张一楠

2017年6月26日

Apache Beam 研究报告

Apache Beam（原名Google DataFlow）是Google在2016年2月份贡献给Apache基金会的Apache孵化项目，被认为是继MapReduce，GFS和BigQuery等之后，Google在大数据处理领域对开源社区的又一个非常大的贡献。Apache Beam的主要创新点是统一了批处理(batch)和流处理(streaming)的编程范式，特别是为无限、乱序的数据集处理提供简单灵活、功能丰富以及表达能力十分强大的SDK，其目标是在任何分布式计算引擎上都可以执行基于Beam开发的数据处理程序。

1.背景信息

1.1.基本术语以及相关概念

为了更好的理解本文，先介绍一些相关的基本概念。

1）流处理（Streaming）：一种针对无限数据集设计的数据处理引擎。

2）窗口（windowing）：将一个数据集划分为多个小数据块。

3）事件时间（Event Time）：事件产生的时间。

4）处理时间（Processing Time）：事件（数据）到达处理系统的时间。

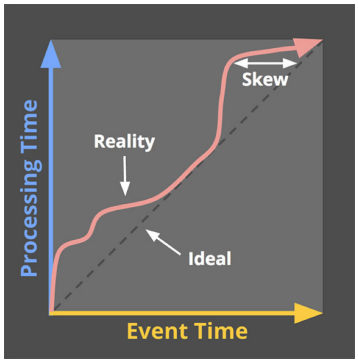


图 1 时间域映射

事件时间和处理时间在理想情况下应该是一致的，也就是说在事件发生时即为处理相关的数据。然而在现实情况下这两个时间往往是不同的（比如在传输数据时网络堵塞），早发生的事件还可能在后发生的事件之后到达系统，产生乱序，这个在分布式的数据处理系统中是十分常见的。对于这种情况，如何确定迟到数据，以及对于迟到数据如何处理是个需要考虑的问题。

5）水印（Watermarks）：用来标记一部分数据输入的完结（根据事件时间），一个值为X的水印表示的意思：所有事件时间在X之前的数据已经全被处理。水印可当做一个衡量数据处理进度的工具。

6）触发器（Triggers）：一种声明计算结果输出时间的机制，可以灵活的控制何时将处理结果输出，触发器为随着时间优化处理结果创造了可能。

7）累加模式（Accumulation）：累加模式用来制定同一窗口下多个结果（结果之间可能相互分离）的累加规则，根据使用场景来选择累加模式。

8）pipeline：封装整个数据处理任务，从始至终，所有Beam　driver程序都需要创建一个pipeline。形状是一个DAG（Directed acyclic graph），点是数据，边是变换

9）PCollection：pipeline每一步的输入和输出数据集

10）PTransforms：数据处理操作

1.2.数据处理模式

1）有限数据集

处理有限数据集我们只需将这个数据集放入批处理引擎（例如MapReduce）进行处理即可。

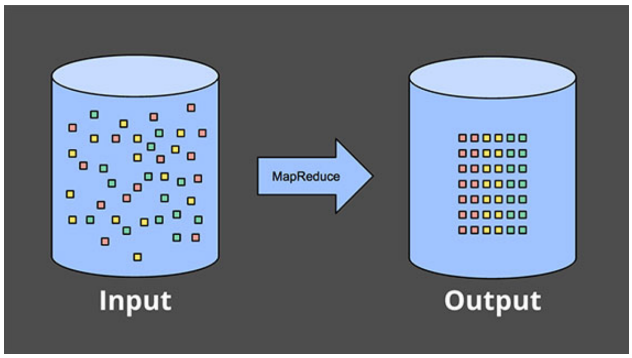


图 2 典型的批处理引擎处理有限的数据集

2）无限数据集（批处理）

虽然批处理引擎不是为无限的数据集设计的，但是也可以用它来处理无限的数据。

最常见的方式是将无限的数据进行固定窗口划分（比如按小时或者按天划分），得到一组大小相同（fixed-sized windows）的数据组然后把每个数据块当做单独的有限数据集进行批处理。

滑动窗口（sliding）：每个窗口有着相同的大小但是窗口的起始划分点不同，比如说按小时进行划分，间隔1分钟划分一次。滑动窗口实际上是固定窗口的超集（当每个窗口大小和时间段一致，得到的是固定窗口划分的结果）。

Sessions窗口：根据事件发生活跃期划分，在进入不活跃时期时终止。在Session之间往往有一段空隙隔开。Session有趣的一点是他的长度无法实现预估，要根据实际的数据得出。然而用传统的批处理引擎去计算sessions往往结果不够理想。

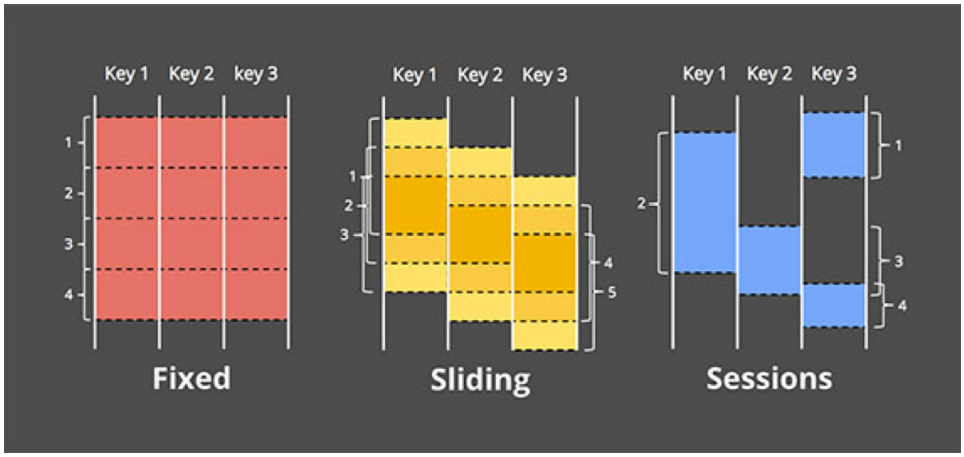


图 3 常见的窗口划分模式

3）无限数据集（流处理）

在现实世界中，数据往往是乱序，所以如果要根据事件发生的时间来分析数据的话必须依靠一些有效的手段让数据变的有序。并且每个数据的时间偏移量（skew）也会在一个很大的区间内，所以不能单纯的假设在某一个时间点已经获取了之前的所有数据。

要处理具有这种特征的数据以下几种方法是比较有效的：

时间无关处理（Time-agnostic）：这种方法主要用在数据的时间（包括事件发生时间以及处理时间）跟数据处理逻辑无关的场景。流处理引擎并不需要提供什么额外的帮助，只要确保数据能正常传递就可以了。批处理也能胜任这种处理方式，只需要将无限数据集分割成一系列有限数据块然后分开处理即可。常见的时间无关处理有：过滤筛选出某些特征的数据，将两组无限数据源进行内关联（inner-join）。

近似算法（Approximation algorithms）：处理无限数据源输出与期望结果相似的数据，这种方式的好处就是开销低，不足之处在于近似算法非常复杂很难根据需求去提出一个新的算法，并且因为得到的结果是近似的，应用场合有限。

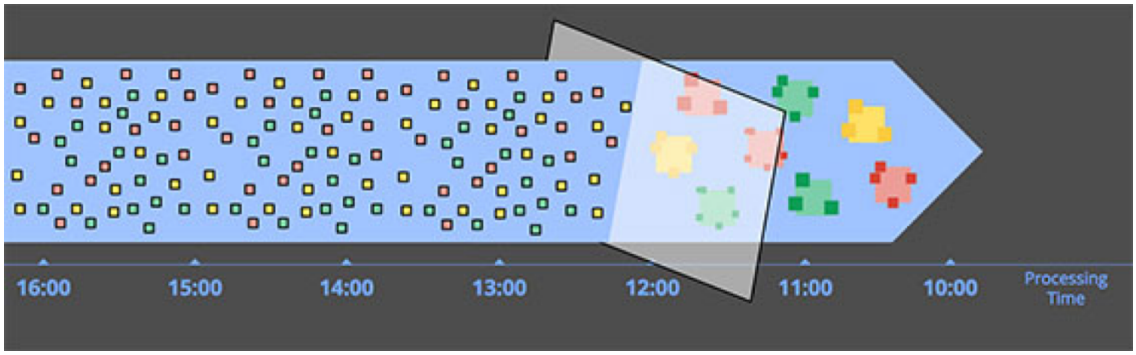


图 4 无限数据集计算近似结果

根据处理时间进行窗口划分（Windowing by processing time）：数据处理系统按数据到达系统的时间即处理时间对数据进行缓存，比如说用一个固定10分钟的窗口，系统会缓存所有该时间段内的数据，然后将这个窗口内的所有数据发给下游处理。这种方式的好处是简单，因为第一不用处理乱序的数据，第二容易判断数据的完整性（根据处理时间，系统可以很清楚的知道数据源输入是否完毕）。这种方式也有一个很大的弊端就是：如果在使用这种窗口划分的情况下，数据要按事件时间的顺序来处理，必须确保数据到达系统的时间与数据产生的顺序一样，然而这在现实中是很难保证的。

根据事件时间进行窗口划分（Windowing by event time）：如果要根据数据产生的时间处理数据块，这是最佳方法。有两个好处：第一确保了事件时间的正确性，第二可动态划分不同大小的windows，类似sessions。然而因为窗口的生命周期在大部分情况下比窗口本身的时间跨度长，所以这种方式有两个弊端：缓存数据是必须的，幸好今天的存储设备已经十分便宜了，这个弊端并不是很棘手。并且很多处理方式不需要存储整个数据集比如说求和以及求平均数，可以采用累加的方式进而缩小所需的存储空间。第二是数据完整性方面，常常只能得到一个预测的窗口结束时间，不能完全确定。需要另外的方法来辅助保证完整性。

1.3 现有的数据处理框架分析

随着分布式数据处理不断发展，新的分布式数据处理技术也不断被提出，业界涌现出了越来越多的分布式数据处理框架，从最早的Hadoop MapReduce，Apache Spark，Apache Storm，以及更近的Apache Flink，Apache Apex等。这些现代的数据处理框架可以将大规模，杂乱无章的数据处理成按一定规则，具有很大分析价值的数据。然而这些框架和系统在很多应用场景中还存在缺陷。

假设有这样一个场景：一个流媒体公司想要通过广告观看量来向广告商收取相应的费用。该公司的平台支持在线和离线观看广告两种模式，这家流媒体公司有以下需

求：通过数据分析来统计每天应向广告商收取多少费用；高效处理大规模的离线历史数据；了解他们的广告以何种频率、长度被观众收看，以及收看的内容喜好以及观众的地理位置分布。广告商也需要知道要交多少钱给这个流媒体公司。获取这些信息的效率是很关键的以便广告商调节自己的预算，变更策略，计划未来方向等等。这些数据的正确性很显然是至关重要的。

数据处理系统是非常复杂的，但是这个公司想让编程范式尽可能简单、灵活。而且，互联网触及世界各个角落，他们也需要这个新的系统有能力处理大规模、分布在全球各地的数据。计算出在这种场景适用的数据需要以下几个关键的数据：用户观看广告的时间点、长度以及对应的广告。

针对这种应用场景，现有的数据处理框架和系统有以下的不足：

批处理系统（MapReduce，FlumeJava还有Spark）有延迟的问题，因为他们都先要将数据收集再进行批处理。对于很多流处理系统很难在处理大规模数据的情况下具有良好的容错性，比如Aurora，TelegraphCQ，Niagara等。许多也缺少exactly-once语意比如Storm，Samza还有Pulsar，这影响到了数据的正确性。有些提供了窗口划分语意但是仅限于tuple窗口或者是processing-time-based窗口，比如Spark Streaming， Sonora，Trident。大部分提供了event-time-based窗口划分的系统，要不就是要求被处理的数据必须为有序的数据（SQLStream）或者是在窗口触发语意上有限制（Flink）。MillWheel和Spark Streaming具有良好的扩展性、容错率还有低延迟的特性，但是缺少high-level的编程范式。Lambda Architecture 可以满足上述的大部分要求，但是不够简易，因为需要建立和维护两个系统。Summingbird通过抽象化统一了批处理和流处理的接口，改善了这种维护系统的复杂性，但是在可执行的数据运算方式上存在局限性。

上述所提到的缺陷都不是无法解决的，随着系统的发展这些问题都会在以后被克服。然而，上述提到的框架和模型存在一个共同的缺陷就是他们都认为输入的数据会在某一个时间点完结。考虑到当今的大规模、高度无序的数据，这显然是不可能的。

2.Beam目前发展情况

2017年1月10日，Apache基金会宣布Apache Beam成为一个新的顶级项目，基于Apache V2许可证开源，目前主流的数据处理框架都已支持Beam包括Flink，Spark，Apex以及Google Cloud DataFlow，此外Apache Storm，Apache Hadoop，Apache Gearpump等执行引擎的支持也在讨论和开发中。目前支持Java和Python SDK。

Beam的发展历程：

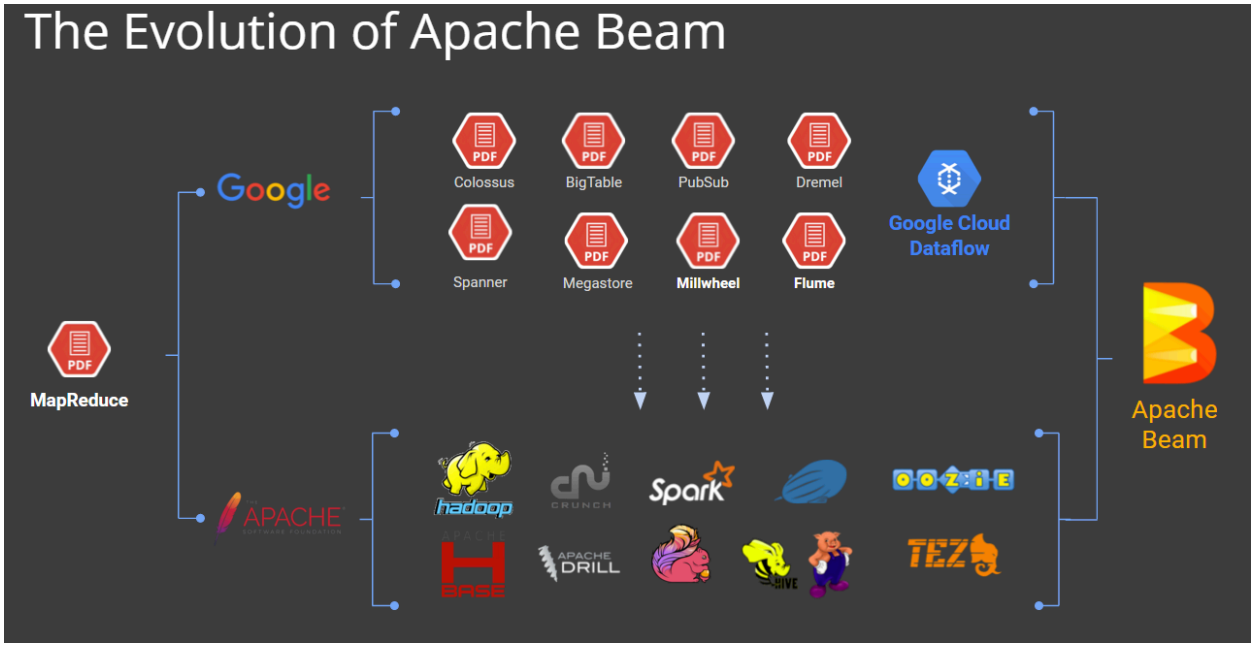


图 5 Beam的发展历程

3.Apache Beam 原理

3.1 Apache Beam 基本架构

Apache Beam主要由Beam SDK和Beam Runner组成，Beam SDK定义了开发分布式数据处理任务业务逻辑的API接口，生成的分布式数据处理任务Pipeline交给具体的Beam Runner执行引擎。

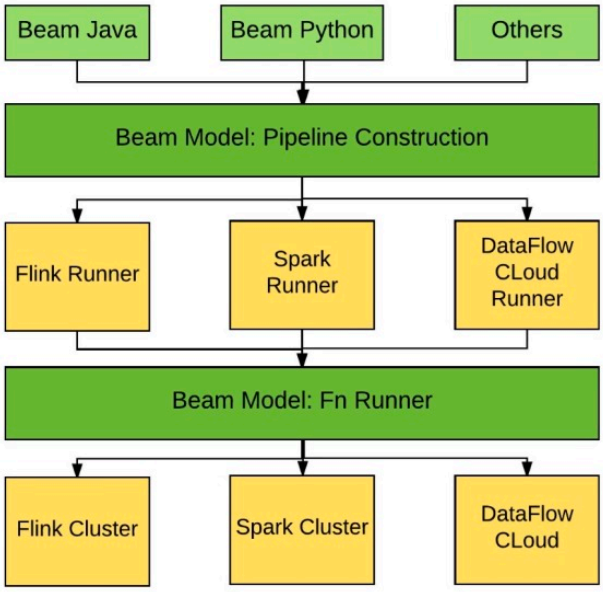


图 6 Apache Beam架构图

3.2 Beam Model解析

Beam Model处理的目标数据是无限、乱序的数据流，不考虑时间顺序或者有限的数据集可以看做是无限乱序数据流的一个子集。可以通过一个具体例子从以下4个问题来分析Beam：

1）对数据进行何种计算（What results are calculated?）

2）数据在什么范围中进行计算（Where in event time are results calculated?）

3）在什么时候将计算结果输出（When in processing time are results materialized?）

4）如何修正计算结果（How do refinements of results relate?）

3.2.1 对数据进行何种计算

对数据进行何种计算是由Pipeline中的操作符决定，可以对数据集里面每个单独的元素进行处理，也可以是将多个元素累加也可以是一个由多个数据变换组成的操作。

假设有一个简单场景，我们有一个PCollection<KV<String,Integer>>（一个由String和Integer组成的键值对集合的PCollection，String可以是队伍的名称，Integer可以是对应队伍取得的分数），假设我们从一个IO源中获取了raw data，然后将其变换成上述的键值对型的PCollection，然后进行按key求和操作，可以用以下代码实现：

```
PCollection<String> raw = IO.read(...);
PCollection<KV<String, Integer>> input=raw.apply(ParDo.of(new ParseFn());
PCollection<KV<String, Integer>> scores = input
  .apply(Sum.integersPerKey());
```

为了更好的描述实例，假设有一个键对应10个值，从下图可以看到，当pipeline获取到数据时，它会将这些数据累加，最后得到一个总和（51），下图中没有进行窗口划分，直接将整个事件时间里的所有值相加，所以这更加像一个典型的批处理。如果要处理无限的数据集，这种方式显然是行不通的，我们需要窗口划分，由此引入下一个问题，数据在什么范围中进行计算?

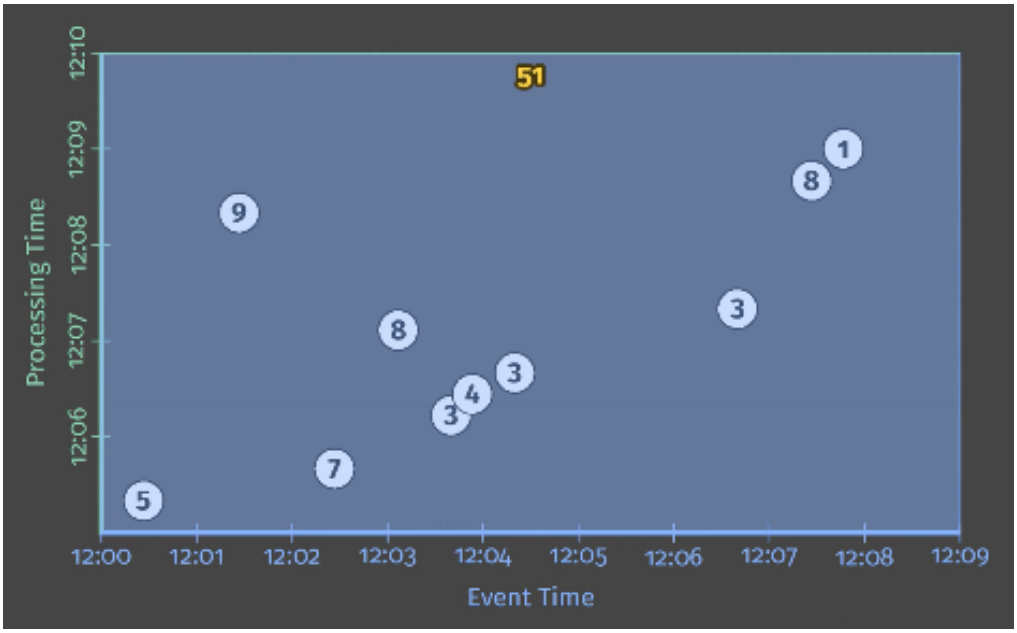


图 7 求和操作

3.2.2 数据在什么范围中进行计算

如果要将上述例子加上固定窗口（窗口大小为2分钟）划分，代码会变成如下（省去readIO步骤）：

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))
))
  .apply(Sum.integersPerKey());
```

Beam的优势就是统一了批处理和流处理的编程范式，所以可以在批处理引擎上执行上述代码构成的pipeline，结果如下图：

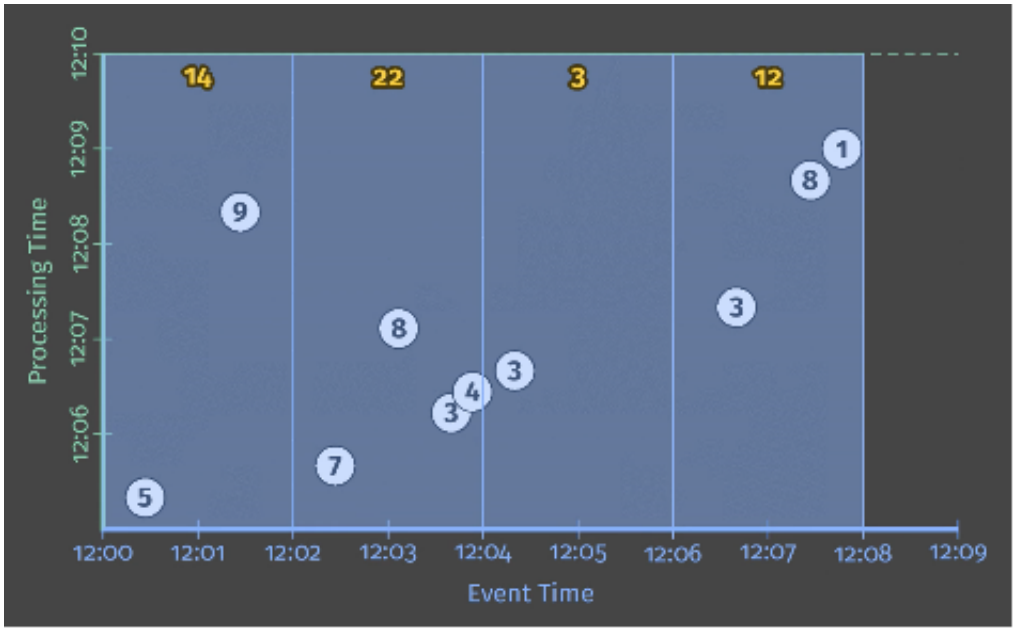


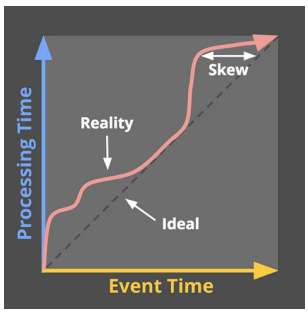
图 8 窗口化（窗口大小为2分钟）求和操作

跟之前一样当数据进入系统时进行累加，但这次产生了4个输出，每个输出对应一个2分钟的事件时间窗口。Beam 自动为每个时间窗口创建一个小的批处理作业，在处理时间纵轴 12:10 的时候触发计算。但是这样，只能等到最后的一个时间点（12:10）才能得到计算结果。如果想在更早的时间点得到时间窗口的统计，需要引入下一个问题。

3.2.3 在什么时候将计算结果输出

根据上面的窗口划分的批处理方式，可以得到正确的结果，但是得到结果的延迟较高，不够理想。改用流处理引擎是一个比较理想的解决方法，但是在批处理系统中我们可以明确的知道每个窗口的完结点，如果要在流处理引擎中处理无限的数据集，需要一种可行的确定数据完结的机制。

水印（Watermark）是系统用来衡量一个数据流中的进度和完结情况，对有限和无限的数据集都适用，但显然在无限数据集中更有用。再次看到这张时间域的图，红色的线即可看做是一个水印，它在处理时间上记录了相对应的事件时间的进度，也就是说假设在处理时间tp，我们可以根据红线求得对应的事件时间te，也就表明，当在处理时间tp时，所有在te之前产生的数据都已经被处理了。



水印有两种：

1）完美水印（Perfect watermarks）：如果我们知道了输入数据的所有信息，那么就可以构造一个完美的水印。在这种场景中，没有延迟的数据，所有数据到达系统的时间都是提前或者准时的。

2）预测水印（Heuristic watermarks）：对于大多数分布式数据源，要知道输入数据的所有信息是不现实的。这时，要利用已知的信息得出一个尽可能准确的预测水印。在许多场景中，这种水印也是很准确的，但是既然是预测性的了，那么肯定有时候会出错（会有迟到的数据）。

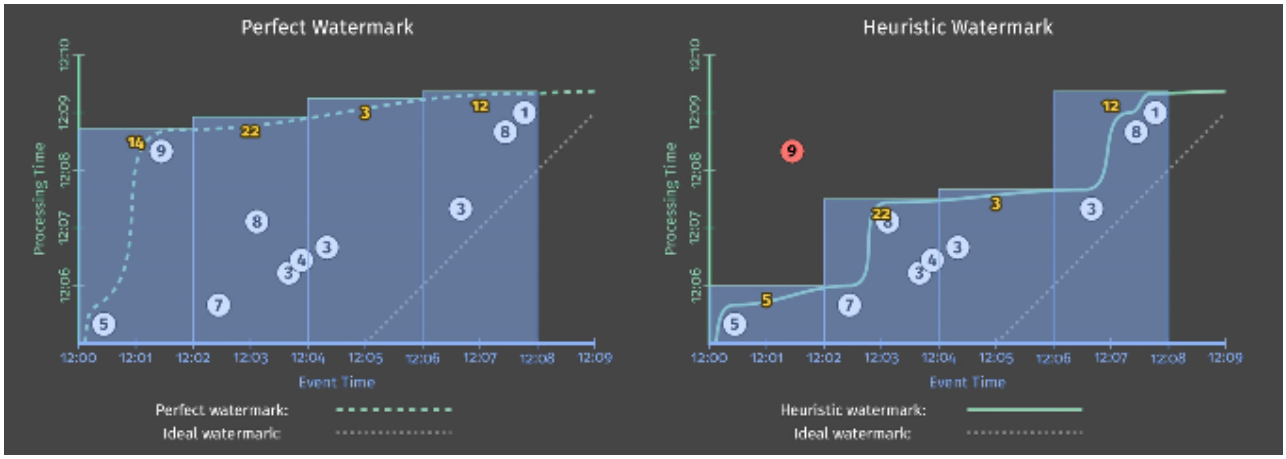


图 9 利用水印在流处理引擎中窗口化求和（左图为完美水印，右图为预测水印）

在图9的两种场景中，窗口的结果都是在水印穿过窗口的右边界后输出。两图的区别是使用预测水印的，没有计算标为红色的值为9的数据。一个数据的区别就使两种水印在形状上有很大的差别。从图9可以看出水印有2个缺点：

1）太慢： 当知道有还未处理的数据，水印的时间也会对应的延迟，这直接导致了输出结果了延迟。在左图中，因为值为9的数据的迟到，让后续的窗口的完结也推迟。在12：02 - 12：04这个窗口特别明显，从第一个数据的产生（值为7）到窗口结果的输出花了将近7分钟时间。

水印对帮助确定数据完结有很大的作用，但相应产生的结果在低延迟这方面做的不是很好。假设有一个长度为一小时或者一天的窗口，不会有人愿意等一小时或者一天来得到这个窗口的处理结果，这也是传统的批处理系统的痛点。如果窗口结果能随着时间一步步完善类似一个进化的过程并且到最后完结，这样会高效很多。

2）太快：如果水印标记完结的时间过早，可能会产生迟到数据，比如图9右图的第一个窗口，窗口过早的完结，导致输出了错误的结果5而不是14。所以预测水印有时会出错。所以单单依靠水印并不能保证数据的正确性。为了解决这个问题，触发器（Triggers）发挥作用的时候到了。

触发器（Triggers）在处理时间轴上决定了何时将窗口的计算结果输出（它可能会利用其它时间域的信息，比如说依赖于事件时间的水印）。可以用来触发触发器的信号有：

1）水印（Watermark　progress）：根据图8，我们可以看到当水印穿过了窗口的右边界，窗口结果输出触发。

2）特定的处理时间(Processing　time　progress)：可以用来当做日常周期性的信号，因为根据处理时间可以更好的统一处理数据并且延迟较低。

3）数据元素计数(Element counts)：在统计到特定有限的数据元素个数进行触发。

4）标点（Punctuations）：遇到一些特殊的字符（比如说EOF）时进行触发。

此外，还可以由多个单独的触发器组成一个具有更复杂逻辑的触发器，比如：Repetition：实现周期性触发的逻辑；Conjunction：And逻辑，只有所有的子触发器都触发了这个主触发器才会触发；Disjunctions：Or逻辑，任意一个子触发器触发，主触发器也触发；Sequences：按特定的顺序触发子触发器。

图9所示的过程其实已经用到了默认的触发器（根据水印触发），如果要额外写出来的话，代码如下：

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
                .triggering(AtWatermark()))
  .apply(Sum.integersPerKey());
```

触发器可以有效解决水印的两个缺点，在太慢的场景中，我们可以假设会有稳定的数据流输入，触发器按一定的周期（按处理时间）触发，获取数据的多少不会影响到触发的次数，这样可以快速的输出结果，而且在数据输入结束后，我们得到的触发不会影响结果。在太快的场景中，假设我们的水印是较为准确的，所以迟到的数据不会很常见，但如果收到了迟到的数据，我们需要对结果进行修改重新输出，我们可以设定在水印后如果收到1个数据元素（countof1)即更新结果再次输出新的结果。注意：这只适用于上述事例，在现实中需要应对不同的使用场景选择不同的触发器策略。

修改代码，在过水印前每隔1分钟触发一次EarlyFirings，在到达水印处触发一次on-time默认的触发器，在水印后如果收到额外的1个数据元素触发lateFirings，代码如下：

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
                .triggering(
                  AtWatermark()
                    .withEarlyFirings(AtPeriod(Duration.standardMinutes(1)))
                    .withLateFirings(AtCount(1))))
  .apply(Sum.integersPerKey());
```

在使用该代码后，可以得到下图的结果（左图为使用完美水印，右图为使用预测水印）：

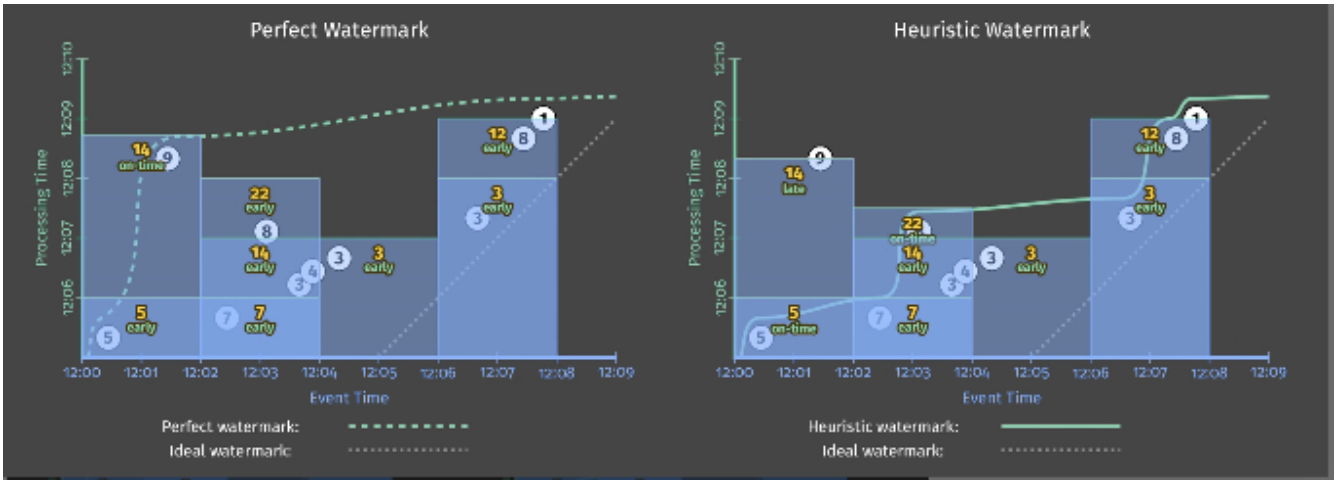


图 10 利用水印和触发器在流处理引擎中窗口化求和（左图为完美水印，右图为预测水印）

相比图9可以看到有两点提升：一是针对之前完美水印场景在12：02至12：04这个时间段水印太慢的缺点现在因为触发器可以提供提前更新结果（每间隔1分钟）的功能，第一次结果输出的时间由图9将近7分钟缩短到三分半，对于预测水印也有提升，现在两个场景都可以稳定的优化结果，可以从第二个窗口看出来，从7到14再到22，降低了原先结果输出的延迟。二是，图9之前预测水印在第一个窗口标记的时间（12：00，12：02）太早的场景，我们现在利用latefirings将它算入，得到了正确的结果。

有趣的一点是，在图9完美水印和预测水印两图相差较大，在图10两幅图看起来已经很相似，证明了触发器的作用。

现在两个场景还剩一种区别就是窗口的生命周期，在完美水印中我们清楚的知道在水印后不可能会再得到新的数据，所以我们可以把在水印后的所有状态、信息都舍去，但在预测水印的场景中，我们必须继续观察一段时间等待迟到的数据。现在我们的系统并不能知道我们需要保持这个等待的状态多久。我们可以手动的设定一个时间，当时间，修改代码如下：

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
                .triggering(
                  AtWatermark()
                    .withEarlyFirings(AtPeriod(Duration.standardMin
utes(1)))
                    .withLateFirings(AtCount(1)))
                .withAllowedLateness(Duration.standardMinutes(1)))
  .apply(Sum.integersPerKey());
```

3.2.4 如何修正计算结果

从图10可以看到触发器会把窗口“划分”成许多小窗口，我们需要选择适合应用场景的累加模式，累加模式有以下三种：

1）Discarding：当一个窗口中触发器成功触发了一次后，将之前存的状态舍弃，在这种模式下，每个小窗口都是独立的，这种模式在下游的处理者会自行将每次小窗口的结果汇总的时候适用。

2）Accumulating：如图10所示，每产生一个新的结果时，会在之前存的结果上直接累加，这种模式适用于新的出的结果可以随意覆盖之前结果的场景。

3）Accumulating&retracting：和Accumulating模式类似，只不过在产生新结果的时候额外多一个撤回值（等于之前的结果），这种模式使用于两个场景：一是新结果的key可能也改了，而下游处理着处理的时候不能简单的覆盖掉老的数据，所以需要撤回旧的key的数据，并且把新的数据加入到对应的key下。二是当用到动态窗口时候比如sessions，因为窗口的合并新的结果可能会代替多个窗口的结果，这样子很难决定替代哪些窗口，撤回合并之前所有旧结果窗口的结果解决了这一问题。

结合图10的第二个窗口（12：02，12：04）来看三种不同的模式：

	Discarding	Accumulating	Accumulating & Retracting
Pane 1: [7]	7	7	7
Pane 2: [3, 4]	7	14	14, -7
Pane 3: [8]	8	22	22, -14
Last Value Observed	8	22	22
Total Sum	22	51	22

Discarding：每个pane只包含自己小窗口中的值（7，7，8），所以最终结果并不是总和，需要自己累加每个小窗口的值得到22，代码以及效果图如下：

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
                .triggering(
                  AtWatermark()
                    .withEarlyFirings(AtPeriod(Duration.standardMinutes(1)))
                    .withLateFirings(AtCount(1)))
                .discardingFiredPanes())
  .apply(Sum.integersPerKey());
```

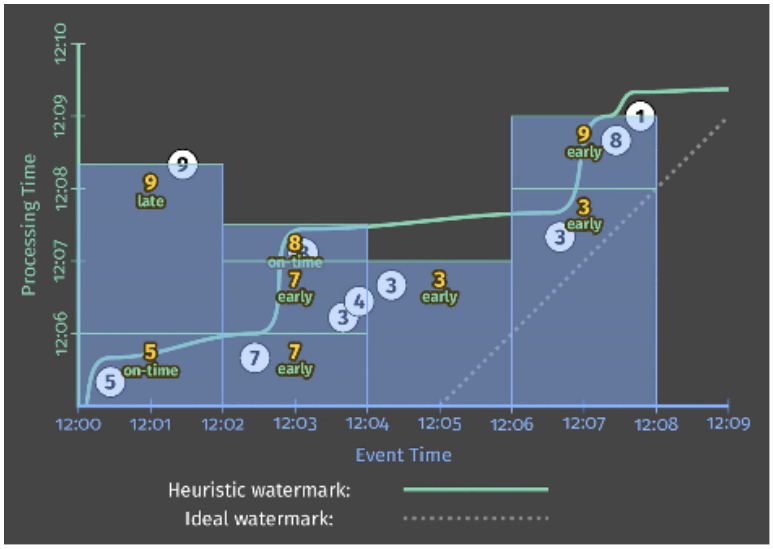


图 11 利用水印，触发器以及accumulating模式在流处理引擎中窗口化求和

Accumulating：每个小窗口在之前结果的基础上累加最终得到的结果即为总和22，效果图跟图10一致。

Accumulating&retracting：得到的最终结果也是22，代码以及效果图如下：

```
PCollection<KV<String, Integer>> scores = input
  .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
                .triggering(
                  AtWatermark()
                    .withEarlyFirings(AtPeriod(Duration.standardMinutes(1)))
                    .withLateFirings(AtCount(1)))
                .accumulatingAndRetractingFiredPanes())
  .apply(Sum.integersPerKey());
```

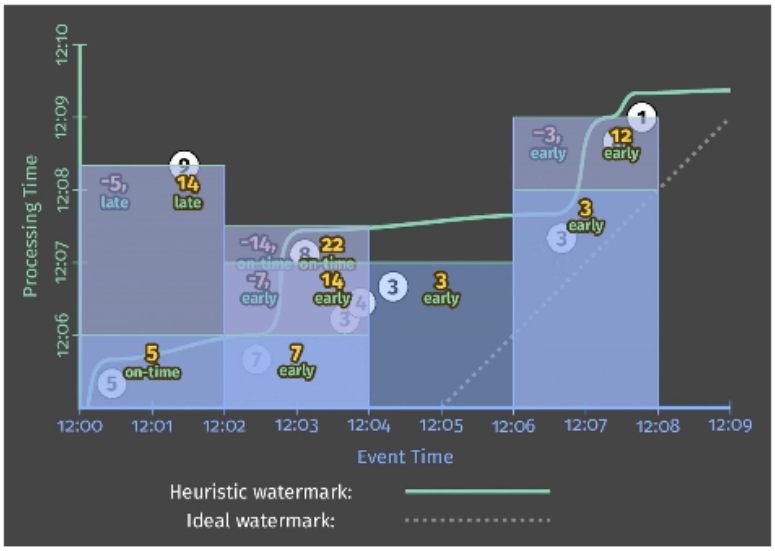


图 12 利用水印，触发器以及accumulating&retracting模式在流处理引擎中窗口化求和

3.3 Beam Runner介绍

Beam支持在多种底层执行引擎：

1）DirectRunner：在自己的本地电脑上运行。

2）ApexRunner：在Apache Hadoop YARN cluster上运行，是Hadoop体系（YARN&HDFS）的流处理框架，倾向Compositional的编程模型，框架提供一系列的operator，吞吐和延时等性能指标不错，并且支持动态DAG变更。

3）DataflowRunner：在Google Cloud Dataflow上运行，适合大规模持续性工作，动态平衡工作量，有一套完整的管理服务。

4）FlinkRunnder: 在Apache Flink cluster上运行，具备Native Streaming运行模型，同时又具备更高抽象层次API的流处理框架，适用场景广泛。同时支持批处理和流处理，吞吐量很大，延迟低，保证exactly-once语意，可以自定义内存管理。

5）SparkRunner：在Apache Spark cluster上运行，同时提供批处理和流处理管道，不错的容错率、安全性，原生支持Beam side-inputs。

4. 总结

随着分布式数据处理的不断发展，业界涌现出了越来越多的分布式数据处理框架，从最早的Hadoop MapReduce，到Apache Spark, Apache Storm，以及更近的Apache Flink, Apache Apex等。新的分布式处理框架可能带来更高的效率，更强大的功能，更低的延迟等，但用户切换到新的分布式处理框架的代价也非常大：需要学习一个新的数据处理框架，并重写所有的业务逻辑。解决这个问题的思路包括两个部分，首先，需要一个编程范式，能够统一，规范分布式数据处理的需求，例如，统一批处理和流处理的需求。其次，生成的分布式数据处理任务应该能够在各个分布式执行引擎上执行，用户可以自由切换分布式数据处理任务的执行引擎与执行环境。Apache Beam正是为了解决以上问题而提出的。Beam具有以下特性：

1）Unified: 用统一的编程模型,来适用于batch和streaming的情景.

2）Efficient: 在分布式环境下提供了workload balance和straggler问题的解决方案.

3）Portable: 用Beam 编程模型写出的数据处理pipeline可以在不同的引擎上运行,包括Apache Apex,Apache Flink,Apache Spark,和Google Cloud Dataflow.

Apache Beam的模型对无限乱序数据流的数据处理进行了非常优雅的抽象，“WWWH”四个维度对数据处理的描述非常清晰与合理，Beam模型在统一了对无限数据流和有限数据集的处理模式的同时，也明确了对无限数据流的数据处理方式的编程

范式，扩大了流处理系统可应用的业务范围。但同时需要注意的是，Apache Beam社区非常希望所有的Beam执行引擎都能够支持Beam SDK定义的功能全集，但是在实际应用中可能不一定，例如基于MapReduce的Runner显然很难实现和流处理相关的功能特性。所以能将Beam提供的编程范式在现存的所有Runner上执行还存在一定实现难度。