使用Byzer-lang访问JDBC数据源

Hello world

Byzer-lang 使用 JDBC 数据源非常简单。目前Byzer-lang内置了 MySQL 的驱动，所以可以直接使用如下代码访问 MySQL:

JavaScript

```
connect jdbc where
 url="jdbc:mysql://127.0.0.1:3306/wow?characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&tinyInt1isBit=false"
 and driver="com.mysql.jdbc.Driver"
 and user="xxxx"
 and password="xxxxxx"
 as mysql_instance;

load jdbc.`mysql_instance.test1` as test1;
select * from test1 as output;
```

> 如果你的用户名或者密码包含了一些特殊字符，请使用 `#[[]]#` 括起来

实际上，第一句话的所有参数都可以放到 load语法里，此时代码会变成下面这个样子。

Python

```
load jdbc.`wow.test1`
where url="jdbc:mysql://127.0.0.1:3306/wow?characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&tinyInt1isBit=false"
and driver="com.mysql.jdbc.Driver"
and user="xxxx"
and password="xxxxxx"
as test1;

select * from test1 as output;
```

但是考虑到我们需要加载很多表，如果每张表都配置如此的多参数，会非常繁琐，所以我们引入connect语法，将公共的参数抽取出来，之后使用别名引用。

保存数据到MySQL中也非常简单：

Ruby

```
connect jdbc where
 url="jdbc:mysql://127.0.0.1:3306/wow?characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&tinyInt1isBit=false"
 and driver="com.mysql.jdbc.Driver"
 and user="xxxx"
 and password="xxxxxx"
 as mysql_instance;


load Rest.`https://cnodejs.org/api/v1/topics` where
 `config.connect-timeout`="10s"
 and `config.method`="get"
 -- The following `form` is the request line parameter, which supports setting dynamic rendering parameters
 and `form.page`="1"
 and `form.tab`="share"
 and `form.limit`="2"
 and `form.mdrender`="false"
as cnodejs_articles;

select string(content) as content, status from cnodejs_articles
as topics_table;

save overwrite c as jdbc.`mysql_instance.test1`;
```

并发读取

假设你的表有可以分区的字段，比如有自增 id, 那么我们就可以并发读取。

示例如下：

SQL

```
connect jdbc where
 url="jdbc:mysql://127.0.0.1:3306/wow?characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&tinyInt1isBit=false&useCursorFetch=false"
 and driver="com.mysql.jdbc.Driver"
 and user="xxxx"
 and password="xxxxxx"
 as mysql_instance;

load jdbc.`mysql_instance.test1` where directQuery='''
select max(id) from test1
''' as maxIdTable;

set maxId=`select * from maxIdTable ` where type="sql" and mode="runtime";

load jdbc.`mysql_instance.test1` where
partitionColumn="id"
and lowerBound="0"
and upperBound ="${maxId}"
and numPartitions="8"
and fetchsize="-2147483648"
as test1;

select * from test1 as output;
```

在这个代码中， 我们先通过 connect 语法得到了一个实例引用。 其中配置了 `useCursorFetch=false` 避免读取太慢。 同时通过 directQuery 获取 最大的 id 值，然后将其作为 upperBound。

如果你希望写入到 JDBC 更快，可以增加参数 `batchsize` 来控制写入的速速度。

并发读取原理

首先，大家先思考一个问题，如果写一个程序，读取 MySQL 的数据，你怎么写?
典型的如下：

Kotlin

```kotlin
val driver = options("driver")
    val url = options("url")
    Class.forName(driver)
    val connection = java.sql.DriverManager.getConnection(url, options("user"), options("password"))
    val stat = connection.prepareStatement(sql)
    val rs = stat.executeQuery()
    while(rs.next){
      val row = rs.next()
    }
    rs.close
    connection.close
```

对于一个几千万甚至上亿条数据的大表，遍历一遍需要的时间可能会很久。 这里问题的根源是，只有一个线程去读，从头读到尾，那肯定很慢。

如果想加快速度，大家自然会想到多线程，如果用多线程大家应该怎么读呢？多线程无非就是分而治之，每个线程读一部分数据。比如假设有 id 为0到100的数据，id 不连续，并且实际上有50条记录，如果我们希望分而治之，那么最好是五个线程，每个线程读取10条数据。但是肯定是做不到这么精确的。最简单的办法是，产生五条如下的SQL，

SQL

```sql
select * from xxx where id<20;
select * from xxx where id>=20 and id <40;
select * from xxx where id>=40 and id <60;
select * from xxx where id>=60 and id <80;
select * from xxx where id>=80;
```

因为id中间有空隙，所以每条SQL实际拿到的数据并不一样。但没关系，通过五个线程执行这五条SQL，我们肯定可以通过更少的时间获取到全量数据。

现在回过头来，我们想想 Byzer-lang，Byzer-lang 快很大一部分原因其实是分布式多线程的执行方式，这样可以充分利用多核算力。但是面对一个JDBC接口，Byzer-lang 并不知道如何并行的去拉取一个表的数据，所以就傻傻的用一个线程去拉取数据。

那如果我们想让数据拉取变快，最直接的做法就是让 Byzer-lang 并行化拉取数据。这个时候你需要提供一个类似我们前面提到的，可以切分查询的字段给 Byzer-lang。那么怎么给 Byzer-lang 呢？可以通过下面四个字段来控制：

1. partitionColumn 按哪个列进行分区
2. lowerBound, upperBound, 分区字段的最小值，最大值（可以使用 `directQuery` 获取）
3. numPartitions 分区数目。一般8个线程比较合适。

首先，你得告诉我，哪个字段是可以切分的，你可以通过partitionColumn告诉我。接着该怎么切分呢，如果像上面这样一组一组的告诉我，比如这样（...,20）,[20,40),[40,60),[60,80)....[80,...) 。少还可以，如果面对上百组的情况，你肯定会疯掉。而且这样也不好后续做修改，比如我想搞多点搞少点都会修改的很麻烦。所以权衡后，最后给的设计是这样的，你告诉第一组的最大值，最后一组的最小值，然后告诉我到底要几组，就可以了。比如上面的例子，第一组的最小值是20,最后一组的最大值是80,然后总共5组，这样 Byzer-lang 就知道20-80之间还要再分三组。

所以此时：

4. lowerBound=20
5. upperBound=80
6. numPartitions=5

系统会自动产生（...,20）,[20,40)....[80,...) 这样的序列，然后每个小组产生一条SQL。这样就可以并行拉去数据了。

因为我们希望尽可能的根据这个切分，能切分均匀，所以最好的字段肯定是自增字段。

并发分区字段的选择

理解了上面的问题之后，大家普遍还会遇到三个疑问：

第一个，那如果我没有自增字段该怎么办呢？甚至没有数字字段。能不能用比如oracle的虚拟字段rownum,或者利用mysql的虚拟行号字段？ 其实是可以的，但是可能会对数据源产生比较大的压力，比如MySQL如果使用虚拟行号，会产生巨大的临时表。

第二个，是只能数字字段么？日期行不行。答案是可以。目前分区字段支持的类型有三种：

1. 数字类型(统一会转换为long类型)
2. DateType， 对应的文本格式为yyyy-MM-dd
3. TimestampType 对应的文本格式为 yyyy-MM-dd HH:mm:ss

第三个是，我该如何知道lowerBound,upperBound的值呢。其实很简单，你获取这个字段的min/max值即可。 如果你的数据新增量不大，你不用担心最后[max,) 这个分区的数据太多。

关于MySQL 驱动 OOM问题

当 Driver 为 MySQL 时，可以将 fetchSize 设置为 -2147483648（在spark2中不支持设置fetchsize为负数，默认值为1000，此时可以在Url中设置useCursorFetch=true）来拉取数据，避免全量加载导致OOM。

在 2.2.0版本的 Byzer-lang 中，如果 url 参数没有设置 useCursorFetch ， 那么系统会自动将 useCursorFetch=true。 这可能导致数据获取缓慢，用户可以显式的去设置为 false.

DirectQuery 介绍

Byzer-lang 还提供一个方式可以直接把用户的查询提交给对应的JDBC数据源。我们称之为DirectQuery。比如上面的例子：

Python

```
load jdbc.`mysql_instance.test1` where directQuery='''
select * from test1 limit 10
''' as newtable;

select * from newtable as output;
```

我们在load语句里新增了一个参数，叫directQuery。系统会首先把这个语句直接发送给底层数据源，然后获得数据源的10条数据，接着把这个数据重新映射成表newtable. DirectQuery模式请务必确保数据集不要太大，否则可能会引起引擎的OOM。

那在什么场景我们会用到DirectQuery呢?

1. 聚合查询，并且结果较小。
2. 只是为了快速的查看一些数据，并且数据集较小
3. 还有就是其实底层数据源支持的不是SQL，比如ElasticSearch是json格式的查询。此时使用他也是比较好的一种方式，避免全量拉取数据源数据做计算。

JDBC数据源DDL执行

DirectQuery 仅能支持select查询语句。如果你需要对数据源做一些DDL,那么可以使用ET JDBC . 使用如下语法：

JavaScript

```
run command as JDBC.`db_1._` where
`driver-statement-0`="drop table test1"
and `driver-statement-1`="create table test1.....";

save append tmp_article_table as jdbc.`db_1.test1`;
```

该ET本质上是在Driver端通过JDBC驱动执行各种操作指令。 参数最后的-0, -1 表示的是执行顺序。

MySQL Upsert 语义支持

要让Byzer-lang在保存数据时执行Upsert语义的话，你只需要提供提供idCol字段即可。下面是一个简单的例子：

SQL

```
save append tmp_article_table as jdbc.`db_1.test1`
where idCol="a,b,c";
```

Byzer-lang内部使用了MySQL的duplicate key语法，所以用户需要对应的数据库表确实有重复联合主键的约束。那如果没有实现在数据库层面定义联合约束主键呢？ 结果会是数据不断增加，而没有执行update操作。

idCol的作用有两个，一个是标记，标记数据需要执行Upsert操作，第二个是确定需要的更新字段，因为主键自身的字段是不需要更新的。MLSQL会将表所有的字段减去 idCol定义的字段，得到需要更新的字段。

流式计算中，如何使用JDBC

SQL

```
set streamName="mysql-test";

.......

save append table21
as streamJDBC.`mysql1.test1`
options mode="Complete"
and `driver-statement-0`="create table  if not exists test1(k TEXT,c BIGINT)"
and `statement-0`="insert into wow.test1(k,c) values(?,?)"
and duration="3"
and checkpointLocation="/tmp/cpl3";
```

我们提供了一个叫streamJDBC的数据源。`driver-statement-0` 类型的参数可以做一些工作比如提前创建表等，该代码只会执行一次。statement-0 则可以将每条记录转化为一条insert 语句。

注意1，该streamJDBC 无法保证exactly once语义。

注意2，Insert语句中的占位符顺序需要和table21中的列顺序保持一致。

如何新增数据源

Byzer-lang语言引擎默认集成了 MySQL 的支持，使用 `mysql-connector-java-5.1.46` 版本。但用户可能需要支持各种JDBC数据源，而且即使同一数据源，也可能版本各不相同，驱动也各不相同。

那么如何使用新的JDBC数据源呢？只需做两件事情：

1. 下载对应的JDBC驱动
2. 将驱动放到指定目录

分布式版本

将驱动放到 SPARK-HOME/jars 目录即可。比如 spark-3.1.1-bin-hadoop3.2/jars

桌面版

一般是在 ~/.vscode/extensions/mlsql~/libs 目录

Byzer-lang SDK

将驱动放置于Byzer-HOME/libs 目录下即可。

对于限制了prepared statement的JDBC协议

部分系统因为权限问题，限制了 prepared statement。此时可以使用Java/Scala创建ET来进行数据导出。如果只支持Python,那么可以使用Byzer-lang 对Python的支持来获取数据并且导出到数据湖或者特定目录，方便后续处理。

一些常见参数

	A	B
1	Property Name	Meaning
2	url	The JDBC URL to connect to. The source-specific connection properties may be specified in the URL. e.g., jdbc:postgresql://localhost/test?user=fred&password=secret
3	dbtable	The JDBC table that should be read. Note that anything that is valid in a FROM clause of a SQL query can be used. For example, instead of a full table you could also use a subquery in parentheses.
4	driver	The class name of the JDBC driver to use to connect to this URL.
5	partitionColumn, lowerBound, upperBound	These options must all be specified if any of them is specified. In addition, numPartitions must be specified. They describe how to partition the table when reading in parallel from multiple workers. partitionColumn must be a numeric column from the table in question. Notice that lowerBound and upperBound are just used to decide the partition stride, not for filtering the rows in table. So all rows in the table will be partitioned and returned. This option applies only to reading.
6	numPartitions	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling coalesce(numPartitions) before writing.
		The JDBC fetch size, which determines how many rows to fetch per round trip

7	fetchsize	The JDBC fetch size, which determines how many rows to fetch per round trip. This can help performance on JDBC drivers which default to low fetch size (eg. Oracle with 10 rows). This option applies only to reading. When the data source of jdbc is mysql, It defaults to -2147483648 (1000 for previous versions of spark by default, we support row-by-row data loading by default.
8	batchsize	The JDBC batch size, which determines how many rows to insert per round trip This can help performance on JDBC drivers. This option applies only to writing defaults to 1000.
9	isolationLevel	The transaction isolation level, which applies to current connection. It can be one of NONE, READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, o SERIALIZABLE, corresponding to standard transaction isolation levels defined b JDBC's Connection object, with default of READ_UNCOMMITTED. This option applies only to writing. Please refer the documentation in java.sql.Connection.
10	sessionInitStatement	After each database session is opened to the remote DB and before starting to read data, this option executes a custom SQL statement (or a PL/SQL block). Us this to implement session initialization code. Example: option("sessionInitStatement", """BEGIN execute immediate 'alter session set "_serial_direct_read"=true'; END;""")
11	truncate	This is a JDBC writer related option. When SaveMode.Overwrite is enabled, this option causes Spark to truncate an existing table instead of dropping and recreating it. This can be more efficient, and prevents the table metadata (e.g., indices) from being removed. However, it will not work in some cases, such as when the new data has a different schema. It defaults to false. This option applies only to writing.
12	createTableOptions	This is a JDBC writer related option. If specified, this option allows setting of database-specific table and partition options when creating a table (e.g., CREA TABLE t (name string) ENGINE=InnoDB.). This option applies only to writing.
13	createTableColumnTypes	The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g: "name CHAR(64), comments VARCHAR(1024)"). The specified types should be valid spark sql data types. Thi option applies only to writing.
14	customSchema	The custom schema to use for reading data from JDBC connectors. For exampl "id DECIMAL(38, 0), name STRING". You can also specify partial fields, and the others use the default type mapping. For example, "id DECIMAL(38, 0)". The column names should be identical to the corresponding column names of JDB table. Users can specify the corresponding data types of Spark SQL instead of