flink-connector-dorisdb

背景

flink的用户想要将数据sink到doris当中，但是flink官方只提供了flink-connector-jdbc, 不足以满足导入性能要求，需要新增一个flink-connector-dorisdb，内部实现为缓存并批量stream load导入。

设计目标

基于flink的DynamicTableSink API实现一个flink-connector-dorisdb将用户的insert行为转为batch stream load进行导入

官方connectors api有三种器均y由 Source/Sink 共同构成，我们这里只实现Sink功能

其中第一类使用方式如下：DataStream Connectors

Java

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env
        .fromElements(...)
        .addSink(JdbcSink.sink(
                "insert into books (id, title, author, price, qty) values
(?,?,?,?,?)",
                (ps, t) -> {
                    ps.setInt(1, t.id);
                    ps.setString(2, t.title);
                    ps.setString(3, t.author);
                    ps.setDouble(4, t.price);
                    ps.setInt(5, t.qty);
                },
                new JdbcConnectionOptions.JdbcConnectionOptionsBuilder()
                        .withUrl(getDbMetadata().getUrl())
                        .withDriverName(getDbMetadata().getDriverClass())
                        .build()));
env.execute();
```

第二类为Table & SQL Connectors如：

Java

```
    tEnv.executeSql(
      "CREATE TABLE USER_RESULT(" +
        "NAME VARCHAR," +
        "SCORE BIGINT" +
        ") WITH ( " +
        "'connector.type' = 'jdbc'," +
        "'connector.url='" + DB_URL + "'," +
        "'connector.table' = '" + OUTPUT_TABLE3 + "'" +
        ")");

    tEnv.executeSql("INSERT INTO USER_RESULT\n" +
        "SELECT user_name, score " +
        "FROM (VALUES (1, 'Bob'), (22, 'Tom'), (42, 'Kim'), " +
        "(42, 'Kim'), (1, 'Bob')) " +
        "AS UserCountTable(score, user_name)").await();

```

第三类 DataSet Connectors 用于读取/写入其他类型的文件系统，所以暂不在考虑范围

最终我们根据用户使用的方式选择第二类Table & SQL Connectors基于flink的checkpoint机制以两种strategy实现Sink功能

- At-least-once
- Exactly-once

用户接口说明

用户输入

因此我们实现的connector定义接口参数如下：

SQL

```sql
CREATE TEMPORARY TABLE cvs_user_tree_sink (
    k1 TINYINT,
    k2 VARCHAR,
    v1 BIGINT,
    v2 VARCHAR
) WITH (
    'connector' = 'dorisdb',
    'jdbc-url' = 'jdbc:mysql://fe_ip1:query_port,fe_ip2:query_port...',
    'load-url' = 'fe_ip1:http_port,fe_ip2:http_port,fe_ip3:http_port',
    'username' = 'root',
    'password' = 'xxx',
    'database-name' = 'db',
    'table-name' = 'tbl',
    'sink.semantic' = 'at-least-once',
    'sink.buffer-flush.max-bytes' = '10000',
    'sink.buffer-flush.max-rows' = '10000',
    'sink.buffer-flush.interval-ms' = '1000',
    'sink.max-retries' = '3',
    'sink.use-temp-partition' = 'false',
    'sink.properties.*' = 'xxx'
)
```

必填项

- Connector: 我们的名称 固定为 dorisdb
- jdbc_url: 指定jdbc连接地址（多个）使用的driver为

XML

```xml
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.49</version>
        </dependency>
```

- load_url: stream load 的ip:port 组，使用轮询选择可用的地址进行flush
- Username, password: doris用户名密码
- Database-name, table-name: 要导入的库.表

选填项

- sink.semantic: 数据容错保证。at-least-once 或者 exactly-once， default：at-least-once
- sink.buffer-flush.max-bytes：缓存最大的数据size 单位 byte，使用转换后的json数据做计算。[64MB, 10GB] default: 64MB
- sink.buffer-flush.max-rows：缓存最大的数据行数。[64k， 500w] default：64k
- sink.buffer-flush.interval-ms: 自上一次flush结束（包括重试）,后开始计算interval时间到达后进行flush。[1s, 3600s] 0无限制，default: 300
- sink.max-retries：最大重试次数。 [0, 10] default: 1
- sink.use-temp-partition: 导入时是否先创建临时分区并导入临时分区后替换原分区，仅当 sink.properties.* 中包含 `partition: value` 时有效。 default: false
- sink.properties.*：doris stream load 的参数列表 sink.properties.partition, sink.properties.columns .....

当sink.semantic为exactly-once 时以下选项无效：

`sink.buffer-flush.max-bytes, sink.buffer-flush.max-rows, sink.buffer-flush.interval-ms`

此时自动设置 `flush-interval=checkpoint-interval`

技术内幕解析

不影响现有dorisdb系统，单纯调用stream load接口进行数据sink。

设计折衷

支持exactly-once的数据sink保证，需要外部系统的 twophasecommit 机制。

由于dorisdb无此机制，我们只能依赖flink的checkpoint-interval在每次checkpoint时阻塞flush所有缓存数据，以此达到精准一次但如果doris挂掉了会导致用户的flink sink stream 算子长时间阻塞并引起flink的监控报警或强制kill。

因此我们默认执行 at-least-once 的数据保证，即在每次checkpoint保存当前所有（以相同的label进行打包）缓存数据，当系统恢复时强制flush上次保存的所有数据并清空状态。暂时改为不保存数据直接阻塞flush（缺点为重复数据可能会更多，优点为不会占用更多内存）

详细设计

继承 RichSinkFunction 并实现 CheckpointedFunction

Java

```java
// RichSinkFunction
@Override
    public void open(Configuration parameters) throws Exception
@Override
    public void invoke(RowData value, Context context) throws IOException
@Override
    public void close() throws Exception

// CheckpointedFunction
@Override
    public void initializeState(FunctionInitializationContext context)
@Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception
```

表结构校验

此操作仅在 `sink.properties.*` 不包含 `columns: value` 时进行。

用户执行insert into的sink时触发open事件：

a. 校验用户的输入参数: 必填参数（flink默认validate util）
b. 检查jdbc连通性
c. 执行以下语句获取doris中列信息

SQL

```sql
select `COLUMN_NAME`, `DATA_TYPE`, `COLUMN_SIZE`, `DECIMAL_DIGITS`
from `information_schema`.`COLUMNS`
where `TABLE_SCHEMA`=? and `TABLE_SCHEMA`=?;
```

并check doris中的字段类型与flink是否匹配

Java

```java
        typesMap.put("bigint", LogicalTypeRoot.BIGINT);
        typesMap.put("largeint", LogicalTypeRoot.DECIMAL);
        typesMap.put("boolean", LogicalTypeRoot.BOOLEAN);
        typesMap.put("char", LogicalTypeRoot.CHAR);
        typesMap.put("date", LogicalTypeRoot.DATE);
        typesMap.put("datetime", LogicalTypeRoot.TIMESTAMP_WITHOUT_TIME_ZONE);
        typesMap.put("decimal", LogicalTypeRoot.DECIMAL);
        typesMap.put("double", LogicalTypeRoot.DOUBLE);
        typesMap.put("float", LogicalTypeRoot.FLOAT);
        typesMap.put("int", LogicalTypeRoot.INTEGER);
        typesMap.put("tinyint", LogicalTypeRoot.TINYINT);
        typesMap.put("smallint", LogicalTypeRoot.SMALLINT);
        typesMap.put("varchar", LogicalTypeRoot.VARCHAR);
        typesMap.put("bitmap", LogicalTypeRoot.VARCHAR);
```

缓存flush逻辑

1. flink会调用 invoke方法传入 RowData，依据以下转换规则序列化为json数据并 addToBatch

Java

```java
private Object typeConvertion(LogicalType type, RowData record, int pos) {
        switch (type.getTypeRoot()) {
            case BOOLEAN:
                return record.getBoolean(pos);
            case TINYINT:
                return record.getByte(pos);
            case SMALLINT:
                return record.getShort(pos);
            case INTEGER:
                return record.getInt(pos);
            case BIGINT:
                return record.getLong(pos);
            case FLOAT:
                return record.getFloat(pos);
            case DOUBLE:
                return record.getDouble(pos);
            case CHAR:
            case VARCHAR:
                return record.getString(pos).toString();
            case DATE:
                return
dateFormatter.format(Date.valueOf(LocalDate.ofEpochDay(record.getInt(pos))).toSt
ring());
            case TIMESTAMP_WITHOUT_TIME_ZONE:
                final int timestampPrecision = ((TimestampType)
type).getPrecision();
                return dateTimeFormatter.format(new
Date(record.getTimestamp(pos, timestampPrecision).toTimestamp().getTime()));
            case DECIMAL: // for both largeint and decimal
                final int decimalPrecision = ((DecimalType)
type).getPrecision();
                final int decimalScale = ((DecimalType) type).getScale();
                return record.getDecimal(pos, decimalPrecision,
decimalScale).toBigDecimal();
            default:
                throw new UnsupportedOperationException("Unsupported type:" +
type);
        }
    }
```

2. flush条件

`if sink.semantic = at-least-once:`

(sink.buffer-flush.interval-ms > 0 && now - lastFlushTime > sink.buffer-flush.interval-ms)

or

(batch.size() >= sink.buffer-flush.max-rows)

or

(calculateBatchBytes(batch) >= sink.buffer-flush.max-bytes)

`if sink.semantic = exactly-once:`

snapshotState() {

flush();

}

3. flush 过程：

tryToFlush() && retry <= sink.max-retries

调用stream load 以json格式put数据并将 sink.properties.* split trim 后拼入请求中。

`if retry > sink.max-retries || stream load failed(was not caused by duplicate key):`

Sink exits with exception.

`If succeeded:`

reset flushTimer

数据容错保证

`at-least-once:`

flink会调用 snapshotState

Java

```
snapshotState {
    flush();
}
```

`exactly-once:`

flink会调用 snapshotState

Java

```
snapshotState {
    // 方法只需复写并在其中阻塞 flush 数据即可 并且只在这里进行flush
    flush();
}
```

常见错误：

缺点

- 此方案的 exactly-once 依赖于用户的checkpoint周期，太长的话会导致缓存数据暴增，太短的话可能导致stream load次数过多。 其次如果snapshotstate过程中如果出现请求阻塞则 此算子可能会导致flink报警或异常。
- 当at-least-once容错时，如果用户设置的 max-rows过大，则重复数据可能会很大。
- 由于各个版本的接口不同，跟客户沟通后基于flink >= 0.11.0 的接口进行开发

测试要点

- 大量数据缓存的flush测试
- exactly-once与at-least-once的数据容错测试
- ~~flush失败时的临时分区删除~~