Delta Lake 和 Apache Hudi 两种数据湖产品全方面对比

Delta Lake
是数砖公司在2017年10月推出来的一个项目，并于2019年4月24日在美国旧金山召开的 Spark+AI Summit 2019 会上开源的一个存储层。它是 Databricks Runtime 重要组成部分。为 Apache Spark 和大数据 workloads 提供 ACID
事务能力，其通过写和快照隔离之间的乐观并发控制（optimistic concurrency control），在写入数据期间提供一致性的读取，从而为构建在 HDFS 和云存储上的数据湖（data lakes）带来可靠性。Delta Lake 还提供内置数据版本控制，以便轻松回滚。更多关于 Delta Lake 的介绍可以参见过往记忆大数据的《Apache Spark 社区期待的 Delta Lake 开源了》以及《深入理解 Apache Spark Delta Lake 的事务日志》的介绍。

Hudi（Hoodie）是 Uber 为了解决大数据生态系统中需要插入更新及增量消费原语的摄取管道和 ETL 管道的低效问题，该项目在2016年开始开发，并于2017年开源，2019年1月进入 Apache 孵化器。更多关于 Apache Hudi 的介绍可以参见过往记忆大数据的《Apache Hudi: Uber 开源的大数据增量处理框架》以及《Uber 大数据平台的演进（2014~2019）》的介绍。

本文以中立的态度从高层次简单对比一下这两个数据湖产品的优缺点。

功能对比

这两款产品都支持对存储在 HDFS 上的数据进行增删改查，当然不只是支持 HDFS，其他兼容 HDFS 的存储系统也是支持的，比如 S3 等。下面我们简单介绍如何使用 Scala 来使用 Delta Lake 以及 Hudi。

添加数据

如果你之前在 Spark 里面写过 Parquet 文件，那么你可以很轻松的换成 Delta Lake 存储引擎，具体如下：

```
val data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-table")
```

而如果你要往 Hudi 写入数据，会稍微麻烦一些，具体如下：

```
val tableName = "hudi_archive_test"
val basePath = "file:///tmp/hudi_archive_test"

val inserts = List(
    """{"id" : 1,  "name": "iteblog", "age" : 101, "ts" : 1, "dt" : "20191212"}""",
    """{"id" : 2, "name": "iteblog_hadoop", "age" : 102, "ts" : 1, "dt" : "20191213"}""",
    """{"id" : 3, "name": "hudi", "age" : 103, "ts" : 1, "dt" : "20191212"}""")

val df = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
df.write.format("org.apache.hudi").
option(PRECOMBINE_FIELD_OPT_KEY, "ts").
option(RECORDKEY_FIELD_OPT_KEY, "id").
option(PARTITIONPATH_FIELD_OPT_KEY, "dt").
option(TABLE_NAME, tableName).
mode(Append).
save(basePath)
```

Hudi 写数据的时候需要指定 PRECOMBINE_FIELD_OPT_KEY，RECORDKEY_FIELD_OPT_KEY 以及 PARTITIONPATH_FIELD_OPT_KEY。RECORDKEY_FIELD_OPT_KEY 是每条记录的唯一 id，支持多个字段；PRECOMBINE_FIELD_OPT_KEY 在数据合并的时候使用到，当 RECORDKEY_FIELD_OPT_KEY 相同时，默认取 PRECOMBINE_FIELD_OPT_KEY 属性配置的字段最大值所对应的行；PARTITIONPATH_FIELD_OPT_KEY 用于存放数据的分区。

更新数据

在 Delta Lake 中更新数据可以指定条件，满足条件的数据将会被更新，使用如下：

```
import io.delta.tables._
import org.apache.spark.sql.functions._

val iteblogDeltaTable = DeltaTable.forPath(path)
```

```
// 对 id 为偶数的行 content 字段后面加上 -100
iteblogDeltaTable.update(
  condition = expr("id % 2 == 0"),
  set = Map("content" -> expr("concat(content, '-100')")))
```

而在 Hudi 中更新数据和插入数据类似，使用如下：

```
val tableName = "hudi_archive_test"
val basePath = "file:///tmp/hudi_archive_test"

val inserts = List(
    """{"id" : 1,  "name": "iteblog", "age" : 102, "ts" : 2, "dt" : "20191212"}""",
    """{"id" : 2, "name": "iteblog_hadoop", "age" : 103, "ts" : 2, "dt" : "20191213"}""",
    """{"id" : 3,  "name": "hudi", "age" : 104, "ts" : 2, "dt" : "20191212"}""")

val df = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
df.write.format("org.apache.hudi").
option(PRECOMBINE_FIELD_OPT_KEY, "ts").
option(RECORDKEY_FIELD_OPT_KEY, "id").
option(PARTITIONPATH_FIELD_OPT_KEY, "dt").
option(TABLE_NAME, tableName).
mode(Append).
save(basePath)
```

需要将 mode 设置为 Append，在底层，Hudi 会根据 RECORDKEY_FIELD_OPT_KEY、PRECOMBINE_FIELD_OPT_KEY 以及 PARTITIONPATH_FIELD_OPT_KEY 三个字段对数据进行 Merge，也就是更新。如两条记录相同，取 PRECOMBINE_FIELD_OPT_KEY 参数配置的值最大的哪一行。

删除数据

在 Delta Lake 中，删除数据也很简单，可以指定删除数据的条件，如下：

```
import io.delta.tables._

val iteblogDeltaTable = DeltaTable.forPath(spark, path)

// 删除 id 小于 4 的数据
```

```
iteblogDeltaTable.delete("id <= '4'")
```

Hudi 的数据删除支持软删除（Soft Deletes）和硬删除（Hard Deletes）。使用软删除时，用户希望保留键，但仅使所有其他字段的值都为空；而硬删除是从数据集中彻底删除记录在存储上的任何痕迹。

```
deleteDF
   .write().format("org.apache.hudi").
   option(...).
   option(PRECOMBINE_FIELD_OPT_KEY, "ts").
   option(RECORDKEY_FIELD_OPT_KEY, "id").
 option(PARTITIONPATH_FIELD_OPT_KEY, "dt").
   .option(DataSourceWriteOptions.PAYLOAD_CLASS_OPT_KEY, "org.apache.hudi.EmptyHoodie
RecordPayload")
```

查询数据

两者的查询数据功能都很简单，具体如下：

```
// Delta Lake
val df = spark.read.format("delta").load("/tmp/delta-table")
df.show()

// Hudi
val roViewDF = spark.
   read.
   format("org.apache.hudi").
   load(basePath + "/*/*/*/*")
```

上面简单介绍了 Delta Lake 和 Hudi 两种数据湖的使用。

数据 Merge 策略

Delta Lake 支持对存储的数据进行更新，并且仅支持写入的时候进行数据合并（Write On Merge），它会获取需要更新的数据对应的文件，然后直接读取这些文件并使用 Spark 的 Join 进行计算产生新的文件。

同理，Hudi 也是支持写入数据的时候进行合并，但是相比 Delta Lake，Hudi 还支持 Read On Merge 模式，也就是将增量数据写入到一个 delta
文件，然后默认情况下在更新完数据后会启动一个作业进行 compaction 操作。当然，这个也是可以关闭的，也就是更新的时候值负责数据的写入，合并操作可以单独用一个作业来跑。

从功能上来说，这方面 Hudi 比 Delta Lake 设计的要好。在少写多读的情况下，Write On Merge 模式很不错；而在多写少读的情况下，Read On Merge 模式很不错，而 Delta Lake 目前还不支持 Read On Merge 模式。

另外，Hudi 提供了索引机制，在数据合并的时候，可以使用索引的信息快速定位到某行数据所在的文件，从而加快数据更新的速度。

和其他系统兼容性

Delta Lake 的源码大量使用了 Spark 的代码实现，而 Hudi 虽然也是 Spark
的一个类库，但是其读写底层文件时并不依赖 Spark 去实现，而是实现了自己独有的 InputFormat，比如 HoodieParquetInputFormat、HoodieParquetRealtimeInputFormat 等。所以从这个角度上看，Hudi 和其他系统去做兼容比 Delta Lake 要容易。

目前 Hudi 原生支持 Spark、Presto、MapReduce 以及 Hive 等大数据生态系统，Flink
的支持正在开发中，参见 HUDI-184。Delta Lake 作为一个数据湖的存储层，其实不应该和 Spark 进行深入的绑定，数砖肯定意识到这个问题，所以数砖的大佬们弄了一个新的项目，用于和其他大数据生态系统进行整合，参见 https://github.com/delta-io/connectors，相关的 ISSUE 参见 https://github.com/delta-io/delta/issues/245。在前几天发布的 Delta Lake 0.5.0 也支持 Hive、Presto 等常见的大数据查询引擎。

SQL 语法

SQL 的支持能够为用户提供极大的便利。从 0.4.0 版本开始，Delta Lake
已经开始支持一些命令的 SQL 语法了。由于 Delta Lake
是单独的一个项目，如果需要让它支持所有的 SQL 语法，需要从 Apache Spark
里面拷贝大量的代码到 Delta Lake 项目中，不便于维护，所以这个版本只支持 vacuum 和 history 简单命令的 SQL 语法。

其他的 delete、update 以及 merge 的 DML 操作支持可能得等到 Spark 3.0
版本才会支持的。目前社区也在 Spark 3.0 里面的 DataSource V2 API 里面添加了对 DELETE/UPDATE/MERGE 的支持，详情参见 SPARK-28303。

Hudi 目前还不支持使用 SQL 进行 DDL / DML
相关操作，不过社区已经有小伙伴提到这个东西了，具体参见 HUDI-388。

相信在未来版本，Delta Lake 以及 Hudi 应该都会支持使用 SQL 来读写底层的数据了。

多语言支持

Delta Lake 的多语言这块做的比较好，目前支持使用 Scala、Java 以及 Python（0.4.0 版本开始支持）；而 Apache Hudi 目前只支持 Scala 以及 Java 语言。

并发控制

Delta Lake 原生提供了 ACID 事务的支持，每次写入都是一个事务，并且在事务日志中记录了写入的序列顺序。事务日志跟踪文件级别的写入并使用乐观并发控制。在存在冲突的情况下，Delta Lake 会抛出并发修改异常以便用户能够处理它们并重试其作业。Delta Lake 还提供强大的可序列化隔离级别，允许工程师持续写入目录或表，并允许消费者继续从同一目录或表中读取，读者将看到阅读开始时存在的最新快照。因为 ACID 的支持，Delta Lake 允许同一时刻多个用户同时写一张表。

Apache Hudi 每次修改也会生成一个事务提交，但是其不支持同一时刻多个用户同时写一张表，因为如果是这种情况最后的数据是不对的。

版本回退

Delta Lake 原生提供了一个叫做时间旅行（Time Travel）的功能，通过这个功能我们可以回退到任意版本的数据，具体如下：

```
val df = spark.read.format("delta").option("versionAsOf", 0).load("/tmp/delta-table")
df.show()
```

这个功能非常有用，比如我们发布新的系统时，然后发现了一个 Bug，这时候我们可以使用这个功能将数据回退到正确的状态。

Apache Hudi 也支持类似的概念，也就是增量视图（Incremental View），我们可以通过指定一个时间段，然后查询到这个时间段新增或者修改的数据，具体如下：

```
//incrementally query data
val incViewDF = spark.read.format("org.apache.hudi").
    option(VIEW_TYPE_OPT_KEY, VIEW_TYPE_INCREMENTAL_OPT_VAL).
    option(BEGIN_INSTANTTIME_OPT_KEY, beginTime).
    option(END_INSTANTTIME_OPT_KEY, endTime).
    load(basePath);
incViewDF.registerTempTable("hudi_incr_table")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from  hudi_incr_table where fare > 20.0").show()
```

如果我们要回退到某一版本的数据，可以把 BEGIN_INSTANTTIME_OPT_KEY 参数对应的值设置为 0，然后 END_INSTANTTIME_OPT_KEY 对应的参数设置为需要回退的 commit 时间（_hoodie_commit_time）

小文件问题

Databricks Runtime 里面的 Delta Lake 是支持使用 OPTIMIZE 命令将小文件进行压缩的，但是据说该实现与 Databricks 中的某些特定逻辑相关，所以数砖目前并不打算开源这部分的代码，这就导致开源版本的 Delta Lake 需要我们自己处理小文件问题了，具体方法可以参见 https://github.com/delta-io/delta/issues/49。

而 Hudi 可以通过将对该分区中的插入作为对现有小文件的更新来解决小文件的问题，可以通过 hoodie.parquet.small.file.limit 参数实现。

底层文件系统格式

Delta Lake 中的所有数据都是使用 Apache Parquet 格式存储，使 Delta Lake 能够利用 Parquet 原生的高效压缩和编码方案。

在 Hudi 中，如果我们使用 Write On Merge 存储类型（默认），那么底层也是使用 Apache Parquet 格式存储数据；而如果使用了 Read On Merge 存储类型，则同时使用 Avro 和 Parquet 来存储数据。其中 Avro 格式以行的形式存储增量的数据，而 Parquet 以列的形式存储已经 Merge 好的数据。

表模式管理

新增列

在 Delta Lake 中，默认情况下，如果插入的数据包含 Delta Lake 表中没有的列，会报错，如下：

```
scala> val df = spark.read.format("delta").load("/tmp/delta-table")
df1: org.apache.spark.sql.DataFrame = [age: bigint, dt: string ... 3 more fields]

scala> df.printSchema
root
 |-- age: long (nullable = true)
 |-- dt: string (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- ts: long (nullable = true)
```

如果我们往上面的 Delta Lake 表中插入以下的数据，这会报错：

```
scala> val inserts = List("""{"id" : 1,  "name": "iteblog", "age" : 102, "ts" : 2, "email": "spark@itebl
og.com", "dt" : "20191212"}""")
inserts: List[String] = List({"id" : 1,  "name": "iteblog", "age" : 102, "ts" : 2, "email": "spark@itebl
og.com", "dt" : "20191212"})

scala> val df = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
warning: there was one deprecation warning; re-run with -deprecation for details
df: org.apache.spark.sql.DataFrame = [age: bigint, dt: string ... 4 more fields]

scala> df.write.format("delta").mode("append").save("/tmp/delta-table")
org.apache.spark.sql.AnalysisException: A schema mismatch detected when writing to the Delt
a table.
To enable schema migration, please set:
'.option("mergeSchema", "true")'.

Table schema:
root
-- age: long (nullable = true)
-- dt: string (nullable = true)
-- id: long (nullable = true)
-- name: string (nullable = true)
-- ts: long (nullable = true)


Data schema:
root
-- age: long (nullable = true)
-- dt: string (nullable = true)
-- email: string (nullable = true)
-- id: long (nullable = true)
-- name: string (nullable = true)
-- ts: long (nullable = true)
```

上面我们在插入的数据加上了 email 这列，而这列在之前的 Delta Lake 表中是不存在的，默认情况下是会报错的。正如上面提示，我们在写入数据的时候加上 .option("mergeSchema", "true") 参数即可，这时候 Delta Lake 表就会增加新的一列。

在 Apache Hudi 中，新增加的列可以直接添加到已有的表中，如下：

```
val roViewDF = spark.
  read.format("org.apache.hudi").
```

```
  load(basePath + "/*/")
roViewDF.printSchema()

root
 |-- _hoodie_commit_time: string (nullable = true)
 |-- _hoodie_commit_seqno: string (nullable = true)
 |-- _hoodie_record_key: string (nullable = true)
 |-- _hoodie_partition_path: string (nullable = true)
 |-- _hoodie_file_name: string (nullable = true)
 |-- age: long (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- ts: long (nullable = true)
 |-- type: string (nullable = true)

val inserts = List("""{"id" : 7, "type": "N", "name": "new spark", "age" : 103, "email" : "spark@ite
blog.com", "ts" : 1}""")

val df = spark.read.json(spark.sparkContext.parallelize(inserts))
df.write.format("org.apache.hudi").
  option(PRECOMBINE_FIELD_OPT_KEY, "ts").
  option(RECORDKEY_FIELD_OPT_KEY, "id").
  option(TABLE_NAME, tableName).
  mode(Append).
  save(basePath)

val roViewDF1 = spark.
  read.
  format("org.apache.hudi").
  load(basePath + "/*/")
roViewDF1.printSchema()

root
 |-- _hoodie_commit_time: string (nullable = true)
 |-- _hoodie_commit_seqno: string (nullable = true)
 |-- _hoodie_record_key: string (nullable = true)
 |-- _hoodie_partition_path: string (nullable = true)
 |-- _hoodie_file_name: string (nullable = true)
 |-- age: long (nullable = true)
 |-- email: string (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- ts: long (nullable = true)
 |-- type: string (nullable = true)
```

从上可见新增加的字段已经加到 Hudi 的对应表中了。

缺省列处理

在 Delta Lake 中，如果我们插入的数据不包含对应表的全部字段，是可以插入进去的，并且缺少的列对应的数据为 null。但是在 Hudi 中，插入数据必须指定表的全部字段，否则会报错。

列字段类型修改

目前 Delta Lake 和 Hudi 都不支持修改表中已有字段的类型。

表类型

在 Delta Lake 中，数据存储到 Delta Lake 和读出来的类型是一致的，如下：

```
scala> val df = spark.read.schema(schema).json(spark.sparkContext.parallelize(inserts))
warning: there was one deprecation warning; re-run with -deprecation for details
df: org.apache.spark.sql.DataFrame = [id: int, create_time: timestamp ... 4 more fields]

scala> df.printSchema
root
 |-- id: integer (nullable = false)
 |-- create_time: timestamp (nullable = false)
 |-- name: string (nullable = false)
 |-- ts: long (nullable = false)
 |-- dt: string (nullable = false)
 |-- is_vip: boolean (nullable = false)

scala> df.write.format("delta").save("/tmp/delta-table")

scala> val r = spark.read.format("delta").load("/tmp/delta-table")
df: org.apache.spark.sql.DataFrame = [id: int, create_time: timestamp ... 4 more fields]

scala> r.printSchema
root
 |-- id: integer (nullable = true)
 |-- create_time: timestamp (nullable = true)
 |-- name: string (nullable = true)
 |-- ts: long (nullable = true)
 |-- dt: string (nullable = true)
 |-- is_vip: boolean (nullable = true)
```

这是因为 Delta Lake 在每次 commit 的文件里面存储了 DataFrame 的 StructType。但是在 Apache Hudi 中，存储到数据湖里面的数据类型会发生变化，如下：

```
scala> val df = spark.read.schema(schema).json(spark.sparkContext.parallelize(inserts))
warning: there was one deprecation warning; re-run with -deprecation for details
df: org.apache.spark.sql.DataFrame = [id: int, create_time: timestamp ... 4 more fields]

scala> df.printSchema
root
 |-- id: integer (nullable = false)
 |-- create_time: timestamp (nullable = false)
 |-- name: string (nullable = false)
 |-- ts: long (nullable = false)
 |-- dt: string (nullable = false)
 |-- is_vip: boolean (nullable = false)

df.write.format("org.apache.hudi").
  option(PRECOMBINE_FIELD_OPT_KEY, "ts").
  option(RECORDKEY_FIELD_OPT_KEY, "id").
  option(PARTITIONPATH_FIELD_OPT_KEY, "dt").
  option(TABLE_NAME, tableName).
  mode(Append).
  save(basePath)


val roViewDF = spark.
  read.format("org.apache.hudi").
  load(basePath + "/*")

scala> roViewDF.printSchema()

root
 |-- _hoodie_commit_time: string (nullable = true)
 |-- _hoodie_commit_seqno: string (nullable = true)
 |-- _hoodie_record_key: string (nullable = true)
 |-- _hoodie_partition_path: string (nullable = true)
 |-- _hoodie_file_name: string (nullable = true)
 |-- id: integer (nullable = true)
 |-- create_time: long (nullable = true)
 |-- name: string (nullable = true)
 |-- ts: long (nullable = true)
 |-- dt: string (nullable = true)
 |-- is_vip: boolean (nullable = true)
```

可以看出，create_time 已经由 timestamp 类型变成 long 类型了。这是因为 Apache Hudi 在每次提交生成的 commit 文件里面存储的是 Avro 的 Schema，这个过程需要将 DataFrame 的 StructType 转换成 Avro 支持的类型，而 Avro 支持的类型有 ECORD, ENUM, ARRAY, MAP, UNION, FIXED, STRING, BYTES, INT, LONG, FLOAT, DOUBLE, BOOLEAN, NULL。所以读取的时候也就只能读取到 Avro 支持的数据类型。

关于为什么 Apache Hudi 存储的是 Avro 支持的类型而不支持存储 Spark 的 StructType，我猜想应该是 Apache Hudi 把 Spark 只是当做其一个框架。使用 Spark 写入 Hudi 的数据可以使用 Apache Hive/Presto 读出来，如果存储的是 Spark StructType，那么在 Hive/Presto 里面是会报错的。

总结

Delta Lake 和 Hudi 两个都是数据湖的两款产品，各有优缺点。Hudi 的产生背景就是为了解决 Uber 的增量更新的问题，所以如果你对 Delta Lake 的数据 Merge 不是很满意，那么 Hudi 可能会适合你。如果你对多客户端同时写一张表很在意，那么 Delta Lake 可能会更适合你。