**How to Achieve Exactly-Once Semantics in Spark Streaming**

Exactly-once semantics is one of the advanced topics of stream processing. To process every message once and only once, in spite of system or network failure, not only the stream processing framework needs to provide such functionality, but also the message delivery system, the output data store, as well as how we implement the processing procedure, altogether can we ensure the exactly-once semantics. In this article, I’ll demonstrate how to use Spark Streaming, with Kafka as data source and MySQL the output storage, to achieve exactly-once stream processing.

[](http://spark.apache.org/docs/latest/img/streaming-arch.png)Spark Streaming

**An Introductory Example**

First let’s implement a simple yet complete stream processing application that receive access logs from Kafka, parse and count the errors, then write the errors per minute metric into MySQL database.

Sample access logs:

|  |  |
| --- | --- |
| 1 2 3 | 2017-07-30 14:09:08 ERROR some message 2017-07-30 14:09:20 INFO some message 2017-07-30 14:10:50 ERROR some message |

Output table, where log\_time should be truncated to minutes:

|  |  |
| --- | --- |
| 1 2 3 4 | create table error\_log (  log\_time datetime primary key,  log\_count int not null default 0 ); |

Scala projects are usually managed by sbt tool. Let’s add the following dependencies into build.sbt file. We’re using Spark 2.2 with Kafka 0.10. The choice of database library is ScalikeJDBC 3.0.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | scalaVersion := "2.11.11"  libraryDependencies ++= Seq(  "org.apache.spark" %% "spark-streaming" % "2.2.0" % "provided",  "org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.2.0",  "org.scalikejdbc" %% "scalikejdbc" % "3.0.1",  "mysql" % "mysql-connector-java" % "5.1.43" ) |

The complete code can be found on GitHub ([link](https://github.com/jizhang/spark-sandbox/blob/master/src/main/scala/ExactlyOnce.scala)), so here only shows the major parts of the application:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 | // initialize database connection ConnectionPool.singleton("jdbc:mysql://localhost:3306/spark", "root", "")  // create Spark streaming context val conf = new SparkConf().setAppName("ExactlyOnce").setIfMissing("spark.master", "local[2]") val ssc = new StreamingContext(conf, Seconds(5))  // create Kafka DStream with Direct API val messages = KafkaUtils.createDirectStream[String, String](ssc,  LocationStrategies.PreferConsistent,  ConsumerStrategies.Subscribe[String, String](Seq("alog"), kafkaParams))  messages.foreachRDD { rdd =>  // do transformation  val result = rdd.map(\_.value)  .flatMap(parseLog) // utility function to parse log line into case class  .filter(\_.level == "ERROR")  .map(log => log.time.truncatedTo(ChronoUnit.MINUTES) -> 1)  .reduceByKey(\_ + \_)  .collect()   // store result into database  DB.autoCommit { implicit session =>  result.foreach { case (time, count) =>  sql"""  insert into error\_log (log\_time, log\_count)  value (${time}, ${count})  on duplicate key update log\_count = log\_count + values(log\_count)  """.update.apply()  }  } } |

**Stream Processing Semantics**

There’re three semantics in stream processing, namely at-most-once, at-least-once, and exactly-once. In a typical Spark Streaming application, there’re three processing phases: receive data, do transformation, and push outputs. Each phase takes different efforts to achieve different semantics.

For **receiving data**, it largely depends on the data source. For instance, reading files from a fault-tolerant file system like HDFS, gives us exactly-once semantics. For upstream queues that support acknowledgement, e.g. RabbitMQ, we can combine it with Spark’s write ahead logs to achieve at-least-once semantics. For unreliable receivers like socketTextStream, there might be data loss due to worker/driver failure and gives us undefined semantics. Kafka, on the other hand, is offset based, and its direct API can give us exactly-once semantics.

When **transforming data** with Spark’s RDD, we automatically get exactly-once semantics, for RDD is itself immutable, fault-tolerant and deterministically re-computable. As long as the source data is available, and there’s no side effects during transformation, the result will always be the same.

**Output operation** by default has at-least-once semantics. The foreachRDD function will execute more than once if there’s worker failure, thus writing same data to external storage multiple times. There’re two approaches to solve this issue, idempotent updates, and transactional updates. They are further discussed in the following sections.

**Exactly-once with Idempotent Writes**

If multiple writes produce the same data, then this output operation is idempotent. saveAsTextFile is a typical idempotent update; messages with unique keys can be written to database without duplication. This approach will give us the equivalent exactly-once semantics. Note though it’s usually for map-only procedures, and it requires some setup on Kafka DStream.

* Set enable.auto.commit to false. By default, Kafka DStream will commit the consumer offsets right after it receives the data. We want to postpone this action unitl the batch is fully processed.
* Turn on Spark Streaming’s checkpointing to store Kafka offsets. But if the application code changes, checkpointed data is not reusable. This leads to a second option:
* Commit Kafka offsets after outputs. Kafka provides a commitAsync API, and the HasOffsetRanges class can be used to extract offsets from the initial RDD:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | messages.foreachRDD { rdd =>  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  rdd.foreachPartition { iter =>  // output to database  }  messages.asInstanceOf[CanCommitOffsets].commitAsync(offsetRanges) } |

**Exactly-once with Transactional Writes**

Transactional updates require a unique identifier. One can generate from batch time, partition id, or Kafka offsets, and then write the result along with the identifier into external storage within a single transaction. This atomic operation gives us exactly-once semantics, and can be applied to both map-only and aggregation procedures.

Usually writing to database should happen in foreachPartition, i.e. in worker nodes. It is true for map-only procedure, because Kafka RDD’s partition is correspondent to Kafka partition, so we can extract each partition’s offset like this:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | messages.foreachRDD { rdd =>  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  rdd.foreachPartition { iter =>  val offsetRange = offsetRanges(TaskContext.get.partitionId)  } } |

But for shuffled operations like the error log count example, we need to first collect the result back into driver and then perform the transaction.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | messages.foreachRDD { rdd =>  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  val result = processLogs(rdd).collect() // parse log and count error  DB.localTx { implicit session =>  result.foreach { case (time, count) =>  // save to error\_log table  }  offsetRanges.foreach { offsetRange =>  val affectedRows = sql"""  update kafka\_offset set offset = ${offsetRange.untilOffset}  where topic = ${topic} and `partition` = ${offsetRange.partition}  and offset = ${offsetRange.fromOffset}  """.update.apply()   if (affectedRows != 1) {  throw new Exception("fail to update offset")  }  }  } } |

If the offsets fail to update, or there’s a duplicate offset range detected by offset != $fromOffset, the whole transaction will rollback, which guarantees the exactly-once semantics.

**Conclusion**

Exactly-once is a very strong semantics in stream processing, and will inevitably bring some overhead to your application and impact the throughput. It’s also not applicable to [windowed](https://github.com/koeninger/kafka-exactly-once/blob/master/src/main/scala/example/Windowed.scala) operations. So you need to decide whether it’s necessary to spend such efforts, or weaker semantics even with few data loss will suffice. But surely knowing how to achieve exactly-once is a good chance of learning, and it’s a great fun.

**References**

* <http://blog.cloudera.com/blog/2015/03/exactly-once-spark-streaming-from-apache-kafka/>
* <http://spark.apache.org/docs/latest/streaming-programming-guide.html>
* <http://spark.apache.org/docs/latest/streaming-kafka-0-10-integration.html>
* <http://kafka.apache.org/documentation.html#semantics>