使用 Python APIs 对 Delta Lake 表进行简单可靠的更新和删除操作

在这篇我们介绍了 Spark Delta Lake 0.4.0 的发布，并提到这个版本支持 Python API 和部分 SQL。本文我们将详细介绍 Delta Lake 0.4.0 Python API 的使用。

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

在本文中，我们将基于 Apache Spark™ 2.4.3，演示一个准时航班情况业务场景中，如何使用全新的 Delta Lake 0.4.0 Python API。我们将展示如何 upsert 与删除数据，用时间旅行 (time travel) 查询数据的旧版本，以及用 vacuum 清理旧版本。

如何使用 Delta Lake

使用 --packages 选项使用 Delta Lake 包。在本文的例子中，我们也会演示在 Spark 中执行文件的 VACUUM 操作，以及执行 Delta Lake SQL 命令。为了完成这个简短的演示，我们要做以下设置：

- spark.databricks.delta.retentionDurationCheck.enabled=false
  允许对默认保留时长（7天）之内的文件执行 VACUUM 。注意，只有 SQL 命令 VACUUM 才需要这个配置。

- spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension 允许在 Apache Spark 中执行 Delta Lake SQL 命令；Python or Scala API 调用不需要这个配置。

```
# Using Spark Packages
./bin/pyspark --packages io.delta:delta-core_2.11:0.4.0 --conf "spark.databricks.delta.retention
DurationCheck.enabled=false" --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExte
nsion"
```

加载并保存 Delta Lake 数据

业务场景使用“航班准时延误情况数据集”，由美国交通运输局的 航班出发统计
生成。使用后者的案例有 2014 Flight Departure Performance via d3.js Crossfilter 和 On-Time Flight Performance with GraphFrames for Apache
Spark™。这份数据集可以从这个github地址下载到您本地。在 pyspark 中开始读取数据集。

```
# Location variables
tripdelaysFilePath = "/root/data/departuredelays.csv"
pathToEventsTable = "/root/deltalake/departureDelays.delta"

# Read flight delay data
departureDelays = spark.read ₩
.option("header", "true") ₩
.option("inferSchema", "true") ₩
.csv(tripdelaysFilePath)
```

接下来，让我们将出发延误数据集保存到 Delta Lake 表。通过将表存储到 Delta
Lake，我们可以利用这些特性：ACID 事务，统一批流处理，以及时间旅行 (time travel) 。

```
# Save flight delay data into Delta Lake format
departureDelays ₩
.write ₩
.format("delta") ₩
.mode("overwrite") ₩
.save("departureDelays.delta")
```

注意，此做法与一般的保存 Parquet 数据类似，不过您现在要指定 format("delta") 而不是 format("parquet") 。如果您查看下层的文件系统，可以发现 Delta Lake 表 departureDelays

创建了四个文件。

```
/departureDelays.delta$ ls -l
.
..
_delta_log
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
```

注意：_delta_log 是包含 Delta Lake 事务日志的文件夹。您可以在这个文档 Diving Into Delta Lake: Unpacking The Transaction Log 查看更多信息。

现在，让我们重新加载数据，不过现在我们的 DataFrame 将由 Delta Lake 支持。

```
# Load flight delay data in Delta Lake format
delays_delta = spark ₩
.read ₩
.format("delta") ₩
.load("departureDelays.delta")

# Create temporary view
delays_delta.createOrReplaceTempView("delays_delta")

# How many flights are between Seattle and San Francisco
spark.sql("select count(1) from delays_delta where origin = 'SEA' and destination = 'SFO'").sho
w()
```

	count(1)
0	1698

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

最后，让我们查明从西雅图到旧金山的航班次数。在本数据集中有1698次。

就地转换到 Delta Lake

如果您已有 Parquet 表，您可以将表就地转换为 Delta Lake
表，因此不需要重写表。您可以运行以下命令来转换：

```
from delta.tables import *

# Convert non partitioned parquet table at path '/path/to/table'
deltaTable = DeltaTable.convertToDelta(spark, "parquet.`/path/to/table`")

# Convert partitioned parquet table at path '/path/to/table' and partitioned by integer column named 'part'
partitionedDeltaTable = DeltaTable.convertToDelta(spark, "parquet.`/path/to/table`", "part int")
```

想了解如何更多信息，包括如何在 Scala 和 SQL 中转换，参考 Convert to Delta Lake

删除航班数据

从传统数据湖表中删除数据，您需要：

1. select所有数据，排除您要删的那些行
2. 基于上述查询创建一个新表
3. 删除原表
4. 新表重命名为原表，以满足下游依赖

在 Delta Lake 中，我们可以简单运行一个 DELETE
语句来完成删除，而不需要以上的步骤。为了演示，我们来删除所有提前和准时的航班（即 dalay < 0 ）。

```
from delta.tables import *
from pyspark.sql.functions import *

# Access the Delta Lake table
deltaTable = DeltaTable.forPath(spark, pathToEventsTable
)
# Delete all on-time and early flights
deltaTable.delete("delay < 0")

# How many flights are between Seattle and San Francisco
```

```
spark.sql("select count(1) from delays_delta where origin = 'SEA' and destination = 'SFO'").show()
```

	count(1)
0	837

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

我们删除所有准时和提前的航班之后，在查询中可以看到，西雅图出发到旧金山的延误航班有837次。如果您查看文件系统，可以注意到，删除数据之后文件反而变多了。

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-a2a19ba4-17e9-4931-9bbf-3c9d4997780b-c000.snappy.parquet
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00001-a0423a18-62eb-46b3-a82f-ca9aac1f1e93-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
part-00002-bfaa0a2a-0a31-4abf-aa63-162402f802cc-c000.snappy.parquet
part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
part-00003-b0247e1d-f5ce-4b45-91cd-16413c784a66-c000.snappy.parquet
```

在传统的数据湖中，删除是以重写被删除数据之外的整张表来实现。在 Delta Lake 中，删除操作则是选择性地写入文件的新版本，其中包括被删除的数据，而原先的文件只被置为删除。这是因为 Delta Lake 使用了多版本并发控制 (MVCC) 来完成表的原子操作：例如，当一名用户正在删除数据，另一名用户可能在查询表的先前版本。这种多版本模型让我们可以进行时间旅行（time travel），查询到先前的版本。稍后我们将看到这个例子。

更新航班数据

在传统数据湖中更新表，您需要：

1. select出所有数据，排除您要更新的那些行
2. 修改需要更新/变化的行

3. 将这两张表合并为一张新表
4. 删除原表
5. 新表重命名为原表，以满足下游依赖

在 Delta Lake 中，我们可以简单运行一个 UPDATE 语句来完成删除，而不需要以上的步骤。为了演示，我们来更新所有底特律出发到西雅图的航班。

```
# Update all flights originating from Detroit to now be originating from Seattle
deltaTable.update("origin = 'DTW'", { "origin": "'SEA'" } )

# How many flights are between Seattle and San Francisco
spark.sql("select count(1) from delays_delta where origin = 'SEA' and destination = 'SFO'").sho
w()
```

	count(1)
0	986

把底特律航班标记为西雅图航班之后，现在我们有986次从西雅图出发到旧金山的航班了。如果您在文件系统中列出 departureDelays 文件夹 ($../departureDelays/ls
-l)，会发现现在有11个文件（不同于删除文件后的8个，以及创建表之后的4个）。

合并航班数据

有个常见的场景，在数据湖中持续地给表追加数据。这常常导致重复数据（您不想再次插入表的行）—— 新行需要插入，有些行需要更新。在 Delta Lake
中，这些操作可以使用合并操作实现（类似 SQL 中的 MERGE 语句）。

我们以一个简单的数据集开始，您需要在其中使用下列查询来更新、插入或去重。

```
# What flights between SEA and SFO for these date periods
spark.sql("select * from delays_delta where origin = 'SEA' and destination = 'SFO' and date like
'1010%' limit 10").show()
```

查询结果如下表。注意，本文给数据打上了颜色，以清楚地区分哪些行是去重的（蓝色），更新的（黄色），以及插入的（绿色）。

	date	delay	distance	origin	destination
0	1010521	0	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010955	104	590	SEA	SFO

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

接下来，让我们编写以下代码片段，生成自己的
merge_table，包含将要插入、更新或去重的数据。

```
items = [(1010710, 31, 590, 'SEA', 'SFO'), (1010521, 10, 590, 'SEA', 'SFO'), (1010822, 31, 590, 'SEA
', 'SFO')]
cols = ['date', 'delay', 'distance', 'origin', 'destination']
merge_table = spark.createDataFrame(items, cols)
merge_table.toPandas()
```

	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010832	31	590	SEA	SFO

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

在以上的表 (merge_table) 中有三行，各有唯一的 date 值：

1. 1010521: 用新的延误值更新这些行（黄色）来 update flights 表
2. 1010710: 这是重复的行（蓝色）
3. 1010822: 这是要插入的新行（绿色）

在 Delta Lake 中，如下列代码片段所示，可以使用一条简单的 merge 语句来完成。

```
# Merge merge_table with flights
deltaTable.alias("flights") ₩
  .merge(merge_table.alias("updates"),"flights.date = updates.date") ₩
  .whenMatchedUpdate(set = { "delay" : "updates.delay" } ) ₩
  .whenNotMatchedInsertAll() ₩
  .execute()

# What flights between SEA and SFO for these date periods
spark.sql("select * from delays_delta where origin = 'SEA' and destination = 'SFO' and date like
'1010%' limit 10").show()
```

去重、更新和插入新行三个操作，使用一条语句高效地完成了。

	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010832	31	590	SEA	SFO
4	1010955	104	590	SEA	SFO

查看表历史

如上所述，在每一个事务（删除、更新）之后，文件系统中产生了更多文件。这是因为每个事务都有不同的Delta Lake 表版本，我们可以从下面的 DeltaTable.history() 方法中查看。

```
deltaTable.history().show()
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+-----------
--+-------------+
|version|        timestamp|userId|userName|operation| operationParameters| job|noteboo
k|clusterId|readVersion|isolationLevel|isBlindAppend|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+-----------
--+-------------+
|      2|2019-09-29 15:41:22|  null|    null|   UPDATE|[predicate -> (or...|null|    null|     null|          1|          null|         false|
|      1|2019-09-29 15:40:45|  null|    null|   DELETE|[predicate -> ["(...|null|    null|     null|          0|          null|         false|
|      0|2019-09-29 15:40:14|  null|    null|    WRITE|[mode -> Overwrit...|null|    null|     null|       null|          null|         false|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+-----------
--+-------------+
```

注意，你也可以使用 SQL 来实现：

```
spark.sql("DESCRIBE HISTORY '" + pathToEventsTable + "'").show()
```

可以看到，三行展现了不同的表版本，对应各自的表操作（创建表、删除表、更新表）。（下面给出缩略版以便阅读）

version	timestamp	operation	operationParameters
2	2019-09-29 15:41:22	UPDATE	[predicate -> (or...
1	2019-09-29 15:40:45	DELETE	[predicate -> [“(...
0	2019-09-29 15:40:14	WRITE	[mode -> Overwrit...

用表历史回溯到过去

使用时间旅行（Time Travel），您可以以版本或者时间戳来查看 Delta Lake 表。可以参考 Delta Lake 文档 Read older versions of data using Time Travel 了解更多。为了查看历史数据，指定 version 或 Timestamp 选项；在以下代码片段中，我们指定 version 选项。

```
# Load DataFrames for each version
dfv0 = spark.read.format("delta").option("versionAsOf", 0).load("departureDelays.delta")
dfv1 = spark.read.format("delta").option("versionAsOf", 1).load("departureDelays.delta")
dfv2 = spark.read.format("delta").option("versionAsOf", 2).load("departureDelays.delta")

# Calculate the SEA to SFO flight counts for each version of history
cnt0 = dfv0.where("origin = 'SEA'").where("destination = 'SFO'").count()
cnt1 = dfv1.where("origin = 'SEA'").where("destination = 'SFO'").count()
cnt2 = dfv2.where("origin = 'SEA'").where("destination = 'SFO'").count()

# Print out the value
print("SEA -> SFO Counts: Create Table: %s, Delete: %s, Update: %s" % (cnt0, cnt1, cnt2))

## Output
SEA -> SFO Counts: Create Table: 1698, Delete: 837, Update: 986
```

不论是企业治理、风险管理和合规 (GRC)，还是回滚错误，Delta Lake 都包含了元数据（例如记录了一个产生删除的操作），以及数据（例如实际被删的行）。不过，因为合规或者数据大小原因删除数据文件，应该怎么做呢？

使用 vacuum 清理旧的表版本

Delta Lake 的 vacuum 方法，可以默认删除7天以前的所有行（与文件），参考：Delta Lake Vacuum。

如果您查看文件系统，可以看到表的11个文件。

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-5e52736b-0e63-48f3-8d56-50f7cfa0494d-c000.snappy.parquet
part-00000-69eb53d5-34b4-408f-a7e4-86e000428c37-c000.snappy.parquet
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-20893eed-9d4f-4c1f-b619-3e6ea1fdd05f-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00001-d4823d2e-8f9d-42e3-918d-4060969e5844-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00002-3027786c-20a9-4b19-868d-dc7586c275d4-c000.snappy.parquet
part-00002-f2609f27-3478-4bf9-aeb7-2c78a05e6ec1-c000.snappy.parquet
part-00003-850436a6-c4dd-4535-a1c0-5dc0f01d3d55-c000.snappy.parquet
part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

为了删除当前快照之外的所有文件，您需要给 vaccum
方法指定一个很小的值（默认的保留时长是7天）。

```
# Remove all files older than 0 hours old.
deltaTable.vacuum(0)
```

注意：您可以使用SQL完成相同任务：

```
# Remove all files older than 0 hours old
spark.sql("VACUUM '" + pathToEventsTable + "' RETAIN 0 HOURS")
```

一旦 vacuum 完成，您再查看文件系统，将看到文件变少，因为历史数据已被删除。

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

注意，执行vacuum 之后，不能对早于预留时间的版本进行时间旅行。

接下来

为了尝试 Delta Lake，您可以在您的 Apache Spark 2.4.3 (或更高版本)
实例尝试以上代码片段。通过 Delta Lake，您的数据湖将更加可靠（不论是创建新的 Delta Lake 或从已有的数据湖迁移）。想学习更多，请查看 https://delta.io/ 并加入 Slack 和 Google Group 上的 Delta Lake 社区。您可以在 github milestones 中跟踪所有后续发布和计划中的特性。

本文英文原文：Simple, Reliable Upserts and Deletes on Delta Lake Tables using Python APIs
中文原文：Delta Lake 0.4.0 新特性演示：使用 Python API 就地转换与处理 Delta Lake 表