主要说明一下两个主题

- Spark 没有提供的关键性能
- Spark 缺失的性能会给数据湖造成什么问题

spark writer API 的原子性

所谓原子性是事务要求(ACID)中的第一项，完成一件事情，要么全部完成，要么全部不完成，spark 在使用writer 写数据的时候会有一个参数 mode， 它的值为 "append"、"overrite"，append 好理解，而overrite的写方式是，先删除已经存在的数据，然后将旧的删除，也就是spark在写数据的时候并没有利用锁的机制去保证原子性，这意味着当在使用overrite的方式写数据的时候，如果刚好将原数据给删掉了，但是如果刚刚准备开始写新数据的时候出了错误，那么原始数据丢了，新数据也没写进去，这会是很危险的事情呀。

```
  spark.range(100).repartition(1).write.mode("overwrite").csv(basicPath + "test-1")
  import spark.implicits._

  Try(
    spark
      .range(100)
      .repartition(1)
      .map { i =>
        if (i > 50) {
          Thread.sleep(5000)
          throw  new RuntimeException("Oops!")
        }
        i
      }
      .write.mode("overwrite")
      .csv(basicPath + "test-1")
  )
```

代码执行的时候，第二次以overwrite写入的时候，抛出异常，原始的数据已经被删除，写第二次写的时候，一条也没有写进去。

如果使用append 的方式写进去，并在写的过程中跑出异常。

```
  spark.range(100).repartition(1).write.mode("overwrite").csv(basicPath + "test-1")
  import spark.implicits._

  Try(
    spark
      .range(100)
      .repartition(1)
      .map { i =>
```

```
        if (i > 50) {
          Thread.sleep(5000)
          throw  new RuntimeException("Oops!")
        }
        i
      }
      .write.mode("append")
      .csv(basicPath + "test-1")
  )
```

这个时候写进去的文件中是100条数据。

如果修改下代码

```
  spark.range(100).repartition(1).write.mode("overwrite").csv(basicPath + "test-1")
  import spark.implicits._

  Try(
    spark
      .range(100)
      .repartition(1)
      .map { i =>
        if (i > 100) {
          Thread.sleep(5000)
          throw  new RuntimeException("Oops!")
        }
        i
      }
      .write.mode("append")
      .csv(basicPath + "test-1")
  )
```

也就是不跑出异常，那么结果中有200条数据。

于是：

- spark 的overrite没有原子性的保证，操作会发生比较大的危险性
- append 看这它的表现行为是原子性的，但实际上这个原子行为并不是spark保证的，而是`Spark would use Hadoop's FileOutputCommitter version 1 algorithm to implement Job commit and abort`, spark 本身是并没有原子行为的，借助 hadoop 实现一定程度的原子性

> spark 借助 hadoop 实现一定程度的原子性，但是这个原子性在不同的平台是不一样的，是说实现这个原子性写数据的时候的速度会不一样，hdfs 是最快的，而在云端的存储设备是非常慢的[文字中间有说明连接]。

连续性Consistancy

所谓连续性就是保证数据的状态都是合法的。`the consistency ensures that the data is always in the valid state`

```
  spark.range(100).repartition(1).write.mode("overwrite").csv(basicPath + "test-1")
  import spark.implicits._

  Try(
    spark
      .range(100)
      .repartition(1)
      .map { i =>
        if (i > 50) {
          Thread.sleep(5000)
          throw  new RuntimeException("Oops!")
        }
        i
      }
      .write.mode("overwrite")
      .csv(basicPath + "test-1")
  )
```

这块代码就能说明，spark并不能保证连续性。执行完代码发现，写入的目录中什么都没有了，包括新数据。

隔离性Isolation 和 持久性Durability

所谓隔离性就是一个没有提交的事务并不会影响到现在正在执行中的需要使用到数据的的任务。spark并没有commit 这一说，或者理解job commit(一个job 会有一个action)， 也就不具备隔离性，`hence, Spark is not offering isolation types even at an individual API level`。

而持久性，是存储层提供了能力，spark 本身是并不具备该能力。

上面的内容是准所周知的，`spark 没有提供ACID能力`，上面已经展示了一次spark 没有提供`ACID`导致的数据污染的问题，那么还有什么问题呢?

Schema 约束(Schema Enforcement) 问题

我现在有这样的两份数据

```
iris copy.csv
  "FName","LName","Phone","Age"
  "aa"   ,"AA"   ,"123"  ,52
  "bb"   ,"BB"   ,"321"  ,48

iris copy doubleAge.csv
  "FName","LName","Phone","Age"
  "aa"   ,"AA"   ,"123"  ,52.323
  "bb"   ,"BB"   ,"321"  ,48.623
```

这里主要是通过Age这个字段演示。我执行这样的任务

```
  val df = spark.read
    .format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(basicPath + "iris copy.csv")

  df.show
  import spark.implicits._
  val df1 = df.select(
    $"FName",
    $"LName",
    $"Phone",
    $"Age",
    (when($"Age" > 50, "Old").otherwise("Young")).alias("AgeGroup")
  )

  df1.show

  df1.write.format("parquet").mode("append").save(basicPath + "iris copy.csv2")

  val df2 = spark.read.format("parquet").load(basicPath + "iris copy.csv2")

  df2.show
```

第一次我跑iris copy.csv数据，然后将结果写到 "iris copy.csv2" 表中。是没问题的，一切执行成功。

第二次执行iris copy doubleAge.csv 数据，同时使用append的方式将数据写进iris copy.csv2：

```
  val df = spark.read
    .format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(basicPath + "iris copy doubleAge.csv")

  df.show
  import spark.implicits._
  val df1 = df.select(
    $"FName",
    $"LName",
    $"Phone",
    $"Age",
    (when($"Age" > 50, "Old").otherwise("Young")).alias("AgeGroup")
  )

  df1.show

```

```
19    df1.write.format("parquet").mode("append").save(basicPath + "iris
copy.csv2")
20
21    val df2 = spark.read.format("parquet").load(basicPath + "iris
copy.csv2")
22
23    df2.show
```

这里很明显的问题是Age的格式第一次是INT, 第二次是Double，但是神奇的是数据可以执行到第19行，也就是数据能写进去，但是当读的时候就出问题了

`Column: [Age], Expected: double, Found: INT32`

这样可能发生数据shcema 变化的场景会很多，如迭代式的job。

大量小文件问题

会有这样的场景，不同收集产生的小文件，然后将这些小文件存为一个大文件，可以有一个定时的sparkJob去做这件事情，如每一个小时收集一次，但是这一个小时能产生多少数据呢，100G 呢还是100M呢，那么处理这些数据效率咋样呢?

先看看处理这些小数据会有那些步骤

1. 搜查目录下的文件
2. list 所有文件
3. 逐个打开，然后移到一个文件中，关闭打开的文件
4. 如果涉及到写Hive 表，还需要涉及到metadata的写

涉及到文件的打开，关闭，这是在文件处理中最耗时间，而且如果小文件很多很多呢，想想就效率低的可怕。

所以才有向kfka 这样的工具，先把所有的文件都收集起来，达到一定的大小之后保存下来。尽管如此，还是有很多小文件的，那么可以先将文件压缩为一个大文件，然后在去处理，这个时候数据处理作业应等待压缩作业完成。为什么？因为压缩需要写入大文件并删除小文件，所以该操作不是事务性的。因此，它会使系统处于不一致状态。压缩期间甚至无法执行可靠的读取操作。

Partition in Apache Spark

在写spark会做哪些事情呢？ 无非就是增、改、过滤、然后存进去，问题就在过滤这步，可能会读进来很多很多很多的数据，但是经过过滤之后数据就很小了，那么需求就来了，能不能直接读进来的数据就正是需要的那部分呢，spark的分区就可以做这回事情，可以将某一类数据存到一个分区中，但是这种方法只能是对如下特点的数据是非常有效果的

- 时间顺序列，例如日期
- 基数很小的数据，如国家code等。

如果不是上述特点的数据呢，如基数很大的列作为依据分区，那么就会有很多很多的分区，似乎就回到了`大量小数据问题`了。

小结

- spark主要不足
    - 缺少ACID等导致很多问题，而由于缺少ACID事务导致：

- 解决处理大量小数据集问题
- 不能做到按需读取数据

源文章

Delta lake 就是为解决上述的问题。