Apache Spark Delta Lake 更新使用及实现原理代码解析

Apache Spark Delta Lake 的更新（update）和删除都是在 0.3.0 版本发布的，参见这里，对应的 Patch 参见这里。和前面几篇源码分析文章一样，我们也是先来看看在 Delta Lake 里面如何使用更新这个功能。

Delta Lake 更新使用

Delta Lake 的官方文档为我们提供如何使用 Update 的几个例子，参见这里，如下：

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

val iteblogDeltaTable = DeltaTable.forPath(path)

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

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

// 对表中所有的行中 content 字段后面加上 -100
iteblogDeltaTable.update(Map("content" -> concat_ws("-", col("content"), lit("100"))))
```

```
// 对表中所有的行中 content 字段后面加上 -100
iteblogDeltaTable.updateExpr(Map("content" -> "concat(content, '-100')"))
```

Delta Lake
的更新可以带条件和不带条件，带条件会更新满足条件的行；如果不带条件会更新整张表。当 Delta Lake 表有满足条件的行，Delta Lake 会更新相关的数据，并在表的 _delta_log
目录下生成一个事务日志，内容类似下面的：

{"commitInfo":{"timestamp":1568102735945,"operation":"UPDATE","operationParameters":{"predicate":"((id#731L % cast(2 as bigint)) = cast(0 as bigint))"},"readVersion":0,"isBlindAppend":false}}
{"remove":{"path":"part-00001-a39b9ad5-1c4f-48e2-8d45-c6bfc6f37127-c000.snappy.parquet","deletionTimestamp":1568102735812,"dataChange":true}}
{"remove":{"path":"part-00000-254cbe0a-ad45-4b24-b45d-507e32cc598d-c000.snappy.parquet","deletionTimestamp":1568102735812,"dataChange":true}}
{"add":{"path":"part-00000-040e39bf-f282-4969-a802-fa40a51ec6e6-c000.snappy.parquet","partitionValues":{},"size":944,"modificationTime":1568102735000,"dataChange":true}}
{"add":{"path":"part-00001-2c36bac1-1c13-42d7-93db-ce1c94d1f5b2-c000.snappy.parquet","partitionValues":{},"size":959,"modificationTime":1568102735000,"dataChange":true}}

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

注意

- 执行 Update 的时候，原文件还存在在当前表的目录下，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 去更新数据，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 更新数据的，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 去更新 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 更新是如何实现的

目前 Delta Lake 的更新方法的入口是 io.delta.tables.DeltaTable 中定义的 update/updateExpr 方法，里面有 Java 和 Scala 版本的实现，具体如下：

```
def update(set: Map[String, Column]): Unit = {
```

```
  executeUpdate(set, None)
}

def update(set: java.util.Map[String, Column]): Unit = {
  executeUpdate(set.asScala, None)
}

def update(condition: Column, set: Map[String, Column]): Unit = {
  executeUpdate(set, Some(condition))
}

def update(condition: Column, set: java.util.Map[String, Column]): Unit = {
  executeUpdate(set.asScala, Some(condition))
}

def updateExpr(set: Map[String, String]): Unit = {
  executeUpdate(toStrColumnMap(set), None)
}

def updateExpr(set: java.util.Map[String, String]): Unit = {
  executeUpdate(toStrColumnMap(set.asScala), None)
}

def updateExpr(condition: String, set: Map[String, String]): Unit = {
  executeUpdate(toStrColumnMap(set), Some(functions.expr(condition)))
}

def updateExpr(condition: String, set: java.util.Map[String, String]): Unit = {
  executeUpdate(toStrColumnMap(set.asScala), Some(functions.expr(condition)))
}
```

带 java.util.Map[String, String] 参数的均为 Java 的
API。可以看到，上面所有的方法最终都是调用了 executeUpdate 方法， 这个方法定义在
io.delta.tables.execution.DeltaTableOperations 特质里面，实现如下：

```
protected def executeUpdate(
    set: Map[String, Column],
    condition: Option[Column]): Unit = {
  val setColumns = set.map{ case (col, expr) => (col, expr) }.toSeq

  // Current UPDATE 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.map {_.expr}, "UPDATE")

  // 
  val update = makeUpdateTable(self, condition, setColumns)
  // 
  val resolvedUpdate =
    UpdateTable.resolveReferences(update, tryResolveReferences(sparkSession)(_, update))
  // 
  val updateCommand = PreprocessTableUpdate(sparkSession.sessionState.conf)(resolvedUpd
ate)
  // 
  updateCommand.run(sparkSession)
}
```

和 delete 一样，目前不支持带有子查询的更新。

- 这里调用 makeUpdateTable 方法，主要将我们传进去的更新列进行规整，比如将 `a.b` 修改为 a.b，然后构造 UpdateTable 对象，UpdateTable 是一个 case class，扩展至 UnaryNode，里面实现很简单。
- 由于我们传进来的 update 更新条件和更新表达式并没有和表进行绑定，比如得看下用户传进来的字段在不在表里面；传进来的字段对应于表的哪个位置等。UpdateTable.resolveReferences 作用就在此，比如本文最开始的 demo 里面 id % 2 == 0 会变成 (id#679L % cast(2 as bigint)) = cast(0 as bigint)，而 concat(content, '-100') 会变成 concat(content#680, '-100') 这样的。
- 构造 UpdateCommand，UpdateCommand 里面的 updateExpressions 参数比较重要，这个参数会包含更新的表所有的列信息。比如我们上面 demo 里面的 Delta 表一共有三列：id、content、dt，那么根据我们更新的逻辑，updateExpressions 里面得到的结果就是 id#679L、concat(content#680, -100)、dt#681，这样的好处是后面处理更新很方便。
- 更新的核心处理逻辑

updateCommand 的 run 方法实现如下：

```
final override def run(sparkSession: SparkSession): Seq[Row] = {
  recordDeltaOperation(tahoeFileIndex.deltaLog, "delta.dml.update") {
    // 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，执行更新操作。
    deltaLog.withNewTransaction { txn =>
      performUpdate(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，所以如果我们执行更新之前需要做一些校验。校验通过之后开始执行更新操作，由于更新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 的实现我们在前面的文章已经介绍了（参见 这里）这里就不再介绍了。

和 Delete 执行操作类似，更新的时候也需要找出哪些文件需要更新，哪些不需要更新。需要更新的文件 Spark 会读出里面的数据，并按照更新的条件对相关列进行更新，并把结果写入到新文件（更新文件里面不需要更新的行也写到这个新文件），并且把原文件标记为删除。所以总结起来 Delta Lake 更新主要有以下三大情况：

- 情况1：如果更新的表是分区表，我们的更新条件里面带分区字段，但是没有命中表的任何分区，这时候就不需要任何更新了，也不用读取 Delta 表，直接返回即可。比如我们的 iteblog 表的分区字段为 dt，而且值只有 2019-09-01、2019-09-02、2019-09-03 这三个，但是你更新的条件为 dt = '2019-09-11' 或者 dt = '2019-09-11' and id = 123。
- 情况2：如果更新的表是分区表，而且我们的更新条件只是分区字段。比如我们需要更新 iteblog 表，分区为 dt，我们使用 dt = '2019-09-11' 之类的条件去更新数据，那这时候我们直接可以从事务日志快照里面拿到需要更新的文件，然后需要更新的文件标记为 RemoveFile。
- 情况3：前面两种情况直接可以通过事务日志的最新快照拿到需要更新的文件。第三种情况需要先扫描 Delta 表，拿出符合更新条件所在行所在的文件，这里又分两种情况：
    - 情况3.1：更新条件没有命中任何文件，这时候就不需要更新了。比如我们表里面 id 都是 10 条件去更新数据，肯定没有满足的行，这时候就不用更新文件了，也不用记录事务日志。
    - 情况3.1：更新条件命中 Delta 表的文件，我们把这部分文件路径拿出来，然后再读出这部分文件里面的数据进行更新，更新的数据写到新文件，原文件标记为 RemoveFile。

好了，下面我们来看下 performUpdate 方法的实现：

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

```
// 监控用的统计数据
var numTouchedFiles: Long = 0
var numRewrittenFiles: Long = 0
var scanTimeMs: Long = 0
var rewriteTimeMs: Long = 0

val startTime = System.nanoTime()
// 当前版本中新增的文件个数
val numFilesTotal = deltaLog.snapshot.numOfFiles

// 更新条件
val updateCondition = condition.getOrElse(Literal(true, BooleanType))
// 将更新条件进行拆分
val (metadataPredicates, dataPredicates) =
  DeltaTableUtils.splitMetadataAndDataPredicates(
    updateCondition, txn.metadata.partitionColumns, sparkSession)
// 获取需要更新的候选文件
val candidateFiles = txn.filterFiles(metadataPredicates ++ dataPredicates)
val nameToAddFile = generateCandidateFileMap(deltaLog.dataPath, candidateFiles)

scanTimeMs = (System.nanoTime() - startTime) / 1000 / 1000

// 对应上面分析的情况1
val actions: Seq[Action] = if (candidateFiles.isEmpty) {
  // Case 1: Do nothing if no row qualifies the partition predicates
  // that are part of Update condition
  Nil
} else if (dataPredicates.isEmpty) {  // 对应上面分析的情况2
  // Case 2: Update all the rows from the files that are in the specified partitions
  // when the data filter is empty
  numTouchedFiles = candidateFiles.length

  val filesToRewrite = candidateFiles.map(_.path)
  val operationTimestamp = System.currentTimeMillis()
  val deleteActions = candidateFiles.map(_.removeWithTimestamp(operationTimestamp))

  val rewrittenFiles = rewriteFiles(sparkSession, txn, tahoeFileIndex.path,
    filesToRewrite, nameToAddFile, updateCondition)

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

  deleteActions ++ rewrittenFiles
} else {
  // 对应上面的情况3
  // Case 3: Find all the affected files using the user-specified condition
```

```
  val fileIndex = new TahoeBatchFileIndex(
    sparkSession, "update", 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)
  val data = Dataset.ofRows(sparkSession, newTarget)
  val filesToRewrite =
    withStatusCode("DELTA", s"Finding files to rewrite for UPDATE operation") {
      data.filter(new Column(updateCondition)).select(input_file_name())
        .distinct().as[String].collect()
    }

  scanTimeMs = (System.nanoTime() - startTime) / 1000 / 1000
  numTouchedFiles = filesToRewrite.length

  if (filesToRewrite.isEmpty) {
    // 对应上面的情况3.1
    // Case 3.1: Do nothing if no row qualifies the UPDATE condition
    Nil
  } else {
    // 对应上面的情况3.2
    // Case 3.2: Delete the old files and generate the new files containing the updated
    // values
    val operationTimestamp = System.currentTimeMillis()
    val deleteActions =
      removeFilesFromPaths(deltaLog, nameToAddFile, filesToRewrite, operationTimestamp)
    val rewrittenFiles =
      withStatusCode("DELTA", s"Rewriting ${filesToRewrite.size} files for UPDATE operation") {
        rewriteFiles(sparkSession, txn, tahoeFileIndex.path,
          filesToRewrite, nameToAddFile, updateCondition)
      }

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

    deleteActions ++ rewrittenFiles
  }
}

if (actions.nonEmpty) {
  txn.commit(actions, DeltaOperations.Update(condition.map(_.toString)))
}

recordDeltaEvent(
  deltaLog,
  "delta.dml.update.stats",
```

```
    data = UpdateMetric(
      condition = condition.map(_.sql).getOrElse("true"),
      numFilesTotal,
      numTouchedFiles,
      numRewrittenFiles,
      scanTimeMs,
      rewriteTimeMs)
  )
}
```