

České vysoké učení technické v Praze  
Fakulta elektrotechnická  
Katedra počítačů



Diplomová práce

**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

6. mája 2018



# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
1.1	Motivácia . . . . .	1
1.2	Cieľ práce . . . . .	2
<b>2</b>	<b>Debezium</b>	<b>3</b>
2.1	Zachytávanie zmenených dát . . . . .	3
2.1.1	Dátové sklady . . . . .	4
2.1.2	Replikácia dát . . . . .	4
2.1.3	Microservice Architecture . . . . .	4
2.1.4	Ostatné . . . . .	5
2.2	Odchytávanie zmien v databázi . . . . .	5
2.2.1	Apache Kafka . . . . .	5
2.2.2	Infraštruktúra správ pomocou Apache Kafka . . . . .	6
2.2.3	Kafka Connect . . . . .	7
2.2.4	Štruktúra správy . . . . .	8
<b>3</b>	<b>Aktuálne riešenie MySQL konektoru</b>	<b>11</b>
3.1	MySQL konektor . . . . .	11
3.1.1	Binárny log . . . . .	11
3.1.2	Aktuálny obraz tabuliek . . . . .	14
3.2	DDL parser . . . . .	15
3.2.1	Framework na parsovanie DDL . . . . .	15
3.2.2	Implementácia MySQL DDL parsru . . . . .	16
<b>4</b>	<b>Syntaktická analýza</b>	<b>19</b>
4.1	Teória parsovania . . . . .	19
4.1.1	Deterministický konečný automat . . . . .	20
4.1.2	Regulárny výraz . . . . .	21
4.1.3	Bezkontextový jazyk . . . . .	21
4.1.4	Backus-Naur Form notácia . . . . .	23
4.1.5	Rozšírená Backus-Naur Form notácia . . . . .	23
4.1.6	Parsing Expression Grammar . . . . .	24
4.2	Parsovanie pomocou regulárnych výrazov . . . . .	24
4.3	Štruktúra bezkontextových parserov . . . . .	24
4.3.1	Lexer . . . . .	25

4.4	Typické problémy parsovania . . . . .	26
4.4.1	Chýbajúci token . . . . .	26
4.4.2	Pravidlá s ľavou rekurziou . . . . .	27
<b>5</b>	<b>Parsovacie algoritmy</b>	<b>29</b>
5.1	Obecný prehľad . . . . .	29
5.1.1	Rekurzívny zostup . . . . .	31
5.1.2	Lookahead a Backtracking . . . . .	31
5.1.3	Lexikálna analýza pomocou DFA . . . . .	32
5.2	LL parser . . . . .	32
5.2.1	LL gramatika . . . . .	32
5.2.2	Príklad práce LL parseru . . . . .	33
5.3	LR parser . . . . .	33
5.3.1	Simple LR a Lookahead LR . . . . .	34
5.3.2	Príklad práce LR parseru . . . . .	34
5.4	Teória proti praxi . . . . .	34
5.4.1	ANTLR alebo Bison? . . . . .	35
<b>6</b>	<b>MySQL DDL parser</b>	<b>37</b>
6.1	Základy ANTLR v4 . . . . .	37
6.1.1	Nastavenie ANTLR . . . . .	37
6.1.1.1	Maven . . . . .	38
6.1.2	ANTLR Gramatika . . . . .	39
6.1.2.1	Pravidlá pre lexer . . . . .	39
6.1.2.2	Dostupné gramatiky . . . . .	40
6.1.3	Ukázková implementácia . . . . .	40
6.2	Návrh DDL parseru . . . . .	42
6.2.1	Aktuálny design . . . . .	42
6.2.2	Úprava aktuálneho designu . . . . .	44
6.2.3	ANTLR DDL parser design . . . . .	45
6.3	Implementácia . . . . .	46
<b>A</b>	<b>Zoznam použitých skratiek</b>	<b>49</b>
<b>B</b>	<b>Ukážka dát</b>	<b>51</b>
<b>C</b>	<b>Ukážky zdrojových kódov</b>	<b>53</b>
<b>D</b>	<b>Postup inštalácie ANTLR nástroja</b>	<b>57</b>
<b>E</b>	<b>Obsah príloženého CD</b>	<b>59</b>

# Zoznam obrázkov

2.1	Koncept distribúcie zmenených dát . . . . .	3
2.2	Hierarcia Apache Kafka . . . . .	6
2.3	Príklad nastavenia Kafka a Kafka connect offsetov . . . . .	7
2.4	CDC topológia z Kafka Connect . . . . .	8
4.1	Príklad parsovania a naplňovania šablóny . . . . .	19
4.2	Príklad DFA znázorneného pomocou stavového diagramu . . . . .	20
4.3	Derivačný strom gramatiky $\mathcal{G}_1$ pre reťazec $000\#111$ . . . . .	22
4.4	Spracovanie reťazca $123 + 321$ lexerom a parserom . . . . .	25
5.1	Typický parsovací strom pre výraz $A = B + C * 2; D = 1$ . . . . .	29
5.2	Postup generovania top down parseru . . . . .	30
5.3	Postup generovania bottom up parseru . . . . .	30
6.1	Diagram tried aktuálneho návrhu MySQL DDL parseru . . . . .	43
6.2	Upravený diagram tried MySQL DDL parseru . . . . .	45
6.3	Diagram tried MySQL ANLTR DDL parseru . . . . .	46
E.1	Seznam přiloženého CD — příklad . . . . .	59



# Zoznam tabuliek

5.1	Prehľad vlastností parsovacích algoritmov . . . . .	31
-----	---	----





# Zoznam ukážok

3.1	Query událost z binárneho logu MySQL . . . . .	13
3.2	Table_map a Update_rows události z binárneho logu MySQL . . . . .	13
3.3	DDL dotaz v MySQL . . . . .	16
3.4	Parsovanie dotazu pomocou MySqlDdlParseru . . . . .	16
6.1	Ukážková gramatika Hello . . . . .	41
6.2	Ukážková implementácia HelloListener.java . . . . .	41
6.3	Ukážková implementácia Main.java . . . . .	42
B.1	Ukážka CDC správy odoslanej Debeziom . . . . .	51
C.1	Parsovacie metódy DDL parserov . . . . .	53
C.2	Implementácia parseNextStatement metódy v MySqlDdlParser . . . . .	54
C.3	Implementácia parseCreateTable metódy v MySqlDdlParser . . . . .	55
D.1	Kroky inštalácie ANTLR nástroja pre Windows . . . . .	57
D.2	Kroky inštalácie ANTLR nástroja pre Linux/Mac OS . . . . .	57



# 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. Nato aby databázový systém vedel, akú akciu sa snaží užívateľ vykonať musí zanalyzovať výraz napísany 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 syntaxiu 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, 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 DDL príkazov alebo akejkoľvek inej dôležitej relačnej databázy sa môže javiť ako skľučujúca úloha. Zvyčajne databázový systém má vysoko prispôbenú gramatiku SQL a hoci výrazy jazyka manipulácie s údajmi (DML) sú často pomerne blízke štandardom, výrazy jazyka pre definíciu dát (DDL) sú zvyčajne menej 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 databázy. Použitie týchto dostupných, no nekompletných implementácií by mohlo pokryť väčšinu požiadavkov projektu Debezium, no zvyšok by musel byť doimplementovaný 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 rozsiahlych dokumentačných materiálov. Projekt Debezium plánuje v budúcnosti rozširovať množstvo podporovaných databázových systémov, a preto by výsledok tejto práce mal byť implementovaný čo najpriateľnejšie voči jeho potencionálnemu prepoužitiu pri syntaktických analyzátoroch iných databázových systémov.

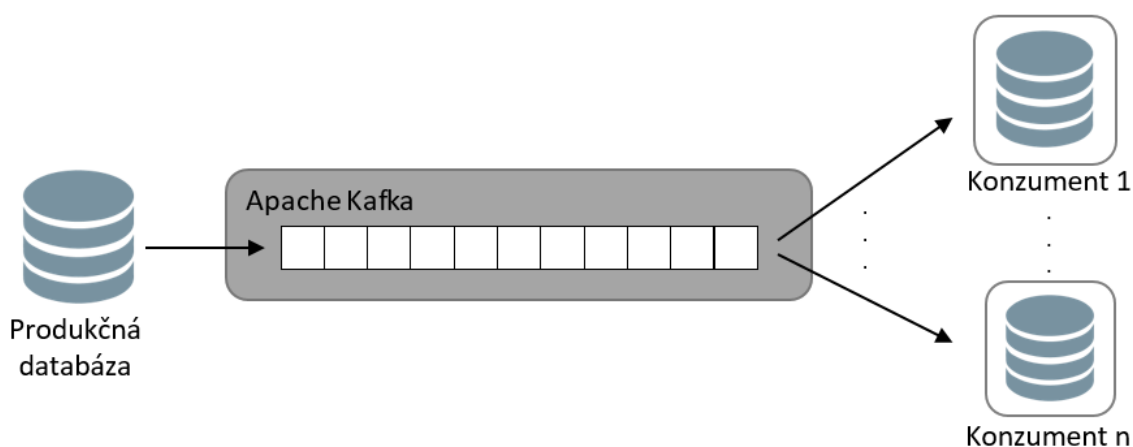
## Kapitola 2

# Debezium

Debezium[3] je projekt, ktorý slúži k zaznamenávaniu zmien v databázi analýzou udalostí v transakčnom logu. Jednou z podporovaných databázi je taktiež MySQL. Na správnu funkcionálnu 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 tabulkách 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 vid' obrázok 2.1 [20]. V projekte Debezium sa na sprostredkovanie sledu udalostí využíva Apache Kafka [22] 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 databází za účelom analytických výpočtov. Aktuálne najpoužívanším riešením na napĺňanie dátových skladov je ETL<sup>1</sup>. 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 naplňajú 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. [1]

### 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 implementácii 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 databázy 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, ktorý 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 databázy, kde si už marketingový tím 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 micro 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<sup>2</sup>, 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<sup>3</sup>. Produktová a skladová služba budú poskytovať sled zmenených dát a objednávacia aplikácia ich bude

---

<sup>1</sup>Export, transform, load

<sup>2</sup><[https://cs.wikipedia.org/wiki/Representational\\_State\\_Transfer](https://cs.wikipedia.org/wiki/Representational_State_Transfer)>

<sup>3</sup><<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 objednávací aplikačný systém 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 invalidácie, pokiaľ sa isté dáta zmenia.

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

## 2.2 Odchytávanie zmien v databázi

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 udržiujú 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 aplikáciám, ktorá do databázy 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. Každéj správe Kafka priradí sekvenčné identifikačné číslo nazývané **offset**, ktoré unikátne identifikuje každý záznam a jeho poradie v topiku.

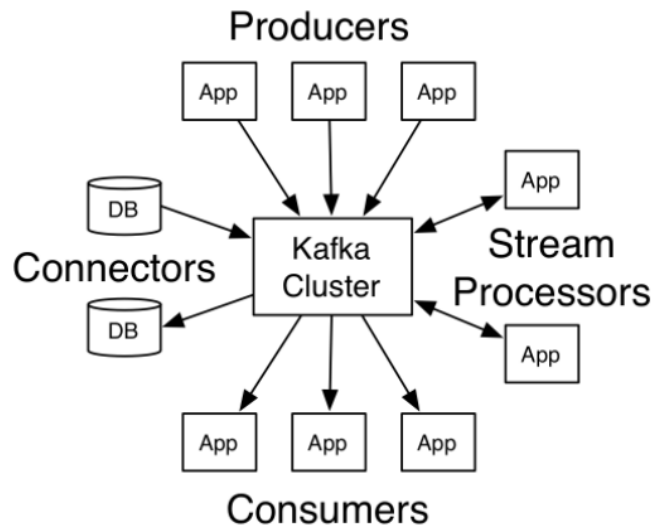
Kafka pozostáva zo štyroch základných API:[22]

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

<sup>4</sup><<http://lucene.apache.org/solr/>>

<sup>5</sup><<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 prepoužiteľné dátové spojenia na publikovanie a konzumovanie záznamov, ktoré pripoja topiky k existujúcim aplikáciám alebo dátovým systémom.



Obr. 2.2: Hierarcia 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 semantické 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<sup>6</sup>, 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 ináč 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

<sup>6</sup>First in First out

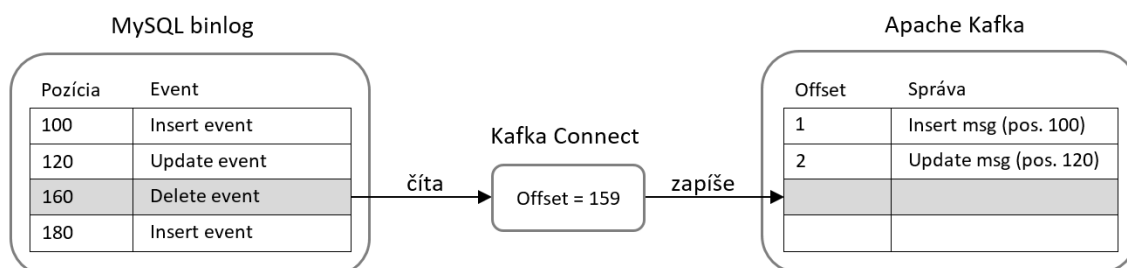


ďalších správ. Takto môže sledovať aktuálne pribúdajúce správy, ale taktiež sa môže zaujímať aj o správy z minulosti.

Zmien v databázach môže byť veľmi veľa, čo spôsobí veľké množstvo udalostí, a preto je nutné spomnúť 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 z 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 byť kozumentom 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 rep. import) konektory alebo *sink* (vysupujúce dáta resp. export) konektory. Debeziové konektory majú na starosti naplňovanie Kafky, takže sa používa *source* konektor. Kafka Connect ponúka možnosť na riešenie offsetu. Narozdiel od Kafka offsetu, ktorý je priradený každej správe v topiku, Kafka Connect offset si udržiava informáciu o pozícii poslednom prečítanom evente z binlogu. Môže nastať situácia, že konektor zhvaruje a bude musieť byť reštartovaný. V takomto prípade konektor potrebuje vedieť ako ďaleko v čítaní logu bol a kde má s čítaním pokračovať.

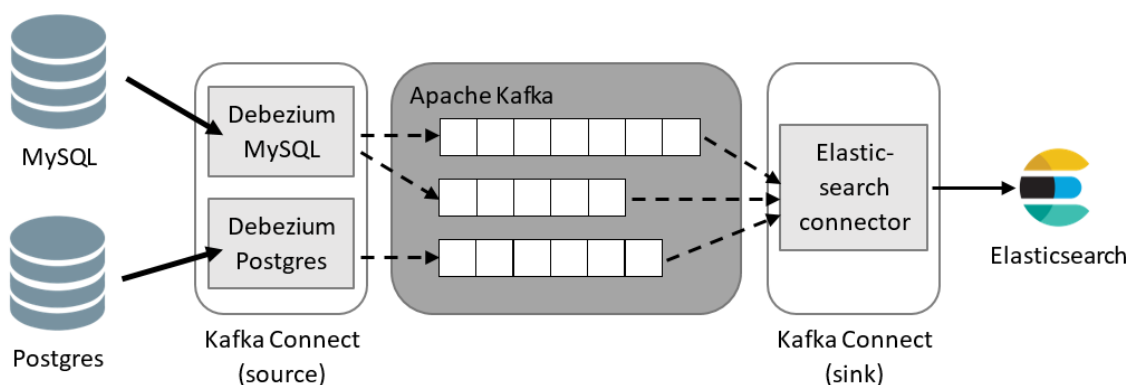


Obr. 2.3: Príklad nastavenia Kafka a Kafka connect offsetov

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 situácii 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 špecifikácii rozdeliť konektor a jeho tasky medzi viacero uzlov. Taktiež ponúka bohatý eko-systém konektorov. Na stránkach Confluent<sup>7</sup> je možné si stiahnuť rôzne typy či už *sink* alebo *source* konektorov. Príklad CDC topológie s použitím Kafka Connect je na obrázku 2.4 [20]. 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 instance 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 v obsahujú metadáta o pôvode danej zmeny. Tieto dáta sú závislé na type databáze, ktorá sa sleduje. V MySQL sa napríklad skladá z informácií ako meno databázového serveru, názvu logovacieho súboru, z ktorého číta a pozíciu v ňom, názvu databázy a tabuľky, timestamp a pod.

<sup>7</sup><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. Vďaka použitiu Kafka Connect je opäť možnosť využiť konvertory, ktoré poskytuje a 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 nieje 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 schém, 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 zkladá z viacerých častí. Hlavnou časťou je modul systému, ktorý je spoločný pre všetky typy konektorov podporovaných Debeziom. Tento modul zaobstaráva základnú funkcionálnu spojenú s CDC, ktorú tento systém podporuje. Definuje model sledovaných dát, na základe ktorých si systém udržiava aktuálne schémata tabuliek a ich dátový stav v pamäti. Jedným z týchto podporovaných konektorov je aj konektor pre MySQL databázu [3.1](#).

### 3.1 MySQL konektor

Minimálnou podporovanou verziou MySQL je aktuálne verzia 5.6. MySQL konektor Debezium 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[[23](#)]. Binlog obsahuje všetky udalosti, ktoré popisujú zmeny vykonané nad MySQL databázou ako napríklad vytváranie tabuliek alebo zmena dát. Poradie týchto udalostí je zachované voči reálnemu poradiu ako boli SQL dotazy vykonávané. Toto binárne logovanie sa využíva na dva základne účely:

- Pre **replikáciu**, kde binlog na master replikačnom serveri sprostredkúva záznamy o zmenách, ktoré majú byť odoslané slave serverom. 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** anlicky nazývanú **recovery**. Po nahraní zálohy databáze sú znovu spustené udalosti zaznamenané v binlogu, ktoré nastali po vytvorení zálohy, čím sa zabezpečí konzistentný stav databázových dát z dátami v dobe zlyhania databázového serveri.

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 logovania môže taktiež obsahovať dotazy, ktoré môžu iba potencionálne meniť dáta napríklad DELETE dotaz, ktorý sa nespáruje so žiadnymi dátami. Pri replikácii slave server číta binlog a zároveň vykonáva SQL dotazy, ktoré obsahujú jednotlivé udalosti.
- **Row-based** logovanie, v ktorom udalosti popisujú zmeny pre jednotlivé riadky v tabuľkách. Pri replikácii 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 nieje nutné dopočítavať dáta, ktoré by boli na základe daného dotazu zmenené. Master server sa snaží ukladať do binlogu iba kompletne a vykonané transakcie, no bohužiaľ prax ukázala, že existujú aj výnimky. To znamená, že výskyt syntakticky nevalidných dotazov je možný, ale zároveň veľmi zriedkavý. V MySQL konektore teda nie je nutné sledovať korektnosť parsovaných dotazov.

Pomocou SQL dotazu *SHOW BINARY LOGS* je možné vylistovať 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 začiatková 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 úrovni riadkov zakódované ako base-64<sup>1</sup> 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 prepínačov získame možnosť pohodlne sledovať obsah udalosti týkajúcich sa zmien v dátach. [23]

Pre Debeziom sú dôležité udalosti typu:

---

<sup>1</sup>Typ kódovania, ktorý prevádza binárne dáta na postupnosť znakov

- **Query**, v ktorom sa objavujú dotazy na zmenu štruktúry databáze (DDL) ako je možné vidieť na poslednom riadku príkladu události 3.1.
- **Table\_map**, pomocou ktorého binlog mapuje konkrétne tabuľky na identifikátor, ktorým sa následne tieto tabuľky referencuje. Príklad tejto události 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 události 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 události 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
7 ### @1=1001 /* INT meta=0 nullable=0 is_null=0 */
8 ### @2='Sally' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
9 ### @3='Thomas' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
10 ### @4='sally.thomas@acme.com' /* VARSTRING(255) meta=255 nullable=0 is_null
   ↪ =0 */

```

```

11 ### @5=NULL /* VARSTRING(15) meta=15 nullable=1 is_null=1 */
12 ### SET
13 ### @1=1001 /* INT meta=0 nullable=0 is_null=0 */
14 ### @2='John' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
15 ### @3='Thomas' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
16 ### @4='sally.thomas@acme.com' /* VARSTRING(255) meta=255 nullable=0 is_null
    ↪ =0 */
17 ### @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. Vo väčšine prípadoch už MySQL binlog neobsahuje kompletnú históriu databáze a preto je tento mód v základnom nastavení.

Pri každom vytváraní aktuálneho obrazu, konektor postupuje podľa týchto krokov<sup>[4]</sup>:

1. Aktivuje globálny zámok čítania (read lock) aby zabránil ostatným databázovým klientom v zapisovaní.
2. Spustí transakciu s izoláciou na opakované čítanie (repeatable read)<sup>2</sup>, aby všetky nasledujúce čítania v rámci tejto transakcie boli voči jednému konzistentnému obrazu.
3. Prečíta aktuálnu pozíciu binlogu.
4. Prečíta schéma databází a tabuliek na základe konfigurácie konektoru.
5. Uvoľní globálny zámok, aby ostatní databázový klienti mohli znovu zapisovať do databáze.
6. Voliteľne zapíše zmeny DDL do Kafka topiku vrátane všetkých potrebných SQL dotazov.
7. Oskenuje 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.

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 schém 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

<sup>2</sup>Stupeň izolácie založený na používaní *read* a *write* zámkoch, ktorý ale nezabráni prítomnosti fantómov vznikajúcich v situácii, 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.



vykonanie krokov tri a štyri. V piatom kroku je tento zámok uvoľnený predtým, než konektor vykoná väčšinu práce pri kopírovaní údajov.

**Note:** moze byt popisane este viac ak by bolo potrebne nahrabat strany :)

## 3.2 DDL parser

Pri čítaní binárneho logu MySQL konektor parsuje DDL dotazy na základe ktorých si v pamäti vytvára modely schém 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 udalosť vznikla. Aktuálne schéma sa nemôže použiť, nakoľko sa môže zmeniť v danom čase pokiaľ na danej pozícii v logu na ktorej konektor číta.

Konektor produkuje správy použitím Kafka Connect Schemas, ktoré definujú jednoduchú dátovú štruktúru obsahujúcu názvy a typy polí a spôsob organizácie týchto polí. Pri generovaní správy na udalosť 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 odchyť DDL udalosť, stačí mu aktualizovať model, ktorý si drží v pamäti a ten následne použiť na generovanie **Schema** objektu. V rovnakom čase sa vytvorí komponenta, 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 udalosti až do doby pokiaľ sa neodchyť 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 konventoru 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[27]:

- 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, ktorá bude odzrkadľovať pravidlá gramatiky. Takúto implementáciu je jednoduchšie udržiavať v priebehu času.
- Umožniť vytvorenie parserov pre PostgreSQL, Oracle, SQLServer a všetkých ostatných DBMS, ktoré budú potrebné.
- Umožniť prispôbenie 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 interpunkčné znamienka, citované reťazce, slová a symboly, 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 sprostredkúva celú parsovaciu logiku špecifickú pre MySQL. Napríklad DDL dotaz 3.3 je možné sparsovať 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 implementácia iného parseru.

Pri každom volaní metódy `parseNextStatement` je predavaný 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, ak by parser chcel parsovať dotaz začínajúci na `CREATE TABLE ....` Prvým parsované slovo je `CREATE`, čím by sa podľa ukážky z kódu C.2 zavolá metóda `parseCreate`. V nej sa toto slovo skonsumuje a rovnakým spôsobom nastáva kontrola druhého slova, kde sa po vyhodnotení hodnoty `TABLE` zavolá metóda `parseCreateTable` (ukážka C.3). Táto metóda odzrkadľuje následovné 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 skonsumovať 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.

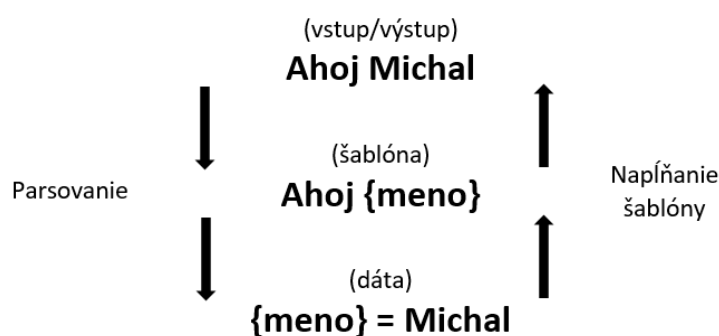


## Kapitola 4

# Syntaktická analýza

Syntaktickou analýzou (slangovo z angličtiny tiež **parsovaním**) sa v teórii rozumie konštrukcia derivačného stromu vety bezkontextového jazyka[21] 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 variablnými 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 napĺňovania šablóny

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

### 4.1 Teória parsovania

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

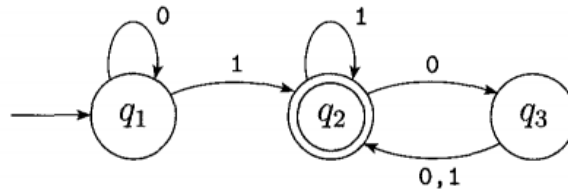
### 4.1.1 Deterministický konečný automat

Konečné automaty sa používajú v rôznych oboroch ako napríklad pri prekladačoch, spracovávaní prirodzeného jazyka, pri návrhu hardwaru a ďalších [5]. Predstavujú model systémov, ktoré rozpoznávajú, 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 päťica  $M = (Q, \Sigma, \delta, q_0, F)$ , kde

- $Q$  je konečná množina stavov
- $\Sigma$  je konečná množina vstupných symbolov
- $\delta$  je prechodová funkcia  $\delta : Q \times \Sigma \rightarrow Q$
- $q_0$  je počiatočný stav
- $F \subseteq Q$  je množina koncových stavov [5]

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 šipkou, 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

$$L(M) = \{\omega \mid \delta^*(q_0, \omega) \in F\}. [5]$$

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 uzavreté 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 dána abeceda  $\Sigma$ . Množina všetkých regulárnych výrazov nad  $\Sigma$  je definovaná indukzívne:

- $\emptyset$  je regulárny výraz,
- $\epsilon$  je regulárny výraz,
- $a$  je regulárny výraz pre každé písmeno  $a \in \Sigma$ ,
- pokiaľ sú  $r_1$  a  $r_2$  regulárne výrazy, tak  $r_1 + r_2$ ,  $r_1 r_2$  a  $r_1^*$  sú regulárne výrazy. [5]

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*<sup>1</sup>
- $\Sigma$  je konečná neprázdna množina tzv. *terminálov*<sup>2</sup>, kde platí  $N \cap \Sigma = \emptyset$
- $S \in N$  je *štartovací symbol*
- $P$  je konečná množina pravidiel typu  $\alpha \rightarrow \beta$ , kde  $\alpha$  a  $\beta$  sú slová nad  $N \cup \Sigma$  taká, že  $\alpha$  obsahuje aspoň jeden neterminál.
- každé pravidlo  $P$  je v tvare  $A \rightarrow \gamma$ , kde  $\gamma \in (n \cup \Sigma)^*$  a  $A$  je neterminál [6]

CFG 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 [29].

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  $G_1$ .

<sup>1</sup>Premenné symmboli, ktoré sa reprezentujú pomocou veľkých písmen

<sup>2</sup>Písmená vstupnej abecedy, často reprezentované malými písmenami, číslami alebo špeciálnymi symbolmi

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

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 (Derivace).** Je daná gramatika  $\mathcal{G} = (N, \Sigma, S, P)$ . Povedzme, že  $\delta$  sa *odvodí* z  $\gamma$  vtedy, ak

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

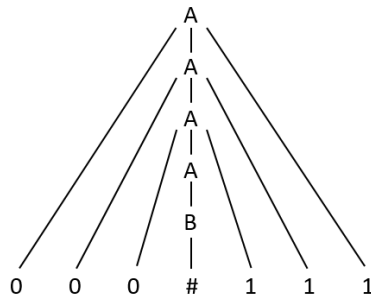
$$\gamma = \gamma_1 \Rightarrow_{\mathcal{G}} \gamma_2 \Rightarrow_{\mathcal{G}} \dots \Rightarrow_{\mathcal{G}} \gamma_k = \delta$$

Tento fakt sa označuje  $\gamma \Rightarrow_{\mathcal{G}}^* \delta$  a tejto konečnej postupnosti hovoríme *derivácia*. [6]

Pre príklad, gramatika  $G_1$  generuje reťazec  $000\#111$ . Derivácia tohto reťazca bude vyzeráť nasledovne

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

Rovnakú informáciu je možné reprezentovať graficky pomocou sparovaného (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 povedať, že jazyk gramatiky  $L(G_1) \in \{0^n\#1^n | n \geq 0\}$ . 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.

$$L(\mathcal{G}) = \{\omega \in \Sigma^* | S \Rightarrow_{\mathcal{G}}^* \omega\}. [6]$$



#### 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.[18]

Každé pravidlo v BNF má nasledujúcu štruktúru:

$$\langle \text{neterminál} \rangle ::= \text{výraz}$$

Všetky neterminály v BNF sa zapisujú do špicatých zátvoriek  $\langle \rangle$ , či už sú použité na pravej alebo ľavej strane pravidla. Výraz sa môže obsahovať terminály aj neterminály a je definovaný ich spojením, alebo výberom. Symboli 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[14]. Terminály môžu byť vyjadrené konkrétnym postupom znakov v úvodzovkách, alebo pomocou triedy literálov, ktorú je možné zapísať pomocou regulárneho výrazu. Priradzovací znak pravidla je zmenený z  $::=$  na jednoduché  $=$  a vylučujú 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:

$$\begin{aligned} \text{term} &= ["-"] \text{ factor} \\ \text{term} &= "-"? \text{ factor} \end{aligned}$$

**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:

$$\text{expr} = \text{term} ("+" | "-") \text{expr}$$

**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 zasebou. Toto je pravidlo je taktiež možné zapísať pomocou znaku  $*$ . Príklad:

$$\begin{aligned} \text{args} &= \text{arg} \{", " \text{arg}\} \\ \text{args} &= \text{arg} ("," \text{arg})^* \end{aligned}$$

**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 2 rôzne zápisy v EBNF gramatiky 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 syntaxie, 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 ( $\leftarrow$ ) pre priradenie, namiesto bežnejšieho symbolu rovníc ( $=$ ).

Problém nejednoznačnosti spočíva v možnosti sparsovať 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 sparsovaná 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 [10]. 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* *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 horoví, 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 regulérne jazyky. Pokiaľ sa v jazyku, ktorý sa snažíme parsovať, objavia vnorené alebo rekurzívne elementy, nejedná sa už o regulérny jazyk, ale o jazyk *bezkontextový* (4.1.3). Parsovanie takéhoto jazyka pomocou regulárneho výrazu by spôsobilo degenerovanú slučku [10].

## 4.3 Štruktúra bezkontextových parserov

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

operátory, oddeľovače a pod. Pre parser sú tokeny ďalej nedeliteľné stavebné jednotky, ktoré používa pri interpretácii vstupných dát. Program vykonávajúci túto úlohu sa nazýva štruktúrally 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 hraje v procese parsovania iba úlohu pomocníka na uľahčenie práce parseru.

Parsre sú významnou súčasťou kompilátorov alebo interpretorov 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.

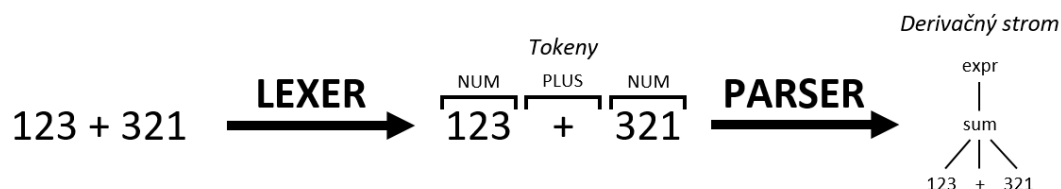
### 4.3.1 Lexer

Lexery zohrávajú dôležitú rolu pri parsovaní, pretože transformujú počiatočný vstup na jednoduchšie spracovateľnú formu pre parser. Napísanie gramatiky pre lexer je zvyčajne jednodušie nakoľko nie je nutné riešiť vymoženosti bezkontextového 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). [30]

Lexer prečíta vstupný preťažec a rozdelí ho na predom definované typy tokenov. Na definíciu týchto typov sa používajú regulérne 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 s 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 vyzeráť nasledovne:

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

Vzhľadom na to, že lexery sú takmer výlučne používané v spojení s parsermi, je nutné si určiť hranicu, kde končí práca lexeru a 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.

$$\text{IPv4} = [0-9]^+ \text{"."} [0-9]^+ \text{"."} [0-9]^+ \text{"."} [0-9]^+$$

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 dve druhy tokenov (číslo a bodka).

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

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

## 4.4 Typické problémy parsovania

Prí 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 zdefinuje 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 [30].

#### 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 referenciou 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:

$$\begin{aligned}A &\rightarrow B\alpha \\ B &\rightarrow A\beta\end{aligned}$$

Predpokladajme, že sa snažíme sparsovať 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 podúlohou by bolo parsovať pravidlo  $A$  na tom istom mieste [19]. 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 [19]. **Note: možnosť pridať príklad odstraňovania ľavej rekurzie**



## Kapitola 5

# Parsovacie algoritmy

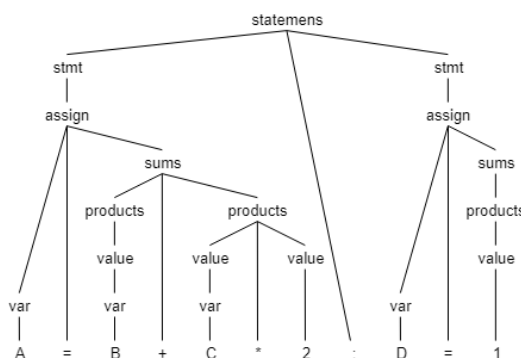
Teoreticky je parsovanie vyriešný problém, ale je to druh problému, ktorý sa stále a znovu rieši. To znamená, že existuje veľa rôznych algoritmov, každý so silnými a slabými bodmi a akademici ich stále zlepšujú.[30]

### 5.1 Obecný prehľad

Metód parsovania existuje veľmi veľa. Pre praktické použitie však majú význam metódy, ktoré je možné rozdeliť do dvoch skupín:[21]

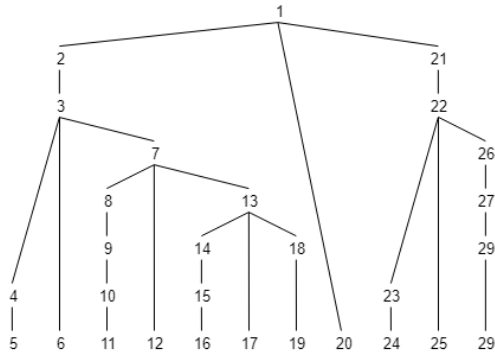
- Parsovanie **zhora dole (top down)** u ktorého sa parser najprv snaží identifikovať koreň parsovaného stromu, a potom postupuje dole cez podstromy až kým nenájde na listy stromu. Top down metóda je najrozšírenejšia zo spomenutých dvoch a existuje na ňu niekoľko úspešných algoritmov, ktoré ju uplatňujú. Tieto algoritmy najčastejšie využívajú funkčnosť 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.

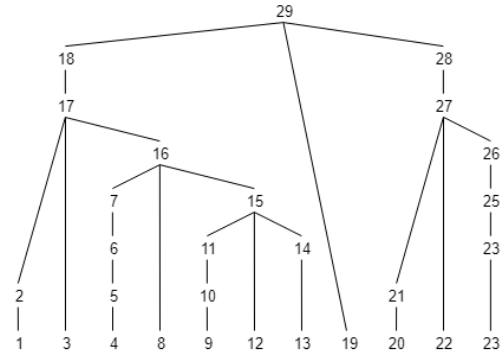


Obr. 5.1: Typický parsovací strom pre výraz  $A = B + C * 2; D = 1$

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 reps. implementáciu, no bottom up parsre boli silnejšie. Aktuálne je situácia vyvázenejšia a to hlavne vďaka pokroku v stratégiách parsovania top down metódou.

Nasledujúca tabuľka obsahuje súhrn hlavných vlastností existujúcich algoritmov a ich použitie[30].

Algoritmus	Hlavné vlastnosti	Použitie
<b>CYK</b> [16]	V najhoršom prípade zložitosť $O(n^3)$ Gramatika vyžaduje zápis CNF forme	Špecifické problémy
<b>Earley</b> [8]	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
<b>LL</b> [25]	Jednoduchý na implementáciu Nie až tak schopný ako väčšina algoritmov (nepodporuje ľavú rekúziu) Historicky je najpopulárnejší	Ručne vytvárané parsre a generátory parsrov, ktoré sú jednoduchšie na poskladanie
<b>LR</b> [17]	Náročný na implementáciu Dokážu spracovať väčšinu gramatík, niektoré varianty dokonca všetky Zvyčajne lineárna zložitosť, u dokonaľších variánt je zložitosť v najhoršom prípade $O(n^3)$	Najvýkonnejšie generátory parserov



<b>Packrat (PEG)</b> [9]	Lineárna zložitosť Používa špeciálny formát zápisu gramatiky Designovaný na parsovanie počítačových jazykov	Jednoduché a zároveň silné parsre alebo generátory parserov pre počítačové jazyky
--------------------------	---	---

Tabuľka 5.1: Prehľad vlastností parsovacích algoritmov

Väčšina parsovacích nástrojov pre bezkontextové gramatiky, ktoré podporujú Javu<sup>1</sup>, 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. [21] 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, začínať 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 sparovaný(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 nasledujúcich 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 [13].

Táto funkcionálna 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 jednoduhšie 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.[30]

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

<sup>1</sup><[https://en.wikipedia.org/wiki/Comparison\\_of\\_parser\\_generators](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 [28].

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 viesť celý vstupný reťazec. Takéto algoritmy prijímajú sekvenciu znakov a vyhodnocujú ju priebežne [15]. 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 štartovacím symbolem a následne sa opakovane rozširuje do ľavého neterminálu, kým nekonštruuje 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 je 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 gramatikám. To znamená, že LL parser dokáže analyzovať LL gramatiku [12].

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

- **LL(1)** algoritmus s lineárnou zložitou, ktorý používa jeden lookahead token
- **LL(k)** algoritmus taktiež s lineárnou zložitou, ktorý používa  $k$  lookahead tokenov.
- **LL(\*)** algoritmus, ktorý môže použiť neobmedzené množstvo lookahead tokenov. Teoretická asymptotická zložitosť tohto algoritmu je  $O(n^2)$ , no v praxi väčšinou kontrolujú jeden až dva lookahead tokeny [25].

### 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 ekvivaletnej podoby bez ľavej rekurzie aby mohli byť parsované pomocou LL parserov.

Tento problém má za následok miernu stratu produktivity a výkonu [30]. Výkonu z toho dôvodu, že pokiaľ gramatike z 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.

### 5.2.2 Príklad práce LL parseru

NOTE: ak je nutné, príadne ak bude treba nahrabat stray môžem pridať

## 5.3 LR parser

Algoritmy postavené na LR parsovaní sú hlavným dôvodom úspechu parsovania pomocou *Bottom Up* metódy. Aj napriek tomu, že sú schopnejšie ako tradičné LL(1) gramatiky, ich popularita je bohužiaľ nízka a to najmä z dôvodu, že ich vytvorenie bolo v minulosti veľmi náročné [30]. 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 nájde pravidlo gramatiky vytvorené z aktuálnych parsovacích stromov. Aplikovaním pravidla sa dané stromy spoja do jedného stormu z novým koreňom.

V jedoduchosti 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 gramaticku existuje pevný (konečný) počet takýchto stavov [17].

LL a LR parsre vznikali 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 používajú. Tie sa ručne veľmi ťažko 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, výsledný analyzátor sa nazýva Simple LR (**SLR**), Lookahead LR (**LALR**) alebo kanonický LR parser. Použitím týchto alternatívnych LR parsrov sa stráca výkonnosť parsru oproti originálnemu LR parsru. Poradie podľa výkonnosti je nasledovné:

$$\text{LR}(1) > \text{LALR}(1) > \text{SLR}(1) > \text{LR}(0) \text{ [11]}$$

### 5.3.2 Príklad práce LR parseru

NOTE: ak je nutné, príadne ak bude treba nahrabat stray môžem pridať

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 že 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 lookahead sety. Tieto úpravy prinášajú určité obmedzenia na gramatiky, ktoré sú schopné sparsovať [30].

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 [17] 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* [11] z roku 2007, ktorej bibliografia obsahuje viac ako 1700 citovaných článkov! [12]

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 backtrackingu a generalizované parsovanie.

Dokonca aj tieto existujúce nástroje, niesú na 100% dokonalé a je nutné aby sa stále vyvíjali aj v rámci použitia, hlásenia chýb, lepšej vizualizácie, lepšej integrácie z programovacími jazykmi atd. Napríklad ANTLR v4 úplne prepracoval svoj analytický algoritmus na zlepšenie jednoduchosti použitia oproti predchádzajúcej verzii. Algoritmus predstavil pod názvom **ALL(\*)** [26] v roku 2014. Bison experimentuje s algoritmom **IELR** [7], 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* [26].

Bison generuje parsre, ktorých práca je založená na tabuľkách [7]. 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. Narozdiel 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 *ANTLR-works*<sup>2</sup>. 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 sparsovať, 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 vyvinutému algoritmu ALL(\*), ktorého teoretická zložitosť je  $O(n^4)$ , no na gramatikách používaných v praxi pracuje **lineárne** [26].

<sup>2</sup><<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 doporučujem knihu *The Definitive ANTLR4 reference* [24] 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<sup>1</sup>, alebo jeho vybuildením zo zdrojových kódov dostupných v GitHub repositáru<sup>2</sup>.

Kroky nutné na nainštalovanie ANTLR nástroja:

---

<sup>1</sup><<http://www.antlr.org/download.html>>

<sup>2</sup><<https://github.com/antlr/antlr4>>

1. Nakopírovať stiahnutý nástroj na miesto, kde sú uchovávané java knižnice tretích strán.
2. Pridať cestu k nástroju do systemových premenných.
3. Na zjednodušenie používania vytvoriť spúšťačí script 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 gramatky vyzerá nasledovne.

```
antlr4 <options> <grammar-file-g4>
```

Prvým 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 sparsovaného stromu ANTLR v základe generuje objekt poslucháča, ktorý reaguje na udalosti vznikajúce pri príchode aj pri odchode z uzlu stromu. Ak užívateľ preferuje použitie visitor paternu pri prechádzaní stromom, je možné ANTLR nástroju nastaviť generovanie visitor objektov.

#### 6.1.1.1 Maven

Pre projekty, ktoré využívajú Maven<sup>3</sup> na riadenie a správu buildov aplikácií, je možné túto správu využiť aj pre nastavenie ANTLR.

Prvým krokom je si v *pom.xml* nadefinovať závisosť 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 configuračného parametru *sourceDirectory*.

---

<sup>3</sup><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 zápis gramatiky EBNF notáciu, ktorá je zároveň používaná v dokumentáciách programovacích jazykov na popis ich syntaxie. 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 prepoužívateľné 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 desatiner čiarke, 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 (*WHITESPACE*). 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<sup>4</sup>. 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<sup>5</sup>.

### 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.

---

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

<sup>5</sup>MIT licencia je jedna z najmenej reštriktívnych open source licencií. Ktokoľvek môže takto licencovaný program bez obmedzenia používať a šíriť, pokiaľ z programu neodstráni kópiu 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 události vzniknuté pri príchode a odchode z pravidiel gramatiky. Implementujeme si vlastný `HelloListenerCustom`, pomocou ktorého budeme reagovať na nami zvolené události. V tomto prípade to je výpis na systemový 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 sparovaný strom. Posledným krokom je prechod stromu pomocou `ParseTreeWalker` triedy, ktorej priradíme nami vytvorený `HelloListenerCustom`. Naš `HelloListenerCustom` je možné priradiť aj parseru, vďaka čomu by sme mohli odchytať události už počas toho, ako sa vygenerovaný parser snaží vytvoriť sparovaný strom. V takomto prípade je ale nutné počítať zo situáciami, že aktuálne sparovaný kontext nie je úplny a taktiež nie je zaručené, že pravidlo bude obsiahnuté vo výslednom sparovanom 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

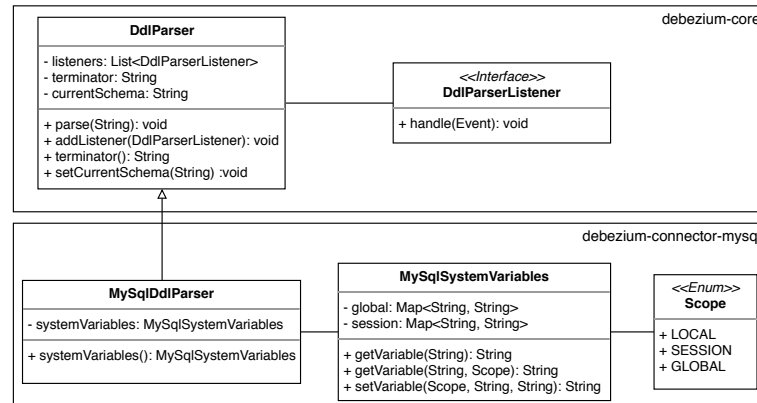
Štruktúra projektu Debezium je členená do viacerých modulov. Každý konektor pre jednotlivé DBMS je implementovaný v samostatnom module. V kontexte DDL parsovania MySQL sú významné dva moduly:

- **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 Debezium podporovaným databázovým systémom, v rámci ktorého sa parsuju DDL. Pravdepodobne z tohto dôvodu nebola implementácia dostatočne generalizovaná a mnohé funkcionality, ktoré do budúcnosti 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 schém 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 pre MySQL DDL parser.



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 `MySqlSchema`.

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 udržiava systémové premenné na úrovni priestoru definovaného pomocou enum `Scope`. Autor túto funkcionality 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.

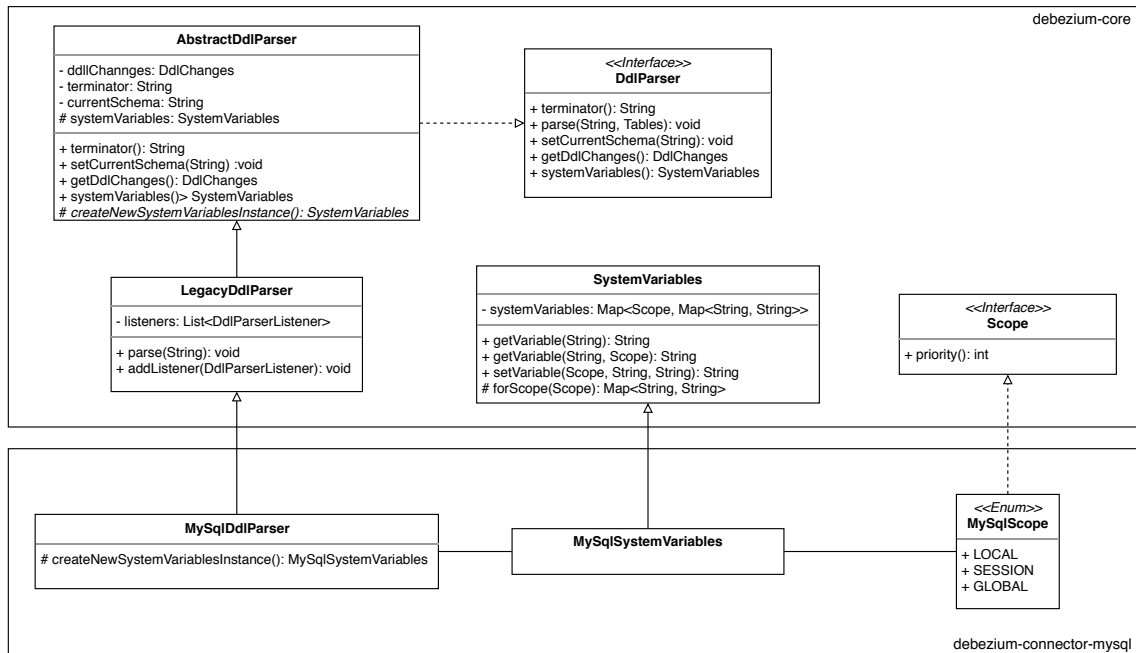
Prvou zmenou v návrhu je vytvorenie rozhrania pre DDL parsre, ktoré bude definovať možnosť komunikácie pre komponenty, ktoré potrebujú parsovať DDL dotazy. Toto rozhranie bude definovať všetky verejné metódy z tried `DdlParser` a `MySqlDdlParser`. Tým sa dostávame k prvému problému, ktorým sú systémové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

Aktuálna implemenácia triedy `MySqlSystemVariables` obsahuje dve `Map` objekty, ktoré udržiavali hodnoty systémových premenných tak, že na základe hodnoty enum `Scope` bola zvolená správna mapa z ktorej sa hodnota premennej čítala. Takýto prístup v generovanej 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?

Extendovanie 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`. [2]

Nakoľko chceme prepoužiť resp. zanechať čo možno najväčšiu časť stávajúcej aplikácie, bolo by vhodné vyseparovať z triedy `DdlParser` všetko, čo by mohola 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 iba v tom, že si túto udalosť pridá do listu udalostí, s ktorým sa následne pracuje až po ukončení práce parseru. Nakoľko neexistujú žiadny iný poslucháči, vedenie projektu sa rozhodlo túto funkcionality zanechať v rámci `LegacyDdlParser` implementácie, ale nechcelo ju mať v novej implementácii. `DdlChanges` trieda sa teda stane súčasťou abstraktnej implementácie parseru 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.

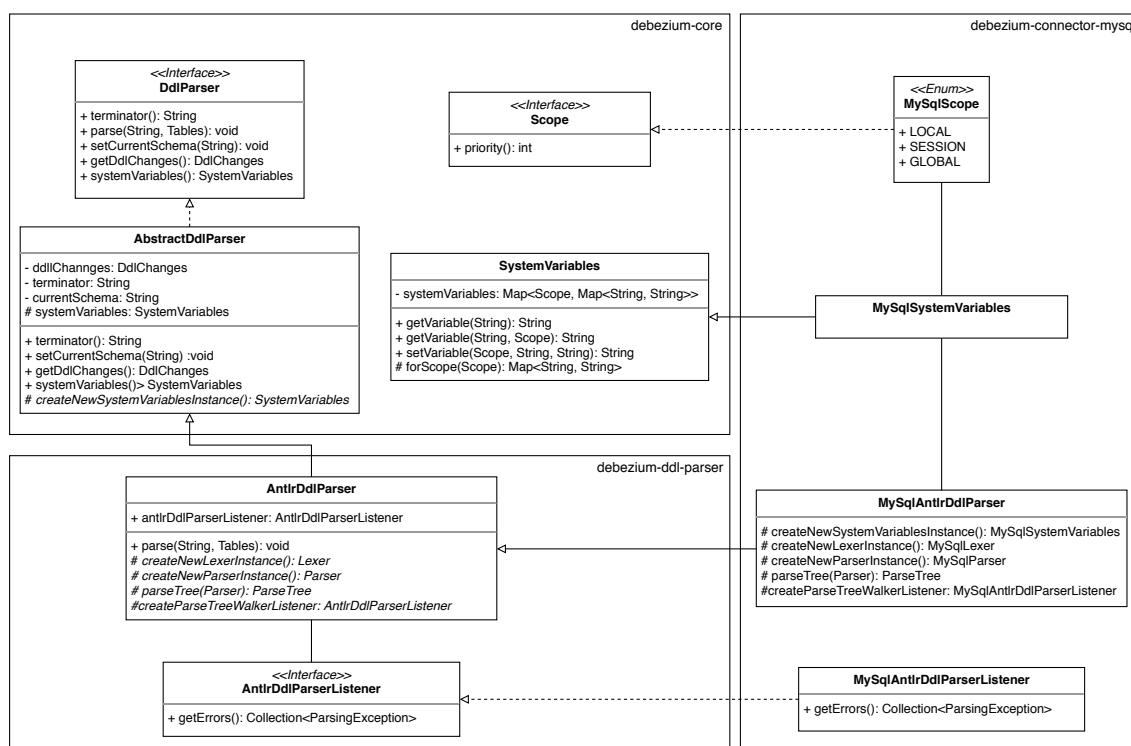
Pri návrhu je nutné počítať aj z generovaním gramatík, čo pre modul, ktorý bude definície gramatík obsahovať spôsobí, že jeho build bude trvať dlhšie. Práve z dôvodu aby sa vyvojárom ušetril čas buildu pri implementovaní špecifických parserov, bude základná implementácia spolu z definíciou gramatík a vygenerovanými triedami separovaná 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 buildiť 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 regovať na sparovaný strom a vykonávať náležité akcie nad databázovým modelom, ktorý si udržiava 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 daatabázovým modelom môže nastať

viacero chýb. Tieto chyby bude nutné lokálne ukladať a spracovať ich až po prejdenní celého stromu. Kvôli tomu bude od konkrétnych poslúchačov 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 instanciu triedy musí sprostredkovať implementácia konkrétneho parseru. Samotný postup inicializácie sa ale dá generovať a na získanie nových instancií sa môže použiť abstraktná metóda, ktorej implementácia bude súčasťou implementácia konkrétneho parseru. Ronaký 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 ANTLR DDL parseru

## 6.3 Implementácia



# Literatúra

- [1] Attunity Ltd. Attunity Connect Product White Paper. *Change Data Capture – next generation ETL*. Máj 2004.
- [2] BLOCH, J. *Effective java*. Addison-Wesley Professional, 2017. ISBN 978-0321356680.
- [3] Debezium Community. *Debezium* [online]. 2018. [cit. 28.1.2018]. Dostupné z: <<http://debezium.io/>>.
- [4] Debezium Community. *Debezium Connector for MySQL: Snapshots* [online]. 2018. [cit. 22.2.2018]. Dostupné z: <<http://debezium.io/docs/connectors/mysql/#snapshots>>.
- [5] DEMLOVÁ, M. Jazyky, automaty a gramatiky. *Konečné automaty*. 2017.
- [6] DEMLOVÁ, M. Jazyky, automaty a gramatiky. *Gramatiky*. 2017.
- [7] 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.
- [8] EARLEY, J. An efficient context-free parsing algorithm. *Communications of the ACM*. 1970.
- [9] FORD, B. *Packrat parsing: a Practical Linear-Time Algorithm with Backtracking*. PhD thesis, Massachusetts Institute of Technology, 2002.
- [10] FORD, B. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*. ACM, 2004.
- [11] GRUNE, D. – JACOBS, C. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer New York, 2007. ISBN 9780387689548.
- [12] HABERMAN, J. *LL and LR in Context: Why Parsing Tools Are Hard* [online]. [cit. 25.4.2018]. Dostupné z: <<http://blog.reverberate.org/2013/09/11-and-lr-in-context-why-parsing-tools.html>>.
- [13] HABERMAN, J. *LL and LR Parsing Demystified* [online]. [cit. 21.4.2018]. Dostupné z: <<http://blog.reverberate.org/2013/07/11-and-lr-parsing-demystified.html>>.

- [14] ISO 14977:1996(E). Information technology – Syntactic metalanguage – Extended BNF. Standard, International Organization for Standardization, Geneva, CH, 1996.
- [15] KARP, R. M. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *IFIP Congress (1)*, 1992.
- [16] KASAMI, T. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*. 1966.
- [17] KNUTH, D. E. On the translation of languages from left to right. *Information and control*. 1965.
- [18] MIGHT, M. *The language of languages* [online]. [cit. 3.4.2018]. Dostupné z: <<http://matt.might.net/articles/grammars-bnf-ebnf>>.
- [19] 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.
- [20] MORLING, G. Streaming Database Changes with Debezium, 2017. <https://www.youtube.com/watch?v=I0Z2Um6e430>, publikované 9.11.2017.
- [21] MÜLLER, K. *Programovací jazyky*. Česká technika - nakladatelství ČVUT, 2005. ISBN 80-01-02458-X.
- [22] 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.
- [23] *MySQL 5.7 Reference Manual*. Oracle Corporation and/or its affiliates, 2018. <https://dev.mysql.com/doc/refman/5.7/en/>.
- [24] PARR, T. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
- [25] PARR, T. – FISHER, K. LL (\*): the foundation of the ANTLR parser generator. *ACM Sigplan Notices*. 2011.
- [26] PARR, T. – HARWELL, S. – FISHER, K. Adaptive LL (\*) parsing: the power of dynamic analysis. In *ACM SIGPLAN Notices*. ACM, 2014.
- [27] Randall Hauch. *Parsing DDL* [online]. 2018. [cit. 25.2.2018]. Dostupné z: <<http://debezium.io/blog/2016/04/15/parsing-ddl/>>.
- [28] ROCHE, E. – SCHABES, Y. *Finite-state language processing*. MIT press, 1997. ISBN 978-0262181822.
- [29] SIPSER, M. *Introduction to the Theory of Computation*. Boston: Thomson Course Technology, 2 edition, 2006. ISBN 0-534-95097-3.
- [30] TOMASSETTI, G. *A Guide to Parsing: Algorithms and Terminology* [online]. 9 2017. [cit. 28.3.2018]. Dostupné z: <<https://tomasetti.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
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

```
1 {
2     "schema" : {
3         ...
4     },
5     "payload" : {
6         "before" : null,
7         "after" : {
8             "id" : 352,
9             "name" : "Janko",
10            "surename" : "Hrasko",
11            "email" : "janko@hrasko.sk"
12        },
13        "source" : {
14            "name" : "dbserver1",
15            "server_id" : 0,
16            "ts_sec" : 0,
17            "file" : "mysql-bin.000001",
18            "pos" : 12,
19            "row" : 0,
20            "snapshot" : true,
21            "db" : "todo_list",
22            "table" : "users"
23        },
24        "op" : "c",
25        "ts_ms" : 1517152654614
26    }
27 }
```



## Dodatok C

# Ukážky zdrojových kódov

Ukážka C.1: Parsovacie metódy DDL parserov

```
1 public final void parse(String ddlContent, Tables databaseTables) {
2
3     ⇨ Tokenizer tokenizer = new DdlTokenizer(!skipComments(), this::determineTokenType);
4     TokenStream stream = new TokenStream(ddlContent, tokenizer, false);
5     stream.start();
6     parse(stream, databaseTables);
7 }
8 public final void parse(TokenStream ddlContent, Tables databaseTables) throws ParsingExc
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
23        ⇨ (" + t.getMessage() + ") parsing", t);
24    }
```

Ukážka C.2: Implementácia parseNextStatement metódy v MySqlDdlParser

```
1 @Override
2     protected void parseNextStatement(Marker marker) {
3         if (tokens.matches(DdlTokenizer.COMMENT)) {
4             parseComment(marker);
5         } else if (tokens.matches("CREATE")) {
6             parseCreate(marker);
7         } else if (tokens.matches("ALTER")) {
8             parseAlter(marker);
9         } else if (tokens.matches("DROP")) {
10            parseDrop(marker);
11        } else if (tokens.matches("RENAME")) {
12            parseRename(marker);
13        } else {
14            parseUnknownStatement(marker);
15        }
16    }
```



---

Ukážka C.3: Implementácia parseCreateTable metódy v MySQLDdlParser

```
1  protected void parseCreateTable(Marker start) {
2      tokens.canConsume("TEMPORARY");
3      tokens.consume("TABLE");
4      boolean onlyIfNotExists = tokens.canConsume("IF", "NOT", "EXISTS");
5      TableId tableId = parseQualifiedTableName(start);
6      if (tokens.canConsume("LIKE")) {
7          TableId originalId = parseQualifiedTableName(start);
8          Table original = databaseTables.forTable(originalId);
9          if (original != null) {
10
11              ↪ databaseTables.overwriteTable(tableId, original.columns(), original.primaryKeyColumnNames(), original.c
12              }
13              consumeRemainingStatement(start);
14              signalCreateTable(tableId, start);
15              debugParsed(start);
16              return;
17          }
18          if (onlyIfNotExists && databaseTables.forTable(tableId) != null) {
19              // The table does exist, so we should do nothing ...
20              consumeRemainingStatement(start);
21              signalCreateTable(tableId, start);
22              debugParsed(start);
23              return;
24          }
25          TableEditor table = databaseTables.editOrCreateTable(tableId);
26          // create_definition ...
27          if (tokens.matches('(')) parseCreateDefinitionList(start, table);
28          // table_options ...
29          parseTableOptions(start, table);
30          // partition_options ...
31          if (tokens.matches("PARTITION")) {
32              parsePartitionOptions(start, table);
33          }
34          // select_statement
35
36          ↪ if (tokens.canConsume("AS") || tokens.canConsume("IGNORE", "AS") || tokens.canConsume("REPLAC
37              parseAsSelectStatement(start, table);
38          }
39          // Make sure that the table's character set has been set ...
40          if (!table.hasDefaultCharset()) {
41              table.setDefaultCharsetName(currentDatabaseCharset());
42          }
43          // Update the table definition ...
44          databaseTables.overwriteTable(table.create());
45          signalCreateTable(tableId, start);
46          debugParsed(start);
47      }
```



## 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 příloženého CD

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



Obr. E.1: Seznam příloženého CD — příklad

můžete snadno vyrobit příkazem:

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

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případně **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.