统一批处理和流式计算框架

王晓阳[1,2] 张云聪[1] 孙广宇[2]
[1]百度公司
[2]北京大学

关键词：分布式计算 批处理 流式计算

引言

随着数据量的增长，分布式计算模式逐渐成为数据中心的主流架构。为了满足用户程序的健壮性和实时性要求，人们设计出了许多分布式计算框架，以屏蔽底层复杂的任务划分和集群调度细节，其中最通用的两种被称作批处理框架和流式计算框架。它们有着迥异的编程模型和编程接口，适用于不同的计算场景。

然而在实际应用中，经常会遇到两种框架共同工作的情况。一个常见的例子是 Lambda 架构所面临的情形：数据以流的方式产生，希望对历史数据执行查询操作。批处理引擎可以用来进行定时的离线计算，生成一些预查询的结果来加速查询过程；而流式计算引擎负责处理上次离线计算以来新输入数据的预查询结果，保证查询的实时性。通常情况下，程序员需要在两个不同的引擎上实现相同的执行逻辑，还需要手工合并不同引擎的输出结果。假若需要更改查询逻辑，两套代码也需要同时进行改动。这会极大地增加工程的开发和维护成本。

因此，统一批处理和流式计算框架成为了大数据领域一个很重要的发展方向。目前的解决方案主要分为三类：基于批处理框架的扩展；基于流式计算框架的扩展；基于底层引擎的高层次抽象。本文的第一部分重点介绍这三种解决方案的思路，以及各自的代表性工作；第二部分介绍百度公司在这方面的工作——Bigflow 框架。

三种解决方案

为了统一批处理和流式计算框架，人们提出了各种解决方案。有的方案依托于已有的计算引擎，通过模拟另一类框架的行为来达到统一的目标。也有的方案定义一套适用于二者的通用接口，下层分别使用批处理和流式计算引擎来执行真正的计算过程。

基于批处理框架的扩展

批处理框架的理念是先存储后处理，它注重的是容错性和吞吐量。在批处理程序运行的时候，它总是期望能够获取到所有相关数据。这些计算没有太高的实时性要求，需要保证处理结果的最终一致性，同时也要兼顾各种可靠性、横向扩展性要求，因此常用来做离线计算。

谷歌的 MapReduce 模型是最典型的批处理框架，极大地屏蔽了底层的并行细节。为了应对实时性要求比较高的流式计算场景，一种比较直观的做法是将模型中的批量输入直接替换为流。在文献 [1] 中，流以事件为单位输送给映射 (Map) 端，每个输入事件会产生零个或多个下游事件，继而输送给规约 (Reduce) 端作为其输入流。与原始的 MapReduce 不同的是，此时规约端看到的将是多个源源不断的流而不是有限的数据集，因此它无法像传统的批处

理应用那样得到一个“最终”的结果。为了解决这一问题，MapUpdate 在内存中维护数据结构来保存当前每个关键词 (key) 下所有事件的规约结果。它们同时也会持久化到一个键值存储系统，下游的应用可以从中访问最新的计算结果。

Spark 项目起源于加州大学伯克利分校 AMP 实验室，并被 Apache 社区接受为开源项目之一。它将数据包装在一个容错、并行的数据结构——弹性分布式数据集 (Resilient Distributed Datasets, RDD) 中，将数据的运算过程抽象成 RDD 的变换过程。由于它将中间结果尽可能多地尝试保存在内存中，并针对算子的特性设计了一系列优化策略，因而获得了比 Hadoop 系统（MapReduce 的开源实现）更低的延迟，这也引出了 Spark 模拟流式计算的离散流模型[2]（见图 1）。流式的数据被切分成小批量 RDD 序列，把对流数据的操作转变为普通的 RDD 变换。这种方法直观有效，可以将数据处理的延迟压缩到数秒以内，部分满足了实时性要求不太高的流式计算应用场景。

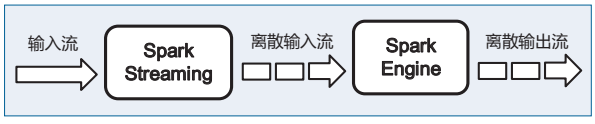

图1 Spark离散流

基于流式计算框架的扩展

流式计算框架的理念是数据到来时直接处理，注重的是延时时间。一般来说，这些计算本身不是太复杂，但输入的数据流会有乱序性、突发性和易失性的问题。在某些场景下，应用甚至不要求数据统计的精确性，牺牲一些准确度来保证低延时的效果也是可以忍受的。因此，流式计算框架的处理延时一般远低于批处理框架，常用来做秒级甚至亚秒级的实时计算。

大规模的分布式流式计算始于 2011 年，推特 (Twitter) 的 Storm 引擎[3]是当时的代表性流式计算框架之一。它的 ACK 组件和异或校验机制保证了每条数据至少被计算一次。为了提高系统的吞吐并保证数据计算的精确性和可靠性，Storm 提供了 Trident 计算原语，以进行小批量的运算。然而由于计算模型的限制，Storm 难以直观地完成批处理中诸如连接和排序之类的常见操作。Trident 原语本质上并没有改变 Storm 的流式计算模式，只能处理一些简单的批处理计算任务。

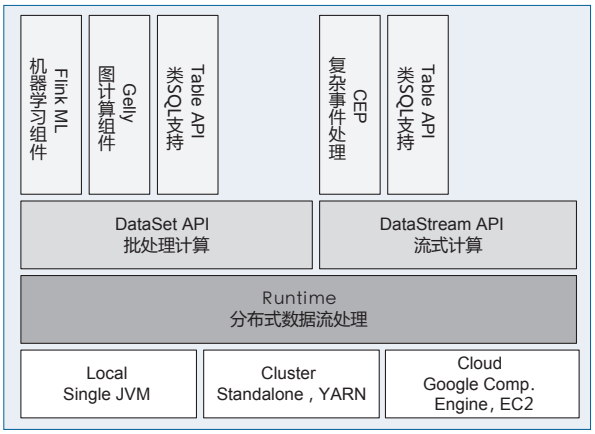

图2 Flink系统生态

Flink 引擎[4]近年来在流式计算领域中备受关注。它的本质是一个流式计算引擎，在 Flink 运行时，分别搭建了批处理 (DataSet) 和流式计算 (DataStream) 的编程接口和相配套的生态系统（见图 2）。这是一个使用流式计算引擎来实现批处理计算的典型例子。在 Flink 中，每个逻辑节点会将结果存入缓存块，缓存块写满或缓存块滞留时间超过限制都会触发与下一个逻辑节点之间的数据交换过程。这种方式看上去和 Storm 的 Trident 原语有些类似，但后者的数据块是预先划分好的，本质上是以小批量的单位进行的流式计算。Flink 还引入了控制事件的概念，并依此实现了分布式快照算法 ABS、水位线标记、迭代计算等复杂操作。控制事件被插入到普通的数据流中，逻辑节点收到后会触发执行对应的预定义操作，这与 Chandy-Lamport 分布式全局快照算法中的标记传递思想非常相似。

依据这些组件，Flink 很容易达到批处理计算中的容错和吞吐要求。Flink 运行时的输入是一些无穷的数据流，而批处理运算的有穷数据集会被 Flink 转换成无穷数据流的特殊情况。为了更好地支持批处理计算任务，Flink 框架也实现了一系列相应的优

化策略，降低了磁盘读写和网络传输开销。

基于底层引擎的高层次抽象

除了扩展引擎语义和功能的方式，基于已有的底层引擎搭建上层的统一高层次抽象也是一种重要的融合思路。它需要定义一个合理的模型抽象层，然后在抽象层下方对接各种批处理和流式计算引擎。这种设计思路的好处在于，假设出现了更为强大的计算引擎或底层引擎引入了更高效的特性时，可以仅对模型抽象层和底层引擎的对接进行必要的调整，用户的实例代码无须改动。同时，用户也无须学习多套引擎的书写方式和实现技巧，可极大地降低用户的学习成本和代码的维护成本。

如前文所述，Hadoop 之类的批处理引擎有着良好的容错性、较高的吞吐特性。然而，数据处理的延时不尽如人意。Storm 之类的流式计算引擎的延时非常低，难以支持复杂的计算，也缺乏良好的容错机制。Lambda 架构[5]是这类问题的一种很有效的解决方式，但直接使用 Hadoop 和 Storm 去实现则存在二次编码的问题。推特公司的开源项目 Summingbird，提供了一种上层领域专用语言 (DSL) 来实现一个类似于 MapReduce 的编程模型[6]，同时支持了批处理计算模式、流式计算模式和混合计算模式。

Apache 社区的 Beam 项目是基于谷歌的 Dataflow 模型[7]创建的。这个模型将流式计算的四个维度剥离开来：What（输入数据将如何被处理），Where（基于日志时间的数据划分），When（基于墙上时间的结果输出）和 How（掉队数据的处理方式）。降低了流式计算描述的复杂性。Beam 将流式数据当作一个无穷数据集，将离线数据当作一个有穷数据集。当底层是批处理引擎时，有穷数据集直接被当作一个批处理任务，而无穷数据集可以被划分成一个批处理任务的序列。当底层引擎是流式计算引擎时，它应当支持有穷数据集的运算任务。

Dataflow 模型是一种优秀的模型。它主要遵循了以下三条设计理念：

1. 同样的代码在不同的引擎上应有相同的结果语义。

2. 不同引擎对时效性、准确性、资源消耗等方面各有偏倚。

3. 触发器不同不应改变计算的结果语义，但有可能改变时效性、准确性、资源消耗。

目前 Beam 项目已对接完成的引擎有 Apache Flink、Apache Spark 和 Google Cloud Dataflow 等。

Bigflow框架

基于底层引擎的高层次抽象的策略可以降低用户的学习成本和代码的维护成本，高层次抽象的设计和实现是其中关键的部分。百度 Bigflow 项目提供了一种可嵌套分布式数据集的抽象，达到了兼顾模型直观性和系统高效性的目的。

分布式数据集模型与分组操作

随着分布式计算的发展，采用以数据为中心的函数式编程模型的系统越来越普遍。在这种模型下，用户可以更多地考虑计算逻辑语义本身而不是代码执行时的流程控制，让算子去追随数据的方式也极大地减少了数据的迁移代价。这种抽象使得用户的分布式程序更加接近用户的直观思维，大大降低了用户学习、使用的成本，屏蔽了较多的底层细节，给分布式框架为用户代码进行自动优化提供了空间。

Spark 的 RDD 是函数式编程模型的优秀代表。它是一个容错的、并行的、只读的分区数据集合，经过变换操作后会产生新的 RDD。只要定义好这些 RDD 的变换过程，Spark 就可以自动将数据切分到各个物理节点，并把算子传递到相应的节点展开计算。分组操作是这些变换中最重要的一类操作。它根据算子产生的键对数据集进行重新分组，对应于 MapReduce 模型中 Map 阶段末尾的提取键值操作和整个 Shuffle 阶段的数据转移操作，使得 Spark 框架的通用型得到保证。

在目前 RDD 模型的设计中，分组后相同键的键值对会变成一个单机的列表，所有的列表组成了一个新的 RDD。然而，这种设计会导致下面三个问题：

第一，后续的变换中每个单机列表内进行的操作对分布式框架而言是一个黑盒，系统很难根据这些操作的性质来优化整个程序的执行。比如，Spark 提供了 reduceByKey、aggregateByKey 和 combineByKey 等一系列接口用来进行上游数据预聚合优化，并提示用户在有这种预聚合可能的情况下尽量避免使用单纯的 groupByKey 算子。这是因为 Spark 框架无法将 groupByKey 分组后每个单机列表上进行的操作中的聚合语义提取出来。

第二，这些单机列表也限制了分布式框架将组内数据按需求分布式散列到多机进行处理的可能。因此 Spark 用户经常需要手工优化自己的代码，以避免某些节点产生的内存不足问题。而多数情况下，用户对代码进行过多的手工优化会使得代码变得丑陋且难以理解，增大代码维护的难度。

第三，现有的分布式算法也很难复用于每个分组之上。例如，用户已经实现了一个针对整个 RDD 的逻辑回归算法，但在另一个场合下，用户又需要得到相同键下的所有键值对的逻辑回归结果。此时，用户只能重新实现一套运算逻辑完全相同的单机版本的逻辑回归算法，增大了开发的代价。

可嵌套分布式数据集模型

为了解决 RDD 中分组操作的问题，Bigflow 采用了一种称为可嵌套分布式数据集 (Nested Distributed Database, NDD) 的抽象。它由三种数据结构组成：

- PCollection 是 NDD 数据模型的基本单位，它代表一个只读的分布式数据集合，对应于 Spark 的 RDD 概念。
- PObject 是只有单一元素的数据集合，是 PCollection 的特殊情况。它一般是由聚集算子产生。
- PTable 在逻辑上是一个键值列表，其中每个键值对中的值可以是 PCollection 和 PObject，也可以是 PTable。因此 PTable 是 Bigflow 数据集抽象中“可嵌套”概念的来源。

NDD 模型吸收并扩大了 Spark 的 RDD 模型中的变换概念，任何应用在一个或多个 NDD 上并产生新的 NDD 的操作都被称作变换。例如，reduce 之类的聚集操作在 Spark 中是一个行动算子，而在 Bigflow 中仍旧是一个变换算子。行动算子会触发真正的计算操作，后续无法跟随其他变换算子。

在 NDD 模型的设计中，PCollection 经过分组操作会变成一个 PTable，键相同的元素被组织成 PTable 中的一个 PCollection。用户可以将定义在 PCollection 上的变换操作传递给 PTable，然后 PTable 会将这些变换应用到它的每一个元素上。这种设计解决了前文提到的 RDD 模型的三个问题：

首先，传递给 PTable 的变换是应用在 PCollection 上的算子组成，因而很容易得知这些算子的性质，并自动进行预聚合等优化。

其次，PTable 的元素 PCollection 依旧是分布式数据集，并不一定需要放在同一个物理节点上形成一个单机的列表，有助于解决数据倾斜的问题。

最后，代码的复用性是显然的。考虑前文逻辑回归问题的例子，假若用户已经在 PCollection 上实现了一个逻辑回归算法，那么此时他可以直接将这个算法传递给 PTable，应用在其中的每一个 PCollection 元素上。

另外，NDD 模型使得用户完全无须关心数据

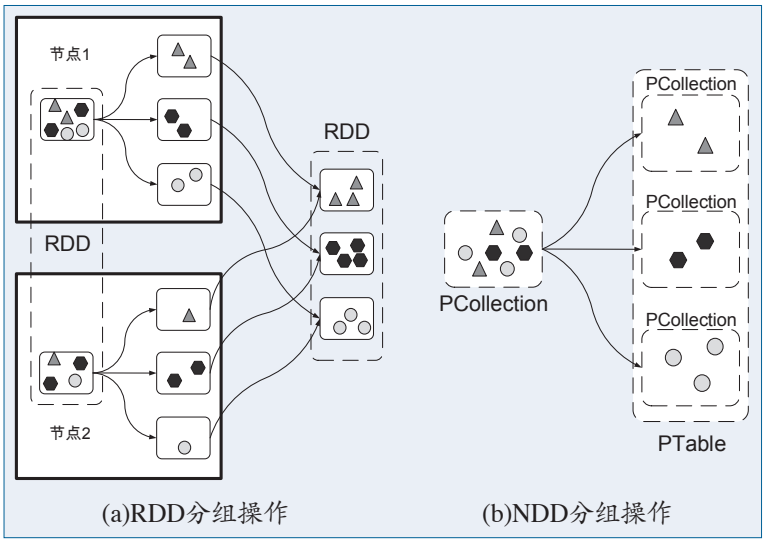

图3 RDD与NDD分组操作对比

的分组方式，这意味着用户并不能对框架的分组方式进行任何假设，框架可以自由地实现更多的优化算法。

图3对比了RDD和NDD在分组操作上的异同之处。图中的圆角实线框表示内部的数据都在相同的节点中，可以看到RDD分组操作后形成的新RDD。相同键下的数据聚集在同一个运算节点，在RDD上实现的算法无法直接应用于这些单机列表。而NDD的PCollection在分组操作后形成的是一个PTable，内部仍旧是分布式的PCollection，因此它们可以直接使用在PCollection上实现的所有算法。

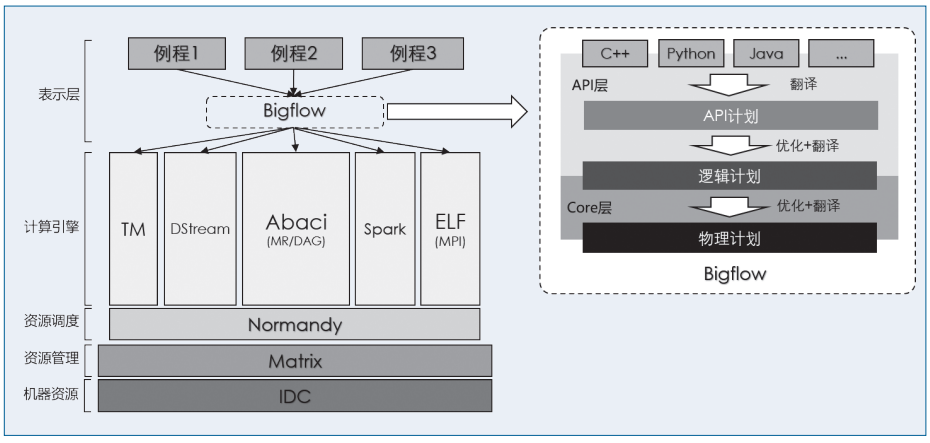

图4 Bigflow系统架构

Bigflow系统整体架构

Bigflow系统的整体架构如图4所示，主要包含API层和Core层两个组成部分。Bigflow应用代码在API层被翻译成一个API计划，经过优化后翻译成逻辑计划；逻辑计划在Core层继续被优化，最终翻译成物理计划，调用底层执行引擎，并执行实际的运算过程。

API计划记录了应用代码的具体细节。它被组织成一个有向无环图(DAG)，每一个结点代表一个算子，结点之间的有向边代表数据流动的方向。

逻辑计划在API计划的DAG图的基础上引入了作用域树的概念。每个内部结点代表一个作用域，每个叶子结点代表一个运算结点。运算结点由其父亲结点规约其作用域范围，即算子应该作用在哪些数据分片上。

为了能更直观地展示各个作用域之间、运算结点和作用域之间的从属关系，图5将作用域树中的父子关系表述成方形框的嵌套关系，根结点是最外面的全局域。图中弧形框代表叶子结点，它们之间的箭头与原始作用域树的内部边无关，而是来源于

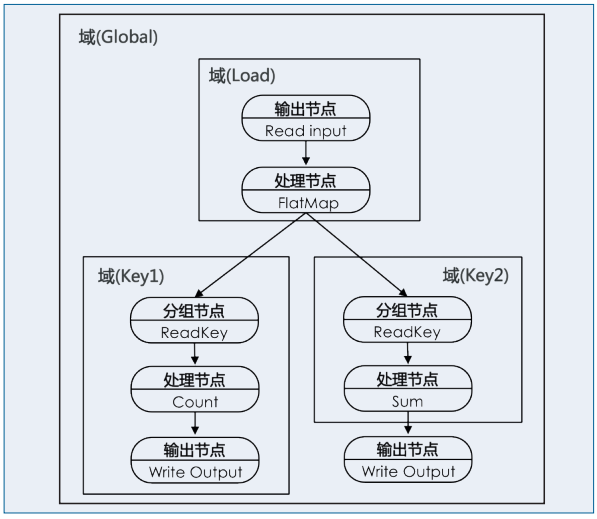

图5 逻辑计划举例

API计划的DAG图中的有向边，代表了数据流动的方向。

Bigflow系统在实现NDD模型的同时，针对嵌套数据集的特点实现了一系列自动优化的算法。这些优化方法在传统的系统中只能依靠手工完成，难以编写和维护。Bigflow依赖这些自动优化策略，可以用最简单直观的代码达到和手工优化性能相近的效果。在真实业务场景中，在同一种执行引擎上执行批处理任务时，Bigflow的性能较用户手写代码平均提升50%以上。

批处理计算和流式计算融合

流式计算的输入和批处理数据的一个不同点在于，每个元素通常会有一个日志时间的概念，很多操作都是围绕着这个时间进行的。在NDD的实现中，我们借鉴了 Spark 中 schema 的概念，支持将 PCollection 中的元素表述成一个多维表项，每个维度都带有自己的名字。我们恰好可以使用 NDD 的这个特性，为流式数据的每个表项保留一个隐藏的时间字段，记录它的日志时间。

为了让 NDD 同时支持批处理和流式计算模型，我们为 PCollection 和 PTable 添加了标记，指明它包含的元素个数是有穷还是无穷。当在无穷 PCollection 上应用分组操作时，结果依旧是一个 PTable。这个 PTable 可能是有限的，也可能是无限的，但它的每一个元素都是一个无限的 PCollection。这其实是一个分流的过程，算子根据流中元素的性质（算出来的键）将原始流拆分成一个或多个下游的流。NDD 模型也引入了窗口操作。它按照时间字段将无穷 PCollection 变换成一个无穷 PTable，它的每个元素都是一个有穷的 PCollection。

图 6 展示了应用在无穷数据集上的分组操作和窗口操作的例子。其中，分组操作根据数据的奇偶性将原始流分成两个流，可以看到结果是一个包含两个无穷 PCollection 元素的有穷 PTable。而窗口操作按照数据的日志时间将原始的流切分为一系列的有穷数据集，操作产生了一个包含无穷多个有穷 PCollection 的无穷 PTable。

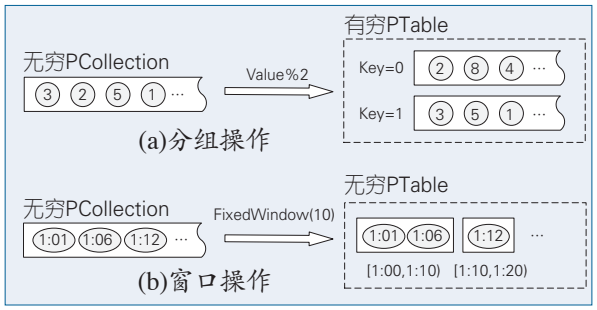

图6 流式计算分组和窗口操作示例

在这种设计的支持下，原有的变换操作的语义没有发生任何变化，窗口操作也独立于这些操作。因此，批处理计算部分实现的 Bigflow 算法可以无缝迁移到流式计算部分中来，极大地提高了代码的复用性。

Bigflow 也借鉴了 Dataflow 的四维度模式和前文提到的三点设计理念，同时受益于 NDD 模型的设计，扩展了以下两点新的原则：

1. 程序写起来应更像单机程序。这表明，所有的数据集都可对应于一种现有的常见单机数据结构，计算过程中完全屏蔽分区的细节。对比来说，Dataflow 的有界 / 无界数据集模型抽象不太直观，而 Spark 又经常因为性能原因需要用户显式地执行重新分区操作。

2. 框架应尽可能了解用户程序执行细节。这意味着，分组操作后所产生的在每个分组上执行的后续操作都应由框架提供的原生变换组成，而不应是用户实现的黑盒代码。 ■

王晓阳

北京大学信息科学技术学院博士生，百度基础架构部实习生。主要研究方向为面向大数据的高性能系统研究。

yaoer@pku.edu.cn

张云聪

百度基础架构部分布式计算架构师，致力于百度分布式计算平台的建设工作，曾参与百度内部两个主要流式计算系统的研发，现任百度统一分布式优化表示层Bigflow的技术负责人。

zhangyuncong@baidu.com

孙广宇

CCF专业会员，CCF体系结构专业委员会、计算机工程与工艺专业委员会委员。北京大学信息学院高能效计算与应用中心长聘副教授、研究员。*ACM TECS*和*ACM JETC*杂志副主编。主要研究方向为高能效计算机系统与体系结构、异构加速器系统等。gsun@pku.edu.cn

参考文献

[1] Lam W, Liu L, Prasad S, et al. Muppet: MapReduce-style processing of fast data[J]. *Proceedings of the VLDB Endowment*. 2012, 5(12): 1814-1825.

[2] Zaharia M, Das T, Li H ,et al. Discretized streams: Fault-tolerant streaming computation at scale[C]//*Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013: 423-438.

[3] Apache Storm[OL].http://storm.apache.org/.

[4] Carbone P, Katsifodimos A, Ewen S, et al. Apache flink: Stream and batch processing in a single engine[J].*Data Engineering*, 2015,38(4):28-38.

[5] Marz N, Warren J. *Big Data: Principles and best practices of scalable realtime data systems*[M]. Manning Publications Co., 2015.

[6] Boykin O, Ritchie S, O'Connell, et al. Summingbird: A framework for integrating batch and online mapreduce computations[J].*Proceedings of the VLDB Endowment*, 2014,7(13): 1441-1451.

[7] Akidau T, Bradshaw R, Chambers C, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing[J]. *Proceedings of the VLDB Endowment*, 2015, 8(12): 1792-1803.