# window（窗口）定义

Flink是基于流处理的，而在流处理中，数据是连续不断的，于是对于一些聚合等操作，我们不能等到所有的数据结束了才进行处理（没有结束），但是我们可以定义这样一种操作，例如：在过去的1分钟内有多少用户点击了我们的网页，在这种情况下，我们必须定义一个范围，用来收集最近一分钟内的数据，并对这个范围内的数据进行计算，这个范围就叫做窗口。

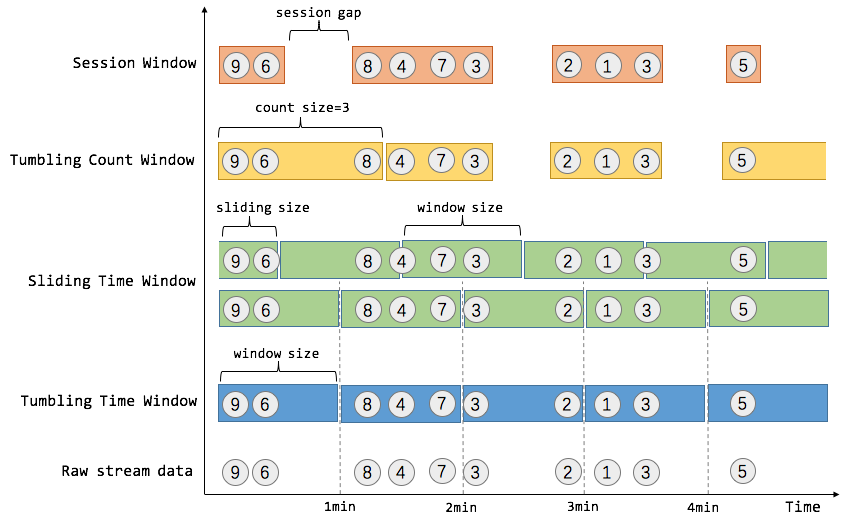
窗口可以是时间驱动的（Time Window，例如：每30秒钟），也可以是数据驱动的（Count Window，例如：每一百个元素）。

一种经典的窗口分类可以分成：

翻滚窗口（Tumbling Window，无重叠），

滚动窗口（Sliding Window，有重叠），

会话窗口（Session Window，活动间隙）



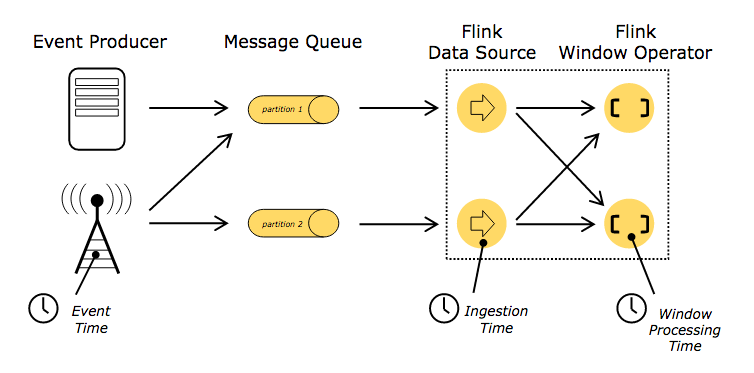
## Time window

三种时间概念：

event time（事件时间：事件发生时的时间，可能是由生产事件的传感器或生产服务来附加），

ingestion time（摄取时间：事件进入流处理系统的时间），

processing time（处理时间：事件被计算处理的时间）



### Tumbling Time Window

如上图，我们需要统计每一分钟中用户购买的商品的总数，需要将用户的行为事件按每一分钟进行切分，这种切分被成为翻滚时间窗口（Tumbling Time Window）。翻滚窗口能将数据流切分成不重叠的窗口，每一个事件只能属于一个窗口。通过使用 DataStream API，我们可以这样实现：

|  |
| --- |
| // Stream of (userId, buyCnt) val buyCnts: DataStream[(Int, Int)] = ...  val tumblingCnts: DataStream[(Int, Int)] = buyCnts  // key stream by userId  .keyBy(0)   // tumbling time window of 1 minute length  .timeWindow(Time.minutes(1))  // compute sum over buyCnt  .sum(1) |

### Sliding Time Window

但是对于某些应用，它们需要的窗口是不间断的，需要平滑地进行窗口聚合。比如，我们可以每30秒计算一次最近一分钟用户购买的商品总数。这种窗口我们称为滑动时间窗口（Sliding Time Window）。在滑窗中，一个元素可以对应多个窗口。通过使用 DataStream API，我们可以这样实现：

|  |
| --- |
|  |

|  |
| --- |
| val slidingCnts: DataStream[(Int, Int)] = buyCnts  .keyBy(0)   // sliding time window of 1 minute length and 30 secs trigger interval  .timeWindow(Time.minutes(1), Time.seconds(30))  .sum(1) |

## Count Window

基于事件个数对窗口分组

### Tumbling Count Window

当我们想要每100个用户购买行为事件统计购买总数，那么每当窗口中填满100个元素了，就会对窗口进行计算，这种窗口我们称之为翻滚计数窗口（Tumbling Count Window），上图所示窗口大小为3个。通过使用 DataStream API，我们可以这样实现：

|  |
| --- |
| // Stream of (userId, buyCnts) val buyCnts: DataStream[(Int, Int)] = ...  val tumblingCnts: DataStream[(Int, Int)] = buyCnts  // key stream by sensorId  .keyBy(0)  // tumbling count window of 100 elements size  .countWindow(100)  // compute the buyCnt sum   .sum(1) |

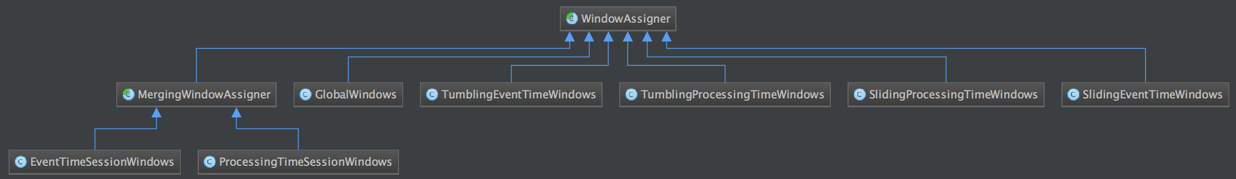
### Sliding Count Window

例如计算每10个元素计算一次最近100个元素的总和，代码示例如下。

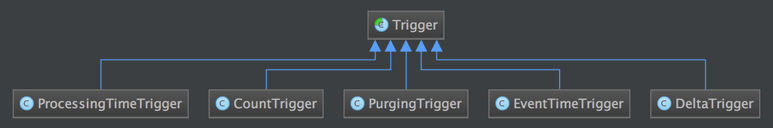
|  |
| --- |
| val slidingCnts: DataStream[(Int, Int)] = vehicleCnts  .keyBy(0)  // sliding count window of 100 elements size and 10 elements trigger interval  .countWindow(100, 10)  .sum(1) |

# Window工作流程

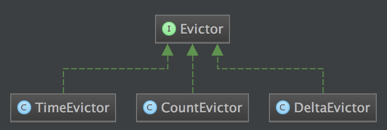
Window assigner：决定某个元素被分配到哪个/哪些窗口中去



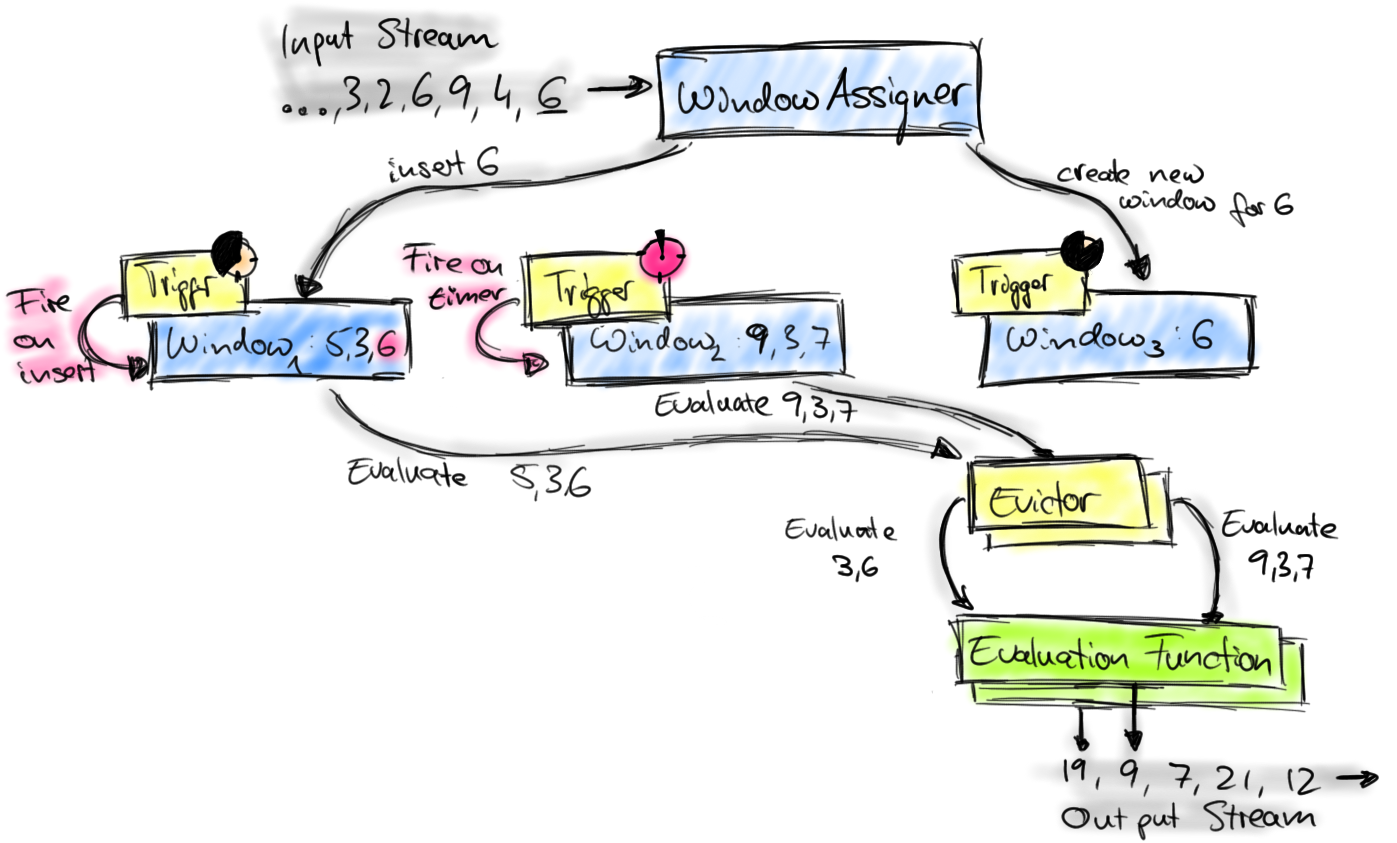
Trigger：触发器，决定一个窗口何时被计算或清除，每个窗口都有一个自己的Trigger，返回值有几种，continue（不做任何操作），fire（处理窗口数据），purge（移除窗口和窗口数据），或者fire+purge



Evictor：驱逐器，在Trigger触发之后，在窗口被处理之前，Evictor（如果有Evictor的话）会用来剔除窗口中不需要的元素，相当于一个filter。



各组件之间相互工作图



Window assigner -> trigger -> evictor -> function

步骤如下：

首先上图中的组件都位于一个算子（window operator）中，数据流源源不断地进入算子，每一个到达的元素都会被交给 WindowAssigner。WindowAssigner 会决定元素被放到哪个或哪些窗口（window），可能会创建新窗口。因为一个元素可以被放入多个窗口中，所以同时存在多个窗口是可能的。注意，Window本身只是一个ID标识符，其内部可能存储了一些元数据，如TimeWindow中有开始和结束时间，但是并不会存储窗口中的元素。窗口中的元素实际存储在 Key/Value State 中，key为Window，value为元素集合（或聚合值）。

每一个窗口都拥有一个属于自己的 Trigger，Trigger上会有定时器，用来决定一个窗口何时能够被计算或清除。每当有元素加入到该窗口，或者之前注册的定时器超时了，那么Trigger都会被调用。Trigger的返回结果可以是 continue（不做任何操作），fire（处理窗口数据），purge（移除窗口和窗口中的数据），或者 fire + purge。一个Trigger的调用结果只是fire的话，那么会计算窗口并保留窗口原样，也就是说窗口中的数据仍然保留不变，等待下次Trigger fire的时候再次执行计算。一个窗口可以被重复计算多次直到它被 purge 了。在purge之前，窗口会一直占用着内存。

当Trigger fire了，窗口中的元素集合就会交给Evictor（如果指定了的话）。Evictor 主要用来遍历窗口中的元素列表，并决定最先进入窗口的多少个元素需要被移除。剩余的元素会交给用户指定的函数进行窗口的计算。如果没有 Evictor 的话，窗口中的所有元素会一起交给函数进行计算。

计算函数收到了窗口的元素（可能经过了 Evictor 的过滤），并计算出窗口的结果值，并发送给下游。窗口的结果值可以是一个也可以是多个。DataStream API 上可以接收不同类型的计算函数，包括预定义的sum(),min(),max()，还有 ReduceFunction，FoldFunction，还有WindowFunction。WindowFunction 是最通用的计算函数，其他的预定义的函数基本都是基于该函数实现的。

Flink 对于一些聚合类的窗口计算（如sum,min）做了优化，因为聚合类的计算不需要将窗口中的所有数据都保存下来，只需要保存一个result值就可以了。每个进入窗口的元素都会执行一次聚合函数并修改result值。这样可以大大降低内存的消耗并提升性能。但是如果用户定义了 Evictor，则不会启用对聚合窗口的优化，因为 Evictor 需要遍历窗口中的所有元素，必须要将窗口中所有元素都存下来。

# Window源码分析 //todo

## Count window

|  |
| --- |
| // tumbling count window public WindowedStream<T, KEY, GlobalWindow> countWindow(long size) {  return window(GlobalWindows.create()) // create window stream using GlobalWindows  .trigger(PurgingTrigger.of(CountTrigger.of(size))); // trigger is window size } |

|  |
| --- |
| // sliding count window public WindowedStream<T, KEY, GlobalWindow> countWindow(long size, long slide) {  return window(GlobalWindows.create())  .evictor(CountEvictor.of(size)) // evictor is window size  .trigger(CountTrigger.of(slide)); // trigger is slide size } |

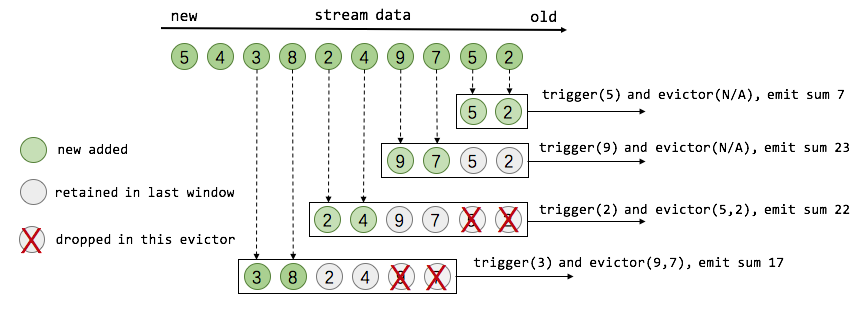
第一个函数是申请翻滚计数窗口，参数为窗口大小。

第二个函数是申请滑动计数窗口，参数分别为窗口大小和滑动大小。

它们都是基于 GlobalWindows 这个 WindowAssigner 来创建的窗口，该assigner会将所有元素都分配到同一个global window中，所有GlobalWindows的返回值一直是 GlobalWindow 单例。基本上自定义的窗口都会基于该assigner实现。

翻滚计数窗口并不带evictor，只注册了一个trigger。该trigger是带purge功能的 CountTrigger。也就是说每当窗口中的元素数量达到了 window-size，trigger就会返回fire+purge，窗口就会执行计算并清空窗口中的所有元素，再接着储备新的元素。从而实现了tumbling的窗口之间无重叠。

滑动计数窗口的各窗口之间是有重叠的，但我们用的 GlobalWindows assinger 从始至终只有一个窗口，不像 sliding time assigner 可以同时存在多个窗口。所以trigger结果不能带purge，也就是说计算完窗口后窗口中的数据要保留下来（供下个滑窗使用,但是在物理上只存在一个窗口）。另外，trigger的间隔是slide-size，evictor的保留的元素个数是window-size。也就是说，每个滑动间隔就触发一次窗口计算，并保留下最新进入窗口的window-size个元素，剔除旧元素。类似于下面图所示



## Time window

我们知道有三种时间类型，processing time/event time/ingest time

|  |
| --- |
| // tumbling time window public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size) {  if (environment.getStreamTimeCharacteristic() == TimeCharacteristic.ProcessingTime) {  return window(TumblingProcessingTimeWindows.of(size));  } else {  return window(TumblingEventTimeWindows.of(size));  } } |
| // sliding time window public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size, Time slide) {  if (environment.getStreamTimeCharacteristic() == TimeCharacteristic.ProcessingTime) {  return window(SlidingProcessingTimeWindows.of(size, slide));  } else {  return window(SlidingEventTimeWindows.of(size, slide));  } } |

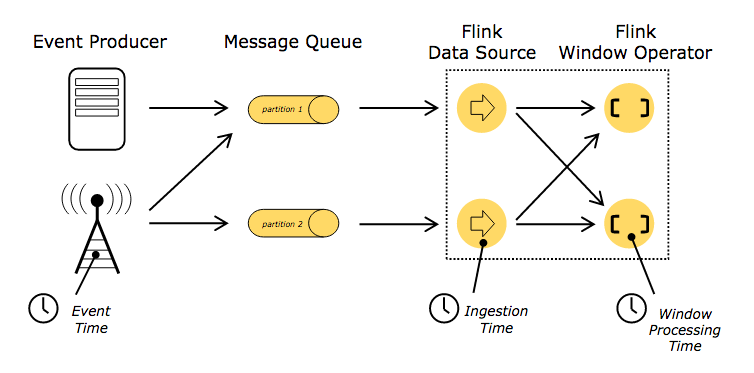
可见，event time 和ingestion time 在底层的实现方式的是一样的，都使用了XXXEventTimeWindow这个assigner。

# 消息乱序与watermark

processing Time。根据Task所在节点的本地时钟来切分的时间窗口。

Event Time。消息自带时间戳，根据消息的时间戳进行处理，确保时间戳在同一个时间窗口的所有消息一定会被正确处理。由于消息可能乱序流入Task，所以Task需要缓存当前时间窗口消息处理的状态，直到确认属于该时间窗口的所有消息都被处理，才可以释放，如果乱序的消息延迟很高会影响分布式系统的吞吐量和延迟。

Ingestion Time。有时消息本身并不带有时间戳信息，但用户依然希望按照消息而不是节点时钟划分时间窗口，此时可以在消息源流入Flink流处理系统时自动生成增量的时间戳赋予消息，之后处理的流程与Event Time相同。Ingestion Time可以看成是Event Time的一个特例，由于其在消息源处时间戳一定是有序的，所以在流处理系统中，相对于Event Time，其乱序的消息延迟不会很高，因此对Flink分布式系统的吞吐量和延迟的影响也会更小。



## Watermark

Flink借鉴了Google的MillWheel项目，通过WaterMark来支持基于Event Time的时间窗口。

当操作符通过基于Event Time的时间窗口来处理数据时，它必须在确定所有属于该时间窗口的消息全部流入此操作符后才能开始数据处理。但是由于消息可能是乱序的，所以操作符无法直接确认何时所有属于该时间窗口的消息全部流入此操作符。WaterMark包含一个时间戳，Flink使用WaterMark标记所有小于该时间戳的消息都已流入，Flink的数据源在确认所有小于某个时间戳的消息都已输出到Flink流处理系统后，会生成一个包含该时间戳的WaterMark，插入到消息流中输出到Flink流处理系统中，Flink操作符按照时间窗口缓存所有流入的消息，当操作符处理到WaterMark时，它对所有小于该WaterMark时间戳的时间窗口数据进行处理并发送到下一个操作符节点，然后也将WaterMark发送到下一个操作符节点。

当然，即使我们采用了watermark的方式，还是可能有一些在watermark(t)之后来的数据，它的timestamp t’<t(因为我们不能一直推迟watermark的发送，这样会推迟窗口的计算，一般设置最大乱序时间)，我们将这些数据称为late element

////////////////////////////// todo

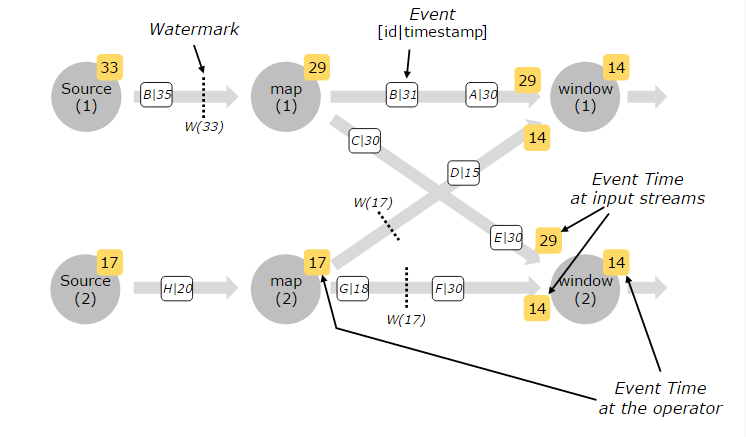
## Watermarks in Parallel Streams

Watermarks are generated at source functions, or directly after source functions. Each parallel subtask of a source function usually generates its watermarks independently. These watermarks define the event time at that particular parallel source.

As the watermarks flow through the streaming program, they advance the event time at the operators where they arrive. Whenever an operator advances its event time, it generates a new watermark downstream for its successor operators.

Operators that consume multiple input streams (e.g., after a keyBy(…) or partition(…) function, or a union) track the event time on each of their input streams. The operator’s current event time is the minimum of the input streams’ event time. As the input streams update their event time, so does the operator.

The figure below shows an example of events and watermarks flowing through parallel streams, and operators tracking event time.



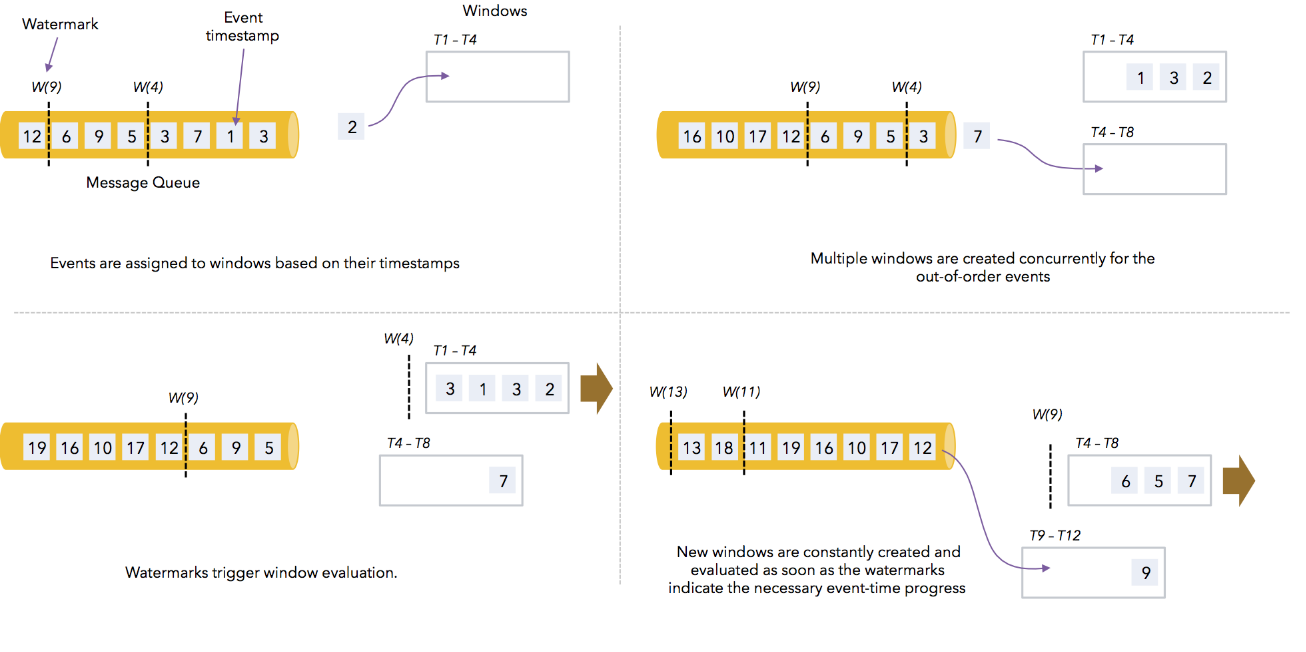
/////////////////////////////////

## Window+watermark

当我们将watermark运用到window机制中时，由于有watermark的存在，window的触发条件为：

对于正常数据和out of order数据：

1. 在[window\_start\_time,window\_end\_time)中有数据存在
2. 当前最大的Watermark时间 >= window\_end\_time



对于late element

event time < watermark （来一条就触发一条）