálov*[2], kde platí $N \cap \Sigma = \emptyset$
- $S \in N$ je *štartovací symbol*
- P je konečná množina pravidiel typu $\alpha \to \beta$, kde α a β sú slová nad $N \cup \Sigma$ taká, že α obsahuje aspoň jeden neterminál.
- každé pravidlo P je v tvare $A \to \gamma$, kde $\gamma \in (n \cup \Sigma)*$ a A je neterminál [7]

Bezkontextové gramatiky sa prvýkrát používali pri štúdií ľudských jazykov na pochopenie vzťahu medzi podstatným menom, slovesom a predložkou. Ich kombináciou vznikajú frázy, ktoré vedú k prirodzenej rekurzii, nakoľko podstatné meno môže byť súčasťou slovesnej frázy a pod. Bezkontextové gramatiky dokážu zachytiť dôležité aspekty týchto vzťahov [32].

Špecifikácia a kompilácia programovacích jazykov je jedným z použití CFG. Gramatika programovacieho jazyka sa často používa na pochopenie jeho syntaxe.

V nasledujúcom príklade je ukážka bezkontextovej gramatiky G_1.

[1] Premenné symmboli, ktoré sa reprezentjú pomocou velkých písmen
[2] Písmená vstupnej abecedy, často reprezentované malými písmenami, číslami alebo špeciálnymi symbolmi

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

Z týchto pravidiel je možné poskladať strom pravidiel, v ktorom je množina terminálov $\Sigma \in \{0, 1, \#\}$, množina neterminálov $N \in \{A, B\}$ a štartovací symbol je A.

Definícia 4.1.5 (**Derivácia**)**.** Je daná gramatika $\mathcal{G} = (N, \Sigma, S, P)$. Povedzme, že δ sa *odvodí* z γ vtedy, ak

- buď $\gamma = \delta$
- alebo existuje postupnosť priamych odvodení

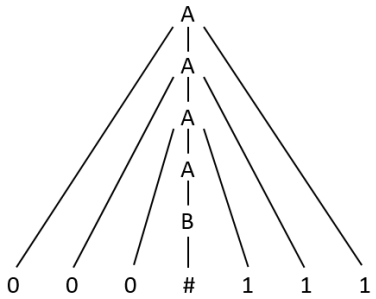
$$\gamma = \gamma_1 \Rightarrow_{\mathcal{G}} \gamma_2 \Rightarrow_{\mathcal{G}} ... \Rightarrow_{\mathcal{G}} \gamma_k = \delta$$

Tento fakt sa označuje $\gamma \Rightarrow^*_{\mathcal{G}} \delta$ a tejto konečnej postupnosti hovoríme *derivácia*. [7]

Pre príklad, gramatika G_1 generuje reťazec *000#111*. Derivácia tohto reťazca bude vyzerať nasledovne

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Rovnakú informáciu je možné reprezentovať graficky pomocou zparsovaného (derivačného) stromu. Príklad derivačného stromu je na obrázku 4.3.



Obr. 4.3: Derivačný strom gramatiky $\mathcal{G}_1$ pre reťazec *000#111*

Množina všetkých reťazcov, ktoré je možné generovať týmto spôsobom sa nazýva jazyk gramatiky L. Jednoduchým pohľadom na gramatiku G_1 je možné zapísať jazyk gramatiky ako $L(G_1) \in \{0^n\#1^n | n \geq 0\}$. Všetky jazyky generované bezkontextovou gramatikou sa nazývajú **bezkontextové jazyky**.

Definícia 4.1.6 (**Jazyk generovaný gramatikou**)**.** Povedzme, že slovo $\omega \in \Sigma^*$ je *generované* gramatikou $\mathcal{G}$, ak existuje derivácia $S \Rightarrow^*_{\mathcal{G}} \omega$.

Jazyk $L(\mathcal{G})$ *generovaný* gramatikou $\mathcal{G}$ sa skladá zo všetkých slov generovaných gramatikou $\mathcal{G}$, tj.

$$L(\mathcal{G}) = \{\omega \in \Sigma^* | S \Rightarrow^*_{\mathcal{G}} \omega\}.[7]$$

4.1.4 Backus-Naur Form notácia

Pri popisovaní jazyka mnohých programovacích jazykov, protokolov alebo formátov sa vo svojej špecifikácii používa zápis pomocou Backus-Naur Form (BNF) notácie.[21]

Každé pravidlo v BNF má nasledujúcu štruktúru:

$$<neterminál> ::= výraz$$

Všetky netreminály v BNF sa zapisujú do špicatých zátvoriek $< >$, či už sú použité na pravej alebo ľavej strane pravidla. Výraz sa môže obsahovať terminály aj neterminály a je definovaný ich spojením, alebo výberom. Symbolmi vo výraze postavené vedľa seba určujú postupnosť symbolov a použitie znaku vertikálnej lišty určuje výber zo symbolov.

4.1.5 Rozšírená Backus-Naur Form notácia

Pre zjednodušenie zápisu gramatiky, a aby bolo možné jednoduchšie definovať určité typy pravidiel, vznikla kolekcia rozšírení k Backus-Naur Form notácii (EBNF), ktorá bola štandardizovaná ako ISO/IEC 14997[17]. Terminály môžu byť vyjadrené konkrétnym postupom znakov v úvodzovkách, alebo pomocou triedy literálov, ktorú je možné zapísať pomocou regulárneho výrazu. Priraďovací znak pravidla je zmenený z ::= na jednoduché = a vynecháva sa zápis špicatých zátvoriek okolo neterminálov. Tieto malé syntaktické zmeny nie sú tak dôležité ako dodatočné operácie EBNF, ktoré sa môžu použiť vo výraze.

Nepovinnosť – Použitím hranatých zátvoriek okolo výrazu *[výraz]* sa indikuje možnosť použitia tohto výrazu v sekvencii. Jednoduchšie povedané, výraz môže, ale nemusí byť použitý vo výslednej sekvencii. Toto pravidlo je taktiež možné zapísať pomocou znaku ?. Príklad:

```
term = ["-"] factor
term = "-"? factor
```

Zlučovanie – Aby bolo možné identifikovať prioritu sekvencie symbolov, EBNF používa klasické zátvorky, čím jednoznačne definuje poradie výrazov. V príklade je zapísaná gramatika, ktorá prijíma matematické sčítanie a odčítanie:

```
expr = term ("+"| "-") expr
```

Opakovanie – Použitím zložených zátvoriek okolo výrazu *{výraz}* je možné indikovať opakovanie výrazu. To znamená, že výraz sa nemusí v sekvencii vyskytovať, ale zároveň môže byť nekonečne krát za sebou. Toto je pravidlo je taktiež možné zapísať pomocou znaku *. Príklad:

```
args = arg {"," arg}
args = arg ("," arg)*
```

Spájanie – Namiesto toho aby sa autor gramatiky spoliehal na postavenie výrazov vedľa seba, má možnosť spájať výrazy aj pomocou znaku čiarky.

Každú gramatiku zapísanú cez EBNF je možné taktiež zapísať pomocou BNF, to ale vedie k omnoho obsiahlejšiemu množstvu definičných pravidiel. V nasledujúcich príkladoch sú znázornené dva rôzne zápisy gramatiky v EBNF z kapitoly 4.1.3:

$$A = ("0"\ A\ "1") \mid B$$
$$B = "\#"$$

$$A = ("0")^*\ "\#"\ ("1")^*$$

4.1.6 Parsing Expression Grammar

Parsing Expression Grammar (PEG) poskytuje alternatívu na popisovanie strojovo orientovanej syntaxi, ktorý rieši problém nejednoznačnosti tým, že ju nepodporuje už od začiatku. Zápis PEG je veľmi podobný zápisu gramatiky pomocou EBNF. Taktiež priamo podporuje veci, ktoré sa bežne používajú, ako sú rozsahy znakov (triedy znakov). Má aj niektoré rozdiely, ktoré v skutočnosti nie sú pragmatické, ako napríklad použitie formálnejšieho symbolu šípky (←) pre priradenie, namiesto bežnejšieho symbolu rovníc (=).

Problém nejednoznačnosti spočíva v možnosti zparsovať jeden vstupný reťazec viacerými spôsobmi. Ak by CFG parser spracovával takýto reťazec, mal by v takomto prípade problém. Nakoľko CFG spracováva možnosti pravidiel nedeterministicky, pri parsovaní nejednoznačného vstupu vráti chybu, pretože nevie ktorá zparsovaná možnosť je správna. Na druhú stranu pre PEG riadi výber možností pomocou *prioritnej voľby*, a preto pri parsovaní nejednoznačného vstupu vždy použije prvú možnosť, ktorá je akceptovateľná [13]. Nevýhoda tohto prístupu je v tom, že pri písaní PEG je potreba dbať na správne poradie, inak by mohli vzniknúť pravidlá, ktoré nikdy nebudú vyhovovať. V nasledujúcom príklade slovo *doge* nebude nikdy vyhovovať, nakoľko slovo *dog* je na prvom mieste a bude ihneď vybrané.

$$\text{word} \leftarrow \text{'dog'} \ / \ \text{'doge'}$$

4.2 Parsovanie pomocou regulárnych výrazov

Regulárne výrazy (4.1.2) poskytujú možnosť zápisu regulárnych jazykov, ktorých fungovanie je postavené na deterministických konečných automatoch (4.1.1).

O regulárnych výrazoch sa často hovorí, že by nemali byť použité na parsovanie. Nie vždy to ale je pravda, pretože je možné použiť regulárne výrazy na parsovanie jednoduchých vstupov. Niektorí programátori nepoznajú iné možnosti a snažia sa všetko parsovať s použitím regulárnych výrazov aj keď by nemali. Výsledkom toho je séria regulárnych výrazov spojených v jeden, čím sa parsovanie môže jednoducho stať vysoko náchylným k chybám.

Parsovanie pomocou regulárnych výrazov je naozaj možné, ale iba pre regulárne jazyky. Pokiaľ sa v jazyku, ktorý sa snažíme parsovať, objavia vnorené alebo rekurzívne elementy, nejedná sa už o regulárny jazyk, ale o jazyk *bezkontextový* (4.1.3). Parsovanie takéhoto jazyka pomocou regulárneho výrazu by spôsobilo degenerovanú slučku [13].

4.3 Štruktúra bezkontextových parserov

Syntaktickej analýze spravidla predchádza *lexikálna analýza*, pri ktorej sa vstupný reťazec rozdeľuje na postupnosť lexikálnych symbolov (lexémov). V programovacích jazykoch sa taktiež nazývajú **tokeny** a definujú identifikátory, literály (čísla, reťazce), kľúčové slová,

operátory, oddeľovače a pod. Pre parser sú tokeny ďalej nedeliteľné stavebné jednotky, ktoré používa pri interpretácií vstupných dát. Program vykonávajúci túto úlohu sa nazýva štrukturálny analyzátor, no v programovaní sa častejšie narazí na výraz **lexer** alebo **tokenizer** bližšie popísaný v kapitole 4.3.1.

V kontexte parsovania sa slovo parser môže odkazovať na program, ktorý vykonáva celý proces ale aj na správny parser (syntaktický analyzátor), ktorý analyzuje tokeny vytvorené lexerom. Dôvodom toho je, že parser sa stará o najdôležitejšiu a najťažšiu časť celého procesu parsovania. Lexer hrá v procese parsovania iba úlohu pomocníka na uľahčenie práce parseru.

Parsre sú významnou súčasťou kompilátorov alebo interpreterov programovacích jazykov, no samozrejme môžu byť súčasťou aj rôznych typov programov. Čo sa týka parsovania programovacích jazykov, parser dokáže určiť iba syntaktickú korektnosť parsovaného výrazu. Výstup parseru je ale základom pre zistenie sémantickej korektnosti.

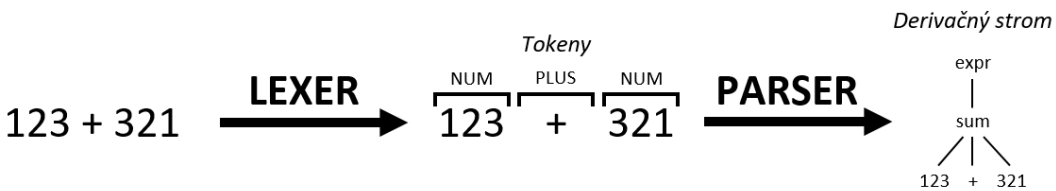
4.3.1 Lexer

Lexery zohrávajú dôležitú rolu pri parsovaní, pretože transformujú počiatočný vstup na jednoduchšie spracovateľnú formu pre parser. Napísanie gramatiky pre lexer je zvyčajne jednoduchušie, nakoľko nie je nutné riešiť vymoženosti bezkotextového jazyka ako je napríklad opakovanie, rekurzia a podobne.

Jedna z veľmi dôležitých úloh lexera je vysporiadanie sa z medzerami v parsovanom výraze. Vo väčšine prípadoch chceme, aby prázdne medzery boli lexerom odstránené. Ak by sa tak nestalo, znamenalo by to, že by sa s nimi musel vysporiadať samotný parser. To by znamenalo ich kontrolu pri každom jednom použitom tokene, čo by sa rýchlo stalo nepríjemným.

Existujú prípady, kedy to nemôžeme urobiť, pretože medzery sú pre daný jazyk relevantné, ako napríklad v prípade Pythonu, kde sa používa identifikácia bloku kódu a je nutné určiť, ktoré medzery sú pre parser dôležité. Aj napriek tomu je zvyčajne lexer zodpovedný za riešenie problému, ktorá medzera je relevantná a ktorá nie. Napríklad pri parsovaní Pythonu chceme, aby lexer overil, či medzery definujú odsadenie (relevantná) alebo medzery medzi slovami (irelevantná). [33]

Lexer prečíta vstupný reťazec a rozdelí ho na predom definované typy tokenov. Na definíciu týchto typov sa používajú regulárne výrazy, nakoľko rozdelenie na tokeny spadá pod problém regulárnej gramatiky. Ako už bolo spomenuté na spracovanie regulárnej gramatiky sa používa algoritmus pre DFA(4.1.1).



Obr. 4.4: Spracovanie reťazca *123 + 321* lexerom a parserom

Pre príklad z obrázku 4.4 máme dva typy tokenov. **NUM** vyjadrujúci akékoľvek prirodzené číslo a **PLUS** vyjadrujúci znak súčtu (+). Keď sa lexer bude snažiť analyzovať reťazec **123 + 321**, bude čítať znaky *1,2,3* a potom znak medzery. V tomto momente lexer rozpozná, že postupnosť znakov *123* súhlasí z definíciou tokenu typu NUM. Následne prečíta znak *+*, ktorý sa zhoduje s druhým typom tokenu PLUS a nakoniec objaví posledný token typu NUM. Takto definované tokeny použije parser na vyhodnotenie výsledného výrazu. Bezkontextová gramatika pre takýto parser by mohla vyzerať nasledovne:

$$\textbf{sum} = \text{NUM } \{\text{PLUS NUM}\}$$

Vzhľadom na to, že lexery sú takmer výlučne používané v spojení s parsermi, je nutné si určiť hranicu, kde končí práca lexeru a kde začína práca parseru. Táto hranica nemusí byť vždy jasná a všetko to závisí na konkrétnej potrebe programu, pre ktorý je parser vytváraný. Pre príklad si môžeme predstaviť program, ktorý parsuje vstup obsahujúci IP adresu. Pokiaľ programu stačí poznať hodnotu IP adresy, tak je možné vytvoriť token v lexeru, ktorý popisuje celý formát IP adresy a parser pri svojej analýze použije iba tento token.

IPv4 = [0-9]+ "." [0-9]+ "." [0-9]+ "." [0-9]+

Ak by bol ale problém zložitejší a program by chcel analyzovať IP adresu a zistiť z nej informácie, ako napríklad krajinu, bude parser potrebovať jednotlivé hodnoty IP adresy samostatne. V tomto prípade lexer rozdelí IP adresu na dva druhy tokenov (číslo a bodka).

```
/* Lexer */
DOT = "."
OCTEC = [0-9]+

/* Parser */
ipv4 = OCTET DOT OCTET DOT OCTET DOT OCTET
```

4.4 Typické problémy parsovania

Pri definovaní gramatiky pre parsre existuje niekoľko typických problémov, s ktorými sa jednotlivé parsre musia vysporiadať.

4.4.1 Chýbajúci token

Častým problémom v gramatikách sú chýbajúce resp. nedefinované tokeny. V niektorých gramatikách sa v rámci lexeru zadefinuje iba časť tokenov ako napríklad

```
/* Lexer */
NAME = [a-zA-Z]+

/* Parser */
greeting = "Hello" NAME
```

Token "Hello" nie je pre parser definovaný. Niektoré nástroje na parsovanie sa dokážu s týmto problémom vysporiadať tým, že si sami vygenerujú definíciu pre tieto tokeny, čím zároveň ušetria užívateľovi trochu času [33].

4.4.2 Pravidlá s ľavou rekurziou

V rámci bezkontextových gramatík sa často využíva ľavá rekurzia na definovanie zľava asociatívnych operácií. Tento problém sa najčastejšie rieši v rámci Top Down parserov (viď kapitola 5.1) s rekurzívnym zostupom (viď kapitola 5.1.1).

Pravidlá s ľavou rekurziou sú také pravidlá, ktoré začínajú s referenciu sami na seba, ako napríklad $A \rightarrow A\alpha$. Do toho problému spadá taktiež nepriama ľavá rekurzia, čo znamená že referencia na samého seba sa objaví v rámci iného pravidla, ako napríklad:

$$A \rightarrow B\alpha$$
$$B \rightarrow A\beta$$

Predpokladajme, že sa snažím zparsovať pravidlo A na danom mieste vo vstupnom reťazci. Ak by sme na to použili Top Down parser, ktorý pracuje z ľava do prava, našou prvou pod úlohou by bolo parsovať pravidlo A na tom istom mieste [22]. Takto sa okamžite dostávame do nekonečnej slučky. Rovnaký problém nastane aj s použitím gramatiky, ktorá obsahuje nepriamu ľavú rekurziu, kde sa do nekonečnej slučky dostaneme prechodom cez viac pravidiel.

V teórii, obmedzenie na bezkontextové gramatiky bez ľavej rekurzie nepridáva žiadne obmedzenie na jazyk, ktorý sa snažíme popísať gramatikou. Existujú pravidlá pomocou ktorých sa dá ľavá rekurzia odstrániť. V zásade každá bezkontextová gramatika obsahujúca ľavú rekurziu vie byť transformovaná na gramatiku bez ľavej rekurzie [22].

Note: moznost pridat priklad odstranovania lavej rekurzie

Kapitola 5

Parsovacie algoritmy

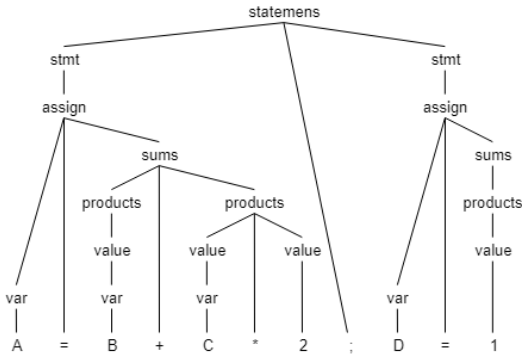
Teoreticky je parsovanie vyriešený problém, ale je to druh problému, ktorý sa stále a znovu rieši. To znamená, že existuje veľa rôznych algoritmov, každý so silnými a slabými bodmi a akademici ich stále zlepšujú.[33]

5.1 Obecný prehľad

Metód parsovania existuje veľmi veľa. Pre praktické použitie však majú význam metódy, ktoré je možné rozdeliť do dvoch skupín:[24]

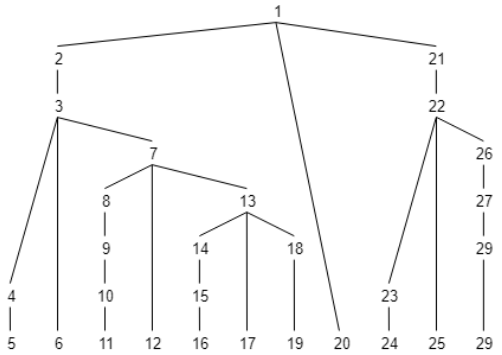
- Parsovanie **zhora dole (top down)** u ktorého sa parser najprv snaží identifikovať koreň parsovaného stromu, a potom postupuje dole cez podstromy až kým nenarazí na listy stromu. Top down metóda je najrozšírenejšia zo spomenutých dvoch a existuje na ňu niekoľko úspešných algoritmov, ktoré ju uplatňujú. Tieto algoritmy najčastejšie využívajú funkcionalitu lookahead.
- Parsovanie **zdola hore (bottom up)** u ktorej parser začína od najnižšej časti stromu, teda od listov a stúpa až do určenia koreňa stromu.

Vezmime si pre príklad parsovací strom na obrázku 5.1.

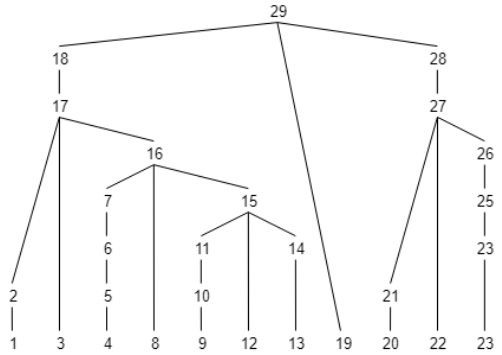


Obr. 5.1: Typický parsovací strom pre výraz $A = B + C * 2; D = 1$

Tento strom môže byť vygenerovaný oboma spomenutými metódami. Rozdiel bude iba v postupe jeho generovania. Na obrázkoch 5.2 a 5.3 je možné vidieť poradie krokov ako postupovali oba typy parserov pri vytváraní uzlov stromu.



Obr. 5.2: Postup generovania top down parseru



Obr. 5.3: Postup generovania bottom up parseru

Top down parsre sú jednoduchšie na poskladanie resp. implementáciu. Hoci sú *Bottom up* parsere bežne považované za "výkonnejšie" ako *Top down* parsre, ich trieda jazykov ktorú dokážu rozoznať je rovnaká[1]. Tento výraz sa skôr vzťahuje k väčšej flexibilite v písaní gramatiky, kde oproti *Top down* parserom nie je nutné dbať na pravidlá s ľavou rekurziou (viď kapitola 5.2.1). Aktuálne je situácia vyváženejšia, a to hlavne vďaka pokroku v stratégiách parsovania *Top down* metódou.

Nasledujúca tabuľka obsahuje súhrn hlavných vlastností existujúcich algoritmov a ich použitie[33].

Algoritmus	**Hlavné vlastnosti**	**Použtie**
CYK [19]	V najhoršom prípade zložitosť $O(n^3)$ Gramatika vyžaduje zápis CNF forme	Špecifické problémy
Earley [11]	V najhoršom prípade zložitosť $O(n^3)$, no zvyčajne je lineárna Dokáže spracovať všetky typy gramatík a jazykov	Generátory parserov, ktoré musia zvládať všetky typy gramatík
LL [28]	Jednoduchý na implementáciu Nie až tak schopný ako väčšina algoritmov (nepodporuje ľavú rekurziu) Historicky je najpopulárnejší	Ručne vytvárané parsre a generátory parsrov, ktoré sú jednoduchšie na poskladanie
LR [20]	Náročné na implementáciu Dokážu spracovať väčšinu gramatík, niektoré varianty dokonca všetky Zvyčajne lineárna zložitosť, u dokonalejších variánt je zložitosť v najhoršom prípade $O(n^3)$	Najvýkonnejšie generátory parserov

Packrat (PEG) [12]	Lineárna zložitosť Používa špeciálny formát zápisu gramatiky Designovaný na parsovanie počítačových jazykov	Jednoduché a zároveň výkonné parsre alebo generátory parserov pre počítačové jazyky

Tabuľka 5.1: Prehľad vlastností parsovacích algoritmov

Väčšina parsovacích nástrojov pre bezkontextové gramatiky, ktoré sú schopné generovať parsre v programovacom jazyku Java[1], používa parsovacie algoritmy typu LL a LR. Z tohto dôvodu sa ďalej budeme venovať práve týmto typom algoritmov.

5.1.1 Rekurzívny zostup

Metóda rekurzívneho zostupu je technika parsovania, ktorá spočíva vo vytvorení samostatných procedúr na analýzu každého neterminálového symbolu. [24] Označenie poradia, v ktorom sa neterminálové prvky nachádzajúce sa na pravej strane použijú na získanie neterminálového symbolu na pravej strane pravidla, sa volá *odvodenie* alebo *derivácia*. Existujú dve možnosti: **ľavá derivácia** a **pravá derivácia**. Prvá z nich znamená, že pravidlo sa uplatňuje zľava doprava, zatiaľ čo druhá presne naopak. **Note: možnosť pridať príklad ak je nutné**

Uplaňovanie derivácie sa vykonáva *rekurzívne*. Pri top down parsovaní sa použije ľavá derivácia a pri parsovaní bottom up pravá derivácia. Derivácia nemá žiadny vplyv na výsledný zparsovaný(derivčný) strom, ale má vplyv na použitý algoritmus.

5.1.2 Lookahead a Backtracking

Termíny lookahead a backtracking majú v parsovaní rovnaký význam ako v iných oblastiach informatiky. Lookahead označuje nasledujúci počet prvkov, ktoré sa berú do úvahy pri rozhodovaní o aktuálnom prvku. Parser môže skontrolovať ďalší token a rozhodnúť sa, ktoré pravidlo sa má uplatniť. Takýmto parsrom sa tiež hovorí *prediktívne* parsre [16].

Táto funkcionalita je dôležitá pre parsovacie algoritmy **LL**(viď kapitola 5.2) a **LR**(viď kapitola 5.3), pretože parsery pre jazyky, ktoré potrebujú iba jeden token lookahead sa jednoduchšie vytvárajú a sú rýchlejšie. Počet lookahead tokenov použitých v algoritme sa uvádza v zátvorkách za menom algoritmu (napr. LL(1), LR (k)). Použitie znaku hviezdy v zátvorkách naznačuje, že algoritmu môže skontrolovať nekonečné množstvo tokenov, aj keď to môže mať nepriaznivé účinky na výkon algoritmu.[33]

Backtracking je technika algoritmu, ktorá spočíva v hľadaní riešenia komplexnejších problémov tým, že skúša riešenia čiastkových problémov, ktoré následne porovná a kontroluje to najsľubnejšie. Ak aktuálne kontrolované riešenie zlyhá, parser sa vráti späť na poslednú úspešne zanalyzovanú pozíciu a vyskúša iné riešenie.

[1] <https://en.wikipedia.org/wiki/Comparison_of_parser_generators>

5.1.3 Lexikálna analýza pomocou DFA

Ako bolo definované v kapitole 4.1.1, základným kameňom deterministického konečného automatu je množina stavov a prechodových funkcií. Tieto prechodové funkcie určujú, ako môže automat v závislosti na udalosti prechádzať z jedného stavu na iný. Pri použití v rámci lexikálnej analýzy, prijíma automat znak po znaku zo vstupného reťazca, až pokiaľ nedosiahne finálneho (akceptačného) stavu, čo znamená, že dokáže vytvoriť token [31].

Lexikálna analýza spadá pod regulárne jazyky. Na základe definície 4.1.2 všetky regulárne jazyky sú prijímané deterministickým konečným automatom, a to je najzákladnejším dôvodom použitia DFA pre lexikálnu analýzu. Ďalším z dôvodov je možnosť spolupráce s online algoritmami.

Online algoritmus nepotrebuje na svoju prácu vedieť celý vstupný reťazec. Takéto algoritmy prijímajú sekvenciu znakov a vyhodnocujú ju priebežne [18]. Pre lexer to znamená, že algoritmus dokáže rozpoznať token v momente keď vidí všetky znaky, ktoré ho definujú.

5.2 LL parser

Kategória LL parsrov spadá pod *Top Down* parsre a ich názov vychádza z anglických názvov použitých techník (**L**eft-to-right read of the input, **L**eftmost derivation). To znamená, že číta vstupné symboly zľava doprava a snaží sa vytvoriť ľavú deriváciu. Tento algoritmus začína na štartovacom symbole a následne sa opakovane rozširuje do ľavého neterminálu, kým neskonštruuje cieľový reťazec.

LL parsre sú založené na práci s parsovacou tabuľkou, pomocou ktorej sa rozhodujú, aké pravidlo gramatiky bude uplatnené. Využívajú techniku lookahead na sledovanie nasledujúcich tokenov, čím dokáže toto pravidlo presnejšie vybrať. LL parsre je ale možné, a v praxi aj častejšie využívané, implementovať pomocou rekurzívneho zostupu.

Koncept LL parserov sa nevzťahuje na žiadny konkrétny algoritmus, ale skôr na triedu parserov, ktoré sa definujú vo vzťahu ku gramatikám. To znamená, že LL parser dokáže analyzovať LL gramatiku [15].

Presná definícia LL gramatiky, sa ale vzťahuje ku počtu lookahead tokenov potrebných na jej zparsovanie. Na základe toho existuje niekoľko parsovacích algoritmov pre LL parsre. Základné tri algoritmy sú:

- **LL(1)** algoritmus s lineárnou zložitosťou, ktorý používa jeden lookahead token
- **LL(k)** algoritmus taktiež z lineárnou zložitosťou, ktorý používa k lookahead tokenov.
- **LL(*)** algoritmus, ktorý môže použiť neobmedzené množstvo lookahead tokenov. Teoretická asymtotická zložitosť tohto algoritmu je $O(n^2)$, no v praxi väčšinou kontrolujú jeden až dva lookahead tokeny [28].

5.2.1 LL gramatika

LL gramatiky sú často používané práve kvôli veľkej obľube LL parserov aj napriek závažnej nevýhode. LL gramatiky **nepodporujú ľavú rekurziu**, čo znamená, že gramatiky,

ktoré ju obsahujú, musia byť upravené do ekvivalentnej podoby bez ľavej rekurzie aby mohli byť parsované pomocou LL parserov.

Tento problém má za následok miernu stratu produktivity a výkonu [33]. Výkonu z toho dôvodu, že pokiaľ gramatike s ľavou rekurziou mohol stačiť jeden lookahead token, u transformovanej gramatiky sa tento počet môže navýšiť na dva až tri tokeny. Problém produktivity spočíva v tom, že autor gramatiky musí písať gramatiku špeciálnym spôsobom, čo vyžaduje viac času. Tieto problémy bohužiaľ jemne podkopávajú silu algoritmu.

Na vysporiadanie sa s problémom ľavej rekurzie existuje ešte druhá možnosť. Namiesto ručnej úpravy gramatiky je možné použiť algoritmy, ktoré dokážu gramatiku transformovať. Niektoré s nástrojov využívajúce LL parsovanie majú takúto možnosť v sebe implementovanú, no ak by si autor chcel napísať vlastný parser, musí sa s tým vysporiadať sám.

5.3 LR parser

Algoritmy postavené na LR parsovaní sú hlavným dôvodom úspechu parsovania pomocou *Bottom Up* metódy. Aj napriek tomu, že písanie LR gramatík prináša väčšiu flexibilitu oproti tradičným LL(1) gramatikám, ich popularita je bohužiaľ nízka a to najmä z dôvodu, že konštrukcia LR parserov bola v minulosti veľmi náročná [33]. Názov LR vychádza z anglických názvov použitých techník (**L**eft-to-right read of the input, **R**ightmost derivation).

LR parsre spadajú do kategórie tzv. *shift-reduce* parserov, ktoré pracujú v dvoch krokoch:

- **Shift**: Prečíta jeden token zo vstupného reťazca. Z tohto tokenu sa stane nový parsovací strom s jedným uzlom.
- **Reduce**: Tento krok sa aplikuje ak sa sa nájde pravidlo gramatiky vytvorené z aktuálnych parsovacích stromov. Aplikovaním pravidla sa dané stromy spoja do jedného stromu z novým koreňom.

Jednoducho povedané, *shift* operácia číta vstupný reťazec a *reduce* operácia vytvára finálnu podobu parsovacieho stromu. V LR parsroch sa rozhodovanie o *shift* a *reduce* operáciách zakladá na všetkom, čo už bolo zparsované, a nie iba na jednom, najvyššom symbole v zásobníku. LR parsre dokážu riešiť toto rozhodovanie konštantnou rýchlosťou, a to zhromaždením všetkých príslušných informácií o sparsovanom kontexte do jedného čísla nazývaného stav parseru. Pre každú gramatiku existuje pevný (konečný) počet takýchto stavov [20].

LL a LR parsre vznikli približne v rovnakej dobe a preto majú veľa spoločných faktorov. Rovnako ako u LL parserov aj LR využíva technológiu lookahead a počet používaných lookahead tokenov sa značí rovnako. To znamená, že LR(k) parsovací algoritmus dokáže parsovať gramatiku, ktorá vyžaduje k lookahead tokenov. LR gramatiky sú menej obmedzujúce, a preto silnejšie oproti LL gramatikám. Jednou obrovskou výhodou je práve **podpora ľavej rekurzie**.

LR parsre taktiež pracujú s tabuľkami, ale namiesto jednej potrebujú dva komplikovanejšie.

- Prvá napovedá parseru, čo má robiť na základe aktuálne čítaného tokenu, aktuálneho stavu a možných lookahead tokenov.

- Druhá rieši prechod medzi jednotlivými stavmi po každej akcii.

Z popísaných informácií je vidieť, že LR parsre sú dostatočne výkonne a majú lineárnu zložitosť. Aj napriek tomu boli používané menej ako LL parsre a to práve z dôvodu tabuliek, ktoré používa. Tie sa ručne veľmi tažko vytvárajú a pri zložitejších gramatikách môžu byť veľmi veľké. V dnešnej dobe už existujú nástroje na generovanie týchto tabuliek priamo z gramatiky, no ak by si užívateľ volil vlastný ručne písaný parser, radšej by volil cestu *Top Down* metódy.

5.3.1 Simple LR a Lookahead LR

Na základe toho ako sú tabuľky pre LR parser vygenerované, existuje viacero typov analyzátorov ako napríklad:

- **Lookahead LR (LALR)**, ktorý bol vynájdený profesorom Frankom DeRemerom v rámci jeho dizertačnej práce[10]. V tejto práci ukázal, že LALR parser má väčšiu schopnosť rozpoznávať jazyky ako LR(0) parser, pričom vyžaduje rovnaký počet stavov pre jazyk rozpoznateľný oboma analyzátormi. To robí LALR parser pamäťovo efektívnou alternatívou k analyzátoru LR(1) pre jazyky, ktoré sú LALR.
- **Simple LR (SLR)**, ktorý vypočítava lookahead tokeny jednoduchou aproximačnou metódou založenou priamo na gramatike, pričom ignouje individuálne stavy a prechody parsera. Použitie SLR gramatiky zabezpečí, že pri parsovaní nenastanú žiadne *shift/reduce* alebo *reduce/reduce* konflikty[9].

Použitím týchto alternatívnych LR parsrov sa stráca výkonnosť parsru oproti originálnemu LR parsru. Poradie podľa výkonnosti je následovne:

$$\text{LR}(1) > \text{LALR}(1) > \text{SLR}(1) > \text{LR}(0)\ [14]$$

Názvy SLR a LALR parsrov sú trochu zavádzajúce, nakoľko SLR nie je až tak jednoduchý a LALR nie je jediný, ktorý používa lookahead tokeny. Dá sa povedať ze SLR je jednoduchší a rozhodovanie LALR veľmi úzko súvisí na lookahead tokenoch. Podstata ich rozdielu je v tabuľkách, kde menia časti o tom, čo majú robiť a ako majú vytvárať lookahead sety. Tieto úpravy prinášajú určité obmedzenia na gramatiky, ktoré sú schopné zparsovať [33].

SLR parser je veľmi obmedzujúci a v praxi sa až tak veľmi nepoužíva. Na druhú stranu LALR parser dokáže spracovať väčšinu praktických gramatík a preto je široko používaný.

5.4 Teória proti praxi

Teória LL a LR parsovania je stará už viac ako 50 rokov. Prvá písomná zmienka o LR parsovaní [20] bola zverejnená v roku 1965. Od tej doby vzniklo obrovské množstvo článkov o parsovaní a teórie jazykov, v ktorých akademici skúmali matematické rozmery parsovania. Aj napriek tomu sa v posledných rokoch objavujú nové a dôležité výsledky, čomu nasvedčuje aj prieskum spísaný v knihe *Parsing Techniques: A Practical Guide* [14] z roku 2007, ktorej bibliografia obsahuje viac ako 1700 citovaných článkov! [15]

Je bezpečné povedať, že základné LL a LR parsre sa ukázali ako veľmi nepostačujúce pre prípady používania v reálnom svete. Mnohé gramatiky, ktoré by boli prirodzene napísané pre tieto prípady, nie sú LL alebo LR, no stále existujú možnosti na ich rozšírenie, pomocou ktorých si dokážu zachovať svoje silné stránky. Dva najobľúbenejšie parsovacie nástroje založené na LL a LR (ANTLR a Bison) rozširujú algoritmy LL a LR rôznymi spôsobmi, pričom pridávajú funkcie ako prednosť operátora, syntaktické / sémantické predikáty, možnosť bactrackingu a generalizované parsovanie.

Dokonca aj tieto existujúce nástroje nie sú stopercentne dokonalé a je nutné aby sa stále vyvíjali aj v rámci použitia, hlásenia chýb, lepšej vizualizácie, lepšej integrácie z programovacími jazykmi atď. Napríklad ANTLR v4 úplne prepracoval svoj analytický algoritmus na zlepšenie jednoduchosti použitia oproti predchádzajúcej verzie. Algoritmus predstavil pod názvom **ALL(*)**[29] v roku 2014. Bison experimentuje s algoritmom **IELR** [8], čo je alternatíva k LALR, ktorá bola zverejnená v roku 2008 a má za cieľ rozšíriť počet gramatík, ktoré môže prijímať a parsovať efektívne.

5.4.1 ANTLR alebo Bison?

ANTLR a Bison sú nástroje na generovanie parserov, ktoré dokážu parsovať bezkontextové gramatiky. Najväčším rozdielom medzi nimi je konkrétny typ gramatiky, ktorú parsujú. Zatiaľ čo Bison parsuje LALR gramatiky (Bottom Up) ANTLR parsuje LL gramatiky (Top Down).

Čo sa týka týchto gramatík, každá má svoje výhody a nevýhody. ANTLR v4 sa ale dokázal čiastočne zbaviť jednej závažnej nevýhody v LL gramatikách. Ako už bolo spomenuté LL gramatiky nepodporujú pravidlá s ľavou rekurziou. ANTLR v4 si však počas generovania parseru dokáže upraviť poskytnutú gramatiku a zbaviť sa *priamych ľavých rekurzií* [29].

Bison generuje parsre, ktorých práca je založená na tabuľkách [8]. Logika parsovania je teda uložená práve v týchto tabuľkách a nie priamo v kóde parseru. Z toho vyplýva, že parser má aj pre relatívne zložité jazyky malú stopu kódu. Na rozdiel od toho, ANTLR generuje parser s rekurzívnym zostupom. U takýchto parserov je každé pravidlo gramatiky reprezentované funkciou v kóde parseru. Z toho logicky vyplýva, že tieto parsre budú mať veľmi veľkú stopu kódu. Ako príklad uvediem parser vygenerovaný pre gramatiku MySQL obsahuje viac ako 58 000 riadkov kódu. Výhodou tohto prístupu je jednoduchšie pochopenie práce parseru a prípadne debugovanie. Taktiež parsre s rekurzívnym zostupom sú typicky rýchlejšie ako tie, ktoré využívajú na svoju prácu tabuľky.

V neposlednom rade ďalšou výhodou pre ANTLR je grafický nástroj nazvaný *ANTLRworks*[2]. Tento nástroj je dostupný ako plugin do vývojových prostredí IntelliJ, NetBeans, Eclipse, Visual Studio Code, a jEdit. Počas toho ako užívateľ zadáva reťazec, ktorý chce zparsovať, tento nástroj vizualizuje parsovací strom a základe gramatických pravidiel a ak objavý konflikt, oznámi užívateľovi, čo ho spôsobilo. Súčasťou tohto nástroja je aj možnosť profilovania, ktorý vám pomôže pochopiť, ktoré rozhodnutia vo vašej gramatike sú komplikované alebo drahé.

ANTLR v4 je aktuálne z týchto dvoch nástrojov ten lepší najmä vďaka jeho novo vyvinutom algoritme ALL(*), ktorého teoretická zložitosť je $O(n^4)$, no na gramatikách používaných v praxi pracuje **lineárne** [29].

[2] <http://www.antlr.org/tools.html>

Kapitola 6

MySQL DDL parser

Na základe dôvodov popisovaných v poslednej kapitole, som sa na implementáciu DDL parseru pre MySQL rozhodol použiť nástroj ANTLR verzie 4. Tento nástroj je aktuálne vo svete asi najpoužívanejší a prináša veľké množstvo výhod, ktoré uľahčujú implementáciu aj následne ladenie parsovacích programov.

6.1 Základy ANTLR v4

Štvrtá verzia ANTLR má niektoré dôležité nové možnosti, ktoré znižujú učiacu sa krivku a umožňuje vytvárať gramatiky a jazykové aplikácie oveľa jednoduchšie. Najdôležitejšou výhodou je, že ANTLR v4 akceptuje každú gramatiku, ktorú mu poskytnete (s výnimkou týkajúcou sa nepriamej ľavej rekurzie). ANTLR prekladá vašu gramatiku do spustiteľného, ľudsky čitateľného analytického kódu, do ktorého keď zadáte platný vstupný reťazec, parser vždy rozpozná vstup správne, nezávisle na komplikovanosti gramatiky.

V rámci tejto sekcie sú popísané základné informácie potrebné na pochopenie toho, ako s ANTLR pracovať. Na získanie podrobnejších alebo viac detailných informácií odporučujem knihu *The Definitive ANTLR4 reference*[27] napísanú autorom ANTLR nástroja.

6.1.1 Nastavenie ANTLR

ANTLR sa skladá z dvoch častí: Nástroja na generovanie lexeru a parseru z gramatiky a knižnice potrebnej na beh parseru.

Nástroj na generovanie je rovnaký, nezávisle na výstupnom jazyku generovaného lexeru a parseru, a pre samotný beh parseru nie je potrebný. Využijú ho užívatelia, ktorí vytvárajú, alebo upravujú gramatiku. Je to vlastne program napísaný v programovacom jazyku Java, a preto je nutné na prácu s ním mať nainštalovanú aspoň Javu 1.7. Inštalácia ANTLR spočíva v stiahnutí naposledy zverejneného java archívu[1], alebo jeho zostavením zo zdrojových kódov dostupných v GitHub repositáru[2].

Kroky nutné na nainštalovanie ANTLR nástroja:

[1] <http://www.antlr.org/download.html>
[2] <https://github.com/antlr/antlr4>

1. Nakopírovať stiahnutý nástroj na miesto, kde sú uchovávané java knižnice tretích strán.

2. Pridať cestu k nástroju do systémových premenných.

3. Na zjednodušenie používania vytvoriť spúšťací skript a vytvoriť naň alias.

Konkrétne príklady inštalačných krokov je možné si pozrieť v prílohách D.1 pre Windows a D.2 pre Linux a Mac OS.

Vstupom pre ANTLR nástroj je súbor s príponou *.g4*, ktorý obsahuje gramatiku jazyka (viď kapitola 6.1.2), pre ktorý bude analyzátor vygenerovaný. Príkaz na vygenerovanie lexeru a prseru z definovanej gramatiky vyzerá nasledovne.

antlr4 <options> <grammar-file-g4>

Jedným dôležitým nastavením pri generovaní je možnosť zvoliť si cieľový jazyk vygenerovaného parseru. Prednastaveným výstupným jazykom je Java, no ANTLR podporuje aj Python, JavaScript a C#. Na prechádzanie zparsovaného stromu ANTLR v základe generuje objekt poslucháča, ktorý reaguje na udalosti vznikajúce pri príchode aj pri odchode z uzlu stromu. Ak užívateľ preferuje pri prechádzaní stromom použiť návrhového vzoru návštevník, je možné ANTLR nástroju nastaviť generovanie *visitor* objektov.

6.1.1.1 Maven

Pre projekty, ktoré využívajú Maven[3] na riadenie a správu buildov aplikácii, je možné túto správu využiť aj pre nastavenie ANTLR.

Prvým krokom je si v *pom.xml* nadefinovať závislosť na *antlr4-runtime*, ktorý je potrebný na beh vygenerovaného parseru.

```
<dependency>
    <groupId>org.antlr</groupId>
    <artifactId>antlr4-runtime</artifactId>
    <version>4.7</version>
</dependency>
```

Ako druhý krok, je nutné nastaviť maven plugin, pomocou ktorého bude ANTLR generovať parser z poskytnutej gramatiky počas buildu aplikácie. Tento plugin v základe očakáva *.g4* súbory v priečinku *src/main/antlr4*. Túto cestu je možné ručne definovať pomocou konfiguračného parametru *sourceDirectory*.

[3] <https://maven.apache.org/>

```
<plugin>
    <groupId>org.antlr</groupId>
    <artifactId>antlr4-maven-plugin</artifactId>
    <version>4.7</version>
    <configuration>
        <sourceDirectory>${antlr.source.directory}</sourceDirectory>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>antlr4</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Na definíciu cieľového balíčka, v ktorom sa budú nachádzať vygenerované súbory, použije ANTLR štruktúru v zdrojovom priečinku. Ak uvažujeme základný zdrojový priečinok *src/main/antlr4* a cesta k súboru z gramatikou bude napríklad:

src/main/antlr4/mysql/grammar/MySqlParser.g4

Vygenerovaný parser sa bude v tomto prípade nachádzať v balíčku **mysql.grammar**.

6.1.2 ANTLR Gramatika

Písanie gramatiky je podobné písaniu softwaru s výnimkou toho, že namiesto funkcií a procedúr sa používajú pravidlá. ANTLR používa na zápis gramatiky EBNF notáciu, ktorá je zároveň používaná v dokumentáciach programovacích jazykov na popis ich syntaxi. Vďaka nej je možné pri zápise pravidiel používať operácie ako napríklad zlučovanie, opakovanie a nepovinnosť, ktoré sú popísané v kapitole 4.1.5.

Aby bol ANTLR schopný rozpoznať pravidlá pre lexer a parser, je zavedená konvencia na zápis týchto pravidiel. Tá kontroluje, či prvé písmeno pravidla je veľké alebo malé. Síce sa toto pravidlo vzťahuje iba na prvé písmeno, ale aj tak je zvykom definovať názov pravidiel pre lexer iba pomocou veľkých písmen.

6.1.2.1 Pravidlá pre lexer

Je dôležité mať na pamäti, že lexerove pravidlá sú analyzované v poradí, v akom sa objavujú, a môžu byť nejednoznačné. Typickým príkladom v programovacom jazyku je napríklad to, že názov premennej môže byť akékoľvek slovo s výnimkou kľúčových slov. Poradie pravidiel rieši nejednoznačnosť tým, že použije prvú zhodu, a preto sú tokeny označujúce kľúčové slová definované ako prvé, zatiaľ čo identifikačné znaky sú uvedené ako posledné. Na lepšie priblíženie tvorby gramatiky pre lexer sa pozrime na príklad:

```
1 fragment NUMBER          : [0-9]+ ;
2 DECIMAL_NUMBER           : NUMBER ',' NUMBER ;
3 WHITESPACE               : ' ' -> skip ;
```

Tento príklad zobrazuje tri typy pravidiel, ktoré sa používajú na vytvorenie gramatiky pre lexer. V prvom pravidle je možné vidieť kľúčové slovo **fragment**, ktoré definuje prepoužiteľné bloky pre pravidlá lexeru. Vďaka definovaniu fragmentu *NUMBER*, je následné možné tento fragment použiť pri definovaní pravidla *DECIMAL_NUMBER*. Ak by v gramatike existovali definície na fragmenty, ktoré neboli použité na definovanie nejakého pravidla, pre lexer jednoducho nemajú žiadny efekt. Je dôležité si uvedomiť, že fragment sa neberie ako pravidlo lexeru, z čoho vyplýva, že ak by sme pomocou lexeru vytvoreného gramatikou s príkladu chceli analyzovať číslo bez desatinej čiarky, nepodarilo by sa nám to. Aby lexer podporoval aj tento prípad, je nutné upraviť pravidlo na:

```
1 DECIMAL_NUMBER : NUMBER (',' NUMBER)? ;
```

Najviac zaujímavým pravidlom je pravidlo definujúce medzery:

```
1 WHITESPACE : ' ' -> skip ;
```

Zaujímavé je z dôvodu, že predstavuje možnosť ako ANTLR-u indikovať, že má niečo ignorovať. Vďaka tomuto pravidlu sa náramne zjednoduší písanie pravidiel pre parser. Ak by takého pravidlo neexistovalo, musel by to autor gramatiky zahrnúť medzi každú podskupinu pravidiel parseru. ako napríklad:

```
1 sum : WHITESPACE* NUMBER WHITESPACE* '+' WHITESPACE* NUMBER;
```

Tento princíp sa typicky aplikuje aj na komentáre. Tie sa taktiež môžu objaviť kdekoľvek a väčšinou autora pri parsovaní nezaujímajú, takže sa jednoducho ignorujú.

6.1.2.2 Dostupné gramatiky

Kľúčovou výhodou ANTLR v4 je dostupnosť obrovského množstva vytvorených gramatík od autorov z celého sveta, ktoré sú združené v rámci ANTLR GitHub repozitáru[4]. Pre tieto gramatiky neexistuje žiadna globálna licencia. Každá z gramatík má svoju vlastnú licenciu. Súčasťou tejto kolekcie je aj gramatika pre MySQL, ktorá je vytvorená na základe dokumentácie pre MySQL verzie 5.6 a 5.7, čo presne vyhovuje potrebám projektu Debezium. MySQL gramatika je vydaná pod licenciou MIT[5].

6.1.3 Ukážková implementácia

Ako ukážkovú implementáciu použijeme príklad na parsovanie pozdravu. Prvým krokom implementácie je definícia gramatiky (viď ukážka 6.1). V tomto prípade to je veľmi jednoduchá gramatika o jednom pravidle *greeting*, ktoré na prvom mieste očakáva slovo *hello* nasledované akýmkoľvek reťazcom skladajúcim sa z malých písmen abecedy.

[4] <https://github.com/antlr/grammars-v4>

[5] MIT licencia je jedna z najmenej reštriktívnych open source licencií. Ktokoľvek môže takto licencovaný program bez obmedzenia používať a šíriť, pokiaľ z programu neodstráni kopiu licencie a meno autora.

Ukážka 6.1: Ukážková gramatika Hello

```
1 grammar Hello;
2 greeting  : 'hello' ID;
3
4 ID        : [a-z]+ ;
5 WS        : [ \t\r\n]+ -> skip ;
```

Z tejto gramatiky ANTLR vygeneruje HelloParser, HelloLexer, rozhranie HelloListener a jeho základnú (prázdnu) implementáciu HelloBaseListener. Jeho použitím je možné poslúchať udalosti vzniknuté pri príchode a odchode z pravidiel gramatiky. Implementujeme si vlastný HelloListenerCustom, pomocou ktorého budeme reagovať na nami zvolené udalosti. V tomto prípade to je výpis na systémový výstup pri príchode aj odchode na pravidlo *greeting* (viď ukážka 6.2).

Ukážka 6.2: Ukážková implementácia HelloListener.java

```
1  public class HelloListenerCustom extends HelloBaseListener {
2    @Override
3    public void enterGreeting(HelloParser.GreetingContext ctx ) {
4      System.out.println( "Entering greeting : " + ctx.ID().getText() );
5    }
6
7    @Override
8    public void exitGreeting(HelloParser.GreetingContext ctx ) {
9      System.out.println( "Exiting greeting" );
10   }
11 }
```

Posledným krokom je implementovať aplikáciu, ktorá inicializuje HelloLexer, HelloParser a spustí samotné parsovanie (viď ukážka 6.3). HelloLexer sa inicializuje pomocou prúdu znakov reprezentovaných CharStream objektom. Použitím lexeru si vytvoríme prúd tokenov CommonTokenStream, ktorým inicializujeme HelloParser. Zavolaním príkazu na riadku 10 ukážky 6.3 spustíme proces parsovania, ktorého vstupom je zparsovaný strom. Posledným krokom je prechod stromu pomocou ParseTreeWalker triedy, ktorej priradíme nami vytvorený HelloListenerCustom. Náš HelloListenerCustom je možné priradiť aj parseru, vďaka čomu by sme mohli odchytávat udalosti už počas toho, ako sa vygenerovaný parser snaží vytvoriť zparsovaný strom. V takomto prípade je ale nutné počítať zo situáciami, že aktuálne sparovaný kontext nie je úplný a taktiež nie je zaručené, že pravidlo bude obsiahnuté vo výslednom zparsovanom strome.

Ukážka 6.3: Ukážková implementácia Main.java

```
1  import org.antlr.v4.runtime.*;
2  import org.antlr.v4.runtime.tree.*;
3
4  public class Hello {
5    public static void main( String[] args) throws Exception
6    {
7      HelloLexer lexer = new HelloLexer(CharStreams.fromString(args[0]));
8      CommonTokenStream tokens = new CommonTokenStream( lexer );
9      HelloParser parser = new HelloParser( tokens );
10     ParseTree tree = parser.greeting();
11     ParseTreeWalker walker = new ParseTreeWalker();
12     walker.walk( new HelloListenerCustom(), tree );
13   }
14 }
```

6.2 Návrh DDL parseru

Pri návrhu nového generovaného DDL parseru pomocou ANTLR, je dôležité brať v úvahu aktuálny design projektu Debezium a implementáciu stávajúceho parseru. Primárnou snahou je prepoužiť čo možno najväčšiu časť aktuálnej implementácie, ktorú už niekoľko projektov využíva. Počas návrhu taktiež prihliadam na budúcnosť, ktorá nasvedčuje tomu, že projekt Debezium má v pláne podporovať väčšie množstvo DBMS, ktorých prípadne DDL parsery budú taktiež využívať ANTLR nástroj. Práve na základe tejto skutočnosti je snaha zgeneralizovať čo najväčšiu časť novej implementácie.

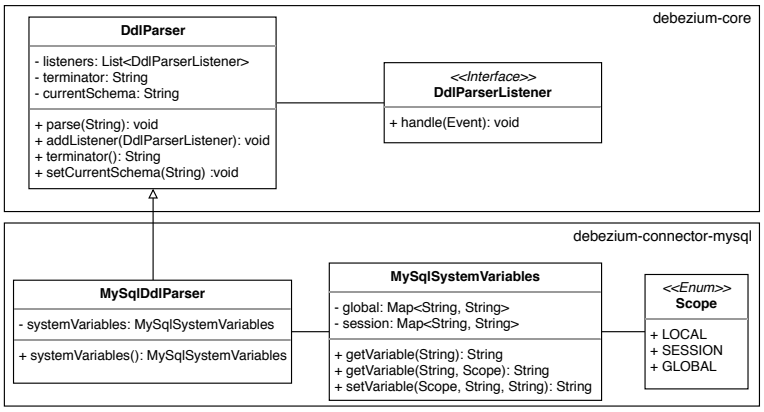
6.2.1 Aktuálny design

Štruktúra projektu Debezium je členená do viacerých modulov. Každý konektor pre jednotlivé DBMS je implementovaný v samostatnom module. V kontexte DDL parsovania MySQL sú významné dva moduly:

- **debezium-core** modul, v ktorom sa nachádza implementácia spoločná pre všetky databázové systémy.
- **debezium-connector-mysql** modul, ktorý je závislý na *debezium-core* module. Súčasťou modulu je celá implementácia špecifická pre MySQL databázový systém, ktorej súčasťou je aj aktuálny DDL parser, ktorý má byť nahradený.

MySQL je aktuálne jediným Debeziom podporovaným databázovým systémom, v rámci ktorého sa parsujú DDL dotazy. Pravdepodobne z tohto dôvodu nebola implementácia dostatočne generalizovaná a mnohé funkcionality, ktoré do budúcna budú využívať aj iné typy DDL parserov neboli implementované v rámci *debezium-core* modulu.

Vstupným bodom pre DDL parser je komponenta MySqlSchema, ktorá zaznamenáva históriu schémat pre databáze hostované na MySQL. V rámci tejto komponenty sa inicializuje MySqlDdlParser, ktorý dedí od DdlParser. Diagram na obrázku 6.1 znázorňuje aktuálnu závislosť tried a verejných metód, ktoré sú podstatné pri práci s MySQL DDL parserom.



Obr. 6.1: Diagram tried aktuálneho návrhu MySQL DDL parseru

Všetky verejné metódy, ktoré obsahuje DdlParser a MySqlDdlParser predstavujú API parseru, ktoré je volané komponentou MySqSchema.

Stávajúci návrh je viac než nedostačujúci pre možnosť rozšírenia, či už novým parserom pre MySQL databázový systém, alebo parserom pre ktorýkoľvek iný DBMS. Na prvý pohľad je napríklad možné vidieť absenciu rozhrania pre DdlParser. Bližší pohľad na jednotlivé triedy z obrázku 6.1 nám načrtne obraz na vylepšenie tohto návrhu.

DdlParser trieda má aktuálne predstavovať základ pre všetky DDL parsre. Obsahuje implementáciu kľúčových metód, ktoré riešia:

- Nastavenie DDL parseru ako napríklad, či sa majú parsovať dotazy pre náhľady tabuliek, ktoré schéma resp. databáza je aktuálne sledovaná a podobne.
- Signalizáciu zmien nad databázou, ktoré boli počas parsovania detekované.
- Spoločné pomocné metódy, ako napríklad metóda na vyseparovanie reťazca, ktorý je obklopený úvodzovkami a mnohé ďalšie.

Súčasťou DdlParser triedy je ale aj implementácia základnej parsovacej logiky a parsovanie konkrétnych typov reťazcov, u ktorých sa predpokladalo, že ich bude možné prepoužiť aj v rámci ďalších implementácií ddl parserov.

Trieda MySqlDdlParser predstavuje implementáciu DDL parseru. Táto trieda je už špecifická pre MySQL databázový systém, a preto nie je možné ju ďalej generalizovať. Jediným problém, ktorý u tejto implementácie nastal, je trieda MySqlSystemVariables, ktorá udržuje systémové premenné na úrovni priestoru definovaného pomocou enumu Scope. Autor túto funkcionalitu implementoval ako súčasť MySQL, no správne by mala byť taktiež generalizovaná, nakoľko každý DBMS má svoje vlastné systémové premenné.

6.2.2 Úprava aktuálneho designu

Na základe nedokonalostí zistených v aktuálnom návrhu DDL parsrov, je potrebné vykonať zmeny v tomto návrhu tak, aby vyhovovali implementácii nového parseru generovaného

pomocou ANTLR, ale zároveň zachovali aktuálne riešenie a prípadnú implementáciu nových DDL parserov týmto "starým" spôsobom.

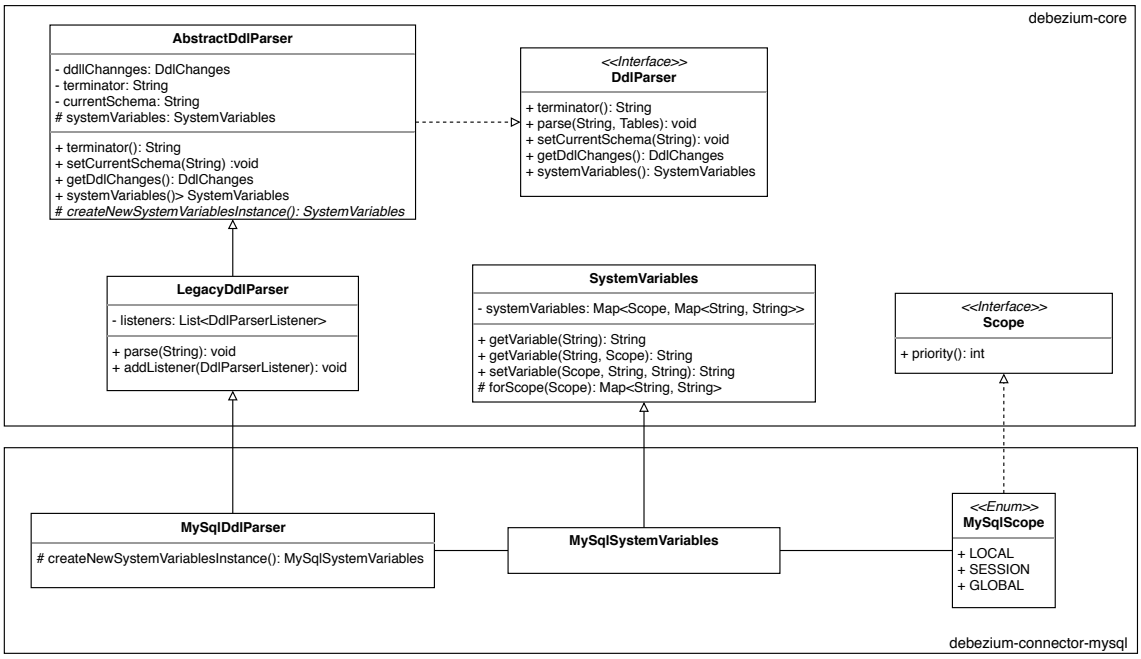
Prvou zmenou v návrhu je vytvorenie rozhrania pre DDL parsre, ktoré bude definovať možnosť komunikácie pre komponenty, ktoré potrebujú parsovať DDL dotazy. Toto rozhranie bude definovať všetky verejné metódy z tried DdlParser a MySqlDdlParser. Tým sa dostávame k prvému problému, ktorým sú systémové premenné. Nakoľko nové rozhranie bude súčasťou *debezium-core* modulu, je nutné generalizovať triedu MySqlSystemVariables.

Aktuálna implementácia triedy MySqlSystemVariables obsahuje dve Map objekty, ktoré udržiavali hodnoty systémových premenných tak, že na základe hodnoty enumu Scope bola zvolená správna mapa z ktorej sa hodnota premennej čítala. Takýto prístup v generalizovanej verzii nie je možné použiť, nakoľko hodnoty priestoru definovaného enumom Scope sú odlišné v závislosti na DBMS. Ukladanie systémových premenných volá po riešení pomocou mapy priradenej k hodnote Scope enumu. Enum je ale špeciálny typ objektu v Jave, ktorý nedokáže dediť od iného enumu, čo prináša dilemu spojenú s tým, ako vyriešiť vlastné hodnoty Scope pre jednotlivé DBMS?

Dedenie enumov je väčšinou zlý nápad, no toto je jeden z prípadov kde to dáva zmysel. Možnosť ako túto situáciu vyriešiť je namiesto enumu Scope použiť rozhranie Scope. V takomto prípade majú jednotlivé implementácie možnosť vytvorenia vlastného enumu s vlastnými hodnotami kde stačí, aby tento enum implementoval rozhranie Scope. [3]

Nakoľko chceme prepoužiť resp. zanechať čo možno najväčšiu časť stávajúcej aplikácie, bolo by vhodné vyseparovať z triedy DdlParser všetko, čo by mohla používať aj nová implementácia parseru. Ako bolo popísané v kapitole 6.2.1, súčasťou implementácie DdlParser sú kľúčové metódy, ktoré nie sú závislé na štýle parsovania, a preto je vhodné ich prepoužiť aj v rámci ANTLR parseru. Vyseparujeme preto túto časť implementácie do abstraktnej triedy, ktorá bude predstavovať najnutnejší základ pre všetky typy DDL parserov. Všetky ostatné metódy, ponecháme v DdlParser. Aby názvoslovie nebolo príliš zmätočné pozostatok triedy DdlParser predstavujúci implementáciu základnej parsovacej logiky pre ručné parsovanie nazveme LegacyDdlParser a názov DdlParser rezervujeme pre novo vytvorené rozhranie DDL parserov.

Pri každej zmene, ktorá nastane nad databázovým modelom uloženým v pamäti sa pomocou udalosti táto zmena signalizuje poslucháčom, ktorý boli priradený DDL parseru. Myšlienka tejto funkcionality ale v stávajúcej implementácii nie je využitá. Jediným poslucháčom je DdlChanges trieda, ktorá sa tvári ako poslucháč, ale reálne tak nepracuje. Jeho reakcia na prijatú udalosť spočíva v iba v tom, že si túto udalost pridá do listu udalostí, s ktorým sa následne pracuje až po ukončení práce parseru. Nakoľko neexistujú žiadny iný poslucháči, vedenie projektu sa rozhodlo túto funkcionalitu zanechat v rámci LegcyDdlParser implementácie, ale nechcelo ju mať v novej implementácii. DdlChanges trieda sa teda stane súčasťou abstraktnej implementácie parsru a prístup k nej bude možný pomocou vystavenej metódy v rámci DdlParser rozhrania.



Obr. 6.2: Upravený diagram tried MySQL DDL parseru

6.2.3 ANTLR DDL parser design

Úpravou designu aktuálnej implementácie, ktorej výsledok je možné vidieť na obrázku 6.2 nám zabezpečí možnosť jednoduchšieho návrhu pre DDL parsre generované pomocou ANTLR. Z celého návrhu je nutné nahradiť triedy LegacyDdlParser a MySqlDdlParser za novú implementáciu. Zachovaním myšlienky na prepoužitie bude opäť návrh obsahovať triedu zo základnou implementáciou ANTLR parserov a konkrétnu implementáciu pre MySQL DBMS.

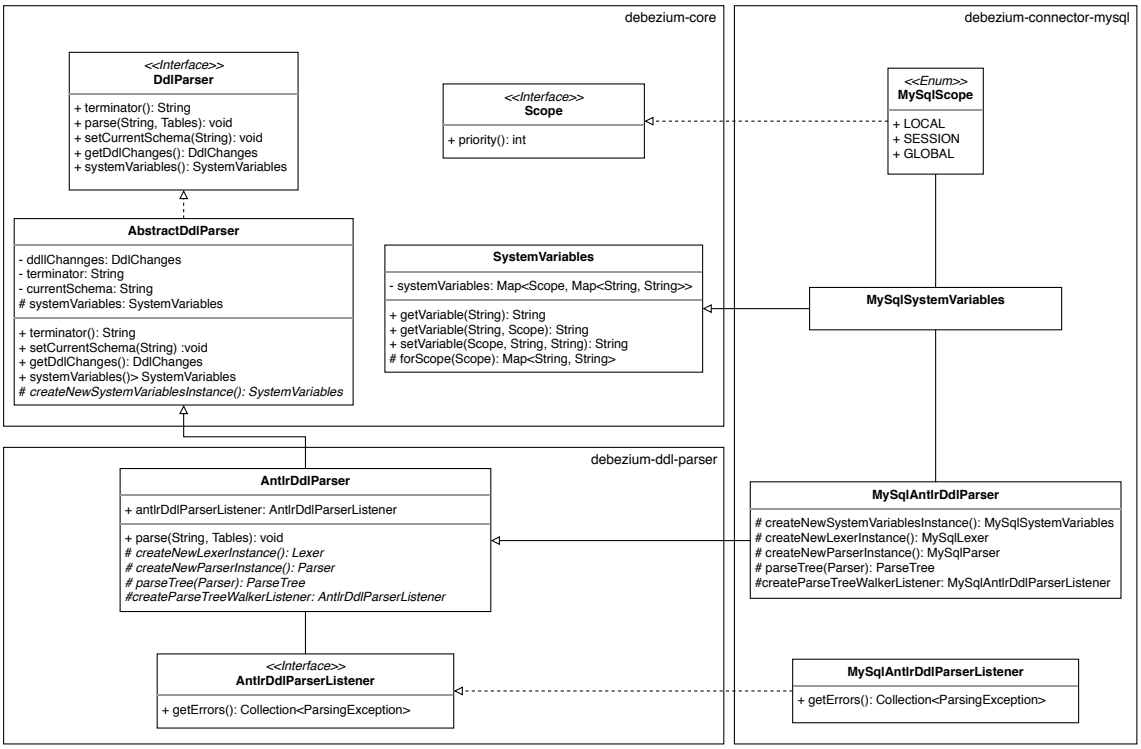
Pri návrhu je nutné počítať aj z generovaním gramatík, čo pre modul, ktorý bude definície gramatík obsahovať spôsobí, že jeho build bude trvať dlhšie. Práve z dôvodu aby sa vývojárom ušetril čas buildu pri implementovaní špecifických parserov, bude základná implementácia spolu z definíciou gramatík a vygenerovanými triedami oddelená do samostatného modulu, ktorý nazve **debezium-ddl-parser**. Samotná implementácia parseru pre MySQL, už ale bude súčasťou modulu *debezium-connector-mysql*. Takýto návrh spôsobí to, že pri implementácii MySQL parseru, bude nutné zostaviť modul *debezium-ddl-parser* iba raz aby sa vygenerovali triedy z gramatiky a následne stačí buildiť iba modul v ktorom sa nachádza konkrétna implementácia, v rámci ktorého sa už nič ohľadom ANTLR parseru generovať nebude.

Ako je vidieť z ukážkovej implementácie v kapitole 6.1.3, ANTLR DDL parser sa skladá z inicializácie lexeru, parseru a priradení poslucháča, ktorý bude reagovať na zparsovaný strom a vykonávať náležité akcie nad databázovým modelom, ktorý si udržuje Debezium. Keďže parsovací strom môže pozostávať z viacerých SQL dotazov, je potrebné počítať s tým, že v rámci jeho prechodu a snahe vykonať úpravy nad databázovým modelom môže nastať viacero chýb. Tieto chyby bude nutné lokálne ukladať a spracovať ich až po prejdení celého stromu.

Kvôli tomu bude od konkrétnych poslacháčov vyžadované aby implementovali metódu, ktorá vráti zoznam chýb, ktoré nastali počas prechodu stromu. To sa zabezpečí použitím rozhrania AntlrDdlParserListener.

Inicializácia lexeru a parseru je závislá na konkrétnych triedach vygenerovaných ANTLR nástrojom, ktoré majú generovaný názov podľa názvu gramatiky. To indikuje, že novú inštanciu triedy musí sprostredkovávať implementácia konkrétneho parseru. Samotný postup inicializácie sa ale dá generovať a na získanie nových inštancií sa môže použiť abstraktná metóda, ktorej implementácia bude súčasťou implementácia konkrétneho parseru. Rovnaký problém sa aplikuje aj na spustenie parsovania a priradenie implementácie AntlrDdlParserListener. O inicializáciu sa bude starať trieda AntlrDdlParser.

Výsledok tohto návrhu je možné vidieť na obrázku 6.3.



Obr. 6.3: Diagram tried MySQL ANLTR DDL parseru

6.3 Implementácia

Implementácia generovaného parseru má nahradiť stávajúci ručne implementovaný parser. Z toho vyplýva, že parser musí zvládnuť analyzovať všetky typy DDL dotazov, ktoré upravujú štruktúru **databáze**, **tabuliek** a **pohľadov** (anglicky VIEW). Súčasťou parsra sú aj administratívne dotazy, ktorými sa nastavujú systémové premenné a taktiež dotazy na prepínanie sa v rámci aktívnej databáze nad, ktorou sa DDL aplikuje (USE dotazy).

6.3.1 Závislosť na veľkosti písmen

SQL dotazovací jazyk nie je závislý na veľkosti písmen. V niektorých prípadoch existuje možnosť nastavenia DBMS tak, že mu závisí na veľkosti písmen iba v pomenovaniach tabuliek a stĺpcov. Bohužiaľ parsre vygenerované ANTLR nástrojom sú závisle veľkosti písmen podľa toho, ako sú definované jednotlivé u tokenov pre lexer. ANTLR podporuje dve rôzne riešenia pre tento problém[6]. Jedným z nich je úprava gramatiky pre lexer, ktoré je postavené na myšlienke definovania fragmentov pre každé písmeno abecedy. Tento fragment by definoval písmeno abecedy ako veľké alebo malé písmeno napríklad: (*fragment A: 'a'|'A';*). Tieto fragmenty by následne boli použité pre definíciu konkrétneho pravidla pre lexer. Takáto zmena by ale vyžadovala kompletné prerobenie lexer gramatiky.

Druhou možnosťou je pri vytváraní prúdu znakov, ktoré predávame lexeru, použiť vlastnú triedu, ktorá preťaží metódy používané lexerom, čím prinúti lexer vidieť daný znak ako veľké alebo malé písmeno (záleží na nastavení). V takomto prípade gramatické pravidlá pre lexer musia byť definované iba veľkými alebo malými písmenami. Gramatika, ktorú používame na parsovanie MySQL má všetky pravidlá definované veľkými písmenami, takže na vyriešenie problému použijeme túto možnosť.

6.3.2 Proxy parse listener

Počas prechádzania zparsovaného stromu pomocou ParseTreeWalker, je možné mať priradený iba jedného poslucháča. Pri množstve odchytávaných udalosti by takýto prístup bol hrozne neprehľadný a bolo by náročné v implementácii jedného poslucháča hľadať odchytávané udalosti pre konkrétny typ spracovavaného SQL dotazu. Je vhodné preto vytvoriť poslucháčov, rozdelených podľa konkrétnych typov dotazov. Bohužiaľ, možnosť priradenia viacerých poslucháčov nie je súčasťou ANTLR runtime balíčku a bola evidovaná aj ako problém[7] v rámci ich verejného repozitára. Doporučením k tomu, aby sme sa vyhli samostatnému prechádzaniu zparsovaného stromu pre každého vytvoreného poslucháča, je vytvorenie jedného hlavného poslucháča, ktorý bude delegovať všetky udalosti na predom definovaných poslucháčov. Vygenerovaný poslucháč od ANTLR túto možnosť poskytuje pomocou metód ParseTreeListener#enterEveryRule a ParseTreeListener#exitEveryRule.

Počas implementácie som sa snažil túto možnosť rozumne zgeneralizovať, nakoľko predpokladám, že ostatné implementácie ANTLR parsru by ju taktiež radi uvítali, nedospel som k dokonalému riešeniu, ktoré by vyhovovalo. Aby takéto riešenie bolo možné a vyhovovalo by v každej situácii, bolo by nutné aby Java podporovala možnosť dediť od viacerých tried naraz. V ôsmej verzii javy bolo umožnené implementovať rozhranie zo základnou implementáciou jeho metód. Táto možnosť sa javila ako riešenie na problém dedenia viacerých tried, no nakoniec tiež zlyhala. Implementácia základných metód fungovala správne, ale vyžadovala závislosť na premenných definovaných v rámci triedy, ktoré sú v rozhraní implicitne statické a konečné. To samozrejme nevyhovovalo nakoľko jedna z premenných mala udržiavať list poslucháčov, ktorým sa majú udalosti delegovať.

Jediným riešením, ako implementovať túto funkcionalitu, aby bola čo najlepšie prepoužiteľná, bolo presunutie metód do jednej pomocnej triedy ProxyParseTreeListenerUtil, ktorú si zavolá každá implementácia samostatne.

[6] <https://github.com/antlr/antlr4/blob/master/doc/case-insensitive-lexing.md>
[7] <https://github.com/antlr/antlr4/issues/841>

6.3.3 Databázové DDL

Dôvodom na parsovanie databázových DDL dotazov na vytvorenie a upravenie databáze resp. schémy, je sledovanie ich nastavenej sady znakov. Táto informácia je dôležitá následne pri vytváraní databázových tabuliek a definícií stĺpcov v tabuľkách, u ktorých je možné nastaviť sadu znakov pomocou kľúčového slova DEFAULT. V takomto prípade sa má prepoužiť hodnota nastavená pre danú databázu[26]. Hodnoty nastavenej sady znakov pre databázu sa ukladajú v samostatnej mape uloženej v rámci DDL parseru.

Súčasťou databázových dotazov je zahodenie resp. zmazanie databáze. Tento prípad nebol ošetrovaný v rámci sávajúceho parseru, no osobne si myslím, že by sa riešiť mal. V prípade tohto dotazu by sa správne mali zmazať všetky tabuľky vytvorené v rámci danej databáze resp. schématu. V Debeziu je každá tabuľka definovaná pomocou *schemaName*, čo odpovedá našej databázi v MySQL, *catalogName*, čo je špecifické iba pre niektoré typy DBMS, a názvu tabuľky. Pri mazaní tabuliek z modelu v pamäti je preto nutné počítať z hodnotami *null*, ktoré sú taktiež validné. Výslednú implementáciu je možné vidieť na ukážke C.4.

6.3.4 Tabuľkové DDL

Parsovanie DDL dotazov, ktoré vytvárajú a upravujú štruktúru tabuliek je pre projekt Debezium najpodstatnejšie. Na základe týchto informácií generuje štruktúry správ pre zmenu dát, ktoré odosiela Kafke. Neoddeliteľnou súčasťou parsrov pre všetky DBMS je analýza dátových typov, ktoré je potrebné mapovať na JDBC[8] typy. Hodnoty odosielané v správach Kafky sú následne konvertované na základe daného JDBC typu.

Parsovanie dotazov na zmazanie tabuliek predstavuje iba vymazanie danej tabuľky z modelu. Na druhú stranu parsovanie dotazov na vytvorenie a úpravu tabuliek je o niečo zaujímavejšie (viď podkapitola 6.3.4.1). Súčasťou tabuľkových dotazov je aj dotaz na skartovanie tabuľky (TRUNCATE). Reakcia na tieto typy dotazov, ktoré vymazávajú všetky dáta z danej tabuľky, sa aktuálne v projekte Debezium rieši. Prvotnou implementáciou je iba signalizácia tohto dotazu.

6.3.4.1 Dotazy na vytvorenie a úpravu tabuliek

Na vytvorenie tabuliek v MySQL existujú tri typy SQL dotazov.

- Vytvorenie tabuľky s použitím definícii stĺpcov.
- Vytvorenie kópie existujúcej tabuľky.
- Vytvorenie tabuľky na základe definovaného vyhľadávacieho (*SELECT*) výrazu.

Vďaka tomu, že Debezium parsuje DDL výrazy, ktoré číta z MySQL binlogu, sa parser nemusí zaoberať parsovaním posledného spomenutého typu, čiže vytvorenia na základe vyhľadávacieho výrazu. Tento typ v MySQL znamená nielen vytvorenie novej tabuľky, ale zároveň jej naplnenie dátami, ktoré sú vysledkom vyhľadávacieho dotazu. Debezium pracuje z MySQL binlogom, ktorý ma nastavený zápis udalosti vo formáte *row-based*. To znamená,

[8] API, ktoré definuje jednotlivé rozhrania pre prístup k relačným databázam v Jave

že vytvorenie tabuľky aj následné vkladanie dát je rozdelené do viacerých udalostí rôznych typov. udalost na samotné vytvorenie tabuľky je v tomto prípade v binlogu uložená vo formáte klasického dotazu s definíciou stĺpcov, čo sa zhoduje s prvým typom SQL dotazu na vytvorenie tabuliek.

Vytváranie kópii existujúcej tabuľky, neprináša žiadne problémy. Definíciu existujúcej tabuľky stačí dotiahnuť z modelu a prepoužiť je štruktúru na vytvorenie novej tabuľky. Posledným typom na vyriešenie zostáva vytvorenie tabuľky pomocou definície stĺpcov a ich dátových typov. Parsovanie tejto definície je roznaké ako pre vytváranie tabuliek, tak aj pre ich úpravu. Z tohto dôvodu vytvoríme samostatného poslucháča, ktorý bude analyzovať definície stĺpcov. Ako som už spomínal, v rámci dátových typov je potrebné implementovať mapovanie na JDBC typ.

6.3.4.2 Mapovanie dátových typov

Na mapovanie dátových som vytvoril komponentu, ktorej primárnou úlohou bude zmienené mapovanie, ale zároveň vie genericky konštruovať názov dátového typu a taktiež umožňuje nastavenie základných hodnôt dimenzií, pre určité dátové typy. Túto komponentu je možné v implementácii nájsť pod názvom DataTypeResolver.

Súčasťou tejto komponenty je dátový objekt DataTypeEntry, ktorý udržuje nastavenie pre jeden konkrétny dátový typ. Jeho základným nastavením je definícia mapovania dátového typu pozostávajúceho z identifikátorov na tokeny, ktorými je dátový typ pomenovaný a jeho ekvivalentu v rámci JDBC typu. Toto nastavnie je povinné, takže sa definuje v rámci konštruktoru DataTypeEntry objektu. Nakoľko komponenta DataTypeResolver má za úlohu rozpoznať celý názov dátového typu, je možné v rámci DataTypeEntry definovať tokeny, ktoré sú pre daný typ nepovinné. Poslednou súčasťou tohto dátového objekty je možnosť nastavenia základných hodnôt dimenzií. Príklad inicializácie tejto komponenty je možné vidieť na ukážke C.5.

Komponenta DataTypeResolver prijíma ako vstupný parameter kontext, ktorý obsahuje definíciu dátového typu. Nakoľko dátové typy môžu spadať pod rôzne inštancie tohto kontextu, komponenta používa kanonický názov triedy kontextu na priradenie dátového typu. Týmto sa zabezpečí zrýchlenie algoritmu, nakoľko DataTypeResolver nebude musieť vždy prechádzať všetky definované DataTypeEntry, ale iba tie, ktoré spadajú pod konkrétne riešený kontext. V rámci definovania dátových typov môže nastať situácia, kedy identifikačné tokeny pre jeden dátový typ budú podmnožinou identifikačných tokenov pre iný dátový typ. Pre príklad MySQL dátový typ *VARCHAR* sa mapuje na JDBC typ *VARCHAR* a MySQL dátový typ *NATIONAL VARCHAR* sa mapuje na JDBC typ *NVARCHAR*. S týmto problémom sa komponenta vysporiada pomocou nastavených priorít, kde priorita značí počet identifikačných tokenov MySQL dátového typu. To znamená, že ak by komponenta pri vyhodnocovaní identifikovala dátový typ, uloží si tento typ ako potencionálneho víťaza a pokračuje vo vyhodnocovaní ďalej. Ak sa jej podarí identifikovať ďalší dátový typ tak sa na základe priority rozhodne ktorý je ten správny. Implementáciu vyhodnocovacieho algoritmu je možné vidieť na ukážke C.6. Výstupom tejto komponenty je objekt DataType, ktorý v sebe udržuje namapovaný JDBC dátový typ, celý názov dátového typu a prípadne nastavené základné hodnoty dimenzií dátového typu.

6.3.5 Pohľadové DDL

Parsovanie pohľadov pre Debezium aktuálne nie je veľmi dôležité. Stávajúci parser analyzoval iba dotazy na vytvorenie pohľadov. Analýza na úpravu a mazanie pohľadov nebola implementovaná vôbec. Dokonca v základnom nastavení DDL parseru je parsovanie pohľadov vypnuté úplne. Tento prístup nie je úplne správny nakoľko je možné vytvárať nové tabuľky kópiou existujúcich, do ktorých spadajú aj existujúce pohľady.

Parsovanie pohľadov nie je úplne triviálna vec, nakoľko ich definícia sa zakladá na vyhľadávacom dotaze. Problém, ktorý takého definovanie pohľadu prináša je ten, že vyhľadávací dotaz môže mať v sebe vnorený ďalší vyhľadávací dotaz a ten ďalší dotaz a tak ďalej. Ďalším problémom je možnosť definovania aliasov pre vyhľadané tabuľky prípadne ich stĺpce. To všetko je treba brať v úvahu pri snahe skonštruovať správny pohľad. Pointa konštrukcie pohľadu je založená na tom, že stĺpce, ktoré sú výsledkom vyhľadávajúceho dotazu sú súčasťou existujúcej tabuľky, v ktorej sa nachádza definícia týchto stĺpcov. Táto definícia sa prepoužije na definovanie stĺpcov v pohľade.

Počas parsovania je potrebné identifikovať vyhľadané stĺpce na základe aliasov a vnorených vyhľadávacích dotazov. Táto funkcionalita je opäť spoločná ako pre vytváranie tak pre úpravu pohľadov, a preto je súčasťou samostatného poslucháča. Algoritmus, ktorý obstaráva túto funkcionalitu postupuje od najvnorenejšieho vyhľadávacieho dotazu až k definícii pohľadu. Pri vynorovaní si vytvára nové tabuľky reprezentujúce výsledok vyhľadávajúceho dotazu. Tieto tabuľky a ich stĺpce si ukladá pod názvom rovným ich predošlému názvu alebo ich aliasu, ak je definovaný. Týmto spôsobom zabezpečíme, že počas analýzy každého vyhľadávajúceho dotazu budeme mať k dispozícii tabuľky s názvami, ktoré daný vyhľadávací dotaz používa.

6.3.6 Procedúry a funkcie

Parsovanie DDL dotazov na vytváranie procedúr a funkcií je pre projekt Debezium nepodstatné. Obsahom týchto DDL ale môžu byt dotazy, ktoré parsované sú. Z logiky gramatiky a algoritmu ktorý generovaný parser používa, sa udalosti o týchto dotazoch spropagujú poslucháčom, ktorý by ich spracovali. Aby nevznikali nekonzistentné dáta je nutné implementovať logiku, ktorou sa validné dotazy, ktoré sú súčasťou dotazov na vytvorenie procedúr a funkcií, neparsovali.

Túto funkcionalitu bude obstarávať hlavný poslucháč, ktorý rieši delegovanie udalostí. Ak sa detekuje parsovanie dotazu, ktorý sa parsovať nemá, vyšle sa hlavnému poslucháčovi signál, ktorý spustí preskakovanie všetkých uzlov sparovaného podstromu preskakovaného dotazu. Aby dokázal indikovať, kedy má s preskakovaním skončiť, použije sa premenná ktorá bude uchováva číslo, ktoré sa každým zanorením v podstrome inkrementuje a každým vynorením dekrementuje. Týmto dosiahneme, že až sa prechádzanie podstromu dostane spať do uzlu v ktorom začalo preskakovanie, bude táto hodnota natavená znovu na nulu a funkcia preskakovania sa vypne.

6.3.7 Parsovací mód

Po implementácii generovaného DDL parseru sa Debezium dostalo do stavu, že pre MySQL databázový konektor existujú dve možné riešenia na parsovanie DDL dotazov. Aby mal

užívateľ možnosť si zvoliť, ktorý parser bude používať, vznikla nové nastavenie konektoru, ktoré definuje parsovací mód. Pri inicializácii konektoru, môže užívateľ použiť nastavenie *ddl.parser.mode*, ktoré prijíma hodnoty:

- **antlr**, ktorou nastaví použitie DDL parsru generovaného ANTLR nástrojom.
- **legacy**, ktorou nastaví použitie ručne napísaného DDL parsru.

Pokiaľ užívateľ nešpecifikuje mód, použije sa základné nastavenie, ktorým je parsovanie ANTLR parserom.

Hodnota tohto nastavenia sa mapuje na javovský enum, pomocou ktorého je implementovaný návrhový vzor továrne (anglicky factory pattern), ktorý priradí konektoru správnu inštanciu DdlParser rozhrania.

6.4 Testovanie

Neoddeliteľnou súčasťou implementácie sú automatické testy, ktoré dopomáhajú v odhalení chýb počas implementácie alebo následných úpravách kódu. Testovanie MySQL DDL parseru generovaného z gramatiky pomocou ANTLR pozostáva z unit a integračných testov a testovania vytvorenej gramatiky MySQL.

6.4.1 Unit a integračné testy

Projekt Debezium aktuálne obsahuje veľké množstvo unit a integračných testov, ktoré pokrývajú implementácie stávajúceho parseru. Niektoré z týchto testov boli vytvorené na základe identifikovaných problémov užívateľmi projektu. Z tohto dôvodu považujem tieto testy za dostačujúce na testovanie novej implementácie generovaného parseru.

Počas implementácie sa mi podarilo odhaliť niekoľko chýb a nedostatkov stávajúcej implementácie. Nakoľko mojou úlohou bola implementácia nového parseru a nie oprava stávajúceho riešenia, tieto chyby boli evidované a v rámci novej implementácie taktiež opravené. Bohužiaľ opravou týchto chýb sa prišlo na to, že niektoré aktuálne testy museli byt upravené, nakoľko počítali z chybnou implementáciou. Jedným z problémov bola aj benevolencia stávajúceho parseru, ktorý pokiaľ detekoval dotaz, ktorý bol pre neho nepodstatný, preskočil ho. Takéto dotazy v boli v testoch väčšinou napísané nevalidne voči syntaxi MySQL. Ich parsovanie pomocou novej implementácie samozrejme zlyhalo, nakoľko generovaný parser sa riadi definovanou gramatikou, ktorej nevalidný dotaz nevyhovoval.

Pri odhalení nedostatkov sa ukázalo, že stávajúce riešenie neparsovalo SQL dotazy, ktoré by malo, ako napríklad dotazy na skartovanie (TRUNCATE) tabuliek, alebo úpravu a mazanie pohľadov. Pre tieto typy dotazov boli pre novú implementáciu parseru napísané nové testy, ktoré kontrolujú správnosť ich parsovania.

Integračné testy projektu Debezium pracujú nad reálnou MySQL databázou a kontrolujú výsledné správy o zmenách, ktoré sa dostanú do Kafky. V rámci týchto testov sa inicializuje MySQL konektor, u ktorého je po novom možné nastaviť parsovací mód, vďaka ktorému je možné otestovať fungovanie konektoru s implementáciou ANTLR parseru.

6.4.2 Testovanie gramatiky

Bohužiaľ pri použití gramatiky MySQL, poskytnutej autormi ANTLR nástroja sa ukázalo, že aj táto gramatika má svoje diery. Preto bolo nutné gramatiku upraviť a nie je vylúčené, že sa môžu objaviť ďalšie problémy, kvôli ktorým bude nutný zásah do gramatiky. Z tohto dôvodu bola vytvorená sada SQL dotazov, pomocou ktorej sa kontroluje správnosť gramatiky počas jej generovania. Na toto testovanie bol použitý maven plugin, ktorý poskytuje ANTLR[9].

V konfigurácii tohto pluginu sa nastavuje:

- názov gramatiky, ktorá má byť testovaná
- vstupné pravidlo testovanej gramatiky
- balíček v ktorom sa nachádzajú vygenerované triedy parseru
- nastavenie veľkosti písmen, ktorými je definovaná gramatika lexeru
- cesta k sade testovacích SQL dotazov.

Príklad nastavenia tohto maven pluginu je možné vidieť na ukážke 6.4

Ukážka 6.4: Príklad konfigurácie maven pluginu na testovanie gramatiky

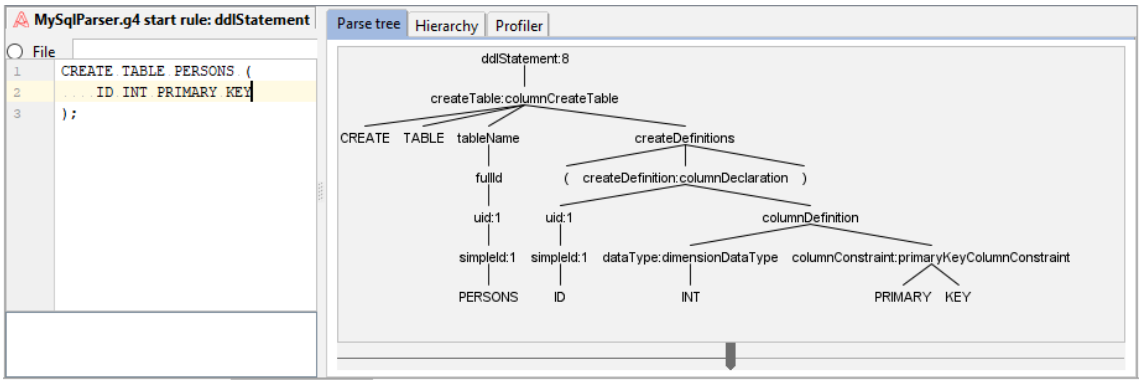
```
<plugin>
    <groupId>com.khubla.antlr</groupId>
    <artifactId>antlr4test-maven-plugin</artifactId>
    <version>1.10</version>
    <configuration>
        <entryPoint>root</entryPoint>
        <grammarName>MySql</grammarName>
        <packageName>io.debezium.ddl.parser.mysql.generated</packageName>
        <caseInsensitiveType>UPPER</caseInsensitiveType>
        <exampleFiles>src/test/resources/mysql/examples</exampleFiles>
    </configuration>
</plugin>
```

6.4.3 ANTLR works nástroj

Počas implementácie alebo prípadných úpravách gramatiky je ANTLR works nástroj obrovským pomocníkom. Po jeho nainštalovaní do svojho vývojového prostredia sa napríklad sprístupni zvýrazovnovanie a kontrola syntaxi pri písaní gramatiky, umožní navigovanie v pravidlách gramatiky a podobne. Najdôležitejou súčastou je možnosť vizualizácie parsovacieho stromu pre konkrétne pravidlá gramatiky, ktorý sa automaticky generuje počas vkládania parsovaného reťazca. V rámci výsledného stromu sa dá vidieť ako lexer rozdelil vstupný reťazec na jednotlivé tokeny a aké pravidlá gramatiky boli uplatené. V reálnom čase

[9] <https://github.com/antlr/antlr4test-maven-plugin>

taktiež upozornuje na chyby, ktoré počas parsovania vstupného reťazca nastali. Súčasťou týchto upozornení je napríklad označenie prvého tokenu, kvôli ktorému sa nepodarilo uplatniť žiadne pravidlo gramatiky, alebo označenia pozície v reťazci, kde očakával konkrény typ tokenu. Na nasledujúcom obrázku 6.4 je možné vidieť ukážku sparsovaného stromu.



Obr. 6.4: Príklad sparsovaného stromu pomocou nástroja ANTLR works

Súčasťou nástroje je taktiež profiler gramatiky, pomocou ktorého sa dá analyzovať, ktoré rozhodnutiia v gramatike sú komplikované alebo drahé. Profilovacie údaje sú k dispozícii vždy rovnako ako sparsovaný strom počas interpretácii gramatiky. Pre bližšie informácie o možnostiach tohto nástroja, konkrétne pre vývojarské prostredie IntelliJ IDEA, doporučujem zhliadnuť jeho dokumentáciu v GitHub repositáre[10] ANTLR.

[10] <https://github.com/antlr/intellij-plugin-v4>

Kapitola 7

Záver

Cieľom tejto práce bolo zanalyzovať možnosti nahradenia stávajúceho DDL parseru pre MySQL, ktorý v projekte Debezium odchytáva zmeny v databázových štruktúrach. Aktuálne riešenie nebolo dostačujúce na spracovanie všetkých možných nuancí jazyka MySQL a náročnosť prípadných úprav rástla z veľkosťou jeho implementácie. V rámci analýzy stávajúceho parsru boli objavené nekonzistencie voči MySQL syntaxi, čo znamená, že parser príjmal aj nevalidné SQL dotazy.

Analýzou možností, ako by stávajúci parser mohol byť nahradení sa ukázalo, že parsovanie je teoreticky vyriešený problém, no v praxi sa tento problém stále znovu rieši. To znamená, že existuje vela rôznych algoritmov, každý so silnými a slabými stránkami a stále sa vylepšujú. Pre správnu voľbu parsovacieho algoritmu existuje niekoľko faktorov, ktoré je potrebné brať v úvahu. Najdôležitejším z nich je uvedomiť si do akejého jazyka spadá gramatika, ktorú sa snažíme parsovať. Jazyk MySQL spadá pod bezkontextové jazyky nakoľko obsahuje možnosti rekurzívnych pravidiel. Väčšina algoritmov, ktorá dokáže parsovať bezkontextové jazyky vyžaduje úpravu gramatiky, ako napríklad odstránenie ľavých rekurzií u LL algoritmov.

Vďaka neustálemu rozširovaniu možností parsovania, ale už existujú nástroje, ktoré odkážu niektoré z týchto úprav vykonať, takže autor gramatiky ich môže ignorovať. Nástroj ANTLR verzie 4, ktorý som si zvolil pre riešenie nového parseru, používa aktuálne najnovší parsovací algoritmus ALL(*), ktorý je postavený na LL algoritmoch a bol vyvinutý v roku 2014 práve autorom nástroja ANTLR. Pri generovaní parseru z gramatiky sa ANTLR dokáže vysporiadať z pravidlami obsahujúcimi priamu ľavú rekurziu, no autor si stále musí dávať pozor na nepriamú ľavú rekurziu.

Pri návrhu implementácie nového parseru, generovaného nástrojom ANTLR sa taktiež ukázali nedostatky v stávajúcom návrhu, ktorý nebol dostatočne pripravený na možnosti rozšírenia parsovania iným spôsobom. Z toho dôvodu bolo nutné upraviť aktuálny návrh tak, aby bolo možné využiť čo najväčšiu časť stávajúcej implementácie. Projekt Debezium bude v budúcnosti rozširovať podporu databázových systémov, ktoré bude možné sledovať, a preto sa pri návrhu počítalo z použitím ANTLR nástroja pri parsovaní iných DBMS. Základná implementácia, ktorá by mala byť spoločná pre všetky ANTLR parsre je implementovaná v samostatnom module, ktorý sa taktiež stará o generovanie parserov z daných gramatík.

Funkcionalita implementácie nového parseru bola úspšne overená testvacou sadou Debezia. V niektorých prípadoch bolo potrebné túto sadu upraviť a to najmä zo spomínaného

dôvodu, že predchodzia implementácia povoľovala parsovanie syntakticky nevalidných SQL dotazov. V rámci nového ANTLR parseru boli taktiež implementované opravy nájdených chýb a parsovanie SQL dotazov, ktoré v bývalej implementácii neboli parsované. Na základe týchto novo parsovaných dotazov bola stávajúca sada rozšírená. Súčasťou novej sady je taktiež kontrola správnosti gramatiky, ktorá môže byť v budúcnosti upravovaná.

V tejto práci sa mi podarilo zanalyzovať možnosti implementácie generovaného parseru a taktiež takýto parser implementovať pre budúce potreby projektu Debezium. Táto implementácia prináša radu výhod pre projekt Debezium a to najmä v zmysle údržby parseru a prípadnej implementácie parseru pre iný databázový systém. Analýza jednotlivých sparsovaných dotazov je rozdelená do viacerých tried, čo prináša väčšiu prehľadnosť a jednoduhšiu orientáciu pre vývojarov, ktorý s DDL parserom prídu do styku.

Literatúra

[1] AHO, A. V. – ULLMAN, J. D. *The theory of parsing, translation, and compiling.* 1. Prentice-Hall Englewood Cliffs, NJ, 1972.

[2] Attunity Ltd. Attunity Connect Product White Paper. *Change Data Capture – next generation ETL.* Máj 2004.

[3] BLOCH, J. *Effective java.* Addison-Wesley Professional, 2017. ISBN 978-0321356680.

[4] Debezium Community. *Debezium* [online]. 2018. [cit. 28. 1. 2018]. Dostupné z: <http://debezium.io/>.

[5] Debezium Community. *Debezium Connector for MySQL: Snapshots* [online]. 2018. [cit. 22. 2. 2018]. Dostupné z: <http://debezium.io/docs/connectors/mysql/#snapshots>.

[6] DEMLOVÁ, M. Jazyky, automaty a gramatiky. *Konečné automaty.* 2017.

[7] DEMLOVÁ, M. Jazyky, automaty a gramatiky. *Gramatiky.* 2017.

[8] DENNY, J. E. – MALLOY, B. A. IELR (1): practical LR (1) parser tables for non-LR (1) grammars with conflict resolution. In *Proceedings of the 2008 ACM symposium on Applied computing.* ACM, 2008.

[9] DEREMER, F. L. Simple LR (k) grammars. *Communications of the ACM.* 1971.

[10] DEREMER, F. L. *Practical translators for LR (k) languages.* PhD thesis, Massachusetts Institute of Technology, 1969.

[11] EARLEY, J. An efficient context-free parsing algorithm. *Communications of the ACM.* 1970.

[12] FORD, B. *Packrat parsing: a Practical Linear-Time Algorithm with Backtracking.* PhD thesis, Massachusetts Institute of Technology, 2002.

[13] FORD, B. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices.* ACM, 2004.

[14] GRUNE, D. – JACOBS, C. *Parsing Techniques: A Practical Guide.* Monographs in Computer Science. Springer New York, 2007. ISBN 9780387689548.

[15] HABERMAN, J. *LL and LR in Context: Why Parsing Tools Are Hard* [online]. [cit. 25. 4. 2018]. Dostupné z: <http://blog.reverberate.org/2013/09/ll-and-lr-in-context-why-parsing-tools.html>.

[16] HABERMAN, J. *LL and LR Parsing Demystified* [online]. [cit. 21. 4. 2018]. Dostupné z: <http://blog.reverberate.org/2013/07/ll-and-lr-parsing-demystified.html>.

[17] ISO 14977:1996(E). Information technology – Syntactic metalanguage – Extended BNF. Standard, International Organization for Standardization, Geneva, CH, 1996.

[18] KARP, R. M. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *IFIP Congress (1)*, 1992.

[19] KASAMI, T. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*. 1966.

[20] KNUTH, D. E. On the translation of languages from left to right. *Information and control*. 1965.

[21] MIGHT, M. *The language of languages* [online]. [cit. 3. 4. 2018]. Dostupné z: <http://matt.might.net/articles/grammars-bnf-ebnf>.

[22] MOORE, R. C. Removing left recursion from context-free grammars. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*. Association for Computational Linguistics, 2000.

[23] MORLING, G. Streaming Database Changes with Debezium, 2017. `https://www.youtube.com/watch?v=I0Z2Um6e430`, publikované 9. 11. 2017.

[24] MÜLLER, K. *Programovaí jazyky*. Česká technika - nakladatelství ČVUT, 2005. ISBN 80-01-02458-X.

[25] NARKHEDE, N. – SHAPIRA, G. – PALINO, T. *Kafka: The Definitive Guide: Real-time data and stream processing at scale*. O'Reilly UK Ltd., 2017. ISBN 978-1-491-99065-0.

[26] *MySQL 5.7 Reference Manual*. Oracle Corporation and/or its affiliates, 2018. `https://dev.mysql.com/doc/refman/5.7/en/`.

[27] PARR, T. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[28] PARR, T. – FISHER, K. LL (*): the foundation of the ANTLR parser generator. *ACM Sigplan Notices*. 2011.

[29] PARR, T. – HARWELL, S. – FISHER, K. Adaptive LL (*) parsing: the power of dynamic analysis. In *ACM SIGPLAN Notices*. ACM, 2014.

[30] Randall Hauch. *Parsing DDL* [online]. 2018. [cit. 25. 2. 2018]. Dostupné z: <http://debezium.io/blog/2016/04/15/parsing-ddl/>.

[31] ROCHE, E. – SCHABES, Y. *Finite-state language processing*. MIT press, 1997. ISBN 978-0262181822.

[32] SIPSER, M. *Introduction to the Therory of Computation*. Boston: Thomson Course Technology, 2 edition, 2006. ISBN 0-534-95097-3.

[33] TOMASSETTI, G. *A Guide to Parsing: Algorithms and Terminology* [online]. 9 2017. [cit. 28. 3. 2018]. Dostupné z: <https://tomassetti.me/guide-parsing-algorithms-terminology>.

Dodatok A

Zoznam použitých skratiek

ALL Adaptive LL

BNF Backus-Naur Form

CDC Change Data Capture

CFG context-free grammar

DBMS Database management system

DDL Data definition language

DFA Deterministic Finite Automaton

EBNF Extended Backus-Naur Form

JDBC Java database connection

LALR Lookahead LR

LL Left-to-right read of the input, Leftmost derivation

LR Left-to-right read of the input, Rightmost derivation

PEG Parsing Expression Grammar

SLR Simple LR

SQL Structured Query Language

Dodatok B

Ukážka dát

Ukážka B.1: Ukážka CDC správy odoslanej Debeziom

```
{
        "schema" : {
                ...
        },
        "payload" : {
                "before" : null,
                "after" : {
                        "id": 352,
                        "name" : "Janko",
                        "surename" : "Hrasko",
                        "email" : "janko@hrasko.sk"
                },
                "source" : {
                        "name" : "dbserver1",
                        "serer_id" : 0,
                        "ts_sec" : 0,
                        "file" : "mysql-bin.000001",
                        "pos" : 12,
                        "row" : 0,
                        "snapshot" : true,
                        "db" : "todo_list",
                        "table" : "users"
                },
                "op" : "c",
                "ts_ms" : 1517152654614
        }
}
```

Dodatok C

Ukážky zdrojových kódov

Ukážka C.1: Parsovacie metódy DDL parserov

```
1  public final void parse(String ddlContent, Tables databaseTables) {
2          Tokenizer tokenizer = new DdlTokenizer(!skipComments(), this::determineTokenType);
3          TokenStream stream = new TokenStream(ddlContent, tokenizer, false);
4          stream.start();
5          parse(stream, databaseTables);
6      }
7
8  public final void parse(TokenStream ddlContent, Tables databaseTables) throws
       ↪ ParsingException, IllegalStateException {
9          this.tokens = ddlContent;
10         this.databaseTables = databaseTables;
11         Marker marker = ddlContent.mark();
12         try {
13             while (ddlContent.hasNext()) {
14                 parseNextStatement(ddlContent.mark());
15                 // Consume the statement terminator if it is still there ...
16                 tokens.canConsume(DdlTokenizer.STATEMENT_TERMINATOR);
17             }
18         } catch (ParsingException e) {
19             ddlContent.rewind(marker);
20             throw e;
21         } catch (Throwable t) {
22             parsingFailed(ddlContent.nextPosition(), "Unexpected exception (" + t.getMessage()
                   ↪ + ") parsing", t);
23         }
24     }
```

Ukážka C.2: Implementácia parseNextStatement metódy v MySqlDdlParser

```
@Override
    protected void parseNextStatement(Marker marker) {
        if (tokens.matches(DdlTokenizer.COMMENT)) {
            parseComment(marker);
        } else if (tokens.matches("CREATE")) {
            parseCreate(marker);
        } else if (tokens.matches("ALTER")) {
            parseAlter(marker);
        } else if (tokens.matches("DROP")) {
            parseDrop(marker);
        } else if (tokens.matches("RENAME")) {
            parseRename(marker);
        } else {
            parseUnknownStatement(marker);
        }
    }
```

Ukážka C.3: Implementácia parseCreateTable metódy v MySqlDdlParser

```
protected void parseCreateTable(Marker start) {
        tokens.canConsume("TEMPORARY");
        tokens.consume("TABLE");
        boolean onlyIfNotExists = tokens.canConsume("IF", "NOT", "EXISTS");
        TableId tableId = parseQualifiedTableName(start);
        if (tokens.canConsume("LIKE")) {
            TableId originalId = parseQualifiedTableName(start);
            Table original = databaseTables.forTable(originalId);
            if (original != null) {
                databaseTables.overwriteTable(tableId, original.columns(),
                    ↪ original.primaryKeyColumnNames(), original.defaultCharsetName());
            }
            consumeRemainingStatement(start);
            signalCreateTable(tableId, start);
            debugParsed(start);
            return;
        }
        if (onlyIfNotExists && databaseTables.forTable(tableId) != null) {
            // The table does exist, so we should do nothing ...
            consumeRemainingStatement(start);
            signalCreateTable(tableId, start);
            debugParsed(start);
            return;
        }
        TableEditor table = databaseTables.editOrCreateTable(tableId);
        // create_definition ...
        if (tokens.matches('(')) parseCreateDefinitionList(start, table);
        // table_options ...
        parseTableOptions(start, table);
        // partition_options ...
        if (tokens.matches("PARTITION")) {
            parsePartitionOptions(start, table);
        }
        // select_statement
        if (tokens.canConsume("AS") || tokens.canConsume("IGNORE", "AS") ||
            ↪ tokens.canConsume("REPLACE", "AS")) {
            parseAsSelectStatement(start, table);
        }
        // Make sure that the table's character set has been set ...
        if (!table.hasDefaultCharsetName()) {
            table.setDefaultCharsetName(currentDatabaseCharset());
        }
        // Update the table definition ...
        databaseTables.overwriteTable(table.create());
        signalCreateTable(tableId, start);
        debugParsed(start);
    }
```

Ukážka C.4: Implementácia metódy na vymazanie tabuliek

```
1  public void removeTablesForDatabase(String catalogName, String schemaName) {
2          lock.write(() -> {
3              tablesByTableId.entrySet().removeIf(tableIdTableEntry -> {
4                  TableId tableId = tableIdTableEntry.getKey();
5                  boolean equalSchema = schemaName == null && tableId.schema() == null
6                          || schemaName != null && schemaName.equals(tableId.schema());
7                  boolean equalCatalog = catalogName == null && tableId.catalog() == null
8                          || catalogName != null && catalogName.equals(tableId.schema());
9                  return equalSchema && equalCatalog;
10             });
11         });
12     }
```

Ukážka C.5: Inicializácia komponenty DataTypeResolver

```
1  dataTypeResolver.registerDataTypes(
       ↪ MySqlParser.StringDataTypeContext.class.getCanonicalName(), Arrays.asList(
2                  new DataTypeEntry(Types.CHAR, MySqlParser.CHAR),
3                  new DataTypeEntry(Types.BINARY, MySqlParser.CHAR,
                       ↪ MySqlParser.BINARY)
4          ));
5  dataTypeResolver.registerDataTypes(
       ↪ MySqlParser.DimensionDataTypeContext.class.getCanonicalName(), Arrays.asList(
6                  new DataTypeEntry(Types.DOUBLE, MySqlParser.DOUBLE)
7                          .setSuffixTokens(MySqlParser.PRECISION, MySqlParser.SIGNED,
                               ↪ MySqlParser.UNSIGNED, MySqlParser.ZEROFILL),
8                  new DataTypeEntry(Types.DECIMAL, MySqlParser.DECIMAL)
9                          .setSuffixTokens(MySqlParser.SIGNED, MySqlParser.UNSIGNED,
                               ↪ MySqlParser.ZEROFILL)
10                         .setDefaultLengthScaleDimension(10, 0)
11         ));
```

Ukážka C.6: Implementácie rozhodovacieho algoritmu komponenty DataTypeResolver

```
1  public DataType resolveDataType(ParserRuleContext dataTypeContext) {
2          DataType dataType = null;
3          // use priority according to number of matched tokens
4          int selectedTypePriority = −1;
5          for (DataTypeEntry dataTypeEntry :
               ↪ contextDataTypesMap.get(dataTypeContext.getClass().getCanonicalName())) {
6              int dataTypePriority = dataTypeEntry.getDbmsDataTypeTokenIdentifiers().length;
7              if (dataTypePriority > selectedTypePriority) {
8                  DataTypeBuilder dataTypeBuilder = new DataTypeBuilder();
9                  boolean correctDataType = true;
10                 for (Integer mainTokenIdentifier :
                       ↪ dataTypeEntry.getDbmsDataTypeTokenIdentifiers()) {
11                     TerminalNode token = dataTypeContext.getToken(mainTokenIdentifier, 0);
12                     if (correctDataType) {
13                         if (token == null) {
14                             correctDataType = false;
15                         }
16                         else {
17                             dataTypeBuilder.addToName(token.getText());
18                         }
19                     }
20                 }
21                 if (correctDataType) {
22                     dataType = buildDataType(dataTypeContext, dataTypeEntry,
                           ↪ dataTypeBuilder);
23                     selectedTypePriority = dataTypePriority;
24                 }
25             }
26         }
27         if (dataType == null) {
28             throw new ParsingException(null, "Unrecognized dataType for " +
                   ↪ AntlrDdlParser.getText(dataTypeContext));
29         }
30         return dataType;
31     }
```

Dodatok D

Postup inštalácie ANTLR nástroja

Ukážka D.1: Kroky inštalácie ANTLR nástroja pre Windows

```
1  # 1. Copy antlr-4.7.1-complete.jar in C:\Program Files\Java\libs (or wherever you
       ↪ prefer)
2  # 2. Create or append to the CLASSPATH variable the location of antlr
3  # you can do to that by pressing WIN + R and typing sysdm.cpl, then selecting
       ↪ Advanced (tab) > Environment variables > System Variables
4  # CLASSPATH -> .;C:\Program Files\Java\libs\antlr-4.7.1-complete.jar;%
       ↪ CLASSPATH%
5  # 3. Add aliases
6  # create antlr4.bat
7  java org.antlr.v4.Tool \%*
8  # create grun.bat
9  java org.antlr.v4.gui.TestRig \%*
10 # put them in the system PATH or any of the directories included in your PATH
```

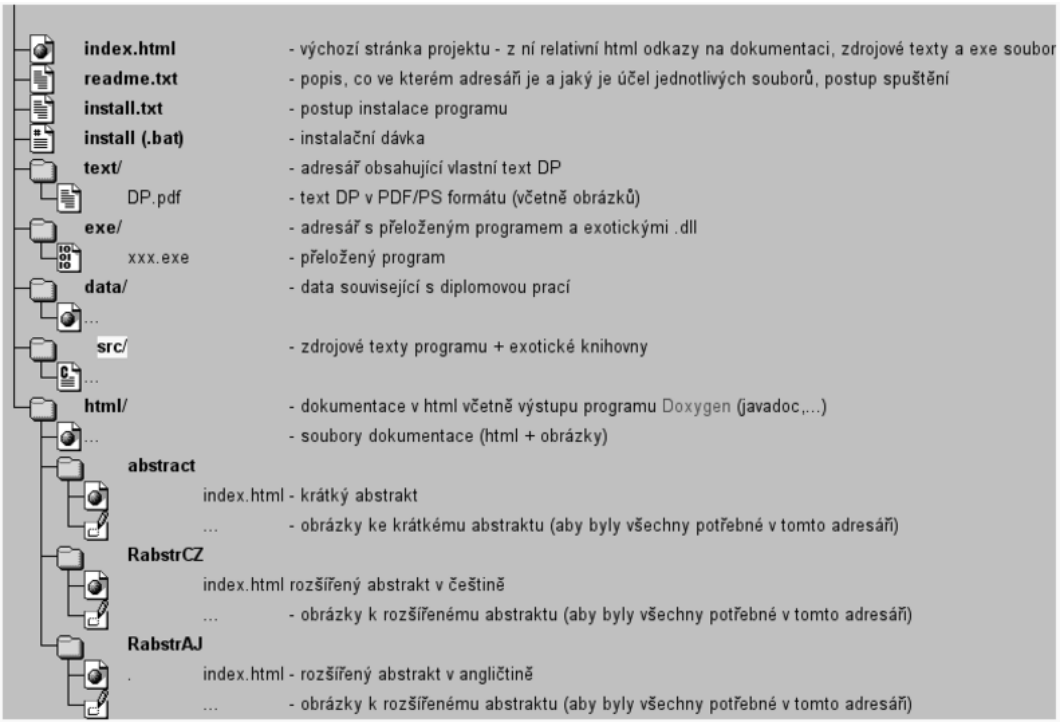
Ukážka D.2: Kroky inštalácie ANTLR nástroja pre Linux/Mac OS

```
1  # 1.
2  sudo cp antlr-4.7.1-complete.jar /usr/local/lib/
3  # 2. and 3.
4  # add this to your .bash_profile
5  export CLASSPATH = ".:/usr/local/lib/antlr-4.7.1-complete.jar:$CLASSPATH"
6  # simplify the use of the tool to generate lexer and parser
7  antlr4='java -Xmx500M -cp "/usr/local/lib/antlr-4.7.1-complete.jar:$CLASSPATH"
       ↪ org.antlr.v4.Tool'
8  # simplify the use of the tool to test the generated code
9  alias grun='java org.antlr.v4.gui.TestRig'
```

Dodatok E

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále. Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce. (viz [**?**]): Na GNU/Linuxu si strukturu přiloženého CD



Obr. E.1: Seznam přiloženého CD — příklad

můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případne index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.