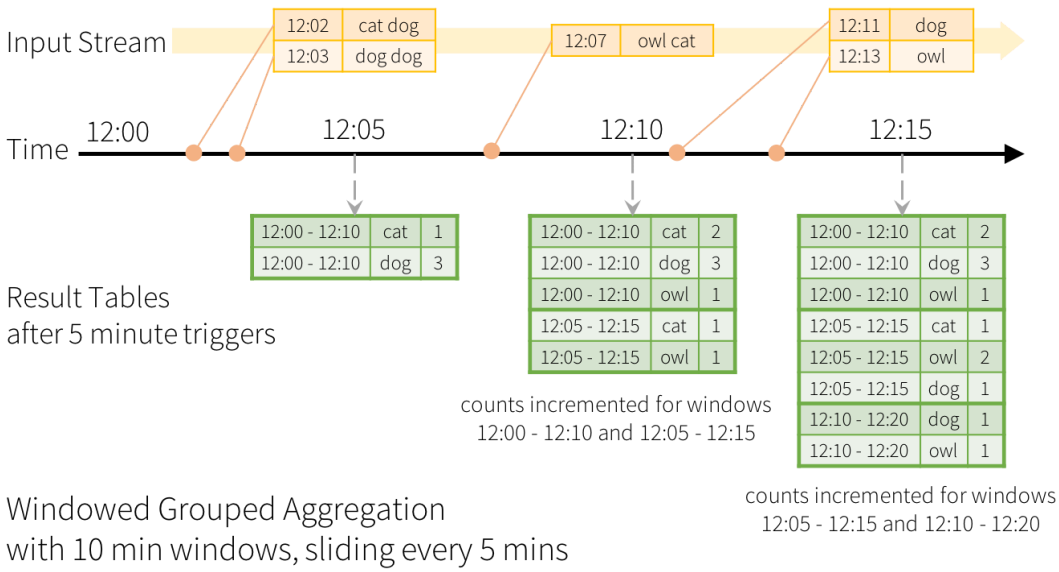
窗口操作

1. 事件时间上的窗口操作

事件时间是嵌入在数据本身的时间，对于许多应用程序，我们可能希望根据事件时间进行聚合操作，为此，Spark2.x 提供了基于滑动窗口的事件时间聚合操作。基于分组的聚合操作和基于窗口的聚合操作是非常相似的，在分组聚合中，依据用户指定的分组列中的每个唯一值维护聚合值，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：



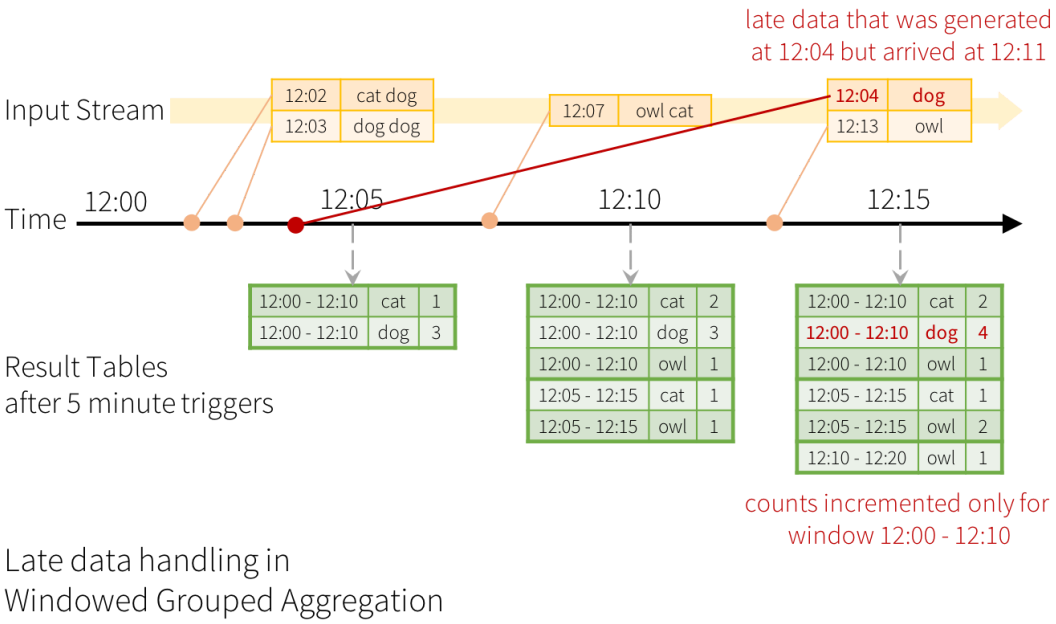

```
def main(args: Array[String]) {

      val spark = SparkSession
      .builder
          .master("local[2]")
      .appName("test")
      .getOrCreate()

      import spark.implicits._
      // Create DataFrame representing the stream of input lines from connection to host:port

      val lines = spark.readStream
      .format("socket")

      .option("host", "node1")
```

```
        .option("port", "8989")
        .option("includeTimestamp", true) //输出内容包括时间戳
        .load()

        // Split the lines into words, retaining timestamps

        val words = lines.as[(String, Timestamp)].flatMap(line =>
        line._1.split(" ").map(word => (word, line._2))
        ).toDF("word", "timestamp")

        // 对数据按照窗口和单词进行分组，并计算每组中不同单词的个数

        //设置窗口大小和滑动窗口步长
      //用于统计每10分钟内，接受到的不同词的个数，其中window($"timestamp", "10 minutes", "5 minutes")的含义为：假设初始时间 t=12:00，定义时间窗口为10分钟，每5分钟窗口滑动一次，也就是每5分钟对大小为10分钟的时间窗口进行一次聚合操作，并且聚合操作完成后，窗口向前滑动5分钟，产生新的窗口

        val windowedCounts = words.groupBy(
          window($"timestamp", "10 minutes", "5 minutes"), $"word"
        ).count().orderBy("window")

        //由于采用聚合操作，所以需要指定"complete"输出形式。指定"truncate"只是为了在控制台输出时，不进行列宽度自动缩小。

        val query = windowedCounts.writeStream
        .outputMode("complete")
        .format("console")
        .option("truncate", "false")
        .start()
        query.awaitTermination()

        }
```

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。

2. 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：

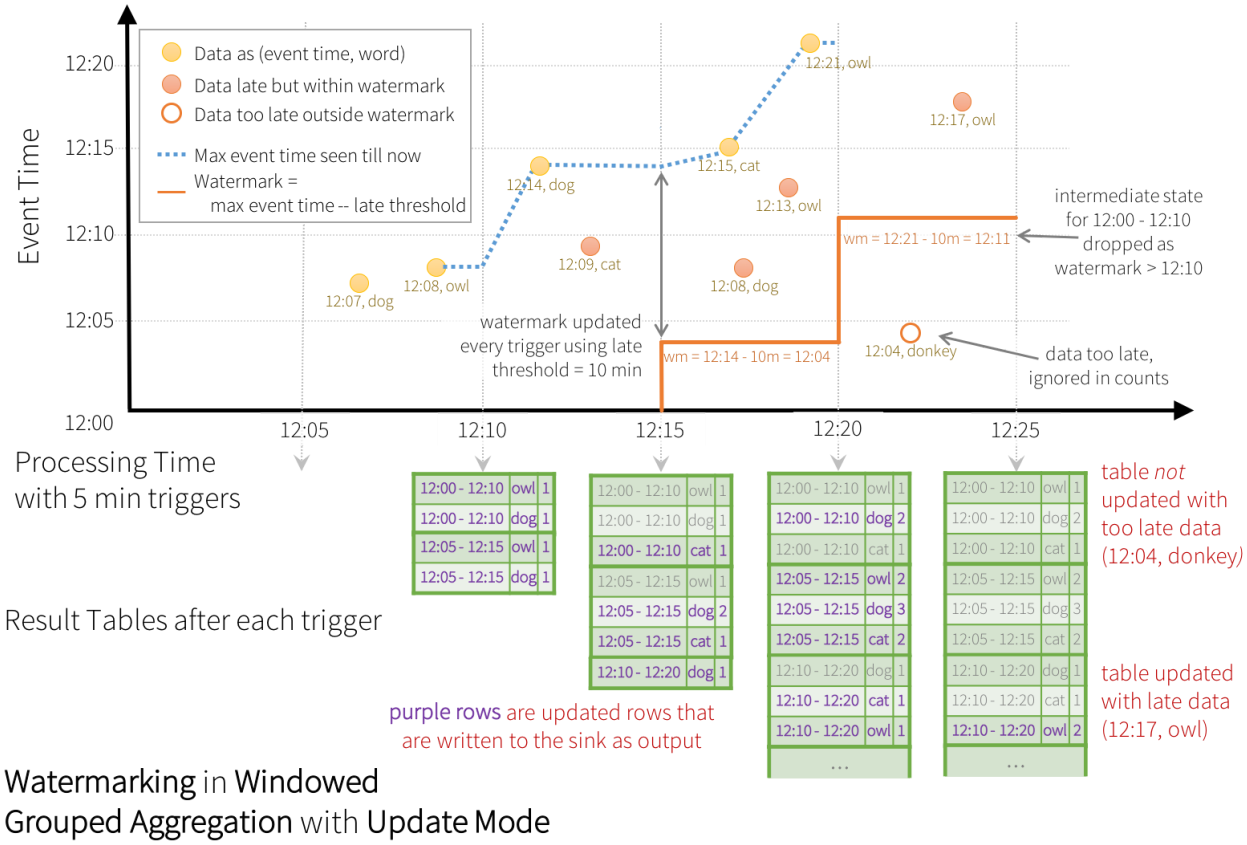


Late data handling in
Windowed Grouped Aggregation

是，要运行此查询数天，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，我们引入了水位线，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位线。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。

```
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，我们定义查询的水位线对列“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。



Watermarking in **Windowed Grouped Aggregation** with **Update Mode**

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。
  该段代码用于用于统计每10分钟内，接受到的不同词的个数，其中window($”timestamp”, “10 minutes”, “5 minutes”)的含义为：假设初始时间 t=12:00，定义时间窗口为10分钟，每5分钟窗口滑动一次，也就是每5分钟对大小为10分钟的时间窗口进行一次聚合操作，并且聚合操作完成后，窗口向前滑动5分钟，产生新的窗口，如上图的一些列窗口 12:00-12:10,12:05-12:15,12:10-12:20。

在这里每个word包含两个时间，word产生的时间和流接收到word的时间，这里的timestamp就是word产生的时间，在很多情况下，word产生后，可能会延迟很久才被流接收，为了处理这种情况，Structured Streaming 引进了Watermarking(时间水印)功能，以保证能正确的对流的聚合结构进行更新

Watermarking的计算方法Watermarking：

- In every trigger, while aggregate the data, we also scan for the max value of event time in the trigger data
- After trigger completes, compute watermark = MAX(event time before trigger, max event time in trigger)

Watermarking表示多长时间以前的数据将不再更新，也就是说每次窗口滑动之前会进行Watermarking的计算，首先统计这次聚合操作返回的最大事件时间，然后减去所然忍受的延迟时间就是Watermarking，当一组数据或新接收的数据事件时间小于Watermarking时，则该数据不会更新，在内存中就不会维护该组数据的状态

Structured Streaming 支持两种更新模式： Update 删除不再更新的时间窗口，每次触发聚合操作时，输出更新的窗口

2. Append 当确定不会更新窗口时，将会输出该窗口的数据并删除，保证每个窗口的数据只会输出一次
3. Complete 不删除任何数据，在 Result Table 中保留所有数据，每次触发操作输出所有窗口数据
   自Spark 2.3开始，Spark Structured Streaming开始支持Stream-stream Joins。两个流之间的join与静态的数据集之间的join有一个很大的不同，那就是，对于流来说，在任意时刻，在join的两边（也就是两个流上），数据都是"不完全"的，当前流上的任何一行数据都可能会和被join的流上的未来某行数据匹配到，为此，Spark必须要缓存流上过去所有的输入，以Stream State信息的形式来维护。当然，和流上的聚合运算一样，我们也可以通过wartermark来处理data late问题。

3. 数据流之间的join操作

内关联（Inner Joins） Spark Structured Streaming支持任何列之间的join，但是这样会带来一个问题：随着stream的长期运行，stream的状态数据会无限制地增长，并且这些状态数据不能被释放，因为如前所诉，不管多么旧的数据，在未来某个时刻都有可能会被join到，因此我们必须在join上追加一些额外的条件来改善state无止境的维持旧有输入数据的问题。我们可以使用的方法有：

定义waterwark, 抛弃超过约定时限到达的输入数据。 在事件时间上添加约束，约定数据多旧之后就不再参与join了，这种约束可以通过以下两种方式之一来定义： 指定事件时间的区间：例如：JOIN ON leftTime BETWEN rightTime AND rightTime + INTERVAL 1 HOUR 指定事件时间的窗口：例如：JOIN ON leftTimeWindow = rightTimeWindow 在Spark的官方文档中有这样一个示例：在广告投放中，把用户打开包含广告的一个页面称为广告被"触达"(曝光)一次，即advertisement impressions次数加1（impression指的就是广告触达），这一 实时数据对应一个流，叫impressions流，如果用户点击了广告，这一数据也会被实时采集到，对应的流叫clicks流，如果我们想实时地分析一个广告的触达率和点击率之间的关系，就需要将这两个流按广告的ID进行join,这就是一个典型的stream-stream join的案例。现在，为了防止无止尽的状态数据，我们做如下限制：

触达（曝光）数据最多允许迟到2小时，点击数据最多允许迟到3小时。点击数据允许的延迟比触达数据长是有道理的，因为点击总是发生在用户看到广告之后，给定一个小时的时间差是一个比较宽泛和可靠的阈值。而在流上进行join时，我们需要相应地允许点击流join触达流上与之时间相同以及触达流上前推1小时内的数据，这与触达流(最大允许2小时延迟）允许比点击流(最大允许3小时延迟）多一小时的最大延迟时间是一致的。

以下代码是上述业务分析的实现：

```
import org.apache.spark.sql.functions.expr
val impressions = spark.readStream. ...
val clicks = spark.readStream. ...
// Apply watermarks on event-time columns
val impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
val clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")
// Join with event-time constraints
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
```

```
    """)
)
```

需要重点解释的是：两个流都通过withWatermark对数据延迟进行了限定，之后在join操作中通过clickTime >= impressionTime AND clickTime <= impressionTime + interval 1 hour来限定时间尺度上的界限条件，这样达到的效果就是：允许并只允许以当前时间为基准向前推2小时内的触达数据和3小时内的点击数据进行join，超出这个时间界限的数据不会被Stream State维护，避免无止尽的State。

外关联（Outer Joins） watermark + event-time对于内关联是非必须的，可选的（虽然大数情况下都应该制定），但是对外关联就是强制的了，因为在外关联中，如果关联的任何一方没有匹配的数据，都需要补齐空值，如果不对关联数据的范围进行限定，外关联的结果集会膨胀地非常厉害，也就是每一条没有匹配到的输入数据都要依据另一个流上的全体数据的总行数使用空值补齐。

前面内关联的例子使用外关联实现的代码如下：

```
import org.apache.spark.sql.functions.expr
val impressions = spark.readStream. ...
val clicks = spark.readStream. ...
// Apply watermarks on event-time columns
val impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
val clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")
// Join with event-time constraints
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """),
  joinType = "leftOuter"      // can be "inner", "leftOuter", "rightOuter"

)
```

区另仅在于追加了一个参数来指定关联类型是leftOuter join.

此外，外关联结果集的生成也有这样一些重要的特点：

外关联的空的结果集必须要等到时间走过了指定的watermark和time range条件之后才会生成，原因和前面解释outer join必须依赖watermark + event-time的原因是一样的，就是外关联下空值必须要补齐到另一方的所有行上，因此引擎必须要等待另一方全部的数据（就是watermark和time range条件限定范围内的数据）就位之后才能进行补齐操作。

与维护任意状态流时没有一个确定的timeout触发时间类似，在没有数据输入的情况下，外关联结果集的输出也会延迟，而且可能会延迟非常长的时间。引起这些情况的原因都与watermark机制有关，因为watermark更新是依赖每一个新进的mirco-batch上的数据的event-time, 如果迟迟没有新的数据输入，就不会驱动watermark的更新，进行所有依赖watermark进行时间范围判定的动作也不会被触发，或者触发也不会有所变化，因为watermark没有更新，因此对产生各种延迟。

4. 处理structured streaming数据流的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。

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）。

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
```

不支持的操作

但是，请注意，有一些适用于静态DataFrames /数据集的操作在流式DataFrames /数据集中不受支持，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（后续可能会有更新），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limit和获取前N行(take)。
- Distinct 操作在流数据集上不支持。
- 排序操作仅在聚合后在完整输出模式下支持流数据集。
- count() - 无法从流数据集返回单个计数。 但是可以使用ds.groupBy.count()返回组内统计结果。
- foreach() - 使用ds.writeStream.foreach（...）代替。
- show() - 使用讲数据下沉到控制台代替。
- 对各种join的支持（spark2.3）

Left Input	Right Input	Join Type	
Static	Static	All types	Supported, since its not on streaming data even though it can be present in a streaming query
Stream	Static	Inner	Supported, not stateful
Left Outer	Supported, not stateful		
Right Outer	Not supported		
Full Outer	Not supported		
Static	Stream	Inner	Supported, not stateful
Left Outer	Not supported		
Right Outer	Supported, not stateful		
Full Outer	Not supported		
Stream	Stream	Inner	Supported, optionally specify watermark on both sides + time constraints for state cleanup
Left Outer	Conditionally supported, must specify watermark on right + time constraints for correct results, optionally specify watermark on left for all state cleanup		
Right Outer	Conditionally supported, must specify watermark on left + time constraints for correct results, optionally specify watermark on right for all state cleanup		
Full Outer	Not supported		