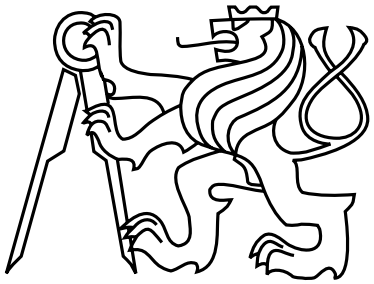
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



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

30. marca 2018

Obsah

1 Debezium **1**
- 1.1 Zachytávanie zmenených dát . . . 1
  - 1.1.1 Replikácia dát . . . 2
  - 1.1.2 Microservice Architecture . . . 2
  - 1.1.3 Ostatné . . . 3
- 1.2 Odchytávanie zmien v databázi . . . 3
  - 1.2.1 Kafka . . . 3
  - 1.2.2 Infraštruktúra správ pomocou Apache Kafka . . . 3
  - 1.2.3 Kafka Connect . . . 4
  - 1.2.4 Štruktúra správy . . . 5

2 Aktuálne riešenie MySQL konektoru **7**
- 2.1 MySQL konektor . . . 7
  - 2.1.1 Binárny log . . . 7
  - 2.1.2 Aktuálny obraz tabuliek . . . 10
- 2.2 DDL parser . . . 11
  - 2.2.1 Framework na parsovanie DDL . . . 11
  - 2.2.2 Implementácia MySQL DDL parsru . . . 13

3 Syntaktická analýza **15**
- 3.1 Parsovanie pomocou regulárnych výrazov . . . 15
  - 3.1.1 Deterministický konečný automat . . . 16
  - 3.1.2 Regulárny výraz . . . 17
  - 3.1.3 Bezkontextový jazyk . . . 17
  - 3.1.4 Backus-Naur Form notácia . . . 19
  - 3.1.5 Rozšírená Backus-Naur Form notácia . . . 19
- 3.2 Štruktúra bezkontextových parserov . . . 20
  - 3.2.1 Lexer . . . 20

4 Parsovacie algoritmy **23**

A Zoznam použitých skratiek **27**

B Ukážka dát **29**

C Ukážky zdrojových kódov **31**

D Obsah přiloženého CD **35**

Zoznam obrázkov

1.1 Koncept distribúcie zmenených dát 2
1.2 CDC topológia z Kafka Connect 5

3.1 Príklad parsovania a naplňovania šablóny 15
3.2 Príklad DFA znázorneného pomocou stavového diagramu 16
3.3 Derivačný strom gramatiky $\mathcal{G}_1$ pre reťazec *000#111* 18
3.4 Spracovanie reťazca *123 + 321* lexerom a parserom 21

D.1 Seznam přiloženého CD — příklad 35

Zoznam tabuliek

Zoznam ukážok

2.1 Query událost z binárneho logu MySQL 9
2.2 Table_map a Update_rows události z binárneho logu MySQL 9
2.3 DDL dotaz v MySQL 12
2.4 Parsovanie dotazu pomocou MySqlDdlParseru 12
B.1 Ukážka CDC správy odoslanej Debeziom 29
C.1 Parsovacie metódy DDL parserov 31
C.2 Implementácia parseNextStatement metódy v MySqlDdlParser 32
C.3 Implementácia parseCreateTable metódy v MySqlDdlParser 33

Kapitola 1

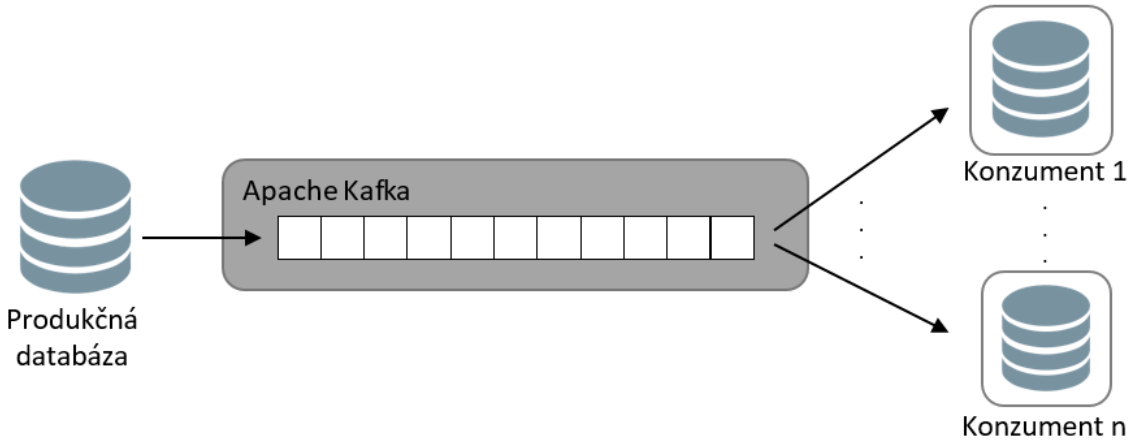
Debezium

Debezium[1] je projekt, ktorý slúži k zaznámenavaniu zmien v databázi analýzou udalostí v transakčnom logu. Jednou z podporovaných databázi je taktiež MySQL. Na správnu funkcionalitu Debezium potrebuje metadáta popisujúce štruktúru databáze v závislosti na čase. Pre MySQL je to možné dosiahnuť tak, že príkazy, ktoré vytvárajú alebo upravujú štruktúru databáze sú zachytávané, parsované a na ich základe je upravený model v pamäti, ktorý popisuje štruktúru databáze. Stávajúci lexikálny parser je ručne napísaný, veľmi jednoduchý a zďaleka nepostihuje všetky nuance SQL jazyka, čím sa stáva nachylným k chybám. Novo implementovaný strojovo generovaný lexikálny parser nahradí aktuálne riešenie v projekte Debezium, čím sa zníži pravdepodobnoť vzniku chýb, a bude možné parser upraviť jednoduchou zmenou v gramatike jazyka nad ktorým bude pracovať.

TODO: vykuchat do uvodu

1.1 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 jedena odpovedajúca udalosť, ktorá bude odoslaná a následne dostupná pre konzumentov tohto sledu viď obrázok 1.1 [7]. V projekte Debezium sa na sprostredkovanie sledu události využíva Apache Kafka [9] infraštruktúra, no myšlienka CDC nie je viazaná na Kafku.



Obr. 1.1: Koncept distribúcie zmenených dát

1.1.1 Replikácia dát

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

1.1.2 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 zretelné, 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[1], čím ale bude úzko spojená a závislá na chode danej služby. Ak by takáto služba zlyhala/spadla tak nebude fungovat celá aplikácia. Druhou možnosťou je práve využiť CDC, kde produktová a skladová služba budú poskytovať sled zmenených dát a objednávacia aplikácia ich bude zachytávať a udržiavať kópiu časti týchto dát, ktoré

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

ju zaujímajú, vo vlastnej lokálnej databáze. Ak by v takomto prípade niektorá zo služieb zlyhala, tak objednávacia aplikácia môže naďalej fungovať.

TODO: zmienit ze ide o event sourcing

1.1.3 Ostatné

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

Riešenie fulltextového vyhľadávania pomocou databáze nie je veľmi vhodné a namiesto toho sa používa SOLR[2] alebo Elasticsearch[3], čo sú systémy, ktoré potrebujú byť synchronizované z dátami v primárnej databáze.

1.2 Odchytávanie zmien v databázi

Každý databázový systém (DBMS) má svoj log súbor, ktorý používa na zotavenie sa po páde a rollbacku transakcií, ktoré ešte neboli commitnuté alebo na replikáciu dát voči sekundárným databázam alebo inej funkcionalite. Či už to sú transakčné, binárne alebo replikačné logy, vždy v sebe udržujú všetky transakcie, ktoré boli úspešne vykonané nad databázou, a preto sú vhodné na odchytávanie zmien v databázach pre projekt Debezium. Konkrétne v MySQL databáze sa volá **binlog** (2.1.1). Nakoľko sú tieto logy plne transparentné voči aplikácií, ktorá do databáze zapisuje, výkon aplikácie nebude nijako ovplyvnený čítaním týchto logov.

1.2.1 Kafka

todo: co to je

1.2.2 Infraštruktúra správ pomocou Apache Kafka

Apache Kafka[9] 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 metodou FIFO[4], čím sa zabezpečí správne poradie zmien, ktoré bude konzument príjmať. Táto vlastnosť je veľmi dôležitá nakoľko ak by nastala situácia *insert* a následne *update* alebo dve *update* akcie za sebou, tak musí byť zabezpečené aby sa ku konzumentovi dostali v správnom poradí inač by mohla nastať nekonzistencia voči dátam v primárnej databáze a dátam, ktoré si udržiava konzument.

[2] <http://lucene.apache.org/solr/>
[3] <https://www.elastic.co/>
[4] First in First out

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

Zmien v databázach môže byť veľmi veľa, čo spôsobí veľké množstvo údálosti, a preto je nutné spomnúť ďalšiu výhodu Kafky a to jej škálovateľnosť. Kafka podporuje horizontálnu škálovateľnosť a jednotivé 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 událosť s iným kľúčom, ktorá reálne nastala neskôr, môže byt kozumentom spracovávana skôr, čo môže, ale aj nemusí vadiť v závisloti na konkrétnej funkcionalite konzumenta.

1.2.3 Kafka Connect

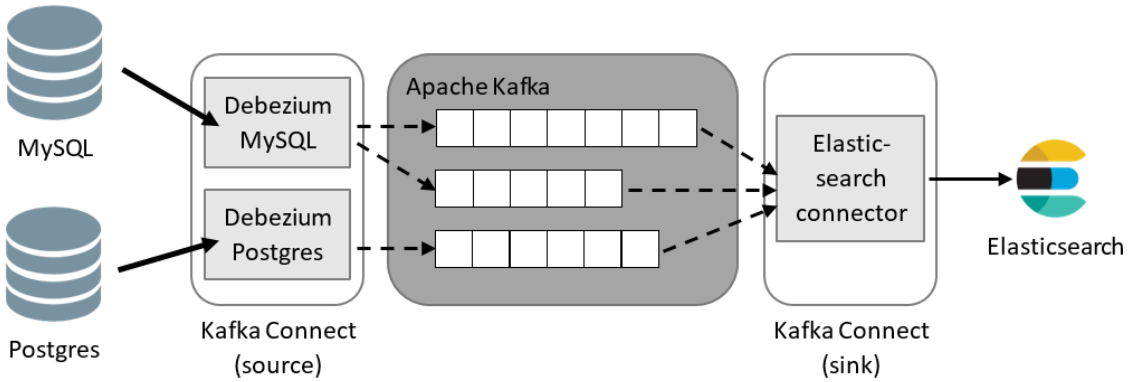
Kafka Connect je framework, ktorý umožnuje jednoduchú implementáciu dátových spojení s Kafkou. Tieto konektory majú na starosti dáta, ktoré vstupujú alebo vystupujú z Kafky. Nazývajú sa *source* konektory alebo *sink* konektory na základe toho, o čo sa starajú. Debeziové konektory majú nastarosti naplňovanie Kafky, takže sa používa *source* konektor. Kafka Connect ponúka možnosť na riešenie offsetu. To znamená, že keď konektor číta eventy z logu udržuje si zároveň pozíciu logu, z ktorej naposledy čítal. Môže nastať situácia, že konektor spadne a bude musieť byť reštartovaný. V takomto prípade konektor potrebuje vedieť ako ďaleko v čítani logu bol a kde má s čítaním pokračovať. Použitím Kafka Connect je zabezpečené, že po každom spracovaní údálosti konektor commitne svoj offset a ak by konektor musel byť reštartovaný tak môže zistit posledný commitnutý offset a pokračovať v čítaní logu z danej pozície.

todo: zdoraznit rozdiel medzi kafka offsetom a connect offsetom

Ďalším prínosom je možnosť konfigurácie schémy správ. Kafka Connect má svoj systém na definovanie typu dát, ktorým umožnuje popísať štruktúru kľúčov a hodnôt v správach. Bližšie popísané v kapitole 1.2.4.

Kafka Connect je clustrovateľná takže je možné v závislosti na špecifikácií rozdeliť konektor a jeho tasky medzi viacero uzlov. Taktiež ponúka bohatý eko-systém konektorov. Na stránkach Confluent[5] je možné si stiahnuť rôzne typy či už *sink* alebo *source* konektorov. Príklad CDC topológie s použitím Kafka Connect je na obrázku 1.2 [7]. 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 instancie 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 spracovávat dané topiky v Apache Kafka a odosielať ich inému systému (konzumentovi). Na konkrétnom príklade je použitý Elasticsearch konektor nakoľko je konzumentom Elasticsearch.

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



Obr. 1.2: CDC topológia z Kafka Connect

1.2.4 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 databáze a tabuľky, timestamp a pod.

Kafka dokáže spracovávať akýkoľvek druh binárných 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 využit 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 zhliadnut v prílohe B.1.
- **Avro**, ktorý má veľmi efektívnu a kompaktnú reprezentáciu vhodnú na produkčné účely. Takáto správa nieje čitateľná, nakoľko je to binárna reprezentácia správy. V týchto správach sa nenachádza informácia o schémy tabuľky, ale iba identifikátor na danú schému a jej verziu, ktorú je možné získať pomocou registru schémat, čo je ďalšia časť ekosystému Kafky. Konzument môže získať konkrétnu schému z registrov a na základe nej interpretovat binárne dáta, ktoré dostal.

Kapitola 2

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 funkcionalitu spojenú s CDC, ktorú tento systém podporuje. Definuje model sledovaných dát, na základe ktorých si systém udržuje aktuálne schémata tabuliek a ich dátový stav v pamäti. Jedným z týchto podporovaných konektorov je aj konektor pre MySQL databázu 2.1.

TODO co viac k tomu ?

2.1 MySQL konektor

Minimálnou podporovanou verziou MySQL je aktálne verzia 5.6. MySQL konektor Debezia dokáže sledovať zmeny v databázy na úrovni jednotlivých riadkov v tabuľkách pomocou čítania databázového binlogu (2.1.1). Pri prvom pripojení na MySQL server si konetor vytvorí aktuálny obraz všekých tabuliek (2.1.2) 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 vočí chybám. Zároveň s čítaním udalostí z binlogu si konektor ukladá ich pozíciu. Ak by nastala akákolvek 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 rovnko správať aj keby chyba a jeho pád nastali počas prvotného vytvárania aktuálneho obrazu.

2.1.1 Binárny log

v MySQL je možné implementovať CDC na základe sledovania binárneho logu v skratke nazývaného binlog[10]. Binlog obsahuje všetky události, ktoré popisujú zmeny vykonané nad MySQL databázou ako napríklad vytváranie tabuliek alebo zmena dát. Poradie týchto události 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 sprostredkuváva 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 logovanie môže taktiež obsahovať dotazy, ktoré môžu iba potencionálne meniť dáta napríklad DELETE dotaz, ktorý sa nespáruje so žiadnymi dátami. Pri replikácií slave server číta binlog a zaradom vykonáva SQL dotazy, ktoré obsahujú jednotlivé udalosti.
- **Row-based** logovanie, v ktorom udalosti popisujú zmeny pre jednotlivé riadky v tabuľkách. Pri replikácií sa kopírujú udalosti, ktoré reprezentujú zmeny riadkov v tabuľkách na slave serveri.

Pre účely CDC v Debeziu je používané row-based logovanie, nakoľko zalogované udalosti obsahujú zmeny pre konkrétne riadky v tabuľkách a tým pádom nieje nutné dopočítavať dáta, ktoré by boli na základe daného dotazu zmenené. Master server ukladá do binlogu iba kompletné a vykonané transakcie, čo znamená, že sa tam nemôžu objaviť syntakticky nevalidné dotazy. V MySQL konektoru 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 začiatočná a jeho konečná pozícia v logu. Na čítanie a spracovávanie binlogu ponúka MySQL nástroj *mysqlbinlog*, ktorý je možné spustiť príkazom *mysqlbinlog [options] log_file*. Prvý riadok udalosti vždy obsahuje prefix *# at* za ktorým následuje číslo reprezentujúce pozíciu udalosti v binlogu. Podľa základného nastavenia MySQL zobrazuje mysqlbinlog udalosti týkajúce sa zmien na úrovní riadkov zakódované ako base-64[1] použitím interného príkazu *BINLOG*. Aby bolo možné vidieť tento pseudokód je možné použiť prepínač *—verbose* alebo *-v*. Na výstupe bude možné vidiet 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 sledovat obsah udalosti týkajúcich sa zmien v dátach. [10]

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

Pre Debezium sú dôležité události typu:

- **Query**, v ktorom sa objavujú dotazy na zmenu štruktúry databáze (DDL) ako je možné vidiet na poslednom riadku príkladu události 2.1.
- **Table_map**, pomocou ktorého binlog mapuje konkrétne tabuľky na identifikátor, ktorým sa následne tieto tabuľky referencuje. Príklad tejto události je možné vidieť v príklade 2.2 na pozícii 552 a jeho následné použitie pre událosť na pozícii 621.
- **Update_rows**, ktorý obsahuje informácie o zmene dát na úrovni riadkov, ako je možné vidiet na události v príklade 2.2 na pozícii 621.
- **Write_rows**, ktorý obsahuje infomrácie o novo vzniknutých dátach.
- **Delete_rows**, ktorý obsahuje informácie o zmazaných dátach.

Ukážka 2.1: Query událost 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 2.2: 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 */
```

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

2.1.2 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[2]:

1. Aktivuje globálny zámok čítania (read lock) aby zabránil ostatným databázovým klientom v zapisovaní.
2. Spustí transakciu s izoláciou na opakované čítanie (repeatable read)[2], aby všetky nasledujúce čítania v rámci tejto transakcie boli voči jednému konzistentnému obrazu.
3. Prečíta aktuálnu pozíciu binlogu.
4. Prečíta schéma databází a tabuliek na základe konfigurácie konektoru.
5. Uvolní globálny zámok, aby ostaný 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* události Kafka topiky pre jednotlivé riadky v tabuľkách.
8. Commitne transakciu.
9. Do konektorového offsetu zaznamená, že úspešne ukončil vytváranie obrazu.

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

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

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

2.2 DDL parser

Pri čítaní binárneho logu MySQL konektor parsuje DDL dotazy na základe ktorých si v pamäti vytvára modely schémat každej tabuľky podľa toho ako sa vyvíjali v čase. Tento proces je veľmi dôležitý, pretože konekor generuje události pre tabuľky, v ktorých definuje schéma tabuľky v čase, kedy daná událosť vznikla. Aktálne schéma sa nemôže použiť, nakoľko sa môže zmeniť v danom čase pípadne 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 událosť týkajúci 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 události.

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

Nato aby bolo možné túto akciu vykonať je nutné parsovať DDL dotazy, pričom pre potreby Debezia stačí vedieť rozpozať 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.

2.2.1 Framework na parsovanie DDL

Keďže Debezium nenašlo žiadnu použiteľnú knižnicu na parsovanie DDL, rozhodlo sa implementovať vlastný framework podľa ich potrieb, ktoré sú[11]:

- Parovanie DDL dotazov a aktualizácia modelu v pamäti.
- Zameranie sa na podstatné dotazy ako sú `CREATE`, `UPDATE` a `DROP` tabuliek, pričom sa ostané 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 jednoduhšie udržiavať v priebehu času.

- Umožniť vytvorenie parserov pre PostreSQL, Oracle, SQLServer a všetkých ostatných DBMS, ktoré budú potrebné.
- Umožniť prispôsobenie pomocou dedičnosti a polymorfismu.
- Uľahčit 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, kľučové slová, komentáre a ukončujúce znaky ako napríklad bodkočiarku pre MySQL. DDL parser prechádza sled tokenov a volá metódy na spracovanie variácii sady tokenov.Parser taktiez využíva interny `DataTypeParser` na spracovanie dátových typov SQL, ktoré si je možné pre jednotlivé DBMS ručne zaregistrovať.

`MySqlDdlParser` trieda dedí od základnej triedy `DdlParser` a sprostredkuváva celú parsovaciu logiku špecifickú pre MySQL. Napríklad DDL dotaz 2.3 je možné sparsovať podľa ukážky 2.4.

Ukážka 2.3: 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 2.4: 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.

2.2.2 Implementácia MySQL DDL parsru

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

Pri každom volaní metódy `parseNextStatement` je predavaný objekt `Marker`, ktorý ukazuje na začiatočnú pozíciu parsovaného dotazu. Vďaka polymorfizmu `MySqlDdlParser` prepisuje implementáciu `parseNextStatement` metódy (ukážka C.2), v ktorej kontroluje, čí prvý token vyhovuje niektorému z typov MySQL DDL gramatiky. Po najdení 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 skonzumuje 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 najprv skonzumovať nepovinné slovo *TEMPORARY*, potom slovo *TABLE*, nepovinný fragment *IF NOT EXISTS* a následne konzumuje a parsuje názov tabuľky. Ak by dotaz obsahoval fragment *LIKE otherTable*, tak sa použije objekt `Tables`, z ktorého sa získa definícia odkazovanej tabuľky. V ostatných prípadoch sa na úpravu stávajúcej tabuľky použije `TableEditor` objekt. Takýmto spôsobom parser pokračuje vo svojej činnositi ďalej a snaží sa parsovať dotaz na základe pravidiel gramatiky.

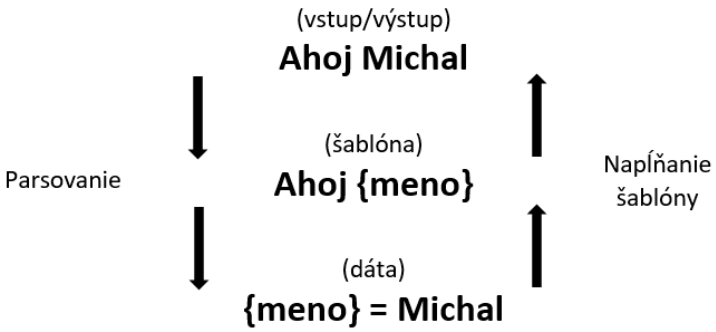
TODO: spisat info ohladne ukladanych informacii

Kapitola 3

Syntaktická analýza

Syntaktickou analýzou (slangovo z angličtiny tiež **parsovaním**) sa v teórii rozumie konštrukcia derivačného stromu vety bezkontextového jazyka[8] popísaného v kapitole 3.1.3. Program, ktorý vykonáva túto úlohu sa volá syntaktický analyzátor (slangovo **parser**). Počas konštrukcie deriačného stromu parser zachováva hierarchické usporiadanie symbolov, koré 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 identufukuje túto šablónu a extrahuje dáta, ktoré boli do nej vložené.



Obr. 3.1: Príklad parsovania a naplňovania šablóny

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

3.1 Parsovanie pomocou regulárnych výrazov

Regulárne výrazy (3.1.2) poskytujú možnosť zápisu regulárnych jazykov, ktorých fungovanie je postavené na deterministických konečných automatoch (3.1.1).

O regulárných 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. Niektorý programátori nepoznajú iné možnosti a snažia sa všetko parsovať s použitím regulárných 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 elemety, nejedná sa už o regulérny jazyk, ale o jazyk *bezkontextový* (3.1.3). Väčšina programovacích jazykov spadá pod bezkontextové jazyky, a preto nie je možné tieto jazyky parsovať regulérnymi výrazmi.

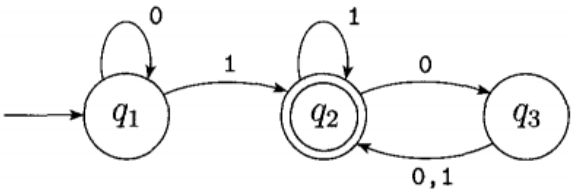
3.1.1 Deterministický konečný automat

Konečné automaty sa používajú v rôznych oboroch ako napríklad pri prekladačoch, spracovávaní prirodzeného jazyka, pri návrhu hardwaru a ďalších [3]. Predstavujú model systémov, ktoré rozpoznajú, či je vstupný reťazec patrí do jazyka. Deterministický konečný automat (DFA), taktiež aj *akceptor* je najpoužívanejší zo štyroch typov automatov.

Definícia 3.1.1. *Deterministický konečný automat* M je pätica $M = (Q, \Sigma, \delta, q_0, F)$, kde

- Q je konečná množina stavov
- Σ je konečná množina vstupných symbolov
- δ je prechodová funkcia $\delta : Q \times \Sigma \to Q$
- q_0 je počiatočný stav
- $F \subseteq Q$ je množina koncových stavov [3]

Konečný automat je možné prehľadne znázorniť formou stavového diagramu. *Stavový diagram* je orientovaný graf, v ktorom sú uzly ohodnotené stavmi automatu a hrany vstupnými symbolmy 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 3.2.



Obr. 3.2: Príklad DFA znázorneného pomocou stavového diagramu

Definícia 3.1.2 (**Jazyk príjmaný konečným automatom**). Je daný DFA $M = (Q, \Sigma, \delta, q_0, F)$. Slovo $u \in \Sigma^*$ je *príjmané* automatom M práve vtedy, keď

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

Množina vštkých slov, ktoré automat príjma sa nazýva *jazyk príjmaný* M a značíme ju $L(M)$. Platí teda

$$L(M) = \{\omega | \delta^*(q_0, u) \in F\}.[3]$$

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

3.1.2 Regulárny výraz

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

Definícia 3.1.3 (**Regulárne výrazy nad abecedou**). Je dána abeceda Σ. Množina všetkých regulárnych výrazov nad Σ je definovaná induktívne:

- $\emptyset$ je regulárny výraz,
- ϵ je regulárny výraz,
- $\mathbf{a}$ je regulárny výraz pre každé písmeno $a \in \Sigma$,
- pokiaľ sú $\mathbf{r}_1$ a $\mathbf{r}_2$ regulárne výrazy, tak $\mathbf{r}_1 + \mathbf{r}_2$, $\mathbf{r}_1\mathbf{r}_2$ a $\mathbf{r}_1^*$ sú regulérne výrazy. [3]

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

3.1.3 Bezkontextový jazyk

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

Definícia 3.1.4. *Bezkontextová gramatika* je usporiadaná štvorica $\mathcal{G} = (N, \Sigma, S, P)$, kde

- N je konečná množina tzv. *neterminálov*[1]
- Σ je konečná neprázdna množina tzv. *terminálov*[2], kde platí $N \cap \Sigma = \emptyset$

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

- $S \in N$ je *štartovací symbol*
- P je konečná množina pravidiel typu $\alpha \to \beta$, kde α a β sú slová nad $N \cup \Sigma$ taká, že α obsahuje aspoň jeden neterminál.
- každé pravidlo P je v tvare $A \to \gamma$, kde $\gamma \in (n \cup \Sigma)*$ a A je neterminál [4]

Bezkontextové gramatiky sa prvykrát 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 [12].

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

V nasledujúcom príklade je ukážka bezkontextovej gramatiky G_1.

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

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

Definícia 3.1.5 (**Derivace**)**.** Je daná gramatika $\mathcal{G} = (N, \Sigma, S, P)$. Povedzme, že δ sa *odvodí* z γ vtedy, ak

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

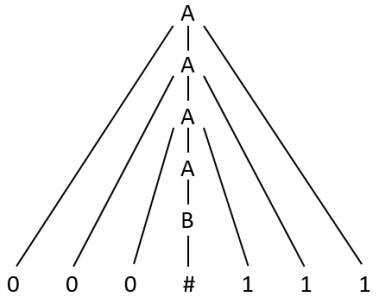
$$\gamma = \gamma_1 \Rightarrow_{\mathcal{G}} \gamma_2 \Rightarrow_{\mathcal{G}} ... \Rightarrow_{\mathcal{G}} \gamma_k = \delta$$

Tento fakt sa označuje $\gamma \Rightarrow^*_{\mathcal{G}} \delta$ a tejto konečnj postupnosti hovoríme *derivácia*. [4]

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

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

Rovnakú informáciu je možné reprezentovat graficky pomocou sparsovaného (derivačného) stromu. Príklad derivačného stromu je na obrázku 3.3.



Obr. 3.3: 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 gramtiku 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 3.1.6 (**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.

$$L(\mathcal{G}) = \{\omega \in \Sigma^* | S \Rightarrow_{\mathcal{G}}^* \omega\}.[4]$$

3.1.4 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.[6]

Každé pravidlo v BNF má následujúcu štruktúru:

$$<neterminál> ::= výraz$$

Všetky netreminály v BNF sa zapisujú do špicatých zátvoriek $< >$, či už sú použité na pravej alebo lavej 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.

3.1.5 Rozšírená Backus-Naur Form notácia

Pre zjednodušenie zápisu gramatiky, a aby bolo možné jednoduhšie definovať určité typy pravidel vznikla kolekcia rozšírení k Backus-Naur Form notácii (EBNF), ktorá bola štandardizovaná ako ISO/IEC 14997[5]. Terminály môžu byť vyjadrené konkrétnym postupom znakov v uvodzovkách, alebo pomocou triedy literálov, ktorú je možné zapísať pomocou regulárneho výrazu. Priradzovací znak pravidla je zmenený z ::= na jednoduché = a vynecáhva sa zápis špicatých zátvoriek okolo neterminálov. Tieto malé syntaktické zmeny nie sú tak dôležité ako dodatočné operácie EBNF, ktoré sa môžu použiť vo výraze.

Nepovinosť - Použitím hranatých zátvoriek okolo výrazu */výraz/* sa indikuje možosť použitia tohto výrazu v sekvencii. Jednoduhšie povedané, výraz môže, ale nemusí byť použitý vo výsldnej sekvencii. Príklad:

$$\text{term} = [\text{"-"}]\ \text{factor}$$

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

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

Opakovanie - Použitím zložených zátvoriek okolo výrazu *{výraz}* je možné indikovať opakovanie výazu. 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:

$$\text{args} = \text{arg } \{\text{","} \text{ arg}\}$$
$$\text{args} = \text{arg } (\text{","} \text{ arg})^*$$

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

Každú gramatiku zapísanú cez EBNF je možné taktiež zpísať pomocou BNF, to ale vedie k omnoho obsiahlejšiemu množstvu definičných pravidiel. V nasledujúcich príkladoch sú znároznené 2 rôzne zápisy v EBNF gramatiky z kapitoly 3.1.3:

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

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

3.2 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 **tokeny** a definujú identifikátory, literály (čísla, reťazce), kľúčové slová, operátory, oddeľovače a pod. Pre parser sú tokeny ďalej nedeliteľné stabevbné jednotky, ktoré používa pri interpretácií vstupných dát. Porgram vykonávajúci túto úlohu sa nazýva štrukturálny analyzátor, no v programovaní sa častejšie narazí na výraz **lexer** alebo **tokenizer** bližšie popísaný v kapitole 3.2.1.

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 súčasťov kompilátorov alebo interpretorov programovacích jazykov, no samozrejme môžu byť súčasťou aj rôznych typov programov. Čo sa týka parsovania programovacích jazykov, parser dokáže určiť iba syntakticú korektnosť parsovaného výrazu. Výstup parseru je ale základom pre zistanie sémantickej korektnosti.

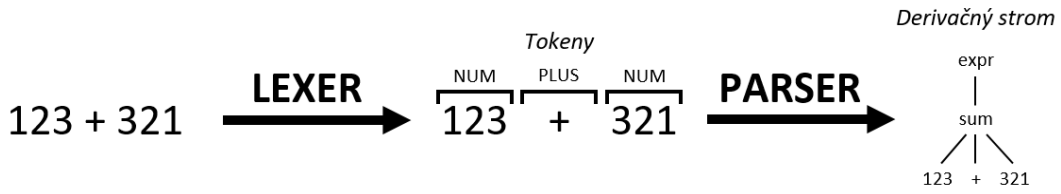
3.2.1 Lexer

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

Jedna z veľmi dôležitých úloh lexera je vysporiadanie sa z medzerami v parsovanom výraze. Vo väčšine prípadoch chceme, aby prázdne medzery boli lexerom odstránené. Ak by sa tak nestalo, znamenalo by to, že by sa s nimi musel vysporiadať samotný parser. To by znamenalo ich kontorlu 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). [13]

Lexer prečíta vstupný preťazec 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(3.1.1).



Obr. 3.4: Spracovanie reťazca *123 + 321* lexerom a parserom

Pre príklad z obrázku 3.4 máme dva typy tokenov. **NUM** vyjadrujúci akékoľvek prirodzené číslo a **PLUS** vyjadrujúci znak súčtu (+). Keď sa lexer bude snažiť analyzovať reťazec **123 + 321**, bude čítat znaky *1,2,3* a potom znak medzery. V tomto momente lexer rozpozná, že postupnosť znakov *123* súhlasí z definíciou tokenu typu NUM. Následne prečíta znak *+*, ktorý sa zhoduje s druhým typom tokenu PLUS a nakoniec objaví posledný token typu NUM. Takto definované tokeny použije parser na vyhodnotenie výsledného výrazu. Bezkontextová gramatika pre takýto parser by mohla vyzerať následovne:

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

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

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

Ak by bol ale problém zložitejší a program by chcel analzovať IP adresu a zistiť z nej informácie ako napríklad krajinu, bude parser potrebovať jednotlivé hodnoty IP adresy samostatne. V tomto prípadde 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
```

Kapitola 4

Parsovacie algoritmy

Literatúra

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

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

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

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

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

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

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

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

[9] 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.

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

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

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

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

Dodatok A

Zoznam použitých skratiek

BNF Backus-Naur Form

CDC Change Data Capture

DBMS Database management system

DDL Data definition language

DFA Deterministic Finite Automaton

EBNF Extended Backus-Naur Form

SQL Structured Query Language

Dodatok B

Ukážka dát

Ukážka B.1: Ukážka CDC správy odoslanej Debeziom

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

Dodatok C

Ukážky zdrojových kódov

Ukážka C.1: Parsovacie metódy DDL parserov

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

Ukážka C.2: Implementácia parseNextStatement metódy v MySqlDdlParser

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

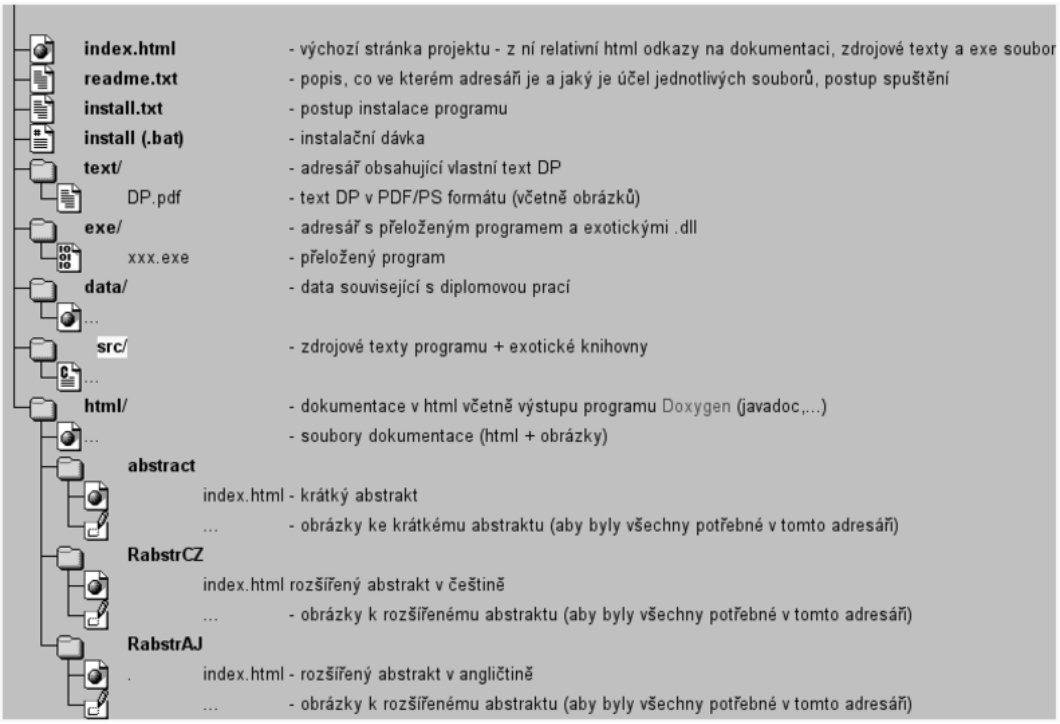
Ukážka C.3: Implementácia parseCreateTable metódy v MySqlDdlParser

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

Dodatok D

Obsah přiloženého CD

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



Obr. D.1: Seznam přiloženého CD — příklad

můžete snadno vyrobit příkazem:

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

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

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

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