初识Calcite——使用实例

Calcite介绍

Calcite（https://calcite.apache.org/）是Apache的一个孵化器项目，它是一个构建JDBC或者ODBC访问数据库的框架，通过自定义一些adapter通过sql访问任意类型的数据，回想起我们之前使用SQL的场景只有使用访问关系数据库如MYSQL、ORACLE等，通过hive查询HDFS上的数据，但是如果我们希望通过SQL接口访问内存中的某个数据结构（首先这个结构有关系模型）、文件里面的内容（例如CSV文件、有一定结构的普通文件，其实这些可以通过hive访问）、访问hbase和一些NOSQL数据库，甚至想要跨数据源访问（hive里面的数据和mysql里面的数据进行join查询）。以上基本上代表了我们平时接触到的各种各样的数据存储的位置，而Calcite要解决的问题就是让你想办法将这些数据建立一个关系模型，然后通过SQL查询这些数据。

假设我们只使用calcite做查询，因为以上的数据基本上都是通过其他方式写入的数据，而我们需要的是通SQL查询，calcite实现了SQL语句的解析，生成物理执行计划以及查询计划的优化，用户需要向Calcite提供数据库的元数据（有哪些database(schema)，每一个数据库下有哪些table，每一个表有哪些字段，每一个字段的类型是什么）和数据（每一个表中的每一行数据是什么）。除此之外，用户也可以重载它提供的执行计划，这里只是提及到了Calcite的一些基本功能，高阶功能诸如Streaming（流式查询）、Lattices（物化视图）等，目前使用Calcite的方式是作为一个本地的框架工具而非作为一个服务存在。 Apache Calcite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、可定制的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。

如何使用

这里有一篇介绍Calcite的文章可以参考：http://www.infoq.com/cn/articles/new-big-data-hadoop-query-engine-apache-calcite下面主要以实践的方式介绍如何使用Calcite查询不同数据源的数据，这里我们的实验的存储是内存中的数据结构，首先我们有一个map：

```
public static final Map<String, Database> MAP = new HashMap<String, Database>();
   public static class Database {
    public List<Table> tables = new LinkedList<Table>();
}


public static class Table{
    public String tableName;
    public List<Column> columns = new LinkedList<Column>();
    public List<List<String>> data = new LinkedList<List<String>>();
}

public static class Column{
    public String name;
    public String type;
}
```

这个MAP对象中存储了数据库名到我们内存中Database结构的映射，每一个Database中存储了多个Table对象，每一个Table对象有一些Column和一个二维的data数组，Column定义了字段名和类型，然后为了测试创建了一个Database对象，名为school，它包含两个Table，分别为Class和Student，Class对象的初始化如下：

```
    cl. tableName = "Class";
    Column name = new Column();
    name.name = "name";
    name.type = "varchar";
    cl.columns.add(name);

    Column id = new Column();
    id.name = "id";
    id.type = "integer";
    cl.columns.add(id);
```

```
    Column teacher = new Column();
    teacher.name = "teacher";
    teacher.type = "varchar";
    cl.columns.add(teacher);
```

Student对象的初始化如下：

```
    student. tableName = "Student";
    Column name = new Column();
    name.name = "name";
    name.type = "varchar";
    student.columns.add(name);

    Column id = new Column();
    id.name = "id";
    id.type = "varchar";
    student.columns.add(id);

    Column classId = new Column();
    classId.name = "classId";
    classId.type = "integer";
    student.columns.add(classId);

    Column birth = new Column();
    birth.name = "birthday";
    birth.type = "date";
    student.columns.add(birth);

    Column home = new Column();
    home.name = "home";
    home.type = "varchar";
    student.columns.add(home);
```

接着向这两个表中分别插入一些数据，保存在data成员变量里面，这样，我们的数据就初始化完了，你可以想象这些数据是存储在csv文件中或者redis中，接着就需要和Calcite进行适配，Calcite建立jdbc连接需要一个json文件，这个文件的内容可以通过配置变量传入，也可以通过配置文件读取，文件的格式如下：

```
{
    version: '1.0',
    defaultSchema: 'school',
    schemas: [
        {
        name: 'school',
        type: 'custom',
         factory: 'org.apache.kylin.calcite.test.MemorySchemaFactory',
        operand: {
          param1: 'hello',
         param2: 'world';
        }
        }
    ]
}
```

这里只是一个比较简单的Calcite json model文件，详细的结构可以参考https://calcite.apache.org/docs/model.html，这个文件用于创建connection，所以这里配置的信息是提供给connection使用的，defaultSchema类似于连接mysql时提供database，可以不使用database名就可以访问该数据库的表，schemas定义了一些schema（database的概念），每一个schema指定了name、type（可以分为Map Schema、Custom Schema和JDBC Schema）。

这三种Schema有不同的使用方式，也决定了下面的参数。使用Map Schema意味着你需要在这个json文件中指定这个schema下的Tables和Functions（具体还需要哪些信息可以参考官方文档），也就是说这个schema是预先定义的（有哪些表，每一个表的结构），所以一般不适用这个（因为大部分情况下需要一些变量才能知道这个这个schema下有哪些表）；Custom Schema意味着你只需要指定factory和可选的operand参数（map结构），schema都是通过指定的factory类创建出来的（它需要实现org.apache.calcite.schema.SchemaFactory接口），具体这个schema下面有哪些表可以通过schema的name和operand变量决定生成。JDBC Schema意味着我们可以直接在这个model文件中配置一个jdbc的连接，所有向Calcite的操作

其实是由这个数据库完成的，一般也不常用（不如直接通过jdbc连这个数据库了）。

这里用的是最常用的Custom Schema，所以需要定义一个factory（MemorySchemaFactory），它实现了org.apache.calcite.schema.SchemaFactory接口，需要实现create函数，实现如下：

```
 public class MemorySchemaFactory implements SchemaFactory{
    @Override
    public Schema create(SchemaPlus parentSchema, String name, Map<String, Object> operand) {
        System. out.println( "param1 : " + operand.get( "param1"));
        System. out.println( "param2 : " + operand.get( "param2"));

        System. out.println( "Get database " + name);
        return new MemorySchema( name);
    }
}
```

这里为了测试打印了一些变量信息，通过测试可以看到name参数传递的是json文件中这个schema的name，operand是文件中这个schema定义的operand。这里要返回一个Schema对象，我们定义了MemorySchema类，需要实现org.apache.calcite.schema.Schema接口，MemorySchema继承了org.apache.calcite.schema.impl.AbstractSchema，后者实现了Schema接口并提供了默认实现，一般情况下我们需要实现下面几个接口：

> - public boolean contentsHaveChangedSince(long lastCheck , long now) 这个接口是为了检查cache是否过期，因为calcite默认会缓存schema的元数据，所以可以通过该函数实现cache有效性检查。
> - protected Map<String, Table> getTableMap() 这个接口是为了获取schema的元数据，返回值为表名和表对象的映射。
> - protected Multimap<String, Function> getFunctionMultimap() 这个接口为了获取该schema支持的UDF函数。

在MemorySchema中我们只实现了getTableMap函数：

```
@Override
public Map<String, Table> getTableMap() {
    Map<String, Table> tables = new HashMap<String, Table>();
    Database database = MemoryData. MAP.get( this. dbName);
    if(database == null)
        return tables;
    for(MemoryData.Table table : database. tables) {
        tables.put( table. tableName, new MemoryTable( table));
    }

    return tables;
}
```

可以看到，我们只是通过schema名在内存中MAP表里面查看对应的Database对象，然后使用Database对象中的Table作为Schema中的表，表的类型为MemoryTable。 根据文档中的指示，一般我们可以实现三种类型的Table：

> - a simple implementation of Table, using the ScannableTable interface, that enumerates all rows directly;
> - a more advanced implementation that implements FilterableTable, and can filter out rows according to simple predicates;
> - advanced implementation of Table, using TranslatableTable, that translates to relational operators using planner rules.

当使用ScannableTable的时候，我们只需要实现函数Enumerable<Object[]> scan(DataContext root);，该函数返回Enumerable对象，通过该对象可以一行行的获取这个Table的全部数据（也就意味着每次的查询都是扫描这个表的数据）；当使用FilterableTable的时候，我们需要实现函数Enumerable<Object[]> scan(DataContext root, List filters);参数中多了filters数组，这个数据包含了针对这个表的过滤条件，这样我们根据过滤条件只返回过滤之后的行，减少上层进行其它运算的数据集；当使用TranslatableTable的时候，我们需要实现RelNode toRel(RelOptTable.ToRelContext context, RelOptTable relOptTable);，该函数可以让我们根据上下文自己定义表扫描的物理执行计划，至于为什么不在返回一个Enumerable对象了，因为上面两种其实使用的是默认的执行计划，转换成EnumerableTableAccessRel算子，通过TranslatableTable我们可以实现自定义的算子，以及执行一些其他的rule，Kylin就是使用这个类型的Table实现查询。

为了简单，我们这里只是使用了ScannableTable，每次做全表扫描。当然除了上面Table需要实现的接口，还需要实现Calcite中最底层Table定义的接口，当然有AbstractTable实现了一些默认的方案，我们只需要实现获取表中元数据的函数getRowType和获取数据的函数scan。

```
@Override
public RelDataType getRowType(RelDataTypeFactory typeFactory) {
    if(dataType == null) {
        RelDataTypeFactory.FieldInfoBuilder fieldInfo = typeFactory.builder();
        for (MemoryData.Column column : this. sourceTable. columns) {
            RelDataType sqlType = typeFactory.createSqlType(
```

```
                    MemoryData.SQLTYPE_MAPPING.get(column .type ));
            sqlType = SqlTypeUtil.addCharsetAndCollation(sqlType, typeFactory);
            fieldInfo.add( column. name, sqlType);
        }
        this. dataType = typeFactory.createStructType( fieldInfo);
    }
    return this.dataType;
}

@Override
public Enumerable<Object[]> scan(DataContext root) {
    final int[] fields = identityList(this.dataType.getFieldCount());
    return new AbstractEnumerable<Object[]>() {
        public Enumerator<Object[]> enumerator() {
            return new MemoryEnumerator<Object[]>( fields, sourceTable. data);
        }
    };
}
```

表中的元数据（字段名和字段类型）是根据初始化数据中Table中每一个Column的类型转换的，MemoryData.SQLTYPE_MAPPING提供了自定义类型到Calcite类型的映射。scan函数返回一个迭代器对象，通过调用该对象的moveNext函数可以获取是否已经遍历完全部的数据，current函数返回当前的一行数据，还可以根据需要实现一些其他的函数，这里不再一一介绍了。

好了，整体的实现就是这个样子的，对于一个查询操作会经历如下的流程：Calcite会解析SQL并将其转换成逻辑执行计划，期间会根据当前connection中schema定义的信息初始化每一个Schema，然后根据查询中指定的schema调用对应的getTableMap函数获取元数据，根据这个信息判断查询中出现的表名、字段名是否正确以及检查SQL语法是否符合规范。然后再使用Calcite内部默认的实现生成物理执行计划，这个查询计划是树状结构的，最底层的节点是ScanTable操作（类似于SQL执行过程中首先执行FROM子句），对每一个表获取该表的数据，这时候使用的算子为默认的EnumerableTableAccessRel，然后再去调用具体ScannableTable的scan方法获取表的数据。完了之后在根据原始表的数据进行上层的JOIN、FILTER、GROUP BY、SORT、LIMIT甚至子查询等操作。

测试

测试代码：

```
public static void main(String[] args) {
 try {
            Class. forName("org.apache.calcite.jdbc.Driver");
       } catch (ClassNotFoundException e1) {
             e1.printStackTrace();
       }

    Properties info = new Properties();
    try {
        Connection connection =
            DriverManager.getConnection("jdbc:calcite:model=E:\\file\\to\\model\\file\\School.json", info );
        ResultSet result = connection.getMetaData().getTables( null, null, null, null);
        while( result.next()) {
            System. out.println( "Catalog : " + result.getString(1) + ",Database : " + result.getString(2) + ",Table : " + result
        }
        result.close();

        Statement st = connection.createStatement();
        result = st.executeQuery( "select \"home\", 1 , count(1) from \"Student\" as S INNER JOIN \"Class\" as C on S.\"classId\"
        while( result.next()) {
            System. out.println( result.getString(1) + "\t" + result.getString(2) + "\t" + result.getString(3));
        }
        result.close();
        connection.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

执行结果：

```
param1 : hello
param2 : world
Get database school
Catalog : null,Database : metadata,Table : COLUMNS
Catalog : null,Database : metadata,Table : TABLES
Catalog : null,Database : school,Table : Class
Catalog : null,Database : school,Table : Student
sichuan       1      1
zhejiang      1      1
henan  1      1
jiangsu       1      1
hebei  1      1
beijing       1      1
anhui  1      2
```

其中前面三行为Calcite创建Schema的时候打印的，下面四行为当前connection中的表，前两个表为系统表，后面两个表是我们自定义的表，接下来执行一次带有JOIN的SQL查询，能够输出正确的结果。需要注意的是，Calcite中元数据类似于Oracle的，所有的表和字段名都会在解析的时候转换成大写，但是在Calcite中又是大小写敏感的，因此除非你将所有的表名和字段名都定义成大写，获取在查询的时候对于字段和表都加上双引号（这样在解析的时候就不会被转换成大写了），否则很可能出现字段或者表找不到的错误（http://stackoverflow.com/questions/31118348/table-not-found-with-apache-calcite）。