如若需要可以查看最新在线文档:

<http://www.zhenchao.cf/bigdata_spark-project_atguigu>

<http://www.zhenchao.tk/bigdata_spark-project_atguigu>

<https://zhenchao125.github.io/bigdata_spark-project_atguigu>

# 第 7 部分 Structured Streaming

## 第 1 章 Structured Streaming 概述

从 spark2.0 开始, spark 引入了一套新的流式计算模型: Structured Streaming.

该组件进一步降低了处理数据的延迟时间, 它实现了“有且仅有一次(Exectly Once)” 语义, 可以保证数据被精准消费.

Structured Streaming 基于 Spark SQl 引擎, 是一个具有弹性和容错的流式处理引擎. 使用 Structure Streaming 处理流式计算的方式和使用批处理计算静态数据(表中的数据)的方式是一样的.

随着流数据的持续到达, Spark SQL 引擎持续不断的运行并得到最终的结果. 我们可以使用 Dataset/DataFrame API 来表达流的聚合, 事件-时间窗口(event-time windows), 流-批处理连接(stream-to-batch joins)等等. 这些计算都是运行在被优化过的 Spark SQL 引擎上. 最终, 通过 chekcpoin 和 WALs(Write-Ahead Logs), 系统保证end-to-end exactly-once.

总之, Structured Streaming 提供了快速, 弹性, 容错, end-to-end exactly-once 的流处理, 而用户不需要对流进行推理(比如 spark streaming 中的流的各种转换).

默认情况下, 在内部, Structured Streaming 查询使用微批处理引擎(micro-batch processing engine)处理, 微批处理引擎把流数据当做一系列的小批job(small batch jobs ) 来处理. 所以, 延迟低至 100 毫秒, 从 Spark2.3, 引入了一个新的低延迟处理模型:Continuous Processing, 延迟低至 1 毫秒.

## 第 2 章 Structure Streaming 快速入门

为了使用最稳定最新的 Structure Streaming, 我们使用最新版本.

本入门案例是从一个网络端口中读取数据, 并统计每个单词出现的数量.

### 2.1 导入依赖

<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-sql\_2.11</artifactId>  
 <version>2.4.3</version>  
</dependency>

### 2.2 具体实现

package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.StreamingQuery  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/12 10:56 AM  
 \*/  
object WordCount1 {  
 def main(args: Array[String]): Unit = {  
 // 1. 创建 SparkSession. 因为 ss 是基于 spark sql 引擎, 所以需要先创建 SparkSession  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("WordCount1")  
 .getOrCreate()  
 import spark.implicits.\_  
 // 2. 从数据源(socket)中加载数据.  
 val lines: DataFrame = spark.readStream  
 .format("socket") // 设置数据源  
 .option("host", "hadoop201")  
 .option("port", 9999)  
 .load  
   
 // 3. 把每行数据切割成单词  
 val words: Dataset[String] = lines.as[String].flatMap(\_.split("\\W"))  
   
 // 4. 计算 word count  
 val wordCounts: DataFrame = words.groupBy("value").count()  
   
 // 5. 启动查询, 把结果打印到控制台  
 val query: StreamingQuery = wordCounts.writeStream  
 .outputMode("complete")  
 .format("console")  
 .start  
 query.awaitTermination()  
   
 spark.stop()  
 }  
}

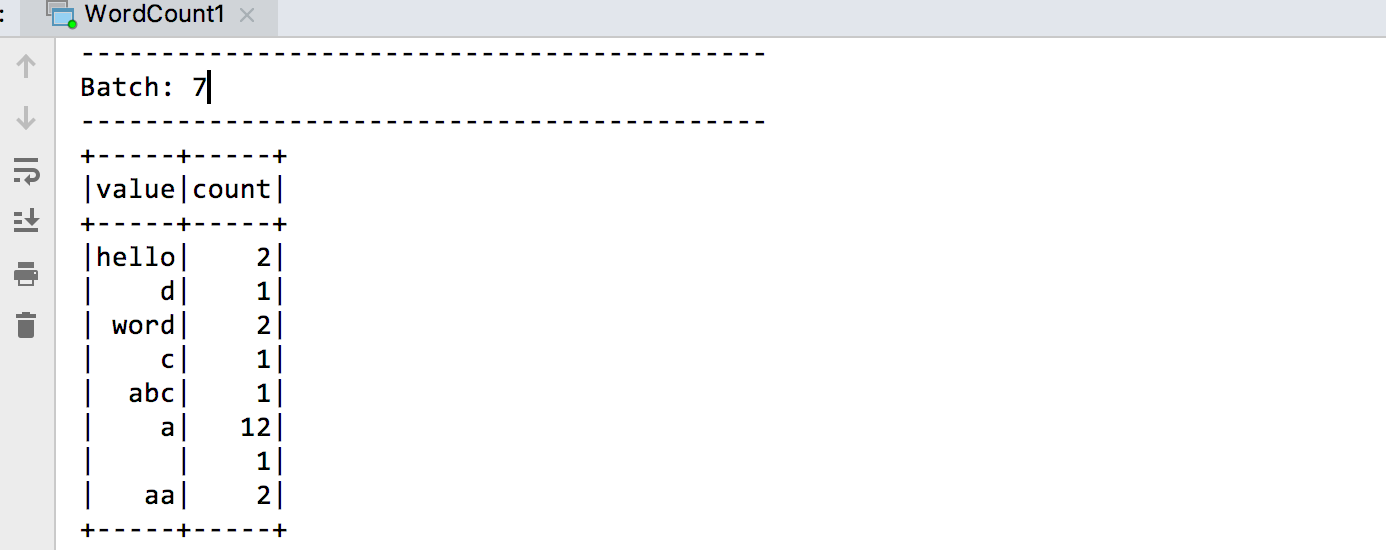
### 2.3 测试

1. 在 hadoop201 启动 socket 服务:

* nc -lk 9999

1. 启动 Structured Steaming 程序

输入一些单词, 查看程序的输出结果:



### 2.4 代码说明

1. DataFrame lines 表示一个“无界表(unbounded table)”, 存储着流中所有的文本数据. 这个无界表包含列名为value的一列数据, 数据的类型为String, 而且在流式文本数据中的每一行(line)就变成了无界表中的的一行(row). 注意, 这时我们仅仅设置了转换操作, 还没有启动它, 所以现在还没有收到任何数据
2. 紧接着我们把 DateFrame 通过 .as[String] 变成了 DataSet, 所以我们可以切割每行为多个单词.得到的 words DataSet包含了所有的单词.
3. 最后, 我们通过value(每个唯一的单词)进行分组得到wordCounts DataFrame, 并且统计每个单词的个数. 注意, wordCounts是一个流式DataFrame, 它表示流中正在运行的单词数(the running word counts of the stream).
4. 我们必须在流式数据(streaming data)上启动查询. 剩下的实际就是开始接收数据和计算个数. 为此, 当数据更新的时候, 我们通过outputMode("complete")来打印完整的计数集到控制台, 然后通过.start来启动流式计算.
5. 代码执行之后, 流式计算将会在后台启动. 查询对象(query: StreamingQuery)可以激活流式查询(streaming query), 然后通过awaitTermination()来等待查询的终止,从而阻止查询激活之后进程退出.

## 第 3 章 Structured Streaming 编程模型

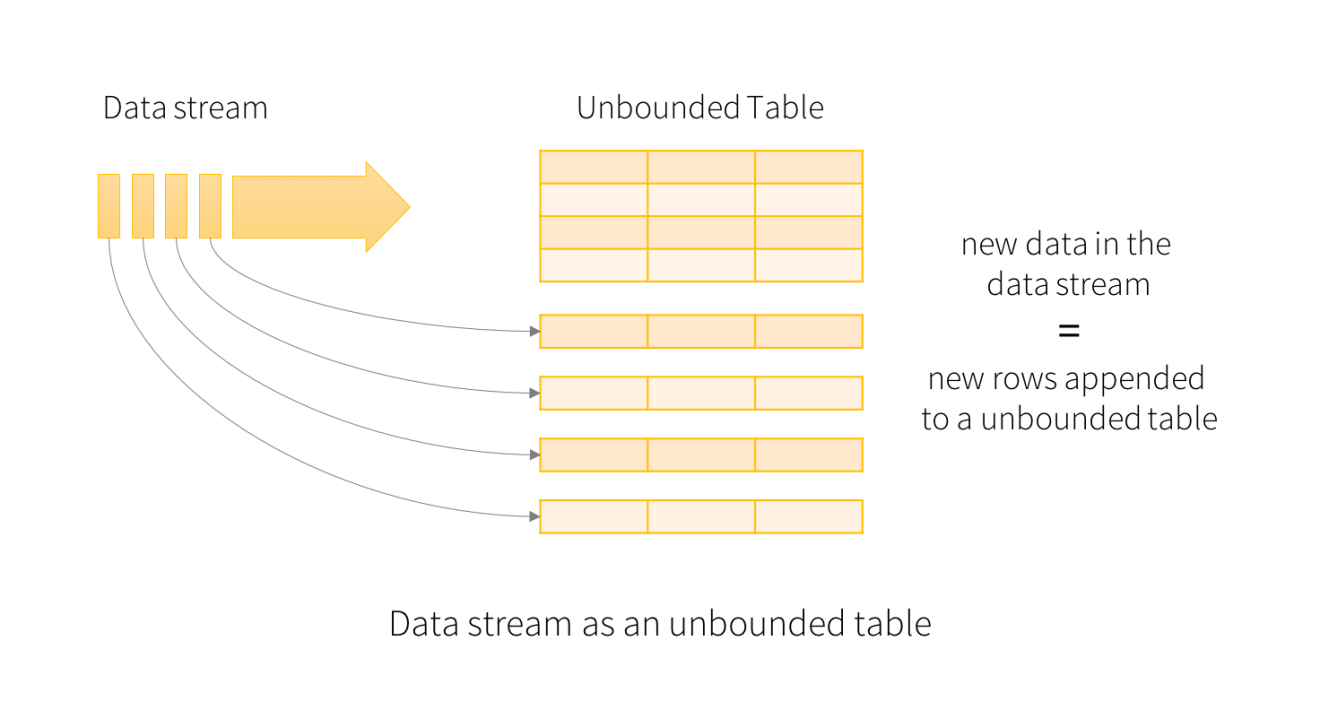
Structured Streaming 的核心思想是:把持续不断的流式数据当做一个不断追加的表.

这使得新的流式处理模型同批处理模型非常相像. 我们可以表示我们的流式计算类似于作用在静态数表上的标准批处理查询, spark 在一个无界表上以增量查询的方式来运行.

### 3.1 基本概念

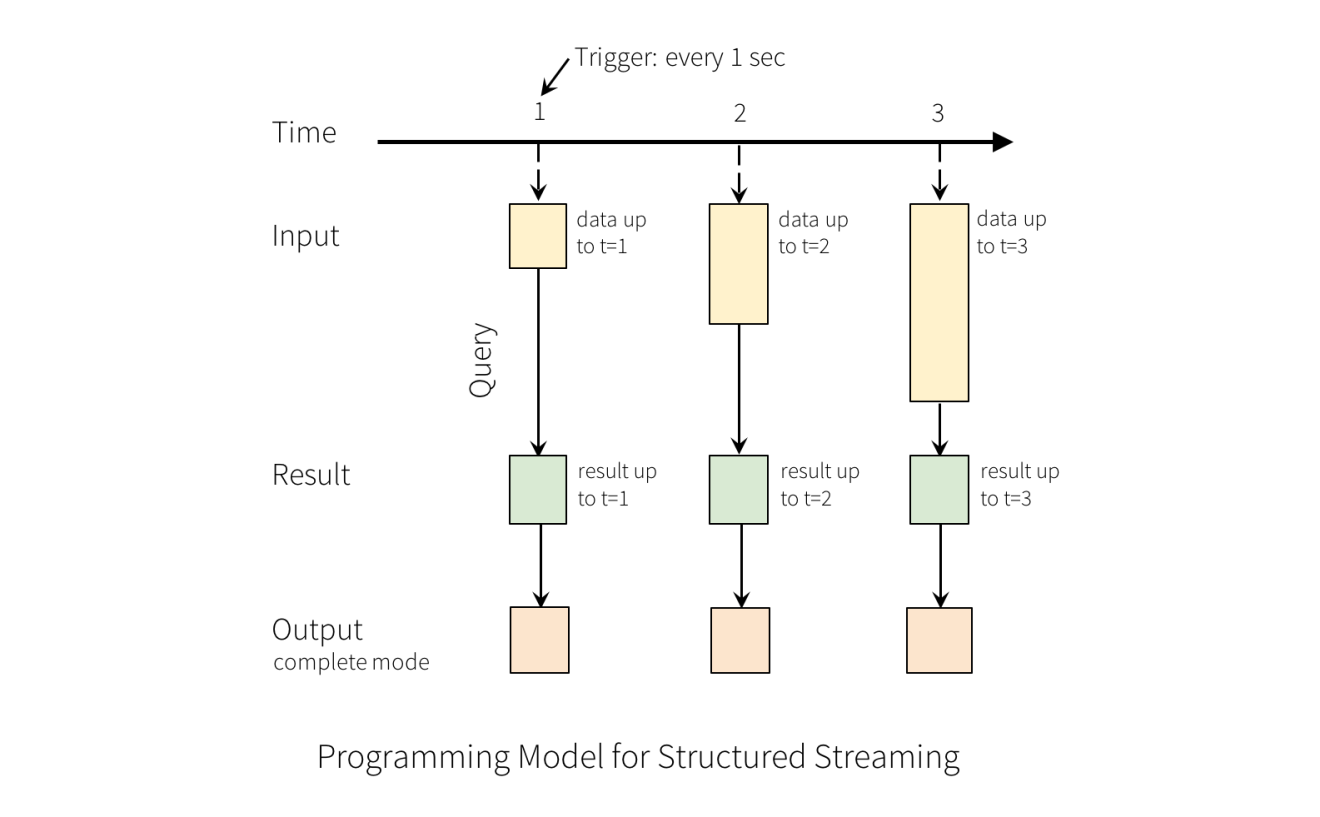
#### 3.1.1 输入表

把输入数据流当做输入表(Input Table). 到达流中的每个数据项(data item)类似于被追加到输入表中的一行.



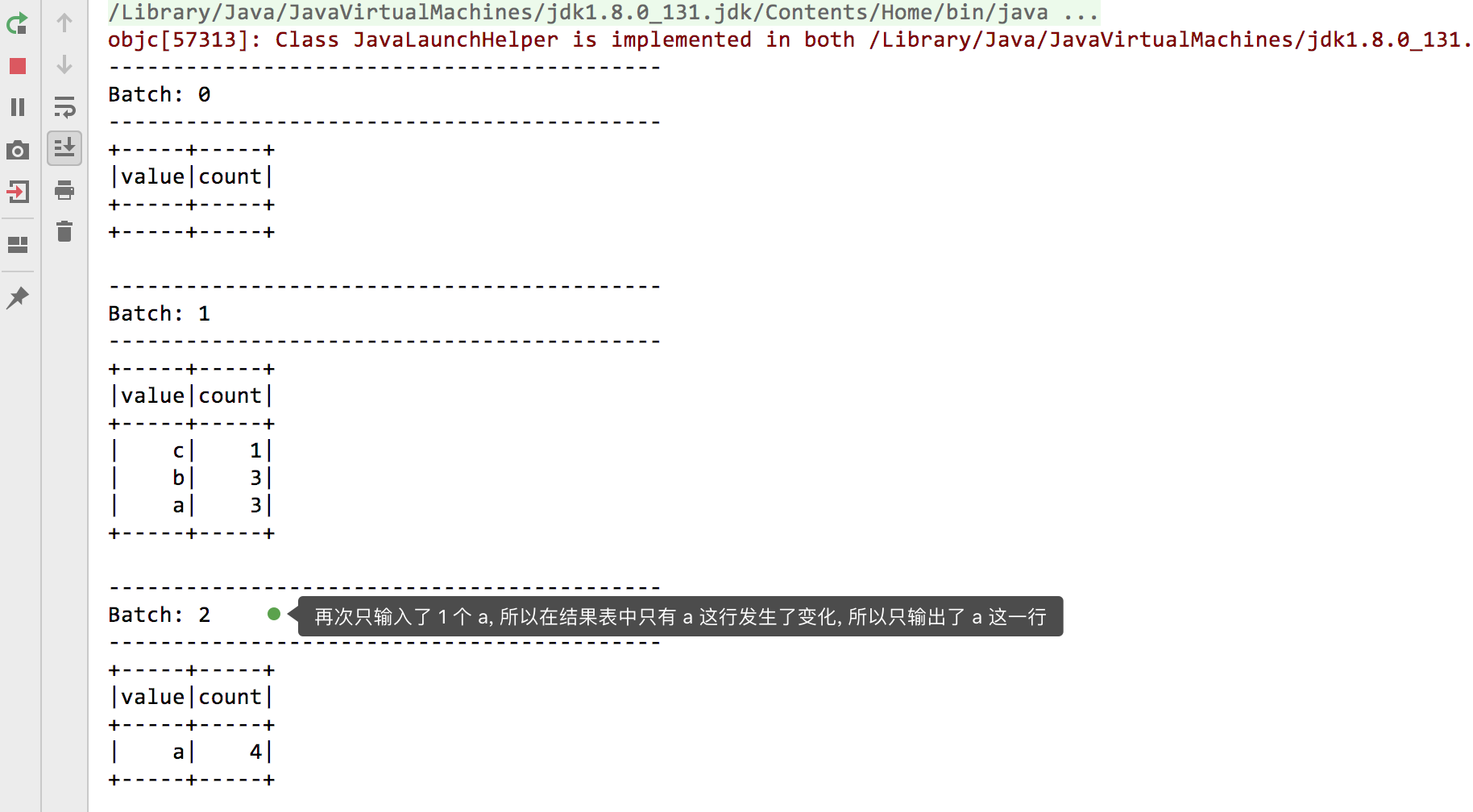
#### 3.1.2 结果表

作用在输入表上的查询将会产生“结果表(Result Table)”. 每个触发间隔(trigger interval, 例如 1s), 新行被追加到输入表, 最终会更新结果表. 无论何时更新结果表, 我们都希望将更改的结果行写入到外部接收器(external sink)



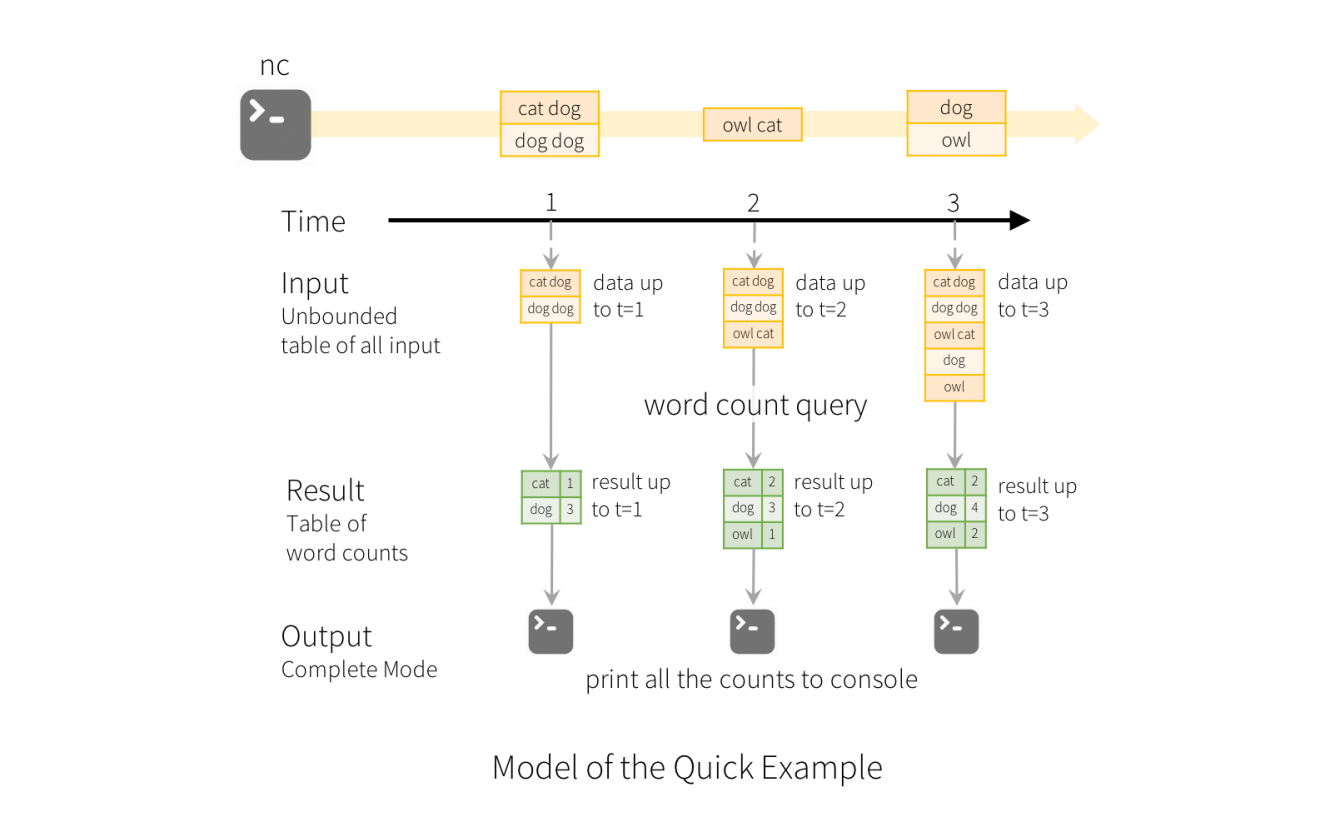
#### 3.1.3 输出

输出(Output)定义为写到外部存储. 输出模式(outputMode)有 3 种:

1. Complete Mode 整个更新的结果表会被写入到外部存储. 存储连接器负责决定如何处理整个表的写出(类似于 spark streaming 中的有转态的转换).
2. Append Mode 从上次触发结束开始算起, 仅仅把那些新追加到结果表中的行写到外部存储(类似于无状态的转换). 该模式仅适用于不会更改结果表中行的那些查询. (如果有聚合操作, 则必须添加 wartemark, 否则不支持此种模式)
3. Update Mode 从上次触发结束开始算起, 仅仅在结果表中更新的行会写入到外部存储. 此模式从 2.1.1 可用. 注意, Update Mode 与 Complete Mode 的不同在于 Update Mode 仅仅输出改变的那些行. 如果查询不包括聚合操作, 则等同于 Append Mode 

#### 3.1.4 快速入门代码的再次说明

lines DataFrame是“输入表”, wordCounts DataFrame 是“结果表”, 从输入表到结果表中间的查询同静态的 DataFrame 是一样的. 查询一旦启动, Spark 会持续不断的在 socket 连接中检测新的数据, 如果其中有了新的数据, Spark 会运行一个增量(incremental)查询, 这个查询会把前面的运行的 count 与新的数据组合在一起去计算更新后的 count.



注意, Structured Streaming 不会实现整个表. 它从流式数据源读取最新的可用数据, 持续不断的处理这些数据, 然后更新结果, 并且会丢弃原始数据. 它仅保持最小的中间状态的数据, 以用于更新结果(例如前面例子中的中间counts)

### 3.2 处理事件-时间和延迟数据(Handling Event-time and Late Data)

Structured streaming 与其他的流式引擎有很大的不同. 许多系统要求用户自己维护运行的聚合, 所以用户自己必须推理数据的一致性(at-least-once, or at-most-once, or exactly-once). 在Structured streaming模型中, 当有新数据的时候, spark 负责更新结果表, 从而减轻了用户的推理工作.

我们来看下个模型如何处理基于事件时间的处理和迟到的数据。

Event-time 是指嵌入到数据本身的时间, 或者指数据产生的时间. 对大多数应用程序来说, 我们想基于这个时间去操作数据. 例如, 如果我们获取 IoT(Internet of Things) 设备每分钟产生的事件数, 我们更愿意使用数据产生时的时间(event-time in the data), 而不是 spark 接收到这些数据时的时间.

在这个模型中, event-time 是非常自然的表达. 来自设备的每个时间都是表中的一行, event-time 是行中的一列. 允许基于窗口的聚合(例如, 每分钟的事件数)仅仅是 event-time 列上的特殊类型的分组（grouping）和聚合（aggregation）: 每个时间窗口是一个组，并且每一行可以属于多个窗口/组。因此，可以在静态数据集和数据流上进行基于事件时间窗口（ event-time-window-based）的聚合查询，从而使用户操作更加方便。

此外, 该模型也可以自然的处理晚于 event-time 的数据, 因为spark 一直在更新结果表, 所以它可以完全控制更新旧的聚合数据，或清除旧的聚合以限制中间状态数据的大小。

自 Spark 2.1 起，开始支持 watermark 来允许用于指定数据的超时时间（即接收时间比 event-time 晚多少），并允许引擎相应的清理旧状态。

### 3.3 容错语义

提供端到端的exactly-once语义是 Structured Streaming 设计的主要目标. 为了达成这一目的, spark 设计了结构化流数据源, 接收器和执行引擎(Structured Streaming sources, the sinks and the execution engine)以可靠的跟踪处理的进度, 以便能够对任何失败能够重新启动或者重新处理.

每种流数据源假定都有 offsets(类似于 Kafka offsets) 用于追踪在流中的读取位置. 引擎使用 checkpoint 和 WALs 来记录在每个触发器中正在处理的数据的 offset 范围. 结合可重用的数据源(replayable source)和幂等接收器(idempotent sink), Structured Streaming 可以确保在任何失败的情况下端到端的 exactly-once 语义.

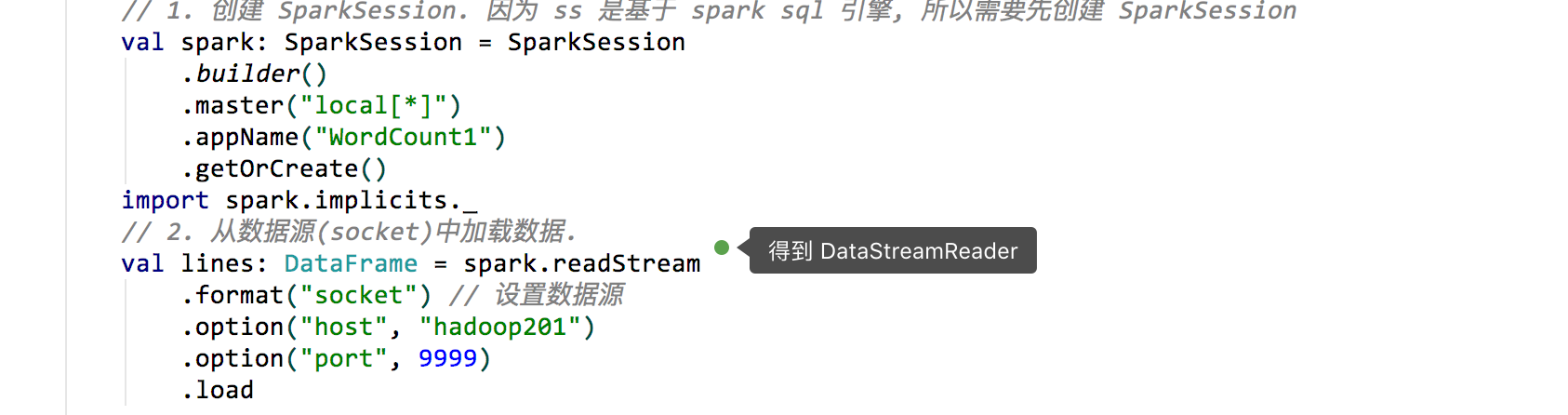
## 第 4 章 操作Streaming DataFrame 和 Streaming DataSet

使用 Structured Streaming 最重要的就是对 Streaming DataFrame 和 Streaming DataSet 进行各种操作.

从 Spark2.0 开始, DataFrame 和 DataSet 可以表示静态有界的表, 也可以表示流式无界表.

与静态 Datasets/DataFrames 类似，我们可以使用公共入口点 SparkSession 从流数据源创建流式 Datasets/DataFrames，并对它们应用与静态 Datasets/DataFrames 相同的操作。

通过spark.readStream()得到一个DataStreamReader对象, 然后通过这个对象加载流式数据源, 就得到一个流式的 DataFrame.



spark 内置了几个流式数据源, 基本可以满足我们的所有需求.

1. File source 读取文件夹中的文件作为流式数据. 支持的文件格式: text, csv, josn, orc, parquet. 注意, 文件必须放置的给定的目录中, 在大多数文件系统中, 可以通过移动操作来完成.
2. kafka source 从 kafka 读取数据. 目前兼容 kafka 0.10.0+ 版本
3. socket source 用于测试. 可以从 socket 连接中读取 UTF8 的文本数据. 侦听的 socket 位于驱动中. 注意, 这个数据源仅仅用于测试.
4. rate source 用于测试. 以每秒指定的行数生成数据，每个输出行包含一个 timestamp 和 value。其中 timestamp 是一个 Timestamp类型(信息产生的时间)，并且 value 是 Long 包含消息的数量. 用于测试和基准测试.

|  |  |  |  |
| --- | --- | --- | --- |
| Source | Options | Fault-tolerant | Notes |
| **File source** | path: path to the input directory, and common to all file formats. maxFilesPerTrigger: maximum number of new files to be considered in every trigger (default: no max) latestFirst: whether to process the latest new files first, useful when there is a large backlog of files (default: false) fileNameOnly: whether to check new files based on only the filename instead of on the full path (default: false). With this set to true, the following files would be considered as the same file, because their filenames, “dataset.txt”, are the same: “file:///dataset.txt” “s3://a/dataset.txt” “s3n://a/b/dataset.txt” “s3a://a/b/c/dataset.txt” For file-format-specific options, see the related methods in DataStreamReader([Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.streaming.DataStreamReader)/[Java](http://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/streaming/DataStreamReader.html)/[Python](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.streaming.DataStreamReader)/[R](http://spark.apache.org/docs/latest/api/R/read.stream.html)). E.g. for “parquet” format options see DataStreamReader.parquet(). In addition, there are session configurations that affect certain file-formats. See the [SQL Programming Guide](http://spark.apache.org/docs/latest/sql-programming-guide.html) for more details. E.g., for “parquet”, see [Parquet configuration](http://spark.apache.org/docs/latest/sql-data-sources-parquet.html#configuration) section. | Yes | Supports glob paths, but does not support multiple comma-separated paths/globs. |
| **Socket Source** | host: host to connect to, must be specified port: port to connect to, must be specified | No |  |
| **Rate Source** | rowsPerSecond (e.g. 100, default: 1): How many rows should be generated per second. rampUpTime (e.g. 5s, default: 0s): How long to ramp up before the generating speed becomes rowsPerSecond. Using finer granularities than seconds will be truncated to integer seconds. numPartitions (e.g. 10, default: Spark’s default parallelism): The partition number for the generated rows. The source will try its best to reach rowsPerSecond, but the query may be resource constrained, and numPartitions can be tweaked to help reach the desired speed. | Yes |  |
| **Kafka Source** | See the [Kafka Integration Guide](http://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html). | Yes |  |

### 4.1 socket source

具体案例参考前面的快速入门

### 4.2 file source

#### 4.2.1 读取普通文件夹内的文件

package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.{StreamingQuery, Trigger}  
import org.apache.spark.sql.types.{LongType, StringType, StructType}  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 9:01 AM  
 \*/  
object ReadFromFile {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("ReadFromFile")  
 .getOrCreate()  
   
 // 定义 Schema, 用于指定列名以及列中的数据类型  
 val userSchema: StructType = new StructType().add("name", StringType).add("age", LongType).add("job", StringType)  
 val user: DataFrame = spark.readStream  
 .format("csv")  
 .schema(userSchema)  
 .load("/Users/lzc/Desktop/csv") // 必须是目录, 不能是文件名  
   
 val query: StreamingQuery = user.writeStream  
 .outputMode("append")  
 .trigger(Trigger.ProcessingTime(0)) // 触发器 数字表示毫秒值. 0 表示立即处理  
 .format("console")  
 .start()  
 query.awaitTermination()  
 }  
}

注意:

前面获取user的代码也可以使用下面的替换:

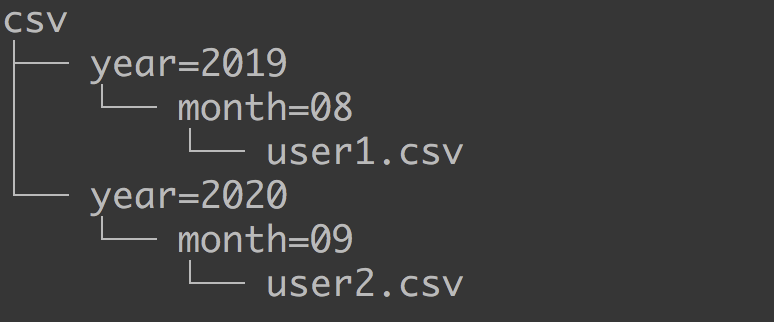
val user: DataFrame = spark.readStream  
 .schema(userSchema)  
 .csv("/Users/lzc/Desktop/csv")

#### 4.2.2 读取自动分区的文件夹内的文件

当文件夹被命名为 “key=value” 形式时, Structured Streaming 会自动递归遍历当前文件夹下的所有子文件夹, 并根据文件名实现自动分区.

如果文件夹的命名规则不是“key=value”形式, 则不会触发自动分区. 另外, 同级目录下的文件夹的命名规则必须一致.

#### 步骤 1: 创建如下目录结构



#### 文件内容:

user1.csv

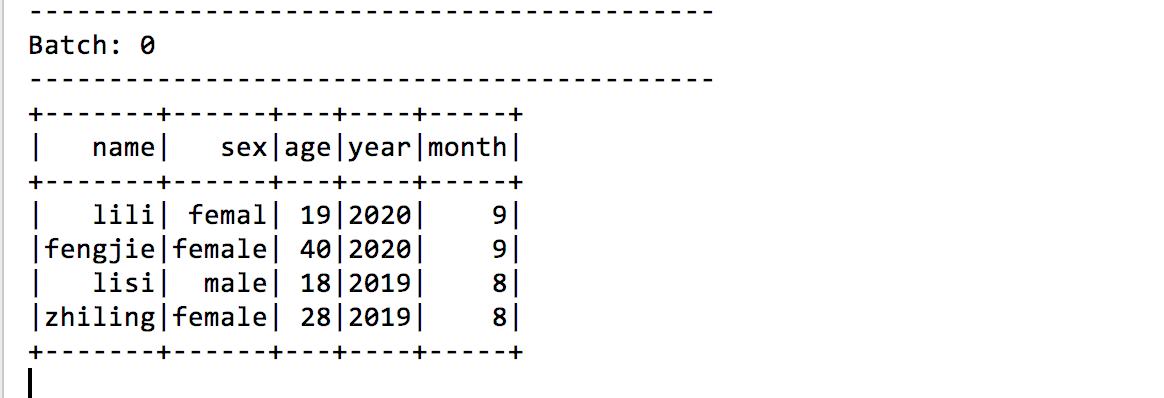
lisi,male,18  
zhiling,female,28

user2.csv

lili,femal,19  
fengjie,female,40

#### 步骤 2: 创建如下代码

package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.{StreamingQuery, Trigger}  
import org.apache.spark.sql.types.{IntegerType, LongType, StringType, StructType}  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 9:01 AM  
 \*/  
object ReadFromFile2 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("ReadFromFile")  
 .getOrCreate()  
   
 // 定义 Schema, 用于指定列名以及列中的数据类型  
 val userSchema: StructType = new StructType().add("name", StringType).add("sex", StringType).add("age", IntegerType)  
   
 val user: DataFrame = spark.readStream  
 .schema(userSchema)  
 .csv("/Users/lzc/Desktop/csv")  
   
 val query: StreamingQuery = user.writeStream  
 .outputMode("append")  
 .trigger(Trigger.ProcessingTime(0)) // 触发器 数字表示毫秒值. 0 表示立即处理  
 .format("console")  
 .start()  
 query.awaitTermination()  
 }  
}



### 4.3 Kafka source

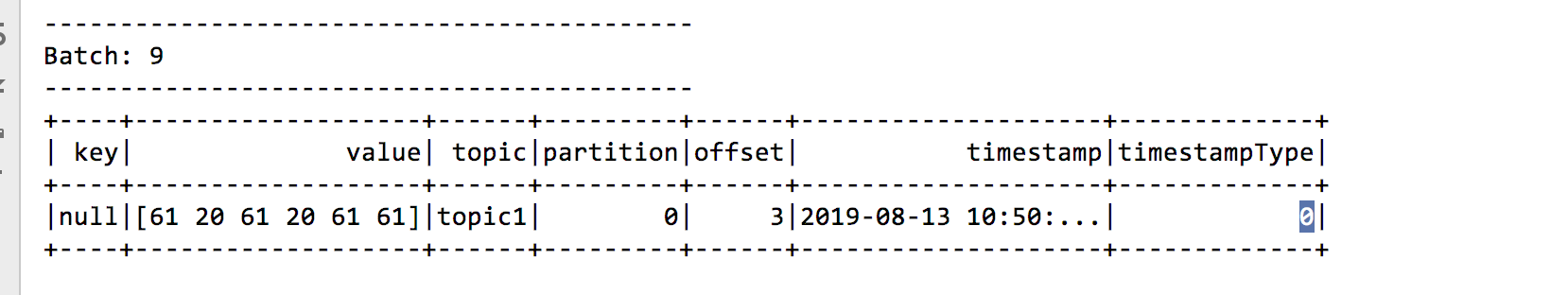
参考文档: http://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html

导入依赖:

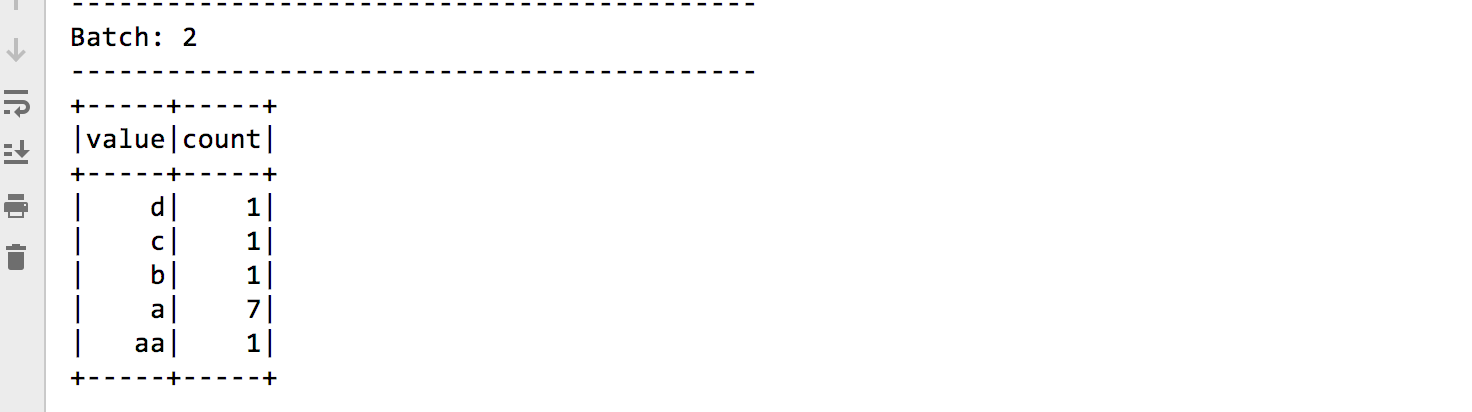
<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-sql-kafka-0-10\_2.12</artifactId>  
 <version>2.4.3</version>  
</dependency

#### 4.3.1 以 Streaming 模式创建 Kafka 工作流

package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.Trigger  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 10:23 AM  
 \*/  
object KafkaSourceDemo {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("KafkaSourceDemo")  
 .getOrCreate()  
   
 // 得到的 df 的 schema 是固定的: key,value,topic,partition,offset,timestamp,timestampType  
 val df: DataFrame = spark.readStream  
 .format("kafka") // 设置 kafka 数据源  
 .option("kafka.bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092")  
 .option("subscribe", "topic1") // 也可以订阅多个主题: "topic1,topic2"  
 .load  
   
   
 df.writeStream  
 .outputMode("update")  
 .format("console")  
 .trigger(Trigger.Continuous(1000))  
 .start  
 .awaitTermination()  
   
   
 }  
}



package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.Trigger  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 10:23 AM  
 \*/  
object KafkaSourceDemo2 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("KafkaSourceDemo")  
 .getOrCreate()  
 import spark.implicits.\_  
 // 得到的 df 的 schema 是固定的: key,value,topic,partition,offset,timestamp,timestampType  
 val lines: Dataset[String] = spark.readStream  
 .format("kafka") // 设置 kafka 数据源  
 .option("kafka.bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092")  
 .option("subscribe", "topic1") // 也可以订阅多个主题: "topic1,topic2"  
 .load  
 .selectExpr("CAST(value AS 26)")  
 .as[String]  
 val query: DataFrame = lines.flatMap(\_.split("\\W+")).groupBy("value").count()  
 query.writeStream  
 .outputMode("complete")  
 .format("console")  
 .option("checkpointLocation", "./ck1") // 下次启动的时候, 可以从上次的位置开始读取  
 .start  
 .awaitTermination()  
 }  
}



#### 4.3.2 通过 Batch 模式创建 Kafka 工作流

这种模式一般需要设置消费的其实偏移量和结束偏移量, 如果不设置 checkpoint 的情况下, 默认起始偏移量 earliest, 结束偏移量为 latest.

该模式为一次性作业(批处理), 而非持续性的处理数据.

package com.atguigu.ss  
  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 10:23 AM  
 \*/  
object KafkaSourceDemo3 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("KafkaSourceDemo")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val lines: Dataset[String] = spark.read // 使用 read 方法,而不是 readStream 方法  
 .format("kafka") // 设置 kafka 数据源  
 .option("kafka.bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092")  
 .option("subscribe", "topic1")  
 .option("startingOffsets", "earliest")  
 .option("endingOffsets", "latest")  
 .load  
 .selectExpr("CAST(value AS STRING)")  
 .as[String]  
   
 val query: DataFrame = lines.flatMap(\_.split("\\W+")).groupBy("value").count()  
   
 query.write // 使用 write 而不是 writeStream  
 .format("console")  
 .save()  
 }  
}

### 4.4 Rate Source

以固定的速率生成固定格式的数据, 用来测试 Structured Streaming 的性能.

package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.Trigger  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 11:42 AM  
 \*/  
object RateSourceDemo {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("RateSourceDemo")  
 .getOrCreate()  
   
 val rows: DataFrame = spark.readStream  
 .format("rate") // 设置数据源为 rate  
 .option("rowsPerSecond", 10) // 设置每秒产生的数据的条数, 默认是 1  
 .option("rampUpTime", 1) // 设置多少秒到达指定速率 默认为 0  
 .option("numPartitions", 2) /// 设置分区数 默认是 spark 的默认并行度  
 .load  
   
 rows.writeStream  
 .outputMode("append")  
 .trigger(Trigger.Continuous(1000))  
 .format("console")  
 .start()  
 .awaitTermination()  
 }  
}

## 第 5 章 操作 Streaming DataFrame/Streaming DataSet

我们可以在streaming DataFrames/Datasets上应用各种操作.

主要分两种: 1. 直接执行 sql 2. 特定类型的 api(DSL)

### 5.1 基本操作

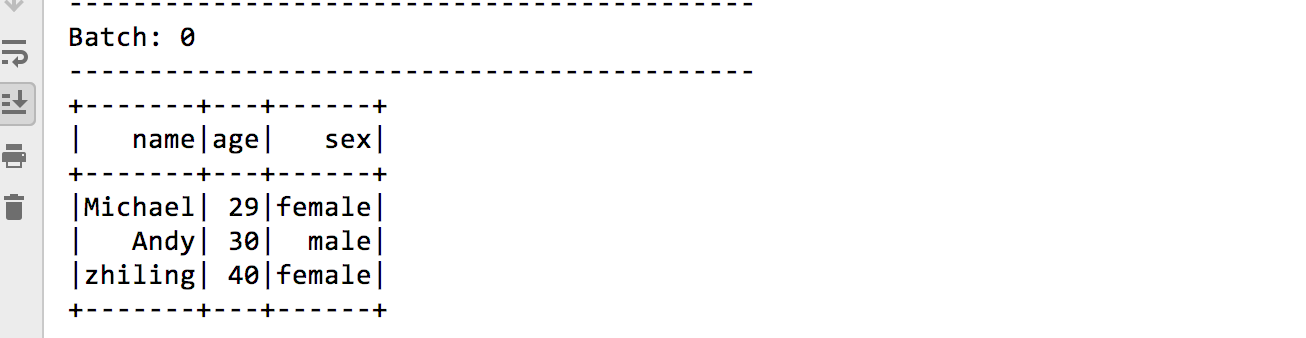
Most of the common operations on DataFrame/Dataset are supported for streaming. 在 DF/DS 上大多数通用操作都支持作用在 Streaming DataFrame/Streaming DataSet 上

一会要处理的数据 people.json 内容:

{"name": "Michael","age": 29,"sex": "female"}  
{"name": "Andy","age": 30,"sex": "male"}  
{"name": "Justin","age": 19,"sex": "male"}  
{"name": "Lisi","age": 18,"sex": "male"}  
{"name": "zs","age": 10,"sex": "female"}  
{"name": "zhiling","age": 40,"sex": "female"}

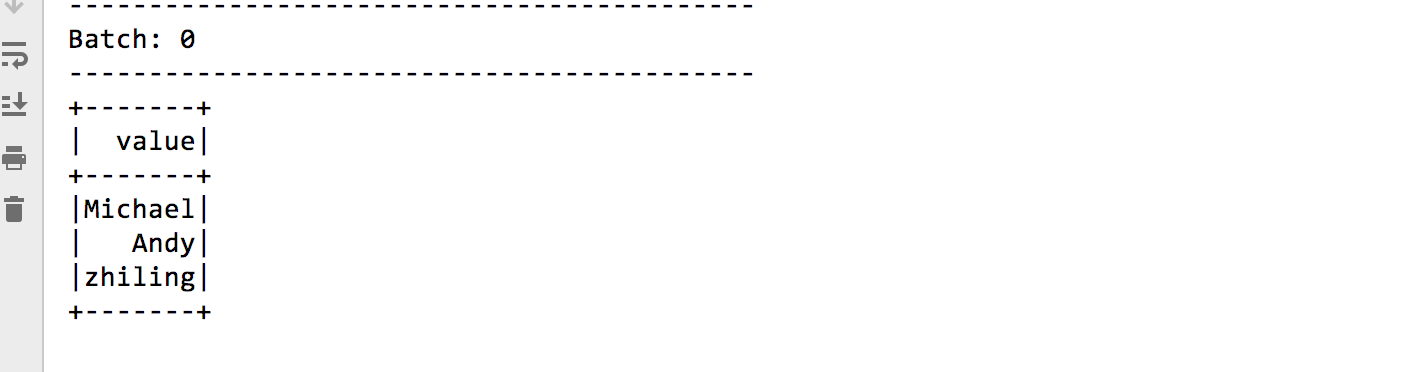
#### 1. 弱类型 api(了解)

package com.atguigu.ss  
  
import org.apache.spark.sql.types.{IntegerType, LongType, StringType, StructType}  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 2:08 PM  
 \*/  
object BasicOperation {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("BasicOperation")  
 .getOrCreate()  
 val peopleSchema: StructType = new StructType()  
 .add("name", StringType)  
 .add("age", LongType)  
 .add("sex", StringType)  
 val peopleDF: DataFrame = spark.readStream  
 .schema(peopleSchema)  
 .json("/Users/lzc/Desktop/data")  
   
   
 val df: DataFrame = peopleDF.select("name","age", "sex").where("age > 20") // 弱类型 api  
 df.writeStream  
 .outputMode("append")  
 .format("console")  
 .start  
 .awaitTermination()  
 }  
}



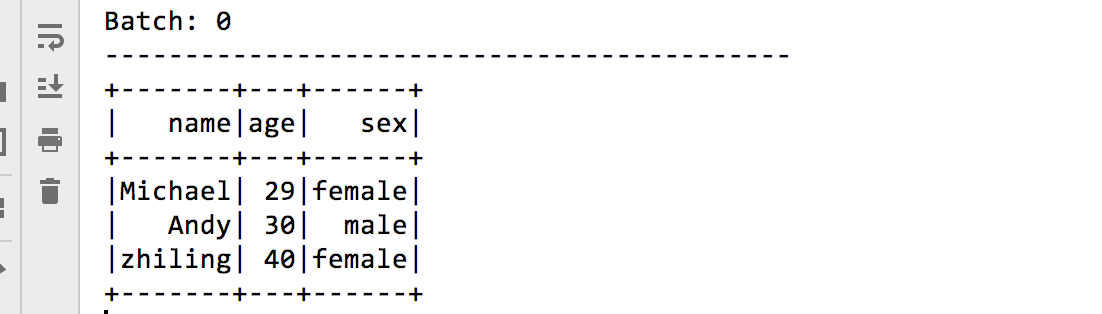
#### 2. 强类型 api(了解)

package com.atguigu.ss  
  
import org.apache.spark.sql.types.{LongType, StringType, StructType}  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 2:08 PM  
 \*/  
object BasicOperation2 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("BasicOperation")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val peopleSchema: StructType = new StructType()  
 .add("name", StringType)  
 .add("age", LongType)  
 .add("sex", StringType)  
 val peopleDF: DataFrame = spark.readStream  
 .schema(peopleSchema)  
 .json("/Users/lzc/Desktop/data")  
   
 val peopleDS: Dataset[People] = peopleDF.as[People] // 转成 ds  
   
   
 val df: Dataset[String] = peopleDS.filter(\_.age > 20).map(\_.name)  
 df.writeStream  
 .outputMode("append")  
 .format("console")  
 .start  
 .awaitTermination()  
   
   
 }  
}  
  
case class People(name: String, age: Long, sex: String)



#### 3. 直接执行 sql(重要)

package com.atguigu.ss  
  
import org.apache.spark.sql.types.{LongType, StringType, StructType}  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 2:08 PM  
 \*/  
object BasicOperation3 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("BasicOperation")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val peopleSchema: StructType = new StructType()  
 .add("name", StringType)  
 .add("age", LongType)  
 .add("sex", StringType)  
 val peopleDF: DataFrame = spark.readStream  
 .schema(peopleSchema)  
 .json("/Users/lzc/Desktop/data")  
   
 peopleDF.createOrReplaceTempView("people") // 创建临时表  
 val df: DataFrame = spark.sql("select \* from people where age > 20")  
   
 df.writeStream  
 .outputMode("append")  
 .format("console")  
 .start  
 .awaitTermination()  
 }  
}



### 5.2 基于 event-time 的窗口操作

#### 5.2.1 event-time 窗口理解

在 Structured Streaming 中, 可以按照事件发生时的时间对数据进行聚合操作, 即基于 event-time 进行操作.

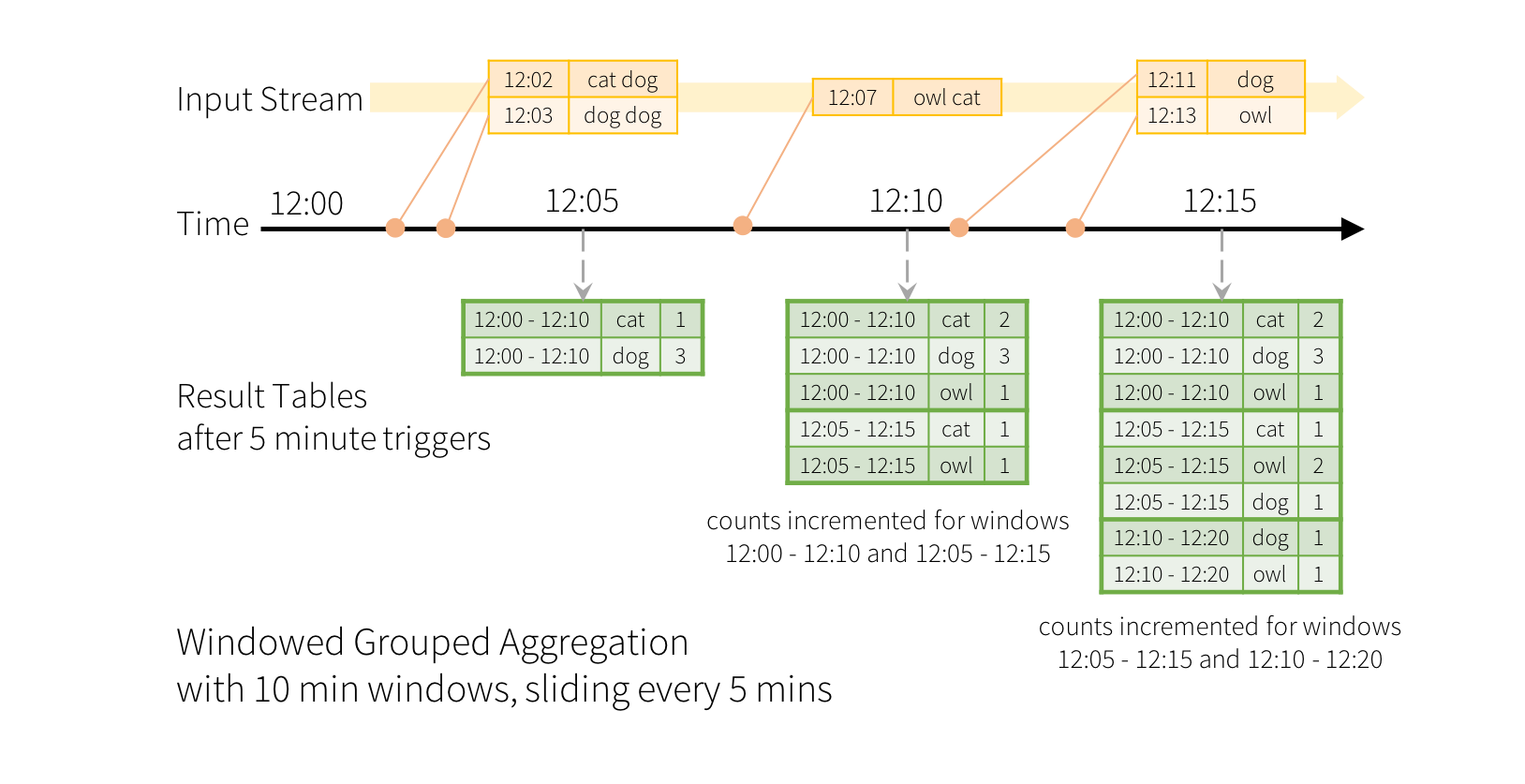
在这种机制下, 即不必考虑 Spark 陆续接收事件的顺序是否与事件发生的顺序一致, 也不必考虑事件到达 Spark 的时间与事件发生时间的关系.

因此, 它在提高数据处理精度的同时, 大大减少了开发者的工作量.

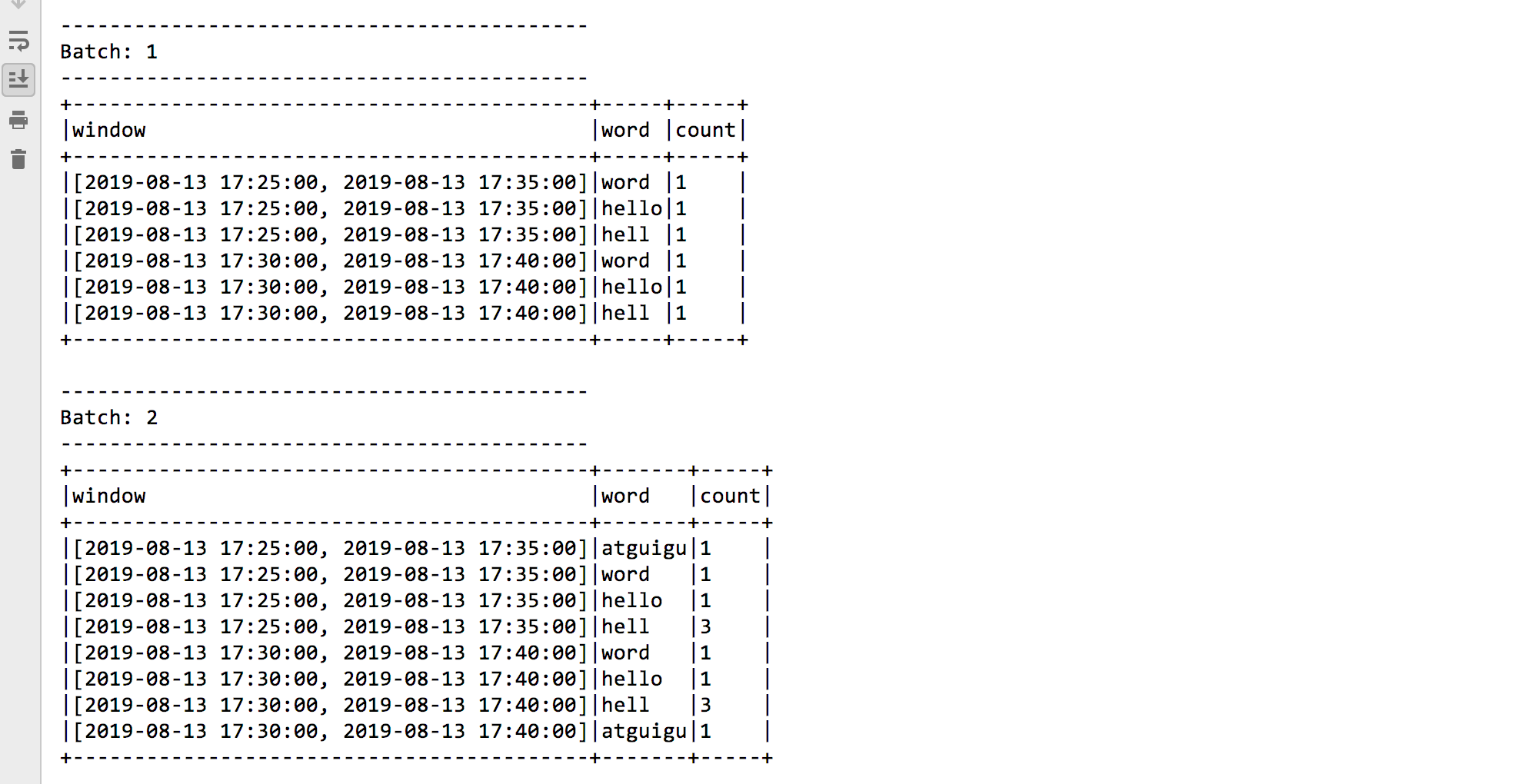
我们现在想计算 10 分钟内的单词, 每 5 分钟更新一次, 也就是说在 10 分钟窗口 12:00 - 12:10, 12:05 - 12:15, 12:10 - 12:20等之间收到的单词量. 注意, 12:00 - 12:10 表示数据在 12:00 之后 12:10 之前到达.

现在，考虑一下在 12:07 收到的单词。单词应该增加对应于两个窗口12:00 - 12:10和12:05 - 12:15的计数。因此，计数将由分组键（即单词）和窗口（可以从事件时间计算）索引。

统计后的结果应该是这样的:



package com.atguigu.ss  
  
import java.sql.Timestamp  
  
import org.apache.spark.sql.streaming.StreamingQuery  
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 4:44 PM  
 \*/  
object WordCountWindow {  
 def main(args: Array[String]): Unit = {  
   
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("WordCount1")  
 .getOrCreate()  
   
 import spark.implicits.\_  
 val lines: DataFrame = spark.readStream  
 .format("socket") // 设置数据源  
 .option("host", "localhost")  
 .option("port", 10000)  
 .option("includeTimestamp", true) // 给产生的数据自动添加时间戳  
 .load  
   
 // 把行切割成单词, 保留时间戳  
 val words: DataFrame = lines.as[(String, Timestamp)].flatMap(line => {  
 line.\_1.split(" ").map((\_, line.\_2))  
 }).toDF("word", "timestamp")  
   
 import org.apache.spark.sql.functions.\_  
   
 // 按照窗口和单词分组, 并且计算每组的单词的个数  
 val wordCounts: Dataset[Row] = words.groupBy(  
 // 调用 window 函数, 返回的是一个 Column 参数 1: df 中表示时间戳的列 参数 2: 窗口长度 参数 3: 滑动步长  
 window($"s", "10 minutes", "5 minutes"),  
 $"word"  
 ).count().orderBy($"window") // 计数, 并按照窗口排序  
   
 val query: StreamingQuery = wordCounts.writeStream  
 .outputMode("complete")  
 .format("console")  
 .option("truncate", "false") // 不截断.为了在控制台能看到完整信息, 最好设置为 false  
 .start  
 query.awaitTermination()  
 }  
}



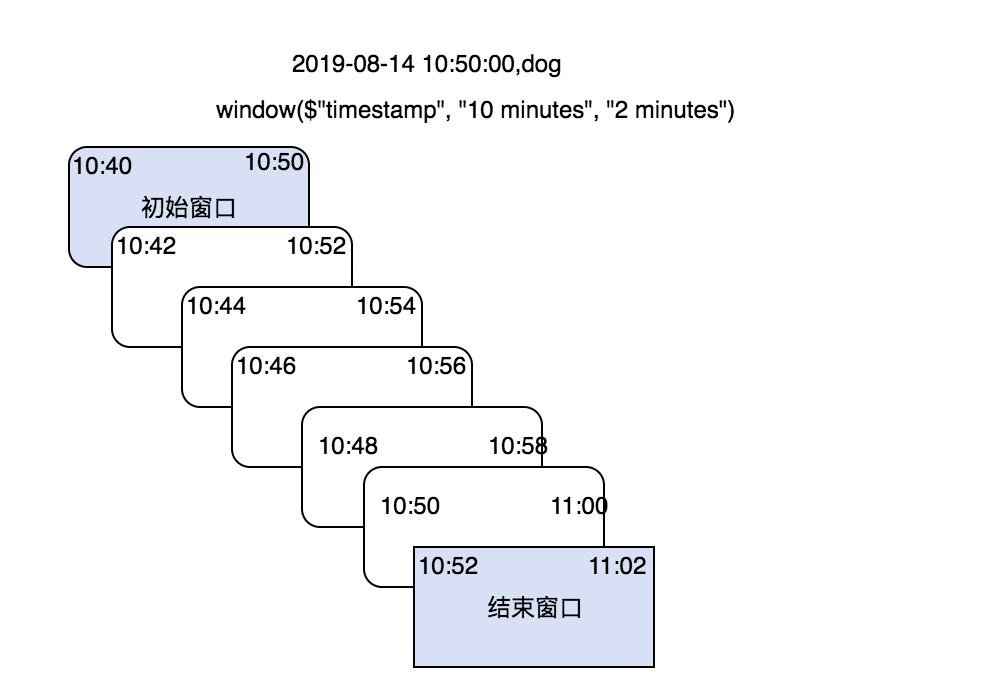
由此可以看出, 在这种窗口机制下, 无论事件何时到达, 以怎样的顺序到达, Structured Streaming 总会根据事件时间生成对应的若干个时间窗口, 然后按照指定的规则聚合.

#### 5.2.2 event-time 窗口生成规则

org.apache.spark.sql.catalyst.analysis.TimeWindowing

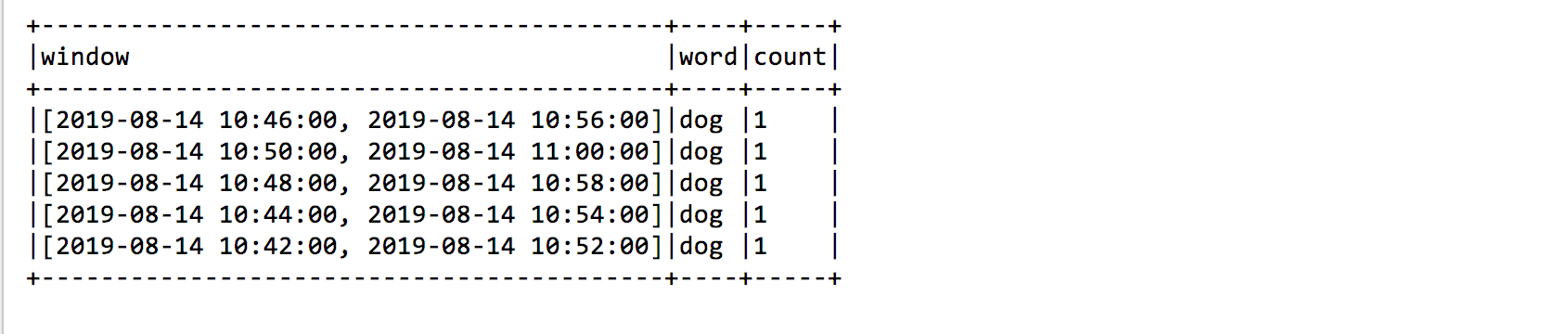
// 窗口个数  
maxNumOverlapping = ceil(windowDuration / slideDuration)  
for (i <- 0 until maxNumOverlapping)  
 windowId <- ceil((timestamp - startTime) / slideDuration)  
 windowStart <- windowId \* slideDuration + (i - maxNumOverlapping) \* slideDuration + startTime  
 windowEnd <- windowStart + windowDuration  
 return windowStart, windowEnd

将event-time 作为“初始窗口”的结束时间, 然后按照窗口滑动宽度逐渐向时间轴前方推进, 直到某个窗口不再包含该 event-time 为止. 最终以“初始窗口”与“结束窗口”之间的若干个窗口作为最终生成的 event-time 的时间窗口.



每个窗口的起始时间与结束时间都是前必后开的区间, 因此初始窗口和结束窗口都不会包含 event-time, 最终不会被使用.

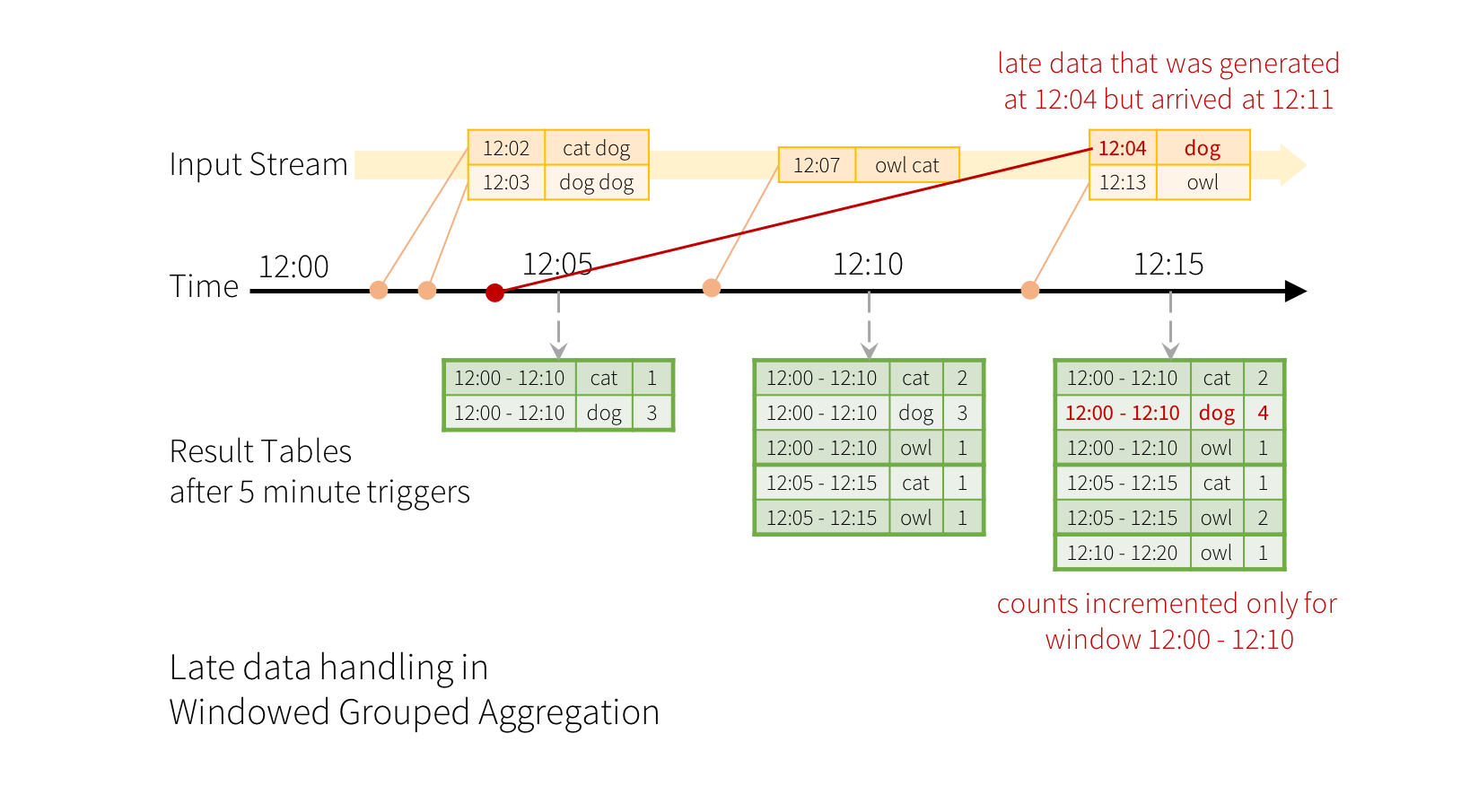
得到窗口如下:



### 5.3 基于 Watermark 处理延迟数据

在数据分析系统中, Structured Streaming 可以持续的按照 event-time 聚合数据, 然而在此过程中并不能保证数据按照时间的先后依次到达. 例如: 当前接收的某一条数据的 event-time 可能远远早于之前已经处理过的 event-time. 在发生这种情况时, 往往需要结合业务需求对延迟数据进行过滤.

现在考虑如果事件延迟到达会有哪些影响. 假如, 一个单词在 12:04(event-time) 产生, 在 12:11 到达应用. 应用应该使用 12:04 来在窗口(12:00 - 12:10)中更新计数, 而不是使用 12:11. 这些情况在我们基于窗口的聚合中是自然发生的, 因为结构化流可以长时间维持部分聚合的中间状态



但是, 如果这个查询运行数天, 系统很有必要限制内存中累积的中间状态的数量. 这意味着系统需要知道何时从内存状态中删除旧聚合, 因为应用不再接受该聚合的后期数据.

为了实现这个需求, 从 spark2.1, 引入了 watermark(水印), 使用引擎可以自动的跟踪当前的事件时间, 并据此尝试删除旧状态.

通过指定 event-time 列和预估事件的延迟时间上限来定义一个查询的 watermark. 针对一个以时间 T 结束的窗口, 引擎会保留状态和允许延迟时间直到(max event time seen by the engine - late threshold > T). 换句话说, 延迟时间在上限内的被聚合, 延迟时间超出上限的开始被丢弃.

可以通过withWatermark() 来定义watermark

watermark 计算: watermark = MaxEventTime - Threshhod

而且, watermark只能逐渐增加, 不能减少

总结:

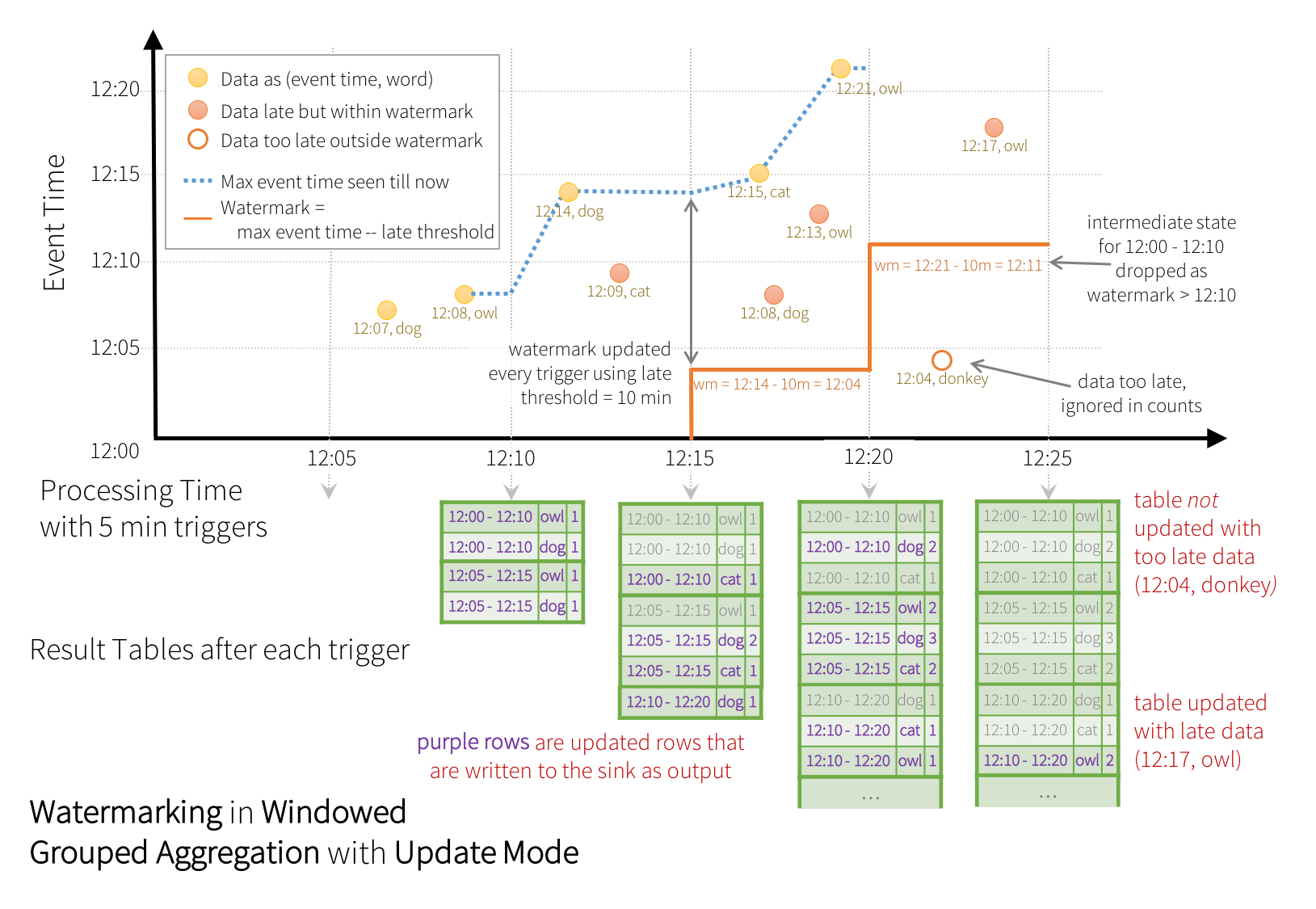
Structured Streaming 引入 Watermark 机制, 主要是为了解决以下两个问题:

1. 处理聚合中的延迟数据
2. 减少内存中维护的聚合状态.

在不同输出模式(complete, append, update)中, Watermark 会产生不同的影响.

#### 5.3.1 update 模式下使用 watermark

在 update 模式下, 仅输出与之前批次的结果相比, 涉及更新或新增的数据.



package com.atguigu.ss  
  
import java.sql.Timestamp  
  
import org.apache.spark.sql.\_  
import org.apache.spark.sql.streaming.{StreamingQuery, Trigger}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/13 4:44 PM  
 \*/  
object WordCountWatermark1 {  
 def main(args: Array[String]): Unit = {  
   
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("WordCountWatermark1")  
 .getOrCreate()  
   
 import spark.implicits.\_  
 val lines: DataFrame = spark.readStream  
 .format("socket")  
 .option("host", "localhost")  
 .option("port", 10000)  
 .load  
   
 // 输入的数据中包含时间戳, 而不是自动添加的时间戳  
 val words: DataFrame = lines.as[String].flatMap(line => {  
 val split = line.split(",")  
 split(1).split(" ").map((\_, Timestamp.valueOf(split(0))))  
 }).toDF("word", "timestamp")  
   
 import org.apache.spark.sql.functions.\_  
   
   
 val wordCounts: Dataset[Row] = words  
 // 添加watermark, 参数 1: event-time 所在列的列名 参数 2: 延迟时间的上限.  
 .withWatermark("timestamp", "2 minutes")  
 .groupBy(window($"timestamp", "10 minutes", "2 minutes"), $"word")  
 .count()  
   
 val query: StreamingQuery = wordCounts.writeStream  
 .outputMode("update")  
 .trigger(Trigger.ProcessingTime(1000))  
 .format("console")  
 .option("truncate", "false")  
 .start  
 query.awaitTermination()  
 }  
}

注意: 初始化wartmark 是 0

有以下几条数据:

测试:

1. 输入数据:2019-08-14 10:55:00,dog

这个条数据作为第一批数据. 按照window($"timestamp", "10 minutes", "2 minutes")得到 5 个窗口. 由于是第一批, 所有的窗口的结束时间都大于 wartermark(0), 所以 5 个窗口都显示.

+------------------------------------------+----+-----+  
|window |word|count|  
+------------------------------------------+----+-----+  
|[2019-08-14 10:46:00, 2019-08-14 10:56:00]|dog |1 |  
|[2019-08-14 10:52:00, 2019-08-14 11:02:00]|dog |1 |  
|[2019-08-14 10:50:00, 2019-08-14 11:00:00]|dog |1 |  
|[2019-08-14 10:48:00, 2019-08-14 10:58:00]|dog |1 |  
|[2019-08-14 10:54:00, 2019-08-14 11:04:00]|dog |1 |  
+------------------------------------------+----+-----+

然后根据当前批次中最大的 event-time, 计算出来下次使用的 watermark. 本批次只有一个数据(10:55), 所有: watermark = 10:55 - 2min = 10:53 2. 输入数据:2019-08-14 11:00:00,dog

这条数据作为第二批数据, 计算得到 5 个窗口. 此时的watermark=10:53, 所有的窗口的结束时间均大于 watermark. 在 update 模式下, 只输出结果表中涉及更新或新增的数据.

+------------------------------------------+----+-----+  
|window |word|count|  
+------------------------------------------+----+-----+  
|[2019-08-14 11:00:00, 2019-08-14 11:10:00]|dog |1 |  
|[2019-08-14 10:52:00, 2019-08-14 11:02:00]|dog |2 |  
|[2019-08-14 10:58:00, 2019-08-14 11:08:00]|dog |1 |  
|[2019-08-14 10:54:00, 2019-08-14 11:04:00]|dog |2 |  
|[2019-08-14 10:56:00, 2019-08-14 11:06:00]|dog |1 |  
+------------------------------------------+----+-----+

其中: count 是 2 的表示更新, count 是 1 的表示新增. 没有变化的就没有显示.(但是内存中仍然保存着)

// 第一批次中的数据仍然在内存保存着  
|[2019-08-14 10:46:00, 2019-08-14 10:56:00]|dog |1 |  
|[2019-08-14 10:50:00, 2019-08-14 11:00:00]|dog |1 |  
|[2019-08-14 10:48:00, 2019-08-14 10:58:00]|dog |1 |

此时的的 watermark = 11:00 - 2min = 10:58 3. 输入数据:2019-08-14 10:55:00,dog

相当于一条延迟数据.

这条数据作为第 3 批次, 计算得到 5 个窗口. 此时的 watermark = 10:58 当前内存中有两个窗口的结束时间已经低于 10: 58.

|[2019-08-14 10:48:00, 2019-08-14 10:58:00]|dog |1 |  
|[2019-08-14 10:46:00, 2019-08-14 10:56:00]|dog |1 |

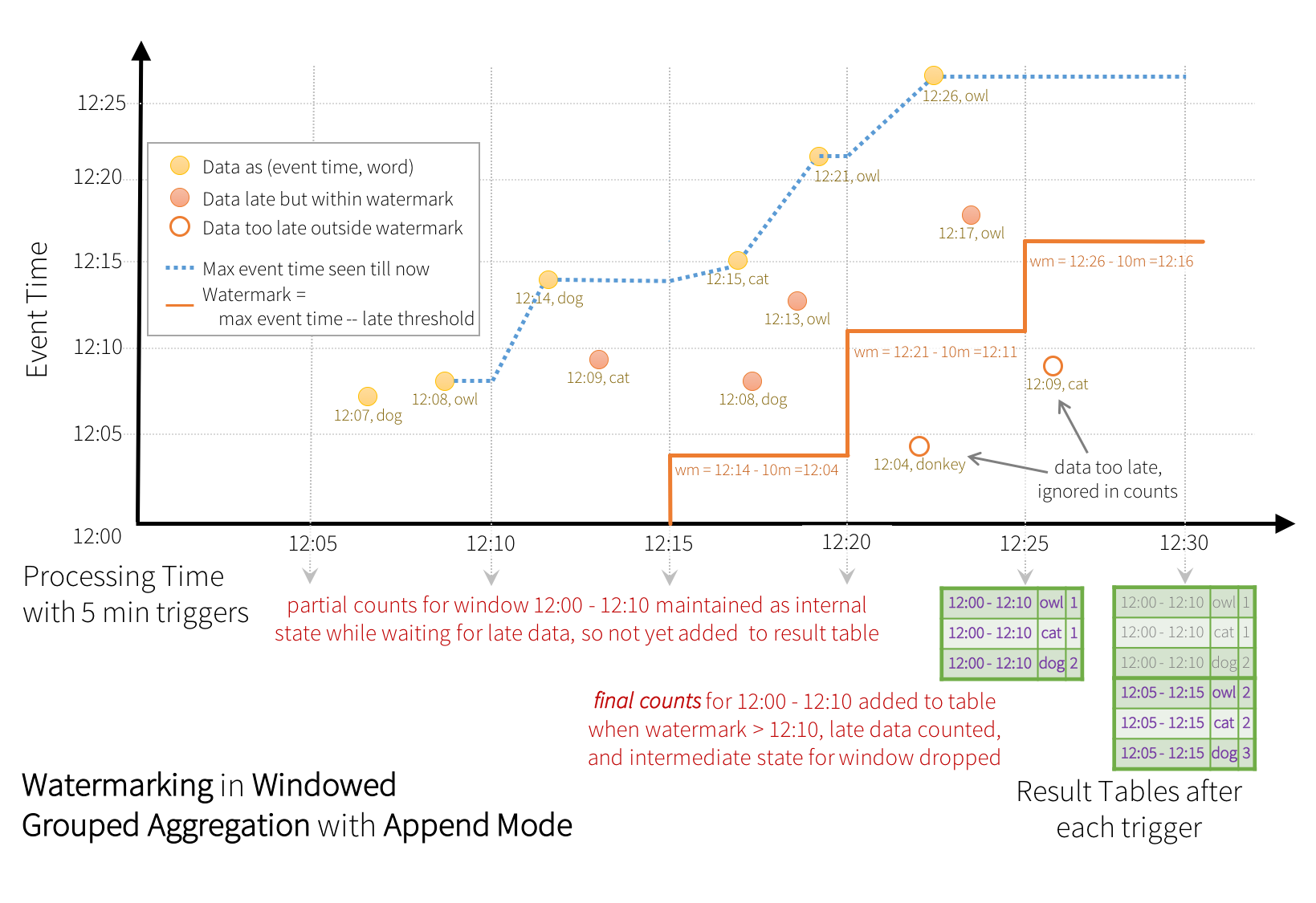
则立即删除这两个窗口在内存中的维护状态. 同时, 当前批次中新加入的数据所划分出来的窗口, 如果窗口结束时间低于 11:58, 则窗口会被过滤掉.

所以这次输出结果:

+------------------------------------------+----+-----+  
|window |word|count|  
+------------------------------------------+----+-----+  
|[2019-08-14 10:52:00, 2019-08-14 11:02:00]|dog |3 |  
|[2019-08-14 10:50:00, 2019-08-14 11:00:00]|dog |2 |  
|[2019-08-14 10:54:00, 2019-08-14 11:04:00]|dog |3 |  
+------------------------------------------+----+-----+

第三个批次的数据处理完成后, 立即计算: watermark= 10:55 - 2min = 10:53, 这个值小于当前的 watermask(10:58), 所以保持不变.(因为 watermask 只能增加不能减少)

#### 5.3.2 append 模式下使用 wartermark



把前一个案例中的update改成append即可.

val query: StreamingQuery = wordCounts.writeStream  
 .outputMode("append")  
 .trigger(Trigger.ProcessingTime(0))  
 .format("console")  
 .option("truncate", "false")  
 .start

在 append 模式中, 仅输出新增的数据, 且输出后的数据无法变更.

测试:

1. 输入数据:2019-08-14 10:55:00,dog

这个条数据作为第一批数据. 按照window($"timestamp", "10 minutes", "2 minutes")得到 5 个窗口. 由于此时初始 watermask=0, 当前批次中所有窗口的结束时间均大于 watermask.

但是 Structured Streaming 无法确定后续批次的数据中是否会更新当前批次的内容. 因此, 基于 Append 模式的特点, 这时并不会输出任何数据(因为输出后数据就无法更改了), 直到某个窗口的结束时间小于 watermask, 即可以确定后续数据不会再变更该窗口的聚合结果时才会将其输出, 并移除内存中对应窗口的聚合状态.

+------+----+-----+  
|window|word|count|  
+------+----+-----+  
+------+----+-----+

然后根据当前批次中最大的 event-time, 计算出来下次使用的 watermark. 本批次只有一个数据(10:55), 所有: watermark = 10:55 - 2min = 10:53 2. 输入数据:2019-08-14 11:00:00,dog

这条数据作为第二批数据, 计算得到 5 个窗口. 此时的watermark=10:53, 所有的窗口的结束时间均大于 watermark, 仍然不会输出.

+------+----+-----+  
|window|word|count|  
+------+----+-----+  
+------+----+-----+

然后计算 watermark = 11:00 - 2min = 10:58 3. 输入数据:2019-08-14 10:55:00,dog

相当于一条延迟数据.

这条数据作为第 3 批次, 计算得到 5 个窗口. 此时的 watermark = 10:58 当前内存中有两个窗口的结束时间已经低于 10: 58.

|[2019-08-14 10:48:00, 2019-08-14 10:58:00]|dog |1 |  
|[2019-08-14 10:46:00, 2019-08-14 10:56:00]|dog |1 |

则意味着这两个窗口的数据不会再发生变化, 此时输出这个两个窗口的聚合结果, 并在内存中清除这两个窗口的状态.

所以这次输出结果:

+------------------------------------------+----+-----+  
|window |word|count|  
+------------------------------------------+----+-----+  
|[2019-08-14 10:46:00, 2019-08-14 10:56:00]|dog |1 |  
|[2019-08-14 10:48:00, 2019-08-14 10:58:00]|dog |1 |  
+------------------------------------------+----+-----+

第三个批次的数据处理完成后, 立即计算: watermark= 10:55 - 2min = 10:53, 这个值小于当前的 watermask(10:58), 所以保持不变.(因为 watermask 只能增加不能减少)

#### 5.3.3 watermark 机制总结

1. watermark 在用于基于时间的状态聚合操作时, 该时间可以基于窗口, 也可以基于 event-timeb本身.
2. 输出模式必须是append或update. 在输出模式是complete的时候(必须有聚合), 要求每次输出所有的聚合结果. 我们使用 watermark 的目的是丢弃一些过时聚合数据, 所以complete模式使用wartermark无效也无意义.
3. 在输出模式是append时, 必须设置 watermask 才能使用聚合操作. 其实, watermask 定义了 append 模式中何时输出聚合聚合结果(状态), 并清理过期状态.
4. 在输出模式是update时, watermask 主要用于过滤过期数据并及时清理过期状态.
5. watermask 会在处理当前批次数据时更新, 并且会在处理下一个批次数据时生效使用. 但如果节点发送故障, 则可能延迟若干批次生效.
6. withWatermark 必须使用与聚合操作中的时间戳列是同一列.df.withWatermark("time", "1 min").groupBy("time2").count() 无效
7. withWatermark 必须在聚合之前调用. f.groupBy("time").count().withWatermark("time", "1 min") 无效

### 5.4 流数据去重

根据唯一的 id 实现数据去重.

数据:

1,2019-09-14 11:50:00,dog  
2,2019-09-14 11:51:00,dog  
1,2019-09-14 11:50:00,dog  
3,2019-09-14 11:53:00,dog  
1,2019-09-14 11:50:00,dog  
4,2019-09-14 11:45:00,dog

package com.atguigu.ss  
  
import java.sql.Timestamp  
  
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 5:52 PM  
 \*/  
object StreamDropDuplicate {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("Test")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val lines: DataFrame = spark.readStream  
 .format("socket")  
 .option("host", "localhost")  
 .option("port", 10000)  
 .load()  
   
 val words: DataFrame = lines.as[String].map(line => {  
 val arr: Array[String] = line.split(",")  
 (arr(0), Timestamp.valueOf(arr(1)), arr(2))  
 }).toDF("uid", "ts", "word")  
   
 val wordCounts: Dataset[Row] = words  
 .withWatermark("ts", "2 minutes")  
 .dropDuplicates("uid") // 去重重复数据 uid 相同就是重复. 可以传递多个列  
   
 wordCounts.writeStream  
 .outputMode("append")  
 .format("console")  
 .start  
 .awaitTermination()  
   
 }  
}

注意: 1. dropDuplicates 不可用在聚合之后, 即通过聚合得到的 df/ds 不能调用dropDuplicates 2. 使用watermask - 如果重复记录的到达时间有上限，则可以在事件时间列上定义水印，并使用guid和事件时间列进行重复数据删除。该查询将使用水印从过去的记录中删除旧的状态数据，这些记录不会再被重复。这限制了查询必须维护的状态量。 3. 没有watermask - 由于重复记录可能到达时没有界限，查询将来自所有过去记录的数据存储为状态。

测试

1. 第一批: 1,2019-09-14 11:50:00,dog

+---+-------------------+----+  
|uid| ts|word|  
+---+-------------------+----+  
| 1|2019-09-14 11:50:00| dog|  
+---+-------------------+----+

1. 第 2 批: 2,2019-09-14 11:51:00,dog

+---+-------------------+----+  
|uid| ts|word|  
+---+-------------------+----+  
| 2|2019-09-14 11:51:00| dog|  
+---+-------------------+----+

1. 第 3 批: 1,2019-09-14 11:50:00,dog  
   id 重复无输出
2. 第 4 批: 3,2019-09-14 11:53:00,dog

+---+-------------------+----+  
|uid| ts|word|  
+---+-------------------+----+  
| 3|2019-09-14 11:53:00| dog|  
+---+-------------------+----+

此时 watermask=11:51 5. 第 5 批: 1,2019-09-14 11:50:00,dog 数据重复, 并且数据过期, 所以无输出 6. 第 6 批 4,2019-09-14 11:45:00,dog 数据过时, 所以无输出

### 5.5 join 操作

Structured Streaming 支持 streaming DataSet/DataFrame 与静态的DataSet/DataFrame 进行 join, 也支持 streaming DataSet/DataFrame与另外一个streaming DataSet/DataFrame 进行 join.

join 的结果也是持续不断的生成, 类似于前面学习的 streaming 的聚合结果.

#### 5.5.1 Stream-static Joins

模拟的静态数据:

lisi,male  
zhiling,female  
zs,male

模拟的流式数据:

lisi,20  
zhiling,40  
ww,30

##### 5.5.1.1 内连接

package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.Trigger  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 4:41 PM  
 \*/  
object StreamingStatic {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("StreamingStatic")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 // 1. 静态 df  
 val arr = Array(("lisi", "male"), ("zhiling", "female"), ("zs", "male"));  
 var staticDF: DataFrame = spark.sparkContext.parallelize(arr).toDF("name", "sex")  
   
 // 2. 流式 df  
 val lines: DataFrame = spark.readStream  
 .format("socket")  
 .option("host", "localhost")  
 .option("port", 10000)  
 .load()  
 val streamDF: DataFrame = lines.as[String].map(line => {  
 val arr = line.split(",")  
 (arr(0), arr(1).toInt)  
 }).toDF("name", "age")  
   
 // 3. join 等值内连接 a.name=b.name  
 val joinResult: DataFrame = streamDF.join(staticDF, "name")  
   
 // 4. 输出  
 joinResult.writeStream  
 .outputMode("append")  
 .format("console")  
 .start  
 .awaitTermination()  
   
 }  
}

+-------+---+------+  
| name|age| sex|  
+-------+---+------+  
|zhiling| 40|female|  
| lisi| 20| male|  
+-------+---+------+

##### 5.5.1.2 外连接

val joinResult: DataFrame = streamDF.join(staticDF, Seq("name"), "left")

+-------+---+------+  
| name|age| sex|  
+-------+---+------+  
|zhiling| 40|female|  
| ww| 30| null|  
| lisi| 20| male|  
+-------+---+------+

#### 5.5.2 Stream-stream Joins

在 Spark2.3, 开始支持 stream-stream join.

Spark 会自动维护两个流的状态, 以保障后续流入的数据能够和之前流入的数据发生 join 操作, 但这会导致状态无限增长. 因此, 在对两个流进行 join 操作时, 依然可以用 watermark 机制来消除过期的状态, 避免状态无限增长.

##### 5.5.2.1 inner join

对 2 个流式数据进行 join 操作. 输出模式仅支持append模式

第 1 个数据格式: 姓名,年龄,事件时间

lisi,female,2019-09-16 11:50:00  
zs,male,2019-09-16 11:51:00  
ww,female,2019-09-16 11:52:00  
zhiling,female,2019-09-16 11:53:00  
fengjie,female,2019-09-16 11:54:00  
yifei,female,2019-09-16 11:55:00

第 2 个数据格式: 姓名,性别,事件时间

lisi,18,2019-09-16 11:50:00  
zs,19,2019-09-16 11:51:00  
ww,20,2019-09-16 11:52:00  
zhiling,22,2019-09-16 11:53:00  
yifei,30,2019-09-16 11:54:00  
fengjie,98,2019-09-16 11:55:00

###### 不带 watermast 的 inner join

package com.atguigu.ss  
  
import java.sql.Timestamp  
  
import org.apache.spark.sql.streaming.Trigger  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/16 5:09 PM  
 \*/  
object StreamStream1 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("StreamStream1")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 // 第 1 个 stream  
 val nameSexStream: DataFrame = spark.readStream  
 .format("socket")  
 .option("host", "hadoop201")  
 .option("port", 10000)  
 .load  
 .as[String]  
 .map(line => {  
 val arr: Array[String] = line.split(",")  
 (arr(0), arr(1), Timestamp.valueOf(arr(2)))  
 }).toDF("name", "sex", "ts1")  
   
 // 第 2 个 stream  
 val nameAgeStream: DataFrame = spark.readStream  
 .format("socket")  
 .option("host", "hadoop201")  
 .option("port", 20000)  
 .load  
 .as[String]  
 .map(line => {  
 val arr: Array[String] = line.split(",")  
 (arr(0), arr(1).toInt, Timestamp.valueOf(arr(2)))  
 }).toDF("name", "age", "ts2")  
   
   
 // join 操作  
 val joinResult: DataFrame = nameSexStream.join(nameAgeStream, "name")  
   
 joinResult.writeStream  
 .outputMode("append")  
 .format("console")  
 .trigger(Trigger.ProcessingTime(0))  
 .start()  
 .awaitTermination()  
 }  
}

+-------+------+-------------------+---+-------------------+  
| name| sex| ts1|age| ts2|  
+-------+------+-------------------+---+-------------------+  
|zhiling|female|2019-09-16 11:53:00| 22|2019-09-16 11:53:00|  
| ww|female|2019-09-16 11:52:00| 20|2019-09-16 11:52:00|  
| yifei|female|2019-09-16 11:55:00| 30|2019-09-16 11:54:00|  
| zs| male|2019-09-16 11:51:00| 19|2019-09-16 11:51:00|  
|fengjie|female|2019-09-16 11:54:00| 98|2019-09-16 11:55:00|  
| lisi|female|2019-09-16 11:50:00| 18|2019-09-16 11:50:00|  
+-------+------+-------------------+---+-------------------+

join 的速度很慢, 需要等待.

###### 带 watermast 的 inner join

package com.atguigu.ss  
  
import java.sql.Timestamp  
  
import org.apache.spark.sql.streaming.Trigger  
import org.apache.spark.sql.{DataFrame, SparkSession}  
import org.apache.spark.sql.functions.\_  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/16 5:09 PM  
 \*/  
object StreamStream2 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[\*]")  
 .appName("StreamStream1")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 // 第 1 个 stream  
 val nameSexStream: DataFrame = spark.readStream  
 .format("socket")  
 .option("host", "hadoop201")  
 .option("port", 10000)  
 .load  
 .as[String]  
 .map(line => {  
 val arr: Array[String] = line.split(",")  
 (arr(0), arr(1), Timestamp.valueOf(arr(2)))  
 }).toDF("name1", "sex", "ts1")  
 .withWatermark("ts1", "2 minutes")  
   
 // 第 2 个 stream  
 val nameAgeStream: DataFrame = spark.readStream  
 .format("socket")  
 .option("host", "hadoop201")  
 .option("port", 20000)  
 .load  
 .as[String]  
 .map(line => {  
 val arr: Array[String] = line.split(",")  
 (arr(0), arr(1).toInt, Timestamp.valueOf(arr(2)))  
 }).toDF("name2", "age", "ts2")  
 .withWatermark("ts2", "1 minutes")  
   
   
 // join 操作  
 val joinResult: DataFrame = nameSexStream.join(  
 nameAgeStream,  
 expr(  
 """  
 |name1=name2 and  
 |ts2 >= ts1 and  
 |ts2 <= ts1 + interval 1 minutes  
 """.stripMargin))  
   
 joinResult.writeStream  
 .outputMode("append")  
 .format("console")  
 .trigger(Trigger.ProcessingTime(0))  
 .start()  
 .awaitTermination()  
 }  
}

+-------+------+-------------------+-------+---+-------------------+  
| name1| sex| ts1| name2|age| ts2|  
+-------+------+-------------------+-------+---+-------------------+  
|zhiling|female|2019-09-16 11:53:00|zhiling| 22|2019-09-16 11:53:00|  
| ww|female|2019-09-16 11:52:00| ww| 20|2019-09-16 11:52:00|  
| zs| male|2019-09-16 11:51:00| zs| 19|2019-09-16 11:51:00|  
|fengjie|female|2019-09-16 11:54:00|fengjie| 98|2019-09-16 11:55:00|  
| lisi|female|2019-09-16 11:50:00| lisi| 18|2019-09-16 11:50:00|  
+-------+------+-------------------+-------+---+-------------------+

##### 5.5.2.2 outer join

外连接必须使用 watermast

和你连接相比, 代码几乎一致, 只需要在连接的时候指定下连接类型即可:joinType = "left\_join"

val joinResult: DataFrame = nameSexStream.join(  
 nameAgeStream,  
 expr(  
 """  
 |name1=name2 and  
 |ts2 >= ts1 and  
 |ts2 <= ts1 + interval 1 minutes  
 """.stripMargin),  
 joinType = "left\_join")

### 5.6 Streaming DF/DS 不支持的操作

到目前, DF/DS 的有些操作 Streaming DF/DS 还不支持.

1. 多个Streaming 聚合(例如在 DF 上的聚合链)目前还不支持
2. limit 和取前 N 行还不支持
3. distinct 也不支持
4. 仅仅支持对 complete 模式下的聚合操作进行排序操作
5. 仅支持有限的外连接
6. 有些方法不能直接用于查询和返回结果, 因为他们用在流式数据上没有意义.

* count() 不能返回单行数据, 必须是s.groupBy().count()
* foreach() 不能直接使用, 而是使用: ds.writeStream.foreach(...)
* show() 不能直接使用, 而是使用 console sink

如果执行上面操作会看到这样的异常: operation XYZ is not supported with streaming DataFrames/Datasets.

## 第 6 章 输出分析结果

一旦定义了最终结果DataFrame / Dataset，剩下的就是开始流式计算。为此，必须使用返回的 DataStreamWriter Dataset.writeStream()。

需要指定一下选项:

1. 输出接收器的详细信息：数据格式，位置等。
2. 输出模式：指定写入输出接收器的内容。
3. 查询名称：可选，指定查询的唯一名称以进行标识。
4. 触发间隔：可选择指定触发间隔。如果未指定，则系统将在前一处理完成后立即检查新数据的可用性。如果由于先前的处理尚未完成而错过了触发时间，则系统将立即触发处理。
5. 检查点位置：对于可以保证端到端容错的某些输出接收器，请指定系统写入所有检查点信息的位置。这应该是与HDFS兼容的容错文件系统中的目录。

### 6.1 输出模式(output mode)

#### 6.1.1 Append 模式(默认)

默认输出模式, 仅仅添加到结果表的新行才会输出.

采用这种输出模式, 可以保证每行数据仅输出一次.

在查询过程中, 如果没有使用 watermask 机制, 则不能使用聚合操作. 如果使用了 watermask 机制, 则只能使用基于 event-time 的聚合操作.

watermask 用于高速 append 模式如何输出不会再发生变动的数据. 即只有过期的聚合结果才会在 Append 模式中被“有且仅有一次”的输出.

#### 6.1.2 Complete 模式

每次触发, 整个结果表的数据都会被输出. 仅仅聚合操作才支持.

同时该模式使用 watermask 无效.

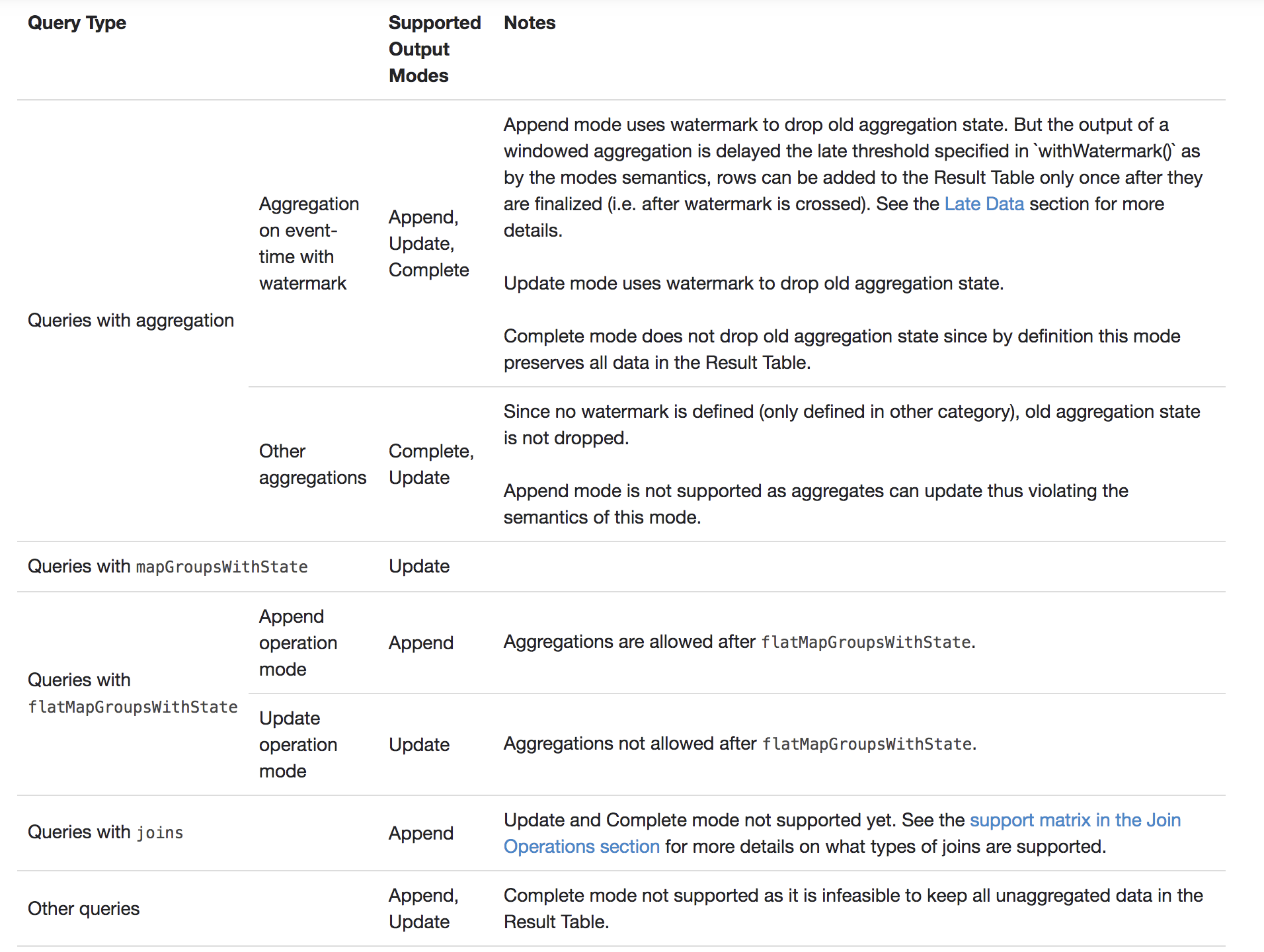
#### 6.1.3 Update 模式

该模式在 从 spark 2.1.1 可用. 在处理完数据之后, 该模式只输出相比上个批次变动的内容(新增或修改).

如果没有聚合操作, 则该模式与 append 模式一直. 如果有聚合操作, 则可以基于 watermast 清理过期的状态.

#### 6.1.4 输出模式总结

不同的查询支持不同的输出模式



### 6.2 输出接收器(output sink)

spark 提供了几个内置的 output-sink

不同 output sink 所适用的 output mode 不尽相同

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sink | Supported Output Modes | Options | Fault-tolerant | Notes |
| **File Sink** | Append | path: path to the output directory, must be specified. For file-format-specific options, see the related methods in DataFrameWriter ([Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameWriter)/[Java](http://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/DataFrameWriter.html)/[Python](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameWriter)/[R](http://spark.apache.org/docs/latest/api/R/write.stream.html)). E.g. for “parquet” format options see DataFrameWriter.parquet() | Yes (exactly-once) | Supports writes to partitioned tables. Partitioning by time may be useful. |
| **Kafka Sink** | Append, Update, Complete | See the [Kafka Integration Guide](http://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html) | Yes (at-least-once) | More details in the [Kafka Integration Guide](http://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html) |
| **Foreach Sink** | Append, Update, Complete | None | Depends on ForeachWriter implementation | More details in the [next section](http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#using-foreach-and-foreachbatch) |
| **ForeachBatch Sink** | Append, Update, Complete | None | Depends on the implementation | More details in the [next section](http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#using-foreach-and-foreachbatch) |
| **Console Sink** | Append, Update, Complete | numRows: Number of rows to print every trigger (default: 20) truncate: Whether to truncate the output if too long (default: true) | No |  |
| **Memory Sink** | Append, Complete | None | No. But in Complete Mode, restarted query will recreate the full table. | Table name is the query name. |

#### 6.2.1 file sink

存储输出到目录中 仅仅支持 append 模式

需求: 把单词和单词的反转组成 json 格式写入到目录中.

package com.atguigu.ss  
  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 7:39 PM  
 \*/  
object FileSink {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[1]")  
 .appName("Test")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val lines: DataFrame = spark.readStream  
 .format("socket") // 设置数据源  
 .option("host", "localhost")  
 .option("port", 10000)  
 .load  
   
 val words: DataFrame = lines.as[String].flatMap(line => {  
 line.split("\\W+").map(word => {  
 (word, word.reverse)  
 })  
 }).toDF("原单词", "反转单词")  
   
 words.writeStream  
 .outputMode("append")  
 .format("json") // // 支持 "orc", "json", "csv"  
 .option("path", "./filesink") // 输出目录  
 .option("checkpointLocation", "./ck1") // 必须指定 checkpoint 目录  
 .start  
 .awaitTermination()  
 }  
}

#### 6.2.2 kafka sink

将 wordcount 结果写入到 kafka

写入到 kafka 的时候应该包含如下列:

|  |  |
| --- | --- |
| Column | Type |
| key (optional) | string or binary |
| value (required) | string or binary |
| topic (optional) | string |

注意:

1. 如果没有添加 topic option 则 topic 列必须有.
2. kafka sink 三种输出模式都支持

##### 6.2.2.1 以 Streaming 方式输出数据

这种方式使用流的方式源源不断的向 kafka 写入数据.

package com.atguigu.ss  
  
import org.apache.spark.sql.streaming.Trigger  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 7:39 PM  
 \*/  
object KafkaSink {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[1]")  
 .appName("Test")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val lines: DataFrame = spark.readStream  
 .format("socket") // 设置数据源  
 .option("host", "localhost")  
 .option("port", 10000)  
 .load  
   
 val words = lines.as[String]  
 .flatMap(\_.split("\\W+"))  
 .groupBy("value")  
 .count()  
 .map(row => row.getString(0) + "," + row.getLong(1))  
 .toDF("value") // 写入数据时候, 必须有一列 "value"  
   
 words.writeStream  
 .outputMode("update")  
 .format("kafka")  
 .trigger(Trigger.ProcessingTime(0))  
 .option("kafka.bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092") // kafka 配置  
 .option("topic", "update") // kafka 主题  
 .option("checkpointLocation", "./ck1") // 必须指定 checkpoint 目录  
 .start  
 .awaitTermination()  
 }  
}

##### 6.2.2.2 以 batch 方式输出数据

这种方式输出离线处理的结果, 将已存在的数据分为若干批次进行处理. 处理完毕后程序退出.

package com.atguigu.ss  
  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 7:39 PM  
 \*/  
object KafkaSink2 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[1]")  
 .appName("Test")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val wordCount: DataFrame = spark.sparkContext.parallelize(Array("hello hello atguigu", "atguigu, hello"))  
 .toDF("word")  
 .groupBy("word")  
 .count()  
 .map(row => row.getString(0) + "," + row.getLong(1))  
 .toDF("value") // 写入数据时候, 必须有一列 "value"  
   
 wordCount.write // batch 方式  
 .format("kafka")  
 .option("kafka.bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092") // kafka 配置  
 .option("topic", "update") // kafka 主题  
 .save()  
 }  
}

#### 6.2.3 console sink

该 sink 主要用于测试.

具体代码见前面

#### 6.2.4 memory sink

该 sink 也是用于测试, 将其统计结果全部输入内中指定的表中, 然后可以通过 sql 与从表中查询数据.

如果数据量非常大, 可能会发送内存溢出.

package com.atguigu.ss  
  
import java.util.{Timer, TimerTask}  
  
import org.apache.spark.sql.streaming.StreamingQuery  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 7:39 PM  
 \*/  
object MemorySink {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[2]")  
 .appName("MemorySink")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val lines: DataFrame = spark.readStream  
 .format("socket") // 设置数据源  
 .option("host", "localhost")  
 .option("port", 10000)  
 .load  
   
 val words: DataFrame = lines.as[String]  
 .flatMap(\_.split("\\W+"))  
 .groupBy("value")  
 .count()  
   
 val query: StreamingQuery = words.writeStream  
 .outputMode("complete")  
 .format("memory") // memory sink  
 .queryName("word\_count") // 内存临时表名  
 .start  
   
 // 测试使用定时器执行查询表  
 val timer = new Timer(true)  
 val task: TimerTask = new TimerTask {  
 override def run(): Unit = spark.sql("select \* from word\_count").show  
 }  
 timer.scheduleAtFixedRate(task, 0, 2000)  
   
 query.awaitTermination()  
   
 }  
}

#### 6.2.5 foreach sink

foreach sink 会遍历表中的每一行, 允许将流查询结果按开发者指定的逻辑输出.

把 wordcount 数据写入到 mysql

##### 步骤 1: 添加 mysql 驱动

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.27</version>  
</dependency>

##### 步骤 2: 在 mysql 中创建数据库和表

create database ss;  
use ss;  
create table word\_count(word varchar(255) primary key not null, count bigint not null);

##### 步骤 3: 实现代码

package com.atguigu.ss  
  
import java.sql.{Connection, DriverManager, PreparedStatement}  
  
import org.apache.spark.sql.streaming.StreamingQuery  
import org.apache.spark.sql.{DataFrame, ForeachWriter, Row, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 7:39 PM  
 \*/  
object ForeachSink {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[2]")  
 .appName("ForeachSink")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val lines: DataFrame = spark.readStream  
 .format("socket") // 设置数据源  
 .option("host", "hadoop201")  
 .option("port", 10000)  
 .load  
   
 val wordCount: DataFrame = lines.as[String]  
 .flatMap(\_.split("\\W+"))  
 .groupBy("value")  
 .count()  
   
 val query: StreamingQuery = wordCount.writeStream  
 .outputMode("update")  
 // 使用 foreach 的时候, 需要传递ForeachWriter实例, 三个抽象方法需要实现. 每个批次的所有分区都会创建 ForeeachWriter 实例  
 .foreach(new ForeachWriter[Row] {  
 var conn: Connection = \_  
 var ps: PreparedStatement = \_  
 var batchCount = 0  
   
 // 一般用于 打开链接. 返回 false 表示跳过该分区的数据,  
 override def open(partitionId: Long, epochId: Long): Boolean = {  
 println("open ..." + partitionId + " " + epochId)  
 Class.forName("com.mysql.jdbc.Driver")  
 conn = DriverManager.getConnection("jdbc:mysql://hadoop201:3306/ss", "root", "aaa")  
 // 插入数据, 当有重复的 key 的时候更新  
 val sql = "insert into word\_count values(?, ?) on duplicate key update word=?, count=?"  
 ps = conn.prepareStatement(sql)  
   
 conn != null && !conn.isClosed && ps != null  
 }  
   
 // 把数据写入到连接  
 override def process(value: Row): Unit = {  
 println("process ...." + value)  
 val word: String = value.getString(0)  
 val count: Long = value.getLong(1)  
 ps.setString(1, word)  
 ps.setLong(2, count)  
 ps.setString(3, word)  
 ps.setLong(4, count)  
 ps.execute()  
 }  
   
 // 用户关闭连接  
 override def close(errorOrNull: Throwable): Unit = {  
 println("close...")  
 ps.close()  
 conn.close()  
 }  
 })  
 .start  
   
 query.awaitTermination()  
   
 }  
}

##### 步骤 4: 测试

#### 6.2.6 ForeachBatch Sink

ForeachBatch Sink 是 spark 2.4 才新增的功能, 该功能只能用于输出批处理的数据.

将统计结果同时输出到本地文件和 mysql 中

package com.atguigu.ss  
  
import java.util.Properties  
  
import org.apache.spark.sql.streaming.StreamingQuery  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
/\*\*  
 \* Author lzc  
 \* Date 2019/8/14 7:39 PM  
 \*/  
object ForeachBatchSink {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[2]")  
 .appName("ForeachBatchSink")  
 .getOrCreate()  
 import spark.implicits.\_  
   
 val lines: DataFrame = spark.readStream  
 .format("socket") // 设置数据源  
 .option("host", "hadoop201")  
 .option("port", 10000)  
 .load  
   
 val wordCount: DataFrame = lines.as[String]  
 .flatMap(\_.split("\\W+"))  
 .groupBy("value")  
 .count()  
   
 val props = new Properties()  
 props.setProperty("user", "root")  
 props.setProperty("password", "aaa")  
 val query: StreamingQuery = wordCount.writeStream  
 .outputMode("complete")  
 .foreachBatch((df, batchId) => { // 当前分区id, 当前批次id  
 if (df.count() != 0) {  
 df.cache()  
 df.write.json(s"./$batchId")  
 df.write.mode("overwrite").jdbc("jdbc:mysql://hadoop201:3306/ss", "word\_count", props)  
 }  
 })  
 .start()  
   
   
 query.awaitTermination()  
   
 }  
}

## 第 7 章 Trigger(触发器)

流式查询的触发器定义了流式数据处理的时间, 流式查询根据触发器的不同, 可以是根据固定的批处理间隔进行微批处理查询, 也可以是连续的查询.

|  |  |
| --- | --- |
| Trigger Type | Description |
| *unspecified (default)* | 没有显示的设定触发器, 表示使用 micro-batch mode, 尽可能块的处理每个批次的数据. 如果无数据可用, 则处于阻塞状态, 等待数据流入 |
| **Fixed interval micro-batches** 固定周期的微批处理 | 查询会在微批处理模式下执行, 其中微批处理将以用户指定的间隔执行. 1. 如果以前的微批处理在间隔内完成, 则引擎会等待间隔结束, 然后开启下一个微批次 2. 如果前一个微批处理在一个间隔内没有完成(即错过了间隔边界), 则下个微批处理会在上一个完成之后立即启动(不会等待下一个间隔边界) 3. 如果没有新数据可用, 则不会启动微批次. 适用于流式数据的批处理作业 |
| **One-time micro-batch** 一次性微批次 | 查询将在所有可用数据上执行一次微批次处理, 然后自行停止. 如果你希望定期启动集群, 然后处理集群关闭期间产生的数据, 然后再关闭集群. 这种情况下很有用. 它可以显著的降低成本. 一般用于非实时的数据分析 |
| **Continuous with fixed checkpoint interval** *(experimental 2.3 引入)* 连续处理 | 以超低延迟处理数据 |

// 1. 默认触发器  
val query: StreamingQuery = df.writeStream  
 .outputMode("append")  
 .format("console")  
 .start()  
// 2. 微批处理模式  
val query: StreamingQuery = df.writeStream  
 .outputMode("append")  
 .format("console")  
 .trigger(Trigger.ProcessingTime("2 seconds"))  
 .start  
  
// 3. 只处理一次. 处理完毕之后会自动退出  
val query: StreamingQuery = df.writeStream  
 .outputMode("append")  
 .format("console")  
 .trigger(Trigger.Once())  
 .start()  
   
// 4. 持续处理  
val query: StreamingQuery = df.writeStream  
 .outputMode("append")  
 .format("console")  
 .trigger(Trigger.Continuous("1 seconds"))  
 .start

### 7.1 连续处理模式(Continuous processing)

连续处理是2.3 引入, 它可以实现低至 1ms 的处理延迟. 并实现了至少一次(at-least-once)的语义.

微批处理模式虽然实现了严格一次(exactly-once)的语义, 但是最低有 100ms 的延迟.

对有些类型的查询, 可以切换到这个模式, 而不需要修改应用的逻辑.(不用更改 df/ds 操作)

若要切换到连续处理模式, 只需要更改触发器即可.

spark  
 .readStream  
 .format("rate")  
 .option("rowsPerSecond", "10")  
 .option("")  
  
spark  
 .readStream  
 .format("kafka")  
 .option("kafka.bootstrap.servers", "host1:port1,host2:port2")  
 .option("subscribe", "topic1")  
 .load()  
 .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")  
 .writeStream  
 .format("kafka")  
 .option("kafka.bootstrap.servers", "host1:port1,host2:port2")  
 .option("topic", "topic1")  
 .trigger(Trigger.Continuous("1 second")) // only change in query  
 .start()

#### 连续处理模式支持的查询

1. 操作: 支持 select, map, flatMap, mapPartitions, etc. 和 selections (where, filter, etc.). 不支持聚合操作
2. 数据源:

* kafka 所有选项都支持
* rate source

1. sink

* 所有的 kafka 参数都支持
* memory sink
* console sink

# 第 8 部分 Structured Streaming 项目实战

## 第 1 章 准备数据

本实战项目使用 Structured Streaming 来实时的分析处理用户对广告点击的行为数据.

### 1.1 数据生成方式

使用代码的方式持续的生成数据, 然后写入到 kafka 中.

然后Structured Streaming 负责从 kafka 消费数据, 并对数据根据需求进行分析.

### 1.2 数据生成模块

模拟出来的数据格式:

时间戳,地区,城市,用户 id,广告 id  
1566035129449,华南,深圳,101,2

#### 步骤1: 开启集群

启动 zookeeper 和 Kafka

#### 步骤2: 创建 Topic

在 kafka 中创建topic: ads\_log

#### 步骤3: 产生循环不断的数据到指定的 topic

创建模块spark-realtime模块

##### 1. 工具类: RandomNumUtil

用于生成随机数

package com.atguigu.realtime.util  
  
import java.util.Random  
  
import scala.collection.mutable  
  
/\*\*  
 \* Author lzc  
 \* Date 2019-07-17 14:07  
 \*  
 \* 随机生成整数的工具类  
 \*/  
object RandomNumUtil {  
 val random = new Random()  
   
 /\*\*  
 \* 返回一个随机的整数 [from, to]  
 \*  
 \* @param from  
 \* @param to  
 \* @return  
 \*/  
 def randomInt(from: Int, to: Int): Int = {  
 if (from > to) throw new IllegalArgumentException(s"from = $from 应该小于 to = $to")  
 // [0, to - from) + from [form, to -from + from ]  
 random.nextInt(to - from + 1) + from  
 }  
   
 /\*\*  
 \* 随机的Long [from, to]  
 \*  
 \* @param from  
 \* @param to  
 \* @return  
 \*/  
 def randomLong(from: Long, to: Long): Long = {  
 if (from > to) throw new IllegalArgumentException(s"from = $from 应该小于 to = $to")  
 random.nextLong().abs % (to - from + 1) + from  
 }  
   
 /\*\*  
 \* 生成一系列的随机值  
 \*  
 \* @param from  
 \* @param to  
 \* @param count  
 \* @param canReat 是否允许随机数重复  
 \*/  
 def randomMultiInt(from: Int, to: Int, count: Int, canReat: Boolean = true): List[Int] = {  
 if (canReat) {  
 (1 to count).map(\_ => randomInt(from, to)).toList  
 } else {  
 val set: mutable.Set[Int] = mutable.Set[Int]()  
 while (set.size < count) {  
 set += randomInt(from, to)  
 }  
 set.toList  
 }  
 }  
   
   
 def main(args: Array[String]): Unit = {  
 println(randomMultiInt(1, 15, 10))  
 println(randomMultiInt(1, 8, 10, false))  
 }  
}

##### 2. 工具类: RandomOptions

用于生成带有比重的随机选项

package com.atguigu.realtime.util  
  
import scala.collection.mutable.ListBuffer  
  
/\*\*  
 \* 根据提供的值和比重, 来创建RandomOptions对象.  
 \* 然后可以通过getRandomOption来获取一个随机的预定义的值  
 \*/  
object RandomOptions {  
 def apply[T](opts: (T, Int)\*): RandomOptions[T] = {  
 val randomOptions = new RandomOptions[T]()  
 randomOptions.totalWeight = (0 /: opts) (\_ + \_.\_2) // 计算出来总的比重  
 opts.foreach {  
 case (value, weight) => randomOptions.options ++= (1 to weight).map(\_ => value)  
 }  
 randomOptions  
 }  
   
   
 def main(args: Array[String]): Unit = {  
 // 测试  
 val opts = RandomOptions(("张三", 10), ("李四", 30), ("ww", 20))  
   
 println(opts.getRandomOption())  
 println(opts.getRandomOption())  
 }  
}  
  
// 工程师 10 程序猿 10 老师 20  
class RandomOptions[T] {  
 var totalWeight: Int = \_  
 var options = ListBuffer[T]()  
 /\*\*  
 \* 获取随机的 Option 的值  
 \*  
 \* @return  
 \*/  
 def getRandomOption() = {  
 options(RandomNumUtil.randomInt(0, totalWeight - 1))  
 }  
}

##### 3. 样例类: CityInfo

package com.atguigu.realtime  
  
  
/\*\*  
 \* 城市表  
 \*  
 \* @param city\_id 城市 id  
 \* @param city\_name 城市名  
 \* @param area 城市区域  
 \*/  
case class CityInfo(city\_id: Long,  
 city\_name: String,  
 area: String)

##### 4. 生成模拟数据: MockRealTime

package com.atguigu.realtime.mock  
  
import java.util.Properties  
  
import com.atguigu.realtime.CityInfo  
import com.atguigu.realtime.util.{RandomNumUtil, RandomOptions}  
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}  
  
import scala.collection.mutable.ArrayBuffer  
  
/\*\*  
 \* 生成实时的模拟数据  
 \*/  
object MockRealtime {  
 /\*  
 数据格式:  
 timestamp area city userid adid  
 某个时间点 某个地区 某个城市 某个用户 某个广告  
   
 \*/  
 def mockRealTimeData(): ArrayBuffer[String] = {  
 // 存储模拟的实时数据  
 val array = ArrayBuffer[String]()  
 // 城市信息  
 val randomOpts = RandomOptions(  
 (CityInfo(1, "北京", "华北"), 30),  
 (CityInfo(2, "上海", "华东"), 30),  
 (CityInfo(3, "广州", "华南"), 10),  
 (CityInfo(4, "深圳", "华南"), 20),  
 (CityInfo(4, "杭州", "华中"), 10))  
 (1 to 50).foreach {  
 i => {  
 val timestamp = System.currentTimeMillis()  
 val cityInfo = randomOpts.getRandomOption()  
 val area = cityInfo.area  
 val city = cityInfo.city\_name  
 val userid = RandomNumUtil.randomInt(100, 105)  
 val adid = RandomNumUtil.randomInt(1, 5)  
 array += s"$timestamp,$area,$city,$userid,$adid"  
 Thread.sleep(10)  
 }  
 }  
 array  
 }  
   
 def createKafkaProducer: KafkaProducer[String, String] = {  
 val props: Properties = new Properties  
 // Kafka服务端的主机名和端口号  
 props.put("bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092")  
 // key序列化  
 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")  
 // value序列化  
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")  
 new KafkaProducer[String, String](props)  
 }  
   
 def main(args: Array[String]): Unit = {  
 val topic = "ads\_log"  
 val producer: KafkaProducer[String, String] = createKafkaProducer  
 while (true) {  
 mockRealTimeData().foreach {  
 msg => {  
 producer.send(new ProducerRecord(topic, msg))  
 Thread.sleep(100)  
 }  
 }  
 Thread.sleep(1000)  
 }  
 }  
}

#### 步骤 4: 确认 kafka 中数据是否生成成功

### 1.3 从 Kafka 读取数据

编写RealTimeApp, 从 kafka 读取数据

#### 1.3.1 bean 类 AdsInfo

用来封装从 Kafka 读取到广告点击信息

package com.atguigu.realtime.bean  
  
import java.sql.Timestamp  
  
case class AdsInfo(ts: Long,   
 timestamp: Timestamp,  
 dayString: String,  
 hmString: String,  
 area: String,  
 city: String,  
 userId: String,  
 adsId: String)

#### 1.3.2 RealtimeApp

测试是否可以从 kafka 消费到数据:

package com.atguigu.realtime.app  
  
import java.sql.Timestamp  
import java.text.SimpleDateFormat  
import java.util.Date  
  
import com.atguigu.realtime.bean.AdsInfo  
import org.apache.spark.sql.\_  
  
  
/\*\*  
 \* Author lzc  
 \* Date 2019-08-17 17:52  
 \*/  
object RealtimeApp {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .master("local[2]")  
 .appName("RealtimeApp")  
 .getOrCreate()  
 spark.sparkContext.setLogLevel("WARN")  
 import spark.implicits.\_  
 val dayStringFormatter: SimpleDateFormat = new SimpleDateFormat("yyyy-MM-dd")  
 val hmStringFormatter: SimpleDateFormat = new SimpleDateFormat("HH:mm")  
 // 1. 从 kafka 读取数据, 为了方便后续处理, 封装数据到 AdsInfo 样例类中  
 val adsInfoDS: Dataset[AdsInfo] = spark.readStream  
 .format("kafka")  
 .option("kafka.bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092")  
 .option("subscribe", "ads\_log")  
 .load  
 .select("value")  
 .as[String]  
 .map(v => {  
 val split: Array[String] = v.split(",")  
 val date: Date = new Date(split(0).toLong)  
 AdsInfo(split(0).toLong, new Timestamp(split(0).toLong), dayStringFormatter.format(date), hmStringFormatter.format(date), split(1), split(2), split(3), split(4))  
 })  
 adsInfoDS.writeStream  
 .format("console")  
 .outputMode("update")  
 .option("truncate", "false")  
 .start  
 .awaitTermination()  
 }  
}

## 第 2 章 广告黑名单实时统计

### 2.1 需求简介

实现实时的动态黑名单检测机制：将每天对某个广告点击超过阈值(比如:100次)的用户拉入黑名单。 1. 黑名单应该是每天更新一次. 如果昨天进入黑名单, 今天应该重新再统计 2. 把黑名单写入到 redis 中, 以供其他应用查看 3. 已经进入黑名单的用户不再进行检测(提高效率)

### 2.2 思路

#### 2.2.1 写入到黑名单

黑名单存放在 redis 中, 使用 set, set 中的每个元素表示一个用户.

通过 sql 查询过滤出来每天每广告点击数超过阈值的用户, 然后使用 foreach 写入到 redis 即可.

#### 2.2.2 过滤黑名单的用户点击记录

先从 redis 读取到所有黑名单数据, 然后过滤, 只保留非黑名单用户的点击记录.

### 2.3 具体实现代码

#### 2.3.1 RedisUtil 工具类

package com.atguigu.realtime.util  
  
  
import redis.clients.jedis.{Jedis, JedisPool, JedisPoolConfig}  
  
object RedisUtil {  
 private val jedisPoolConfig: JedisPoolConfig = new JedisPoolConfig()  
 jedisPoolConfig.setMaxTotal(100) //最大连接数  
 jedisPoolConfig.setMaxIdle(20) //最大空闲  
 jedisPoolConfig.setMinIdle(20) //最小空闲  
 jedisPoolConfig.setBlockWhenExhausted(true) //忙碌时是否等待  
 jedisPoolConfig.setMaxWaitMillis(500) //忙碌时等待时长 毫秒  
 jedisPoolConfig.setTestOnBorrow(true) //每次获得连接的进行测试  
 private val jedisPool: JedisPool = new JedisPool(jedisPoolConfig, "hadoop201", 6379)  
   
 // 直接得到一个 Redis 的连接  
 def getJedisClient: Jedis = {  
 jedisPool.getResource  
 }  
}

#### 2.3.2 BlackListApp 类具体实现:

package com.atguigu.realtime.app  
  
import com.atguigu.realtime.bean.AdsInfo  
import com.atguigu.realtime.util.RedisUtil  
import org.apache.spark.sql.\_  
import org.apache.spark.sql.streaming.Trigger  
import redis.clients.jedis.Jedis  
  
/\*\*  
 \* Author lzc  
 \* Date 2019-08-19 09:37  
 \*  
 \* 需求1: 统计黑名单  
 \*  
 \* 其他需求直接使用过滤后的数据就可以了  
 \*/  
object BlackListApp {  
 def statBlackList(spark: SparkSession, adsInfoDS: Dataset[AdsInfo]): Dataset[AdsInfo] = {  
 import spark.implicits.\_  
   
 // 1. 过滤黑名单的数据: 如果有用户已经进入黑名单, 则不再统计这个用户的广告点击记录  
 val filteredAdsInfoDS: Dataset[AdsInfo] = adsInfoDS.mapPartitions(adsInfoIt => { // 每个分区连接一次到redis读取黑名单, 然后把进入黑名单用户点击记录过滤掉  
 val adsInfoList: List[AdsInfo] = adsInfoIt.toList  
 if (adsInfoList.isEmpty) {  
 adsInfoList.toIterator  
 } else {  
 // 1. 先读取到黑名单  
 val client: Jedis = RedisUtil.getJedisClient  
   
 val blackList: java.util.Set[String] = client.smembers(s"day:blcklist:${adsInfoList(0).dayString}")  
 // 2. 过滤  
 adsInfoList.filter(adsInfo => {  
 !blackList.contains(adsInfo.userId)  
 }).toIterator  
 }  
   
 })  
   
 // 创建临时表: tb\_ads\_info  
 filteredAdsInfoDS.createOrReplaceTempView("tb\_ads\_info")  
   
 // 需求1: 黑名单 每天每用户每广告的点击量  
 // 2. 按照每天每用户每id分组, 然后计数, 计数超过阈值(100)的查询出来  
 val result: DataFrame = spark.sql(  
 """  
 |select  
 | dayString,  
 | userId  
 |from tb\_ads\_info  
 |group by dayString, userId, adsId  
 |having count(1) >= 100000  
 """.stripMargin)  
   
 // 3. 把点击量超过 100 的写入到redis中.  
 result.writeStream  
 .outputMode("update")  
 .trigger(Trigger.ProcessingTime("2 seconds"))  
 .foreach(new ForeachWriter[Row] {  
 var client: Jedis = \_  
   
 override def open(partitionId: Long, epochId: Long): Boolean = {  
 // 打开到redis的连接  
 client = RedisUtil.getJedisClient  
 client != null  
 }  
   
 override def process(value: Row): Unit = {  
 // 写入到redis 把每天的黑名单写入到set中 key: "day:blacklist" value: 黑名单用户  
 val dayString: String = value.getString(0)  
 val userId: String = value.getString(1)  
 client.sadd(s"day:blcklist:$dayString", userId)  
 }  
   
 override def close(errorOrNull: Throwable): Unit = {  
 // 关闭到redis的连接  
 if (client != null) client.close()  
 }  
 })  
 .option("checkpointLocation", "C:/blacklist")  
 .start()  
   
 // 4. 把过滤后的数据返回 (在其他地方也可以使用临时表: tb\_ads\_info)  
 filteredAdsInfoDS  
 }  
}

#### 2.3.3 RealtimeApp

package com.atguigu.realtime.app  
  
import java.sql.Timestamp  
import java.text.SimpleDateFormat  
import java.util.Date  
  
import com.atguigu.realtime.bean.AdsInfo  
import org.apache.spark.sql.\_  
  
  
/\*\*  
 \* Author lzc  
 \* Date 2019-08-17 17:52  
 \*/  
object RealtimeApp {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession  
 .builder()  
 .appName("RealtimeApp")  
 .master("local[\*]")  
 .getOrCreate()  
 import spark.implicits.\_  
 spark.sparkContext.setLogLevel("warn")  
 // 从 kafka 读取数据, 为了方便后续处理, 封装数据到 AdsInfo 样例类中  
 val dayStringFormatter: SimpleDateFormat = new SimpleDateFormat("yyyy-MM-dd")  
 val hmStringFormatter: SimpleDateFormat = new SimpleDateFormat("HH:mm")  
 val adsInfoDS: Dataset[AdsInfo] = spark.readStream  
 .format("kafka")  
 .option("kafka.bootstrap.servers", "hadoop201:9092,hadoop202:9092,hadoop203:9092")  
 .option("subscribe", "ads\_log")  
 .load  
 .select("value")  
 .as[String]  
 .map(v => {  
 val split: Array[String] = v.split(",")  
 val date: Date = new Date(split(0).toLong)  
 AdsInfo(split(0).toLong, new Timestamp(split(0).toLong), dayStringFormatter.format(date), hmStringFormatter.format(date), split(1), split(2), split(3), split(4))  
 })  
 .withWatermark("timestamp", "24 hours") // 都是统计每天的数据, 对迟到24小时的数据废弃不用  
   
   
 // 需求1: 黑名单  
 val filteredAdsInfoDS: Dataset[AdsInfo] = BlackListApp.statBlackList(spark, adsInfoDS)  
 // 需求2:  
 AdsClickCountApp.statAdsClickCount(spark, filteredAdsInfoDS)  
   
   
 }  
}

## 第 3 章 广告点击量实时统计

### 3.1 需求简介

每天每地区每城市每广告的点击流量实时统计

### 3.2 思路

/\*  
统计成功之后写入到redis  
  
值的类型使用hash  
  
key value  
  
"date:area:city:ads" field: value  
 2019-08-19:华北:北京:5 10000  
   
使用sql 查询比较简单  
 \*/

### 3.3 具体实现

package com.atguigu.realtime.app  
  
import com.atguigu.realtime.bean.AdsInfo  
import com.atguigu.realtime.util.RedisUtil  
import org.apache.spark.sql.{Dataset, SparkSession}  
import redis.clients.jedis.Jedis  
  
/\*\*  
 \* Author lzc  
 \* Date 2019-08-19 10:48  
 \*  
 \*  
 \* 每天每地区每城市广告点击量实时统计  
 \*/  
object AdsClickCountApp {  
 val key: String = "date:area:city:ads"  
   
 def statAdsClickCount(spark: SparkSession, filteredAdsInfoDS: Dataset[AdsInfo]) = {  
 spark.sql(  
 s"""  
 |select  
 | dayString,  
 | area,  
 | city,  
 | adsId,  
 | count(1) count  
 |from tb\_ads\_info  
 |group by dayString, area, city, adsId  
 """.stripMargin)  
 .writeStream  
 .outputMode("update")  
 .foreachBatch((df, batchId) => { // 使用foreachBatch  
 if (df.count() > 0) {  
 df.cache() // 做缓存防止重复调用  
 df.foreachPartition(rowIt => {  
 val client: Jedis = RedisUtil.getJedisClient  
 // 1. 把数据存入到map中, 向redis写入的时候比较方便  
 val fieldValueMap: Map[String, String] = rowIt.map(row => {  
 // 2019-08-19:华北:北京:5  
 val field: String = s"${row.getString(0)}:${row.getString(1)}:${row.getString(2)}:${row.getString(3)}"  
 val value: Long = row.getLong(4)  
 (field, value.toString)  
 }).toMap  
 // 2. 写入到redis  
 // 用于把scala的集合转换成java的集合  
 import scala.collection.JavaConversions.\_  
 if (fieldValueMap.nonEmpty) client.hmset(key, fieldValueMap)  
 client.close()  
 })  
   
 df.unpersist() // 释放缓存  
 }  
 })  
 .start  
 .awaitTermination  
   
 }  
}  
  
/\*  
统计成功之后写入到redis  
  
值的类型使用hash  
  
key value  
  
"date:area:city:ads" field: value  
 2019-08-19:华北:北京:5 10000  
   
使用sql 查询比较简单  
 \*/

