Apache Calcite

Apache Calcite是一个动态数据管理框架，为不同计算平台和数据源提供统一的查询引擎，并以类似传统数据库的访问方式（SQL和高级查询优化）来访问数据。具有以下几个技术特性：

* 支持标准SQL语言
* 独立于编程语言和数据源，可以支持不同的前端和后端
* 支持关系代数，可定制逻辑规划规则和基于成本模型优化的查询引擎
* 支持物化视图（materialized view）的管理（创建、丢弃、持久化和自动识别）
* 基于物化视图的Lattice和Tile机制，以应用于OLAP分析
* 支持对流数据的查询

在具有这些特性的同时，也舍弃了一些功能，比如数据存储、处理数据的算法和元数据仓库，由于舍弃这些功能，Calcite可以在应用和数据存储、数据处理引擎之间作为中介的角色。

Apache Calcite是基于关系代数的查询引擎，关系代数支持并、差、笛卡尔积、投影和选择等基本运算。关系代数式Calcite的核心，任何一个查询都可以表示成由关系运算符u组成的树。比如下面的SQL查询：

*SELECT deptno, count(\*) AS c, sum(sal) AS s*

*FROM emp*

*GROUP BY deptno*

*HAVING count(\*) > 10*

可以表达成如下的关系表达式语法树：

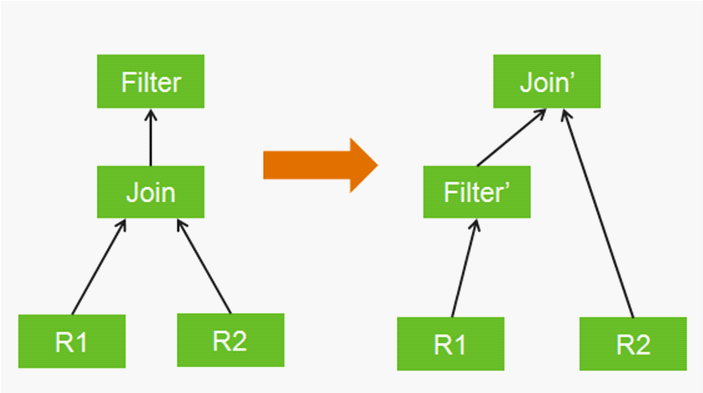
*LogicalFilter(condition=[>($1, 10)])*

*LogicalAggregate(group=[{7}], C=[COUNT()], S=[SUM($5)])*

*LogicalTableScan(table=[[scott, EMP]])*

当上层编程语言，如SQL转换为关系表达式后，就会送到Calcite的逻辑规划器进行规则匹配。在这个过程中，Calcite查询会循环使用规划规则对关系表达式语法树的节点和子图进行优化。这种优化过程会以一个成本模型进行参数，每次优化都在保证语义的情况下利用规则来降低成本，成本主要以查询时间最快、资源消耗最少的维度去度量。

使用逻辑规划规则等同于数学恒等式变换，比如讲一个过滤器推到内连接（inner join）输入的内部执行，当然使用这个规则的嵌套是过滤器不会引用内连接输入之外的数据列。下图是将Filter操作下推到Join的示例，这么做的好处是减少Join操作记录的数量



Calcite的查询引擎是可以定制和扩展的，自定义关系运算符、规划规则、成本模型和相关的统计，从而应用到不同需求的场景。

Apache Calcite使用基于关系代数的查询引擎，重点在关系代数的语法分析和查询逻辑的规划制定上，不受上层编程语言的限制。前端可以使用SQL、Pig、Cascading或者Scalding等，只要通过Calcite提供的API将它们转换成关系代数的抽象语法树即可。Calcite不涉及到物理规划层，可以扩展适配器来连接多种后端的数据源和处理引擎，如Spark,Splunk,

MongoDB等。目前它已经或被规划集成到Hadoop的诸多项目中，比如Lingual (Cascading项目的SQL接口)、Apache Drill、Apache Hive、Apache Kylin、Apache Phoenix、Apache Samza和Apache Flink。

# 1.Apache Calcite特性

## 1.1 物化视图的应用

Calcite物化视图是从传统的关系型数据库系统借鉴而来，包含一个SQL查询和这个查询所生成的数据表，下面是一个物化视图的例子，按部门、性能统计相应的员工数量和工资总额：

*CREATE MATERIALIZED VIEW emp\_summary AS*

*SELECT deptno, gender, COUNT(\*) AS c, SUM(salary) AS s*

*FROM emp*

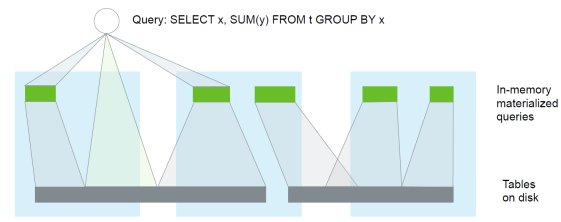
*GROUP BY deptno, gender;*

物化视图本质上也是一个数据表，可以直接查询，比如查询男员工人数大于20的部门

*SELECT deptno FROM emp\_summary*

*WHERE gender = ‘M’ AND c > 20;*

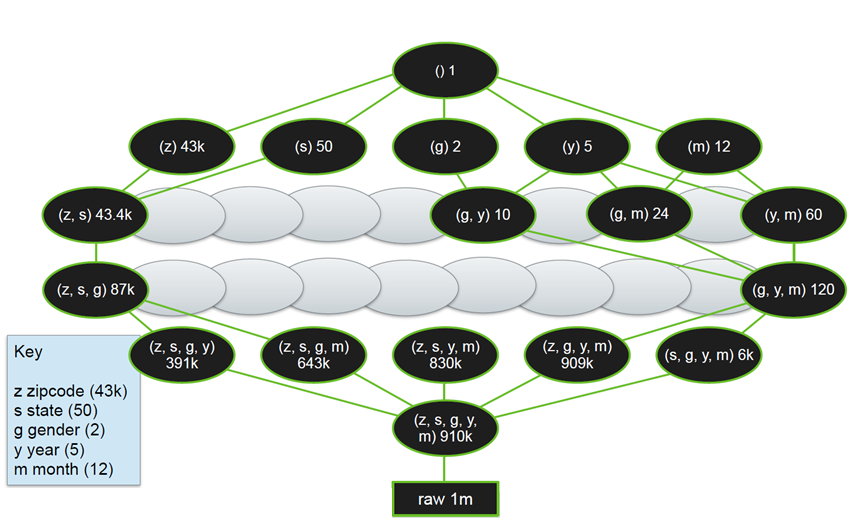
还可以通过物化视图的查询取代对相关数据表的查询，由于物化视图一般存储在内存中，而且数据更接近最终结果，所以查询速度会比较快：



## 1.2 在线分析处理

为了加速在线分析处理，除了物化视图，Calcite还引入[Lattice（格子）和Tile（瓷片）](http://www.slideshare.net/julianhyde/calcite-stratany2014" \t "_blank)的概念。Lattice可以看做是在[星模式（star schema）](https://en.wikipedia.org/wiki/Star_schema" \t "_blank)数 据模型下对物化视图的推荐、创建和识别的机制。这种推荐可以根据查询的频次统计，也可以基于某些分析维度的重要等级。Tile则是Lattice中的一个 逻辑的物化视图，它可以通过三种方法来实体化：1）在lattice中声明；2）通过推荐算法实现；3）在响应查询时创建。

下图是Lattice和Tile的一个图例，这个OLAP分析涉及五个维度的数据：邮政编码、州、性别、年和月。每个椭圆代表一个Tile，黑色椭圆是实体化后物化视图，椭圆中的数字代表该物化视图对应的记录数



由于Calcite可以很好地支持物化视图和星模式这些OLAP分析的关键特性，所以Apache基金会的[Kylin项目](http://kylin.incubator.apache.org/" \t "_blank)（Hadoop上OLAP系统）在选用查询引擎时就直接集成了Calcite。

## 1.3 支持流查询

Calcite对其SQL和关系代数进行了扩展以支持[流查询](https://calcite.incubator.apache.org/docs/stream.html" \t "_blank)。Calcite的SQL语言是标准SQL的扩展，而不是类SQL（SQL-like），这个差别非常重要，因为：

* 如果你懂标准SQL，那么流的SQL也会非常容易学；
* 因为在流和表上使用相同的机制，语义会很清楚；
* 你可以写同时对流和表结合的查询语句；
* 很多工具可以直接生成标准的SQL。

Calcite的流查询除了支持排序、聚合、过滤等常用操作和子查询外，也支持各种窗口操作，比如翻滚窗口（Tumbling window）、跳跃窗口（Hopping window）[、](https://calcite.incubator.apache.org/docs/stream.html" \l "hopping-windows" \t "_blank)滑动窗口（Sliding windows）、级联窗口（Cascading window）。其中级联窗口可以看作是滑动窗口和翻滚窗口的结合。

# 2. Apache Calcite教程-CSV

Calcite-example-CSV是从CSV文件中读取数据的适配器，在Example中提供了SQL查询包含的所有能力，CSV为其他格式的数据提供了adapter构建的模板。其重要概念如下：

* 采用SchemaFactory和Schema接口自定义Schema
* 在模型JSON文件中声明schema
* 在模型JSON文件中声明视图
* 使用Table接口自定义表
* 记录表的记录类型
* 采用SchannableTable接口提供一个简单的Table实现，直接枚举出表的每一行
* 采用FilteredTable接口提供一个较高级的Table实现，针对简单的谓词实现行的过滤
* 采用TranslateableTable接口实现更高阶的Table，可以使用计划规则来翻译关系操作符

## 2.1 CSV Example安装和使用

CSV执行如下所示：

1. 下载源码并编译

*$ git clone https://github.com/apache/calcite.git*

*$ cd calcite*

*$ mvn install -DskipTests -Dcheckstyle.skip=true*

*$ cd example/csv*

1. 使用sqlline，SQL Shell工具

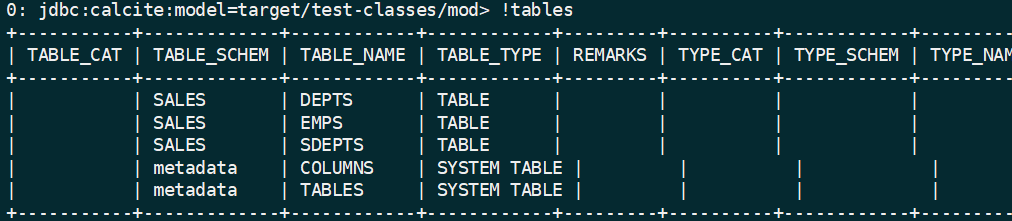
*$ ./sqlline*

*sqlline> !connect jdbc:calcite:model=target/test-classes/model.json admin admin*

进入CSV数据库

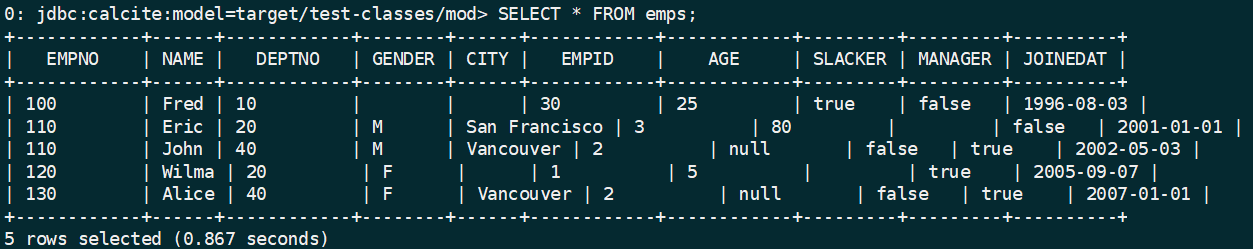
*0: jdbc:calcite:model=target/test-classes/mod>*

1. 执行metadata查询

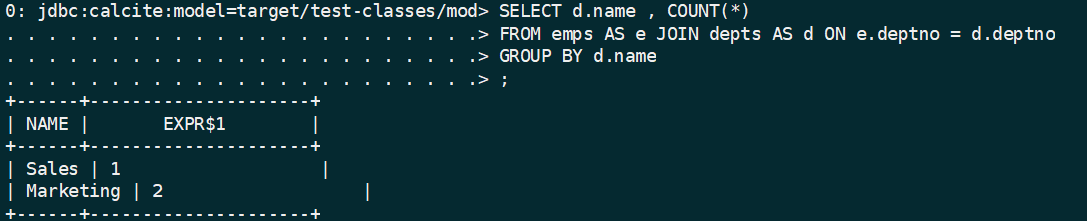


!tables命令是执行DatabaseMetaData.getTables()方法。还有其他的命令来查询metadata，例如：!columns和!describe。在上面的例子上有5个系统表：在SALES schema中有EMPS、DEPTS及HOBBIES三个表。EMPS和DEPTS表分别基于target/test-classes目录下的EMPS.csv和DEPTS.csv文件。

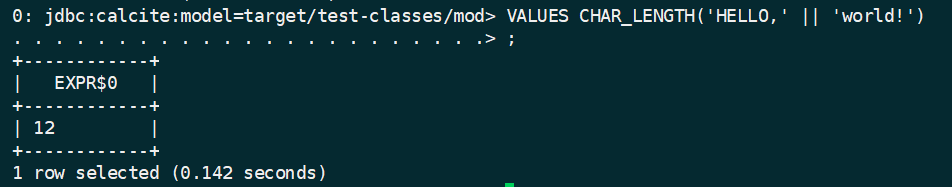
4）在执行这些表上进行查询



执行join及group by操作：



同时定义一些operator，例如VALUES operator生成一个单行row



## 2.2 Schema discovery

Calcite不进行数据存储，通过执行命令来指定这些tables，如上面所述的命令：

*!connect jdbc:calcite:model=target/test-classes/model.json admin admin*

实现Calcite的CSV Schema要有以下几个步骤：

1. 基于Schema Factory类定义Schema，在JDBC Connection String中指定该Schema文件的路径。该文件的格式：

*{*

*version: '1.0',*

*defaultSchema: 'SALES',*

*schemas: [*

*{*

*name: 'SALES',*

*type: 'custom',*

*factory: 'org.apache.calcite.adapter.csv.CsvSchemaFactory',*

*operand: {*

*directory: 'target/test-classes/sales'*

*}*

*}*

*]}*

该模型中定义了SALES的Schema，使用CsvSchemaFactory来实例化这个Schema，该类实现了ShemaFactory接口，在Model文件中设置目录参数。CsvSchemaFactory#create方法如下：

*public Schema create(SchemaPlus parentSchema, String name,*

*Map<String, Object> operand) {*

*String directory = (String) operand.get("directory");*

*String flavorName = (String) operand.get("flavor");*

*CsvTable.Flavor flavor;*

*if (flavorName == null) {*

*flavor = CsvTable.Flavor.SCANNABLE;*

*} else {*

*flavor = CsvTable.Flavor.valueOf(flavorName.toUpperCase());*

*}*

*return new CsvSchema(*

*new File(directory),*

*flavor);*

*}*

1. 该Factory创建Schema，其中包含了几个表，并且同时定义表的数据源

CsvSchema实现Schema接口，生成tables的列表，同时列举sub-schemas和table-functions，还有很多高级特性，但是CSV不支持。这些Table实现Calcite的表接口，CsvSchema生成的Table实现CsvTable接口。

CsvSchema覆盖getTableMap方法，源码如下：

*protected Map<String, Table> getTableMap() {*

*File[] files = directoryFile.listFiles(*

*new FilenameFilter() {*

*public boolean accept(File dir, String name) {*

*final String nameSansGz = trim(name, ".gz");*

*return nameSansGz.endsWith(".csv")*

*|| nameSansGz.endsWith(".json");*

*}*

*});*

*if (files == null) {*

*System.out.println("directory " + directoryFile + " not found");*

*files = new File[0];*

*}*

*// Build a map from table name to table; each file becomes a table.*

*final ImmutableMap.Builder<String, Table> builder = ImmutableMap.builder();*

*for (File file : files) {*

*String tableName = trim(file.getName(), ".gz");*

*final String tableNameSansJson = trimOrNull(tableName, ".json");*

*if (tableNameSansJson != null) {*

*JsonTable table = new JsonTable(file);*

*builder.put(tableNameSansJson, table);*

*continue;*

*}*

*tableName = trim(tableName, ".csv");*

*final Table table = createTable(file);*

*builder.put(tableName, table);*

*}*

*return builder.build();*

*}*

*private Table createTable(File file) {*

*switch (flavor) {*

*case TRANSLATABLE:*

*return new CsvTranslatableTable(file, null);*

*case SCANNABLE:*

*return new CsvScannableTable(file, null);*

*case FILTERABLE:*

*return new CsvFilterableTable(file, null);*

*default:*

*throw new AssertionError("Unknown flavor " + flavor);*

*}*

*}*

Schema遍历这些目录，获取后缀为.csv的所有文件，并且创建这些表。在该模型中目录配置为target/test-classes/sales，包括的文件为EMPS.csv、DEPTS.csv。

### 2.2.1 View

在Schema中定义个View，如下所示：

*{*

*version: '1.0',*

*defaultSchema: 'SALES',*

*schemas: [*

*{*

*name: 'SALES',*

*type: 'custom',*

*factory: 'org.apache.calcite.adapter.csv.CsvSchemaFactory',*

*operand: {*

*directory: 'target/test-classes/sales'*

*},*

*tables: [*

*{*

*name: 'FEMALE\_EMPS',*

*type: 'view',*

*sql: 'SELECT \* FROM emps WHERE gender = \'F\''*

*}*

*]*

*}*

*]*

*}*

在示例中type:’view’的标签名为FEMAL\_EMPS，

*{*

*name: 'FEMALE\_EMPS',*

*type: 'view',*

*sql: 'SELECT \* FROM emps WHERE gender = \'F\''*

*}*

在View上可以执行的查询如下：

*sqlline> SELECT e.name, d.name FROM female\_emps AS e JOIN depts AS d on e.deptno = d.deptno;*

*+--------+------------+*

*| NAME | NAME |*

*+--------+------------+*

*| Wilma | Marketing |*

*+--------+------------+*

### 2.2.2 Custom Tables

在示例中定义在model-with-custom-table.json

*{*

*version: '1.0',*

*defaultSchema: 'CUSTOM\_TABLE',*

*schemas: [*

*{*

*name: 'CUSTOM\_TABLE',*

*tables: [*

*{*

*name: 'EMPS',*

*type: 'custom',*

*factory: 'org.apache.calcite.adapter.csv.CsvTableFactory',*

*operand: {*

*file: 'target/test-classes/sales/EMPS.csv.gz',*

*flavor: "scannable"*

*}*

*}*

*]*

*}*

*]*

*}*

可以在表上执行下面的查询

*sqlline> !connect jdbc:calcite:model=target/test-classes/model-with-custom-table.json admin admin*

*sqlline> SELECT empno, name FROM custom\_table.emps;*

*+--------+--------+*

*| EMPNO | NAME |*

*+--------+--------+*

*| 100 | Fred |*

*| 110 | Eric |*

*| 110 | John |*

*| 120 | Wilma |*

*| 130 | Alice |*

*+--------+--------+*

该表有CsvTableFactory来生成，实现的接口为TableFactory，实例化一个CsvScannableTable，使用model文件中的文件参数，生成表如下：

*public CsvTable create(SchemaPlus schema, String name,*

*Map<String, Object> map, RelDataType rowType) {*

*String fileName = (String) map.get("file");*

*final File file = new File(fileName);*

*final RelProtoDataType protoRowType =*

*rowType != null ? RelDataTypeImpl.proto(rowType) : null;*

*return new CsvScannableTable(file, protoRowType);*

*}*

### 2.2.3 使用planner rules来优化查询

当表中有大量数据时，需要和adpater协作来进行数据访问的优化。Calcite支持添加Planner rules来优化查询。Planner Rules在query parse tree上进行操作，根据规则替换相对应的节点。下面的示例是使用优化前后的对比：

*sqlline> !connect jdbc:calcite:model=target/test-classes/model.json admin admin*

*sqlline> explain plan for select name from emps;*

*+-----------------------------------------------------+*

*| PLAN |*

*+-----------------------------------------------------+*

*| EnumerableCalcRel(expr#0..9=[{inputs}], NAME=[$t1]) |*

*| EnumerableTableScan(table=[[SALES, EMPS]]) |*

*+-----------------------------------------------------+*

*sqlline> !connect jdbc:calcite:model=target/test-classes/smart.json admin admin*

*sqlline> explain plan for select name from emps;*

*+-----------------------------------------------------+*

*| PLAN |*

*+-----------------------------------------------------+*

*| EnumerableCalcRel(expr#0..9=[{inputs}], NAME=[$t1]) |*

*| CsvTableScan(table=[[SALES, EMPS]]) |*

*+-----------------------------------------------------+*

例子上使用smart.json模型文件进行了优化，在createTable中生成CsvTranslatableTable而不是CsvScannableTable。CsvTranslatableTable重载了TranslatableTable.toRel方法来创建CsvTableScan，替换了原EnumerableTableScan，进行了一定化后。规则的定义如下：

*public class CsvProjectTableScanRule extends RelOptRule {*

*public static final CsvProjectTableScanRule INSTANCE =*

*new CsvProjectTableScanRule();*

*private CsvProjectTableScanRule() {*

*super(*

*operand(Project.class,*

*operand(CsvTableScan.class, none())),*

*"CsvProjectTableScanRule");*

*}*

*@Override*

*public void onMatch(RelOptRuleCall call) {*

*final Project project = call.rel(0);*

*final CsvTableScan scan = call.rel(1);*

*int[] fields = getProjectFields(project.getProjects());*

*if (fields == null) {*

*// Project contains expressions more complex than just field references.*

*return;*

*}*

*call.transformTo(*

*new CsvTableScan(*

*scan.getCluster(),*

*scan.getTable(),*

*scan.csvTable,*

*fields));*

*}*

*private int[] getProjectFields(List<RexNode> exps) {*

*final int[] fields = new int[exps.size()];*

*for (int i = 0; i < exps.size(); i++) {*

*final RexNode exp = exps.get(i);*

*if (exp instanceof RexInputRef) {*

*fields[i] = ((RexInputRef) exp).getIndex();*

*} else {*

*return null; // not a simple projection*

*}*

*}*

*return fields;*

*}*

*}*

### 2.2.4 JDBC adapter

JDBC adapter将JDC Data Source映射为Calcite Shema，示例如下：

*{*

*version: '1.0',*

*defaultSchema: 'FOODMART',*

*schemas: [*

*{*

*name: 'FOODMART',*

*type: 'custom',*

*factory: 'org.apache.calcite.adapter.jdbc.JdbcSchema$Factory',*

*operand: {*

*jdbcDriver: 'com.mysql.jdbc.Driver',*

*jdbcUrl: 'jdbc:mysql://localhost/foodmart',*

*jdbcUser: 'foodmart',*

*jdbcPassword: 'foodmart'*

*}*

*}*

*]*

*}*

1. Calcite解析查询语句，并且生成对表的查询plan
2. Calcite调用这些表，执行查询获取数据

# 3. Apache Calcite -CSV源码分析

Calcite通过JDBC执行数据的查询，其是应用程序和数据存储之间的数据处理引擎。Calcite并不管理数据，也没有标准的数据格式，访问数据需要编写一个适配器来告诉Calcite，将数据源中某种格式的数据集合视为Table来进行操作。在example/csv子项目下提供CSV适配器来读取CSV格式的文本文件，将一个CSV文件目录以包含Schema信息的table表形式呈现。Csv是配置可以作为其他数据格式的适配器构建参考模板：

1. 用户通过使用SchemaFactory和Schema Interface来自定义Schema
2. 以JSON模型文件声明Schemas
3. 以JSON模型文件声明视图Views
4. 通过Table Interface自定义table
5. 定义table的record类型
6. 使用Scannable Table Interface作为Table的简单实现，来直接枚举所有的rows
7. 进阶实现FilterableTable，来根据简单的谓词predicates过滤rows
8. 以Translate Table进阶实现Table，将关系型算子翻译为执行计划规则

下面以下面的流程来分析CSV SQL的执行：

* 使用SQLline进入查询入口，包括CalciteDriver、Connection等查询相关类
* 分析Model的定义Schema，通过SchemaFactroy创建该Schema
* 该Schema创建Tables，每个Table中定义如何扫描数据
* Calcite解析查询语句SQL并计划如何查询这些Table
* Caclcite通过表去读取数据，以完成查询语句SQL的执行

## 3.1 查询测试入口SQLline

通过工程内置的sqlline脚本来连接到Calcite

*$ ./sqlline*

*sqlline> !connect jdbc:calcite:model=target/test-classes/model.json admin admin*

sqlline从model.json中加载metadata信息。SQLLine是使用Java编写的关系型数据库连接工具，用来执行SQL命令，例如与Oracle的sqlplus，Mysql的mysql等工具。在CSV项目中sqlline的执行如下：

*exec java $VM\_OPTS -cp "${CP}" sqlline.SqlLine "$@"*

通过该命令，生成对应的CalciteDriver、CalciteConnection，在Connection上根据model.json定义的schema实例化CalciteSchema，相关的类图如下所示：



SQLline调用CalciteDriver来连接数据源，并返回数据，其中数据源的连接通过CalciteConectionImpl来实现。在Connection中核心的成员变量为CalciteSchema。

## 3.2 Schema的定义

数据库的核心是Schema，其中描述了表结构，数据来源等信息，在Calcite中Schema的核心类如下所示：



针对每一种数据源都要实现相对的Schema，如MongoSchema、CassandraSchema、DruidSchema、SplunkSchema等，不再一一介绍。

对于Csv来说通过model.json来描述，文件如下所示：

*{*

*version: '1.0',*

*defaultSchema: 'SALES',*

*schemas: [*

*{*

*name: 'SALES',*

*type: 'custom',*

*factory: 'org.apache.calcite.adapter.csv.CsvSchemaFactory',*

*operand: {*

*directory: 'target/test-classes/sales'*

*}*

*}*

*]*

其描述类为JsonSchema，类图如下：



SchemaFactory基于JsonSchema对象创建Schema，由ModuleHandler调用:

*public void visit(JsonCustomSchema jsonSchema) {*

*try {*

*final SchemaPlus parentSchema = currentMutableSchema("sub-schema");*

*checkRequiredAttributes(jsonSchema, "name", "factory");*

*final SchemaFactory schemaFactory =*

*AvaticaUtils.instantiatePlugin(SchemaFactory.class,*

*jsonSchema.factory);*

*final Schema schema =*

*schemaFactory.create(*

*parentSchema, jsonSchema.name, operandMap(jsonSchema, jsonSchema.operand));*

*final SchemaPlus schemaPlus = parentSchema.add(jsonSchema.name, schema);*

*populateSchema(jsonSchema, schemaPlus);*

*} catch (Exception e) {*

*throw new RuntimeException("Error instantiating " + jsonSchema, e);*

*}}*

对于CSV来说SchemaFactory对schema的创建如下：

*public Schema create(SchemaPlus parentSchema, String name,*

*Map<String, Object> operand) {*

*final String directory = (String) operand.get("directory");*

*final File base =*

*(File) operand.get(ModelHandler.ExtraOperand.BASE\_DIRECTORY.camelName);*

*File directoryFile = new File(directory);*

*if (base != null && !directoryFile.isAbsolute()) {*

*directoryFile = new File(base, directory);*

*}*

*String flavorName = (String) operand.get("flavor");*

*CsvTable.Flavor flavor;*

*if (flavorName == null) {*

*flavor = CsvTable.Flavor.SCANNABLE;*

*} else {*

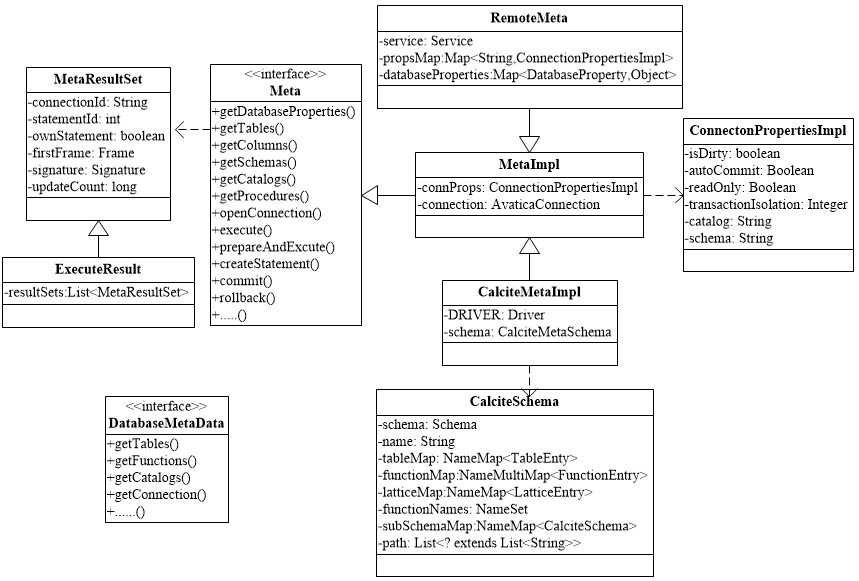
*flavor = CsvTable.Flavor.valueOf(flavorName.toUpperCase(Locale.ROOT));*

*}*

*return new CsvSchema(directoryFile, flavor);*

*}*

在Schema中，使用Meta来保存元数据，其类图如下所示：



在ConnectionPropertiesImpl中使用schema:String来描述定义字符串。

## 3.3 Table的创建

创建好了CsvSchema，由其提供Table信息，Table类型如下所示：



对于Csv来说用到的Table如下：



在SchemaFactory中创建Table如下：

*private Map<String, Table> createTableMap() {*

*final Source baseSource = Sources.of(directoryFile);*

*File[] files = directoryFile.listFiles( //获取表数据文件*

*new FilenameFilter() {*

*public boolean accept(File dir, String name) {*

*final String nameSansGz = trim(name, ".gz");*

*return nameSansGz.endsWith(".csv")*

*|| nameSansGz.endsWith(".json");*

*}*

*});*

*if (files == null) {*

*System.out.println("directory " + directoryFile + " not found");*

*files = new File[0];*

*}*

*// Build a map from table name to table; each file becomes a table.*

*final ImmutableMap.Builder<String, Table> builder = ImmutableMap.builder();*

*for (File file : files) { //根据文件创建对应表*

*Source source = Sources.of(file);*

*Source sourceSansGz = source.trim(".gz");*

*final Source sourceSansJson = sourceSansGz.trimOrNull(".json");*

*if (sourceSansJson != null) {*

*JsonTable table = new JsonTable(source);*

*builder.put(sourceSansJson.relative(baseSource).path(), table);*

*continue;*

*}*

*final Source sourceSansCsv = sourceSansGz.trim(".csv");*

*final Table table = createTable(source);*

*builder.put(sourceSansCsv.relative(baseSource).path(), table);*

*}*

*return builder.build();*

*}*

*/\*\*根据表flavor类型创建不同类型的表 \*/*

*private Table createTable(Source source) {*

*switch (flavor) {*

*case TRANSLATABLE:*

*return new CsvTranslatableTable(source, null);*

*case SCANNABLE:*

*return new CsvScannableTable(source, null);*

*case FILTERABLE:*

*return new CsvFilterableTable(source, null);*

*default:*

*throw new AssertionError("Unknown flavor " + this.flavor);*

*}*

*}*

## 3.4 从Table中读取数据

每个Table都对应一个CSV文件，其中CSVReader用于读取数据并完成数据的解析。不再详述。

example-csv简析-CsvScannableTable

https://blog.csdn.net/hjw199089/article/details/78896947

Calcite概览：

https://blog.csdn.net/junerli/article/details/78286575

https://blog.csdn.net/hfcenter/article/details/53643095

https://blog.csdn.net/yu616568/article/details/49915577

为List封装SQL接口：https://blog.csdn.net/bluejoe2000/article/details/51457086

为CSV封装SQL接口：

https://blog.csdn.net/junerli/article/details/78286575

Adapters:

https://blog.csdn.net/hjw199089/article/details/78765310

https://calcite.apache.org/docs/tutorial.html

<https://blog.csdn.net/hjw199089/article/details/78896947>

参考链接：

https://github.com/apache/calcite/tree/master/example/csv

https://github.com/apache/calcite

https://calcite.apache.org/