Delta Lake 提供纯 Scala₩Java₩Python 操作 API，和 Flink 整合更加容易

最近，Delta Lake 发布了一项新功能，也就是支持直接使用 Scala、Java 或者 Python 来查询 Delta Lake 里面的数据，这个是不需要通过 Spark 引擎来实现的。Scala 和 Java 读取 Delta Lake 里面的数据是通过 Delta Standalone Reader 实现的；而 Python 则是通过 Delta Rust API 实现的。Delta Lake 是一个开源存储层，为数据湖带来了可靠性。Delta Lake 提供 ACID 事务、可扩展的元数据处理，并统一流数据和批数据处理。其完全兼容 Apache Spark™ 的 API。该项目已经部署在数千个组织中，每周处理超过 EB 级别的数据，成为数据和 AI 架构中不可或缺的一部分。在 Databricks 平台中，超过75%的数据都在 Delta Lake 中。

当前，Delta lake 除了能够使用 Apache Spark 读取，还支持 Amazon Redshift、Redshift Spectrum、Athena、Presto 以及 Hive，更多这方面的信息可以参见 Delta Lake 集成 。这篇文章中我将介绍如何使用纯 Scala、Java 或者 Python 来读取 Delta Lake 里面的数据。

Delta Standalone Reader

Delta Standalone Reader (下文简称 DSR) 是一个 JVM 类库，它允许你在不使用 Apache Spark 的情况下读取 Delta Lake 表。也就是说，它可以被任何不能运行 Spark 的应用程序使用。创建 DSR 的动机是为了更好地与 Presto、Athena、Redshift Spectrum、Snowflake 和 Apache Hive 等系统集成。对于 Apache Hive，里面就是使用了 DSR 来实现读取 Delta Lake 表的。有了这个 API，通过 Apache Flink 读取 Delta Lake 表也不是事了。

为了在 Java 或 Scala 中使用 DSR，我们需要 JDK 8 或以上版本，Scala 版本为 2.11 或者 2.12；同时还需要在项目中加入以下依赖：

```
<dependency>
  <groupId>io.delta</groupId>
  <artifactId>delta-standalone_2.12</artifactId>
  <version>0.2.0</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-client</artifactId>
  <version>2.7.2</version>
</dependency>
<dependency>
  <groupId>org.apache.parquet</groupId>
  <artifactId>parquet-hadoop</artifactId>
  <version>1.10.1</version>
</dependency>
```

如果你使用的是 SBT，请加上以下依赖：

```
libraryDependencies ++= Seq(
  "io.delta" %% "delta-standalone" % "0.2.0",
  "org.apache.hadoop" % "hadoop-client" % "2.7.2",
  "org.apache.parquet" % "parquet-hadoop" % "1.10.1")
```

使用 DSR 读取 Delta Lake 的元数据

DSR API 提供了读取 Delta Lake 元数据的
API。比如我们可以读取当前快照的版本、底层文件数目等。下面我们来看下如何操作：

```
import io.delta.standalone.DeltaLog;
import io.delta.standalone.Snapshot;
import io.delta.standalone.data.CloseableIterator;
import io.delta.standalone.data.RowRecord;

import org.apache.hadoop.conf.Configuration;

public class HelloWold {

    public static void printSnapshotDetails(String title, Snapshot snapshot) {
        System.out.println("===== " + title + " =====");
        System.out.println("version: " + snapshot.getVersion());
        System.out.println("number data files: " + snapshot.getAllFiles().size());
        System.out.println("data files:");
        snapshot.getAllFiles().forEach(file -> System.out.println(file.getPath()));
    }

    public static void main(String[] args) {
        DeltaLog log = DeltaLog.forTable(new Configuration(), "/tmp/iteblog/");

        printSnapshotDetails("current snapshot", log.snapshot());
        printSnapshotDetails("version 0 snapshot", log.getSnapshotForVersionAsOf(0));
        printSnapshotDetails("version 1 snapshot", log.getSnapshotForVersionAsOf(1));
        printSnapshotDetails("version 2 snapshot", log.getSnapshotForVersionAsOf(2));
    }
}
```

下面是输出结果：

```
===== current snapshot =====
version: 2
number data files: 48
data files:
part-00012-3e601227-c42a-4aa2-8d4e-47b346e3768b-c000.snappy.parquet
part-00001-e6bc1a0d-5165-40e4-bfea-04287348e239-c000.snappy.parquet
part-00011-b6e0f1da-083f-43eb-9eda-ad7a77727844-c000.snappy.parquet
part-00003-72411df0-2f8f-4ff2-8679-31984d378454-c000.snappy.parquet
part-00003-d24b9cae-1bbd-4a3d-9c80-c835ee2839da-c000.snappy.parquet
...

===== version 0 snapshot =====
version: 0
number data files: 16
data files:
part-00011-b6e0f1da-083f-43eb-9eda-ad7a77727844-c000.snappy.parquet
part-00003-d24b9cae-1bbd-4a3d-9c80-c835ee2839da-c000.snappy.parquet
part-00005-b0136635-773d-47ba-8e79-650831e5ba59-c000.snappy.parquet
part-00001-c0569730-5008-42fa-b6cb-5a152c133fde-c000.snappy.parquet
part-00007-49af50ad-a270-4c55-9b33-82127ef21943-c000.snappy.parquet
...

===== version 1 snapshot =====
version: 1
number data files: 32
data files:
part-00012-3e601227-c42a-4aa2-8d4e-47b346e3768b-c000.snappy.parquet
part-00011-b6e0f1da-083f-43eb-9eda-ad7a77727844-c000.snappy.parquet
part-00003-72411df0-2f8f-4ff2-8679-31984d378454-c000.snappy.parquet
part-00003-d24b9cae-1bbd-4a3d-9c80-c835ee2839da-c000.snappy.parquet
part-00005-b0136635-773d-47ba-8e79-650831e5ba59-c000.snappy.parquet
...

===== version 2 snapshot =====
version: 2
number data files: 48
data files:
part-00012-3e601227-c42a-4aa2-8d4e-47b346e3768b-c000.snappy.parquet
part-00001-e6bc1a0d-5165-40e4-bfea-04287348e239-c000.snappy.parquet
part-00011-b6e0f1da-083f-43eb-9eda-ad7a77727844-c000.snappy.parquet
part-00003-72411df0-2f8f-4ff2-8679-31984d378454-c000.snappy.parquet
part-00003-d24b9cae-1bbd-4a3d-9c80-c835ee2839da-c000.snappy.parquet
...
```

使用 DSR 读取 Delta Lake 的数据

DSR 提供的另一个重要的功能就是能够读取 Delta Lake 的数据，这个也使得可以在第三方计算引擎里面和 Delta Lake 进行整合。下面是使用 DSR 读取 Delta Lake 数据的例子：

```
// Create a closeable iterator
CloseableIterator iter = snapshot.open();

RowRecord row = null;
int numRows = 0;

// Schema of Delta table is {long, long, string}
while (iter.hasNext()) {
row = iter.next();
    numRows++;

    Long c1 = row.isNullAt("c1") ? null : row.getLong("c1");
    Long c2 = row.isNullAt("c2") ? null : row.getLong("c2");
    String c3 = row.getString("c3");
    System.out.println(c1 + " " + c2 + " " + c3);
}

// 输出结果
175 0 foo-1
176 1 foo-0
177 2 foo-1
178 3 foo-0
179 4 foo-1
```

Delta Rust API

如果你使用 Python，Delta Lake 为我们提供了 Delta Rust API 来读取 Delta Lake 的元数据和数据。

delta.rs 是 Delta Lake for Rust 的实验接口。该类库提供了对 Delta 表的低级访问，旨在与数据处理框架（如 datafusion, ballista, rust-dataframe 以及 vega）一起使用。它也可以作为其他语言（例如 Python，Ruby 或 Golang）的本地基础类库。

使用 Cargo 读取 Delta Lake 元数据

我们可以使用 API 或 CLI 来读取 Delta Lake

表的文件，同时也可以读取表的元数据信息。下面是通过 cargo 使用 CLI
的示例命令。我们可以通过 cargo install deltalake 拿到 delta-inspect 的二进制文件。

```
// 使用 cargo 读取 delta lake 表的文件列表
 cargo run --bin delta-inspect files ./tests/data/delta-0.2.0
```

输出结果

```
part-00000-cb6b150b-30b8-4662-ad28-ff32ddab96d2-c000.snappy.parquet
part-00000-7c2deba3-1994-4fb8-bc07-d46c948aa415-c000.snappy.parquet
part-00001-c373a5bd-85f0-4758-815e-7eb62007a15c-c000.snappy.parquet
```

```
// 使用 cargo 读取 delta lake 表的元数据
 cargo run --bin delta-inspect info ./tests/data/delta-0.2.0
```

输出结果

```
DeltaTable(./tests/data/delta-0.2.0) version: 3 metadata: GUID=22ef18ba-191c-4c36-a606-3dad
5cdf3830, name=None, description=None, partitionColumns=[], configuration={} min_version:
read=1, write=2 files count: 3
```

使用 Python 读取 Delta Lake 元数据

当然，我们也可以直接使用 Python 来读取 Delta Lake 的相关数据。比如我们可以使用 .version()
和 .files() 方法分别读取 Delta Lake 表的版本和文件列表：

```
from deltalake import

DeltaTable dt = DeltaTable("../rust/tests/data/delta-0.2.0")

# Get the Delta Lake Table version
dt.version()

# 输出
3

# List the Delta Lake table files
dt.files()

# 输出
['part-00000-cb6b150b-30b8-4662-ad28-ff32ddab96d2-c000.snappy.parquet', 'part-00000-7c2
deba3-1994-4fb8-bc07-d46c948aa415-c000.snappy.parquet', 'part-00001-c373a5bd-85f0-4758-
815e-7eb62007a15c-c000.snappy.parquet']
```

使用 Python 读取 Delta Lake 数据

通过 delta.rs Python 类库读取 Delta Lake 的数据，你需要将 Delta table 转换成 PyArrow Table 以及 Pandas Dataframe：

```
# Import Delta Table
from deltalake import DeltaTable

# 使用 Rust API 读取 Delta Lake 表
dt = DeltaTable("../rust/tests/data/simple_table")

# 通过将 Delta Lake 表转换成 PyArrow 表，然后创建 Pandas Dataframe
df = dt.to_pyarrow_table().to_pandas()
# 查询 Pandas 表
df

# 输出
0 5
1 7
2 9
```

当然，我们还可以利用 load_version 方法来进行读取之前版本的数据（Time Travel），如下：

```
# Load version 2 of the table
dt.load_version(2)
```

总结

Delta Lake 本质上是一种数据存储格式，不应该和任何计算引擎绑定。本文介绍的 DSR 以及 Delta Rust API 就是为了能够在更多的计算框架里面使用 Delta Lake。有了这两个 API，我们在 Apache Flink 中使用 Delta Lake 也将会在未来实现。

本文参考链接：Natively Query Your Delta Lake With Scala, Java, and Python、