ALIBABA CLOUD

# 阿里云

## Databricks 数据洞察
## Databricks Runtime引擎

文档版本：20201222

阿里云

# 法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

# 通用约定

| 格式 | 说明 | 样例 |
| --- | --- | --- |
| 危险 | 该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。 | 危险<br>重置操作将丢失用户配置数据。 |
| 警告 | 该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。 | 警告<br>重启操作将导致业务中断，恢复业务时间约十分钟。 |
| 注意 | 用于警示信息、补充说明等，是用户必须了解的内容。 | 注意<br>权重设置为0，该服务器不会再接受新请求。 |
| 说明 | 用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。 | 说明<br>您也可以通过按Ctrl+A选中全部文件。 |
| > | 多级菜单递进。 | 单击**设置**> **网络**> **设置网络类型**。 |
| **粗体** | 表示按键、菜单、页面名称等UI元素。 | 在**结果确认**页面，单击**确定**。 |
| Courier字体 | 命令或代码。 | 执行 `cd /d C:/window` 命令，进入Windows系统文件夹。 |
| *斜体* | 表示参数、变量。 | `bae log list --instanceid`<br>*Instance_ID* |
| [] 或者 [a\|b] | 表示可选项，至多选择一个。 | `ipconfig [-all\|-t]` |
| {} 或者 {a\|b} | 表示必选项，至多选择一个。 | `switch {active\|stand}` |

# 目录

# 1.Databricks Runtime引擎特性

## Introduction

The Databricks Runtime is a data processing engine built on a highly optimized version of Apache Spark, for up to 50x performance gains. It runs on Ali-Cloud ECS infrastructure for easy self-service without DevOps, while also providing security and administrative controls needed for production. Build pipelines, schedule jobs, and train models faster than before.

## Benefits

### PERFORMANCE

The Databricks Runtime has been highly optimized by the original creators of Apache Spark. The significant increase in performance enables new use cases not previously possible for data processing and pipelines and improves data team productivity.

### COST EFFECTIVE

The runtime leverages auto-scaling compute and storage to manage infrastructure costs. Clusters intelligently start and terminate, and the high cost-to-performance efficiency reduces infrastructure spend.

### SIMPLICITY

Databricks has wrapped Spark with a suite of integrated services for automation and management to make it easier for data teams to build and manage pipelines, while giving IT teams administrative control.

## Features

**Caching:** Copies of remote files are cached in local storage using a fast intermediate data format, which results in improved successive read speeds of the same data.

**Z-Order Clustering:** The colocation of related information in the same set of files dramatically reduces the amount of data that needs to be read, resulting in faster query responses.

**Join Optimizations:** Significant performance gains are possible with range join and skew join optimizations through different query patterns and skew hints.

**Data Skipping:** Statistical information on minimum and maximum values automatically collected when data is written is used at query time to provide faster queries.

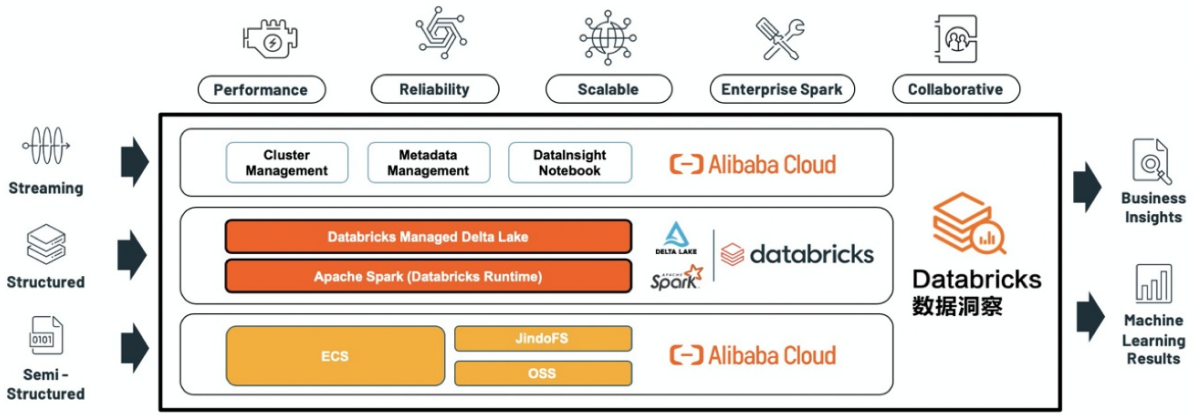
**Easy-to-Use Cluster Management:** User-friendly user interface simplifying the creation, restarting, and termination of clusters, providing increased visibility to your clusters for easier manageability and cost control.

**High Availability:** The Databricks cluster manager transparently relaunches any worker instance that is revoked or crashes, ensuring your service is always up and running without the need to manage it yourself.

## How It Works

## Databricks Runtime

The Databricks Runtime implements the open Apache Spark APIs with a highly optimized execution engine, which provides significant performance gains compared to standard open source Apache Spark found on other cloud Spark platforms. This core engine is then wrapped with additional services for developer productivity and enterprise governance.




## More Information

For more information about Databricks Runtime, please refer to official document: https://databricks.com/product/databricks-runtime

# 2.Databricks Runtime

Databricks Runtimes是在Databricks群集上运行的一组核心组件。Databricks提供了几种类型的Runtime。

| | |
|---|---|
| **Databricks Runtime**<br>Databricks Runtime包括Apache Spark，但还添加了许多组件和更新，这些组件和更新极大地提高了大数据分析的可用性，性能和安全性。 | **用于机器学习的Databricks Runtime（敬请期待）**<br>Databricks Runtime ML是Databricks Runtime的变体，它添加了多个流行的机器学习库，包括TensorFlow，Keras，PyTorch和XGBoost。 |
| **用于基因组的Databricks Runtime （敬请期待）**<br>用于基因组的 Databricks Runtime 是Databricks Runtime的一种变体，已针对基因组和生物医学数据进行了优化。 | **Databricks Light（敬请期待）**<br>Databricks Light 为不需要Databricks Runtime提供的高级性能、可靠性或自动缩放优势的作业提供了运行时选项。 |

## Databricks Runtime

Databricks Runtime包括Apache Spark，但还添加了许多组件和更新，这些组件和更新大大改善了大数据分析的可用性，性能和安全性：

- Delta Lake是在Apache Spark之上构建的下一代存储层，可提供ACID事务，优化的布局和索引以及用于构建数据管道的执行引擎改进。
- 已安装的Java，Scala，Python和R库
- Ubuntu及其随附的系统库
- 适用于启用GPU的集群的GPU库
- 与平台的其他组件集成的Databricks服务，例如笔记本，作业和集群管理器

## Runtime 版本控制

Databricks Runtime 版本会定期发布：

- **主要版本**以小数点之前的版本号递增表示（例如，从3.5跳到4.0）。当发生重大更改时，它们会被释放，其中某些更改可能无法向后兼容。
- **功能部件版本**以小数点后的版本号递增表示（例如，从3.4跳到3.5）。每个主要版本均包含多个功能版本。功能版本始终与其主要版本中的先前版本向后兼容。
- **长期支持版本**由LTS限定符表示（例如3.5 LTS）。对于每个主要版本，Databricks 都会声明一个“规范”功能版本，并提供为期两年的支持。有关更多信息，请参见Databricks运行时支持生命周期。

> **说明**
>
> 当前Databrick 数据洞察支持版本为：DBR5.5，Spark2.4.3，Scala 2.11。

## 用于机器学习的Databricks Runtime

用于机器学习的Databricks Runtime（Databricks Runtime ML）自动创建针对机器学习优化的集群。Databricks Runtime ML集群包括最受欢迎的机器学习库，例如TensorFlow，PyTorch，Keras和XGBoost，还包括分布式培训所需的库，例如Horovod。使用Databricks Runtime ML可以加快群集的创建速度，并确保已安装的库版本兼容。

### 用于机器学习的Databricks Runtime简介

本教程是为Databricks Runtime ML的新用户设计的。完成此过程大约需要10分钟，并显示了一个完整的端到端示例，该示例包含加载表格数据，训练模型，分布式超参数调整和模型推断的示例。它还说明了如何使用MLflow API和MLflow模型注册表。

### Databricks教程Note

Note链接地址：ML End-to-End Example (AWS)

### Databricks Runtime ML中包含的库

> 注意
>
> Library utilities are not available in Databricks Runtime ML.

Databricks Runtime ML包括各种流行的ML库。该库随每个发行版进行更新，以包括新功能和修复。

Databricks已将受支持的库的子集指定为顶级库。对于这些库，Databricks提供了更快的更新节奏，可以在每个运行时版本中更新到最新的程序包版本（除非存在依赖冲突）。Databricks还为顶级库提供高级支持，测试和嵌入式优化。

有关顶级库和其他提供的库的完整列表，请参见以下有关每个可用Runtime的文章：

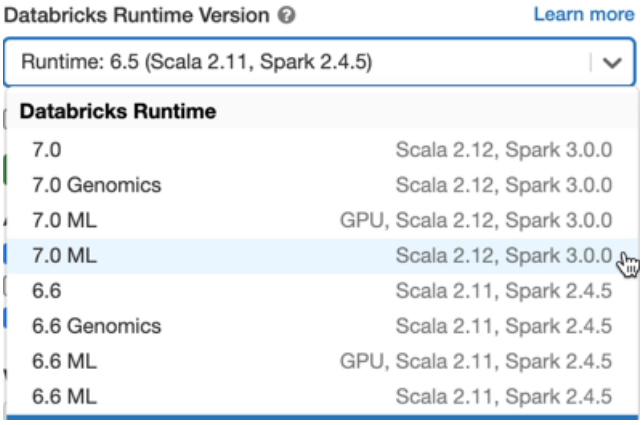
- Databricksr Runtime 7.4 ML（测试版）
- Databricks Runtime 7.3 LTS ML
- Databricks Runtime 7.2 ML
- Databricks Runtime 7.1 ML
- Databricks Runtime 7.0 ML
- Databricks Runtime 6.6 ML
- Databricks Runtime 6.5 ML（不受支持）
- Databricks Runtime 6.4 ML
- Databricks Runtime 5.5 LTS ML

### 如何使用Databricks Runtime ML

除了预安装的库之外，Databricks Runtime ML在群集配置以及如何管理Python包方面与Databricks Runtime不同。

### 使用Databricks Runtime ML创建集群

当你创建一个集群，请从Databricks运行时版本下拉列表中Databricks运行ML版本。支持CPU和GPU的ML运行时均可用。




如果选择支持GPU的ML运行时，系统将提示您选择兼容的**Driver Type**和**Worker Type**。下拉列表中不兼容的实例类型显示为灰色。GPU**加速**标签下列出了启用GPU的实例类型。

> 说明
>
> 当前Dat abrick 数据洞察支持版本为：DBR5.5，Spark2.4.3，Scala 2.11。

**管理Pyt hon套件**

在Dat abricks Runt ime ML中，Conda软件包管理器用于安装Python软件包。所有Pyt hon软件包都安装在一个环境中：/dat abricks/pyt hon2在使用Pyt hon2的/dat abricks/pyt hon3集群上和在使用Python3的集群上。不支持切换（或激活）Conda环境。

**AutoML支持**

Dat abricks Runt ime ML包括可自动进行模型开发过程并帮助您有效地找到最佳性能模型的工具。

- 托管MLFlow管理端到端模型生命周期，包括跟踪实验运行，部署和共享模型以及维护集中式模型注册表。
- 通过SparkTrials类增加的Hyperopt自动执行并分发ML模型参数调整。

## Databricks Runtime for Genomics

Dat abricks Runt ime for Genomics（Dat abricks Runt ime Genomics）是Dat abricks Runt ime的一个版本，该版本针对处理基因组和生物医学数据进行了优化。它是Dat abricks基因组统一分析平台的一部分。

从版本6.0开始，通常可以使用（GA）Dat abricks Genomics Runt ime。

**Dat abricks Runt ime for Genomics中有什么？**

- Dat abricks-Regeneron开源库Glow的优化版本，具有所有功能以及：
  - Spark SQL支持读取和写入变量数据
  - 通用工作流程元素的功能
  - 常见查询模式的优化
- 与Apache Spark并行的Turn-key pipelines：
  - DNA序列
  - RNA序列
  - 肿瘤正常测序（MutSeq）
  - 联合基因分型

- SnpEff variant annotation

- Hail 0.2 integration
- 常用的开源库，针对性能和可靠性进行了优化：
  - ADAM
  - GATK
  - Hadoop-bam
- 常用的命令行工具：
  - samtools
- 参考数据（grch37或38，已知SNP位点）

**要求**
您的Databricks工作区必须启用了用于基因组学的Databricks Runtime 。
**使用Databricks Runtime for Genomics创建集群**
当你创建一个集群，请从Databricks Runtime版本下拉列表中Databricks Runtime的基因组版本。

## Databricks Light

Databricks Light是开源Apache Spark Runtime的Databricks打包。它为不需要Databricks Runtime提供的高级性能，可靠性或自动缩放优势的作业提供了运行时选项。特别是，Databricks Light不支持：

- Delta Lake
- 自动驾驶功能，如自动缩放
- 高度并发的通用集群
- 笔记本，仪表板和协作功能
- 各种数据源和BI工具的连接器

Databricks Light是作业（或“自动工作负载”）的Runtime环境。当您在Databricks轻型集群上运行作业时，它们会受到较低的作业轻计算定价的影响。只有在创建或计划JAR、Python或spark提交作业并将集群附加到该作业时，才能选择Databricks Light；不能使用Databricks Light运行笔记本作业或交互式工作负载。

Databricks Light可以与在其他Databricks Runtime和定价层上运行的集群一起在同一工作空间中使用。您无需请求单独的工作区即可上手。

### Databricks Light中有什么？

Databricks Light运行时的发布时间表遵循Apache Spark运行时的发布时间表。任何Databricks Light版本均基于特定版本的Apache Spark。有关更多信息，请参见以下发行说明：

- Databricks Light 2.4

### 使用Databricks Light创建集群

创建作业集群时，从Databricks运行时版本下拉列表中选择Databricks Light版本。

> 注意
>
> 有关Databricks的详细信息，请参考官方文档：Databricks Runtime

# 3.Delta Lake

## 3.1. Delta Lake 简介

Delta Lake 是可提高数据湖可靠性的开源存储层。Delta Lake 提供 ACID 事务和可缩放的元数据处理，并统一流和批数据处理。 Delta Lake 在现有 Data Lake 的顶层运行，与 Apache Spark API 完全兼容。

具体来说，Delta Lake提供：

- **Spark ACID事务**：可序列化的隔离级别确保用户永远不会看到不一致的数据。
- **可扩展的元数据处理**：利用Spark的分布式处理能力，可以轻松处理数十亿个文件的PB级表的所有元数据。
- **流式处理和批处理统一**：Delta Lake中的表既是批处理表，又是也是流式处理源和接收器。流式处理数据引入、批处理历史回填、交互式查询功能都是现成的。
- **架构强制**：自动处理架构变体，以防在引入过程中插入错误的记录。
- **按时间顺序查看**：数据版本控制支持回滚、完整的历史审核线索和可重现的机器学习试验
- **更新插入和删除**：支持合并、更新和删除操作，以启用复杂用例，如更改数据捕获、渐变维度 (SCD) 操作、流式处理更新插入等。

Delta Engine 优化使 Delta Lake 操作具有高性能，并支持各种工作负载，从大规模 ETL 处理到临时交互式查询均可。有关Delta Engine的信息，请参阅Delta Engine的相关文档。

### 开始

Delta Lake 快速入门概述了与Delta Lake相关的基础知识。该快速入门介绍了如何生成将 JSON 数据读取到 Delta 表中的管道以及如何修改表、读取表、显示表历史记录和优化表。

对于具有这些功能的Databricks笔记本，请参阅入门笔记本。

### 资源

- 有关常见问题的答案，请参阅常见问题（FAQ）。
- 有关Delta Lake SQL命令的参考信息，请参阅Delta Lake SQL语法。
- 有关更多资源，包括博客文章，讲座和示例，请参见参考资料。

> 注意
>
> 有关本文章详细信息，请参考Databricks官方文档：Delta介绍

## 3.2. Delta Lake 快速入门

Delta Lake快速入门概述了使用Delta Lake的基础知识。此快速入门演示如何生成管道，以便将JSON数据读入Delta表、修改表、读取表、显示表历史记录，以及优化表。

有关演示这些功能的Databricks笔记本，请参阅入门笔记本。

### 创建表

若要创建一个delta表，可以使用现有的Apache Spark SQL代码，也可以将parquet、csv、json等数据格式转换为delta。

对于所有文件类型，您将文件读入DataFrame并将格式转为delta：

Python

```
%pyspark
events = spark.read.json("/xz/events_data.json")
events.write.format("delta").save("/xz/delta/events")
spark.sql("CREATE TABLE events USING DELTA LOCATION '/xz/delta/events/'")
```


SQL

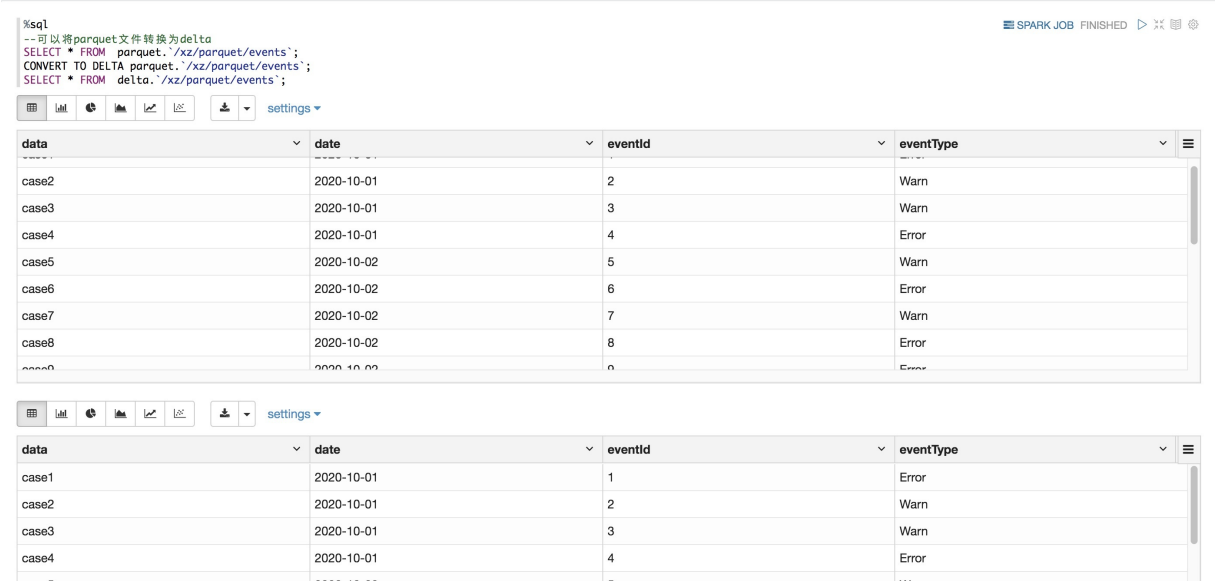
```
%sql
CREATE TABLE events
USING delta
AS SELECT *
FROM json.`/data/events/`
```

这些操作使用从JSON数据推断出的架构创建一个新的非托管表。有关创建新Delta表时可用的全套选项的信息，请参见创建表和写入表。

如果源文件是Parquet格式，则可以使用SQL Convert to Delta语句就地转换文件，以创建非托管表

SQL

```
%sql
CONVERT TO DELTA parquet.`/mnt/delta/events`
```



## 分区数据

要加快包含涉及分区列的谓词查询，可以对数据进行分区。

Python

```
%pyspark
events = spark.read.json("/databricks-datasets/structured-streaming/events/")
events.write.partitionBy("date").format("delta").save("/mnt/delta/events")
spark.sql("CREATE TABLE events USING DELTA LOCATION '/mnt/delta/events/'")
```

SQL

```
%sql
CREATE TABLE events (
 date DATE,
 eventId STRING,
 eventType STRING,
 data STRING)
USING delta
PARTITIONED BY (date)
```

## 修改表格

Delta Lake支持一组丰富的操作来修改表。

1. **对流写入表**

   您可以使用结构化流式处理将数据写入Delta表。即使有其他流或批查询同时运行表，Delta Lake事务日志也可以保证一次性处理。默认情况下，流在附加模式下运行，这会将新记录添加到表中。

   **Python**

```
%pyspark
from pyspark.sql.types import *

inputPath = "/databricks-datasets/structured-streaming/events/"

jsonSchema = StructType([ StructField("time", TimestampType(), True), StructField("action", StringType(), True) ])

eventsDF = (
 spark
  .readStream
  .schema(jsonSchema) # Set the schema of the JSON data
  .option("maxFilesPerTrigger", 1) # Treat a sequence of files as a stream by picking one file at a time
  .json(inputPath)
)

(eventsDF.writeStream
 .outputMode("append")
 .option("checkpointLocation", "/mnt/delta/events/_checkpoints/etl-from-json")
 .table("events")
)
```

有关Delta Lake与结构化流式处理集成的更多信息，请参阅表流式处理读写。

2. **批处理更新**

要将一组更新和插入合并到现有表中，请使用merge-into语句。例如，以下语句获取更新流并将其合并到表中。当已经存在具有相同事件ID的事件时，Delta Lake将使用给定的表达式更新数据列。如果没有匹配事件时，Delta Lake将添加一个新行。

**SQL**

```
%sql
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)
```

执行INSERT时（例如，当现有数据集中没有匹配的行时），必须为表中的每一列指定一个值。但是，您不需要更新所有值。

3. **读一个表**

   在这个部分：

   - 显示表格历史记录
   - 查询表的早期版本（时间行程）

     您可以通过在DBFS（"/mnt/delta/events"）或表名（"event"）上指定路径来访问Delta表中的数据：

     **Scala**

     ```
     %spark
     SELECT * FROM delta.`/mnt/delta/events`
     ```

     或

     ```
     %spark
     val events = spark.table("events")
     ```

     **SQL**

     ```
     %sql
     SELECT * FROM delta.`/mnt/delta/events`
     ```

     或

     ```
     %sql
     SELECT * FROM events
     ```

4. **显示表的历史记录**

   使用DESCRIBE HISTORY语句，查看表的历史记录。该语句提供每次为写入表出处信息，包括表版本、操作、用户等。检索增量表历史记录

5. **查询表的早期版本（按时间顺序查询）**

   使用Delta Lake时间行程，您可以查询Delta表的旧快照。

   对于timestamp_string，只接受日期或时间戳字符串。例如，“2019-01-01”和“2019-01-01'T'00:00:00.000Z”。

   若要查询较早版本的表，请在语句中指定版本或时间戳SELECT。例如，若要从上述历史记录中查询版本0，请使用

   SQL

   ```
   %sql
   SELECT * FROM events VERSION AS OF 0
   ```

   or

   ```
   %sql
   SELECT * FROM events VERSION AS OF 0
   ```

> **注意**
>
> 因为版本1的时间戳为'2019-01-29 00:38:10'，要查询版本0，您可以使用范围为'2019-01-29 00:37:58'到'2019-01-29 00:38:09'的任何时间戳。


使用DataFrameReader选项，您可以通过Delta table中一个固定的特定版本表创建DataFrame。

Python

```
%pyspark
df1 = spark.read.format("delta").option("timestampAsOf", timestamp_string).load("/mnt/delta/events")
df2 = spark.read.format("delta").option("versionAsOf", version).load("/mnt/delta/events")
```

有关详细信息，请参阅查询表的旧快照（按时间顺序查询）。

6. **优化表**

   对表执行多次更改后，可能会有很多小文件。为了提高读取查询的速度，可以使用OPTIMIZE将小文件折叠为较大的文件：

   SQL

   ```
   %sql
   OPTIMIZE delta.`/mnt/delta/events`
   ```

   或

   ```
   %sql
   OPTIMIZE events
   ```

7. **Z-order排序**

   为了进一步提高读取性能，可以通过Z-Ordering在同一组文件中共同定位相关信息。Delta Lake数据跳过算法会自动使用这种共区域性来显著减少需要读取的数据量。对于Z-Order数据，您可以在子句中指定要排序的列。例如：要通过共同定位，请运行：ZORDER BY Clause

   SQL

```
%sql
OPTIMIZE events
ZORDER BY (eventType)
```

8. **清理快照**

   Delta Lake为读取提供快照隔离，这意味着即使其他用户或作业正在查询表时，也可以安全地运行OPTIMIZE。但是最终，您应该清理旧快照。您可以通过运行以下VACUUM命令来执行此操作

   SQL

   ```
   %sql
   VACUUM events
   ```

   您可以使用 RETAIN<N>HOURS 选项来控制最新快照的保留时间：

   SQL

   ```
   %sql
   VACUUM events RETAIN 24 HOURS
   ```

   有关VACUUM有效使用的详细信息，请参阅删除不再由Delta表引用的文件。

> **注意**
>
> 有关本文章详细信息，请参考官方文档：Delta快速入门

# 3.3. 入门笔记本

这些笔记本显示了如何将JSON数据转换为Delta Lake格式，创建Delta表，追加到表，优化结果表，最后使用Delta Lake元数据命令显示表的历史记录，格式和详细信息。

## Delta Lake Quickstart Python笔记本

Note 链接地址：Databricks Delta Quickstart (Python)

## Delta Lake Quickstart Scala笔记本

Note 链接地址：Databricks Delta Quickstart (Scala)

## Delta Lake快速入门SQL笔记本

Note 链接地址：Databricks Delta Quickstart (SQL)

> **说明**
>
> 详情可参考参考Databricks官网入门笔记本文章

# 3.4. 表批读写

Delta Lake支持Apache Spark DataFrame读写API提供的大多数选项，用于对表执行批量读写。

> **说明**
>
> 详细内容可参考Databricks官网文章：表批读写

有关演示这些功能的Databricks笔记本，请参阅入门笔记本二。

有关Delta Lake SQL命令的信息，请参见

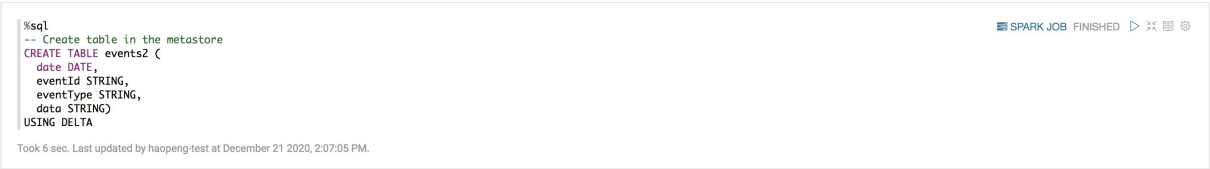
- Databricks Runtime 7.0及更高版本：Databricks Runtime 7.x SQL参考
- Databricks Runtime 6.x及以下版本：Databricks Runtime 5.5 LTS和6.x SQL参考

## 建立表格

Delta Lake支持使用DataFrameWriter（Scala/Java / Python）直接基于路径创建表。Delta Lake还支持使用标准DDL CREATE TABLE在元存储中创建表。 使用Delta Lake在元存储中创建表时，它将表数据的位置存储在元存储中。此方式使其他用户更容易发现和引用数据，而无需担心数据存储的准确位置。但是，元存储不是表中有效内容的真实来源。数据内容仍然由Delta Lake负责存储。。

### SQL

```
%sql
-- Create table in the metastore
CREATE TABLE events (
 date DATE,
 eventId STRING,
 eventType STRING,
 data STRING)
USING DELTA
```




### Python

```
%pyspark
df = spark.createDataFrame([("case21", '2020-10-12', 21, 'INFO'),("case22", '2020-10-13', 22, 'INFO')], ['data', '
date', 'eventId', 'eventType'])
df.write.format("delta").saveAsTable("events2")    # create table in the metastore

df.write.format("delta").save("/mnt/delta/events3") # create table by path
```


### Scala

```
%spark
val df = spark.createDataFrame(Seq(("case21", "2020-10-12", 21, "INFO"))).toDF("data", "date", "eventId", "eventType")
df.write.format("delta").saveAsTable("events4")    // create table in the metastore

df.write.format("delta").save("/mnt/delta/events5")  // create table by path
```


在Dat abricks Runt ime 7.0及更高版本中，您可以使用Dat aFrameWrit erV2接口创建Delt a表。SQL还支持在路径中创建表，而无需在Hive元存储中创建条目。

SQL

```
%sql
-- Create a table by path
CREATE OR REPLACE TABLE delta.`/mnt/delta/events` (
 date DATE,
 eventId STRING,
 eventType STRING,
 data STRING)
USING DELTA
PARTITIONED BY (date);

-- Create a table in the metastore
CREATE OR REPLACE TABLE events (
 date DATE,
 eventId STRING,
 eventType STRING,
 data STRING)
USING DELTA
PARTITIONED BY (date);
```

SCALA

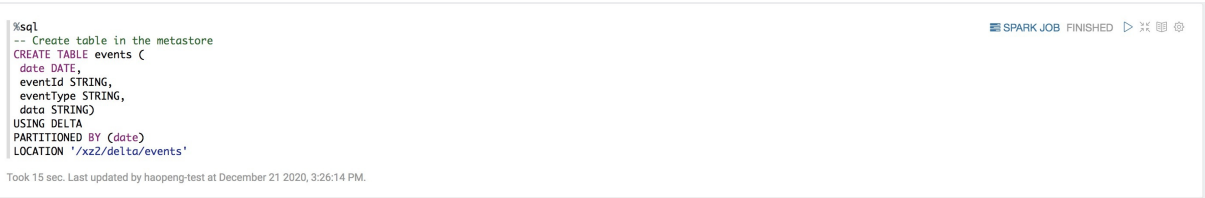
```
%spark
val df = spark.createDataFrame(Seq(("case21", "2020-10-12", 21, "INFO"))).toDF("data", "date", "eventId", "e
ventType")
df.writeTo("delta.`/mnt/delta/events`").using("delta").partitionedBy("date").createOrReplace() // create t
able by path

df.writeTo("events").using("delta").partitionedBy("date").createOrReplace()            // create table in the m
etastore
```

## 分区数据

您可以对数据进行分区以加快其谓词及分区列的查询和DML。要在创建Delt a表时对数据进行分区，请按列指定分区。常见的模式是按日期划分，例如：

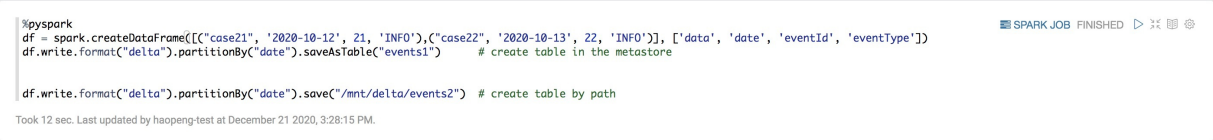
SQL

```
%sql
-- Create table in the metastore
CREATE TABLE events (
 date DATE,
 eventId STRING,
 eventType STRING,
 data STRING)
USING DELTA
PARTITIONED BY (date)
LOCATION '/mnt/delta/events'
```



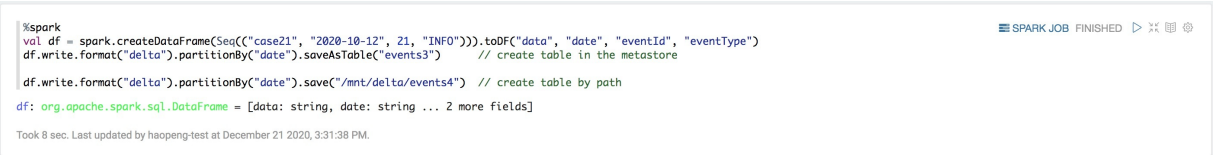

Pyt hon

```
%pyspark
df = spark.createDataFrame([("case21", '2020-10-12', 21, 'INFO'),("case22", '2020-10-13', 22, 'INFO')], ['data', '
date', 'eventId', 'eventType'])
df.write.format("delta").partitionBy("date").saveAsTable("events1")    # create table in the metastore


df.write.format("delta").partitionBy("date").save("/mnt/delta/events2") # create table by path
```




Scala

```
%spark
val df = spark.createDataFrame(Seq(("case21", "2020-10-12", 21, "INFO"))).toDF("data", "date", "eventId", "eventType")
df.write.format("delta").partitionBy("date").saveAsTable("events3")    // create table in the metastore

df.write.format("delta").partitionBy("date").save("/mnt/delta/events4") // create table by path
```




## 控制数据位置

要控制Delta表文件的位置，可以选择将LOCATION指定为OSS上的路径。与不指定路径的内部表不同，当您使用DROP表时，不会删除外部表的文件

如果运行CREATE TABLE的位置已经包含使用Delta Lake存储的数据，Delta Lake将执行以下操作：

如果只指定表名和位置，例如：

SQL

```
%sql
CREATE TABLE events
USING DELTA
LOCATION '/mnt/delta/events'
```

Hive Metastore中的表会自动继承现有数据的Schema，分区和表属性。此功能可用于将数据“导入”到元存储中。

如果指定任何配置（Schema，分区或表属性），则Delta Lake会验证该规范与现有数据的配置是否完全匹。

> **警告**
>
> 如果指定的配置与数据的配置不完全匹配，则Delta Lake会抛出描述差异的异常。

## 读取表

您可以通过指定表名或路径将Delta表加载到DataFrame中：

SQL

```
%sql
SELECT * FROM events  -- query table in the metastore

SELECT * FROM delta.`/mnt/delta/events` -- query table by path
```

### Python

```
%pyspark
spark.table("events")  # query table in the metastore

spark.read.format("delta").load("/mnt/delta/events")  # query table by path
```

### Scala

```
%spark
spark.table("events")   // query table in the metastore

spark.read.format("delta").load("/mnt/delta/events") // create table by path
```

返回的DataFrame会自动读取表中之前查询结果的最新快照；您不需要运行REFRESH TABLE。当查询中有适用的谓词时，Delta-Lake会自动使用分区和统计来读取最小数量的数据。

## 查询表的旧快照（按时间顺序查看）

Delta Lake按时间顺序查看允许您查询Delta表的旧快照。按时间顺序查看有很多用例，包括：

- 重新创建分析，报告或输出（例如，机器学习模型的输出）。这对于排查问题或者审计尤其有用，特别是在管控行业中。
- 编写复杂的时间查询。
- 修正数据中的错误。
- 在快速变更的表中，提供一系列的快照查询功能。

本节介绍了旧版本表和数据保存相关的查询方法，并提供了示例。

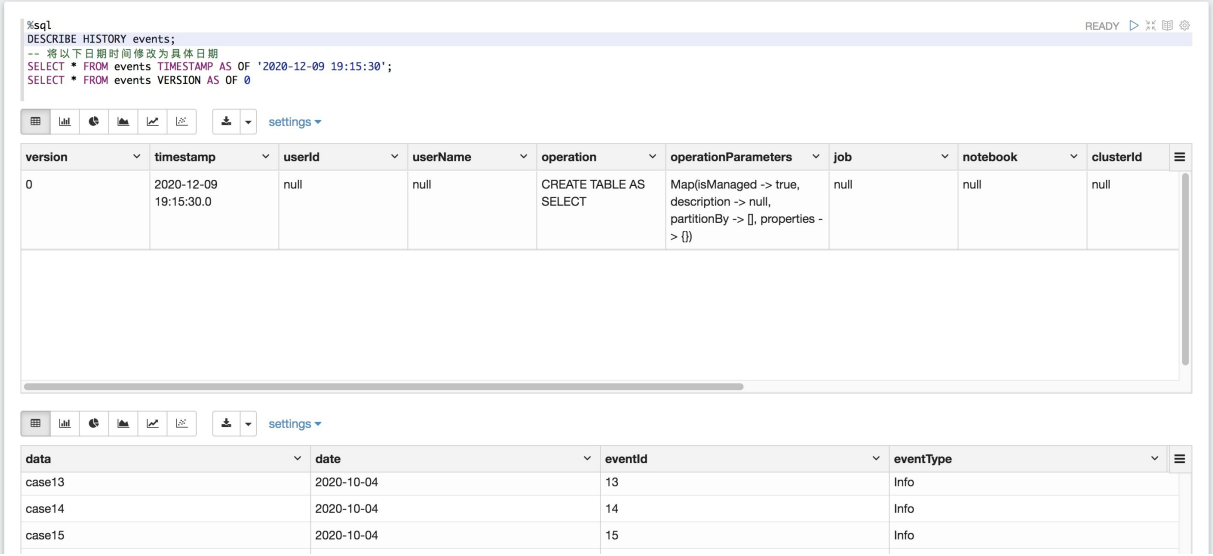
**语法**

本节说明如何查询较旧版sql本的Delta表。

SQL AS OF 语法

```
%sql
SELECT * FROM
events
 TIMESTAMP AS OF timestamp_expression
SELECT * FROM events VERSION AS OF version
```

> **说明**
>
> timestamp_expression为实际的时间，你可以通过DESCRIBE HISTORY events查看表的历史版本




table_identifier

- [database_name.]table_name：一个表名，可以选择用数据库名限定。
- delta.`<path-to-table>`：现有Delta表的位置。

时间戳表达式可以是以下任一项

1. '2018-10-18T22:15:12.013z',即可以转换为时间戳的字符串
2. cast('2018-10-18 13:36:32 CEST' as timestamp)
3. '2018-10-18'，即日期字符串
4. 在Databricks运行时6.6及更高版本中：
    i. current_timestamp() - interval 12 hours
    ii. date_sub(current_date(), 1)
    iii. 任意可以转换为时间戳的其它表达式

version是一个长整型数值，可以从DESCRIBE HISTORY table_spec查询中获取到。

时间戳表达式和版本都不能是子查询。

**SQL**

```
%sql
SELECT * FROM events TIMESTAMP AS OF '2018-10-18T22:15:12.013Z'
SELECT * FROM delta.`/mnt/delta/events` VERSION AS OF 123
```

**DataFrameReader选项**

您可以使用DataFrameReader从特定版本的Delta表中创建DataFrame

**Python**

```
%pyspark
df1 = spark.read.format("delta").option("timestampAsOf", timestamp_string).load("/mnt/delta/events")
df2 = spark.read.format("delta").option("versionAsOf", version).load("/mnt/delta/events")
```


对于timestamp_string，仅接受日期或时间戳记字符串。例如"2019-01-01"和"2019-01-01T00:00:00.000Z"。

一种常见的模式是在执行Databricks作业期间使用Delta表的最新状态来更新下游应用程序。

由于 Delta 表会自动更新，因此，如果基础数据进行了更新，则在多次调用时，从 Delta 表加载的DataFrame可能返回不同的结果。通过使用按时间顺序查看，您可以修复多次调用的DataFrame返回的数据：

**Python**

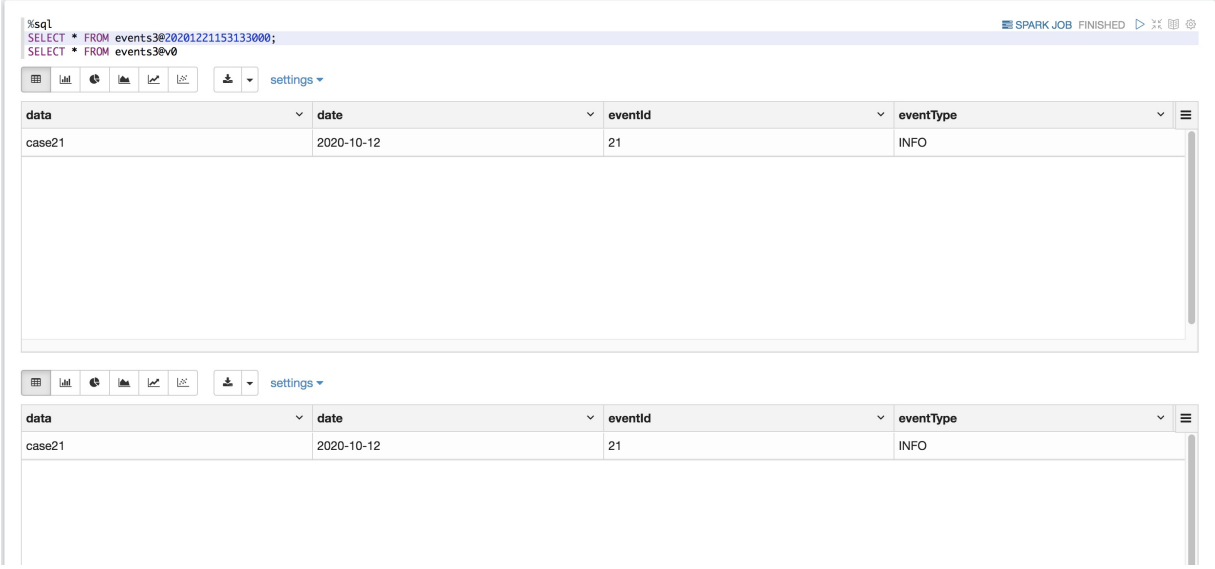
```
%pyspark
latest_version = spark.sql("SELECT max(version) FROM (DESCRIBE HISTORY delta.`/mnt/delta/events`)").co
llect()
df = spark.read.format("delta").option("versionAsOf", latest_version[0][0]).load("/mnt/delta/events")
```


**语法**

您可能有一个参数化的管道，其中管道的输入路径是作业的参数。在执行作业之后，您可能希望在将来某个时间重新生成输出。在这种情况下，可以使用@语法指定时间戳或版本。时间戳必须为yyyymmddhhmmssss格式。您可以在@之后指定一个版本，方法是在版本前面加一个v。例如，要查询表evnets件的版本123，请指定evnets@v123。

**SQL**

```
%sql
SELECT * FROM events@20190101000000000;
SELECT * FROM events@v123
```




**Python**

```
%pyspark
spark.read.format("delta").load("/mnt/delta/events@20190101000000000") # table on 2019-01-01 00:00:00.000
spark.read.format("delta").load("/mnt/delta/events@v123")          # table on version 123
```

## 数据保存

默认情况下，Delta表将提交历史记录保留30天。这意味着您可以声明一个30天以内的版本。但是，有以下注意事项：

- 您没有在Delta表上运行VACUUM。如果运行VACUUM，您将无法恢复到默认的7天数据保留期之前的版本。

您可以使用以下表属性来配置保留期：

- delta.logRetentionDuration = "interval <interval>"：控制表的历史记录保留时间长度。每次写入一个检查点时，Databricks都会自动清除早于保留间隔的日志条目。如果将此配置设置为足够大的值，则会保留许多日志条目。这不会影响性能，因为针对日志的操作恒定时间。历史记录的操作是并行的（但是随着日志大小的增加，它将变得更加昂贵）。默认值为interval 30 days
- delta.deletedFileRetentionDuration = "interval <interval>"：控制选择的文件必须选择时间段，默认值为间隔7天。若要访问 30 天的历史数据，请设置 delta.deletedFileRetentionDuration = "interval 30 days"。此设置可能会导致您的存储成本上升。

  > **注意**
  >
  > VACUUM 不清理日志文件；写入检查点后，日志文件将自动清除。

  按时间顺序查看到以前的版本，必须保留日志文件和该版本的数据文件。

## 案例

- 修复用户111表的意外删除问题：

  **SQL**

```
%sql
INSERT INTO my_table
 SELECT * FROM my_table TIMESTAMP AS OF date_sub(current_date(), 1)
 WHERE userId = 111
```

- 修复对表的意外错误更新：

  SQL

```
%sql
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 *
```

- 查询过去一周增加的新客户数量。

  SQL

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

## 写入表格

### 追加

使用append模式，可以将新数据以原子的方式添加到现有的Delta表中：

SQL

```
%sql
INSERT INTO events SELECT * FROM newEvents
```

Python

```
%pyspark
df.write.format("delta").mode("append").save("/mnt/delta/events")
df.write.format("delta").mode("append").saveAsTable("events")
```

Scala

```
%spark
df.write
 .format("delta")
 .mode("overwrite")
 .option("replaceWhere", "date >= '2017-01-01' AND date <= '2017-01-31'")
 .save("/mnt/delta/events")
```

**覆盖**

要原子式地替换表中的所有数据，可以使用overwrite模式：

SQL

```
%sql
INSERT OVERWRITE TABLE events SELECT * FROM newEvents
```


Python

```
%pyspark
from pyspark.sql.functions import  to_date
df = spark.createDataFrame([("case21", '2020-10-12', 23, 'INFO'),("case22", '2020-10-13', 24, 'INFO')], ['data', '
date', 'eventId', 'eventType'])
df1 = df.select('data', to_date('date', 'yyyy-MM-dd').alias('date'), 'eventId', 'eventType')
df1.write.format("delta").mode("append").save("/mnt/delta/events")
df1.write.format("delta").mode("append").saveAsTable("events")
```

Scala

```
%spark
df.write
 .format("delta")
 .mode("overwrite")
 .option("replaceWhere", "date >= '2017-01-01' AND date <= '2017-01-31'")
 .save("/mnt/delta/events")
```

使用DataFrames，还可以选择性地只覆盖与分区列上的谓词匹配的数据。以下命令将原子式地把大于10-4的数据替换为10-12、10-13号的数据

Python

```
%pyspark
df = spark.createDataFrame([("case21", '2020-10-12', 21, 'INFO'),("case22", '2020-10-13', 22, 'INFO')], ['data', 'date', 'eventId', 'eventType'])
df1 = df.select('data', to_date('date', 'yyyy-MM-dd').alias('date'), 'eventId', 'eventType')
# 可以替换分区部分数据（replaceWhere中字段必须是分区字段）;此处替换大于等于10-4的数据，并将10-12、10-13号的数据填入
df1.write.format("delta").mode("overwrite").option("replaceWhere", "date >= '2020-10-4'").saveAsTable("events_partition")
spark.table("events_partition").show(50)
```


Scala

```
%spark
df.write
 .format("delta")
 .mode("overwrite")
 .option("replaceWhere", "date >= '2017-01-01' AND date <= '2017-01-31'")
 .save("/mnt/delta/events")
```

此示例代码将数据写入df，验证所有数据均位于指定分区内，并执行atomic替换。

> **说明**
>
> 与Apache Spark中的文件API不同，Delta Lake会记住并强制执行表的Schema。这意味着默认情况下，覆盖不会替换现有表的Schema。

有关Delta Lake在更新表方面的支持，请参阅表删除，更新和合并。

## 设置用户定义的提交元数据

您可以使用DataFrameWriter选项userMetadata或SparkSession配置spark.databricks.delta.commitInfo.userMetadata，将用户定义的字符串指定为这些操作进行的提交中的元数据。如果同时指定了两个参数，则以DataFrameWriter选项userMetadata优先。用户定义的元数据在历史记录操作中是可读的。

SQL

```
%sql
SET spark.databricks.delta.commitInfo.userMetadata=overwritten-for-fixing-incorrect-data
INSERT OVERWRITE events SELECT * FROM newEvents
```

Python

```
%python
df.write.format("delta") \
 .mode("overwrite") \
 .option("userMetadata", "overwritten-for-fixing-incorrect-data") \
 .save("/mnt/delta/events")
```

Scala

```
%spark
df.write.format("delta")
 .mode("overwrite")
 .option("userMetadata", "overwritten-for-fixing-incorrect-data")
 .save("/mnt/delta/events")
```

## Schema验证

Delta Lake自动验证正在写入的DataFrame的Schema与表的Schema兼容。Delta Lake使用以下规则来确定从DataFrame到表的写入是否兼容：

- 所有DataFrame列都必须存在于目标表中。如DataFrame中有表中不存在列，则会抛出异常。表中存在但DataFrame中不存在的列设置为NULL。
- DataFrame列数据类型必须与目标表中的列数据类型匹配。如果它们不匹配，则会抛出异常。
- DataFrame列名称不能仅通过大小写来区分。这意味着您不能在同一表中定义诸如“Foo”和“foo”之类的列。虽然可以在区分大小写或不区分大小写（默认）模式下使用Spark，但在存储和返回列信息时，Parquet区分大小写。在存储Schema时，Delta Lake 保留但不区分大小写，并采用此限制来避免潜在的错误、数据损坏或丢失问题。

Delta Lake支持DDL显式添加新列，并具有自动更新Schema的功能。

如果您指定其他选项（例如partitionBy与附加模式结合使用），则Delta Lake会验证它们是否匹配，并在任何不匹配项时发生错误。如果分区不存在，会在对现有数据分区之后自动进行追加。

> 注意
>
> 在Databricks Runtime 7.0及更高版本中，INSERT语法提供了Schema强制实施并支持Schema演变。如果不能将列的数据类型安全地强制转换为Delta Lake表的数据类型，则将抛出运行时异常。如果启用Schema演化，则新列可以作为Schema的最后一列（或嵌套列）存在，以使Schema得以演化。

## 更新表Schema

Delta Lake可让您更新表的Schema。支持以下类型的更改：

- 添加新列（在任意位置）
- 重新排序现有列

您可以使用DDL显式进行更改，也可以使用DML隐式进行更改。

> 警告
>
> 更新Delta表Schema时，从该表读取的流将终止。如果你希望流继续进行，则必须重新启动它。

有关推荐的方法，请参见生产中的结构化流。

## 显示更新Schema

您可以使用以下DDL显式更改表的Schema。

## 添加列

SQL

```
%sql
ALTER TABLE table_name ADD COLUMNS (col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name], ...)
```

默认情况下，可空性为ture。

要将列添加到嵌套字段，请使用：

SQL

```
%sql
ALTER TABLE table_name ADD COLUMNS (col_name.nested_col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name], ...)
```

**例**

如果运行之前的Schema为：ALTER TABLE boxes ADD COLUMNS (colB.nested STRING AFTER field1)

```
- root
| - colA
| - colB
| +-field1
| +-field2
```

之后的Schema是：

```
- root
| - colA
| - colB
| +-field1
| +-nested
| +-field2
```

> 说明
> 仅支持为结构添加嵌套列。不支持数组和映射。

**更改列注释或顺序**

SQL

```
%sql
ALTER TABLE table_name CHANGE [COLUMN] col_name col_name data_type [COMMENT col_comment] [FIR
ST|AFTER colA_name]
```

要更改嵌套字段中的列，请使用

SQL

```
%sql
ALTER TABLE table_name CHANGE [COLUMN] col_name.nested_col_name nested_col_name data_type [CO
MMENT col_comment] [FIRST|AFTER colA_name]
```

**例**

如果运行之前的Schema为：ALTER TABLE boxes CHANGE COLUMN colB.field2 field2 STRING FIRST

```
- root
| - colA
| - colB
| +-field1
| +-field2
```

之后的模式是：

```
- root
| - colA
| - colB
| +-field2
| +-field1
```

**更换列**

SQL

```
%sql
ALTER TABLE table_name REPLACE COLUMNS (col_name1 col_type1 [COMMENT col_comment1], ...)
```

**例**

运行以下DSL时：

SQL

```
%sql
ALTER TABLE boxes REPLACE COLUMNS (colC STRING, colB STRUCT<field2:STRING, nested:STRING, field1:ST
RING>, colA STRING)
```

如果之前的Schema是：

```
- root
| - colA
| - colB
| +-field1
| +-field2
```

之后的模式是：

```
- root
| - colC
| - colB
| +-field2
| +-nested
| +-field1
| - colA
```

**更改列类型或名称**

更改列的类型或名称或删除列需要重写表。为此，请使用以下overwriteSchema选项：

**更改列类型**

Python

```
%pyspark
spark.read.table(...)
 .withColumn("date", col("date").cast("date"))
 .write
 .format("delta")
 .mode("overwrite")
 .option("overwriteSchema", "true")
 .saveAsTable(...)
```

**更改列名**

Python

```
%pyspark
spark.read.table(...)
 .withColumnRenamed("date", "date_created")
 .write
 .format("delta")
 .mode("overwrite")
 .option("overwriteSchema", "true")
 .saveAsTable(...)
```

## 自动Schema更新

Delta Lake可以作为DML事务的一部分（附加或覆盖）自动更新表的Schema，并使该Schema与正在写入的数据兼容。

**添加列**

在以下情况下，会自动添加DataFrame中存在但表中缺少的列将作为写事务的一部分：

- write或writeStream有.option("mergeSchema", "true")
- spark.databricks.delta.schema.autoMerge.enabled是 true

如果同时指定了两个选项，则以该DataFrameWriter的option选项为准。添加的列将追加到它们所在结构的末尾。追加新列时将保留大小写。

> **说明**
>
> - 当启用了表访问控制时，mergeSchema不受支持（因为它将需要MODIFY的请求为ALL PRIVILEGES的请求）。
> - mergeSchema 不能与INSERT INTO或.write.insertInto()一起使用。

**NullType 列**

由于Parquet不支持Nulltype，因此在写入Delta表时会将Nulltype列从DataFrame中删除，但仍存储在Schema中。当为该列接收到不同的数据类型时，Delta Lake会将Schema合并到新的数据类型。如果Delta Lake收到现有列的Nulltype，则在写入过程中将保留旧Schema，并删除新列。

Nulltype不支持流式传输。由于必须在使用流式传输时设置Schema，因此这种情况很少见。Nulltype也不适用于ArrayType和MapType的复杂类型。

## 替换表Schema

默认情况下，覆盖表中的数据不会覆盖Schema。在不使用replaceWhere的情况下使用mode（“overwrite”）重写表时，您可能仍然希望覆盖正在写入的数据的Schema。通过将overwriteSchema选项设置为true，可以替换表的Schema和分区：

Python

```
%pyspark
df.write.option("overwriteSchema", "true")
```

## Table上的视图

就像您可能使用数据源表一样，Delta Lake支持在Delta表之上创建视图。

这些视图与表访问控制集成在一起，以实现列级和行级安全性。

使用视图进行操作时的核心挑战是解析Schema。如果更改Delta表Schema，则必须重新创建派生视图以说明对该Schema的添加任何内容。例如，如果向Delta表中添加一个新列，则必须确保此列在该基于该基表构建的相应视图中可用。

## 表属性

你可以在CREATE和ALTER时使用TBLPROPERTIES作为表属性来存储你的元数据

TBLPROPERTIES作为Delta表元数据一部分存储。如果给定位置中已经存在Delta表，则不能在CREATE语句中定义新的TBLPROPERTIES 。

此外，为了调整行为和性能，Delta Lake支持某些Delta表属性：

- 阻止Delta表中删除和更新：delta.appendOnly=ture。
- 配置按时间顺序查看保留属性：delta.logRetentionDuration=<interval-string>和delta.deletedFileRetentionDuration=<interval-string>。
- 配置要收集其统计信息的列数：delta.dataSkippingNumIndexedCols=<number-of-columns>。此属性仅对写入的新数据有效。

> **说明**
>
> - 这些是唯一受支持的delta.前缀表属性。
> - 修改Delta表属性是一种写操作，它将与其他并发的写操作发生冲突，从而导致它们失败。我们建议您仅在表上没有并发写操作时才修改表属性。

您还可以使用Spark配置在首次提交delta表时设置delta.-prefixed属性。例如，要使用该属性初始化Delta表delta.appendOnly=true，请将Spark配置spark.databricks.delta.properties.defaults.appendOnly设置为ture。例如：

SQL

```
%sql
spark.sql("SET spark.databricks.delta.properties.defaults.appendOnly = true")
```

Scala

```
%spark
spark.conf.set("spark.databricks.delta.properties.defaults.appendOnly", "true")
```

Python

```
%pyspark
spark.conf.set("spark.databricks.delta.properties.defaults.appendOnly", "true")
```

### 元数据表

Delta Lake具有丰富的功能，可用于浏览元数据表。

它支持SHOW[PARTITIONS | COLUMNS]和DESCRIBE TABLE。查看

- Databricks Runtime 7.0及更高版本
- Databricks Runtime 6.x及以下版本

它还提供以下独特命令：

- DESCRIBE DETAIL
- DESCRIBE HISTORY

**DESCRIBE DETAIL**

提供有关Schema，分区，表大小等的信息。

**DESCRIBE HISTORY**

提供源信息，包括操作，用户等，以及每次写入表的操作指标。表格历史记录将保留30天。

数据侧栏提供了Delta表的详细表信息和历史的可视化视图。除了表Schema和示例数据外，还可以单击“历史记录”选项卡来查看与“DESCRIBE HISTORY”一起显示的表历史记录。

### Notebook

有关各种Delta表元数据命令的示例，请参见以下笔记本的末尾：

**Delta Lake批处理命令笔记本**

Note链接地址：Databricks Delta Quickstart (SQL)

# 3.5. 表流读写

> **说明**
>
> 详细内容请参考Databricks官网文章：表流读写

有关演示这些功能的Databricks笔记本，请参阅入门笔记本二。

Delta Lake通过readStream和writeStream与Spark结构化流式处理深度集成。Delta Lake克服了许多流式处理系统和文件相关的常见限制，例如：

- 合并低延迟引入产生的小文件
- 保持多个流（或并发批处理作业）执行“仅一次”处理

- 使用文件作为流源时，可以有效地发现哪些文件是新文件

## Delta表作为流源

当您将Delta表加载为流源并在流式查询中使用它时，该查询将处理表中存在的所有数据以及流启动后到达的所有新数据。

您可以将路径和表都作为流加载。

Scala

```
%spark
spark.readStream.format("delta").load("/mnt/delta/events")
```

或

Scala

```
%spark
spark.readStream.format("delta").table("events")
```

你也可以执行以下操作：

- 通过设置maxFilesPerTrigger选项，控制Delta Lake提供给流的任何微批处理的最大大小。这指定了每个触发器中要考虑的新文件的最大数量。默认值为1000。
- 通过设置maxBytesPerTrigger选项来限制每个微批处理的数据量的速率。这将设置一个“软最大值”，这意味着批处理大约此数量的数据，并可能处理超过该限制的数据量。如果你使用Trigger。如果Trigger.Once用于流式传输，则忽略此选项。如果将此选项与maxFilesPerTrigger结合使用，则微批处理将处理数据，，直到达到maxFilesPerTrigger或maxBytesPerTrigger限制。

**忽略更新和删除**

结构化流式处理不处理非追加的输入，如果在用作源的表上进行了任何修改，则引发异常。有两种主要策略可以处理无法自动向下游传播的更改：

- 您可以删除输出和检查点，并从头开始重新启动流。
- 您可以设置以下两个选项之一：
  - ignoreDeletes：忽略在分区边界删除数据的事务。
  - ignoreChanges：如果由于更新、合并、删除（在分区内）或覆盖等数据更改操作而必须重写源表中的文件，则重新处理更新。未更改的行可能仍会发出，因此您的下游使用者应该能够处理重复项。删除不会传播到下游。ignoreChanges包含ignoreDeletes。因此，如果使用ignoreChanges，则源表的删除或更新不会中断流。

**示例**

例如，假设您有一个表user_events，其中包含date、user_email和action列，这些列是按日期分区的。由于GDPR的原因，您需要从user_events表中删除数据。

当您在分区边界处执行删除（即，WHERE在分区列上）操作时，文件已经按值分段，因此删除操作只是从元数据中删除这些文件。因此，如果只想从某些分区删除数据，可以使用：

Scala

```
%spark
events.readStream
 .format("delta")
 .option("ignoreDeletes", "true")
 .load("/mnt/delta/user_events")
```

但是，如果您必须基于user_email删除数据，则需要使用：

Scala

```
%spark
events.readStream
 .format("delta")
 .option("ignoreChanges", "true")
 .load("/mnt/delta/user_events")
```

如果使用update语句更新user_email，则会重写包含user_email相关的文件。使用ignoreChanges时，新记录将与位于同一文件中的所有其他未更改记录一起传播到下游。逻辑应该能够处理这些传入的重复记录。

## 指定初始位置

> **说明**
>
> 该功能在Databricks Runtime 7.3 LTS及更高版本上可用。

您可以使用以下选项来指定Delta Lake流式处理源的起点，而无需处理整个表。

- startingVersion：Delta Lake版本开始。从该版本（包括该版本）开始的所有表更改都将由流式处理源读取。您可以从命令“DESCRIBE HISTORY events”输出的version列中获取提交版本。
  - 要仅返回最新更改，请在Databricks Runtime 7.4及更高版本中指定latest。
- startingTimestamp：开始的时间戳。流式处理源将读取在时间戳（包括时间戳）或之后提交的所有表更改。可以是以下任何一项：
  - '2018-10-18T22:15:12.013Z'，即可以转换为时间戳的字符串
  - cast（'2018-10-18 13:36:32 CEST' as timestamp）
  - '2018-10-18'，即日期字符串
  - 本身就是时间戳或可强制转换为时间的任何其他表达式，如：current_timestamp() - interval 12 hour sdate_sub(current_date(), 1)

您不能同时设置两个选项。您只需要使用其中之一个选项即可。它们仅在启动新的流查询时才生效。如果流式处理查询已启动并且进度已记录在其检查点中，则将忽略这些选项。

> **警告**
>
> 尽管可以从指定的版本或时间戳启动流式处理源，但是流式处理源的架构始终是Delta表的最新架构。您必须确保在指定的版本或时间戳记之后，不对Delta表进行不兼容的架构更改。否则，当使用不正确的架构读取数据时，流式传输源可能会返回不正确的结果。

**示例**

例如，假设您有一个表格User_events。如果要阅读版本5之后的更改，可以使用：

Scala

```
%spark
events.readStream
 .format("delta")
 .option("startingVersion", "5")
 .load("/mnt/delta/user_events")
```

如果您想阅读自2018年10月18日以来的更改，可以使用：

Scala

```
%spark
events.readStream
 .format("delta")
 .option("startingTimestamp", "2018-10-18")
 .load("/mnt/delta/user_events")
```

## 用作接收器的Delta表

您也可以使用结构化流将数据写入Delta表。事务日志使Delta Lake能够保证"仅一次"处理，即使针对该表同时运行其他流或批查询。

**追加模式**

默认情况下，流以追加模式运行，这会将新记录添加到表中。

您可以使用路径方法：

Python

```
%pyspark
events.writeStream
 .format("delta")
 .outputMode("append")
 .option("checkpointLocation", "/delta/events/_checkpoints/etl-from-json")
 .start("/delta/events")
```

Scala

```
%spark
events.writeStream
 .format("delta")
 .outputMode("append")
 .option("checkpointLocation", "/mnt/delta/events/_checkpoints/etl-from-json")
 .start("/mnt/delta/events")
```

或表格方法：

Python

```
%pyspark
events.writeStream
 .format("delta")
 .outputMode("append")
 .option("checkpointLocation", "/delta/events/_checkpoints/etl-from-json")
 .table("events")
```

Scala

```
%spark
events.writeStream
 .outputMode("append")
 .option("checkpointLocation", "/mnt/delta/events/_checkpoints/etl-from-json")
 .table("events")
```

**完整模式**

您还可以使用结构化流式处理技术将整个批替换为每个批。一个示例用例是使用聚合来计算摘要：

Scala

```
%spark
spark.readStream
 .format("delta")
 .load("/mnt/delta/events")
 .groupBy("customerId")
 .count()
 .writeStream
 .format("delta")
 .outputMode("complete")
 .option("checkpointLocation", "/mnt/delta/eventsByCustomer/_checkpoints/streaming-agg")
 .start("/mnt/delta/eventsByCustomer")
```


前面的示例不断更新包含客户的事件总数的表。

对于延迟要求更宽松的应用程序，您可以使用一次性触发器来节省计算资源。使用这些更新按给定的时间表更新汇总聚合表，仅处理自上次更新以来已到达的新数据。

# 3.6. 表删除，更新和合并

> **说明**
>
> 详细内容请参考Databricks官网文章：表删除，更新和合并

有关演示这些功能的Databricks笔记本，请参阅入门笔记本二。

Delta Lake支持多个语句，以方便从Delta表中删除数据和更新数据。

## 从表中删除

可以从Delta表中删除与谓词匹配的数据。例如，要删除2017年之前的所有事件，可以运行以下命令：

SQL

```
%sql
DELETE FROM events WHERE date < '2017-01-01'
DELETE FROM delta.`/data/events/` WHERE date < '2017-01-01'
```

Python

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

deltaTable = DeltaTable.forPath(spark, "/data/events/")

deltaTable.delete("date < '2017-01-01'")      # predicate using SQL formatted string

deltaTable.delete(col("date") < "2017-01-01")  # predicate using Spark SQL functions
```

Scala

```
%spark
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, "/data/events/")

deltaTable.delete("date < '2017-01-01'")      // predicate using SQL formatted string

import org.apache.spark.sql.functions._
import spark.implicits._

deltaTable.delete(col("date") < "2017-01-01")     // predicate using Spark SQL functions and implicits
```

**警告**

delete从最新版本的Delt a表中删除数据，但直到显示清除旧版本后才从物理存储中删除数据。

**说明**

如果可能，请在分区的Delt a表的分区列上提供谓词，因为这样的谓词可以显著加快操作速度。

有关详细信息，请参见API参考。

## 更新表格

可以更新与Delt a表中谓词匹配的数据。例如，要修复eventType中的拼写错误，可以运行以下命令：

SQL

```
%sql
UPDATE events SET eventType = 'click' WHERE eventType = 'clck'

UPDATE delta.`/data/events/` SET eventType = 'click' WHERE eventType = 'clck'
```

Python

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

deltaTable = DeltaTable.forPath(spark, "/data/events/")

deltaTable.update("eventType = 'clck'", { "eventType": "'click'" } )  # predicate using SQL formatted string

deltaTable.update(col("eventType") == "clck", { "eventType": lit("click") } )  # predicate using Spark SQL fun
ctions
```

Scala

```
%spark
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, "/data/events/")

deltaTable.updateExpr(        // predicate and update expressions using SQL formatted string
 "eventType = 'clck'",
 Map("eventType" -> "'click'")

import org.apache.spark.sql.functions._
import spark.implicits._

deltaTable.update(           // predicate using Spark SQL functions and implicits
 col("eventType") === "clck",
 Map("eventType" -> lit("click")));
```

有关详细信息，请参见API参考。

> **说明**
>
> 与删除类似，在分区上使用谓词可以显著提高更新操作的速度。

## 使用合并操作在表中执行更新插入

您可以使用该合并操作将数据从源表，视图或DataFrame插入目标Delta表。此操作类似于SQL MERGE INTO命令，但对更新、插入和删除中删除操作有额外的支持。

假设您有一个Spark DataFrame，其中包含eventId的事件的新数据，其中一些事件可能已经存在于events表中，需要更新匹配的行（即eventId已经存在）并插入新行（即eventId不存在）。您可以运行以下命令：

SQL

```
%sql
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)
```

Python

```
%pyspark
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, "/data/events/")
df = spark.createDataFrame([("update-case2", '2020-10-12', 2, 'INFO'),("case25", '2020-10-13', 25, 'INFO')], ['d
ata', 'date', 'eventId', 'eventType'])
updatesDF = df.select('data', to_date('date', 'yyyy-MM-dd').alias('date'), 'eventId', 'eventType')
# 合并表，eventId相等则更新数据，不相等就插入数据
deltaTable.alias("events").merge(
  updatesDF.alias("updates"),
  "events.eventId = updates.eventId") \
 .whenMatchedUpdate(set = { "data" : "updates.data" } ) \
 .whenNotMatchedInsert(values =
  {
   "date": "updates.date",
   "eventId": "updates.eventId",
   "data": "updates.data"
  }
 ) \
 .execute()
```

```
%pyspark
from delta.tables import *


deltaTable = DeltaTable.forPath(spark, "/lm-test/delta/events/")
df = spark.createDataFrame([("update-case2", '2020-10-12', 2, 'INFO'),("case25", '2020-10-13', 25, 'INFO')], ['data', 'date', 'eventId', 'eventType'])
updatesDF = df.select('data', to_date('date', 'yyyy-MM-dd').alias('date'), 'eventId', 'eventType')

# 合并表，eventId相等则更新数据，不相等就插入数据
deltaTable.alias("events").merge(
    updatesDF.alias("updates"),
    "events.eventId = updates.eventId") \
  .whenMatchedUpdate(set = { "data" : "updates.data" } ) \
  .whenNotMatchedInsert(values =
    {
      "date": "updates.date",
      "eventId": "updates.eventId",
      "data": "updates.data",
      "eventType": "updates.eventType"
    }
  ).execute()
deltaTable.toDF().show()
```

```
+------------+----------+-------+---------+
|        data|      date|eventId|eventType|
+------------+----------+-------+---------+
|update-case1|2020-10-01|      1|    error|
|update-case2|2020-10-01|      2|     Warn|
|       case3|2020-10-01|      3|     Warn|
|       case4|2020-10-01|      4|    error|
|       case5|2020-10-02|      5|     Warn|
|       case6|2020-10-02|      6|    error|
|       case7|2020-10-02|      7|     Warn|
|       case8|2020-10-02|      8|    error|
|       case9|2020-10-02|      9|    error|
|      case10|2020-10-03|     10|     Warn|
```

Scala

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

val updatesDF = ...  // define the updates DataFrame[date, eventId, data]

DeltaTable.forPath(spark, "/data/events/")
  .as("events")
  .merge(
    updatesDF.as("updates"),
    "events.eventId = updates.eventId")
  .whenMatched
  .updateExpr(
    Map("data" -> "updates.data"))
  .whenNotMatched
  .insertExpr(
    Map(
      "date" -> "updates.date",
      "eventId" -> "updates.eventId",
      "data" -> "updates.data"))
  .execute()
```

## 操作语义

这是merge编程操作的详细说明。

可以有任意数量的whenMat ched和whenNot Mat ched子句。

> **注意**
>
> 在Databricks Runtime 7.2及更低版本中，merge最多可以有2个whenMatched子句，最多可以有1个whenNotMatched子句。

- 当源行根据匹配条件与目标表行匹配时，将执行whenMatched子句。这些子句具有以下语义。
  - whenMatched子句最多只能有一个更新和一个删除操作。merge中的更新操作只更新匹配的目标行的指定列（类似于更新操作）。删除操作删除匹配的行。
  - 每个whenMatched子句都可以有一个可选条件。如果存在此子句条件，则仅当子句条件为true时，才对任何匹配的源-目标行对行执行更新或删除操作。
  - 如果有多个whenMatched子句，则按照指定的顺序（即，子句的顺序很重要）对它们进行求值。除最后一个外，所有whenMatched子句都必须有条件。
  - 如果两个whenMatched子句都有条件，并且对于匹配的源-目标行对，两个条件都不为true，则匹配的目标行保持不变。
  - 要用源数据集的相应列更新目标Delta表的所有列，请使用whenMatched（...）.updateAll（）。这相当于：

    Scala

    ```
    %spark
    whenMatched(...).updateExpr(Map("col1" -> "source.col1", "col2" -> "source.col2", ...))
    ```

    对于目标Delta表的所有列。因此，此操作假定源表与目标表中的列相同，否则查询将引发分析错误。

    > **说明**
    >
    > 当启用自动架构迁移时，此行为将更改。有关详细信息，请参见自动模式演化。

- 当源行与基于匹配条件的任何目标行都不匹配时，将执行whenNotMatched子句。这些子句具有以下语义。
  - whenNotMatched子句只能有insert操作。新行是根据指定的列和相应的表达式生成的。不需要指定目标表中的所有列。对于未指定的目标列，将插入NULL。

    > **说明**
    >
    > 在Databricks Runtime 6.5及更低版本中，您必须在目标表中提供该INSERT操作的所有列。

  - 每个whenNotMatched子句可以有一个可选条件。如果存在子句条件，则仅当源条件对该行为ture时才插入该行。否则，将忽略源列。
  - 要将目标Delta表的所有列与源数据集的相应列一起插入，请使用whenNotMatched(...).insertAll()。这相当于：

    Scala

    ```
    %spark
    whenNotMatched(...).insertExpr(Map("col1" -> "source.col1", "col2" -> "source.col2", ...))
    ```

目标Delta表的所有列。因此，此操作假定源表的列与目标表的列相同，否则查询将引发分析错误。

> **注意**
>
> 启用自动模式迁移后，此行为将更改。有关详细信息，请参见自动模式演变。

> **警告**
>
> 如果源数据集的多行匹配并试图更新目标Delta表的相同行，则merge操作可能失败。根据merge的SQL语义，这种更新操作是不明确的，因为不清楚应该使用哪个源行来更新匹配的目标行。可以预处理源表，以消除多个匹配的可能性。请参阅"更改数据捕获"示例，它预处理变更数据集（即源数据集），以便在将更改应用到目标Delta表之前仅保留每个键的最新更改。

> **注意**
>
> 在Databricks Runtime 7.3 LTS中，无条件删除匹配项时允许多个匹配项（因为即使有多个匹配项，无条件删除也不是模棱两可的）。

## 架构验证

merge自动验证通过插入和更新表达式生成的数据的架构是否与表的架构兼容。它使用以下规则来确定merge操作是否兼容：

- 对于update和insert操作，目标Delta表中必须存在指定的目标列。
- 对于updateAll和insertAll动作，源数据集必须具有目标Delta表的所有列。源数据集可以包含额外的列，它们将被忽略。
- 对于所有操作，如果由生成目标列的表达式生成的数据类型与目标Delta表中的对应列不同，merge会尝试将其转换为表中的类型。

## 自动架构演变

> **注意**
>
> merge 中的架构演变在Databricks Runtime 6.6及更高版本中可用。

默认情况下，updateAll和insertAll使用来自源数据集的同名列来分配目标Delta表中的所有列。将忽略源数据集中与目标表中的列不匹配的任何列。但是，在某些用例中，需要自动将源列添加到目标Delta表中。要在使用updateAll和insertAll（至少其中一个）执行merge操作期间自动更新表架构，可以在运行merge操作之前设置Spark会话配置spark.databricks.delta.schema.autoMerge.enabled为true。

> **说明**
>
> - 架构演变仅在同时存在一个updateAll或一个insertAll动作或两者同时存在时发生。
> - 在merge中的模式演化过程中，只有顶层列（即不是嵌套字段）会被更改。
> - update和insert操作不能显式引用目标表中不存在的目标列（即使其中有updateAll或insertAll作为子句之一）。请参阅下面的示例。

这里有几个例子说明了在模式演化和没有模式演化的情况下merge操作的效果。

| 列 | 查询（在Scala中） | 没有架构演变的行为（默认值） | 有架构演变行为 |
| --- | --- | --- | --- |
| 目标列： key, value源列： key, value, newValue | targetDeltaTable.alias("t")<br>.merge(<br>sourceDataFrame.alias("s"),<br>"t.key = s.key")<br>.whenMatched().updateAll()<br>.whenNotMatched().insertAll()<br>.execute() | 表架构保持不变；仅已更新/插入列key，value | 表架构更改为（key，value，newValue）。updateAll更新列value和newValue，insertAll插入行（key、value、newValue）。 |
| 目标列： key, oldValue<br>源列： key, newValue | targetDeltaTable.alias("t")<br>.merge(<br>sourceDataFrame.alias("s"),<br>"t.key = s.key")<br>.whenMatched().updateAll()<br>.whenNotMatched().insertAll()<br>.execute() | update和insertAll由于目标列oldValue不在源中，因此操作会引发错误。 | 表架构更改为（key，oldValue，newValue）。updateAll更新列key和newValue，而oldValue保持不变，insertAll插入行（key、NULL、newValue）（也就是说，oldValue作为NULL插入）。 |

| 列 | 查询（在Scala中） | 没有架构演变的行为（默认值） | 有架构演变行为 |
| --- | --- | --- | --- |
| 目标列： key, oldValue<br>源列： key, newValue | targetDeltaTable.alias("t")<br>.merge(<br>sourceDataFrame.alias("s"),<br>"t.key = s.key")<br>.whenMatched().update(Map(<br>"newValue" -> col("s.newValue")))<br>.whenNotMatched().insertAll()<br>.execute() | update引发错误，因为目标表中不存在该列newValue。 | update仍然会引发错误，因为目标表中不存在该列newValue。 |
| 目标列： key, oldValue<br>源列： key, newValue | targetDeltaTable.alias("t")<br>.merge(<br>sourceDataFrame.alias("s"),<br>"t.key = s.key")<br>.whenMatched().updateAll()<br>.whenNotMatched().insert(Map(<br>"key" -> col("s.key"),<br>"newValue" -> col("s.newValue")))<br>.execute() | insert引发错误，因为目标表中不存在该列newValue。 | insert由于目标表中不存在该列newValue，因此仍然会引发错误。 |

## 性能调优

您可以使用以下方法减少merge所花费的时间：

- **减少匹配项的搜索空间：** 默认情况下，merge操作搜索整个Delta表以在源表中查找匹配项。加速merge的一种方法是通过在匹配条件中添加已知约束来减少搜索空间。例如，假设您有一个按country/date分区的表，并且希望使用merge更新最后一天和特定国家/地区的信息。添加条件

  SQL

  ```
  %sql
  events.date = current_date() AND events.country = 'USA'
  ```

  将加快查询速度，因为它仅在相关分区中查找匹配项。此外，这还将减少与其他并发操作发生冲突的机会。有关更多详细信息，请参见并发控制。

- **紧凑文件：** 如果数据存储在许多小文件中，则读取数据以搜索匹配项可能会变慢。您可以将小文件压缩为更大的文件，以提高读取吞吐量。有关详细信息，请参见压缩文件。
- **控制写入的无序分区：** merge操作多次对数据进行随机排列以计算和写入更新的数据。用于随机的任务数由Spark会话配置spark.sql.shuffle.partitions控制。设置此参数不仅可以控制并发度，还可以确定输出文件的数量。增加该值会提高并发度，但也会生成大量较小的数据文件。
- **启用优化写入：** 对于分区表，meage可以生成比随机分区数量多得多的小文件。这是因为每个随机任务都可以在多个分区中写入多个文件，并可能成为性能瓶颈。您可以通过启用优化写入来优化这一点。

## 合并范例

以下是一些有关如何merge在不同情况下使用的示例。

在这个部分：

- 写入Delta表时的重复数据删除
- 缓慢将数据（SCD）类型2操作更改为Delta表
- 将更改数据写入Delta表
- 使用Upsert 从流式处理查询foreachBatch

## 写入Delta表时的重复数据删除

一个常见的ETL用例是通过将日志附加到表中来将日志收集到Delta表中。但是，源通常可以生成重复的日志记录，因此需要下游重复数据删除步骤来处理它们。使用Merge，您可以避免插入重复的记录。

SQL

```
%sql
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId
WHEN NOT MATCHED
 THEN INSERT *
```

Python

```
%pyspark
deltaTable.alias("logs").merge(
  newDedupedLogs.alias("newDedupedLogs"),
  "logs.uniqueId = newDedupedLogs.uniqueId") \
 .whenNotMatchedInsertAll() \
 .execute()
```

Scala

```
%spark
deltaTable
 .as("logs")
 .merge(
  newDedupedLogs.as("newDedupedLogs"),
  "logs.uniqueId = newDedupedLogs.uniqueId")
 .whenNotMatched()
 .insertAll()
 .execute()
```

> 注意
>
> 包含新日志的数据集需要在其内部进行重复数据删除。通过合并的SQL语义，它将新数据与表中的现有数据进行匹配并删除重复数据，但是如果新数据集中存在重复数据，则将其插入。因此，在合并到表之前，对新数据进行重复数据删除。

如果您知道几天之内可能会得到重复的记录，则可以通过按日期对表进行分区，然后指定要匹配的目标表的日期范围来进一步优化查询。

SQl

```
%sql
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId AND logs.date > current_date() - INTERVAL 7 DAYS
WHEN NOT MATCHED AND newDedupedLogs.date > current_date() - INTERVAL 7 DAYS
 THEN INSERT *
```

Python

```
%pyspark
deltaTable.alias("logs").merge(
  newDedupedLogs.alias("newDedupedLogs"),
  "logs.uniqueId = newDedupedLogs.uniqueId") \
 .whenNotMatchedInsertAll() \
 .execute()
```

Scala

```
%pyspark
deltaTable
 .as("logs")
 .merge(
  newDedupedLogs.as("newDedupedLogs"),
  "logs.uniqueId = newDedupedLogs.uniqueId")
 .whenNotMatched()
 .insertAll()
 .execute()、
```

这比以前的命令效率更高，因为它仅在日志的最后7天而不是整个表中查找重复项。此外，您可以将此仅插入合并与结构化流一起使用，以执行日志的连续重复数据删除。

- 在流查询中，可以使用foreachBatch中的merge操作将具有重复数据删除功能的所有流数据连续写入Delta表。
- 在另一个流式查询中，可以连续从该Delta表中读取已删除重复的数据。这是可能的，因为insert-only merge只向Delta表追加新数据。

> **注意**
>
> insert-only merge被优化为仅在Databricks Runtime 6.2及更高版本中追加数据。在Databricks Runtime 6.1及更低版本中，insert-only merge操作的写入不能作为流读取。

## 缓慢将数据（SCD）Type2操作更改为Delta表

另一个常见的操作是SCD Type2，它维护维度表中每个键所做的所有更改的历史记录。这样的操作需要更新现有行以将key的以前值标记为旧值，并将新行作为最新值插入。给定一个包含更新的源表和包含维度数据的目标表，SCD Type2可以用merge表示。

下面是一个具体的例子，它维护客户的地址历史以及每个地址的活动日期范围。当客户的地址需要更新时，您必须将以前的地址标记为非当前地址，更新其活动日期范围，并将新地址添加为当前地址。

**使用合并笔记本的SCD 类型二**

Note链接地址：SCD Type 2 using Merge(Scala)

## 将更改数据写入Delta表

与SCD类似，另一个常见用例，通常称为变更数据捕获（CDC），是将从外部数据库生成的所有数据更改应

用到Delta表中。换句话说，应用于外部表的更新、删除和插入集需要应用于Delta表。您可以使用merge执行以下操作。

**使用Merge笔记本写入更改数据**

Note链接地址：CDC using Merge(Scala)

### 使用Upsert 从流式处理查询foreachBath

您可以结合使用merge和foreachBatch（有关更多信息，请参阅foreachBatch）将流式处理查询中的复杂更新操作写入Delta表。例如：

- **在更新模式下写入流式聚合：**这比Complete Mode要高效得多。
- **将数据库更改流写入Delta表：**用于写入更改数据的合并查询可用于foreachBatch中，以连续地将更改流应用于Delta表。
- **使用重复数据删除将流数据写入Delta表：**用于重复数据删除的 insert-only merge 查询可以在 foreachBatch 中使用自动重复数据删除功能将数据（带有重复项）连续写入到 Delta 表中。

> **说明**
>
> - 确保foreachBatch中的merge语句是等幂的，因为流查询的重新启动可以多次对同一批数据应用该操作。
> - 在foreachBatch中使用merge时，流式查询的输入数据速率（通过StreamingQueryProgress报告并在笔记本速率图中可见）可以报告为在源处生成数据的实际速率的倍数。这是因为merge多次读取输入数据，导致输入指标成倍增加。如果这是一个瓶颈，可以在merge之前缓存批处理数据帧，然后在merge后取消缓存。

**使用merge和foreachBatch笔记本以更新模式编写流式聚合**

**Note链接地址：**Upsert streaming aggregates using foreachBatch and Merge(Scala)

# 3.7. 表实用程序命令

Delta 表支持许多实用程序命令。

> **说明**
>
> 详细文章请参考Databricks官网文章：表实用程序命令

有关演示这些功能的Databricks笔记本，请参阅入门笔记本二。

### 删除Delta表不再引用的文件

您可以通过在表上运行vacuum命令来删除Delta表不再引用且早于保留阈值的文件。vacuum 不会自动触发。文件的默认保留阀值为7天。

> **警告**
>
> - vacuum仅删除数据文件，而不删除日志文件。检查点操作后，日志文件将自动异步删除。日志文件的默认保留期为30天，可通过使用ALTER TABLE SET TBLPROPERTIES SQL方法设置的delta.logRetentionPeriod属性进行配置。请参阅表属性。
> - 运行vacuum后，无法再按时间顺序查看在保留期之前创建的版本。

SQL

```
%sql
VACUUM eventsTable   -- vacuum files not required by versions older than the default retention period

VACUUM '/data/events' -- vacuum files in path-based table

VACUUM delta.`/data/events/`

VACUUM delta.`/data/events/` RETAIN 100 HOURS  -- vacuum files not required by versions more than 100 h
ours old

VACUUM eventsTable DRY RUN    -- do dry run to get the list of files to be deleted
```

Python

```
%pyspark
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, pathToTable)  # path-based tables, or
deltaTable = DeltaTable.forName(spark, tableName)    # Hive metastore-based tables

deltaTable.vacuum()      # vacuum files not required by versions older than the default retention period

deltaTable.vacuum(100)    # vacuum files not required by versions more than 100 hours old
```

Scala

```
%spark
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, pathToTable)

deltaTable.vacuum()      // vacuum files not required by versions older than the default retention period

deltaTable.vacuum(100)    // vacuum files not required by versions more than 100 hours old
```

有关语法的详细信息，请参见

- Databricks Runtime 7.0及更高版本：VACUUM
- Databricks Runtime 6.x及以下：VACUUM
- 有关Scala，Java和Python语法的详细信息，请参见API参考。

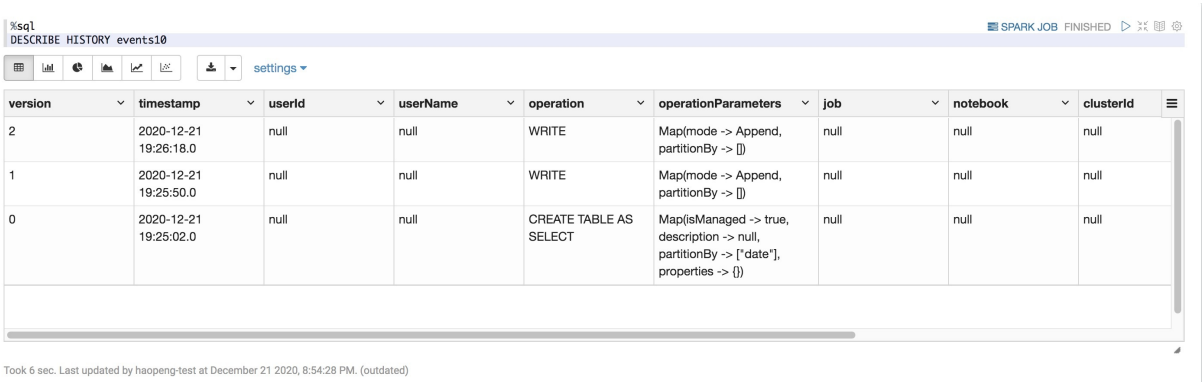
> **警告**
>
> 我们不建议您将保留间隔设置为短于7天，因为并发的读取器或写入器仍然可以将旧快照和未提交的文件用于表。如果 vacuum清除活动文件，则并发阅读器可能会失败，或者更糟的是，当vacuum删除尚未提交的文件时，表可能会损坏。
>
> Delta Lake具有一项安全检查，以防止您执行危险的vacuum命令。如果您确定在此表上执行的操作没有超过计划指定的保留时间间隔，你可以通过设置ApacheSpark属性spark.databricks.delta.retentionDurationCheck.enabled设置为false来关闭此安全检查。选择的时间间隔，必须比最长的并发事务长，也必须比任何流可以滞后于表的最新更新的最长时间长。

## 检索Delta表历史记录

您可以通过运行history命令来检索每次写入Delta表的操作、用户、时间戳等信息。以相反的时间顺序返回操作。默认情况下，表历史记录会保留30天。

SQL

```
%sql
DESCRIBE HISTORY '/data/events/'       -- get the full history of the table

DESCRIBE HISTORY delta.`/data/events/`

DESCRIBE HISTORY '/data/events/' LIMIT 1  -- get the last operation only

DESCRIBE HISTORY eventsTable
```




Python

```
%pyspark
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, pathToTable)

fullHistoryDF = deltaTable.history()  # get the full history of the table

lastOperationDF = deltaTable.history(1) # get the last operation
```

Scala

```
%spark
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, pathToTable)

val fullHistoryDF = deltaTable.history()  // get the full history of the table

val lastOperationDF = deltaTable.history(1) // get the last operation
```

有关Spark SQL语法的详细信息，请参见

- Databricks Runtime 7.0及更高版本：DESCRIBE HISTORY (Delta Lake on Databricks)
- Databricks Runtime 6.x及更低版本：Describe History (Delta Lake on Databricks)

有关Scala，Java和Python语法的详细信息，请参见API参考。

**历史架构**

| 列 | 类型 | 说明 |
| --- | --- | --- |
| 版本 | long | 通过操作生成的表版本 |
| timestamp | timestamp | 提交此版本的时间 |
| userId | 字符串 | 运行操作的用户的ID |
| userName | 字符串 | 运行操作的用户的姓名。 |
| operation | 字符串 | 操作的名称。 |

| 列 | 类型 | 说明 |
| --- | --- | --- |
| operationParameters | map | 操作的参数（例如谓词。） |
| 作业（job） | struct | 运行操作的作业的详细信息。 |
| 笔记本 | struct | 运行操作的笔记本的详细信息。 |
| clusterId | 字符串 | 运行操作的群集的 ID。 |
| readVersion | long | 读取以执行写入操作的表的版本。 |
| isolationLevel | 字符串 | 用于此操作的隔离级别。 |
| isBlindAppend | boolean | 此操作是否追加数据。 |
| operationMetrics | map | 操作的指标（例如已修改的行数和文件数。） |
| userMetadata | 字符串 | 用户定义的提交元数据（如果已指定） |

该history操作的输出包含以下列。

```
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+-----------------+---------
----+--------------------+
|version|       timestamp|userId|userName|operation| operationParameters| job|notebook|clusterId|readVer
sion|  isolationLevel|isBlindAppend|    operationMetrics|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+-----------------+---------
----+--------------------+
|      5|2019-07-29 14:07:47|  ###|    ###|  DELETE|[predicate -> ["(...|null|    ###|    ###|       4|WriteSerializable|
false|[numTotalRows -> ...|
|      4|2019-07-29 14:07:41|  ###|    ###|  UPDATE|[predicate -> (id...|null|    ###|    ###|       3|WriteSerializable|
false|[numTotalRows -> ...|
|      3|2019-07-29 14:07:29|  ###|    ###|  DELETE|[predicate -> ["(...|null|    ###|    ###|       2|WriteSerializable|
false|[numTotalRows -> ...|
|      2|2019-07-29 14:06:56|  ###|    ###|  UPDATE|[predicate -> (id...|null|    ###|    ###|       1|WriteSerializable|
false|[numTotalRows -> ...|
|      1|2019-07-29 14:04:31|  ###|    ###|  DELETE|[predicate -> ["(...|null|    ###|    ###|       0|WriteSerializable|
false|[numTotalRows -> ...|
|      0|2019-07-29 14:01:40|  ###|    ###|   WRITE|[mode -> ErrorIfE...|null|    ###|    ###|    null|WriteSerializable|
true|[numFiles -> 2, n...|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+-----------------+---------
----+--------------------+
```

**说明**

- 仅当使用 Databricks Runtime 6.5 或更高版本运行历史记录中的 history 命令和操作时，操作指标才可用。
- 如果使用以下方法写入Delta表，则其他一些列将不可用：
  - JDBC或ODBC
  - JAR工作
  - Spark提交工具
  - 使用REST API运行命令
- 将来添加的列将始终添加在最后一列之后。

## 操作指标键

该history操作在operationMetrics列映射中返回操作指标的集合。

下表按操作列出了映射键定义。

| 操作方式 | 指标名称 | 描述 |
| --- | --- | --- |
| WRITE, CREATE TABLE AS SELECT, REPLACE TABLE AS SELECT, COPY INTO | | |
| | numFiles | 写入的文件数。 |
| | numOutputBytes | 已写入的内容的大小（以字节为单位）。 |
| | numOutputRows | 写入的行数。 |
| STREAMING UPDATE | | |
| | numAddedFiles | 添加的文件数。 |
| | numRemovedFiles | 删除的文件数。 |
| | numOutputRows | 写入的行数。 |
| | numOutputBytes | 写入大小（以字节为单位）。 |
| DELETE | | |
| | numAddedFiles | 添加的文件数。 删除表的分区时未提供。 |
| | numRemovedFiles | 删除的文件数。 |
| | numDeletedRows | 删除的行数。 删除表的分区时未提供。 |
| | numCopiedRows | 在删除文件期间复制的行数。 |
| TRUNCATE | numRemovedFiles | 删除的文件数。 |

| 操作方式 | 指标名称 | 描述 |
| --- | --- | --- |
| MERGE | | |
| | numSourceRows | 源数据帧中的行数。 |
| | numTargetRowsInserted | 插入到目标表的行数。 |
| | numTargetRowsUpdated | 目标表中更新的行数。 |
| | numTargetRowsDeleted | 目标表中删除的行数。 |
| | numTargetRowsCopied | 复制的目标行数。 |
| | numOutputRows | 写出的总行数。 |
| | numTargetFilesAdded | 添加到接收器（目标）的文件数。 |
| | numTargetFilesRemoved | 从接收器（目标）删除的文件数。 |
| UPDATE | | |
| | numAddedFiles | 添加的文件数 |
| | numRemovedFiles | 删除的文件数。 |
| | numUpdatedRows | 更新的行数。 |
| | numCopiedRows | 刚才在更新文件期间复制的行数。 |
| FSCK | numRemovedFiles | 删除的文件数。 |
| CONVERT | numConvertedFiles | 已转换的 Parquet 文件数。 |

| 操作方式 | 指标名称 | 描述 |
| --- | --- | --- |
| CLONE | | |
| | sourceTableSize | 所克隆版本的源表的大小（以字节为单位）。 |
| | sourceNumOfFiles | 源表中已克隆版本的文件数。 |
| | numRemovedFiles | 目标表中删除的文件数（如果替换了先前的 Delta 表）。 |
| | removedFilesSize | 如果替换了先前的 Delta 表，则为目标表中删除文件的总大小（以字节为单位）。 |
| | numCopiedFiles | 复制到新位置的文件数。 如果是浅表克隆，则为 0。 |
| | copiedFilesSize | 复制到新位置的文件总大小（以字节为单位）。 如果是浅表克隆，则为 0。 |
| RESTORE | | |
| | tableSizeAfterRestore | 还原后的表大小（以字节为单位）。 |
| | numOfFilesAfterRestore | 还原后表中的文件数。 |
| | numRemovedFiles | 还原操作删除的文件数。 |
| | numRestoredFiles | 由于还原而添加的文件数。 |
| | removedFilesSize | 还原删除的文件的大小（以字节为单位）。 |
| | restoredFilesSize | 还原添加的文件的大小（以字节为单位）。 |

| 操作方式 | 指标名称 | 描述 |
| --- | --- | --- |
| OPTIMIZE | | |
| | numAddedFiles | 添加的文件数。 |
| | numRemovedFiles | 优化的文件数。 |
| | numAddedBytes | 优化表后添加的字节数。 |
| | numRemovedBytes | 删除的字节数。删除的字节数。 |
| | minFileSize | 优化表后最小文件的大小。 |
| | p25FileSize | 优化表后第 25 个百分位文件的大小。 |
| | p50FileSize | 优化表后的文件大小中值。 |
| | p75FileSize | 优化表后第 75 个百分位文件的大小。 |
| | maxFileSize | 优化表后最大文件的大小。 |

- 需要Databricks Runtime 7.3或更高版本。
- 需要Databricks Runtime 7.4或更高版本。

## 检索Delta表详细信息

可以使用“描述详细信息”检索有关增量表的详细信息（例如，文件数、数据大小）。

SQL

```
%sql
DESCRIBE DETAIL '/data/events/'

DESCRIBE DETAIL eventsTable
```

**详细架构**

此操作的输出只有一行具有以下架构。

| 列 | 类型 | 说明 |
| --- | --- | --- |
| format | 字符串 | 表的格式，即“delta”。 |
| id | 字符串 | 表的唯一 ID。 |
| name | 字符串 | 在元存储中定义的表名称。 |
| description | 字符串 | 表的说明。 |
| location | 字符串 | 表的位置。 |
| createdAt | timestamp | 表创建时间。 |
| lastModified | timestamp | 表的上次修改时间。 |
| partitionColumns | 字符串数组 | 如果表已分区，则为分区列的名称。 |
| numFiles | long | 表最新版本中的文件数。 |
| properties | string-string map | 此表的所有属性集。 |
| minReaderVersion | int | 可读取表的读取器最低版本（由日志协议而定）。 |
| minWriterVersion | int | 可写入表的写入器最低版本（由日志协议而定）。 |

```
+------+--------------------+-----------------+-----------+--------------------+--------------------+--------------------+---------------+--
------+-----------+----------+---------------+---------------+
|format|                  id|             name|description|            location|           createdAt|        lastModified|partitionColumns|
numFiles|sizeInBytes|properties|minReaderVersion|minWriterVersion|
+------+--------------------+-----------------+-----------+--------------------+--------------------+--------------------+---------------+--
------+-----------+----------+---------------+---------------+
| delta|d31f82d2-a69f-42e...|default.deltatable|       null|file:/Users/tdas/...|2020-06-05 12:20:...|2020-06-05 12:2
0:20|          []|    10|    12345|     []|           1|           2|
+------+--------------------+-----------------+-----------+--------------------+--------------------+--------------------+---------------+--
------+-----------+----------+---------------+---------------+
```

## 生成清单文件

您可以为Delta表生成清单文件，供其他处理引擎（即Apache Spark以外的其他引擎）用来读取Delta表。例如，要生成清单文件，Presto和Athena可以使用它们来读取Delta表，请运行以下命令：

SQL

```
%sql
GENERATE symlink_format_manifest FOR TABLE delta.`/mnt/events`

GENERATE symlink_format_manifest FOR TABLE eventsTable
```

Python

```
%pyspark
deltaTable = DeltaTable.forPath(<path-to-delta-table>)
deltaTable.generate("symlink_format_manifest")
```

Scala

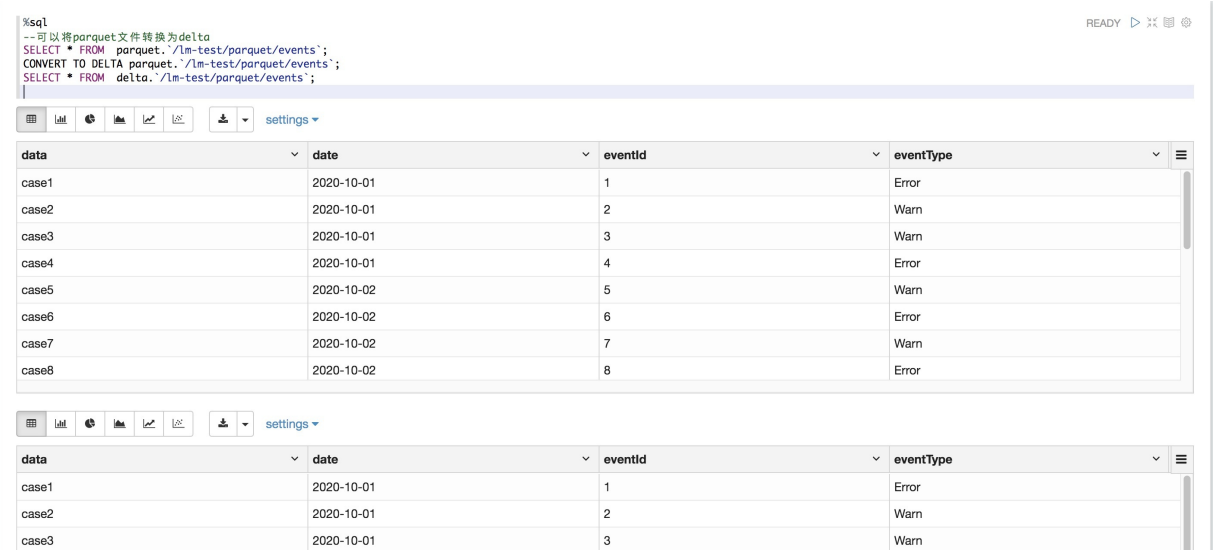
```
%spark
val deltaTable = DeltaTable.forPath(<path-to-delta-table>)
deltaTable.generate("symlink_format_manifest")
```

## 将Parquet表转换为Delta表

就地将Parquet表转换为Delta表。此命令会列出目录中的所有文件，创建一个Delta Lake事务日志来跟踪这些文件，并通过读取所有Parquet文件的页脚自动推断数据架构。如果您的数据已分区，则必须将分区列的架构指定为DDL格式的字符串（即）。<column-name1> <type>, <column-name2> <type>, ...

SQL

```
%SQL
-- Convert unpartitioned parquet table at path '<path-to-table>'
CONVERT TO DELTA parquet.`<path-to-table>`

-- Convert partitioned Parquet table at path '<path-to-table>' and partitioned by integer columns named 'part' and 'part2'
CONVERT TO DELTA parquet.`<path-to-table>` PARTITIONED BY (part int, part2 int)
```




Python

```
%pyspark
from delta.tables import *

# Convert unpartitioned 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")
```

**注意**

可在 Databricks Runtime 6.1 及更高版本中使用 Python API。

Scala

```
%spark
import io.delta.tables._

// Convert unpartitioned Parquet table at path '<path-to-table>'
val deltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-table>`")

// Convert partitioned Parquet table at path '<path-to-table>' and partitioned by integer columns named 'part' and 'part2'
val partitionedDeltaTable = DeltaTable.convertToDelta(spark, "parquet.`<path-to-table>`", "part int, part2 int")
```

> **注意**
>
> 可在 Databricks Runtime 6.0 及更高版本中使用 Scala API。

有关语法的详细信息，请参见

- Databricks运行时7.0及更高版本：转换为DELTA（Databricks上的Delta Lake）
- Databricks Runtime 6.x及更低版本：转换为Delta（Databricks上的Delta Lake）

> **注意**
>
> Delta Lake跟踪的文件都是不可见的，运行vacuum时可以删除。您勿在转换过程中更新或附加数据文件。转换表后，请确保通过Delta Lake写入所有文件。

## 将Delta表转换为Parquet表

您可以使用以下步骤轻松地将Delta表转换回Parquet表：

1. 如果执行了可以更改数据文件的Delta Lake操作（例如delete或merge），请运行vacuum并将保留期限设为0小时，从而删除表的最新版本中未包含的所有数据文件。
2. 删除目录中的_delta_log目录。

## 将Delta表还原到较早的状态

> **说明**
>
> 此功能目前以 公共预览版提供。

> **注意**
>
> 在Databricks Runtime 7.4及更高版本中可用。

您可以使用以下RESTORE命令将Delta表还原到其早期状态。Delta表在内部维护该表的历史版本，以使其能够还原到较早的状态。RESTORE命令支持与早期状态相对应的版本或创建早期状态的时间戳。

警告

- 您可以还原已经还原的表和克隆的表。
- 将表还原到手动或删除数据文件的旧版本vacuum将失败。如果spark.sql.files.ignoreMissingFiles设置为true,仍然可以部分还原到该版本。
- 恢复到较早状态的时间戳格式为yyyy-MM-dd HH:mm:ssyyyy-MM-dd。还支持仅提供date（）字符串。

SQL

```
%sql
RESTORE TABLE db.target_table TO VERSION AS OF <version>
RESTORE TABLE delta.`/data/target/` TO TIMESTAMP AS OF <timestamp>
```

Python

```
%pyspark
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, <path-to-table>)  # path-based tables, or
deltaTable = DeltaTable.forName(spark, <table-name>)    # Hive metastore-based tables

deltaTable.restoreToVersion(0) # restore table to oldest version

deltaTable.restoreToTimestamp('2019-02-14') # restore to a specific timestamp
```

Scala

```
%spark
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, <path-to-table>)
val deltaTable = DeltaTable.forName(spark, <table-name>)

deltaTable.restoreToVersion(0) // restore table to oldest version

deltaTable.restoreToTimestamp("2019-02-14") // restore to a specific timestamp
```

有关语法的详细信息，请参见RESTORE（Databricks上的Delta Lake）。

**表访问控制**

您必须MODIFY对要还原的表具有权限

# 克隆 Delta 表

> **说明**
>
> 此功能目前以 公共预览版提供。

> **注意**
>
> 在Databricks Runtime 7.2及更高版本中可用。

您可以使用以下clone命令以特定版本创建现有Delta表的副本。克隆可以是深层或浅层。

- 深层克隆 是指除了现有表的元数据外，还会将源表数据复制到克隆目标的克隆。 此外，它还会克隆流元数据，使写入 Delta 表的流可在源表上停止，并在克隆的目标位置（即停止位置）继续进行克隆。
- 浅表克隆不会将数据文件复制到克隆目标。 表元数据等效于源。 创建这些克隆的成本较低。

对深层克隆或浅层克隆所做的任何更改都只会影响克隆本身，而不会影响源表。

克隆的元数据包括：架构，分区信息，不变量，可为Null性。对于深层克隆，还将克隆流和COPY INTO（Databricks上的Delta Lake）元数据。未克隆的元数据是表描述和用户定义的提交元数据。

> **警告**
>
> - 浅克隆引用源目录中的数据文件。如果在源表上运行vacuum，客户端将无法再读取引用的数据文件，并将引发FileNotFoundException。在这种情况下，在浅层克隆上运行clone with replace将修复克隆。如果经常发生这种情况，请考虑使用不依赖于源表的深层克隆。
> - 深度克隆不依赖于其克隆来源，因为深度克隆会复制数据以及元数据，所以创建深度克隆的成本很高。
> - 使用 replace 克隆到已在该路径具有表的目标时，如果该路径不存在，会创建一个 Delta 日志。您可以通过vacuum运行清理任何现有数据。如果现有表是Delta表，则会在现有Delta表上创建新的提交，其中包括源表中的新元数据和新数据。
> - 克隆表与 Create Table As Select 或 CTAS 不同，除了数据之外，克隆还复制源表的元数据。克隆的语法更为简单：不需要指定分区、格式、不变量、可Null性等，因为它们取自源表。
> - 克隆表与其具有源表无关的历史记录。在克隆表上的按时间查询时，这些查询使用的输入与它们在其源表上查询时使用的不同。

SQL

```
%sql
CREATE TABLE delta.`/data/target/` CLONE delta.`/data/source/` -- Create a deep clone of /data/source at /data/target

CREATE OR REPLACE TABLE db.target_table CLONE db.source_table -- Replace the target

CREATE TABLE IF NOT EXISTS TABLE delta.`/data/target/` CLONE db.source_table -- No-op if the target table exists

CREATE TABLE db.target_table SHALLOW CLONE delta.`/data/source`

CREATE TABLE db.target_table SHALLOW CLONE delta.`/data/source` VERSION AS OF version

CREATE TABLE db.target_table SHALLOW CLONE delta.`/data/source` TIMESTAMP AS OF timestamp_expression -- timestamp can be like “2019-01-01” or like date_sub(current_date(), 1)
```

Python

```
%pyspark
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, pathToTable)  # path-based tables, or
deltaTable = DeltaTable.forName(spark, tableName)    # Hive metastore-based tables

deltaTable.clone(target, isShallow, replace) # clone the source at latest version

deltaTable.cloneAtVersion(version, target, isShallow, replace) # clone the source at a specific version

# clone the source at a specific timestamp such as timestamp=“2019-01-01”
deltaTable.cloneAtTimestamp(timestamp, target, isShallow, replace)
```

Scala

```
%spark
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, pathToTable)
val deltaTable = DeltaTable.forName(spark, tableName)

deltaTable.clone(target, isShallow, replace) // clone the source at latest version

deltaTable.cloneAtVersion(version, target, isShallow, replace) // clone the source at a specific version

deltaTable.cloneAtTimestamp(timestamp, target, isShallow, replace) // clone the source at a specific timest
amp
```

有关语法的详细信息，请参见CLONE（Dat abricks上的Delt a Lake）

**权限**

您必须CLONEDat abricks表访问控制和您的云提供的所需的权限。

**表访问控制**

深层和浅层克隆都需要具有以下权限：

- 源表上的SELECT权限
- 如果要CLONE用于创建新表，请对创建表的数据库具有CREAT E权限。
- 如果要CLONE用来替换表，则必须具有该表的MODIFY权限。

**云权限**

如果创建了深度克隆，则任何读取该深度克隆的用户都必须具有对该克隆目录的读取权限。要更改克隆，用户必须具有对克隆目录的写入权限。

如果创建了浅表克隆，则读取浅表克隆的任何用户都需要权限才能读取原始表中的文件，因为数据文件保留在源表中，并且包含浅表克隆以及该表的目录。若要要更改克隆，用户需要对克隆目录具有写入权限。

## 克隆用例

- 数据存档：数据保存的时间可能会比按时间查看或灾难恢复的时间更长。在这些情况下，您可以创建一个深层克隆，保留表的某个时间点的状态以供存档。还可以通过增量存档保留源表的持续更新状态，以进行灾难恢复。

  SQL

  ```
  %sql
  -- Every month run
  CREATE OR REPLACE TABLE delta.`/some/archive/path` CLONE my_prod_table
  ```

- 机器学习流重现：在进行机器学习时，你可能希望将已训练 ML 模型的表的特定版本进行存档。可以使用此存档数据集测试将来的模型。

  SQL

```
%sql
-- Trained model on version 15 of Delta table
CREATE TABLE delta.`/model/dataset` CLONE entire_dataset VERSION AS OF 15
```

- 在生产表上进行短期实验：为了在不损坏表的情况下测试生产表中的工作流，可以轻松创建浅表克隆。这样，就可在包含所有生产数据的克隆表上运行任意工作流，而不会影响任何生产工作负载。

  SQL

```
%sql
-- Perform shallow clone
CREATE OR REPLACE TABLE my_test SHALLOW CLONE my_prod_table;

UPDATE my_test WHERE user_id is null SET invalid=true;
-- Run a bunch of validations. Once happy:

-- This should leverage the update information in the clone to prune to only
-- changed files in the clone if possible
MERGE INTO my_prod_table
USING my_test
ON my_test.user_id <=> my_prod_table.user_id
WHEN MATCHED AND my_test.user_id is null THEN UPDATE *;

DROP TABLE my_test;
```

- 数据共享：单个组织内的其他业务部门可能希望访问上述的数据，但可能不需要最新更新。可以为不同的业务部门提供具有不同权限的克隆，而不是直接授予对源表的访问权限。克隆的性能比简单视图的性能更高。

  SQL

```
%sql
-- Perform deep clone
CREATE OR REPLACE TABLE shared_table CLONE my_prod_table;

-- Grant other users access to the shared table
GRANT SELECT ON shared_table TO `<user-name>@<user-domain>.com`;
```

# 3.8. 约束条件

> **说明**
>
> 详细内容请参考Databricks官网文章：约束条件

> 注意
>
> 在Databricks Runtime 7.4及更高版本中可用。

Delta表支持标准的SQL约束管理子句，以确保自动验证添加到表中的数据的质量和完整性。当违反约束时，Delta Lake会抛出一个InvariantViolationException信号，表示无法添加新数据。

支持两种类型的约束：

- NOT NULL：表示特定列中的值不能为Null。
- CHECK：表示每个输入行的指定布尔表达式必须为true

## NOT NULL约束

您在创建表时在架构中指定约束，然后使用命令删除约束。

NOT NULLNOT NULLALTER TABLE CHANGE COLUMN。

SQL

```
%sql
CREATE TABLE events(
 id LONG NOT NULL,
 date STRING NOT NULL,
 location STRING,
 description STRING
);

ALTER TABLE events CHANGE COLUMN date DROP NOT NULL;
```

您可以使用以下命令对添加到现有的Delta表进行约束。NOT NULLALTER TABLE CHANGE COLUMN

SQL

```
%sql
CREATE TABLE events(
 id LONG,
 date STRING,
 location STRING,
 description STRING
);

ALTER TABLE events CHANGE COLUMN id SET NOT NULL;
```

如果在嵌套在结构内的列上指定了约束，则父结构也将约束为不为Null。嵌套在数组或映射类型内的列不接受约束。NOT NULLNOT NULL

## CHECK 约束

> **注意**
>
> - 在Databricks Runtime 7.4及更高版本中可用。
> - 在Databricks Runtime 7.3中，您可以写入已CHECK定义约束的表，但不能创建CHECK约束。

您可以CHECK使用和命令管理约束。在将其添加到表之前，验证所有现有行均满足约束。

ALTER TABLE ADD CONSTRAINTALTER TABLE DROP CONSTRAINTALTER TABLE ADD CONSTRAINT

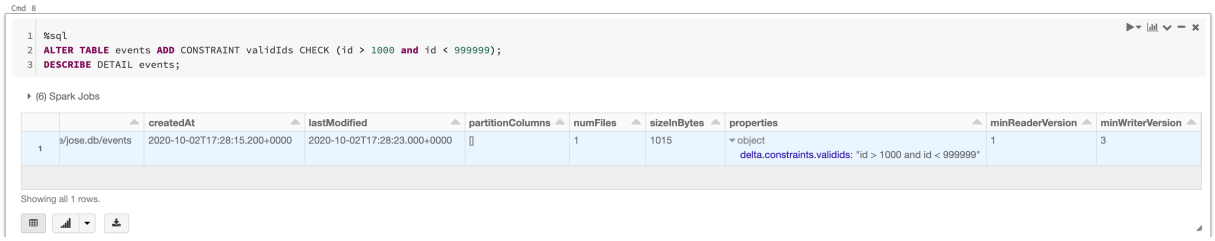
SQL

```
%sql
CREATE TABLE events(
 id LONG NOT NULL,
 date STRING,
 location STRING,
 description STRING
);

ALTER TABLE events ADD CONSTRAINT dateWithinRange CHECK date > '1900-01-01';
ALTER TABLE events DROP CONSTRAINT dateWithinRange;
```

CHECK约束在和命令的输出中显示为表属性。DESCRIBE DETAILSHOW TBLPROPERTIES

SQL

```
%sql
ALTER TABLE events ADD CONSTRAINT validIds CHECK (id > 1000 and id < 999999);
DESCRIBE DETAIL events;
```




# 3.9. 表版本控制

> **说明**
>
> 详细文章请参考Databricks官网文章：表版本控制

Delta表的事务日志包含支持Delta Lake演变的版本控制信息。Delta Lake分别跟踪最低检索Delta表详细信息。

Delta Lake保证向后兼容。较高版本的Databricks Runtime始终能够读取由较低版本写入的数据。

Delta Lake偶尔会突破兼容性。较低版本的Databricks Runtime可能无法读取和写入由较高版本的Databricks Runtime写入的数据。如果您尝试使用太低的Databricks Runtime版本来读取和写入表，则会收到一条错误消息，提示您需要升级。

创建表时，Delta Lake将根据表特征（例如架构或表属性）选择所需的最低协议版本。您还可以通过设置SQL配置来设置默认协议版本：

- spark.databricks.delta.protocol.minWriterVersion = 2 (default)
- spark.databricks.delta.protocol.minReaderVersion = 1 (default)

要将表升级到较新的协议版本，请使用以下DeltaTable.upgradeTableProtocol方法

Python

```
%pyspark
from delta.tables import DeltaTable
delta = DeltaTable.forPath(spark, "path_to_table") # or DeltaTable.forName
delta.upgradeTableProtocol(1, 3) # upgrades to readerVersion=1, writerVersion=3
```

Scala

```
%spark
import io.delta.tables.DeltaTable
val delta = DeltaTable.forPath(spark, "path_to_table") // or DeltaTable.forName
delta.upgradeTableProtocol(1, 3) // upgrades to readerVersion=1, writerVersion=3
```

**警告**

协议升级是不可逆的，因此我们建议您仅在需要时才升级特定表，例如选择加入Delta Lake中的新功能。

# 3.10. API参考

对于Delta表上最常见的读写操作，可以使用Apache Spark读取器和编写器API（请参阅表批读写和表流读写）。但是，有一些特定于Delta Lake的操作，您必须使用Delta Lake编程API。本文介绍了这些编程API。

**注意**

某些编程式API仍在不断发展，并在API文档中以“发展”限定附表示。

Databricks确保Delta Lake项目和Databricks Runtime中的Delta Lake之间的二进制兼容性。兼容性矩阵列出了每个Databricks Runtime版本中打包的Delta Lake API版本以及指向相应API文档的链接。

**说明**

详情文章请参考Databricks官网文章：API参考

# 3.11. 并发控制

> 说明
>
> 详情请参考Databricks官网文章：并发控制

Delta Lake在读取和写入之间提供ACID事务保证。这意味着：

- 跨多个集群的多个编写器可以同时修改表分区，并查看表的一致性快照视图，并且这些写入操作将具有序列顺序。
- 即使在作业过程中修改了某个表，读取器仍会继续查看Databricks 作业开始使用的表的一致快照视图。

## 乐观并发控制

Delta Lake使用乐观并发控制 来提供两次写入之间的事务保证。在这种机制下，写操作分为三个阶段：

1. **读取**：读取（如果需要）表的最新可用版本，以标识需要修改（即重写）的文件。
2. **写入**：通过写入新数据文件来暂存所有更改
3. **验证并提交**：在提交更改之前，检查**建议的更改**是否与**自读取快照后并发提交的任何其他更改**冲突。如果没有冲突，则所有已转移的更改都将作为新版本的快照提交，并且成功写入操作。但是，如果存在冲突，则写入操作失败，并出现并发修改异常，而不是像对Parquet表执行写操作那样损坏表。

表的隔离级别定义了必须将事务与并发操作所做的修改隔离的程度。有关在Databricks上的Delta Lake支持的隔离级别的信息，请参阅隔离级别。

## 写冲突

下表描述了在每个隔离级别中哪些写操作对可能发生冲突。

| | 插入 | 更新，删除，合并 | 平台 |
|---|---|---|---|
| 插入 | 不能冲突 | | |
| 更新，删除，合并 | 在可序列化时可能会冲突，WriteSerializable 中不能发生冲突 | 在可序列化和 WriteSerializable 中可能发生冲突 | |
| 平台 | 不能冲突 | 在可序列化和 WriteSerializable 中可能发生冲突 | 在可序列化和 WriteSerializable 中可能发生冲突 |

## 使用分区和非连续命令条件来避免冲突

在所有标记为“可能冲突”的情况下，这两个操作是否会发生冲突取决于它们是否对同一组文件进行操作。可以通过将表分区为与操作条件中使用的列相同的列来使两组文件不相交。 例如， `UPDATE table WHERE date > '2010-01-01' ...` `DELETE table WHERE date < '2010-01-01'` 如果表未按日期分区，则这两个命令和将发生冲突，因为这两个命令都可以尝试修改相同的一组文件。 对表进行分区 `date` 将避免冲突。 因此，根据通常用于命令的条件对表进行分区可以显著减少冲突。 不过，按基数较高的列对表进行分区可能会导致由大量子目录引起的其他性能问题。

## 冲突例外

发生事务冲突时，您将观察到以下异常之一：

- ConcurrentAppendException
- ConcurrentDeleteReadException
- ConcurrentDeleteDeleteException
- MetadataChangedException
- ConcurrentTransactionException
- ProtocolChangedException

### ConcurrentAppendException

当并发操作在操作读取的同一分区（或未分区表中的任何位置）中添加文件时，会发生异常。该文件增加的部分可以由插入、删除、更新或合并操作引起。

使用默认隔离级别为WriteSerializable，盲 INSERT操作（即，在未读取任何数据的情况下盲目追加数据）添加的文件不会与任何操作冲突，即使它们接触相同的分区（或未分区表中的任何位置）也是如此。如果隔离级别设置为Serializable，则盲追加加可能会冲突。

DELETE，UPDATE或MERGE并发操作经常引发异常。尽管并发操作可能会物理上更新不同的分区目录，但其中一个操作可能会读取与其他分区目录同时更新的同一分区，从而导致冲突。可以通过在操作条件中进行分隔显式来避免这种情况。 请考虑以下示例。

Scala

```
%spark
// Target 'deltaTable' is partitioned by date and country
deltaTable.as("t").merge(
  source.as("s"),
  "s.user_id = t.user_id AND s.date = t.date AND s.country = t.country")
 .whenMatched().updateAll()
 .whenNotMatched().insertAll()
 .execute()
```

假设您在不同的日期或国家/地区同时运行上述代码。由于每个作业都在目标Delta表上的独立分区上运行，因此您不会遇到任何冲突。但是，该条件不够明确，可以扫描整个表，并且可能与更新任何其他分区的并发操作冲突。相反，您可以重写语句以将特定日期和国家/地区添加到合并条件中，如以下示例所示。

Scala

```
%spark
// Target 'deltaTable' is partitioned by date and country
deltaTable.as("t").merge(
  source.as("s"),
  "s.user_id = t.user_id AND s.date = t.date AND s.country = t.country AND t.date = '" + <date> + "' AND t.country = '" + <country> + "'")
 .whenMatched().updateAll()
 .whenNotMatched().insertAll()
 .execute()
```

现在可以安全地在不同日期和国家/地区同时运行此操作。

### ConcurrentDeleteReadException

当并发操作删除您的操作读取的文件时，会发生此异常。常见的原因是DELETE，UPDATE或MERGE操作时重写文件。

### ConcurrentDeleteDeleteException

当并发操作删除了操作还删除的文件时，会发生此异常。 这可能是由两个并发 操作重写相同文件引起的。

### MetadataChangedException

当并发事务更新增量表的元数据时，将发生此异常。 常见的原因是 `ALTER TABLE` 操作或写入Delta表，用于更新表的架构

### ConcurrentTransactionException

如果使用相同检查点位置的流式处理查询同时启动多次，并尝试同时写入Delta表。 永远不应让两个流式处理查询使用相同的检查点位置并同时运行。

### ProtocolChangedException

当您的Delta表升级到新版本时，就会发生这种情况。为了使将来的操作成功，您可能需要升级Delta Lake版本。

# 3.12. 迁移指南

> **说明**
>
> 详情可参考Databricks官网文章：迁移指南

## 将工作负载迁移到Delta Lake

当您将工作负载迁移到Delta-Lake时，您应该注意到以下简化和与apachespark和apachehive提供的数据源相比的区别。

Delta Lake自动处理以下操作，您永远不要手动执行这些操作：

- REFRESH TABLE：Delta表始终返回最新信息，因此在更改之后不需要手动调用REFRESH TABLE。
- 添加和删除分区：Delta lake自动跟踪表中的分区集，并在添加或删除数据时更新列表。因此，不需要运行ALTER TABLE[ADD | DROP]PARTITION或MSCK。

- 加载单个分区：作为一种优化，有时您可能会直接加载您感兴趣的数据分区。例如，spark.read.parquet("/data/date=2017-01-01")。Delta Lake不需要这样做，因为它可以从事务日志中快速读取文件列表以找到相关文件。如果您对单个分区感兴趣，请使用WHERE子句指定它。例如：spark.read.delta("/data").where("date = '2017-01-01'"). 对于分区中有许多文件的大型表，这可能比从Parquet表加载单个分区（使用直接分区路径或WHERE）要快得多，因为在目录中列出文件通常比从事务日志中读取文件列表慢。

将现有应用程序移植到Delta Lake时，应避免执行以下操作，这些操作会绕过事务日志：

- 手动修改数据：Delta Lake使用事务日志将更改自动提交到表中。因为日志是事实的来源，所以Spark不会读取已写出但未添加到事务日志中的文件。同样，即使您手动删除文件，事务日志中仍然存在指向该文件的指针。始终使用本指南中描述的命令来代替手动修改存储在Delta表中的文件。
- 外部读取器：直接读取存储在Delta Lake中的数据。有关如何读取Delta表的信息，请参阅Integrations。

**示例** 假设您已将Parquet数据存储在directory/data-pipeline 中，并希望创建一个名为events的表。您始终可以读入DataFrame并另存为Delta表。这种方法复制数据，并让Spark管理表。另外，您可以转换到较快的Delta Lake，但会导致表格不受管理。

**另存为Delta表**

1. 将数据读入DataFrame并将其保存为以下delta格式的新目录：

   Python

   ```
   %pyspark
   data = spark.read.parquet("/data-pipeline")
   data.write.format("delta").save("/mnt/delta/data-pipeline/")
   ```

2. 创建一个Delta表events，该表引用Delta Lake目录中的文件：

   Python

   ```
   %pyspark
   spark.sql("CREATE TABLE events USING DELTA LOCATION '/mnt/delta/data-pipeline/'")
   ```

**转换为增量表**

您有两种选择将Parquet表转换为Delta表：

- 将文件转换为Delta Lake格式并创建Delta表：

  SQL

  ```
  %sql
  CONVERT TO DELTA parquet.`/data-pipeline/`
  CREATE TABLE events USING DELTA LOCATION '/data-pipeline/'
  ```

- 创建Parquet表并转换为Delta表：

  SQL

  ```
  %sql
  CREATE TABLE events USING PARQUET OPTIONS (path '/data-pipeline/')
  CONVERT TO DELTA events
  ```

# 3.13. 最佳实践

> 说明
>
> 详情请参考Databricks官网文章：最佳实践

本文介绍了使用Delta Lake时的最佳做法。

## 提供数据位置提示

如果您通常希望在查询谓词中使用一个列，并且该列具有较高的基数（即，大量不同的值），则使用Z-ORDER-BY。Delta-Lake根据列值自动布局文件中的数据，并在查询时使用布局信息跳过不相关的数据。

有关详细信息，请参见Z-Ordering（多维集群）。

## 选择正确的分区列

您可以按列对Delta表进行分区。最常用的分区列是date。遵循以下两个经验法则来确定要按哪个分区进行分区：

- 如果列的基数很高，请不要使用该列进行分区。例如，如果按列进行分区，userId并且可以有1M个不同的用户ID，则这是一个不好的分区策略。
- 每个分区中的数据量：如果您希望该分区中的数据至少为1 GB，则可以按列进行分区。

## 压缩文件

如果您不断将数据写入Delta表，随着时间的流逝，它将累积大量文件，尤其是如果您少量添加数据时。这可能会对表读取的效率产生不利影响，也可能影响文件系统的性能。理想情况下，应定期将大量小文件重写为少量大文件。这称为压缩。

您可以使用OPTIMIZE命令来压缩表。

## 替换表的内容或架构

有时您可能要替换Delta表。例如：

- 您发现表中的数据不正确，并且想要替换内容。
- 您想要重写整个表以进行不兼容的架构更改（删除列或更改列类型）。

虽然您可以删除Delta表的整个目录并在同一路径上创建新表，但不建议这样做，因为：

- 删除目录效率不高。包含非常大文件的目录可能要花费数小时甚至数天才能删除。
- 您会丢失已删除文件中的所有内容；如果删除错误的表，将很难恢复。
- 目录删除不是原子的。在删除表时，读取表的并发查询可能会失败或看到部分表。

如果不需要更改表架构，则可以从Delta表中删除数据并插入新数据，或者更新表以修复不正确的值。

如果要更改表架构，则可以atomic替换整个表。例如：

```
dataframe.write \
 .format("delta") \
 .mode("overwrite") \
 .option("overwriteSchema", "true") \
 .partitionBy(<your-partition-columns>) \
 .saveAsTable("<your-table>") # Managed table
dataframe.write \
 .format("delta") \
 .mode("overwrite") \
 .option("overwriteSchema", "true") \
 .option("path", "<your-table-path>") \
 .partitionBy(<your-partition-columns>) \
 .saveAsTable("<your-table>") # External table
```

这种方法有很多好处：

- 覆盖表的速度要快得多，因为它不需要递归列出目录或删除任何文件。
- 该表的旧版本仍然存在。如果删除错误的表，则可以使用时间段轻松检索旧数据。
- 这是一个原子操作。删除表时，并发查询仍然可以读取表。
- 由于Delta Lake ACID事务保证，如果覆盖表失败，该表将处于其先前状态。

另外，如果要在覆盖表后删除旧文件以节省存储成本，则可以使用VACUUM删除它们。它针对文件删除进行了优化，通常比删除整个目录要快。

# 3.14. 常见问题（FAQ）

> **说明**
>
> 详情请参考Databricks官网文章：常见问题

## 什么是 Delta Lake?

Delta Lake是一个开源存储层，可为数据湖带来可靠性。Delta Lake提供ACID事务，可伸缩的元数据处理，并统一流处理和批数据处理。Delta Lake在您现有的数据湖之上运行，并且与Apache Spark API完全兼容。

Databricks上的Delta Lake允许您根据工作负载模式配置Delta Lake，并提供优化的布局和索引以进行快速的交互式查询。

## Delta Lake与Apache Spark有何关系?

Delta Lake位于Apache Spark之上。格式和计算层有助于简化大数据管道的构建并提高管道的整体效率。

## Delta Lake使用什么格式存储数据?

Delta Lake使用版本化的Parquet文件将您的数据存储在您的云存储中。除版本外，Delta Lake还存储事务日志，以跟踪对表或Blob存储目录所做的所有提交，以提供ACID事务。

## 如何使用Delta Lake读写数据？

您可以使用自己喜欢的Apache Spark API来使用Delta Lake读写数据。

## Delta Lake在哪里存储数据？

写入数据时，您可以指定云存储中的位置。Delta Lake以Parquet格式将数据存储在该位置。

## 我可以直接将数据流式传输到Delta表中吗？

是的，您可以使用结构化流直接将数据写入Delta表并从Delta表中读取。

## Delta Lake是否支持使用Spark Streaming DStream API进行写入或读取？

Delta不支持DStream API。我们建议进行表流读取和写入。

## 使用Delta Lake时，是否可以轻松地将代码移植到其他Spark平台？

是。使用Delta Lake时，您将使用开放的Apache Spark API，因此可以轻松地将代码移植到其他Spark平台。要移植代码，请将deltaformat替换为parquet格式。

## Delta表与Hive SerDe表相比如何？

Delta表的管理程度更高。特别是，Delta Lake代表您管理的多个Hive SerDe参数，您永远不要手动指定：

- ROWFORMAT
- SERDE
- OUTPUTFORMAT 和 INPUTFORMAT
- COMPRESSION
- STORED AS

## Delta Lake不支持哪些DDL和DML功能？

不支持的DDL功能：

- ANALYZE TABLE PARTITION
- ALTER TABLE [ADD|DROP] PARTITION
- ALTER TABLE RECOVER PARTITIONS
- ALTER TABLE SET SERDEPROPERTIES
- CREATE TABLE LIKE
- INSERT OVERWRITE DIRECTORY
- LOAD DATA

不支持的DML功能：

- INSERT INTO [OVERWRITE] 带有静态分区的表
- INSERT OVERWRITE TABLE 用于具有动态分区的表
- Bucketing
- 从表中读取时指定架构
- 在TRUNCATE表中使用PARTITION（part_spec）指定目标分区

## Delta Lake是否支持多表事务？

Delta Lake不支持多表事务和外键。Delta Lake支持表级别的事务。

## 如何更改列的类型？

更改列的类型或删除列需要重写表。有关示例，请参见更改列类型。

## Delta Lake支持多集群写入，这意味着什么？

这意味着Delta Lake会进行锁定，以确保同时从多个集群写入表的查询不会损坏该表。但是，这并不意味着如果存在写冲突（例如，更新和删除同一事件），则它们都会成功。相反，写入操作将以原子的方式失败，并且该错误将告诉您重试该操作。

## 多集群写入的局限性是什么？

在此模式下运行时，不支持以下功能：

- SparkR
- 使用<DBR 7.2>及更低版本执行spark-submit作业。使用<DBR 7.3>及更高版本运行spark-submit作业支持多集群写入。
- 客户提供的加密密钥的服务器端加密

您可以通过将设置spark.databricks.delta.multiClusterWrites.enabled为来禁用多集群写入false。如果禁用它们，对单个表的写入必须来自单个集群。

## 我可以在Databricks Runtime之外访问Delta表吗？

有两种情况需要考虑：外部写入和外部读取。

- 外部写入：Delta Lake以事务日志的形式维护其他元数据，以启用ACID事务和读取器的快照隔离。为了确保正确更新事务日志并执行正确的验证，写操作必须通过Databricks Runtime
- 外部读取：增量表存储以开放格式（Parquet）编码的数据，允许其他了解此格式的工具读取数据。有关如何读取Delta表的信息。

# 4.Delta Engine

## 4.1. Delta Engine 概述

Delta Engine 是与 Apache Spark 兼容的高性能查询引擎，提供了一种高效的方式来处理数据湖中的数据，包括存储在开源 Delta Lake 中的数据。 Delta Engine 优化可加快数据湖操作速度，并支持各种工作负载，从大规模 ETL 处理到临时交互式查询均可。 其中许多优化都自动进行；只需要通过将Databricks 用于数据湖即可获得这些 Delta Engine 功能的优势。

- 通过文件管理优化性能
- 自动优化
- 通过缓存优化性能
- 动态文件修剪
- 隔离级别
- Bloom 筛选器索引
- 优化联接性能
- 优化的数据转换

## 4.2. 通过文件管理优化性能

为了提高查询速度，阿里云Databricks上的Delta Lake支持优化存储在云存储中的数据布局的功能。Databricks上的Delta Lake支持两种布局算法：bin-packing和Z-Ordering。

> **说明**
>
> 详细内容可参考Databricks官网文章：通过文件管理优化性能

- 本文介绍如何运行优化命令，两种布局算法工作原理以及如何清除陈旧的表快照。
- 该FAQ解释了为什么优化是不是自动的，包括如何运行优化命令的建议。
- 对于演示优化优势的笔记本，请参阅优化示例。
  - Databricks Runtime 7.x：优化（Databricks上的Delta Lake）
  - Databricks运行时5.5 LTS和6.x：优化（Databricks上的Delta Lake）

### 压缩 (bin-packing)

Databricks 上的 Delta Lake 可以将小文件合并为较大的文件，从而提高表中读取查询的速度。 通过运行 `OPTIMIZE` 命令触发压缩：

SQL

```
OPTIMIZE delta.`/data/events`
```

or

```
OPTIMIZE events
```

如果您有大量数据，并且只想优化其中的一个子集，则可以使用WHERE指定可选的分区谓词：

```
OPTIMIZE events WHERE date >= '2017-01-01'
```

> 说明
>
> Bin-packing是幂等的，这意味着如果在同一个数据集上运行两次，第二次运行没有效果。
>
> Bin-packing的目标是根据磁盘大小生成均衡的数据文件，但不一定是每个文件的元组数。然而，这两个度量值通常是相互关联的。

Delta表的读取器使用快照隔离，这意味着当OPTIMIZE从事务日志中删除不必要的文件时，它们不会被中断。OPTIMIZE 不会对表进行任何数据相关更改，因此，在 OPTIMIZE 之前和之后读取都具有相同的结果。 对作为流式处理源的表执行 OPTIMIZE 不会影响将此表视为源的任何当前或未来的流。OPTIMIZE操作返回删除的文件和添加的文件的文件统计信息（最小值、最大值、总计等）。Optimize stats还包含Z-Ordering统计信息、批处理数量和优化的分区。

> 注意
>
> 在Databricks Runtime 6.0及更高版本中可用。

您还可以使用“自动优化”自动压缩小文件。

## Data skipping

将数据写入Delta表时，将自动收集data skipping信息。Databricks上的Delta Lake在查询时利用此信息（最小值和最大值）来提供更快的查询。您无需配置data skipping。该功能会在适用时激活。但是，其有效性取决于数据的布局。为了获得最佳效果，请应用Z-Ordering。

有关Delta Lake在Databricks data skipping和Z-Ordering方面的优点的示例，请参阅“优化示例”中的notebook。默认情况下，Databricks上的Delta Lake会收集有关表架构中定义的前32列的统计信息。您可以使用table属性dataSkippingNumIndexedCols更改此值。添加更多列来收集统计信息将在编写文件时增加额外的开销。

收集长字符串的统计信息成本高昂。若要避免收集有关长字符串的统计信息，可以将表属性dataskippingnumindexedcols配置为避免包含长字符串的列，也可以使用ALTER table CHANGE coln将包含长字符串的列移动到大于dataskipingnumindexedcols的列。为了收集统计信息，嵌套列中的每个字段都被视为一个单独的列。

## Z-Ordering (多维聚类)

Z-Ordering 是并置同一组文件中相关信息的方法。Databricks 上的Delta Lake data skipping算法会自动使用了这种并置，从而大大减少了需要读取的数据量。从而显著减少需要读取的数据量。对于Z-Order数据，可以在ZORDER-BY子句中指定要排序的列：

SQL

```
OPTIMIZE events
WHERE date >= current_timestamp() - INTERVAL 1 day
ZORDER BY (eventType)
```

如果您希望在查询predicates中经常使用一个列，并且该列具有较高的基数（即，有大量不同的值），那么请使用zorderby。

可以将ZORDER BY的多个列指定为逗号分隔的列表。然而，每增加一列，局部性的有效性就会下降。对没有收集到统计信息的列进行Z-Ordering将是无效的，并且由于数据跳过需要列本地统计信息（如min、max和count），因此会浪费资源。通过对架构中的列重新排序或增加要收集统计信息的列数，可以在某些列上配置统计信息收集。有关详细信息，请参阅数据跳过部分。

> 注意
>
> - Z-Ordering不是幂等的，而是一种增量操作。多次运行间不能保证Z-Ordering所需的时间减少。但是，如果没有新的数据添加到一个只有Z-Ordering的分区中，那么该分区的另一个Z-Ordering将不会有任何效果。
> - Z-Ordering旨在根据元组的数量生成均匀平衡的数据文件，但不一定是磁盘上的数据大小。这两个度量值通常是相互关联的，但也有可能出现这种情况，导致优化任务时间出现偏差。
>
>   例如，如果您按日期排序，并且您最近的记录都比过去的记录宽得多（例如，数组或字符串值更长），则OPTIMIZE作业的任务持续时间和生成的文件大小都会出现偏差。但是，这只是OPTIMIZE命令本身的问题；它不应对后续查询产生任何负面影响。

## Notebooks

有关优化的好处的示例，请参阅以下Notebook:

优化实例

- Delta Lake on Databricks优化Python笔记本
- Delta Lake在Databricks上优化Scala笔记本
- Delta Lake on Databricks优化SQL笔记本

## 提高交互式查询性能

Delta Engine提供了一些其他机制来提高查询性能。

**管理数据实效性**

在每个查询开始时，Delta表会自动更新到表的最新版本。当命令状态报告： Updating the Delta table's state时，可以在笔记本中观察到这个过程。但是，在表上运行历史分析时，您可能不需要最新的数据，特别是在频繁引入流式处理数据的表中。 在这些情况下，可以在 Delta 表的过时快照上运行查询。 这会降低从查询获取结果的延迟时间。

可以通过将 Spark 会话配置 spark.databricks.delta.stalenessLimit 设置为时间字符串值（例如 1h、15m、1d 分别为 1 小时、15 分钟和 1 天）来配置表数据的过时程度。此配置是特定session，因此不会影响其他用户从其他笔记本、作业或BI工具访问此表。另外，此设置不会更新表。它只会阻止查询等待表更新。该更新仍在后台进行，并将在整个集群之间公平地共享资源。如果超过过期限制，则查询将在表状态更新上阻止。

## 用于低延迟查询的增强检查点

Delta Lake 写入检查点作为 Delta 表的聚合状态，每 10 次提交写入一次。这些检查点用作计算表的最新状态的起点。如果没有检查点 ，Delta Lake将不得不读取大量的JSON文件（“Delta”文件），表示提交到事务日志以计算表的状态。此外，此外，列级统计信息 Delta Lake 用于执行存储在检查点中的数据跳过操作。

> 警告
>
> Delta Lake 检查点与结构化流checkpoints不同。

在Databricks Runtime 7.2及更低版本中，列级别的统计信息作为JSON列存储在Delta Lake 检查点中。在Databricks Runtime 7.3LTS及更高版本中，列级别的统计信息存储作为结构。结构格式使Delta Lake读取速度更快，因为：

- Delta Lake不会执行昂贵的JSON解析来获取列级统计信息。
- Parquet 列修剪功能可以显著减少读取列的统计信息所需的 I/O

结构格式可启用一系列优化，这些优化可将Delta Lake读取操作的开销从几秒降低到数十毫秒，从而显着减少短查询的延迟。

**管理检查点中的列级统计信息** 您可以使用表属性delta.checkpoint.writestatsasjson以及delta.checkpoint.writeStatsasTrust.管理如何在检查点中写入统计信息。如果两个表属性都为false，则Delta-Lake无法执行跳过数据。

在Databricks Runtime 7.3 LTS及更高版本中：

- 批量写入JSON和结构格式编写写入统计信息。delta.checkpoint.writeStatsAsJson默认值为ture。
- 流式处理只以JSON格式写入统计信息（以最大程度地减少检查点对写入延迟的影响）。若要同时编写结构格式，请参见为结构化流式查询启用增强的检查点。
- 在这两种情况下，都默认未定义 delta.checkpoint.writeStatsAsStruct。
- 读取器在可用时使用结构列，否则退回到使用JSON列。

在Databricks运行时7.2及以下版本中，读者只使用JSON列。因此，如果delta.checkpoint.writestatsasjson为false，此类读取器无法执行跳过数据。

> 警告
>
> 增强的检查点不会破坏与开源Delta-Lake 读取器的兼容性。但是，设置delta.checkpoint.writestatsasjson为false可能会影响到Delta Lake的专有读取器。请与您的供应商联系以了解有关性能影响的更多信息。

**检查点中统计信息的权衡**

由于在检查点中编写统计信息会产生成本（即使对于大型表，通常也要不到一分钟），因此需要权衡在编写检查点所花费的时间与与Databricks Runtime 7.2及更低版本的兼容性。如果您能够将所有工作负载升级到Databricks Runtime 7.3 LTS或更高版本，则可以通过禁用旧版JSON统计信息来降低编写checkpoints的成本。下表总结了这一折衷方案。

如果跳过数据不适用于你的应用程序，则可以将两个属性都设置为false，并且不收集或写入任何统计信息。我们不建议这种配置。

| | writeStatsAsStruct | |
|---|---|---|
| | false | true |

| | | | |
|---|---|---|---|
| writeStatsAsJson | false | • 没有数据skipping | • 在Databricks Runtime 7.3及更高版本上的查询更快<br>• checkpoints稍慢<br>• 在Databricks Runtime 7.2及更低版本的阅读器中，没有数据skipping |
| | true | • Databricks运行时7.2及以下<br>• 较慢的查询 | • 在Databricks Runtime 7.3及更高版本上的查询更快<br>• 保持与Databricks Runtime 7.2及更低版本上的阅读器的兼容性<br>• checkpoints的延迟最高（秒级） |

## 为结构化流查询启用增强的checkpoints

如果您的结构化流工作负载没有低延迟要求（要求延迟在一分钟以内），则可以通过运行以下SQL命令来启用增强检查点：

SQL

```
ALTER TABLE [<table-name>|delta.`<path-to-table>`] SET TBLPROPERTIES
('delta.checkpoint.writeStatsAsStruct' = 'true')
```

如果您不使用Databricks Runtime 7.2或更低版本来查询数据，则还可以通过设置以下表属性来改善检查点写入延迟：

SQL

```
ALTER TABLE [<table-name>|delta.`<path-to-table>`] SET TBLPROPERTIES
(
 'delta.checkpoint.writeStatsAsStruct' = 'true',
 'delta.checkpoint.writeStatsAsJson' = 'false'
)
```

### 禁止从没有统计结构的检查点的集群中写入

Databricks Runtime 7.2及更低版本中的编写器会写入无统计结构检查点，从而妨碍了对Databricks Runtime 7.3 LTS阅读器的优化。要阻止运行Databricks Runtime 7.2及更低版本的集群写入Delta表，可以使用以下upgradeTableProtocol方法升级Delta表：

Python

```
from delta.tables import DeltaTable
delta = DeltaTable.forPath(spark, "path_to_table") # or DeltaTable.forName
delta.upgradeTableProtocol(1, 3)
```

Scala

```
import io.delta.tables.DeltaTable
val delta = DeltaTable.forPath(spark, "path_to_table") // or DeltaTable.forName
delta.upgradeTableProtocol(1, 3)
```

> **警告**
>
> 应用该upgradeTableProtocol方法可防止运行Databricks Runtime 7.2及更低版本的集群写入表，并且此更改是不可逆的。我们建议仅在您采用新格式后才升级表。您可以通过使用Databricks Runtime 7.3为表创建浅表克隆来尝试这些优化。

升级表写入程序版本后，写入程序必须遵守'delta.checkpoint.writeStatsasTrust'和'delta.checkpoint.writestatsasjson'.的设置

下表总结了如何在不同版本的Databricks Runtime、表协议版本和编写器类型中利用增强的检查点。

| | Without Protocol Upgrade | | | With Protocol Upgrade | | |
|---|---|---|---|---|---|---|
| | Databricks Runtime 7.2及以下编写器 | Databricks Runtime 7.3及更高版本的批处理编写器 | Databricks Runtime 7.3及更高版本的流编写器 | Databricks Runtime 7.2及以下编写器 | Databricks Runtime 7.3及更高版本的批处理编写器 | Databricks Runtime 7.3及更高版本的流编写器 |
| Databricks Runtime 7.2及以下读取器性能 | 没有得到改进 | 没有得到改进 | 没有得到改进 | 不能使用编写器 | 没有得到改进 | 没有得到改进 |
| Databricks Runtime 7.3及更高版本的读取器性能 | 没有得到改进 | 默认情况下改进 | 通过表格属性选择Opt-in（1） | 不能使用编写器 | 默认情况下改进 | 通过表格属性选择Opt-in（1） |

（1）设置表属性'delta.checkpoint.writeStatsAsStruct' = 'true'

**禁用使用旧检查点格式的从集群中写入** Databricks Runtime 7.2及更低版本的编写器可以编写旧格式的检查点，这将妨碍对Databricks Runtime 7.3编写器的优化。要阻止运行Databricks Runtime 7.2及更低版本的集群写入Delta表，可以使用upgradeTableProtocol方法升级Delta表：

Python

```
from delta.tables import DeltaTable
delta = DeltaTable.forPath(spark, "path_to_table") # or DeltaTable.forName
delta.upgradeTableProtocol(1, 3)
```

Scala

```
import io.delta.tables.DeltaTable
val delta = DeltaTable.forPath(spark, "path_to_table") // or DeltaTable.forName
delta.upgradeTableProtocol(1, 3)
```

> **注意**
>
> 应用该upgradeTableProtocol方法可防止运行Databricks Runtime 7.2及更低版本的集群写入表。这种更改是不可逆的。因此，我们建议仅在提交新格式后才升级表。您可以通过使用Databricks Runtime 7.3创建表的浅表克隆来尝试这些优化：
>
> - Databricks Runtime 7.x：CLONE（Databricks上的Delta Lake）
> - Databricks运行时5.5 LTS和6.x：CLONE（Databricks上的Delta Lake）

## 常见问题（FAQ）

**为什么OPTIMIZE不是自动的？**

OPTIMIZE操作启动了多个Spark作业，以便通过压缩来优化文件大小（并可以选择执行 Z-Ordering）。由于OPTIMIZE执行的内容大部分是压缩小文件，因此您必须先累积许多小文件，然后此操作才能生效。因此，该OPTIMIZE操作不会自动运行。

此外，运行 OPTIMIZE（特别是 ZORDER）是时间和资源成本高昂的操作。如果Databricks自动运行OPTIMIZE或等待分批写出数据，则将无法运行（以Delta表是源）低延迟的Delta-Lake流。许多客户的Delta表从未进行过优化，因为他们只从这些表流式传输数据，从而避免了优化所带来的查询好处。

最后，Delta Lake会自动收集有关写入表的文件（无论是否通过 `OPTIMIZE` 操作）的统计信息。这意味着从Delta表的读取将利用此信息，无论该表或分区是否运行了 `OPTIMIZE` 操作

**我应该多久跑步一次OPTIMIZE？**

当您选择运行OPTIMUZE的频率时，性能和成本之间就需要权衡取舍。 如果希望获得更好的最终用户查询性能，则应更频繁地运行 `OPTIMIZE` （根据资源使用量，可能需要较高的成本）。如果要优化成本，应该减少运行频率。

**运行 `OPTIMIZE` （二进制打包和 Z 排序）的最佳实例类型是什么？**

这两个操作都是执行大量 Parquet 解码和编码的 CPU 密集型操作。

对于这些工作负载，建议采用 F 或 Fsv2 系列。

# 4.3. 自动优化

“自动优化”是一组可选功能，可在每次写入Delta表时自动压缩小文件。在写入过程中支付少量费用可为主动查询的表提供显著的好处。自动优化在以下情况下特别有用：

> **说明**
>
> 详细内容可参考Databricks官网文章：自动优化

- 流式传输用例，可以接受几分钟的延迟

- MERGE INTO 是写入Delta Lake的首选方法
- CREATE TABLE AS SELECT或INSERT INTO是常用的操作

## 自动优化的工作原理

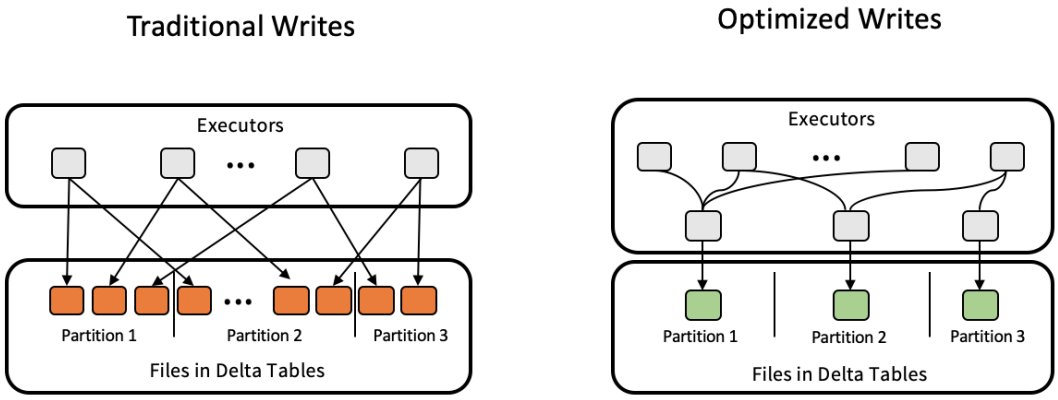
自动优化包含两个补充功能：优化写入和自动压缩。

**优化写入**

Databricks基于实际数据动态优化Apache Spark分区大小，并尝试为每个表分区写出128 MB文件。这是一个近似大小，并且可能会因数据集特征而异。

**自动压缩**

每次写入之后，Databricks会检查文件是否可以进一步压缩，并运行一个快速 OPTIMIZE 作业（文件大小为128mb，而不是1GB），以便进一步压缩包含最多小文件的分区的文件。




## 用法

自动优化旨在针对特定的Delta表进行配置。通过设置table属性delta.autoOptimize.optimizeWrite=true，可以为表启用优化写入。同样地，你设置delta.autoOptimize.autoCompact=true以启用自动压缩。

对于现有表，运行：

SQL

```
ALTER TABLE [table_name | delta.`<table-path>`] SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true, delta.autoOptimize.autoCompact = true)
```

为确保所有新的Delta表都启用了这些功能，请设置SQL配置：

SQL

```
set spark.databricks.delta.properties.defaults.autoOptimize.optimizeWrite = true;
set spark.databricks.delta.properties.defaults.autoOptimize.autoCompact = true;
```

此外，您可以使用以下配置为Spark会话启用和禁用这两个功能：

- spark.databricks.delta.optimizeWrite.enabled

- spark.databricks.delta.autoCompact.enabled

session配置优先于表属性，使您可以更好地控制何时选择启用或停用这些功能。

## 何时选择加入和退出

本节提供有关何时加入和退出自动优化功能的指南。

**优化写入** 优化写入旨在最大程度地提高写入存储服务的数据吞吐量。这可以通过减少写入的文件数来实现，而不会牺牲太多的并发度。

优化写入要求根据目标表的分区结构来shuffling数据。这种改组自然会招致额外的费用。但是，写入过程中获得的吞吐率收益可以偿还shuffling的成本。如果不是这样，则查询数据时，吞吐量会得到提高。

**何时加入**

- 流式传输用例，可以接受几分钟的延迟
- 当使用诸如MERGE、UPDATE、DELETE、INSERT INTO、createtableasselect等SQL命令时

**何时退出**

- 当写入的数据达到TB级时，并且存储优化实例不可用。
- 当使用spot实例时，spot价格不稳定，导致大部分节点丢失。

## 自动压缩

自动压缩在成功写入表之后发生，并且在执行写入的集群上同步运行。这意味着，如果你的代码模式向增量 Lake 进行写入，然后立即调用 OPTIMIZE ，则 OPTIMIZE 如果你启用自动压缩，则可以删除该调用。

OPTIMIZE与自动压缩不同。由于它在写入后同步运行，所以我们对自动压缩进行了优化，使其具有以下属性：

- Databricks不支持带有自动压缩的Z-Ordering，因为Z-Ordering比仅压缩要贵得多。
- 自动压缩（1 GB）生成较小的OPTIMIZE文件（128 MB）。
- 自动压缩贪婪地选择一组有限的分区，这些分区最能充分利用压缩。所选分区的数量将根据启动它的集群的大小而变化。如果您的集群有更多的CPU，就可以优化更多的分区。

**何时加入**

- 流式使用案例，其中延迟分钟是可接受的
- 当您的表上没有常规的OPTIMIZE调用时

**何时退出**

当其他写入程序可能同时执行删除、合并、更新或优化等操作时，因为自动压缩可能会导致这些作业的事务冲突。如果由于事务冲突自动压缩失败，则Databricks不会失败或重试压缩。

## 工作流程示例：流式引入与并发删除或更新

此工作流假定您有一个集群运行24/7流式处理作业引入数据，另一个集群每小时、每天或临时运行一次以删除或更新一批记录。对于这个用例，Databricks建议您：

- 在表级别启用Optimized写入，使用

  SQL

```
ALTER TABLE <table_name|delta.`table_path`> SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
```

这样可以确保流写入的文件数量以及删除和更新作业的大小都是最佳的。

- 在执行删除或更新的作业上使用以下设置，在session级别启用“自动压缩”。

  Scala

  ```
  spark.sql("set spark.databricks.delta.autoCompact.enabled = true")这样可以在整个表中压缩文件。由于它是在删除或更新之后发生的，因此可以减轻发生事务冲突的风险。
  ```

  这样可以在整个表中压缩文件。由于它是在删除或更新之后发生的，因此可以减轻发生事务冲突的风险。

### 常见问题（FAQ）

**是否自动优化Z-Order 文件？**

自动优化仅对小文件执行压缩。它不是Z-Order文件。

**是否自动优化损坏的Z-Order文件吗？**

“自动优化”将忽略Z-Order的文件。它仅压缩新文件。

**如果在流式传输的表上启用了“自动优化”，并且并发事务与优化冲突，我的工作会失败吗？**

不会。导致自动优化失败的事务冲突将被忽略，并且流式传输将继续正常运行。

**OPTIMIZE如果在表上启用了自动优化功能，是否需要计划作业？**

对于大小大于10TB的表，我们建议您继续按计划OPTIMIZE运行，以进一步整合文件，并减少Delta表的元数据。由于自动优化不支持 Z-Ordering，所以您仍然应该调度OPTIMIZE...ZORDER BY定期运行作业 。

# 4.4. 通过缓存优化性能

Delta缓存通过使用快速中间数据格式在节点的本地存储中创建远程文件的副本来加速数据读取。每当需要从远程位置获取文件时，数据都会自动缓存。然后在本地的连续读取上述数据，从而显著提高读取速度。

> **说明**
>
> 详细内容可参考Databricks官网文章：通过缓存优化性能

Delta缓存支持读取HDFS、OSS存储中的Parquet文件。它不支持其他存储格式，如CSV、JSON和ORC。

> **注意**
>
> Delta缓存适用于所有Parquet文件，并且不仅限于Delta Lake格式的文件。

## Delta和Apache Spark缓存

Databricks提供两种类型的缓存：Delta缓存和Apache Spark缓存。这是每种类型的特征：

- **存储的数据类型**：Delta缓存包含远程数据的本地副本。它可以提高各种查询的性能，但不能用于存储任意子查询的结果。Spark缓存可以存储任何子查询数据的结果以及以Parquet以外的格式（例如CSV，JSON和ORC）存储的数据。
- **性能**：Delta缓存中存储的数据比Spark缓存中的数据读取和操作速度更快。这是因为Delta缓存使用高效的解压算法，并以最佳格式输出数据，以便使用整个阶段的代码生成进行进一步处理。

- **自动与手动控制**：启用Delta缓存时，必须从远程源获取的数据将自动添加到缓存中。这个过程是完全透明的，不需要任何操作。但是，要预先将数据预加载到缓存中，可以使用cache命令（请参见缓存数据的子集）。使用Spark缓存时，必须手动指定要缓存的表和查询。
- **磁盘与基于内存**：Delta缓存完全存储在本地磁盘上，这样就不会从Spark中的其他操作中占用内存。由于现代固态硬盘的高读取速度，Delta缓存可以完全驻留在磁盘上，而不会对其性能产生负面影响。相反，Spark缓存使用内存。

> **注意**
>
> 您可以同时使用Delta缓存和Apache Spark缓存。

## 概要

下表总结了Delta和Apache Spark缓存之间的主要区别，以便您选择最合适工作流的工具：

| 功能 | Delta 缓存 | Apache Spark 缓存 |
| --- | --- | --- |
| 储存格式 | 工作节点上的本地文件。 | In-memory blocks,但它取决于存储级别。 |
| 适用对象 | WASB和其他文件系统上存储任何Parquet表。 | 任何RDD或DataFrame。 |
| 触发 | 自动执行，第一次读取时（如果启用了缓存）。 | 手动执行，需要更改代码。 |
| 已评估 | Lazily. | Lazily. |
| 强制缓存 | CACHE 和 SELECT | .cache + 任何实现缓存的操作和.persist. |
| 可用性 | 可以通过配置标志启用或禁用，在某些节点类型上禁用。 | 始终可用 |
| 驱逐 | 在任何文件更改时自动执行，重新启动集群时手动执行。 | 以LRU方式自动执行，使用unpersist手动执行。 |

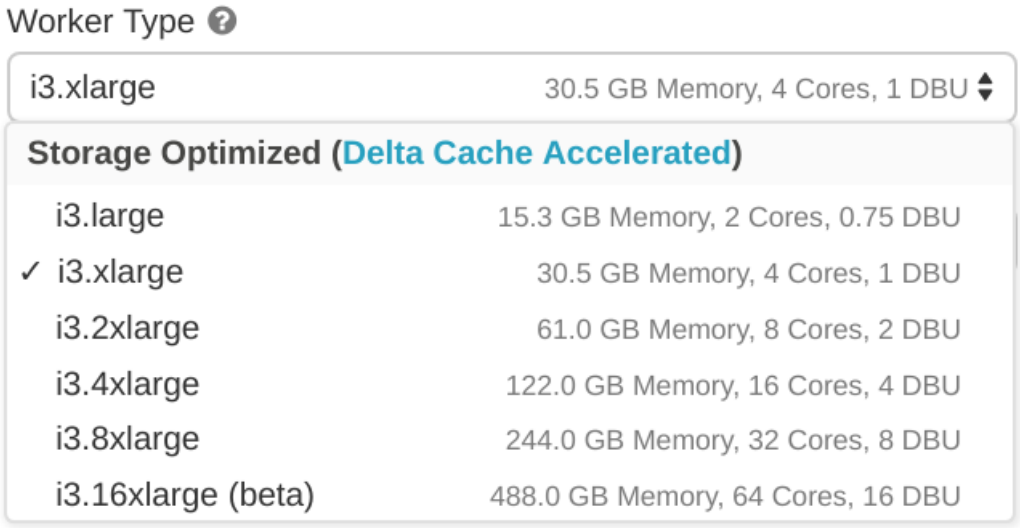
## Delta 缓存一致性

Delta缓存会自动检测何时创建或删除数据文件，并相应地更新其内容。您可以写入，修改和删除表数据，而无需显示的使缓存数据无效。

Delta缓存会自动检测缓存后已被修改或覆盖的文件。所有陈旧的条目都会自动失效并从缓存中逐出。

## 使用Delta缓存

要使用Delta缓存，请在配置集群时选择Delta**缓存加速**工作器类型。




默认启用Delta缓存，并配置为最多使用工作节点随附的本地SSD上一半可用空间。

有关配置选项，请参阅“配置Delta缓存”。

## 缓存数据的子集

要明确选择需要缓存的数据子集，请使用以下语法：

SQL

```
CACHE SELECT column_name[, column_name, ...] FROM [db_name.]table_name [ WHERE boolean_expression ]
```

您无需使用此命令即可正常使用Delta缓存（数据在首次访问时将自动缓存）。但是，当您需要一致的查询性能时，它可能会有所帮助。

有关示例和更多详细信息，请参见

- Databricks Runtime 7.x：CACHE（Databricks上的Delta Lake）
- Databricks运行时5.5 LTS和6.x：CACHE（Databricks上的Delta Lake）

## 监控Delta缓存

您可以在Spark UI的“存储”选项卡中的每个执行器上检查Delta缓存的当前状态。

Storage

Parquet IO Cache

| Host | Disk Usage | Max Disk Usage Limit | Percent Disk Usage | Metadata Cache Size | Max Metadata Cache Size Limit | Percent Metadata Usage |
|---|---|---|---|---|---|---|
| 10.0.131.228 | 1021.1 MB | 1024.0 MB | 99 % | 7.9 MB | 128.0 MB | 6 % |
| 10.0.133.97 | 1022.5 MB | 1024.0 MB | 99 % | 7.9 MB | 128.0 MB | 6 % |
| 10.0.250.163 | 1021.6 MB | 1024.0 MB | 99 % | 7.9 MB | 128.0 MB | 6 % |
| 10.0.253.229 | 1022.4 MB | 1024.0 MB | 99 % | 7.9 MB | 128.0 MB | 6 % |
| Total | 4.0 GB | 4.0 GB | 99 % | 31.6 MB | 512 MB | 6 % |

| Data Read from External Filesystem | Data Read from IO Cache | Data Written to IO Cache | Estimated Size of Repeatedly Read Data | Cache Metadata Manager Peak Disk Usage |
|---|---|---|---|---|
| 12.3 GB | 19.8 GB | 10.1 GB | 20.1 GB (62 %) - 20.1 GB (62 %) | 236.1 KB |

当节点的磁盘使用率达到100％时，缓存管理器将丢弃最近最少使用的缓存条目，以便为新数据腾出空间。

## 配置Delta缓存

> **注意**
>
> Databricks建议您为集群选择缓存加速的工作程序实例类型。此类实例已针对Delta缓存自动执行了最佳配置。

## 配置磁盘使用率

要配置Delta缓存如何使用工作节点的本地存储，请在集群创建期间指定以下Spark配置设置：

- spark.databricks.io.cache.maxDiskUsage -每个节点为缓存的数据保留的磁盘空间（以字节为单位）
- spark.databricks.io.cache.maxMetaDataCache -每个节点为缓存的元数据保留的磁盘空间（以字节为单位）
- spark.databricks.io.cache.compression.enabled -缓存的数据是否应以压缩格式存储

INI

```
spark.databricks.io.cache.maxDiskUsage 50g
spark.databricks.io.cache.maxMetaDataCache 1g
spark.databricks.io.cache.compression.enabled false
```

**启用Delta缓存**

要启用和禁用Delta缓存，请运行：

Scala

```
spark.conf.set("spark.databricks.io.cache.enabled", "[true | false]")
```

# 4.5. 动态文件修剪

动态文件修剪（DFP）可以显着提高Delta表上许多查询的性能。对于非分区表或非分区列上的联接，DFP尤其有效。DFP对性能的影响通常与clustering数据相关，因此请考虑使用Z-Ordering来最大化DFP的收益。

> 说明
>
> 详细内容可参考Databricks官网文章：动态文件修剪

有关DFP广告管理系统的背景和用例，请参阅带有动态文件修剪的Delta Lake上的快速SQL查询。

> 注意
>
> 在Databricks Runtime 6.1及更高版本中可用。

DFP由以下Apache Spark配置选项控制：

- spark.databricks.optimizer.dynamicPartitionPruning（默认值为true）：指示优化器下推DFP过滤器的主要标志。设置为false时，DFP将无效。
- spark.databricks.optimizer.deltaTableSizeThreshold（默认值为1000000000字节（10 GB））：表示连接探测端触发DFP所需的Delta表的最小大小（以字节为单位）。如果探测 端不是很大，可能不值得按下过滤器，我们可以简单地扫描整个表。通过运行DESCRIBE DETAIL table_name命令，然后查看sizeInBytes列，可以找到Delta表的大小。
- spark.databricks.optimizer.deltaTableFilesThreshold（默认值为1000）：表示连接探测端触发DFP所需的Delta表的文件数。当探测端表包含的文件少于阈值时，DPP不会被触发。如果一个表只有几个文件，那么可能不值得启用DFP。通过运行DESCRIBE DETAIL table_name命令，然后查看numFiles列，可以找到Delta表的大小。

# 4.6. 隔离等级

表的隔离级别定义了必须将某事务与并发事务所做的修改隔离的程度。Databricks上的Delta Lake支持两种隔离级别：Serializable和WriteSerializable。

> 说明
>
> 详细内容可参考Databricks官网文章：隔离等级

- **Serializable**：最强的隔离级别。它确保提交的写入操作和所有读取都是可序列化。 只要有一个串行序列一次执行一项操作，且生成与表中所示相同的结果，则可执行这些操作。对于写操作，串行序列与表历史记录中的序列完全相同。
- **WriteSerializable（默认）**：隔离级别比Serializable低。它仅确保写入操作（而非读取）是可序列化的。但是，这仍然比快照隔离更安全。WriteSerializable是默认的隔离级别，因为对大多数常见操作而言，它使数据一致性和可用性之间达到良好的平衡。

  在此模式下，Delta 表的内容可能与表历史记录中所示的操作序列不同。这是因为此模式允许某些并发写入对继续进行（例如，操作X和Y），即使历史记录显示Y是在X之后提交的，但结果是Y在X之前执行的（也就是说，它们之间可序列化）。要禁止这种重新排序，请将表隔离级别设置为Serializable，以使这些事务失败。

## 设置隔离级别

可以使用ALTER TABLE命令设置隔离级别。

SQL

```
ALTER TABLE <table-name> SET TBLPROPERTIES ('delta.isolationLevel' = <level-name>)
```

其中<level name>是Serializable或WriteSerializable。

例如，要将隔离级别从默认WriteSerializable更改为Serializable，请运行：

SQL

```
ALTER TABLE <table-name> SET TBLPROPERTIES ('delta.isolationLevel' = 'Serializable')
```

# 4.7. Bloom过滤器索引

Bloom filter索引是一种节省空间的数据结构，它允许在选定的列上数据skipping，特别是对于包含任意文本的字段。bloom filter通过声明数据肯定不在文件中，或者它可能在文件中，并使用定义的false positive probability（FPP）进行操作。

> 说明
>
> 详细内容可参考Databricks官网文章：Bloom过滤索引

Databricks支持文件级Bloom过滤器；每个数据文件都可关联一个 Bloom 筛选器索引文件。在读取文件之前，Databricks会检查索引文件，并且仅在索引指示该文件可能与数据筛选器匹配时才会读取该文件。如果索引不存在或未为查询的列定义Bloom过滤器，则Databricks会一直读取数据文件。

Bloom 筛选器的大小由两者决定：为其创建了 Bloom 筛选器的集中的数字元素，以及必需的 FPP。FPP越低，每个元素使用的位数越多，它的准确性就越高，但代价是磁盘空间占用更多、下载速度更慢。例如，FPP为10%时，每个元素需要5位。

Bloom过滤器索引是包含单个行的未压缩Parquet文件。索引存储在与数据文件相关的 _delta_index 子目录中，且使用与后缀为 index.v1.parquet 的数据文件相同的名称。例如，数据文件dbfs:/db1/data.0001.parquet.snappy的索引将命名为dbfs:/db1/_delta_index/data.0001.parquet.snappy.index.v1.parquet。

Bloom过滤器支持具有以下（输入）数据类型的列：byte、short、int、long、float、double、date、timestamp和string。不会向 Bloom 筛选器添加 NULL 值，因此任何与NULL相关的筛选器都需要读取数据文件。Databricks支持以下数据源筛选器：and、or、in、equals和equalsnullsafe。嵌套列不支持Bloom筛选器。

## 配置

默认情况下启用Bloom过滤器。要禁用Bloom过滤器，请将会话级别spark.databricks.io.skipping.bloomFilter.enabled配置设置为false。

## 创建Bloom过滤器索引

若要在表格中为新数据或重写数据的所有列或部分列创建 Bloom 筛选器索引，请使用createbloomfilter index DDL语句。例如，以下语句在列 sha 中创建 Bloom 筛选器索引，并在列中将 FPP 0.1 和 50,000,000 作为不同的项。

SQL

```
CREATE BLOOMFILTER INDEX
ON TABLE bloom_test
FOR COLUMNS(sha OPTIONS (fpp=0.1, numItems=50000000))
```

- Databricks Runtime 7.x: CREATE BLOOM FILTER INDEX (Delta Lake on Databricks)
- Databricks Runtime 5.5 LTS and 6.x: Create Bloom Filter Index (Delta Lake on Databricks)

### 删除Bloom过滤器索引

若要从表或表的一组列中删除所有 Bloom 过滤器，请使用DDL语句。例如：DROP BLOOMFILTER INDEX

SQL

```
DROP BLOOMFILTER INDEX ON TABLE bloom_test FOR COLUMNS(sha);
```

- Databricks Runtime 7.x:
  DROP BLOOM FILTER INDEX（Databricks上的Delta Lake）
- Databricks运行时5.5 LTS和6.x：Drop Bloom筛选器索引

### Notebook

以下Notebook演示了如何定义Bloom过滤器索引来加速“needle in a haystack”查询。

Bloom筛选器演示Notebook

Notebook链接地址：Bloom filter index demo

# 4.8. 优化链接性能

Delta Lake on Databricks可优化范围和skew连接。Range连接优化需要根据您的查询模式进行调整，Skew连接可以通过skew提示变得高效。请参阅以下文章以了解如何充分利用这些连接优化：

- Range Join optimization
- Skew Join optimization

> 说明
>
> 详细内容可参考Databricks官网文章：优化链接性能

# 4.9. 优化数据转换

Databricks使用嵌套类型优化高阶函数和 DataFrame 操作的性能。请参阅以下文章以了解如何开始使用这些优化的高阶函数和复杂数据类型：

- Higher-order functions
- Transform complex data types

> **说明**
>
> 详细内容可参考Databricks官网文章：优化数据转换