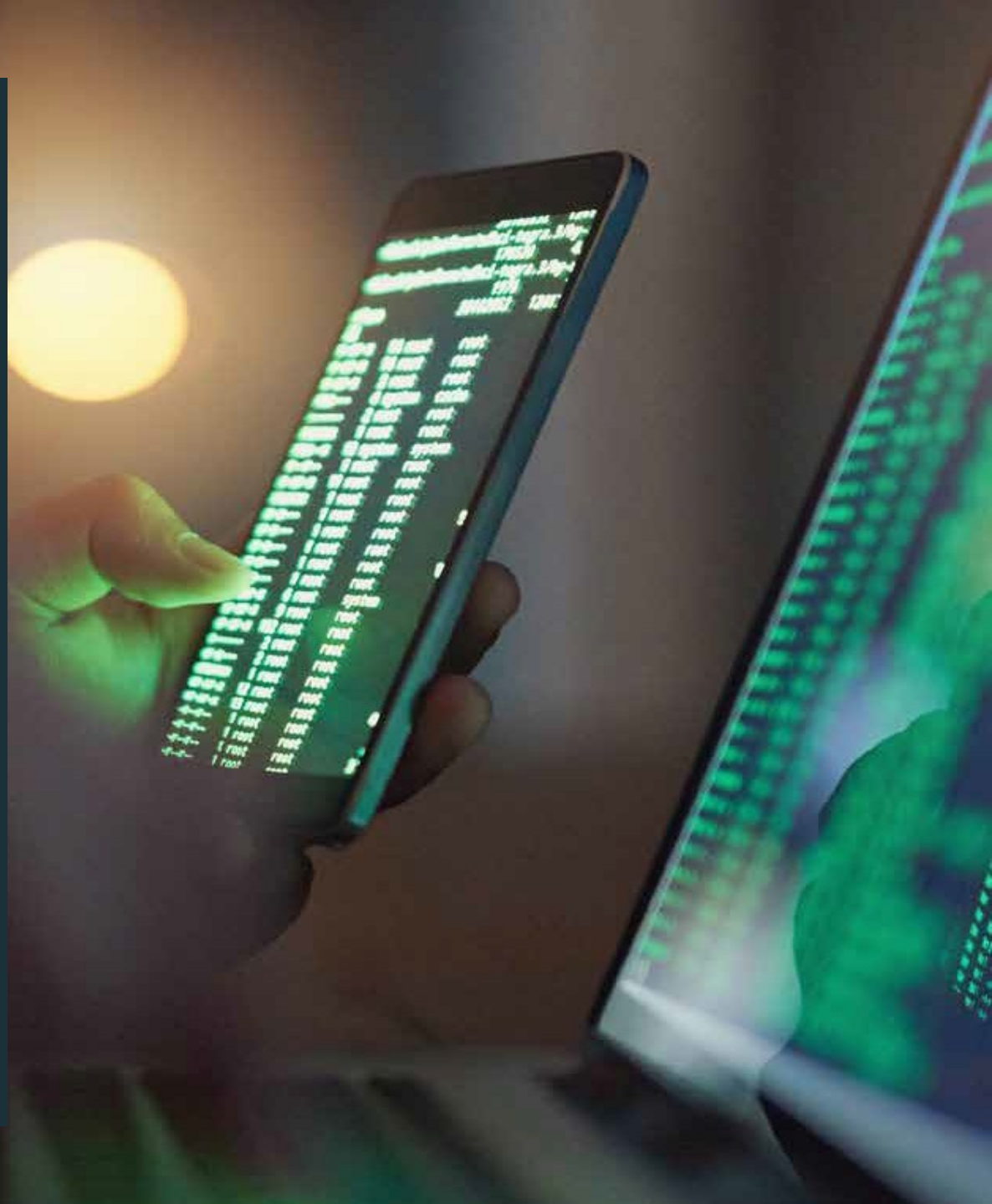
Delta Lake シリーズ

Delta Lake の機能

Delta Lake が可能にする
信頼性の高いデータの運用管理



databricks

このeBookの概要

Databricks の Delta Lake の eBook シリーズは、データを扱う方が Delta Lake のフル機能を理解して利活用するための支援を目的として提供されています。

この eBook「Delta Lake シリーズ：Delta Lake の機能」では、Delta Lake の機能を詳しく解説します。

学べる内容

Delta Lake が提供する機能の詳細と、その機能が性能を大幅に改善する仕組みを理解できます。

databricks

目次

Delta Lake とは

Delta Lake は、データの信頼性を高め、迅速な分析をクラウドのデータレイクにもたらす統合データ管理システムです。既存のデータレイク上で動作し、Apache Spark™ API と完全な互換性があります。

Databricks では、Delta Lake がデータレイクにもたらす信頼性、性能、ライフサイクル管理を実証してきました。私たちのお客様は Delta Lake を活用して、不正形式のデータ取り込みの回避、コンプライアンスに対応するデータ削除、データ取得時のデータ修正など、さまざまな課題を解決しています。

Delta Lake は、高品質データをデータレイクに迅速にもたらし、セキュアでスケーラブルなクラウドサービスで、チームによるデータの利活用を加速させます。

Chapter

01

Delta Lake で MERGE を使う理由

01

Delta Lake で MERGE を使う理由

Apache Spark 上に構築された次世代エンジンである Delta Lake は MERGE コマンドをサポートしており、データレイク内のレコードを効率的にアップサート・削除できます。

MERGE は、多くの一般的なデータ パイプラインを構築する方法をシンプルにします。パーティション全体を非効率的に書き換えていた複雑なマルチホップ処理を、シンプルな MERGE クエリで置き換えることができるようになりました。

このきめ細かいアップデート機能により、変更データの取り込みから GDPR まで、さまざまなユースケースに対応したビッグデータパイプラインの構築方法が簡素化されます。テーブルを上書きするための複雑なロジックを記述したり、スナップショットの分離不足を克服したりする必要はもうありません。

データが変化していく中で、もう一つ必要とされる重要な機能は、書き込みが悪かった場合のロールバック機能です。Delta Lake はタイムトラベル機能を使ったロールバック機能を提供しているので、万が一不正な MERGE を行ってしまった場合でも、以前のバージョンに簡単にロールバックすることができます。

この章では、既存のデータを更新または削除する必要がある場合の一般的な使用例について説明します。また、アップサートに固有の課題を探り、MERGE を使用してどのように対処できるかを説明します。

UPSERT はどんな場合に必要か

データレイク内の既存のデータを更新または削除する必要がある場合、多くの一般的なユースケースがあります。

一般データ保護規則（GDPR）への対応

GDPR で忘れられる権利（データ消去とも呼ばれる）が導入されたことで、組織は要求に応じてユーザーの情報を削除しなければなりません。このデータ消去には、データレイク内のユーザー情報の削除も含まれます。

従来のデータベースからのデータ取得を変更

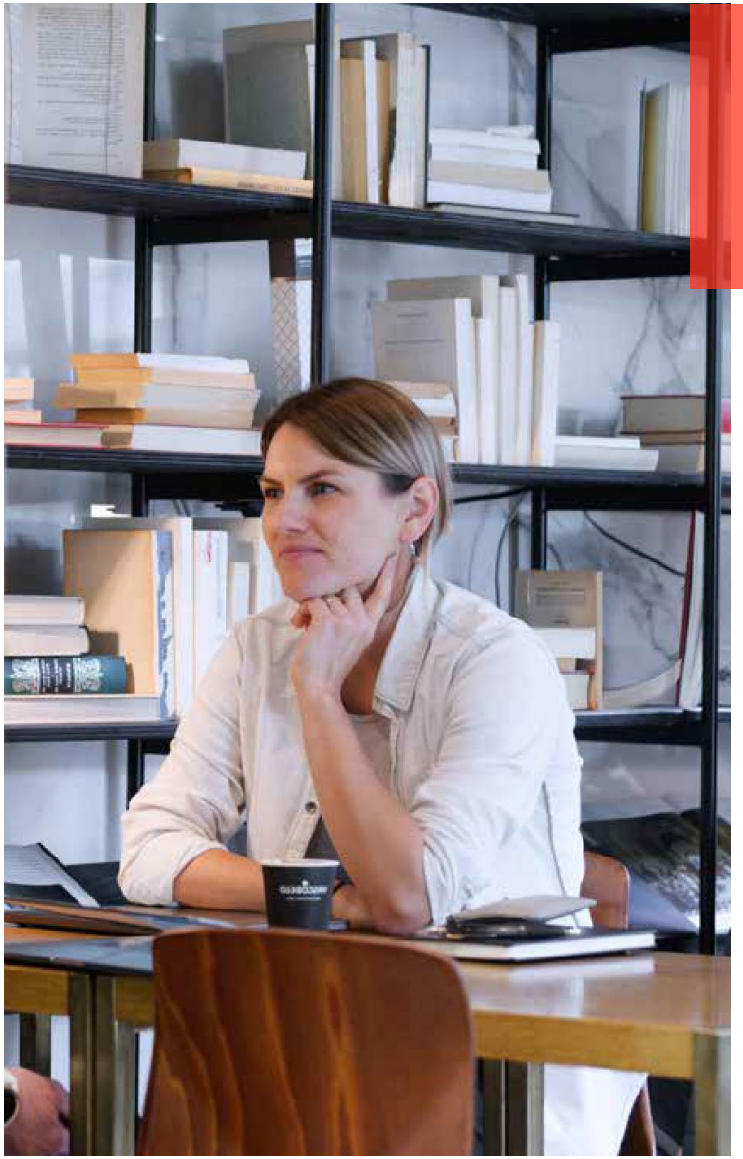
サービス指向アーキテクチャでは、通常、Web アプリケーションやモバイル アプリケーションは、低遅延に最適化された従来の SQL/NoSQL データベース上に構築されたマイクロサービスによって提供されています。組織が直面する最大の課題の1つは、これらのさまざまなサイロ化されたデータシステム間でデータを結合することであり、そのためデータエンジニアは、分析を容易にするために、すべてのデータソースを中央のデータレイクに統合するためのパイプラインを構築します。これらのパイプラインは、従来のSQL/NoSQL テーブルで行われた変更を定期的に読み取り、データレイク内の対応するテーブルに適用しなければならないことがよくあります。このような変更は、さまざまな形で行われます。ディメンションがゆっくりと変化するテーブル、すべての挿入／更新／削除された行の変更データのキャプチャなど。

セッション化

複数のイベントを1つのセッションにグループ化することは、製品分析からターゲティング広告、予測メンテナンスまで、多くの分野で一般的なユースケースです。セッションを追跡し、データレイクに書き込む結果を記録するための継続的なアプリケーションを構築することは、データレイクは常にデータを追加するために最適化されているため、困難です。

重複排除

一般的なデータパイプラインのユースケースは、システムログを Delta Lake のテーブルにデータを追加して収集することです。しかし、多くの場合、ソースから重複したレコードが生成されることがあり、それを処理するために下流の重複排除ステップが必要になります。



データレイクへのアップサートが従来から課題だった理由

データレイクは基本的にファイルをベースにしているため、既存のデータを変更するためではなく、データを追加するために常に最適化されています。したがって、上記のユースケースを構築することは、常に挑戦的なものでした。

ユーザーは通常、テーブル全体（またはパーティションのサブセット）を読み込んでから上書きします。したがって、すべての組織は、SQL、Spark などで複雑なクエリを手書きすることによって、要件のために車輪を再発明しようとします。

非効率

少数のレコードの更新にパーティション／テーブル全体を読み込んだり書き換えたりすると、パイプラインが遅くなり、コストがかかります。テーブルレイアウトやクエリの最適化を手作業で調整するのは面倒で、深いドメイン知識が必要です。

不正確な可能性

データを修正する手書きのコードは、論理エラーやヒューマンエラーが非常に発生しやすいです。例えば、複数のパイプラインがトランザクションのサポートなしに、同時にテーブルを変更している場合、予測不可能なデータの不整合が発生し、最悪の場合、データの損失につながる可能性があります。多くの場合、単一の手書きのパイプラインであっても、ビジネスロジックのエンコーディングのエラーにより、データの破損を簡単に引き起こす可能性があります。

維持が困難

このような手書きのコードは理解・追跡しにくく、維持するのが難しい。長期的に見ると、これだけで組織やインフラのコストが大幅に増加する可能性があります。

Delta Lake に MERGE を導入

Delta Lakeでは、次の MERGE コマンドを使用することで、前述のような問題なく、上記のユースケースに簡単に対応できます。

```
MERGE INTO
USING
ON
[ WHEN MATCHED [ AND ] THEN ]
[ WHEN MATCHED [ AND ] THEN ]
[ WHEN NOT MATCHED [ AND ] THEN ]
```

```
where

=
DELETE|
UPDATE SET * |
UPDATE SET column1 = value1 [, column2 = value2 ...]

=
INSERT * |
INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...])
```

簡単な例を使って MERGE の使用方法を理解しましょう。アドレスなどのユーザ情報を保持する、緩やかに変化するディメンション変換（SCD）テーブルと、既存のユーザと新規ユーザの両方の新しいアドレスのテーブルがあるとします。すべての新しいアドレスをメインユーザテーブルに MERGE するには、次のように実行します。

```
MERGE INTO users  USING
updates
ON users.userId = updates.userId  WHEN
MATCHED THEN
    UPDATE SET address = updates.addresses  WHEN
NOT MATCHED THEN
INSERT (userId, address) VALUES (updates.userId, updates.address)
```

既存のユーザ（すなわち、MATCHED 句）に対してはアドレス列を更新し、新規ユーザ（すなわち、NOT MATCHED 句）に対してはすべての列を挿入します。TB のデータを持つ大規模なテーブルの場合、Delta Lake は関連するファイルのみを読み込んで更新するため、この Delta Lake MERGE 操作はパーティションやテーブル全体を上書きするよりも何桁も速くなる可能性があります。具体的には、Delta Lake の MERGE には以下のような利点があります。

細かい粒度

この操作は、パーティションではなくファイルの粒度でデータを書き換えます。これにより、パーティションの書き換えや Hive メタストアを MSKK で更新するなどの煩雑さを防げます。

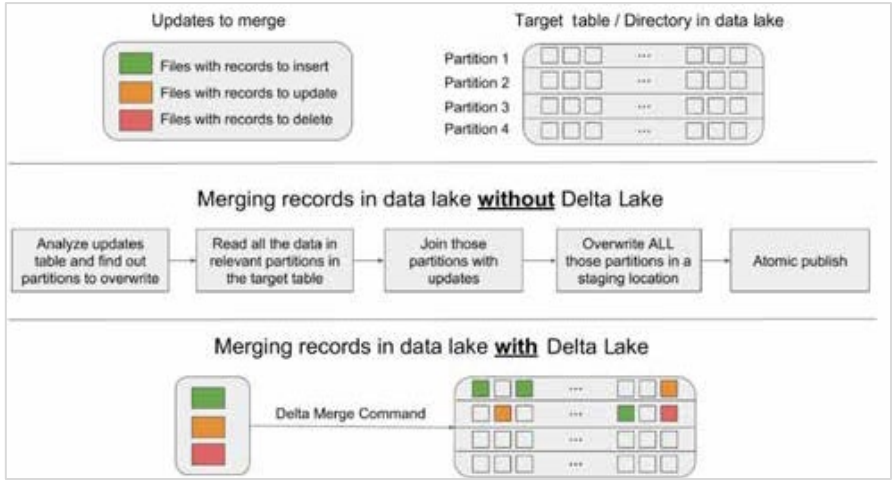
効率的

Delta Lake のデータスキップにより、MERGE は書き換えるファイルを効率的に見つけることができるため、パイプラインを手作業で最適化する必要がありません。さらに、Delta Lake は、I/O と処理の最適化をすべて備えているため、MERGE によるすべてのデータの読み書きが、Apache Spark の同様の操作よりも高速になります。

トランザクションを使用

Delta Lake は楽観的同時実行制御を使用して、同時実行ライターが ACID トランザクションでデータを正しく更新し、同時実行リーダーが常に一貫したデータのスナップショットを表示するようにしています。

以下は、MERGE と手書きのパイプラインの比較を視覚的に説明したものです。



MERGE でユースケースを簡素化

GDPR によるデータの削除

データレイク内のデータについて GDPR の「忘れられる権利」条項を遵守することは、これ以上簡単ではありません。以下のように、サービスからオプトアウトしたすべてのユーザーを削除するためのコードの例で簡単なスケジュールジョブを設定できます。

```
MERGE INTO users
USING opted_out_users
ON opted_out_users.userId = users.userId
WHEN MATCHED THEN DELETE
```

データベースからの変更データの適用

以下のように MERGE 構文を使用して、外部データベースから生成されたすべてのデータの変更、更新、削除、挿入を Delta Lake テーブルに簡単に適用できます。

```
MERGE INTO users  USING (
SELECT userId, latest.address AS address, latest.deleted AS deleted FROM
(
SELECT userId, MAX(struct(TIME, address, deleted)) AS latest
FROM changes GROUP BY userId
)
) latestChange
ON latestChange.userId = users.userId
WHEN MATCHED AND latestChange.deleted = TRUE THEN
DELETE
WHEN MATCHED THEN
UPDATE SET address = latestChange.address
WHEN NOT MATCHED AND latestChange.deleted = FALSE THEN
INSERT (userId, address) VALUES (userId, address)
```

ストリーミングパイプラインからのセッション情報の更新

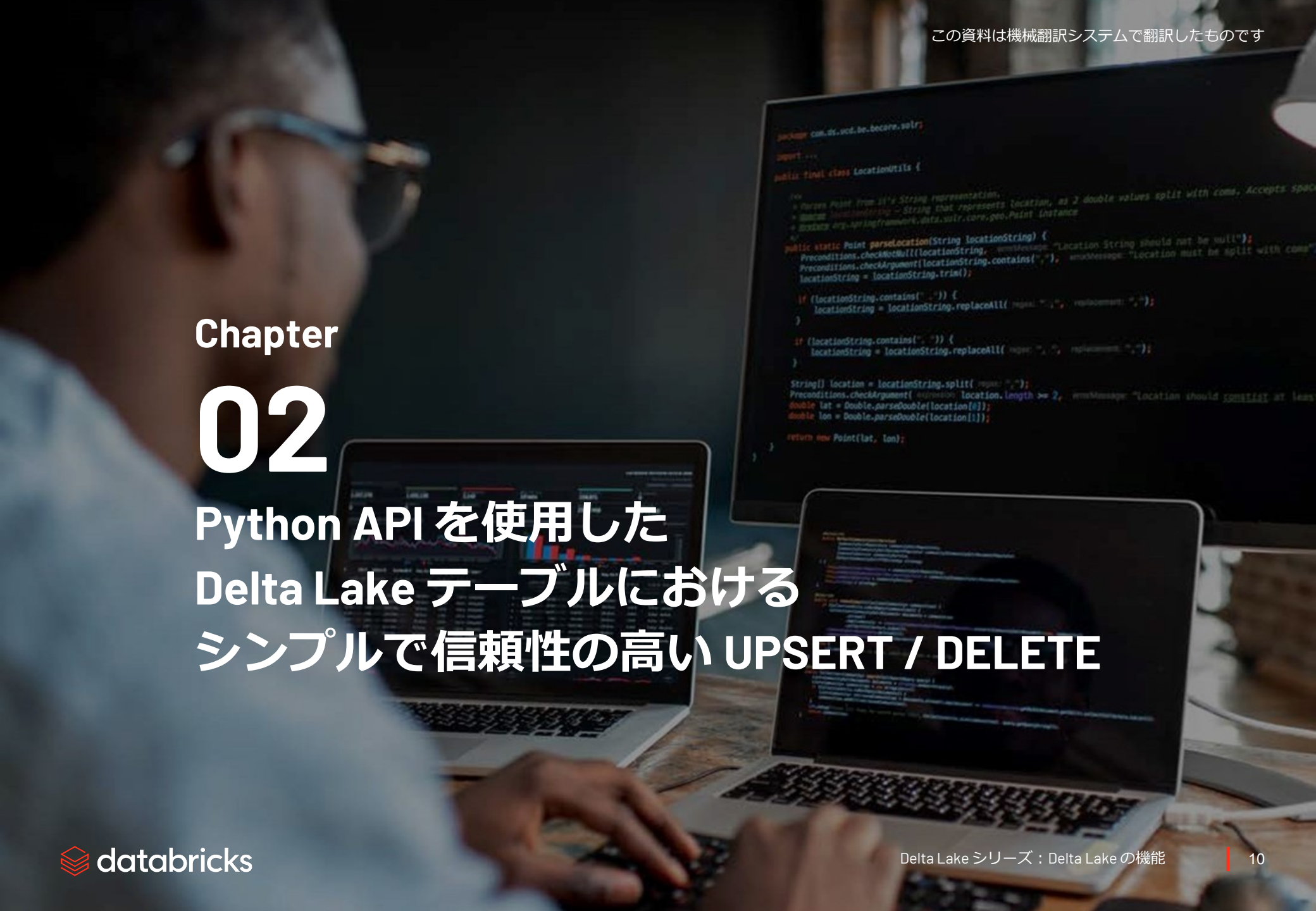
ストリーミング イベント データが流入している場合で、ストリーミング イベント データをセッション化し、セッションをインクリメンタルに更新してDelta Lakeテーブルに保存したい場合は、Structured Streaming と MERGE の foreachBatch を使用してこれを実現できます。 たとえば、各ユーザーの更新されたセッション情報を計算する Structured Streaming DataFrame があるとします。以下のように Delta Lake テーブルにすべてのセッションの更新を適用するストリーミングクエリを開始することができます（Scala）。

```
streamingSessionUpdatesDF.writeStream
.foreachBatch { (microBatchOutputDF: DataFrame, batchId: Long) =>
microBatchOutputDF.createOrReplaceTempView("updates")
microBatchOutputDF.sparkSession.sql(s"""
MERGE INTO sessions
USING updates
ON sessions.sessionId = updates.sessionId
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT * """)
}.start()
```

各 Batch と MERGE の完全な作業例については、 Notebook （Azure | AWS）を参照してください。

その他のリソース

- 技術トーク（動画）：Delta Lake と Apache Spark で GDPR と CCPA に対応
- 技術トーク（動画）：Delta を変更データの取得ソースとして使用
- ブログ：Databricks Delta で変更データの取得を簡素化
- 動画：Databricks Deltaで大規模なセッション化パイプラインを構築
- 技術トーク（動画）：緩やかに変化するディメンション変換（SCD）タイプ 2



Chapter

02

Python API を使用した Delta Lake テーブルにおける シンプルで信頼性の高い UPSERT / DELETE

02

Python API を使用した Delta Lake テーブルにおける シンプルで信頼性の高い UPSERT / DELETE

この章では、定刻フライトのシナリオの中で、 Delta Lake の Python と新しい Python API を使用する方法をデモします。データのアップサートや削除、タイムトラベルを使った古いバージョンのデータの問い合わせ、古いバージョンを VACUUM してクリーンアップする方法を紹介します。

Delta Lakeの利用開始方法

Delta Lake パッケージは、`--packages` オプションを使用して PySpark からインストールすることができます。この例では、Apache Spark 内での VACUUM ファイルと Delta Lake SQL コマンドの実行機能もデモします。短いデモなので、以下の設定も有効にします。

```
spark.databricks.delta.retentionDurationCheck.enabled=false
```

これを使用して、デフォルトの 7 日間の保持期間よりも短い期間でファイルを VACUUM できます。注意：これは SQL コマンド VACUUM にのみ必要です。

```
spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
```

これを使用して Apache Spark 内で Delta Lake SQL コマンドを有効にします。これは Python または Scala API 呼び出しでは必要ありません。

```
# Using Spark Packages
./bin/pyspark --packages io.delta:delta-core_2.11:0.4.0
--conf “spark.databricks.delta.retentionDurationCheck.enabled=false”
--conf “spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension”
```

Delta Lake データの読み込みと保存

このシナリオでは、RITA BTS Flight Departure Statistics から生成された On-time Flight Performance または Departure Delays データセットを使用します。このデータの例としては、2014 Flight Departure Performance via d3.js Crossfilter や、GraphFrames for Apache Spark™ を使った On-time Flight Performance PySpark 内でデータセットを読み込むことから始めます。

```
# 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)
```

次に Departure Delays データセットを Delta Lake テーブルに保存します。このテーブルを Delta Lake ストレージに保存することで、ACID トランザクション、バッチ／ストリーミングの統合、タイムトラベルなどの機能を利用できるようになります。

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

このアプローチは通常の Parquet データの保存方法に似ており、`format("parquet")` の代わりに `format("delta")` を指定することになります。基礎となるファイルシステムを見てみると、Departure Delays Delta Lake テーブル用に作成された 4 つのファイルがあることがわかります。

```
/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
```

次に、データをリロードします。今回の DataFrame は Delta Lake にバックアップされています。

```
# 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'").show()
```

	count(1)
0	1698

シアトル発サンフランシスコ行きのフライト数を調べてみましょう。 このデータセットでは、1698 便のフライトがあります。

Delta Lake へのインプレース変換

既存の Parquet テーブルをお持ちの場合、その場で Delta Lake 形式に変換する機能がありますので、テーブルを書き換える必要はありません。テーブルを変換するには、以下のコマンドを実行します。

```
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")
```

フライトデータの削除

従来のデータレイクテーブルからデータを削除するには、次のようにします。

1. 削除したい行を含まないテーブルからすべてのデータを選択
2. 前のクエリに基づいて新しいテーブルを作成
3. 元のテーブルを削除
4. 下流の依存関係のために新しいテーブルの名前を元のテーブル名に変更

Delta Lake では、これらの手順をすべて実行する代わりに DELETE ステートメントを実行することで、このプロセスを単純化できます。これを示すために、早く到着したフライトや定刻に到着したフライト（遅延 < 0）をすべて削除してみましょう。

```
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

定刻便と定刻より早い便をすべて削除した後（これについては以下で詳しく説明します）、先ほどのクエリからわかるように、シアトルからサンフランシスコに向けて出発する 837 便の遅延便があります。ファイル・システムを確認すると、データを削除したにもかかわらず、より多くのファイルがあることがわかります。

```
/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
```

従来のデータレイクでは、削除する値を除いたテーブル全体を書き換えることで削除が実行されていました。Delta Lake では、削除するデータを含むファイルの新しいバージョンを選択的に書き換えることで削除が実行され、以前のファイルのみが削除されたものとしてマークされます。これは、Delta Lake が MVCC（Multiversion concurrency control）を使用してテーブル上でアトミックな操作を行うためです。例えば、あるユーザーがデータを削除している間に、別のユーザーが前のバージョンのテーブルを照会しているかもしれません。このマルチバージョンモデルでは、後述するように、過去にさかのぼって（タイムトラベル）、以前のバージョンを照会することも可能です。

フライトデータの更新

従来の Data Lake テーブルからデータを更新するには、次のようにします。

1. 変更したい行を含まないテーブルからすべてのデータを選択
2. 更新/変更が必要な行の修正
3. これら2つのテーブルをMERGEして新しいテーブルを作成
4. 元のテーブルを削除
5. 下流の依存関係のために、新しいテーブルの名前を元のテーブル名に変更

Delta Lake では、これらの手順をすべて実行する代わりに、UPDATE ステートメントを実行し、このプロセスを単純化できます。デトロイト発シアトル行きのフライトをすべて更新してみましょう。

```
# 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'").show()
```

	count(1)
0	986

デトロイト便がシアトル便としてタグ付けされたことで、シアトル発サンフランシスコ行きのフライトは 986 便になりました。 Departure Delays フォルダのファイルシステムをリストアップすると（`$../departureDelays/ls -l`）、ファイルを削除した直後の 8 個のファイルと、テーブルを作成した後の 4 個のファイルの代わりに 11 個のファイルがあることがわかります。

フライトデータのMERGE

データレイクを使用しているときによくあるシナリオは、テーブルにデータを継続的に追加することです。その結果、データの重複（テーブルに再び挿入したくない行）、挿入が必要な新しい行、更新が必要な行が発生することがよくあります。Delta Lakeでは、これらすべてを MERGE 操作（SQL MERGE ステートメントに似ています）を使用することで実現できます。

以下のクエリを使用して、更新、挿入、または重複排除したいデータセットのサンプルから始めてみましょう。

```
# 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()
```

このクエリの出力は以下の表のようになります。どの行が重複排除（青）、更新（黄）、挿入（緑）されているかを明確に識別するために色分けが追加されていることに注意してください。

	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

次のコードスニペットを使用して、挿入、更新、重複排除するデータを含む独自の merge_table を生成してみましょう。

```
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

前のテーブル（merge_table）では、一意の日付の値を持つ 3 つの行があります。

1. 1010521：新しい遅延値でフライトテーブルを更新する必要がある行（黄色）
2. 1010710：重複している行（青）
3. 1010832：新たに挿入される行（緑）

Delta Lake では、以下のコードスニペットのように MERGE 文で容易に実現できます。

```
# 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()
```

重複除去、更新、挿入の 3 つのアクションはすべて 1 つのステートメントで効率的に完了しました。

	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

テーブルの履歴を見る

前述のとおり、各トランザクション（削除、更新）の後、ファイルシステム内にはより多くのファイルが作成されました。これは、トランザクションごとに、Delta Lake テーブルの異なるバージョンが存在するためです。これは、次の DeltaTable.history() メソッドを使用することで確認できます。

```
deltaTable.history().show()
+-------+-------------------+------+--------+---------+--------------------+---
-+--------+---------+-----------+-------------+-------------+
|version|          timestamp|userId|userName|operation| operationParameters|
job|notebook|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|
+-------+-------------------+------+--------+---------+--------------------+---
-+--------+---------+-----------+-------------+-------------+
```

このタスクはSQLでも実行できます。

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

ご覧のように、テーブルの異なるバージョンを表す3つの行があります（次の表は読みやすくするための簡略化されたバージョンです）。

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...

テーブルヒストリーでタイムトラベル

タイムトラベルを使用すると、バージョンまたはタイムスタンプの時点でDelta Lakeのテーブルを確認できます。 履歴データを表示するには、バージョンまたはタイムスタンプのオプションを指定します。

```
# 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
```

ガバナンス、リスク管理、コンプライアンス（GRC）、エラーのロールバックのためにしても、Delta Lakeのテーブルには、メタデータ（例えば、これらの演算子で削除が発生したという事実を記録）とデータ（例えば、削除された実際の行）の両方が含まれています。しかし、コンプライアンスやサイズの理由から、データファイルを削除するにはどうすればよいでしょうか。

VACUUMで古いテーブルバージョンをクリーンアップ

Delta LakeのVACUUMは、デフォルトで7日分の参照よりも古い行とファイルをすべて削除します。ファイルシステムには11個のファイルがあることが確認できます。

```
/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
```

すべてのファイルを削除して、現在のデータのスナップショットのみを保持するようにするには、デフォルトの7日間の保持ではなくVACUUM方法に小さな値を指定します。

```
# Remove all files older than 0 hours old.
deltaTable.vacuum(0)
Note, you perform the same task via SQL syntax:¸
# Remove all files older than 0 hours old
spark.sql("VACUUM '" + pathToEventsTable + "' RETAIN 0 HOURS")
```

VACUUMが完了してファイルシステムを見直すと、過去のデータが削除され、ファイルの数が少なくなっていることがわかります。

```
/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：保持期間よりも古いバージョンにタイムトラベルする機能は、VACUUMを実行すると失われます。

Chapter

03

大規模なデータレイクのためのタイムトラベル

03

大規模なデータレイクのための タイムトラベル

Delta Lake ではタイムトラベル機能が利用できます。Delta Lake は、データレイクに信頼性をもたらすオープンソースのストレージレイヤーです。 Delta Lake は、ACID トランザクション、スケーラブルなメタデータ処理を提供し、ストリーミングとバッチデータ処理を統合します。Delta Lake は既存のデータレイクの上で動作し、Apache Spark API と完全に互換性があります。

この機能により、Delta Lake はデータレイクに保存されているビッグデータを自動的にバージョン化し、そのデータの任意の履歴バージョンにアクセスすることができます。この一時的なデータ管理により、監査、誤った書き込みや削除があった場合のデータのロールバック、実験やレポートの再現が容易になり、データパイプラインが簡素化されます。

あなたの組織は、アナリティクスのために、独自のクラウドストレージにあるクリーンで一元化されたバージョンのビッグデータリポジトリをついに標準化することができます。

データの変化に伴う共通の課題

データの変化の監査

データの変更を監査することは、データのコンプライアンスだけでなく、データが時間の経過とともにどのように変化したかを理解するための単純なデバッグの観点からも非常に重要です。従来のデータシステムからビッグデータ技術やクラウドに移行する企業は、このようなシナリオで苦労しています。

実験やレポートを再現

モデルのトレーニング中、データサイエンティストは、与えられたデータセット上で異なるパラメータを用いて様々な実験を実行します。サイエンティストがモデルを再現するために一定期間後に実験を再訪するとき、一般的にソースデータは上流のパイプラインによって変更されています。

多くの場合、サイエンティストはそのような上流のデータ変更に気づかず、実験を再現するのに苦労します。一部のサイエンティストや組織では、データの複数のコピーを作成することでベストプラクティスを設計し、ストレージコストの増加につながっています。レポートを作成するアナリストにも同じことが言えます。

ロールバック

データパイプラインは、下流の消費者のために悪いデータを書き込んでしまうことがあります。これは、インフラストラクチャの不安定さ、データの乱雑さ、パイプラインのバグなどの問題が原因で起こることがあります。ディレクトリやテーブルへの単純な追加を行うパイプラインの場合、ロールバックは日付ベースのパーティショニングで簡単に対処できます。 更新や削除では、これは非常に複雑になる可能性があり、データエンジニアは通常、このようなシナリオに対処するために複雑なパイプラインを設計しなければなりません。

タイムトラベルとの連携

Delta Lake のタイムトラベル機能は、上記のユースケースに対応したデータパイプラインの構築を簡素化します。Delta Lake のタイムトラベルは、開発者の生産性を飛躍的に向上させます。

- データサイエンティストが実験をよりよく管理する
- データエンジニアがパイプラインを簡素化し、不良書き込みをロールバックする
- データアナリストは簡単なレポート作成を行う

アナリティクスのための独自のクラウドストレージで、クリーンで一元化されたバージョンのビッグデータリポジトリをようやく標準化できるようになりました。

Delta Lake のテーブルやディレクトリに書き込むと、すべての操作が自動的にバージョン化されます。異なるバージョンのデータにアクセスするには、 2 つの方法があります。

タイムスタンプを使う

Scala 構文：DataFrame リーダーのオプションとしてタイムスタンプまたは日付文字列を指定できます。

```
val df = spark.read
  .format("delta")
  .option("timestampAsOf", "2019-01-01")
  .load("/path/to/my/table")
```

Python 構文：

```
df = spark.read \
  .format("delta") \
  .option("timestampAsOf", "2019-01-01") \
  .load("/path/to/my/table")
```

SQL 構文：

```
SELECT count(*) FROM my_table TIMESTAMP AS OF "2019-01-01"
SELECT count(*) FROM my_table TIMESTAMP AS OF date_sub(current_date(), 1)
SELECT count(*) FROM my_table TIMESTAMP AS OF "2019-01-01 01:30:00.000"
```

リーダーコードがアクセスできないライブラリにあり、データを読み込むために入力パラメータをライブラリに渡している場合でも、パスに yyyyMMddHHmmssSSS 形式のタイムスタンプを渡すことで、テーブルのタイムトラベルを行うことができます。

```
val inputPath = "/path/to/my/table@20190101000000000"
val df = loadData(inputPath)
// Function in a library that you don't have access to
def loadData(inputPath : String) : DataFrame = {
  spark.read
    .format("delta")
    .load(inputPath)
}
inputPath = "/path/to/my/table@20190101000000000"
df = loadData(inputPath)

# Function in a library that you don't have access to
def loadData(inputPath):
  return spark.read \
    .format("delta") \
    .load(inputPath)
}
```

バージョン番号を使う

Delta Lake では、すべての書き込みにバージョン番号が付いており、バージョン番号を使って同様にタイムトラベルすることができます。

Scala 構文：

```
val df = spark.read
  .format("delta")
  .option("versionAsOf", "5238")
  .load("/path/to/my/table")

val df = spark.read
  .format("delta")
  .load("/path/to/my/table@v5238")
```

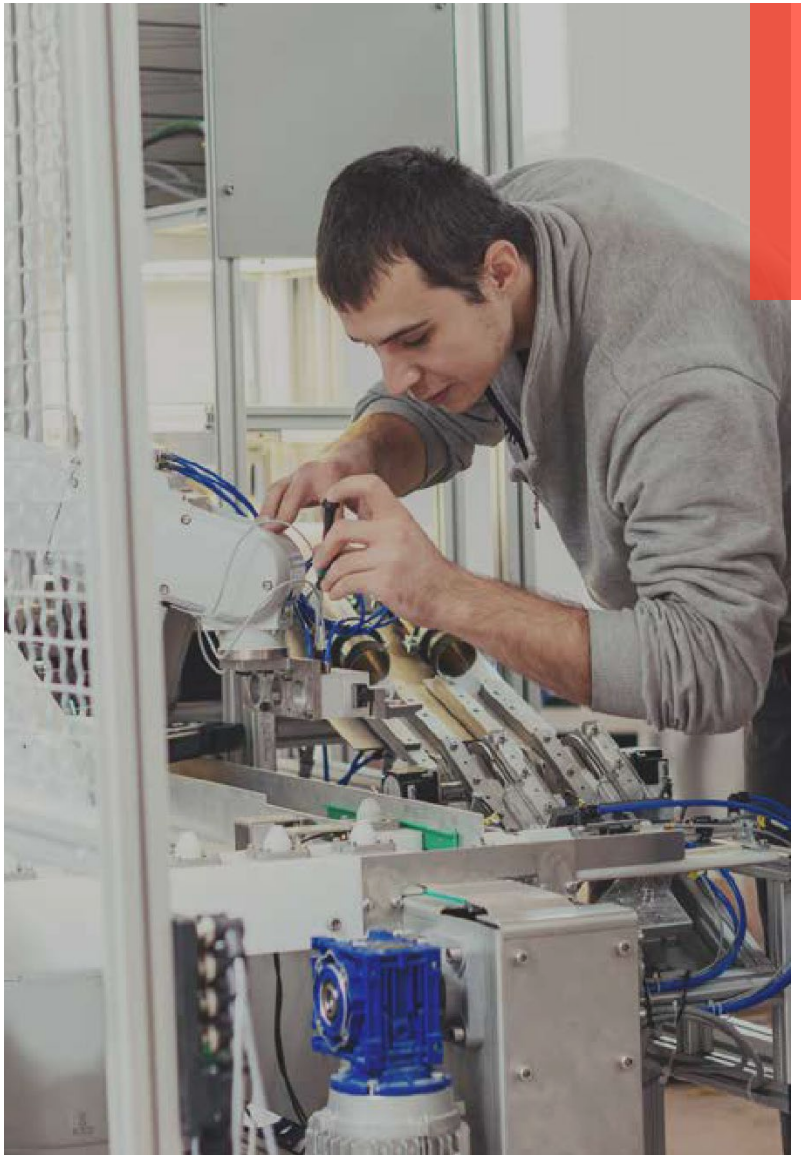
Python 構文：

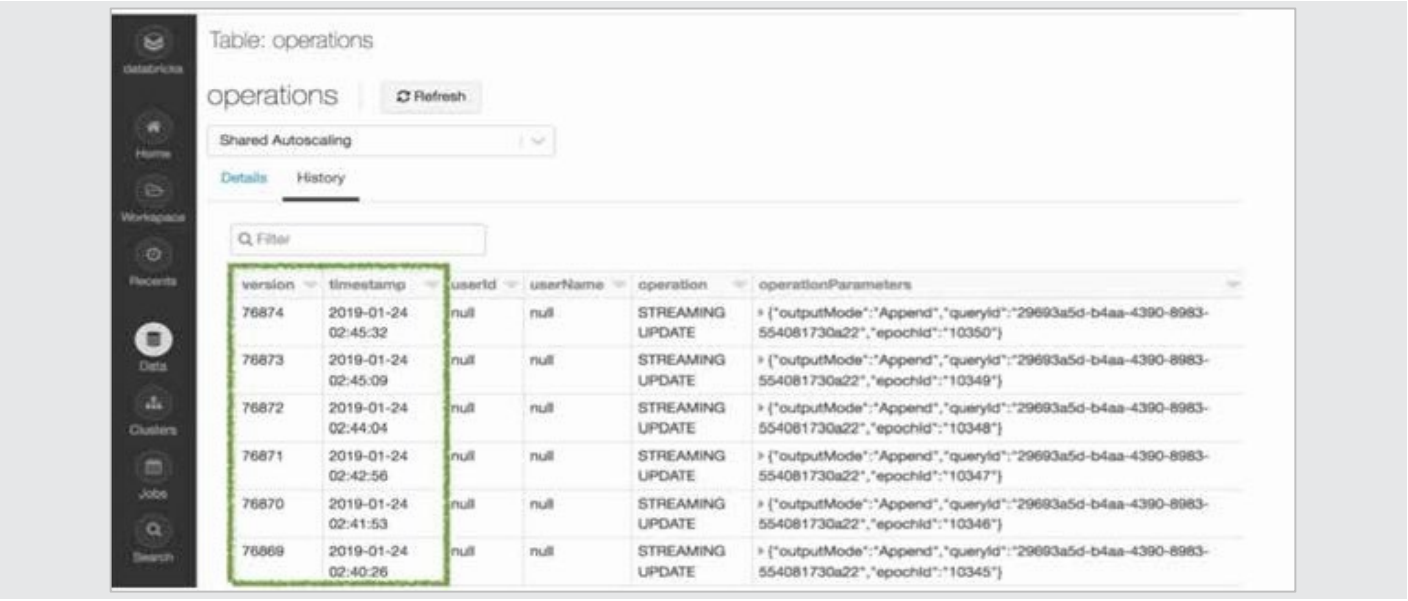
```
df = spark.read \
.format("delta") \
.option("versionAsOf", "5238") \
.load("/path/to/my/table")

df = spark.read \
  .format("delta") \
  .load("/path/to/my/table@v5238")
```

SQL 構文：

```
SELECT count(*) FROM my_table VERSION AS OF 5238
```





監査データの変更

DESCRIBE HISTORY コマンド、または UI からテーブルの変更履歴を参照ができます。

実験・レポートの再現

タイムトラベルは、機械学習やデータサイエンスにおいても重要な役割を果たしています。 モデルや実験の再現性は、データサイエンティストにとって重要な要件です。1つのモデルを本番運用するまでには何百ものモデルを作成し、その過程で、以前のモデルに戻りたいと思うからです。しかし、データ管理はデータサイエンスツールとは別物であることが多いため、これを実現するのは難しいのです。

Databricks は、Delta Lake のタイムトラベル機能を機械学習ライフサイクルのためのオープンソースのプラットフォームである MLflow と統合することで、この再現性の問題を解決します。 再現性のある機械学習トレーニングのために、MLflow のパラメータとしてパスへのタイムスタンプ付き URL をログに記録するだけで、各トレーニングジョブでどのバージョンのデータが使用されたかを追跡できます。

これにより、以前の設定やデータセットに戻って、以前のモデルを再現できます。データについて上流のチームと調整したり、異なる実験のためにデータを複製することを心配する必要はありません。これがユニファイドアナリティクスの力であり、データサイエンスとデータエンジニアリングが密接に結びついています。

ロールバック

また、タイムトラベルを利用することで、不正な書き込みがあった場合のロールバックも簡単に行うことができます。例えば、GDPR のパイプラインジョブでユーザー情報を誤って削除してしまうバグがあった場合、パイプラインを簡単に修正することができます。

```
INSERT INTO my_table
SELECT * FROM my_table TIMESTAMP AS OF date_sub(current_date(), 1)
```

```
WHERE userId = 111
You can also fix incorrect updates as follows:
MERGE INTO my_table target
USING my_table TIMESTAMP AS OF date_sub(current_date(), 1) source
ON source.userId = target.userId
WHEN MATCHED THEN UPDATE SET *
```

単にテーブルを以前のバージョンにロールバックしたい場合は、以下のいずれかのコマンドで行うことができます。

```
RESTORE TABLE my_table VERSION AS OF [version_number]
RESTORE TABLE my_table TIMESTAMP AS OF [timestamp]
```

ダウンストリームの複数のジョブにまたがって継続更新される Delta Lake テーブルのピン留めされたビュー

AS OF クエリを使用すると、継続的に更新される Delta Lake テーブルのスナップショットを複数のダウンストリーム ジョブにピン留めすることができるようになりました。例えば、Delta Lake テーブルが 15 秒ごとなどに継続的に更新されており、この Delta Lake テーブルから定期的に読み込んで異なるデスティネーションを更新するダウンストリームのジョブがある場合を考えてみましょう。 このようなシナリオでは、通常、すべてのデスティネーションテーブルが同じ状態を反映するように、ソースの Delta Lake テーブルを一貫して表示する必要があります。

このようなシナリオを以下のように簡単に扱えるようになりました。

```
version = spark.sql("SELECT max(version) FROM (DESCRIBE HISTORY
my_table)").collect()

# Will use the latest version of the table for all operations
below

data = spark.table("my_table@v%s" % version[0][0]data.where
("event_type = e1").write.jdbc("table1")
```

```
data.where("event_type = e2").write.jdbc("table2")
...
data.where("event_type = e10").write.jdbc("table10")
```

時系列分析のためのクエリがシンプルに

タイムトラベルは時系列分析を単純化します。例えば、先週何人の新規顧客を追加したかを知りたい場合、クエリは次のような非常にシンプルなものになります。

```
SELECT count(distinct userId) - (
SELECT count(distinct userId)
FROM my_table TIMESTAMP AS OF date_sub(current_date(), 7))
FROM my_table
```

その他のリソース

- 技術トーク（動画）：Delta Lake の詳細トランザクションログの解凍
- 技術トーク（動画）：Delta Lake と MLflow でデータサイエンスの準備
- Data + AI Summit Europe 2020（動画）：Delta タイムマシンによるデータタイムトラベル
- Spark + AI Summit NA 2020（動画）：MLflow と Delta Lake を用いた機械学習データの系譜
- Delta Lake を使用した機械学習の実運用化

Chapter

04

Delta Lake の容易なクローン化による テスト、共有、ML の再現性

04

Delta Lake の容易なクローン化による テスト、共有、ML の再現性

Delta Lake にはテーブルクローニング（Table Cloning）という機能があり、ML の再現性を高めるためのテーブルのテスト、共有、再作成を簡単に行うことができます。データレイクやデータウェアハウスでテーブルのコピーを作成することは、いくつかの実用的な用途があります。しかし、データレイク内のテーブルのデータ量とその成長速度を考えると、テーブルの物理的なコピーを作成するのはコストのかかる作業になります。

Delta Lake は現在、テーブルクローンを使用して、プロセスをよりシンプルにし、費用対効果の高いものにしています。

クローンとは？

クローンは、ある時点でのソーステーブルの複製です。クローンはソーステーブルと同じメタデータを持っています。同じスキーマ、制約、カラム記述、統計、パーティショニング。しかし、それらは別の系統や履歴を持つ別のテーブルとして動作します。クローンに加えられた変更は、クローンにのみ影響し、ソースには影響しません。クローン処理中や処理後にソースに発生した変更も、スナップショット分離によりクローンには反映されません。Delta Lake では、シャローとディープの 2 種類のクローンがあります。

シャロークローン

シャロー（ゼロコピーとも呼ばれる）クローンは、クローンされるテーブルのメタデータを複製するだけで、テーブル自体のデータファイルはコピーされません。このタイプのクローンは、データの物理的なコピーを作成しないので、ストレージコストを最小限に抑えることができます。 シャロークローンは安価で、非常に高速に作成することができます。

これらのクローンは自己完結型ではなく、クローンを作成したソースをデータのソースとして依存しています。クローンが依存しているソース内のファイルが削除された場合、例えばVACUUMでは、シャロークローンは使用できなくなる可能性があります。 そのため、シャロークローンは通常、テストや実験のような短期的なユースケースで使用されます。

ディープクローン

シャロークローンは短期的なユースケースには最適ですが、一部のシナリオではテーブルのデータの独立したコピーが必要になります。ディープクローンは、クローンされるテーブルのメタデータとデータファイルの完全なコピーを作成します。その意味では、CTASコマンドによるコピー（CREATE TABLE... AS... SELECT... ）と機能的には似ています。 しかし、指定されたバージョンで元のテーブルの忠実なコピーを作成するので、指定がより簡単になり、CTASのようにパーティショニングや制約などの情報を再指定する必要がなくなります。また、はるかに高速でロバストであり、障害に対してもインクリメンタルに動作することができます。

ディープクローンでは、ストリーミングアプリケーションのトランザクションやCOPY INTOトランザクションなどの追加メタデータをコピーします。

クローンはどこで役に立つの？

時々、家事や手品を手伝ってくれるクローンがいたらいいのにと思うことがあります。しかし、ここでは人間のクローンの話をしているのではありません。MLモデルや分析クエリの探索、共有、テストのためにデータセットのコピーが必要なシナリオはたくさんあります。次に、顧客のユースケースの例をいくつか紹介します。

本番用のテーブルを使ったテストと実験

ユーザーがデータパイプラインの新バージョンをテストする必要がある場合、本番環境のすべてのデータを代表するものではないサンプルのテストデータセットに頼らなければならないことがよくあります。また、データチームは、大規模なテーブルに対するクエリの性能を向上させるために、さまざまなインデックス作成技術を実験したい場合もあります。これらの実験やテストは、本番環境では、本番データのプロセスを危険にさらしたり、ユーザーに影響を与えたりすることなく実施することはできません。テストや開発環境用に本番テーブルのコピーをスピンアップするには、何時間も、あるいは何日もかかることがあります。

さらに、複製されたすべてのデータを保持するための開発環境のための余分なストレージコストがかかります。 本番データを反映したテスト環境の設定には大きなオーバーヘッドがあります。シャロークローンであれば、これは些細なことです。

```
-- SQL
CREATE TABLE delta.`/some/test/location` SHALLOW CLONE prod.events

# Python
DeltaTable.forName("spark", "prod.events").clone("/some/test/location",
isShallow=True)

// Scala
DeltaTable.forName("spark", "prod.events").clone("/some/test/location",
isShallow=true)
```

数秒でテーブルのシャロークローンを作成後は、パイプラインのコピーを実行して新しいコードのテストや、異なるディメンションでテーブルを最適化してクエリの性能を向上させる方法の試行、その他多くのことができます。これらの変更は、元のテーブルではなく、シャロークローンにのみ影響します。

本番テーブルへの主要な変更のステージング

時には、生産テーブルに大きな変更を加えなければならないことがあります。 これらの変更は多くのステップで構成されている場合があり、すべての作業が完了するまでは、他のユーザーに変更内容を見せたくないでしょう。ここでは、シャロークローンが役立ちます。

```
-- SQL
CREATE TABLE temp.staged_changes SHALLOW CLONE prod.events;
DELETE FROM temp.staged_changes WHERE event_id is null;
UPDATE temp.staged_changes SET change_date = current_date()
WHERE change_date is null;
...
-- Perform your verifications
```

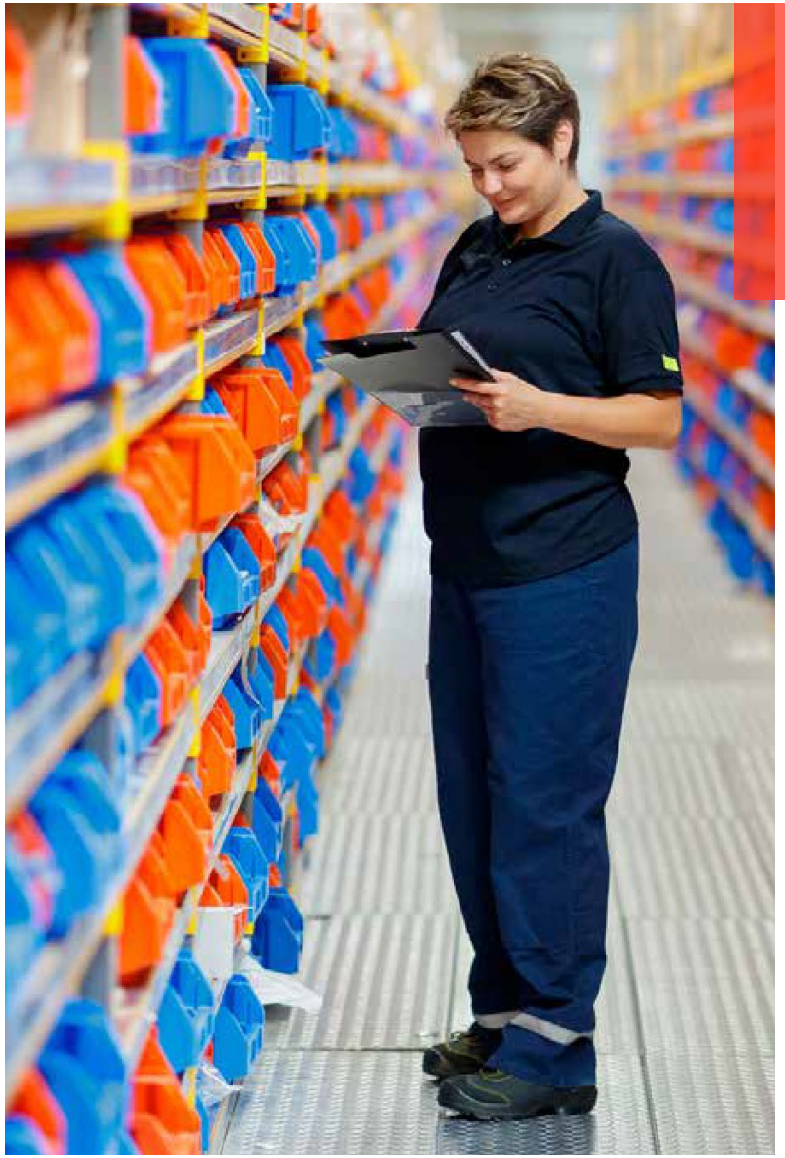
結果に満足したら、2 つの選択肢があります。ソース テーブルに他の変更が加えられていない場合は、ソーステーブルをクローンで置き換えることができます。 ソーステーブルに変更が加えられている場合は、その変更をソーステーブルに MERGE できます。

```
-- If no changes have been made to the source
REPLACE TABLE prod.events CLONE temp.staged_changes;
-- If the source table has changed
MERGE INTO prod.events USING temp.staged_changes
ON events.event_id <=> staged_changes.event_id
WHEN MATCHED THEN UPDATE SET *;
-- Drop the staged table
DROP TABLE temp.staged_changes;
```

機械学習結果の再現性

効果的な ML モデルを考えることは，反復的なプロセスです．モデルのさまざまな部分を調整するこのプロセスを通して、データサイエンティストは固定データセットに対してモデルの精度を評価する必要があります。

これは、データが常にロードされたり更新されたりしているシステムでは難しいことです。モデルの訓練とテストに使用されたデータのスナップショットが必要です。このスナップショットにより、テストやモデルガバナンスの目的でMLモデルの結果を再現可能にできます。



タイムトラベルを活用して、スナップショットをまたいで複数の実験を実行することをお勧めします。これが実際に行われている例は、Machine Learning Data Lineage With MLflowとDelta Lakeで見ることができます。

結果に満足して、次のブラックフライデーなどの後の検索のためにデータをアーカイブしたいと思ったら、ディープクローンを使ってアーカイブプロセスを簡素化することができます。 MLflow は Delta Lake と非常によく統合されており、オートログ機能 （mlflow.spark.autolog()）を使えば、ある実験の実行に使われたテーブルのバージョンを知ることができます。

```
# Run your ML workloads using Python and then
DeltaTable.forName(spark, "feature_store").cloneAtVersion(128, "feature_
store_bf2020")
```

データの移行

大規模なテーブルは、性能やガバナンス上の理由から、新しい専用のバケットやストレージシステムに移動する必要があるかもしれません。元のテーブルは今後新しいアップデートを受けられず、将来の時点で非アクティブ化されて削除されます。 ディープクローンは、マッシブテーブルのコピーをより強固でスケーラブルなものにします。

```
-- SQL
CREATE TABLE delta.`zz://my-new-bucket/events` CLONE prod.events;
ALTER TABLE prod.events SET LOCATION 'zz://my-new-bucket/events';
```

ディープクローンでは、ストリーミングアプリケーションのトランザクションと COPY INTO トランザクションをコピーしますので、この移行後も ETL アプリケーションを離脱した場所から正確に継続すできます。

データの共有

組織では、異なる部門のユーザーが分析やモデルを充実させるために使用できるデータセットを探していることがよくあります。組織内の他のユーザーとデータを共有したい場合もあるでしょう。しかし、データを別のストアに移動させるために精巧なパイプラインを設定するよりも、関連するデータセットのコピーを作成して、ユーザーがデータを探索したりテストしたりして、自社の本番システムに影響を与えずにニーズに合っているかどうかを確認する方が、簡単で経済的であることが多いのです。ここで再びディープクローンの登場です。

```
-- The following code can be scheduled to run at your convenience
CREATE OR REPLACE TABLE data_science.events CLONE prod.events;
```

データのアーカイブ

規制やアーカイブの目的のために、テーブル内のすべてのデータを一定期間保存する必要がありますが、アクティブなテーブルは数ヶ月間データを保持します。データをできるだけ早く更新したいが、数年分のデータを保持する必要がある場合、このデータを1つのテーブルに保存してタイムトラベルを行うと、法外なコストがかかる場合があります。

この場合、日次、週次、月次の方法でデータをアーカイブすることがより良い解決策となります。ここでは、ディープクローンのインクリメンタルクローニング機能が役立ちます。

```
-- The following code can be scheduled to run at your convenience
CREATE OR REPLACE TABLE archive.events CLONE prod.events;
```

このテーブルはソーステーブルと比較して独立した履歴を持つことに注意してください。そのため、ソーステーブルとクローンに対するタイムトラベルクエリは、アーカイブの頻度に応じて異なる結果を返す可能性があります。

よさそうだが、何か落とし穴はないのか？

前述したいくつかをリストとして繰り返しますが、ここでは注意すべきことを挙げておきます。

- クローンはデータのスナップショット上で実行されます。クローン処理の開始後にソーステーブルに加えられた変更は、クローンには反映されません。
- シャロークローンは、ディープクローンのように自己完結型のテーブルではありません。ソーステーブル内でデータが削除された場合（例えば VACUUM を介して）、シャロークローンは使用できない場合があります。
- クローンは、ソース テーブルとは別個の独立した履歴を持ちます。ソース テーブルとクローンに対するタイムトラベル クエリは、同じ結果を返さない場合があります。
- シャロークローンは、ストリームトランザクションをコピーしたり、メタデータを COPY INTO することはありません。ディープクローンを使用してテーブルを移行し、終わったところから ETL プロセスを続行します。

どのように使えばいいのでしょうか？

シャロークローンとディープクローンは、データチームが最新のクラウドデータレイクとウェアハウスをテストして管理する方法の新たな進歩をサポートします。テーブルクローンは、パイプラインの本番レベルのテストの実施、最適なクエリ性能のためのインデックス作成の微調整、共有用のテーブルコピーの作成などを、最小限のオーバーヘッドと費用で行うことができます。あなたの組織で必要とされているのであれば、ぜひ一度テーブルクローンを試してみて、フィードバックをお願いします。

その他のリソース

- Databricks の Delta Lake で DR にディープクローンを使用する
- SPARK + AI Summit 2020 : Delta Lake を使った DR の簡素化

Chapter

05

Apache Spark 3.0 の Delta Lake で Spark SQL DDL と DML を使用する

05

Apache Spark 3.0 の Delta Lake で Spark SQL DDL と DML を使用する

Delta Lake 0.7.0 のリリースは Apache Spark 3.0 のリリースと重なったため、 SQL から Delta Lake を使用して簡素化された新しい機能群が可能になりました。主な機能をいくつか紹介します。

SQL DDL コマンドのサポートで Hive メタストアにテーブルを定義

Hive メタストアでデルタ テーブルを定義し、テーブルの作成（または置換）時にすべての SQL 操作でテーブル名を使用できるようになりました。

テーブルの作成・置換

```
-- Create table in the metastore
CREATE TABLE events (
   date DATE,
   eventId STRING,
   eventType STRING,
   data STRING)
USING DELTA
PARTITIONED BY (date)
LOCATION '/delta/events'
-- If a table with the same name already exists, the table is replaced
with
the new configuration, else it is created
CREATE OR REPLACE TABLE events (
```

```
  date DATE,
  eventId STRING,
  eventType STRING,
  data STRING)
USING DELTA
PARTITIONED BY (date)
LOCATION '/delta/events'
```

テーブルスキーマを明示的に変更する

```
-- Alter table and schema
ALTER TABLE table_name ADD COLUMNS (
  col_name data_type
    [COMMENT col_comment]
    [FIRST|AFTER colA_name],
  ...)
```

また、Scala／Java／Python の API を利用することもできます。

- `DataFrame.saveAsTable(tableName)`、`DataFrameWriterV2` API (#307)
- Scala／Java／Python で Update／Delete／Merge 操作を実行するのに便利な、`io.delta.tables .DeltaTable` インスタンスを作成するための `DeltaTable.forName(tableName)` API

SQLの挿入、削除、更新、MERGEのサポート

Delta Lake 技術 トークでよくある質問は、削除、更新、MERGEなどの DML 操作はいつ Spark SQL で使えるようになるのかというものでした。 SQL でこれらの操作ができるようになりました。次は、削除、更新、MERGE（Spark SQL を使用した挿入、更新、削除、重複排除操作）の書き方の例です。

```
-- Using append mode, you can atomically add new data to an existing
Delta table
INSERT INTO events SELECT * FROM newEvents
-- To atomically replace all of the data in a table, you can use
overwrite mode
INSERT OVERWRITE events SELECT * FROM newEvents

-- Delete events
DELETE FROM events WHERE date < '2017-01-01'

-- Update events
UPDATE events SET eventType = 'click' WHERE eventType = 'click'

-- Upsert data to a target Delta
-- table using merge  MERGE INTO events
USING updates
  ON events.eventId = updates.eventId
WHEN MATCHED THEN UPDATE
    SET events.data = updates.data
WHEN NOT MATCHED THEN INSERT (date, eventId, data)
    VALUES (date, eventId, data)
```

Delta Lake の MERGE 操作は、標準の ANSI SQL 構文よりも高度な構文をサポートしていることは注目に値します。例えば、MERGE は

- Delete アクション：ソース行と一致した場合にターゲットを削除します。
  例："... WHEN MATCHED THEN DELETE ..."
- 句の条件で複数の一致したアクションを指定可能：ターゲット行とソース行が一致する場合の柔軟性が高まります。例えば次のようになります。

```
...
WHEN MATCHED AND events.shouldDelete THEN DELETE
WHEN MATCHED THEN UPDATE SET events.data = updates.data
```

- Star 構文： 類似した名前のソース列でターゲット列の値を設定するための略記法。例えば、以下のようになります。

```
WHEN MATCHED THEN SET *
WHEN NOT MATCHED THEN INSERT *
-- equivalent to updating/inserting with event.date = updates.date,
   events.eventId = updates.eventId, event.data = updates.data
```

自動およびインクリメンタルな Presto／Athena のマニフェスト生成

Delta Lake は、マニフェスト ファイルを使用して他の処理エンジンが Delta Lake を読み取ることをサポートしています（Presto および Athena からの Delta Lake テーブルのクエリ、オペレーションの同時実行性の向上、および MERGE 性能を参照）。前章で説明したように、次のことを行う必要があります。

- Delta Lakeマニフェスト ファイルの生成
- 生成されたマニフェストを読み込むために Presto または Athena を設定する
- マニフェストファイルを手動で再生成（更新）する

Delta Lake 0.7.0 の新機能として、以下のコマンドでマニフェストファイルを自動的に更新する機能が追加されました。

```
ALTER TABLE delta.`pathToDeltaTable`
SET TBLPROPERTIES(
delta.compatibility.symlinkFormatManifest.enabled=true
)
```

テーブルのプロパティからテーブルを設定する

ALTER TABLE SET TBLPROPERTIES を使用してテーブルのプロパティを設定することで、自動マニフェスト生成などの Delta Lake の多くの機能を有効にしたり、無効にしたり、設定したりすることができます。例えば、テーブルのプロパティを使用して、`delta.appendOnly=true` を使用して、Delta テーブル内の削除や更新をブロックすることができます。

また、次のプロパティを使用して、Delta Lake のテーブル保持の履歴を簡単に制御できます。

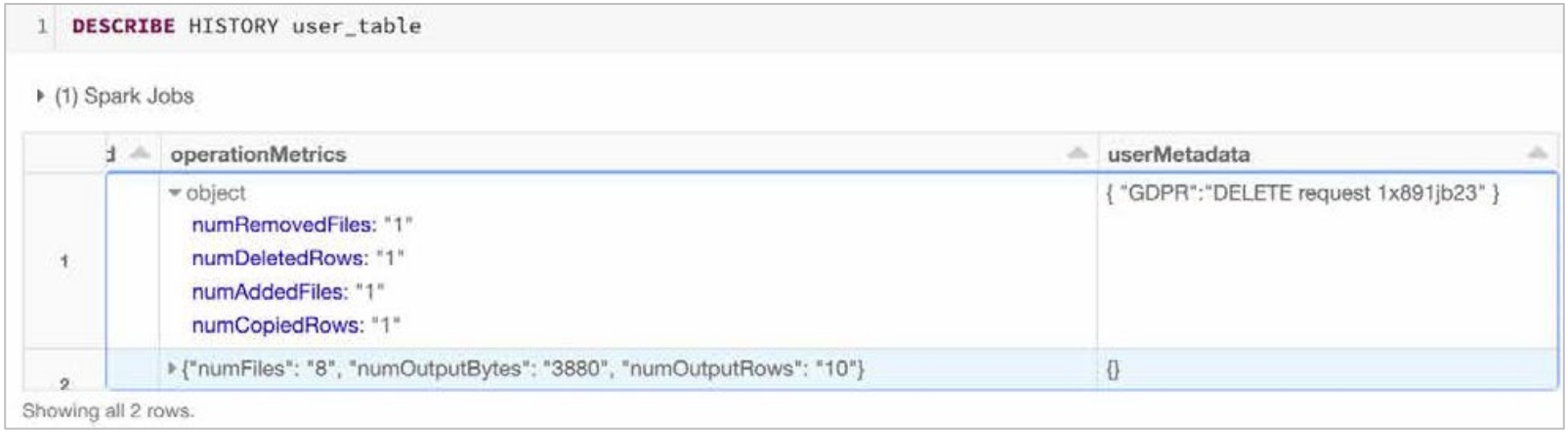
- `delta.logRetentionDuration`：テーブルの履歴（トランザクション ログ履歴など）が保持される期間を制御します。デフォルトでは 30 日間の履歴が保持されますが、要件（GDPR の歴史的背景など）に応じてこの値を変更することもできます。
- `delta.deletedFileRetentionDuration`：VACUUM の候補となる前にファイルを削除する必要がある期間を制御します。デフォルトでは、7 日以上前のデータ ファイルが削除されます。

Delta Lake 0.7.0 の時点では、ALTER TABLE SET TBLPROPERTIES を使用してこれらのプロパティを設定することができます。

```
ALTER TABLE delta.`pathToDeltaTable`
SET TBLPROPERTIES(
  delta.logRetentionDuration = "interval "
  delta.deletedFileRetentionDuration = "interval "
)
```

Delta Lake テーブルコミットでユーザー定義メタデータを追加

Delta Lake のテーブル操作によって行われるコミットでは、DataFrameWriter オプションの userMetadata を使用するか、 SparkSession 設定の `spark.databricks.delta.commitInfo.userMetadata` を使用して、メタデータとしてユーザー定義の文字列を指定できます。



次の例では、ユーザーのリクエストごとにデータレイクからユーザー(1xsdf1)を削除しています。ユーザーのリクエストと削除を確実に関連付けるために、userMetadata に DELETE リクエスト ID を追加しています。

```
SET spark.databricks.delta.commitInfo.userMetadata={
  "GDPR":"DELETE Request 1x891jb23"
};
DELETE FROM user_table WHERE user_id = '1xsdf1'
```

ユーザーテーブル（user_table）の履歴操作を確認する際に、トランザクションログ内で関連する削除要求を簡単に特定することができます。

その他のハイライト

Delta Lake 0.7.0 リリースのその他のハイライトは以下のとおりです。

- Azure Data Lake Storage Gen2 のサポート： Spark 3.0 では Hadoop 3.2 のライブラリがサポートされ、Azure Data Lake Storage Gen2 のサポートが可能になりました。
- ワンタイムトリガーのストリーミングサポートの改善：Spark 3.0では、DataStreamReader オプション maxFilesPerTrigger でレート制限が設定されている場合でも、ワンタイムトリガー（`Trigger.Once`）が1つのマイクロバッチで Delta Lake テーブル内のすべての未処理データを処理できるようになりました。

AMA では、構造化されたストリーミングと `trigger.once` の使用に関する多くの有意義な質問がありました。

以下の詳細には、この概念を説明するいくつかの良いリソースが含まれます。

- 1日1回のストリーミングジョブ実行でコストを 10 分の 1 に削減する方法
- ラムダを超えてデルタアーキテクチャの紹介（動画）で議論されているコストとレイテンシのトレードオフ

その他のリソース

- 技術トーク（動画）：Delta Lake 0.70+ Spark 3.0 AMA
- 技術トーク（動画）：Apache Spark 3.0 + Delta Lake
- ブログ：Apache Spark 3.0 ベースの Delta Lake で Spark SQL DDL と DML を使用する

次のステップ

この eBook では、Delta Lake とその機能が性能を向上させる仕組みについて解説しました。このシリーズの他の eBook では、Delta Lake のリソースを詳しくご紹介しています。

この eBook の後続シリーズ

- Delta Lake シリーズ：基礎と性能
- Delta Lake シリーズ：レイクハウス
- Delta Lake シリーズ：ストリーミング
- Delta Lake シリーズ：顧客ユースケース

Delta Lake をさらに詳しく

- 技術トークシリーズ：Delta Lake 基本
- 技術トークシリーズ：Delta Lake 詳細
- Databricks の Web サイト
- Databricks の無料トライアル
- Web セミナー：Delta Lake オープンソースの信頼性