

Apache Spark Delta Lake 删除使用及实现原理代码解析

Delta Lake 的 Delete 功能是由 0.3.0 版本引入的，参见[这里](#)，对应的 Patch 参见[这里](#)。在介绍 Apache Spark Delta Lake 实现逻辑之前，我们先来看看如何使用 delete 这个功能。

Delta Lake 删除使用

Delta Lake 的官方文档为我们提供如何使用 Delete 的几个例子，参见[这里](#)，如下：

```
import io.delta.tables._

val iteblogDeltaTable = DeltaTable.forPath(spark, path)

// 删除 id 小于 4 的数据
iteblogDeltaTable.delete("id <= '4'")

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

iteblogDeltaTable.delete($"id" <= "4")

// 删除所有的数据
iteblogDeltaTable.delete()
```

执行上面的 Delete 命令，如果确实删除了相应的数据，Delta Lake 会生成一个事务日志，内容类似下面的：

```
{"commitInfo":{"timestamp":1566978478414,"operation":"DELETE","operationParameters":{"predicate":"[`id` <= CAST('4' AS BIGINT)]"},"readVersion":10,"isBlindAppend":false}}
{"remove":{"path":"dt=20190801/part-00000-ca73a0f4-fbeb-4ea8-9b9f-fa466a85724e.c000.snappy.parquet","deletionTimestamp":1566978478405,"dataChange":true}}
{"remove":{"path":"dt=20190803/part-00000-8e11f4cc-a7ac-47a1-8ce6-b9d87eaf6c51.c000.snappy.parquet","deletionTimestamp":1566978478405,"dataChange":true}}
{"add":{"path":"dt=20190801/part-00001-6ff11be3-22db-4ed2-bde3-a97d610fe11d.c000.snappy.parquet","partitionValues":{"dt":"20190801"},"size":429,"modificationTime":1566978478000,"dataChange":true}}
```

事务日志里面详细介绍了 Delete 执行的时间、删除的条件、需要删除的文件以及添加的文件等。

注意

- 执行 Delete 的时候，真实的数据其实并没有删除，只是在事务日志里面记录了，真正删除数据需要通过执行 vacuum 命令。
- 在编写本文的时候，开源版本的 Delta Lake 不支持使用 SQL 去删除数据，databricks 的企业版是支持的。在未来版本开源版本的 Delta Lake 也是会支持使用 SQL 去删除数据的，但具体版本目前还不确定。
- 0.3.0 版本的 Delta Lake 只支持使用 Scala & Java 去删除 Delta Lake 的数据，Python 相关的 API 可能会在 0.4.0 版本发布，参见：<https://github.com/delta-io/delta/issues/89>

Delta Lake 删除是如何实现的

前面小结我们简单体验了一下 Delete 的使用，本小结将深入代码详细介绍 Delta Lake 的 Delete 是如何实现的。delete 的 API 是通过在 io.delta.tables.DeltaTable 类添加相应方法实现的，其中涉及删除的方法主要包括下面三个：

```
def delete(condition: String): Unit = {
  delete(functions.expr(condition))
}

def delete(condition: Column): Unit = {
  executeDelete(Some(condition.expr))
}

def delete(): Unit = {
  executeDelete(None)
}
```

这个就是我们在上面例子看到的 delete 支持的三种用法。这三个函数最终都是调用 io.delta.tables.execution.DeltaTableOperations#executeDelete 函数的，executeDelete 的实现如下：

```
protected def executeDelete(condition: Option[Expression]): Unit = {
  val delete = Delete(self.toDF.queryExecution.analyzed, condition)

  // current DELETE does not support subquery,
  // and the reason why perform checking here is that
  // we want to have more meaningful exception messages,
  // instead of having some random msg generated by executePlan().
  subqueryNotSupportedCheck(condition, "DELETE")
```

```
val qe = sparkSession.sessionState.executePlan(delete)
val resolvedDelete = qe.analyzed.asInstanceOf[Delete]
val deleteCommand = DeleteCommand(resolvedDelete)
deleteCommand.run(sparkSession)
}
```

self.toDF.queryExecution.analyzed 这个就是我们输入 Delta Lake 表的 Analyzed Logical Plan , condition 就是我们执行删除操作的条件表达式 (也就是上面例子的 id <= '4') 。这个方法的核心就是初始化 DeleteCommand , 然后调用 DeleteCommand 的 run 方法执行删除操作。 DeleteCommand 类扩展自 Spark 的 RunnableCommand 特质，并实现其中的 run 方法，我们在 Spark 里面看到的 CREATE TABLE 、 ALTER TABLE 、 SHOE CREATE TABLE 等命令都是继承这个类的，所以 Delta Lake 的 delete 、 update 以及 Merge 也都是继承这个类。 DeleteCommand 的 run 方法实现如下：

```
final override def run(sparkSession: SparkSession): Seq[Row] = {
  recordDeltaOperation(tahoeFileIndex.deltaLog, "delta.dml.delete") {
    // 获取事务日志持有对象
    val deltaLog = tahoeFileIndex.deltaLog
    // 检查 Delta Lake 表是否支持删除操作
    deltaLog.assertRemovable()
    // 开启新事务，执行删除操作。
    deltaLog.withNewTransaction { txn =>
      performDelete(sparkSession, deltaLog, txn)
    }
    // Re-cache all cached plans(including this relation itself, if it's cached) that refer to
    // this data source relation.
    sparkSession.sharedState.cacheManager.recacheByPlan(sparkSession, target)
  }
  Seq.empty[Row]
}
```

Delta Lake 表允许用户设置成 appendOnly (通过 spark.databricks.delta.properties.defaults.appendOnly 参数设置) ，也就是只允许追加操作，所以如果我们执行删除之前需要做一些校验。校验通过之后开始执行删除操作，由于删除操作是需要保证原子性的，所以这个操作需要在事务里面进行， withNewTransaction 的实现如下：

```
def withNewTransaction[T](thunk: OptimisticTransaction => T): T = {
  try {
```

```
// 更新当前表事务日志的快照
update()
// 初始化乐观事务锁对象
val txn = new OptimisticTransaction(this)
// 开启事务
OptimisticTransaction.setActive(txn)
// 执行写数据操作
thunk(txn)
} finally {
// 关闭事务
OptimisticTransaction.clearActive()
}
}
```

在开启事务之前，需要更新当前表事务日志的快照，因为在执行删除操作表之前，这张表可能已经被修改了，执行 update

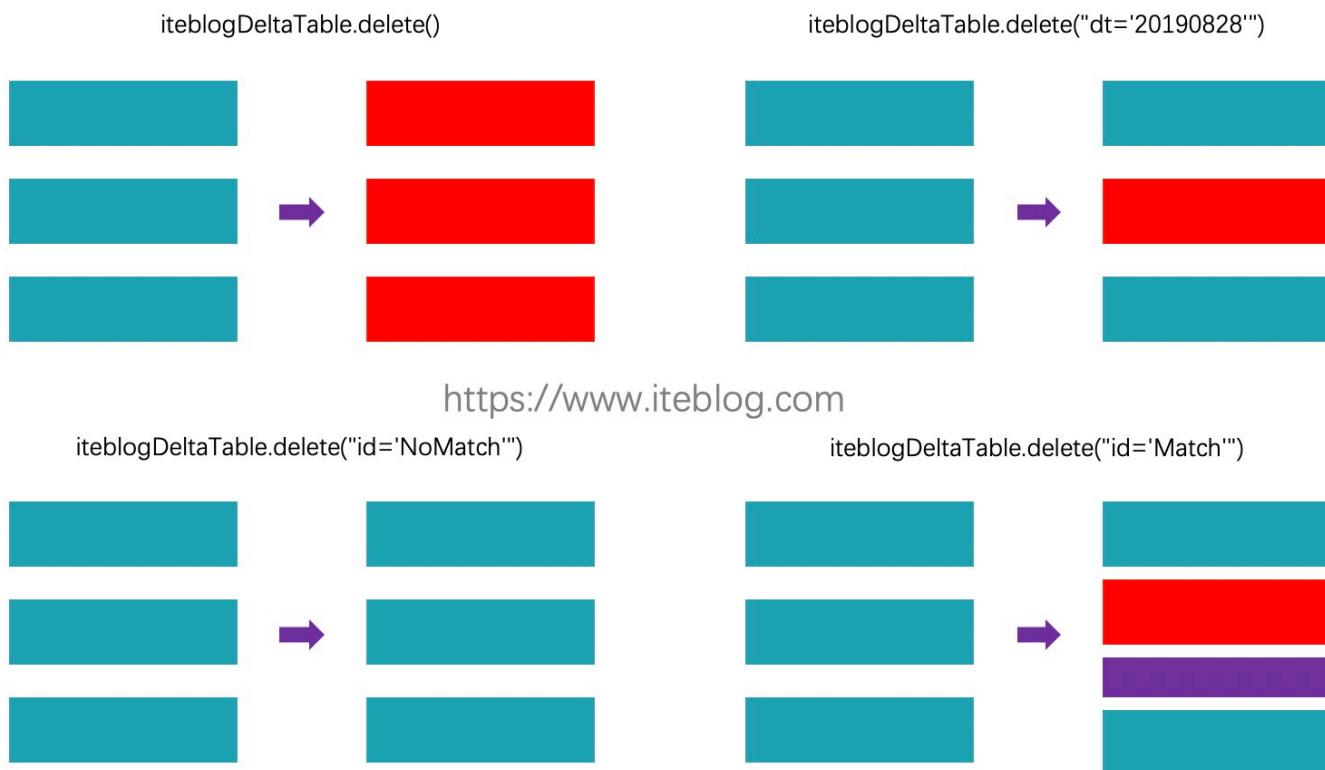
操作之后，就可以拿到当前表的最新版本，紧接着开启乐观事务锁。thunk(txn)

这个就是执行我们上面的 performDelete(sparkSession, deltaLog, txn) 方法。Delta Lake
删除的整个核心就在 performDelete 方法里面了。

如果某个文件里面有数据需要删除，那么这个文件会被标记为删除，然后这个文件里面不需要删除的数据需要重新写到一个新文件里面。那么在 performDelete 方法里面我们就需要知道哪些数据需要删除，这些数据对应的文件在哪里以及是否需要些事务日志。Delta Lake
将删除实现分为三大情况：

- 1、如果执行 delete 的时候并没有传递相关的删除条件，也就是上面例子的 iteblogDeltaTable.delete()，这时候其实就是删除当前 Delta Lake 表的所有数据。那这种情况最好处理了，只需要直接删除 Delta Lake 表对应的所有文件即可；
- 2、如果执行 delete 的时候传递了相关删除条件，而这个删除条件只是分区字段，比如 dt 是 Delta Lake 表的分区字段，然后我们执行了 iteblogDeltaTable.delete("dt = '20190828'")
这样相关的删除操作，那么我们可以直接从缓存在内存中的快照 (snapshot，也就是通过上面的 update()
函数初始化的) 拿到需要删除哪些文件，直接删除即可，而且不需要执行数据重写操作。
- 3、最后一种情况就是用户删除的时候含有一些非分区字段的过滤条件，这时候我们就需要扫描底层数据，获取需要删除的数据在哪个文件里面，这又分两种情况：
 - 3.1、Delta Lake 表并不存在我们需要删除的数据，这时候不需要做任何操作，直接返回，就连事务日志都不用记录；
 - 3.2、这种情况是最复杂的，我们需要计算需要删除的数据在哪个文件里面，然后把对应的文件里面不需要删除的数据重写到新的文件里面（如果没有，就不生成新文件），最后记录事务日志。

为了加深印象，我画了一张图希望大家能够理解上面的过程。



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

上图中每个绿色的框代表一个分区目录下的文件，红色代表标记为删除的文件，也就是事务日志中使用 remove

标记的文件，紫色代表移除需要删除的数据之后新生成的文件，也就是事务日志里面使用 add 标记的文件。

下面我们来详细分析删除的操作。

```
private def performDelete(
    sparkSession: SparkSession, deltaLog: DeltaLog, txn: OptimisticTransaction) = {
  import sparkSession.implicits._

  var numTouchedFiles: Long = 0
  var numRewrittenFiles: Long = 0
  var scanTimeMs: Long = 0
  var rewriteTimeMs: Long = 0

  val startTime = System.nanoTime()
  val numFilesTotal = deltaLog.snapshot.numOfFiles

  val deleteActions: Seq[Action] = condition match {
    // 对应上面情况1，delete 操作没有传递任何条件
  }
}
```

```
case None =>
    // 直接将内存中快照里面所有的 AddFile 文件拿出来
    val allFiles = txn.filterFiles(Nil)

    numTouchedFiles = allFiles.size
    scanTimeMs = (System.nanoTime() - startTime) / 1000 / 1000

    val operationTimestamp = System.currentTimeMillis()
    // 将 AddFile 标记成 RemoveFile
    allFiles.map(_.removeWithTimestamp(operationTimestamp))
    // 传递了删除过滤条件，对应上面情况2、3
    case Some(cond) =>
        // metadataPredicates 为分区删除条件
        // otherPredicates 为其他删除条件
        val (metadataPredicates, otherPredicates) =
            DeltaTableUtils.splitMetadataAndDataPredicates(
                cond, txn.metadata.partitionColumns, sparkSession)

        // 如果只有分区删除条件，也就是 dt= "20190828" 这样的过滤条件，对应上面情况2
        if (otherPredicates.isEmpty) {
            val operationTimestamp = System.currentTimeMillis()
            // 从快照中拿出符合这个分区条件的 AddFile 文件
            val candidateFiles = txn.filterFiles(metadataPredicates)

            scanTimeMs = (System.nanoTime() - startTime) / 1000 / 1000
            numTouchedFiles = candidateFiles.size

            // 将 AddFile 标记成 RemoveFile
            candidateFiles.map(_.removeWithTimestamp(operationTimestamp))
        } else { // 对应上面情况3，含有其他字段的删除条件，这时候我们需要扫描底层数据获取这些
            // 找到删除的数据潜在的 AddFile 文件列表
            val candidateFiles = txn.filterFiles(metadataPredicates ++ otherPredicates)

            numTouchedFiles = candidateFiles.size
            val nameToAddFileMap = generateCandidateFileMap(deltaLog.dataPath, candidateFiles)

            val fileIndex = new TahoeBatchFileIndex(
                sparkSession, "delete", candidateFiles, deltaLog, tahoeFileIndex.path, txn.snapshot)
            // Keep everything from the resolved target except a new TahoeFileIndex
            // that only involves the affected files instead of all files.
            val newTarget = DeltaTableUtils.replaceFileIndex(target, fileIndex)

            // 这个就是潜在需要删除的文件对应的 Dataset
            val data = Dataset.ofRows(sparkSession, newTarget)
            val filesToRewrite =
```

```
withStatusCode("DELTA", s"Finding files to rewrite for DELETE operation") {
    // 没有需要潜在删除的 AddFile 文件
    if (numTouchedFiles == 0) {
        Array.empty[String]
    } else {
        // 找到删除数据所在的文件
        data.filter(new Column(cond)).select(new Column(InputFileName())).distinct()
            .as[String].collect()
    }
}

scanTimeMs = (System.nanoTime() - startTime) / 1000 / 1000
// 对应上面情况3.1，如果没有找到需要删除的数据所在文件，那么删除的文件就是 Nil，不需要做事务日志
if (filesToRewrite.isEmpty) {
    Nil
} else {
    // 对应上面情况3.2，找到我们需要删除的文件列表，
    // 那我们需要将需要删除文件里面不用删除的数据重新写到新文件
    // Do the second pass and just read the affected files
    val baseRelation = buildBaseRelation(
        sparkSession, txn, "delete", tahoeFileIndex.path, filesToRewrite, nameToAddFileMap)
    // Keep everything from the resolved target except a new TahoeFileIndex
    // that only involves the affected files instead of all files.
    val newTarget = DeltaTableUtils.replaceFileIndex(target, baseRelation.location)

    val targetDF = Dataset.ofRows(sparkSession, newTarget)
    // 将删除过滤条件取反，也就是 id > 10 变成 id <= 10
    val filterCond = Not(EqualNullSafe(cond, Literal(true, BooleanType)))
    // 拿到潜在删除文件中不需要删除的数据
    val updatedDF = targetDF.filter(new Column(filterCond))

    // rewrittenFiles 就是新增的文件
    val rewrittenFiles = withStatusCode(
        "DELTA", s"Rewriting ${filesToRewrite.size} files for DELETE operation") {
        // 开始将潜在需要删除文件里面不需要删除的数据写入到新文件
        txn.writeFiles(updatedDF)
    }

    numRewrittenFiles = rewrittenFiles.size
    rewriteTimeMs = (System.nanoTime() - startTime) / 1000 / 1000 - scanTimeMs

    val operationTimestamp = System.currentTimeMillis()
    // 需要删除的文件和新增的文件集合
    removeFilesFromPaths(deltaLog, nameToAddFileMap, filesToRewrite, operationTimestamp)
}
```

```
    ++ rewrittenFiles
}
}
}

// 如果匹配到需要删除的文件，那么需要记录事务日志
if (deleteActions.nonEmpty) {
    // 写事务日志，也就是写到 _delta_log 目录下，这个我们在前面分析了。
    txn.commit(deleteActions, DeltaOperations.Delete(condition.map(_.sql).toSeq))
}

recordDeltaEvent(
    deltaLog,
    "delta.dml.delete.stats",
    data = DeleteMetric(
        condition = condition.map(_.sql).getOrElse("true"),
        numFilesTotal,
        numTouchedFiles,
        numRewrittenFiles,
        scanTimeMs,
        rewriteTimeMs)
)
}
```

上面注释已经很清楚说明了 Delta Lake 的删除过程了。从上面的执行过程也可以看出，Delta Lake 删除操作的代价还是挺高的，所以官方也建议删除数据的时候提供分区过滤条件，这样可以避免扫描全表的数据。

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: [【】\(\)](#)