Spark Structured Streaming入门编程指南

概览

Structured Streaming 是一个可拓展，容错的，基于Spark SQL执行引擎的流处理引擎。使用小量的静态数据模拟流处理。伴随流数据的到来，Spark SQL引擎会逐渐连续处理数据并且更新结果到最终的Table中。你可以在Spark SQL上引擎上使用DataSet/DataFrame API处理流数据的聚集，事件窗口，和流与批次的连接操作等。最后Structured Streaming系统快速，稳定，端到端的恰好一次保证，支持容错的处理。

小样例

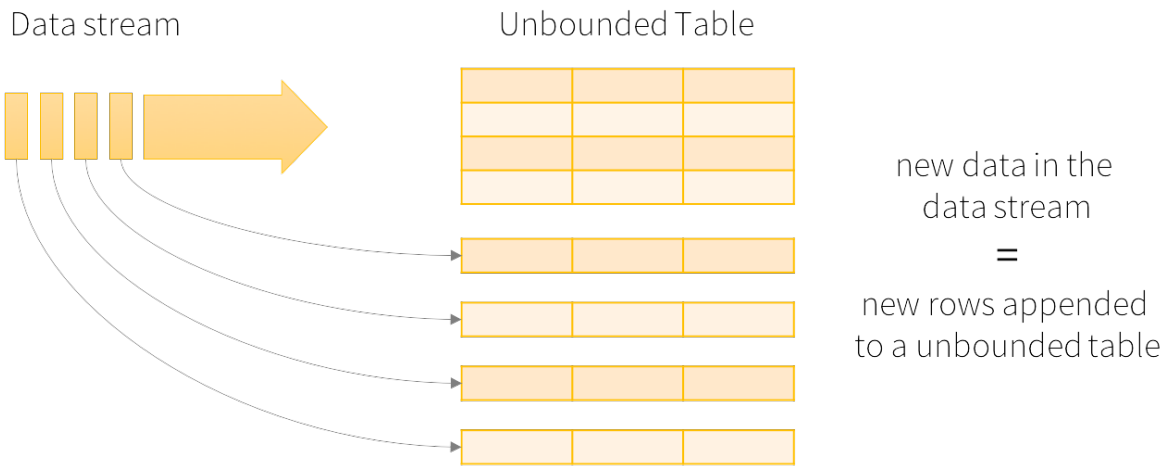
```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder
  .appName("StructuredNetworkWordCount")
  .getOrCreate()

import spark.implicits._

val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

// Split the lines into words
val words = lines.as[String].flatMap(_.split(" "))

// Generate running word count
val wordCounts = words.groupBy("value").count()
val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

编程模型

结构化流的关键思想是将实时数据流视为一个连续附加的表

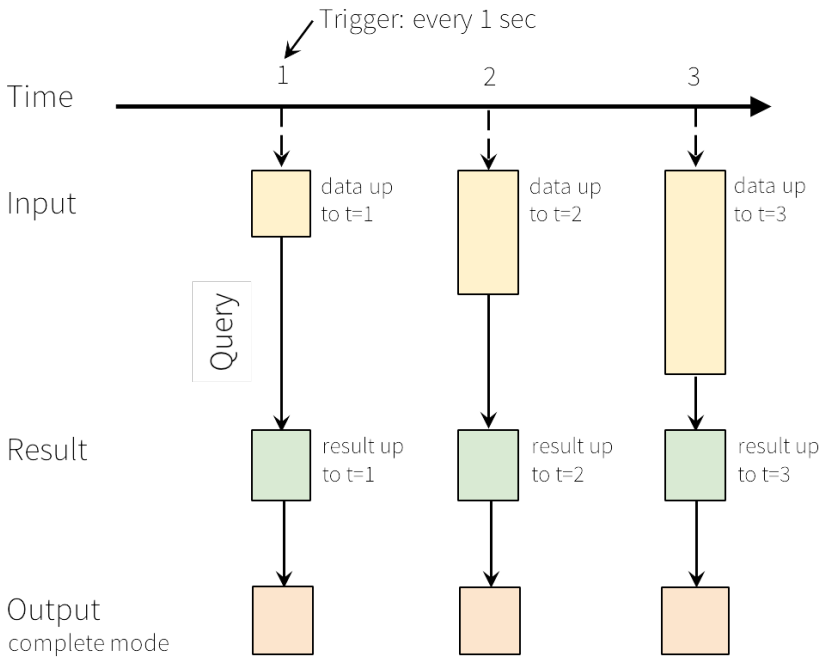
基本概念

将输入的数据当成一个输入的表格，每一个数据当成输入表的一个新行，如下图所示：



Data stream as an unbounded table

如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop



Programming Model for Structured Streaming

如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

Output是写入到外部存储的写方式，写入方式有不同的模式：

- Complete模式： 将整个更新表写入到外部存储，写入整个表的方式由存储连接器决定。
- Append模式：只有自上次触发后在结果表中附加的新行将被写入外部存储器。这仅适用于结果表中的现有行不会更改的查询。
- Update模式：只有自上次触发后在结果表中更新的行将被写入外部存储器（在Spark 2.0中尚不可用）。注意，这与完全模式不同，因为此模式不输出未更改的行。

处理事件时间和延迟数据

事件时间是嵌入在数据本身中的时间。对于许多应用程序，您可能希望在此事件时间操作。例如，如果要获取IoT设备每分钟生成的事件数，则可能需要使用生成数据的时间（即数据中的事件时间），而不是Spark接收的时间他们。此事件时间在此模型中非常自然地表示 - 来自设备的每个事件都是表中的一行，事件时间是该行中的一个列值。这允许基于窗口的聚合（例如每分钟的事件数）仅仅是偶数时间列上的特殊类型的分组和聚合 - 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（例如，来自收集的设备事件日志）以及数据流上一致地定义这种基于事件时间窗的聚合查询，使得用户的生活更容易。

此外，该模型自然地处理基于其事件时间比预期到达的数据。由于Spark正在更新结果表，因此当存在延迟数据时，它可以完全控制更新旧聚合，以及清除旧聚合以限制中间状态数据的大小。

由于Spark 2.1，我们支持水印，允许用户指定后期数据的阈值，并允许引擎相应地清除旧的状态。稍后将在“窗口操作”部分中对此进行详细说明。

容错语义

提供端到端的一次性语义是结构化流的设计背后的关键目标之一。为了实现这一点，我们设计了结构化流源，接收器和执行引擎，以可靠地跟踪处理的确切进展，以便它可以通过重新启动和/或重新处理来处理任何类型的故障。假定每个流源具有偏移量（类似于Kafka偏移量或Kinesis序列号）以跟踪流中的读取位置。引擎使用检查点和预写日志来记录每个触发器中正在处理的数据的偏移范围。流接收器被设计为用于处理再处理的幂等。结合使用可重放源和幂等宿，结构化流可以确保在任何故障下的端到端的一次性语义。

使用DataFrame和DataSet API

从Spark 2.0开始，DataFrames和Datasets可以表示静态，有界数据，以及流式，无界数据。与静态DataSets/ DataFrames类似，您可以使用公共入口点SparkSession（Scala / Java / Python文档）从流源创建流DataFrames /DataSets，并对它们应用与静态DataFrames / Datasets相同的操作。如果您不熟悉Datasets / DataFrames，强烈建议您使用DataFrame / Dataset编程指南熟悉它们。

创建数据框流和数据集流

Streaming
DataFrames可以通过SparkSession.readStream()返回的DataStreamReader接口（Scala / Java / Python docs）创建。类似于用于创建静态DataFrame的读取接口，您可以指定源 - 数据格式，模式，选项等的详细信息。

数据源

在Spark 2.0，有几个内置的数据源：

- 文件源：将写入目录中的文件读取为数据流。支持的文件格式有text，csv，json，parquet。请参阅DataStreamReader界面的文档以获取更新的列表，以及每种文件格式支持的选项。注意，文件必须原子地放置在给定目录中，在大多数文件系统中，可以通过文件移动操作来实现。
- Kafka源：从kafka拉取数据，支持kafka broker versions 0.10.0 or higher.从kafka集成指南获取更多信息。
- Socket源（测试用）：从套接字连接读取UTF8文本数据。监听服务器套接字在驱动程序。注意，这应该仅用于测试，因为这不提供端到端容错保证

这些示例生成无类型的流式DataFrames，这意味着在编译时不检查DataFrame的模式，仅在提交查询时在运行时检查。一些操作，如map，flatMap等，需要在编译时知道类型。要做到这些，你可以使用与静态DataFrame相同的方法将这些无类型的流DataFrames转换为类型化流数据集。有关更多详细信息，请参阅SQL编程指南。此外，有关支持的流媒体源的更多详细信息将在文档中稍后讨论。

数据框/数据集流的模式推理和分区

默认情况下，基于文件的源的结构化流要求您指定模式，而不是依靠Spark自动推断。此限制确保即使在发生故障的情况下，一致的模式也将用于流式查询。对于临时用例，可以通过将spark.sql.streaming.schemaInference设置为true来重新启用模式推断。
当名为/ key = value /的子目录存在时，发生分区发现，并且列表将自动递归到这些目录中。如果这些列出现在用户提供的模式中，它们将由Spark根据正在读取的文件的路径填充。当查询开始时，组成分区方案的目录必须存在，并且必须保持静态。例如，可以添加/ data / year = 2016 / when / data / year = 2015 /存在，但是更改分区列是无效的（即通过创建目录/ data / date = 2016-04-17 /）。

流式DataFrames/Datasets上的操作

您可以对流式DataFrames /数据集应用各种操作 - 从无类型，类似SQL的操作（例如select，where，groupBy）到类型化的RDD类操作（例如map，filter，flatMap）。有关更多详细信息，请参阅SQL编程指南。让我们来看看一些你可以使用的示例操作。

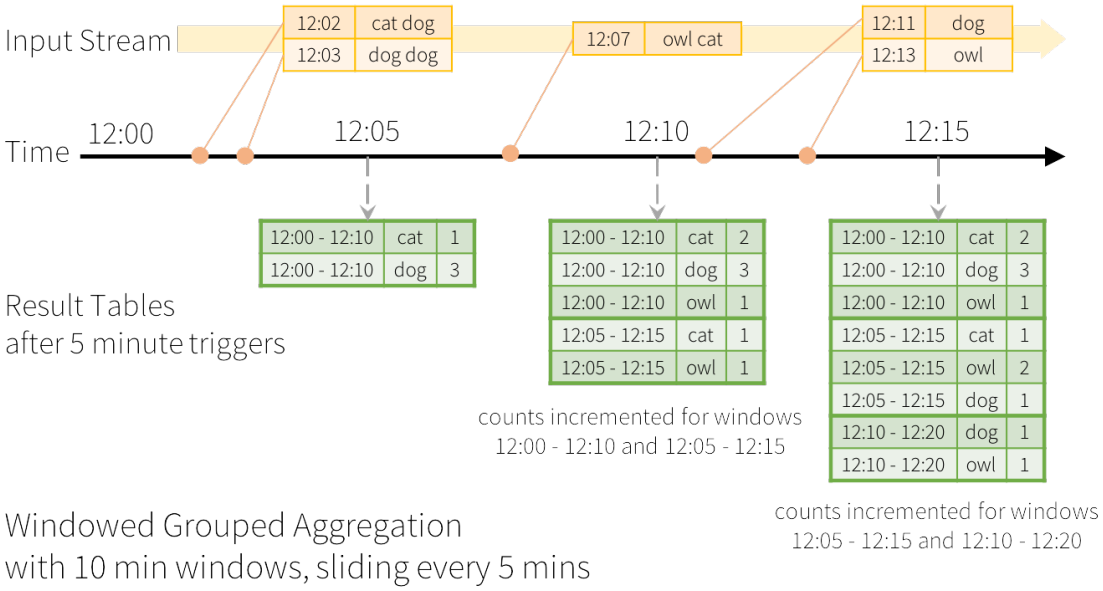
基本操作 - 选择，投影，聚合

```
case class DeviceData(device: String, type: String, signal: Double, time: DateTime)

val df: DataFrame = ... // streaming DataFrame with IOT device data with schema { device: string, type: string, signal: double, time: string }
val ds: Dataset[DeviceData] = df.as[DeviceData]    // streaming Dataset with IOT device data

// Select the devices which have signal more than 10
df.select("device").where("signal > 10")      // using untyped APIs
ds.filter(_.signal > 10).map(_.device)         // using typed APIs

// Running count of the number of updates for each device type
df.groupBy("type").count()                          // using untyped API

// Running average signal for each device type
import org.apache.spark.sql.expressions.scalalang.typed._
ds.groupByKey(_.type).agg(typed.avg(_.signal))    // using typed API
```

事件时间上的窗口操作

滑动事件时间窗口上的聚合通过结构化流直接进行。理解基于窗口的聚合的关键思想与分组聚合非常相似。在分组聚合中，为用户指定的分组列中的每个唯一值维护聚合值（例如计数）。在基于窗口的聚合的情况下，对于行的事件时间落入的每个窗口维持聚合值。让我们用插图来理解这一点。

想象一下，我们的快速示例被修改，流现在包含行以及生成行的时间。我们不想运行字数，而是计算10分钟内的字数，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等之间接收的词中的字数。注意，12:00 -12：10意味着数据在12:00之后但在12:10之前到达。现在，考虑在12:07收到的一个字。这个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，计数将通过分组键（即字）和窗口（可以从事件时间计算）来索引。结果表将如下所示：

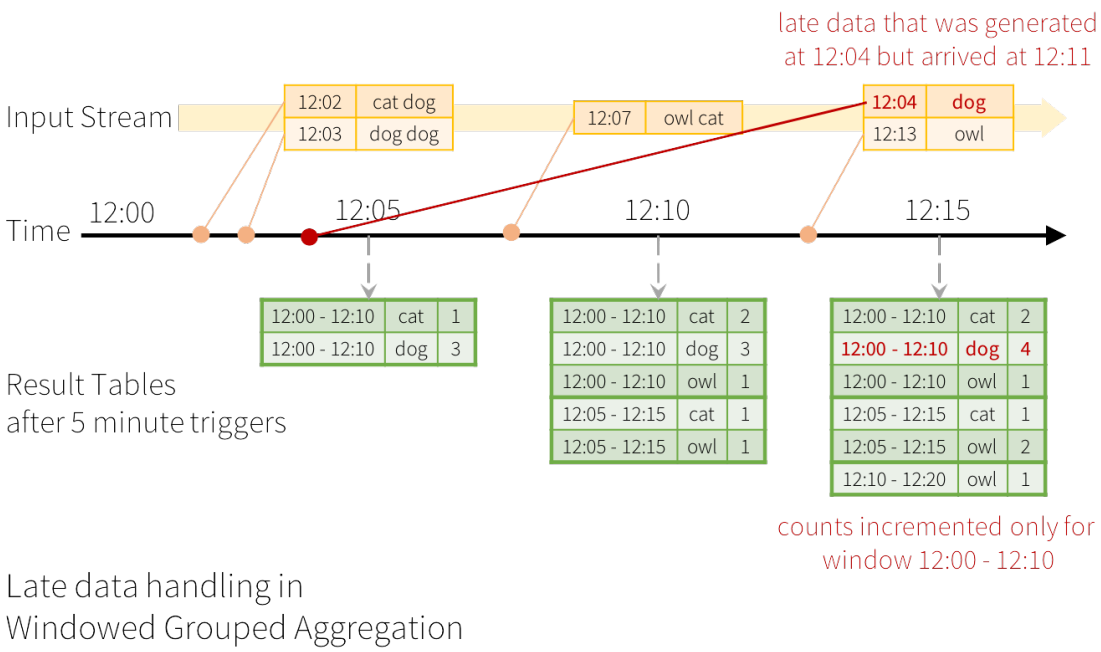


如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

由于此窗口类似于分组，因此在代码中，可以使用groupBy()和window()操作来表示窗口化聚合。您可以在Scala / Java / Python中查看以下示例的完整代码。

处理延迟数据和水位线

现在考虑如果事件中的一个到达应用程序的迟到会发生什么。例如，例如，在12:04（即事件时间）生成的词可以由应用在12:11接收到。应用程序应使用时间12:04而不是12:11来更新窗口12:00 - 12:10的旧计数。这在我们的基于窗口的分组中自然地发生 - 结构化流可以长时间地保持部分聚合的中间状态，使得晚期数据可以正确地更新旧窗口的聚集，如下所示：

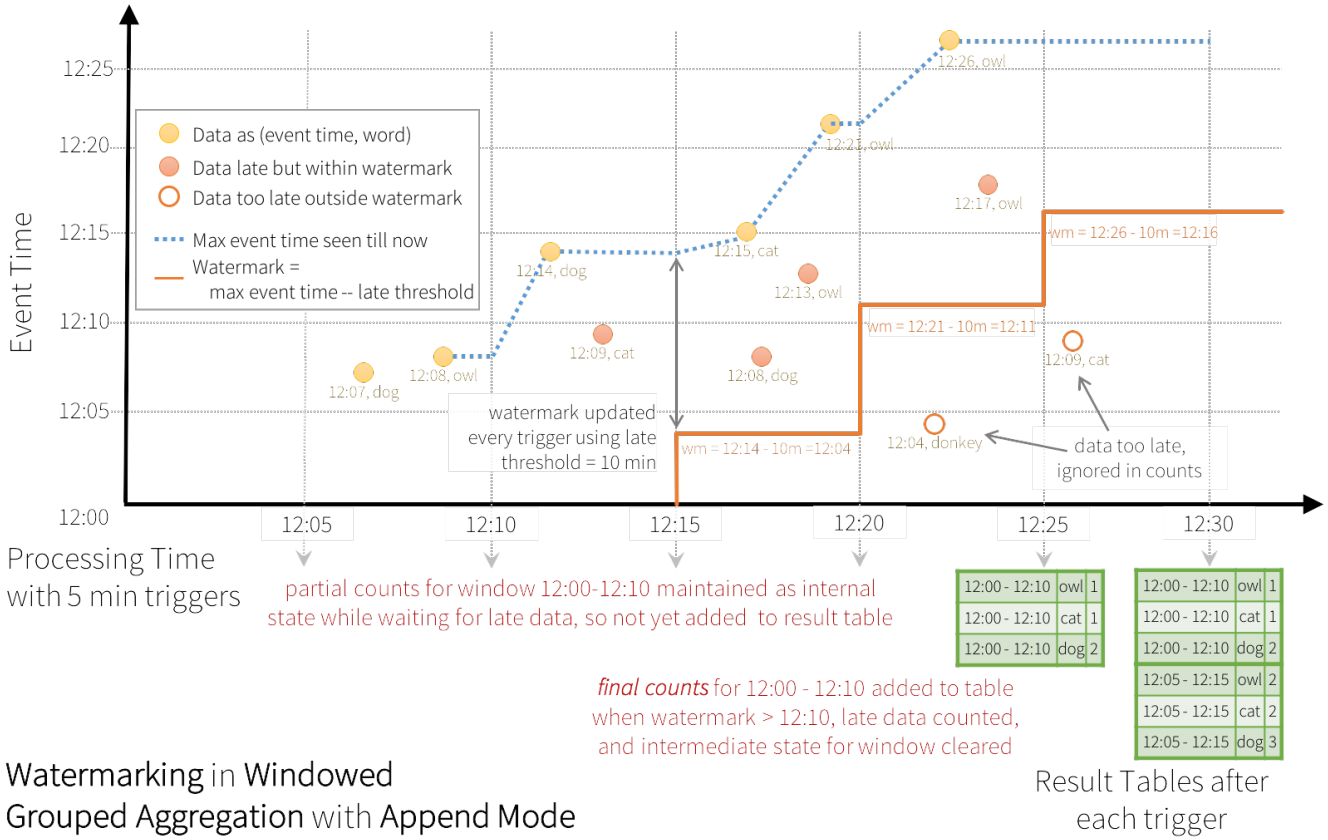


如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

但是，要运行此查询的天数，系统必须绑定其累积的中间内存中状态的数量。这意味着系统需要知道何时可以从内存中状态删除旧聚合，因为应用程序将不再接收该聚合的延迟数据。为了实现这一点，在Spark 2.1中，我们引入了水印，让我们的引擎自动跟踪数据中的当前事件时间，并尝试相应地清理旧的状态。您可以通过指定事件时间列和根据事件时间预计数据延迟的阈值来定义查询的水印。对于在时间T开始的特定窗口，引擎将保持状态并允许后期数据更新状态，直到（由引擎看到的最大事件时间 - 后期阈值> T）。换句话说，阈值内的晚数据将被聚合，但晚于阈值的数据将被丢弃。让我们用一个例子来理解这个。我们可以使用withWatermark()在上面的例子中轻松定义水印，如下所示。

```
import spark.implicits._

val words = ... // streaming DataFrame of schema { timestamp: Timestamp, word: String }

// Group the data by window and word and compute the count of each group
val windowedCounts = words
    .withWatermark("timestamp", "10 minutes")
    .groupBy(
        window($"timestamp", "10 minutes", "5 minutes"),
        $"word")
    .count()
```

在这个例子中，我们定义查询的水印对列“timestamp”的值，并且还定义“10分钟”作为允许数据超时的阈值。如果此查询在Append输出模式（稍后在“输出模式”部分中讨论）中运行，则引擎将从列“timestamp”跟踪当前事件时间，并在最终确定窗口计数和添加之前等待事件时间的额外“10分钟”他们到结果表。这是一个例证。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

如图所示，由引擎跟踪的最大事件时间是蓝色虚线，并且在每个触发的开始处设置为（最大事件时间 - '10分钟'）的水印是红色线。例如，当引擎观察数据（12:14，狗），它将下一个触发器的水印设置为12:04。对于窗口12:00 - 12:10，部分计数保持为内部状态，而系统正在等待延迟数据。在系统发现数据（即（12:21，owl））使得水印超过12:10之后，部分计数被最终确定并附加到表。此计数将不会进一步更改，因为所有超过12:10的“太晚”数据将被忽略。

请注意，在追加输出模式下，系统必须等待“延迟阈值”时间才能输出窗口的聚合。如果数据可能很晚，（例如1天），并且您希望部分计数而不等待一天，这可能不是理想的。将来，我们将添加更新输出模式，这将允许每次更新聚合写入到每个触发器。

用于清除聚合状态的水印的条件重要的是要注意，水印应当满足以下条件以清除聚合查询中的状态（从Spark 2.1开始，将来会改变）。

- 输出模式必须为追加。完成模式要求保留所有聚合数据，因此不能使用水印来删除中间状态。有关每种输出模式的语义的详细说明，请参见“输出模式”部分。
- 聚合必须具有事件时列，或事件时列上的窗口。
- withWatermark必须在与聚合中使用的时间戳列相同的列上调用。例如：df.withWatermark("time", "1 min").groupBy("time2").count()在Append输出模式下无效，因为水印是在与聚合列不同的列上定义的。
- 其中在要使用水印细节的聚合之前必须调用withWatermark。例如：df.groupBy("time").count().withWatermark("time", "1 min")在Append输出模式中无效。

Join操作

流DataFrames可以与静态DataFrames连接以创建新的流DataFrames。这里有几个例子。

```
val staticDf = spark.read. ...
val streamingDf = spark.readStream. ...

streamingDf.join(staticDf, "type")          // inner equi-join with a static DF
streamingDf.join(staticDf, "type", "right_join")  // right outer join with a static DF
```

不支持的操作

但是，请注意，所有适用于静态DataFrames /数据集的操作在流式DataFrames /数据集中不受支持。虽然这些不支持的操作中的一些将在未来的Spark版本中得到支持，但还有一些基本上难以有效地在流数据上实现。例如，输入流数据集不支持排序，因为它需要跟踪流中接收的所有数据。因此，这在根本上难以有效地执行。从Spark 2.0开始，一些不受支持的操作如下：

- 在流数据集上还不支持多个流聚集（即，流DF上的聚合链）。
- 在流数据集上不支持限制和获取前N行。
- 不支持对流数据集进行不同操作。
- 排序操作仅在聚合后在完整输出模式下支持流数据集。
- 条件支持流式传输和静态数据集之间的外连接。
- 不支持带有流数据集的完全外连接
- 不支持左外部连接与右侧的流数据集
- 不支持左侧的流数据集的右外部联接
- 尚不支持两个流数据集之间的任何类型的连接。

此外，还有一些Dataset方法不能用于流数据集。它们是将立即运行查询并返回结果的操作，这对流数据集没有意义。相反，这些功能可以通过显式地启动流查询来完成（参见下一部分）。

- count() - 无法从流数据集返回单个计数。
  相反，使用ds.groupBy.count()返回包含运行计数的流数据集。
- foreach() - 而是使用ds.writeStream.foreach（...）（参见下一节）。
- show() - 而是使用控制台接收器（请参阅下一节）。

如果您尝试任何这些操作，您将看到一个AnalysisException如“操作XYZ不支持与流DataFrames/数据集”。

启动流式查询

一旦定义了最终结果DataFrame / Dataset，剩下的就是启动流计算。为此，您必须使用通过Dataset.writeStream()返回的DataStreamWriter。您必须在此界面中指定以下一个或多个。

- 输出接收器的详细信息：数据格式，位置等
- 输出模式：指定写入输出接收器的内容。
- 查询名称：（可选）指定查询的唯一名称以进行标识。
- 触发间隔：可选择指定触发间隔。如果未指定，系统将在上一个处理完成后立即检查新数据的可用性。如果由于先前处理尚未完成而错过触发时间，则系统将尝试在下一触发点处触发，而不是在处理完成之后立即触发。
- 检查点位置：对于可以保证端到端容错的某些输出接收器，请指定系统将写入所有检查点信息的位置。这应该是HDFS兼容的容错文件系统中的目录。检查点的语义将在下一节中更详细地讨论。

输出模式

有几种类型的输出模式：

- 附加模式（默认） - 这是默认模式，其中只有自上次触发后添加到结果表中的新行将输出到接收器。这仅支持那些添加到结果表中的行从不会更改的查询。因此，该模式保证每行只输出一次（假设容错宿）。例如，只有select，where，map，flatMap，filter，join等的查询将支持Append模式。
- 完成模式 - 每次触发后，整个结果表将输出到接收器。聚合查询支持此选项。
- 更新模式 - （在Spark 2.1中不可用）只有结果表中自上次触发后更新的行才会输出到接收器。更多信息将在未来版本中添加。

不同类型的流查询支持不同的输出模式。这里是兼容性矩阵：

查询类型		支持的输出模式	注
无聚合的查询		Append	支持完整模式，因为不可能将所有数据保存在结果表中。
带有聚合的聚合	在带有水印的event-time上聚合	Append, Complete	