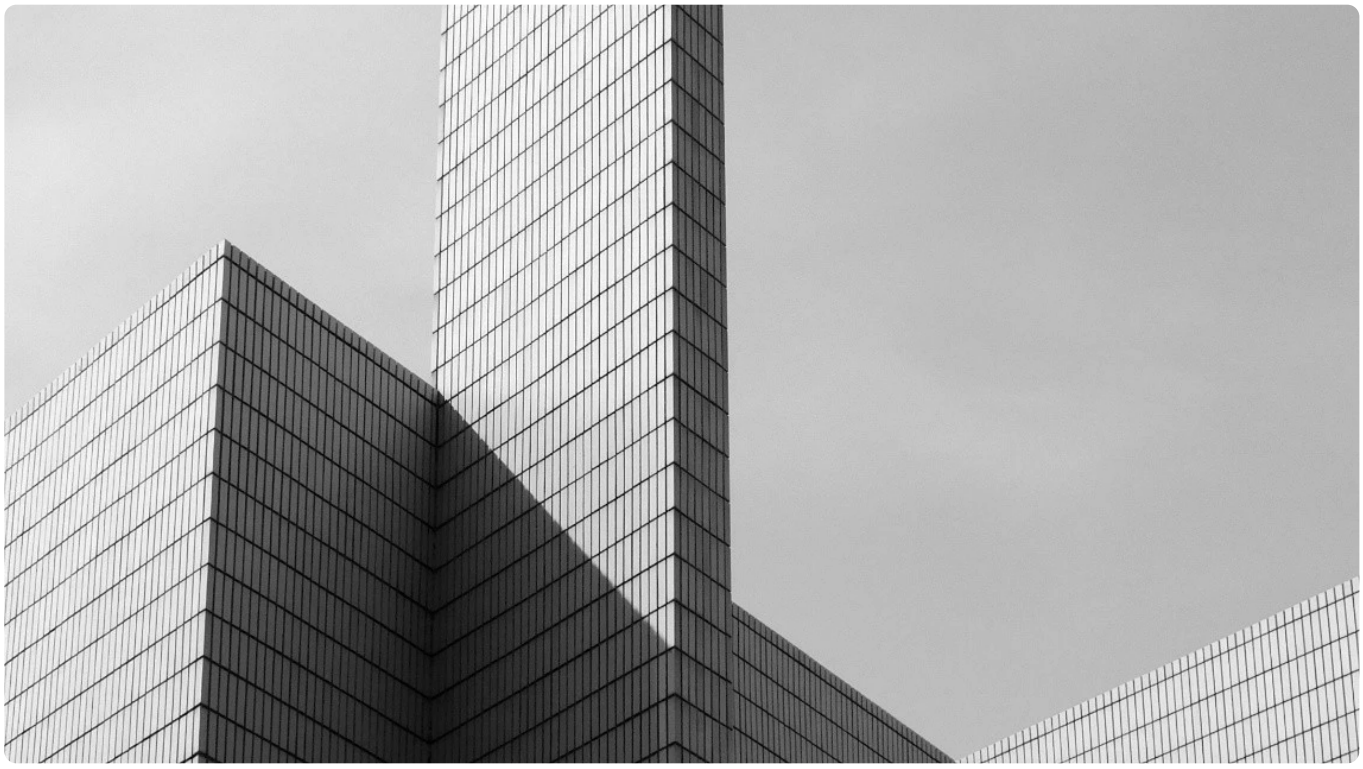
30 | Dataflow（二）：MillWheel，一个早期实现

2021-12-17 徐文浩

《大数据经典论文解读》 课程介绍 >



讲述：徐文浩
时长 20:06 大小 18.41M

你好，我是徐文浩。

上一讲里，我们通过一个简单的统计广告点击率和广告计费的 Storm Topology，看到了第一代流式数据处理系统面临的三个核心挑战，分别是：

> 数据的正确性，也就是需要能够保障“正好一次”的数据处理。
>
> 系统的容错能力，也就是我们不能因为某一台服务器的硬件故障，就丢失掉一部分数据。
>
> 对于时间窗口的正确处理，也就是能够准确地根据事件时间生成报表，而不是简单地使用进行处理的服务器的本地时间。并且，还需要能够考虑到分布式集群中，数据的传输可能会有延时的情况出现。

这三个能力，在我们之前介绍的 Kafka+Storm 的组合下，其实是不具备的。当然，我们也看到了，这些问题并不是解决不了，我们也可以在应用层撰写大量的代码，来进行数据去重、状态持久化。但是，一个合理的解决方案，**应该是在流式计算框架层面就解决这些问题**，而不是把这些问题留给应用开发人员。

围绕着这三个核心挑战，在 2013 年，Google 的一篇论文《MillWheel: Fault-Tolerant Stream Processing at Internet Scale》给我们带来了一套解决方案。这个解决方案，在我看来可以算是第二代流式数据处理系统。

那么今天，我们就来看看，在 2013 年这个时间点上，Google 的工程师是怎么解决这个问题的。希望你在学完这节课之后，能够掌握以下这些知识：

MillWheel 系统的整体架构是怎么样的，它的抽象模型和之前我们看过的 S4 和 Storm 有哪些异同之处。

除了简单地把中间数据持久化之外，流式数据处理系统还需要考虑哪些容错场景下的挑战，MillWheel 又是如何解决的。

下面，我们就先来看看 MillWheel 的系统架构是什么样子的。

MillWheel：S4 和 Storm 的组合模型

和 S4 以及 Storm 一样，MillWheel 的流式数据处理，同样是通过一个**有向无环图**来表示的。整个 MillWheel 的系统，是由这样几个概念组成的。

计算（Computation）和流（Stream）

首先是 Computation，用来作为有向无环图里面的计算节点。它里面包含了三个部分：

它“订阅”了哪些流，也就是消息输入的流向是什么；

它会输出哪些流，也就是消息输出的流向是什么；

它本身的计算逻辑，也就是进行数据统计，或者是数据过滤的逻辑代码。

所以很容易看出来，Computation 对应的就是 Storm 里面的 Bolt 或者 S4 里面的 PE。事实上，把 Computation 和接下来的 Key 组合在一起，其实和 S4 里面的 PE 没有什么差

别。

```
computation SpikeDetector {
  input_streams {
    stream model_updates {
      key_extractor = 'SearchQuery'
    }
    stream window_counts {
      key_extractor = 'SearchQuery'
    }
  }
  output_streams {
    stream anomalies {
      record_format = 'AnomalyMessage'
    }
  }
}
```

> 论文中的图 3，对于一个 Computation 的定义。

键（Key）

然后是 Key，在 MillWheel 的系统里，每一条消息都可以被解析成（Key, Value, TimeStamp）这样一个**三元组**的组合。而我们在前面也看到了，一个 Computation 可以针对输入的消息流，定义自己的 key_extractor。这一点，比起 Storm 和 S4 其实是有所不同的。

在 Storm 和 S4 里，同样的消息，如果我们要根据不同的字段进行维度划分，分发给不同的 PE 或者 Bolt。那么，在抽象层面，我们其实是发送了两个不同的消息流。**而在 MillWheel 里，则是一个相同的消息流，被不同的两个 Computation 订阅了，只是两个 Computation 可以有不同的 key_extractor 而已。**这样，我们在系统的逻辑层面就可以复用同一个流，而不需要有两个几乎是完全相同，只是使用的 Key 的字段不同的流了。

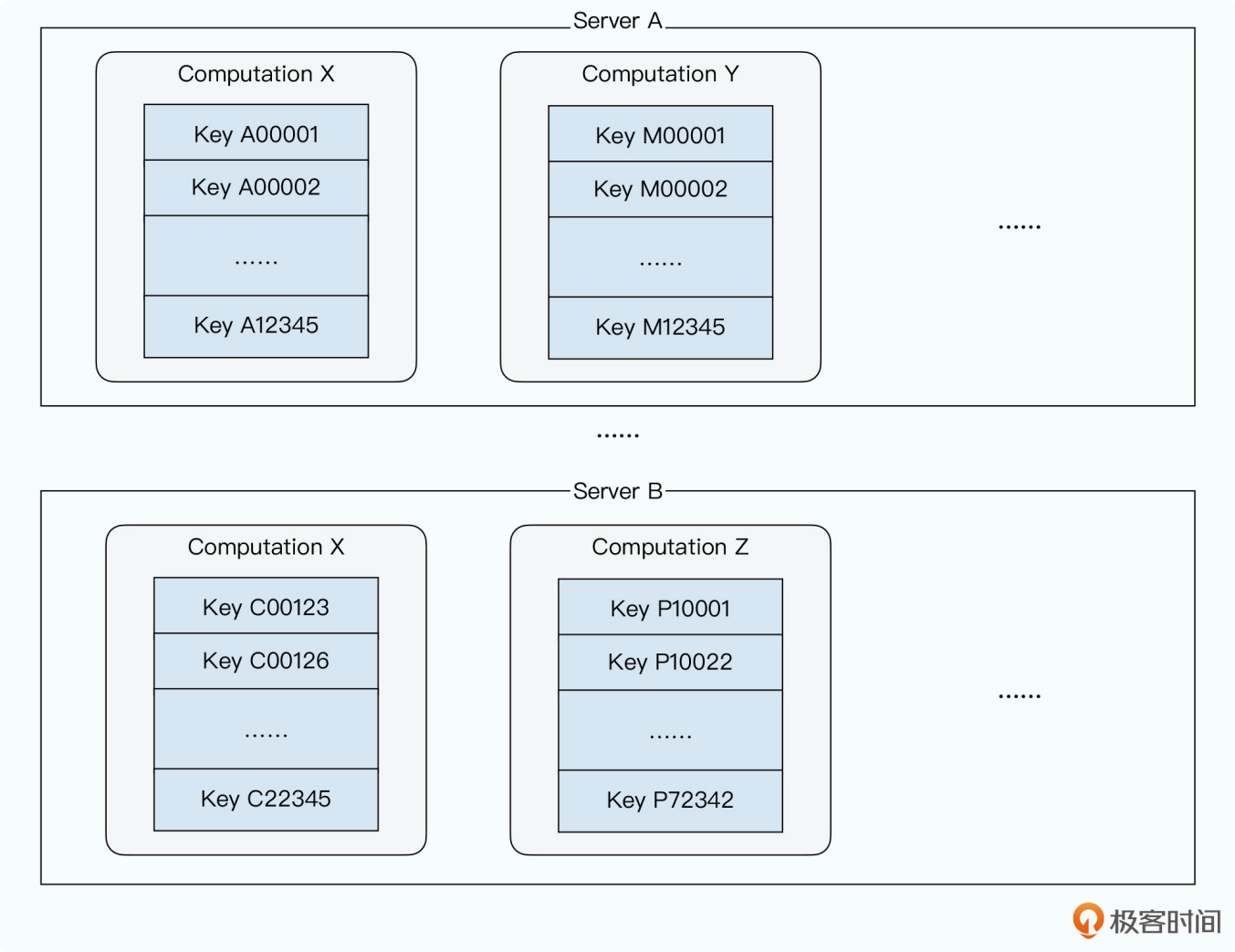
而流自然也很容易理解，它就是不同 Computation 之间的数据流向，一个 Computation 可以订阅多个数据流。每一个被订阅的数据流，就在两个 Computation 之间形成了一条边。

在实际的操作层面，Key 也是 MillWheel 系统里面用来进行计算的**唯一单元**。也就是一个 Computation 的实现里面，获取到的都是同一个 Key 的状态。就拿上节课我们举过的广告点击率统计的例子来说明，每一个广告 ID 其实就是一个 Key，在一个 Computation 里，你获取到的日志记录，都是这个广告 ID 的日志记录。

这个设计，其实和我们之前看过的 S4 的 PE 设计是一样的。可以说，**Computation + Key 的一个组合，就是一个 PE**。而这个 Computation + Key 的组合，是可以在不同机器之间被调度的。也就是说，我们可以因为某一台服务器的负载太大了，来主动把这台服务器上的一些 Computation + Key 的组合，迁移到另外一台服务器上。

不过，和 S4 不一样的是，在实际实现上，MillWheel 并没有真的把一个 Computation + Key 变成一个对象来处理。事实上，在 MillWheel 里，每一个 Computation 里的 Key 是像 Bigtable 里的 Tablet 一样，分成一段一段的。实际根据负载进行调度的时候，调度的也是这一整段的 Key。而这个实现，也就**避免了之前 S4 对应的 PE 对象过多的问题**。

从这个角度，MillWheel 的系统逻辑其实更像是 Storm，而 Computation + 一段 Key 的组合，其实就是一个 Bolt，需要处理某一段 Key。



每一个服务器上，有很多个Computation的进程，每个进程管理某一个Computation的某一段Key

低水位（Low Watermarks）和定时器（Timers）

无论是论文里给出的进行异常搜索量的检测，还是上节课我们举过的广告点击率的统计例子，MillWheel 这样的流式系统，都要面对实际的事件发生的时间，和我们接收到数据的时间有差异的问题。

MillWheel 需要有一个机制，能够让每个 Computation 进程知道，某个时间点之前的日志应该都已经处理完了。所以，它引入了**低水位**这个概念，以及 **Injector** 这个模块。

前面我们讲解 Computation 的时候已经看到了，我们的每一条消息，都会被解析成（Key，Value，TimeStamp）这样一个三元组。这里面的 TimeStamp，其实就是我们需要的事件发生的时间，我们就是根据这个时间戳，来解决这个时间差异的问题的。

在 Computation 处理完一个消息，往下游发送新消息的时候，也需要为新消息创建一个时间戳。这个时候，创建的时间戳不能早于你处理的消息的时间戳。如果你希望后续的数

据处理，都是基于最早事件发生的时间点来进行数据统计，那么最好的办法，就是直接复制输入消息的时间戳。

MillWheel 引入的低水位是这样一个概念，在某一个 Computation A 里，我们可以拿到所有还没有处理完的消息里面，最早的那个时间戳。这个没有处理完的消息，包括了还在消息管道里面待传输的消息，也包括已经在 Computation 里存储下来的消息，以及处理完了，但是还没有向下游发送的消息。**这个最早的时间戳，就是一个“低水位”。**这个“低水位”，其实就是告诉了我们，当前这个 Computation 的计算节点里，哪个最早的时间点还有消息没有处理完。

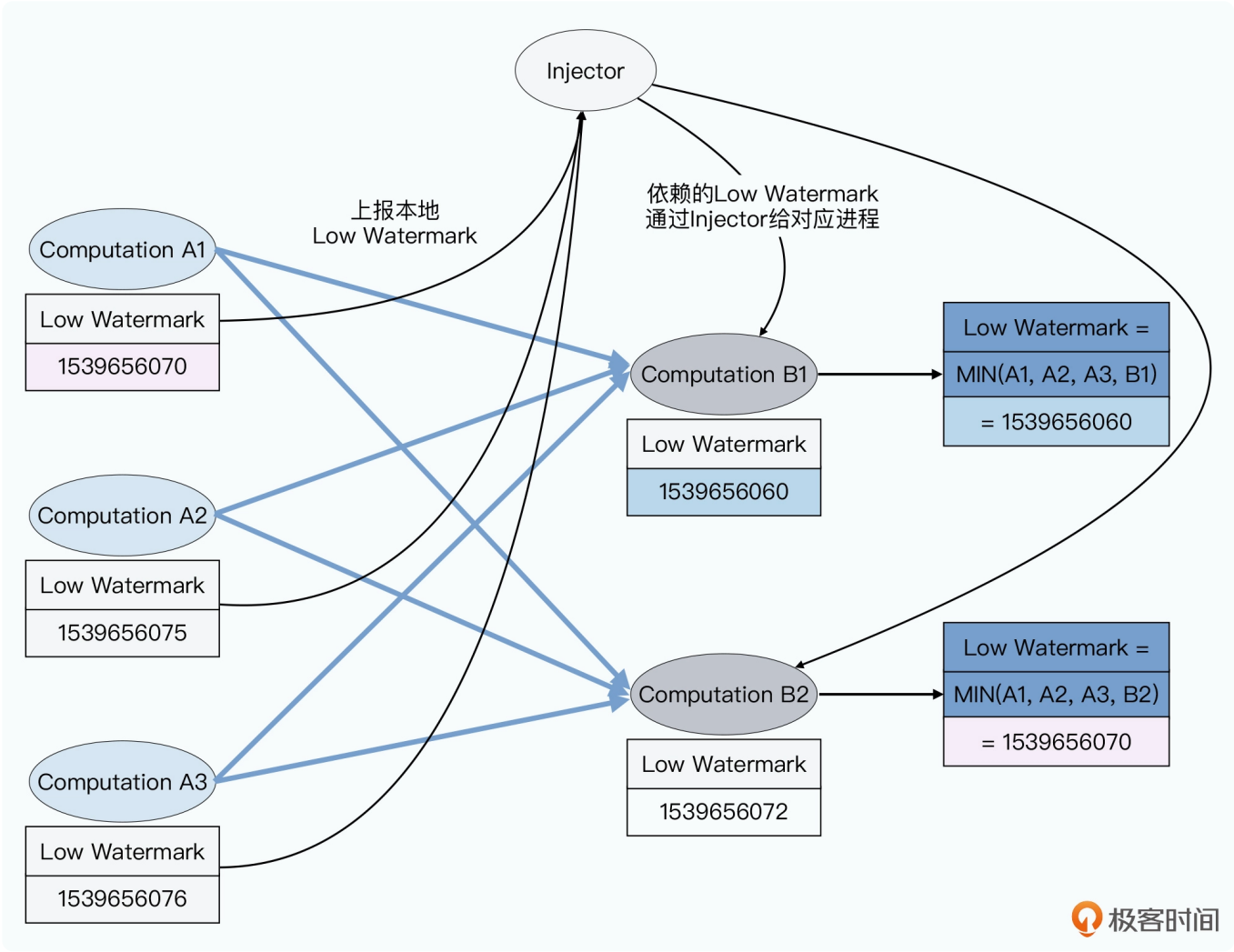
而这个 Computation A，可能还订阅了很多上游的其他 Computation。此时此刻，那些 Computation，也会有一个同样的时间戳。那么，本质上，A 的低水位，就是它和它上游的低水位中，时间戳最早的那一个。也就是对应着论文里 4.5 部分里的这个公式：

复制代码

```
1 min(oldest work of A, low watermark of C:C outputs to A)
```

如果我们的每个 Computation 的进程，能够知道当前自己这个 Computation 的低水位是什么，那么很多问题就变得简单了。获取到了当前 Computation 的低水位，我们就能决策是否应该进一步等待更多的消息，以获得准确的统计数据，还是现有的数据已经是完整的，我们可以把结果输出出去了。

那么，MillWheel 是这么做的：每一个 Computation 进程，都会统计自己处理的所有日志的“低水位”信息，然后上报给一个 Injector 模块，而这个 Injector 模块，会收集所有类型的 Computation 进程的所有低水位信息。接着，它会通过 Injector，把相应的水位信息下发给各个 Computation 进程，由各个 Computation 进程自己，来计算自己最终的低水位是什么。



理解每一个计算节点，都会根据本地以及它依赖的上游Computation计算出自己当前的Low Watermark，以理解目前消息处理的进度

每一种类型的 Computation，都会有一个自己的水位信息。同一个 Computation 下，不同进程的水位信息也是不同的，因为它自己处理的消息的进度可能不一样。而不同类型的 Computation，水位信息就是不同的，因为整个数据流的拓扑图可能会很深。很有可能，前面几层 Computation 的数据已经都处理到 12 点 05 分了，但是后面几层 Computation 才处理到 12 点 01 分。如果我们让整个拓扑图，都用同一个水位信息，就意味着前面几层统计结果的输出的延时会变大。

而有了这个水位信息，我们统计某一个时间段的统计数据，就可以做到基本准确了。

在论文里，Google 给出的经验数据，是只有 0.001%（十万分之一）的日志，在考虑了水位信息之后仍然会因为来得太晚而被丢掉。但实际上，由于所有数据都是持久化的，即使这些消息来得太晚，我们仍然可以纠正之前的统计数据。

并且这些水位信息的计算，以及根据水位信息来决定数据是否计算完成了，并不需要应用开发人员关心，而都是系统内建的。

对于应用开发人员来说，MillWheel 提供了一组**定时器**（Timer）的 API。根据日志里的时间戳，你能拿到这条日志对应的时间窗口是哪一个。然后把对应的数据更新，再根据时间窗口，设置到对应的 Timer 上。系统自己会根据水位信息，触发 Timer 执行，Timer 执行的时候，会把对应的统计结果输出出去。你可以对照着论文里图 9 的代码来看一看。

复制代码

```
// Upon receipt of a record, update the running
// total for its timestamp bucket, and set a
// timer to fire when we have received all
// of the data for that bucket.
void Windower::ProcessRecord(Record input) {
  WindowState state(MutablePersistentState());
  state.UpdateBucketCount(input.timestamp());
  string id = WindowID(input.timestamp())
  SetTimer(id, WindowBoundary(input.timestamp()));
}

// Once we have all of the data for a given
// window, produce the window.
void Windower::ProcessTimer(Timer timer) {
  Record record =
    WindowCount(timer.tag(), MutablePersistentState());
  record.SetTimestamp(timer.timestamp());
  // DipDetector subscribes to this stream.
  ProduceRecord(record, "windows");
}
// Given a bucket count, compare it to the
// expected traffic, and emit a Dip event
// if we have high enough confidence.
void DipDetector::ProcessRecord(Record input) {
  DipState state(MutablePersistentState());
  int prediction = state.GetPrediction(input.timestamp());
  int actual = GetBucketCount(input.data());
  state.UpdateConfidence(prediction, actual);
  if (state.confidence() > kConfidenceThreshold) {
    Record record = Dip(key(), state.confidence());
    record.SetTimestamp(input.timestamp());
    ProduceRecord(record, "dip-stream");
  }
}
```

> 论文中的图 9，实际的业务代码中，通过 Timer 的 API 来定时更新数据，具体 Timer 的触发，由框架根据水位信息自行判断。Windower::ProcessRecord 函数处理消息，并把消息关联给 Timer，Windower::ProcessTimer 根据水位信息会在合适的时候触发，然后我们可以对应计算统计信息并输出。

Strong Production 和状态持久化

无论是在 Timer 还没有触发时，我们统计到的中间阶段的数据结果，还是我们已经确定要向下游发送的计算结果，都需要持久化下来。这个是为了整个 MillWheel 系统的容错能力，以及我们可以“迁移”某段 Computation + Key 到另外一个服务器上。

所以，MillWheel 也封装掉了整个的数据持久化层，你不需要自己有一个外部数据库的连接，而是直接通过 MillWheel 提供的 API，进行数据的读写。

复制代码

```
// Upon receipt of a record, update the running
// total for its timestamp bucket, and set a
// timer to fire when we have received all
// of the data for that bucket.
void Windower::ProcessRecord(Record input) {
  WindowState state(MutablePersistentState());
  state.UpdateBucketCount(input.timestamp());
  string id = WindowID(input.timestamp())
  SetTimer(id, WindowBoundary(input.timestamp()));
}

// Once we have all of the data for a given
// window, produce the window.
void Windower::ProcessTimer(Timer timer) {
  Record record =
    WindowCount(timer.tag(), MutablePersistentState());
  record.SetTimestamp(timer.timestamp());
  // DipDetector subscribes to this stream.
  ProduceRecord(record, "windows");
}
// Given a bucket count, compare it to the
// expected traffic, and emit a Dip event
// if we have high enough confidence.
void DipDetector::ProcessRecord(Record input) {
  DipState state(MutablePersistentState());
  int prediction = state.GetPrediction(input.timestamp());
  int actual = GetBucketCount(input.data());
  state.UpdateConfidence(prediction, actual);
  if (state.confidence() > kConfidenceThreshold) {
    Record record = Dip(key(), state.confidence());
    record.SetTimestamp(input.timestamp());
    ProduceRecord(record, "dip-stream");
  }
}
```

比如，论文中的图 9 里，我们是通过调用 state.UpdateBucketCount() 来更新统计数据，通过 ProduceRecord() 来输出结果。这些数据更新，都会被持久化下来。而 MillWheel 能够做到这一点，很大程度上依赖了强大的基建，也就是自家的 Bigtable 和 Spanner。

每一个 Computation + Key 的组合，在接收到一条消息的处理过程是这样的：

第一步，自然是**消息去重**，这个可以通过分段的 BloomFilter 来解决，我们在上节课已经看到过了。

第二步，就是**处理用户实现的业务逻辑代码**，在这些代码中，所有产生的更新，无论是给 Timers 还是 State，或者是 Production，都被视作是对于“状态”的变更。

第三步，这些**状态的变更，都会被一次性提交给后端的存储层**，也就是 Bigtable 或者 Spanner 里。

第四步，因为这些更新都已经持久化了，所以系统会**发送 Acked 消息**给上游发送消息的 Computation。这里的 Acked 机制和 Storm 的 Acked 机制是类似的，能够确保消息不会丢失，没有 Acked 的消息可以重发，并且会在第一步被去重做到“正好一次的数据处理”。**不同的地方在于**，MillWheel 里，因为处理完的消息会被持久化，所以不需要等消息在整个有向无环图里都处理完，才在起点清理掉消息。每一层可以单独回收掉下游的第一层已经处理完的消息。

最后，则是我们向下游发送的消息会被发送出去。

在第五步的消息对外发送之前，我们会把要发送的记录写入到 Bigtable/Spanner 里先持久化下来。这个被持久化的内容，在 MillWheel 中被称为是检查点（Checkpoint），正是有了这一步，整个 MillWheel 系统才有了容错能力和在线迁移计算节点的能力。而为了性能考虑，在实践上，MillWheel 会把多个记录的操作，放在一个 Checkpoint 里。

这个 Checkpoint，类似于数据库里的预写日志（WAL）。这个时候，即使我们的节点挂掉了，或者我们想要在线迁移计算节点，我们也可以在另外一台服务器上，把这个 Checkpoint 读出来，重新向下游发送就好了。

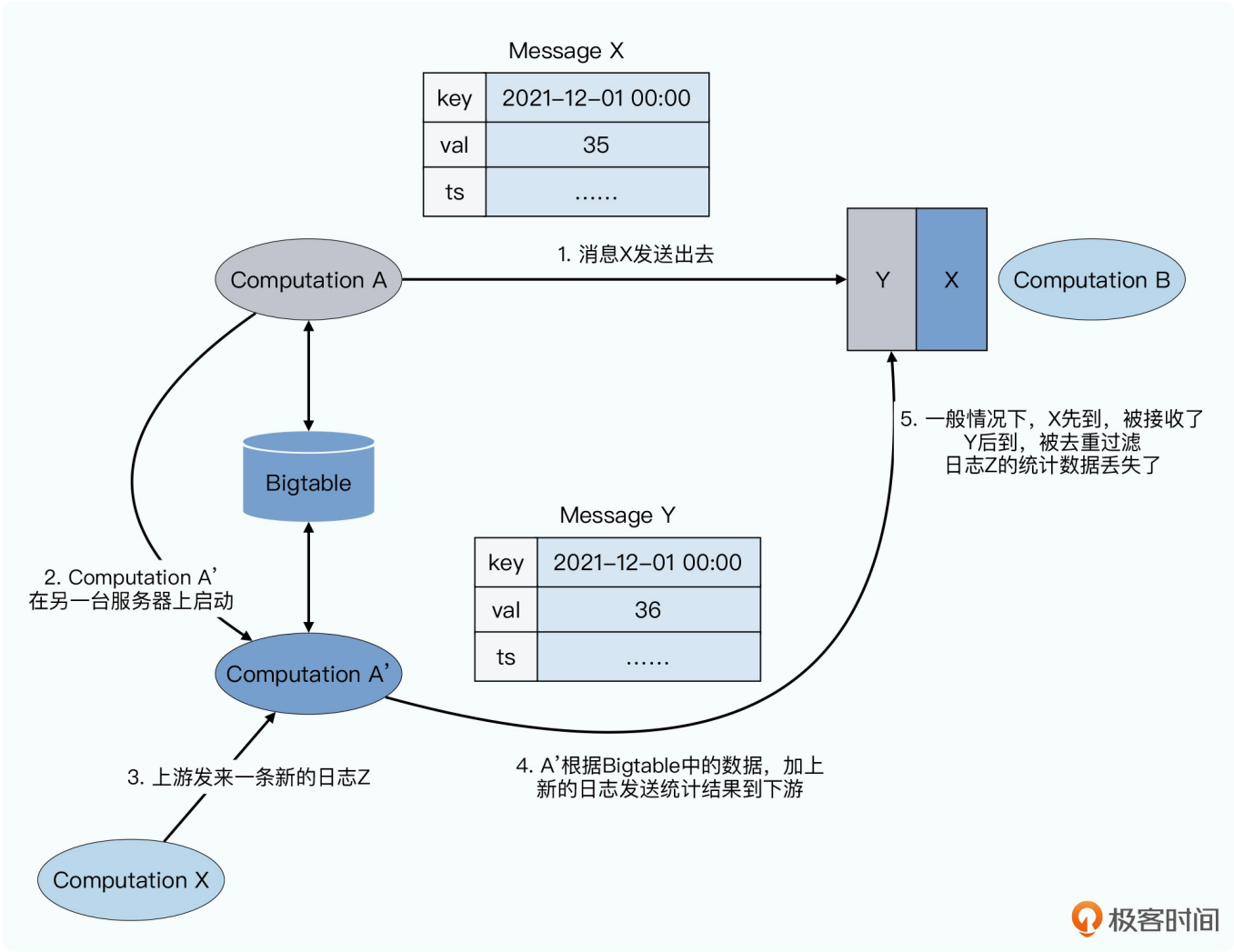
你可能要问，我们为什么需要这个 Checkpoint 呢？我们已经在第三步，把所有的中间结果记录下来了呀。如果节点挂掉，在另外一台服务器重新启动计算进程的时候，我们重新从中间结果里获取数据，然后发送给下游不就好了么？

那我们来看这样一个例子：

> 我们有一个 Computation，它按照服务器的本地时间，也就是**数据的处理时间**，按照时间窗口统计数据并下发。比如，每 5 分钟发送自己接收到的日志总的数量到下游。
>
> 每一条日志过来的时候，我们都会按照前面的第三步，去更新统计数据。Timer 会根据实际时间（Wall Clock），而不用考虑日志里的时间戳或者水位，每 5 分钟触发一次，发送统计结果到下游。
>
> 在某一个 Timer 触发的时候，我们生成了计算结果，想要向下游发送，但是这个时候，我们的计算节点挂掉了。我们的消息发送出去了，但是下游有没有收到我们并不知道。这是我们发送的第一条消息 X。
>
> 这个时候，我们重新在另外一个节点启动了这个 Computation，但是，此时此刻，有一条新的日志进来了。**这个时候，我们还在同一秒钟，所以还是同一个时间窗口，我们会对应更新统计数据。**
>
> 然后，我们会向下游，发送一个包含了新日志的统计结果。这个新的统计结果，和我们的节点挂掉之前想要发送的消息的时间窗口是一样的，但是值不一样。这是我们发送的第二条消息 Y。
>
> 但是由于网络传输是乱序的，我们其实并不知道是 X 会先到下游，还是 Y 会先到下游。如果 X 先到，Y 后到，Y 因为 X 被去重了，那么下游实际计算过程中就丢了一条记录了。

所以，为了避免这个情况，MillWheel 采取了一个简单粗暴的方式。那就是我们对于下游要发送的数据，会**先作为 Checkpoint 写下来**。之后才是简单地重放 Checkpoint 的日志，避免这样基于时间点的隐式依赖，导致不能做到数据层面的一致性。

而这个 Checkpoint 的策略，在 MillWheel 里被称之为 **Strong Production**，这个 Strong 要突出类似于强一致性（Strong Consistency）里的 Strong 的这个概念。也就是 MillWheel 的数据处理，虽然支持乱序的数据，但是所有的输入数据，是严格不会重复，也不会丢弃的。



没有Strong Production，下游的Computation B对于接收到的消息，很难进行去重过滤
无论是用Y覆盖X，还是以X已经收到，去重过滤Y都不合适

僵尸进程和租约问题

事实上，在容错上，我们不仅要考虑这样的时间窗口问题，还需要考虑**僵尸进程**的问题。MillWheel 有一个中心化的 Master 集群，进行负载均衡的调度。并且在节点挂掉的时候，也是由这个 Master，在其他的服务器上去启动新的 Computation 的进程。

但是，Master 判断节点挂掉，并不意味着节点进程真的挂掉了，完全可能是因为网络分区造成的。我们的 Master 节点连不上某一个 Computation 所在的服务器，所以在别的服务器上启动了一个新的 Computation 进程。那这个时候，我们就有可能有两个 Computation 进程，在管理同一段 Key。其中旧的 Computation 进程，其实就是一个我们没能够杀掉的僵尸进程。

虽然，这个时候我们上游的数据，会被 Master 调度，往新的 Computation 进程里发送。但是，旧的 Computation 进程，完全可能有一个几秒钟之后会触发的 Timer，往下游写

入数据，或者往 Bigtable 这样的持久层里更新数据。这样，我们一样会面临数据的不一致性问题。

MillWheel 对这个的解决方案，其实和我们之前看过的 GFS/Bigtable 是类似的，那就是每个 Computation 的进程都会有一个**租约**。每次数据写入，都会带上这个租约的 Token。当 MillWheel 启动一个新的进程来进行容错处理的时候，老的进程的租约就被作废了。

事实上，所有的分布式系统都需要有类似的机制，来确保任何一个 Key 的数据，同时只有一个人能写，也就是所谓的 Single-Writer（写入者）的机制。

Weak Production 和幂等计算

不过，并不是所有的计算都需要 Strong Production，以及对于消息进行去重的机制的。毕竟，消息去重和 Checkpoint 都是有大量开销的，需要消耗我们更多的计算资源。比如，如果我们有一个 Computation A 只是简单地对消息按照某个字段进行过滤，然后在下游的 Computation B 进行数据统计。

那么，在 A 这里，我们不需要进行数据去重，去重只需要在 B 这里做一次就好了。或者，我们想要统计获得某个时间段里的最大值或者最小值，也不需要进行去重，因为即使同一条记录出现两次，我们的最大值和最小值也不会发生变化。

同样，我们也不需要在 Computation A 这一层使用 Strong Production，只需要它的下游做了 Strong Production 就好了。**在 A 这一层出现问题，我们只需要让上游简单地重发消息就好了，因为在 Computation A 这一层是无状态的，没有所谓的中间计算结果，所以持久化状态是一种浪费。**

所以，MillWheel 允许我们去关闭去重机制，以及 Strong Production 机制。关闭之后的 Computation 节点，被称之为是 **Weak Production**。

小结

好了，到这里，我们对于 MillWheel 这个系统的方方面面就已经了解得很清楚了。MillWheel 采用了和 S4 以及 Storm 一样的有向无环图的逻辑结构。为了解决容错问题，

MillWheel 接管了数据的存储层，而计算的中间结果以及输出的内容，都是通过调用框架提供的接口，被 Bigtable 或者 Spanner 存储下来了。

为了解决**数据去重**，MillWheel 通过为每一个收发的记录都创建了一个唯一的 ID，然后在每个 Computation 的每一个 Key 上，都通过 Bloomfilter 对处理过的消息进行去重，确保所有的操作都是幂等的。

而为了解决**流式计算的容错和扩容问题**，MillWheel 会通过 Strong Production 这个方式，对于所有向下游发送的数据先创建 Checkpoint。这个 Checkpoint，其实就是类似于数据库里面的 WAL，通过记录日志的方式，确保即使计算节点挂掉了，也能够在新起来的计算节点上重放 WAL，来重新向下游发送数据。

然后，为了解决**事件创建时间和处理时间之间的差异**，MillWheel 引入了一个独立的 Injector 模块。Injector 模块，一方面会收集所有计算节点当前处理数据的进度，另一方面也会反馈给各个节点，当前数据处理的最新“低水位”是什么。这样，对于需要按照时间窗口进行统计分析的数据，我们就可以在所有数据都已经被处理完之后，再输出一个准确的计算结果。

对于**流式计算的容错问题**，一个很重要的挑战，就是避免僵尸进程仍然在往 Bigtable/Spanner 这样的持久化层里面写入数据。这一点，MillWheel 是通过为每个工作进程注册一个 Sequencer，确保所有的 Key 对应的数据只有一个写入者来做到的。这个处理逻辑，也是通过一个租约机制来做到的，和我们之前见过的 GFS/Bigtable 这样的系统非常类似。

纵观整个 MillWheel，我们的确可以说，无论是数据的正确性、系统的容错能力，还是数据处理的时间窗口，MillWheel 都已经解决掉了，可以说殊为不易。即使是在 2021 年的今天，也不是所有的流式数据处理系统都能做到这一点。不过，这些强大的功能，很大程度上也依赖于 Google 强大的基础设施，特别是 Bigtable/Spanner 这样的系统，能够承载所有的中间数据，以及缓存数据的写入。

推荐阅读

不过，MillWheel 其实离像 Google 的 Dataflow 模型、Apache 的 Flink 这些现代流式处理系统，还有一段距离。一方面，MillWheel 还没有真正考虑把流式处理和批处理统一起

来，做到真正的“流批一体”；另一方面，对于时间窗口的处理，MillWheel 也还很简单，更多是从实际应用的层面出发进行的设计，而没有总结抽象出一个模型。

但是，MillWheel 本身在流式数据处理上已经往前迈进了一大步，给出了一个能够解决容错问题和一致性问题的解决方案。即使你今天已经在使用 Flink 这样更新一代的流式数据处理系统，我也推荐你好好去读一读这篇论文的原文。

思考题

在 MillWheel 的论文里，告诉我们简单地使用 Weak Production，有时候不会提升性能，反而会导致整个系统产出计算结果的延时变大。这个是为什么呢？以及 MillWheel 是如何解决这个问题的呢？

欢迎在留言区分享出你的答案和思考过程，也欢迎你把今天的内容，分享给更多的朋友。

上一篇 29 | Dataflow（一）：正确性、容错和时间窗口

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。