

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

7. 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	Ostané	3
1.2	Odchytávanie zmien v databázi	3
1.2.1	Infraštruktúra správ pomocou Apache Kafka	3
1.2.2	Kafka connect	4
1.2.3	Štruktúra správy	5
2	Aktuálna 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
A	Zoznam použitých skratiek	17
B	Ukážka dát	19
C	Ukážky zdrojových kódov	21
D	Obsah príloženého CD	25

Zoznam obrázkov

1.1	Koncept distribúcie zmenených dát	2
1.2	CDC topológia z Kafka connect	5
D.1	Seznam přiloženého CD — příklad	25

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	19
C.1	Parsovacie metódy DDL parserov	21
C.2	Implementácia parseNextStatement metódy v MySqlDdlParser	22
C.3	Implementácia parseCreateTable metódy v MySqlDdlParser	23

Kapitola 1

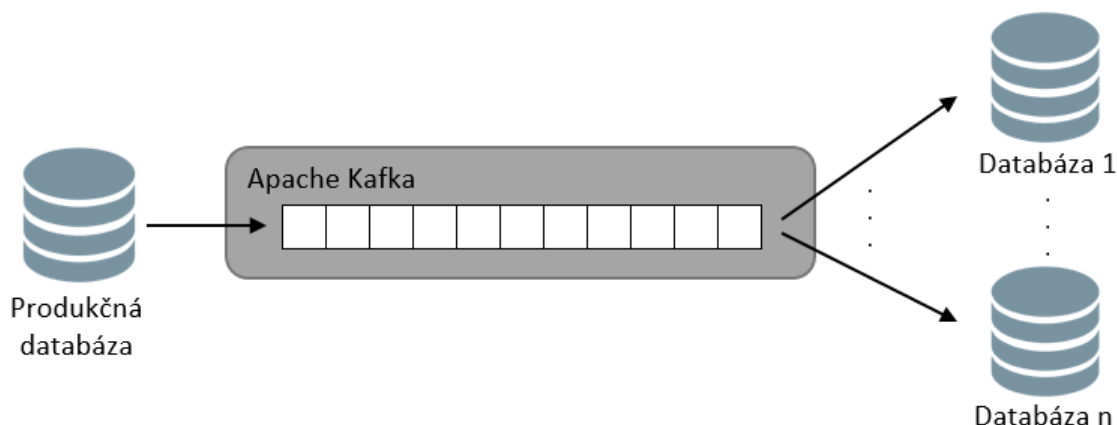
Debezium

Debezium[1] je projekt, ktorý slúži k zaznamenávaniu zmien v databázi analýzou udalostí v transakčnom logu. Jednou z podporovaných databázi je taktiež MySQL. Na správnu funkcionálnosť 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 náchylným k chybám. Novo implementovaný strojovo generovaný lexikálny parser nahradí aktuálne riešenie v projekte Debezium, čím sa zníži pravdepodobnosť 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 jedna odpovedajúca udalosť, ktorá bude odoslaná a následne dostupná pre konzumentov tohto sledu vid' obrázok 1.1 [3]. V projekte Debezium sa na sprostredkovanie sledu udalostí využíva Apache Kafka [4] 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 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 necheme 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.

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 zreteľné, že na správne fungovanie bude objednáva aplikácia vyžadovať dáta od produktovej a skladovej služby. Jednou z možností je, že objednávaciu aplikáciu bude priamo komunikovať s ostatnými službami napríklad pomocou REST api¹, 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, kde produktová a skladová služba budú poskytovať sled zmenených dát a objednávaciu aplikáciu ich bude zachytávať a udržiavať kópiu časti týchto dát, ktoré ju

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

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

1.1.3 Ostané

Bežnou praxou vo väčších aplikáciach 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 databáze nie je veľmi vhodné a namiesto toho sa používa SOLR² alebo Elasticsearch³, 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 rollbacknutie transakcií, ktoré ešte neboli commitnuté 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** (2.1.1). Nakoľko sú tieto logy plne transparentné voči aplikáciám, ktorá do databáze zapisuje, výkon aplikácie nebude nijako ovplyvnený čítaním týchto logov.

1.2.1 Infraštruktúra správ pomocou Apache Kafka

Apache Kafka[4] 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 FIFO⁴, 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 inak by mohla nastať nekonzistencia voči dátam v primárnej databáze a dátam s 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 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 údajosti, a preto je nutné spomnúť ďalšiu výhodu kafky a to jej škálovateľnosť. Kafka podporuje horizontálnu

²<http://lucene.apache.org/solr/>

³<https://www.elastic.co/>

⁴First in First out

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 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.

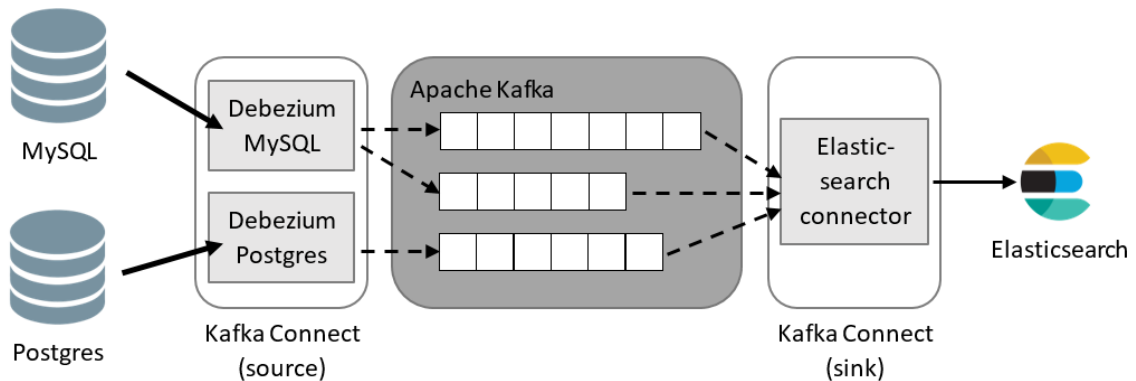
1.2.2 Kafka connect

Kafka connect je framework, ktorý umožňuje jednoduchú implementáciu dátových spojení z 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 na starosti naplňovanie Kafky, takže sa používa *source* konektor. Kafka connect ponúka možnosť na riešenie ofsetu. To znamená, že keď konektor číta eventy z logu udržiava 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 čítaní logu bol a kde má s čítaním pokračovať. Použitím Kafka connect je zabezpečené, že po každom spracovaní udalosti konektor commitne svoj ofset a ak by konektor musel byť reštartovaný tak môže zistiť posledný comutnutý ofset a pokračovať v čítaní logu z danej pozície.

Ďalším prínosom je možnosť konfigurácie schémy správ. Kafka connect má svoj typovací systém, ktorý umožňuje popísať štruktúru kľúčov a hodnôt v správach. Bližšie popísané v kapitole 1.2.3.

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 Confluent⁵ je možné si stiahnuť rôzne typy či už *sink* alebo *source* konektorov. Príklad CDC topológie s použitím Kafka connect je na obrázku 1.2 [3]. 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 to môže bežať na jednom alebo viacerých clustroch. 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 deploynúť 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 spracovávať 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

1.2.3 Štruktúra správy

Ako už bolo spomenuté, správy v Kafke obsahujú kľúč, v prípade Debezia je 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 data. Táto hodnota môže byť opäť prázdna a to v prípade *delete* eventu.
- **source** informácie, ktoré v obsahujú metadáta o pôvode danej zmeny. Skladá sa napríklad z informácií ako meno databázového serveru, názvu logovacieho súboru, z ktorého číta a pozíciu v ňom, názvu databázy a tabuľky, timestamp a pod.

Kafka dokáže spracovávať akýkoľvek druh binárnych dát, takže jej na tejto logickej štruktúre nezáleží. Na odosielanie správ sa používajú konventory, ktoré prevádzajú správu do formy v ktorej bude odosielaná. Vďaka použitiu Kafka connect je opäť možnosť využiť konventory, 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 čitateľná, 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 2

Aktuálna riešenie MySQL konektoru

Projekt Debezium sa zkladá z viacerých častí. Hlavnou časťou je jadro systému, ktoré je spoločné pre všetky typy konektorov podporovaných Debeziom. Jadro systému zaobstaráva základnú funkcionálnu spojenú s CDC, ktorú tento systém podpruje. 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 [2.1](#).

TODO co viac k tomu ?

2.1 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 ([2.1.1](#)). Pri prvom pripojení na MySQL server si konektor vytvorí aktuálny obraz všetkých 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 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.

2.1.1 Binárny log

v MySQL je možné implementovať CDC na základe sledovania binárneho logu v skratke nazývaného binlog[5]. 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 servri sprostredkúva záznamy o zmenách, ktoré majú byť odoslané slave serverom. Master server odošle udalosti nachádzajúce sa v binlogu slave servri, ktorý tieto udalosti vykoná u seba za účelom udržania rovnakého dátového stavu ako je na master replikačnom servri.
- Pre **obnovu systému z chybového stavu** anlicky nazývanú **recovery**. Po nahraní zálohy databáze sú znovuspustené udalosti zaznamenané v binlogu, ktoré nastali po vytvorení zálohy, čím sa zabezpečí konzistentný stav databázových dát z dátami v dobe zlyhania databázového servri.

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 eventy, ktoré reprezentujú zmeny riadkov v tabuľkách na slave servri.

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 kompletne a vykonané transakcie, čo znamená, že sa tam nemôžu objaviť syntakticky nevalidné dotazy. 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 servri. 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-64¹ 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 udalostí týkajúcich sa zmien v dátach.

¹typ kódovania, ktorý prevádza binárne dáta na postupnosť znakov

Pre Debezium sú dôležité události typu:

- **Query**, v ktorom sa objavujú dotazy na zmenu štruktúry databáze (DDL) ako je možné vidieť 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 udalosť na pozícii 621.
- **Update_rows**, ktorý obsahuje informácie o zmene dát na úrovni riadkov, ako je možné vidieť na události v príklade 2.2 na pozícii 621.
- **Write_rows**, ktorý obsahuje informácie o novo vzniknutých dátach.
- **Delete_rows**, ktorý obsahuje informácie o zmazaných dátach.

Ukážka 2.1: 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 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 */

```

```

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 */

```

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

²stupeň 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.

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 udžiavaný na veľmi krátku dobu potrebnú pre konektor na vykonanie krokov tri a štyri. V piatom kroku je tento zámok uvoľnený predtým, než konektor vykoná väčšinu práce pri kopírovaní údajov.

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 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 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 udalosť 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 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 Debezium stačí vedieť rozpoznať iba malú časť z celej DDL gramatiky. Model, ktorý sa udržiava v pamäti a zbytok funkcionality spojený z generovaním **Schema** objektu a konvertoru hodnôt na **Struct** objekt je generické nakoľko nie je priamo spojené z MySQL.

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[6]:

- 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 jednoduchšie udržiavať v priebehu času.

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

Výsledný framework pozostáva z tokenizeru, ktorý konvertuje DDL dotaz v jednom reťazci na sekvenciu tokenov. Každý token reprezentuje 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 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 spracovuje 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 ne našla zhoda, parser vygeneruje `ParsingException`, ktorá obsahuje riadok, stĺpec a chybovú hlášku oznamujúcu aký token bol očakávaný a aký sa našiel. V prípade chyby sa `TokenStream` pretočí na začiatok, aby sa prípadne mohola 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, 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 nasledovné 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 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.

TODO: písat info ohľadne ukladaných informácií

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] MORLING, G. Streaming Database Changes with Debezium. <https://www.youtube.com/watch?v=IOZ2Um6e430>, publikované 9.11.2017.
- [4] 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.
- [5] *MySQL 5.7 Reference Manual*. Oracle Corporation and/or its affiliates, 2018. <https://dev.mysql.com/doc/refman/5.7/en/>.
- [6] Randall Hauch. *Parsing DDL* [online]. 2018. [cit. 25.2.2018]. Dostupné z: <<http://debezium.io/blog/2016/04/15/parsing-ddl/>>.
- [7] web:infodp. K336 Info — pokyny pro psaní diplomových prací. <https://info336.felk.cvut.cz/clanek.php?id=400>, stav ze 4.5.2009.

Dodatok A

Zoznam použitých skratiek

CDC Change Data Capture

DBMS Database management system

DDL Data definition language

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: Parsovanie metódy DDL parserov

```
1 public final void parse(String ddlContent, Tables databaseTables) {
2     Tokenizer tokenizer = new DdlTokenizer(!skipComments(),
3         ↪ 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)
9     ↪ throws ParsingException, IllegalStateException {
10     this.tokens = ddlContent;
11     this.databaseTables = databaseTables;
12     Marker marker = ddlContent.mark();
13     try {
14         while (ddlContent.hasNext()) {
15             parseNextStatement(ddlContent.mark());
16             // Consume the statement terminator if it is still there ...
17             tokens.canConsume(DdlTokenizer.STATEMENT_TERMINATOR);
18         }
19     } catch (ParsingException e) {
20         ddlContent.rewind(marker);
21         throw e;
22     } catch (Throwable t) {
23         parsingFailed(ddlContent.nextPosition(), "Unexpected exception ("
24             ↪ + t.getMessage() + ") parsing", t);
25     }
26 }
```

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    }
```

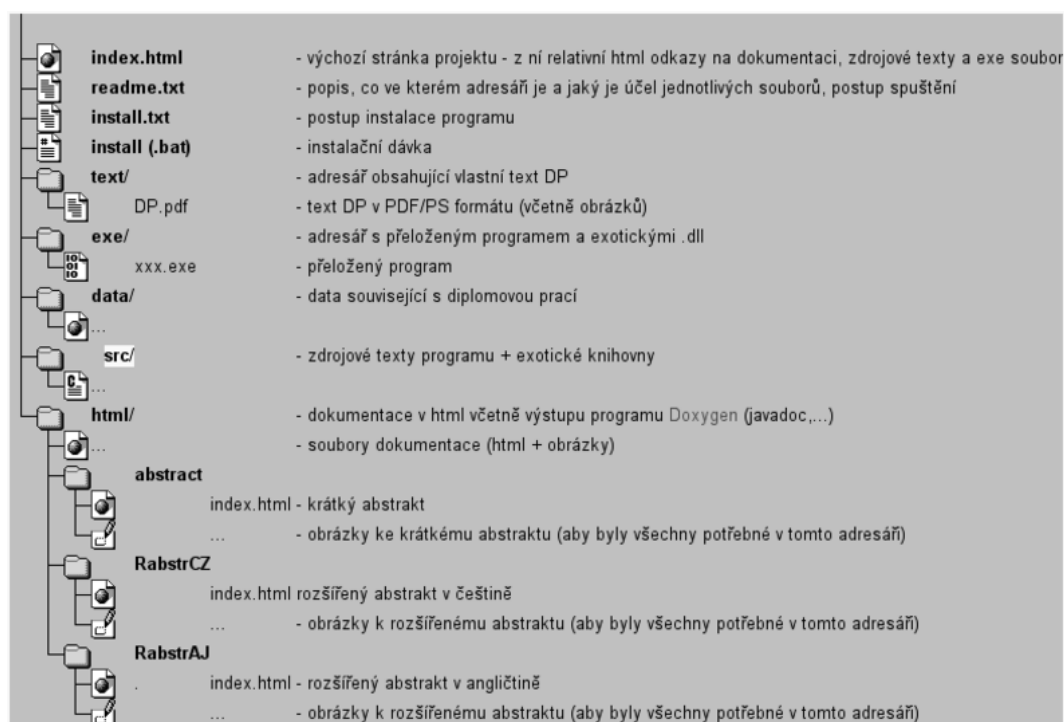
Ukážka C.3: Implementácia parseCreateTable metódy v MySqlDdlParser

```
1  protected void parseCreateTable(Marker start) {
2      tokens.canConsume("TEMPORARY");
3      tokens.consume("TABLE");
4      boolean onlyIfExists = 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             databaseTables.overwriteTable(tableId, original.columns(),
11                 ↪ original.primaryKeyColumnNames(), original.defaultCharsetName());
12         }
13         consumeRemainingStatement(start);
14         signalCreateTable(tableId, start);
15         debugParsed(start);
16         return;
17     }
18     if (onlyIfExists && 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     if (tokens.canConsume("AS") || tokens.canConsume("IGNORE", "AS") ||
36         ↪ tokens.canConsume("REPLACE", "AS")) {
37         parseAsSelectStatement(start, table);
38     }
39     // Make sure that the table's character set has been set ...
40     if (!table.hasDefaultCharsetName()) {
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

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 [7]): Na GNU/Linuxu si strukturu příloženého



Obr. D.1: Seznam příloženého CD — příklad

CD 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.