流计算端到端一致性概述

目录

计算平台基本逻辑

数据流图是用户计算逻辑的抽象，是一个有向无环图。

逻辑数据流图：根据用户的程序得出

物理数据流图：将计算任务发布到计算平台后，每个计算节点上多个任务并行执行

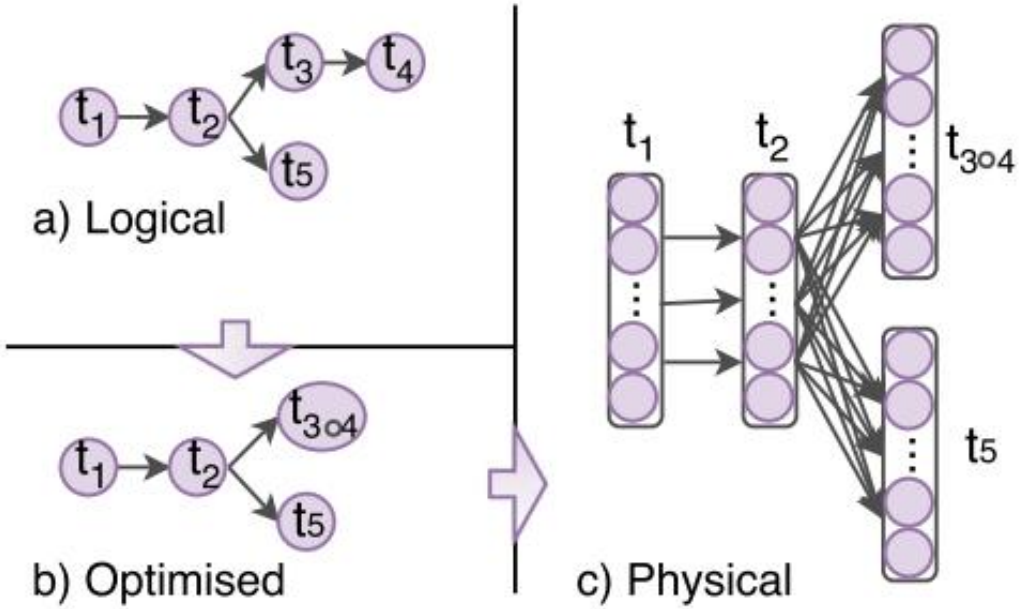


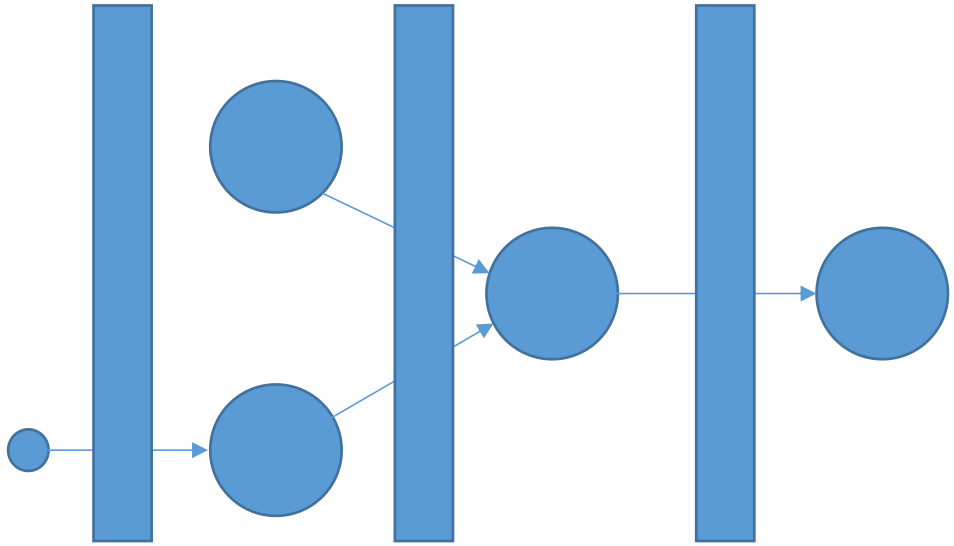
Figure 2: Dataflow Graph Representation Examples.

流批一体

右图是批处理平台的一个简单实现示意图

上游计算节点将数据处理完成后存入文件系统，然后下游计算节点从文件系统中读取数据后进行计算，结束后存入。

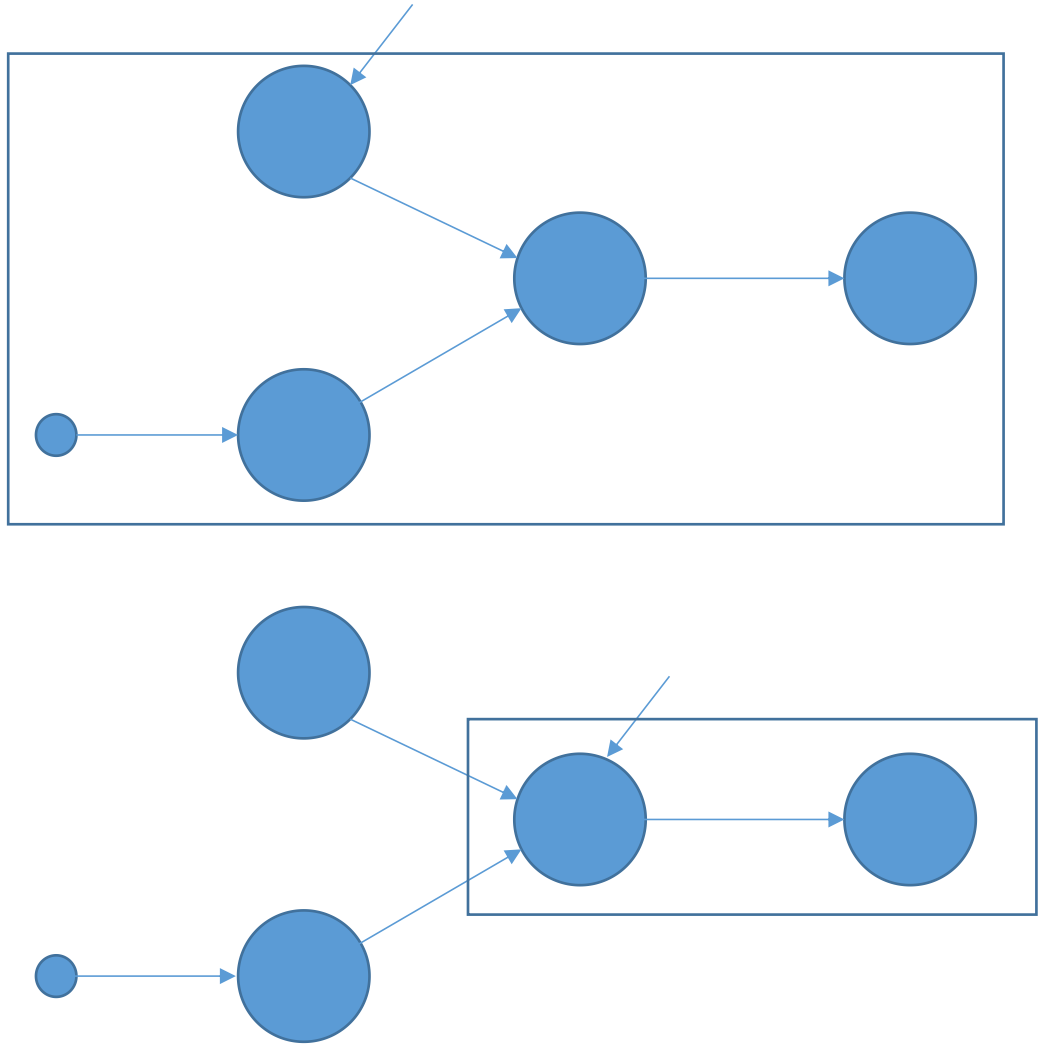
多个计算节点之间形成了一个有向无环的数据流图



流批一体

右图是血缘关系的一个简单示意图
会在：
计算任务比较重的节点上使用快照保存中间结果，防止重发数据时重新计算结果。
在计算任务比较轻的节点上，则不保存中间结果，重发数据时向更上游的计算节点请求数据后重算结果。

也就是说每个计算节点上的需要维护一个数据依赖图（血缘）来获取数据。



流批一体

流数据与批数据最大的区别在于数据是否处在**运动状态**

考虑如下的数据，如果业务需要计算：开始总共接收了多少条数据。

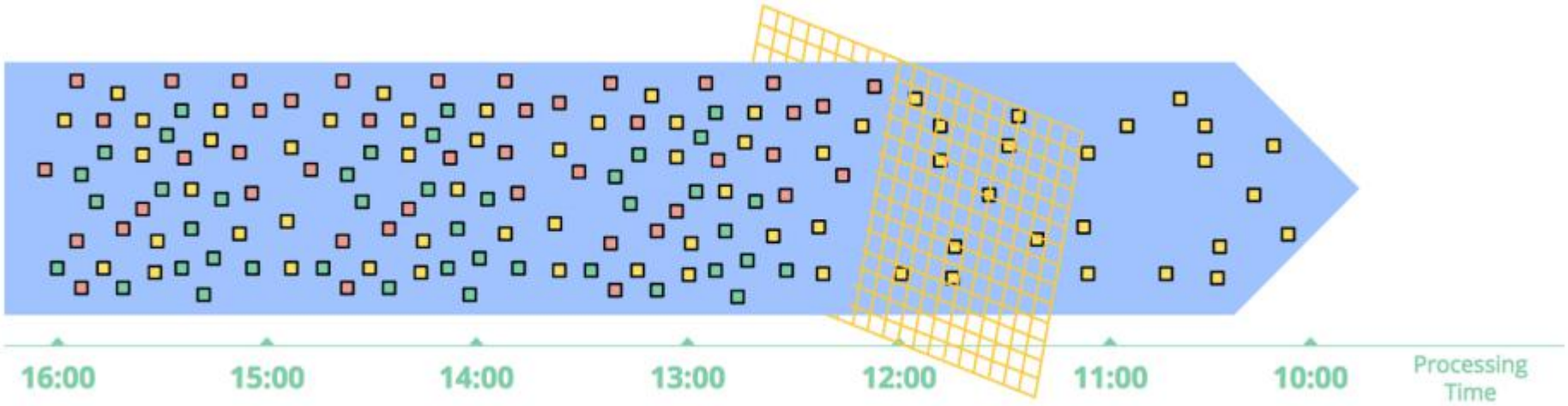
则流处理平台（如）会选择将每一条新到的数据立刻进入系统，然后在计算过程中将维护的的变量（增量计算，实时计算）

批处理平台则无法做到数据立刻进入处理系统，因为批处理平台的每一次输入都是独立的（全量计算、重新计算）

流批一体

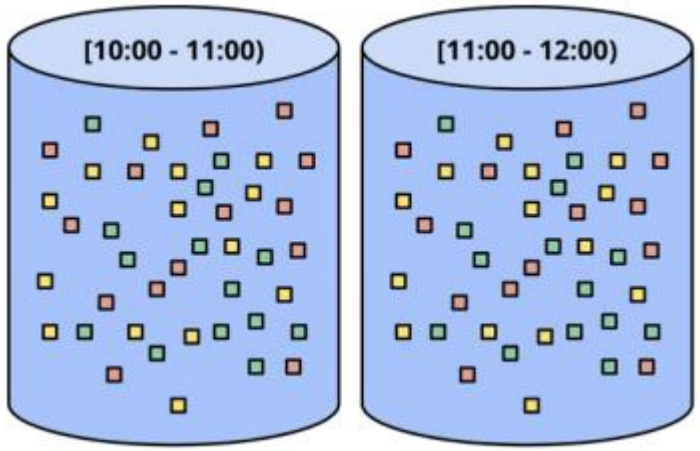
流数据输入：
系统无法确定数据的边界，无法确定数据是否完整，同时新数据不断的到来，不断地流入系统。

流计算平台需要维护用来实现增量计算的状态，每当新数据到来时修改状态



流批一体

批数据输入：
系统可以确定数据的边界，数据认为是完整的，业务中常常表现为历史数据



批处理平台不需要维护状态，因为每个输入的批数据之间相互独立。

显然，流处理平台天然的可以处理批数据（历史数据），而批处理平台则无法处理流数据

流批一体

接下来介绍一下在数据处理角度的流批一体

（批）和本身就是一体的

随着时间的推移，流数据的累积形成表，在计算节点中表现为聚合操作的结果。
流数据的变换过程，在计算节点中表示为计算
一段时间中表的是流数据
没有这种情况，只存在

流批一体

流批一体计算实现算子的一个例子

```
12:10> SELECT TABLE * FROM Left;
--------------------
| Num | Id | Time  |
--------------------
| 1   | L1 | 12:02 |
| 2   | L2 | 12:06 |
| 3   | L3 | 12:03 |
--------------------
```

```
12:10> SELECT TABLE * FROM Right;
--------------------
| Num | Id | Time  |
--------------------
| 2   | R2 | 12:01 |
| 3   | R3 | 12:04 |
| 4   | R4 | 12:05 |
--------------------
```

左表和右表都有三列：，和，其中是的连接条件，是数据到达的时间

右图是上图两表后的结果

```
12:10> SELECT TABLE
         Left.Id as L,
         Right.Id as R,
       FROM Left FULL OUTER JOIN Right
       ON L.Num = R.Num;
--------------
| L    | R    |
--------------
| L1   | null |
| L2   | R2   |
| L3   | R3   |
| null | R4   |
--------------
```

流批一体

从流计算的角度来看算子的实现过程

右图是两表进行流计算时计算节点随着时间推移的结果展示

随着时间的前进，左表和右表的数据不断，也就是说批数据的结果其实是流数据结果某个时间点的快照

同时容易发现左连接，右连接其实是全外连接结果的过滤。

```
12:00> SELECT STREAM
         Left.Id as L,
         Right.Id as R,
         CURRENT_TIMESTAMP as Time,
         Sys.Undo as Undo
       FROM Left FULL OUTER JOIN Right
       ON L.Num = R.Num;
------------------------------
| L    | R    | Time  | Undo |
------------------------------
| null | R2   | 12:01 |      |
| L1   | null | 12:02 |      |
| L3   | null | 12:03 |      |
| L3   | null | 12:04 | undo |
| L3   | R3   | 12:04 |      |
| null | R4   | 12:05 |      |
| null | R2   | 12:06 | undo |
| L2   | R2   | 12:06 |      |
....... [12:00, 12:10] .......
```

目录

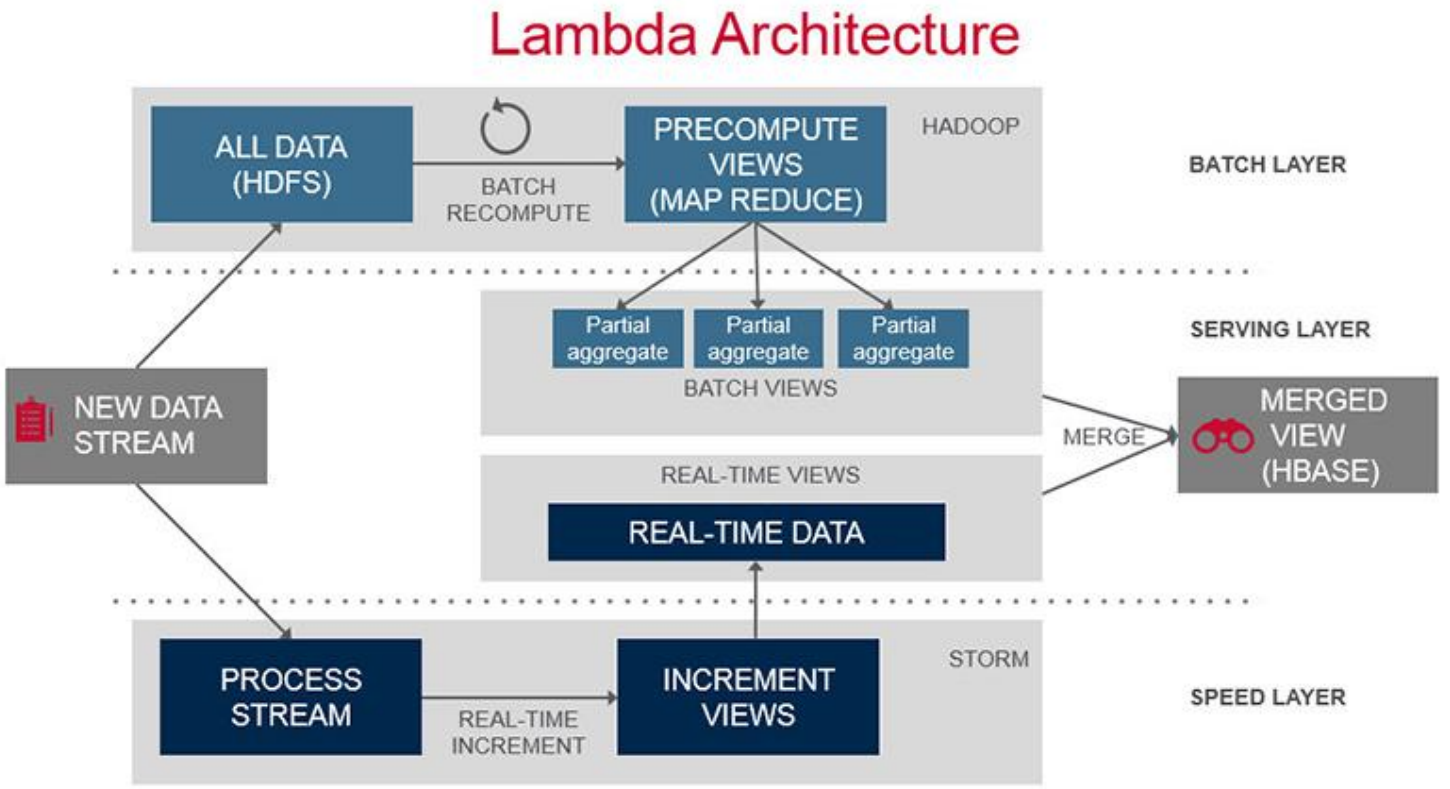
流处理的概念

架构分为三部分：

：批处理部分

：流处理部分

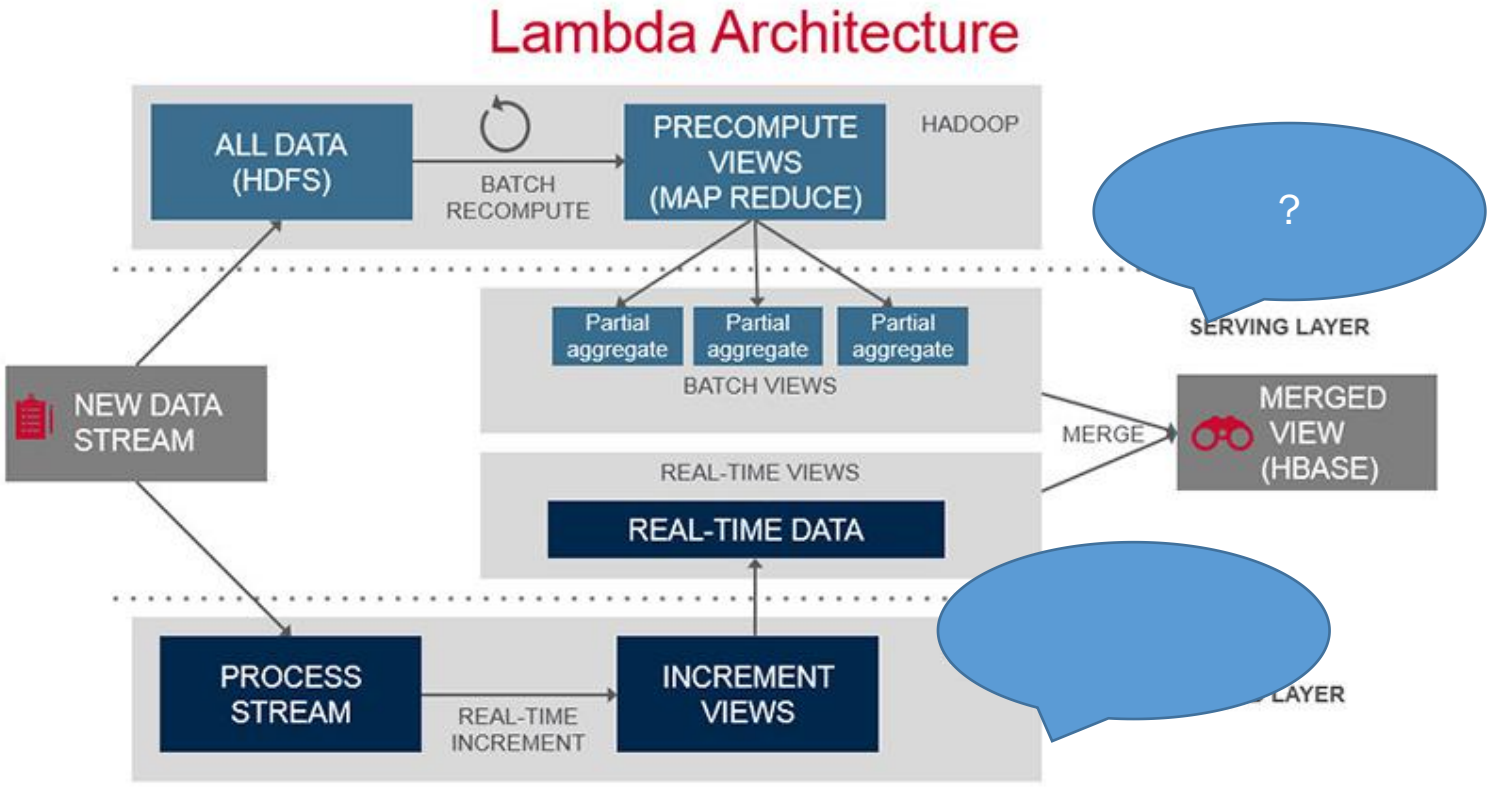
：对两个结果合并后返回给



流处理的概念

早期的流处理平台在短时间内可以保证一定的正确性一致性

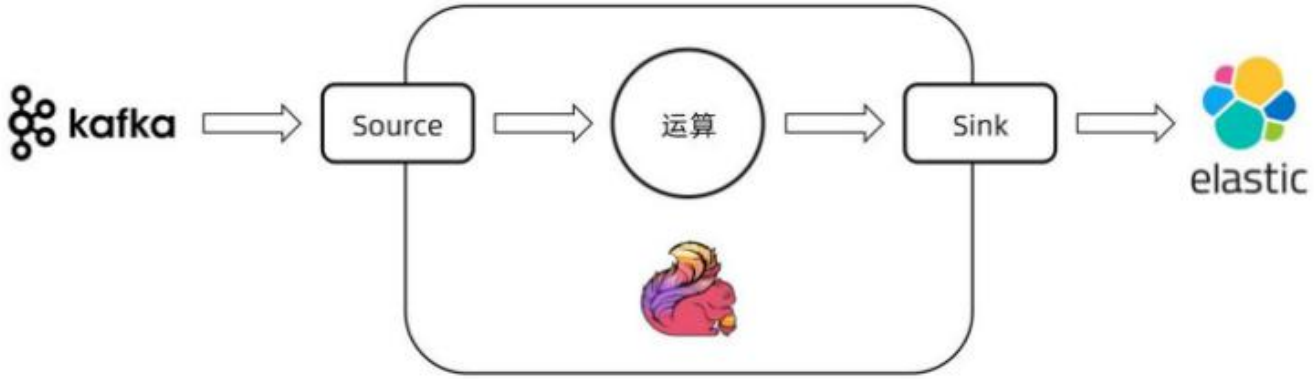
但是随着系统运行时间的增长，流计算平台的结果准确度会越来越差，最终导致用户直接放弃流计算结果，转而等待批处理结果。



流批一体的端到端一致性

以举例，一个流计算应用可以用下图来表示。

端到端一致性（）：表示从数据输入到数据流出阶段，整个处理过程的正确性，在流处理平台中的意思就是，从数据流入的外部数据库到汇出的外部数据库，所有输入的数据都刚好被处理一次且输出一次结果。



目录

端到端一致性理论实现

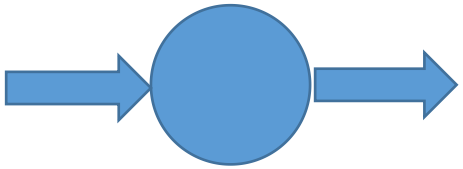
在介绍各个计算平台不同的工程实现之前，首先从理论方面展示端到端一致性中可能遇到的问题，并提出解决方案。

计算节点失败的基本的概念：

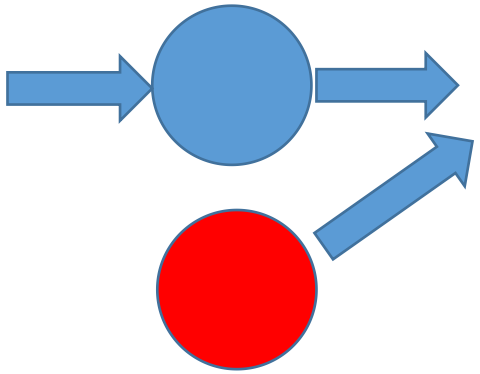
一个计算节点有这两种可能性

节点崩溃，无法工作

僵尸节点，仍在工作，但系统无法感知到，认为这个计算节点崩溃



节点奔溃，处理的数据丢失



僵尸节点，导致下游（快照存储下游算子）的重复数据

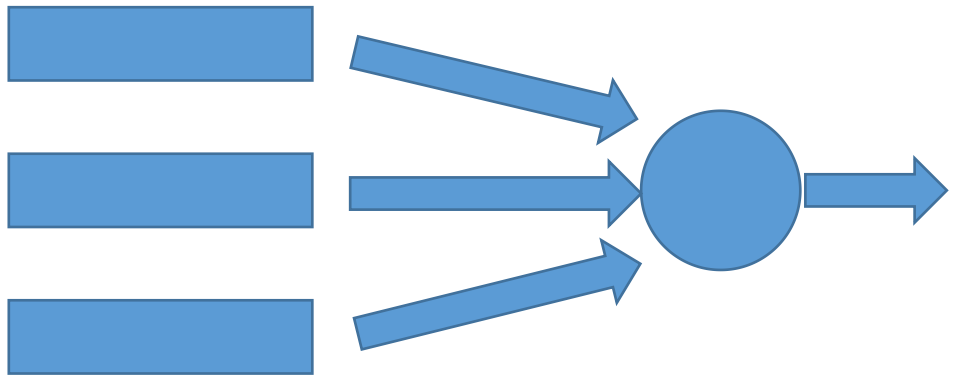
端到端一致性理论实现

计算节点的上游节点可能是数据库、文件系统、消息队列等，计算节点通过操作请求数据。

如果算子

当节点奔溃时，发送到下游的数据丢失，这就要求上游的数据存储要支持回放数据，**需要存储数据的处理进度**。

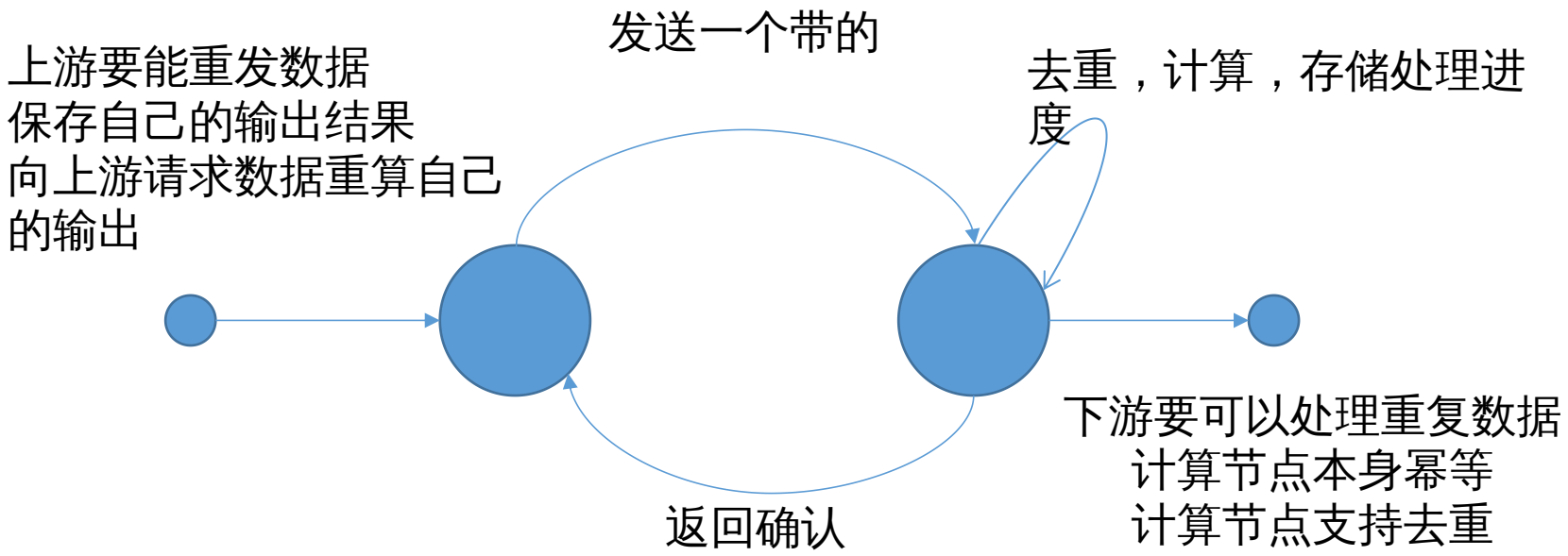
当为僵尸节点时，重启后的新计算节点导致发送给下游节点的数据重复，所以需要下游算子可以对去重。



端到端一致性理论实现

计算框架内部的计算节点的执行过程可以简单抽象成如下结构

在这步的任何时刻，下游计算节点都可能。

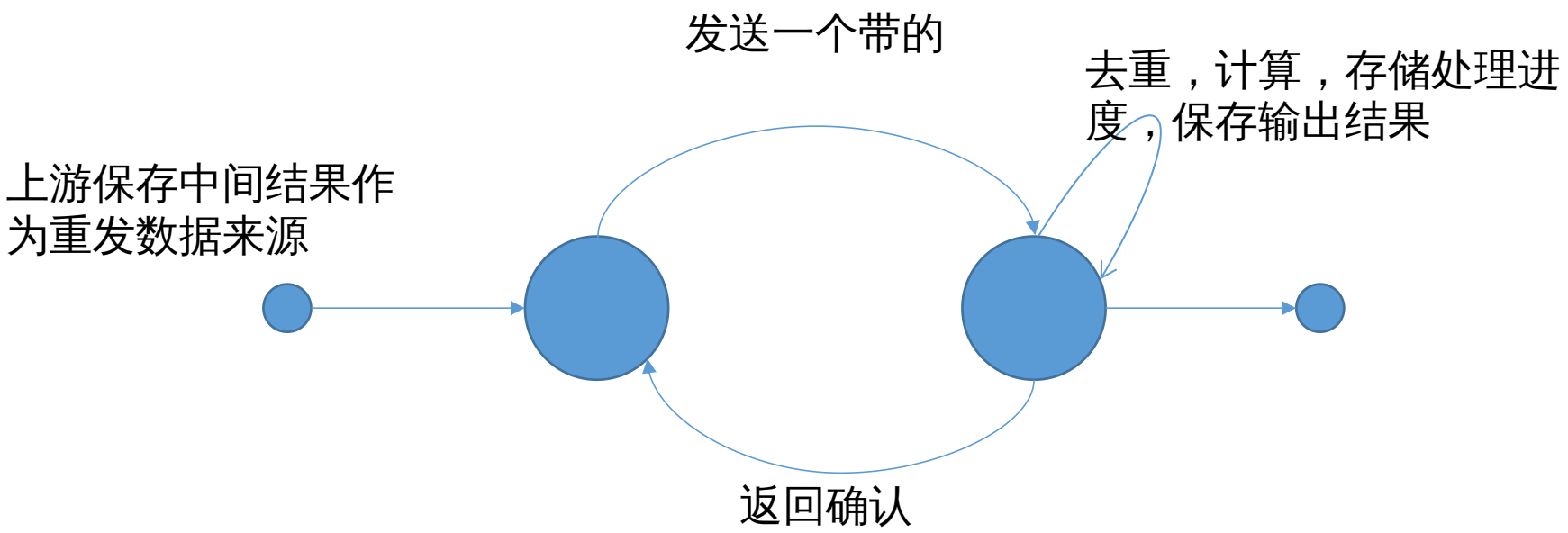


端到端一致性理论实现

首先考虑上游选择**保存输出结果**作为重发数据来源的情况，此时每个计算节点需要

保存数据处理进度到外部持久化存储，用来重启后告诉上游应该重发那些数据

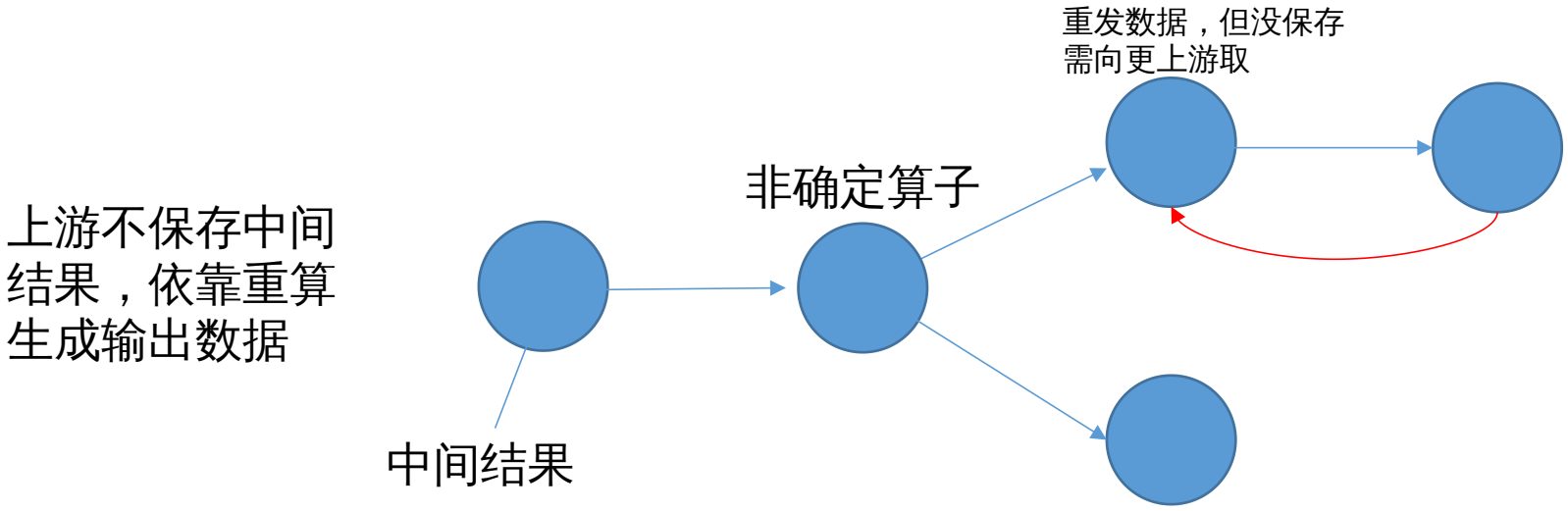
保存计算的结果到外部持久化存储，用来在下游丢失数据后重发数据



端到端一致性理论实现

然后考虑上游不保存输出结果，而是依靠更上游的**数据重算**输出的情况。

此时需要引入一个概念
确定性计算：给定一组相同的数据，重复运行多次，或者打乱数据顺序，最终的结果都是不变的，比如
非确定性计算：除了确定性计算，剩下的都是非确定性计算



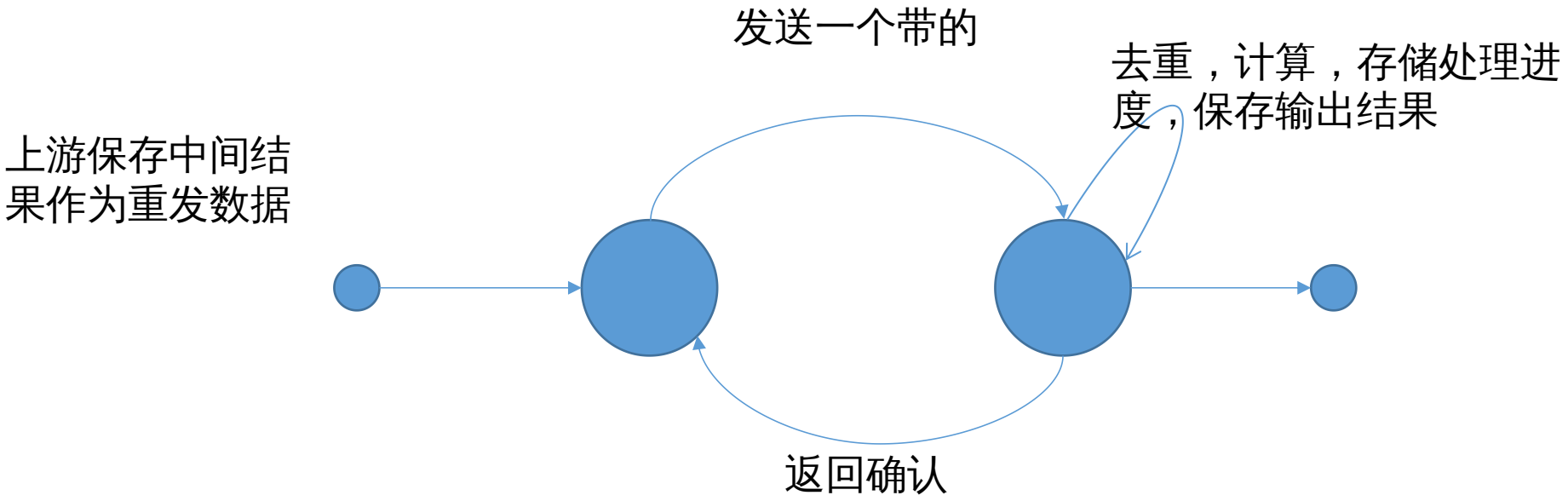
端到端一致性理论实现

回过头来，重新考虑考虑上游选择**保存输出结果**作为重发数据的情况，此时每个算子需要

保存处理进度到外部持久化存储

保存输出结果到外部持久化存储

在面对非确定性计算时，采用先后的方式时，重算可能导致后端收到了同一份数据生成的两份重复的计算结果



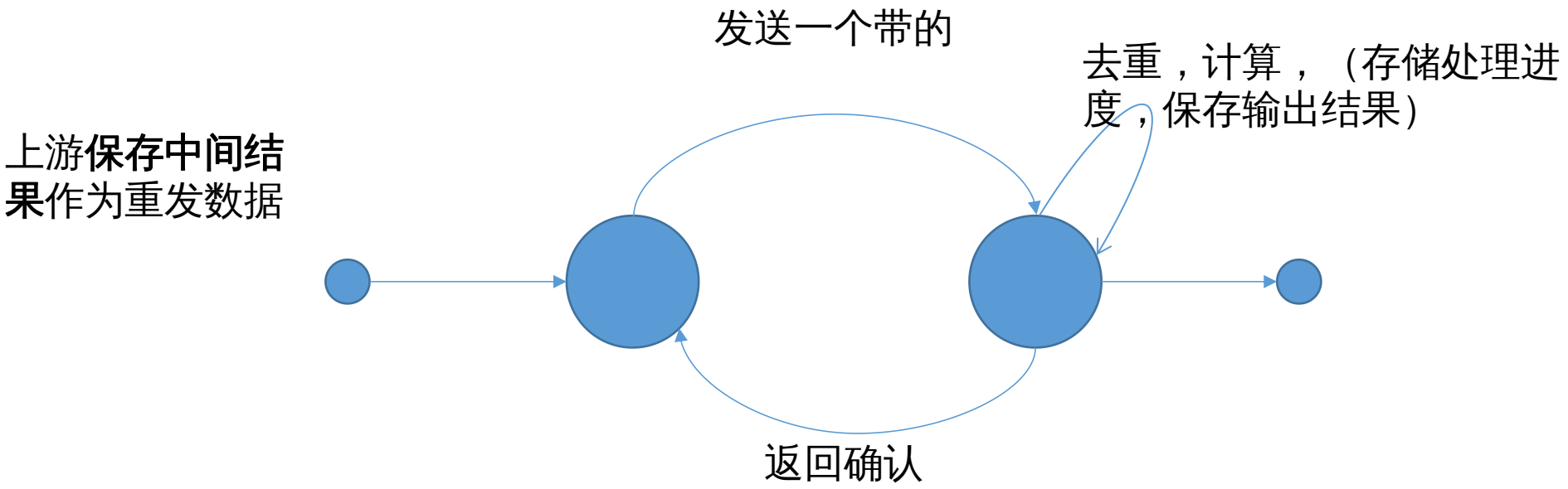
端到端一致性理论实现

每个算子需要

保存处理进度到外部持久化存储

保存输出结果到外部持久化存储

此时更加简单的策略就是，**把保存处理进度和输出结果作为一个原子操作**



端到端一致性理论实现

小结：

此时可以得出一个**基于全部计算节点都存储中间结果**，可以保证非确定计算端到端一致的处理平台，必须保证计算节点可以

- 保存**处理进度**和保存**中间结果**原子性执行
- 可以处理重复数据

当计算平台为流处理平台时，节点维护，我们只需要将条件扩充为

- 保存**处理进度**、**中间结果**、**状态**原子执行
- 可以处理重复数据
- 维护

即可，接下来介绍四种不同的工程实现

（基于重算结果重发数据的计算平台，在部分重点讨论）

目录

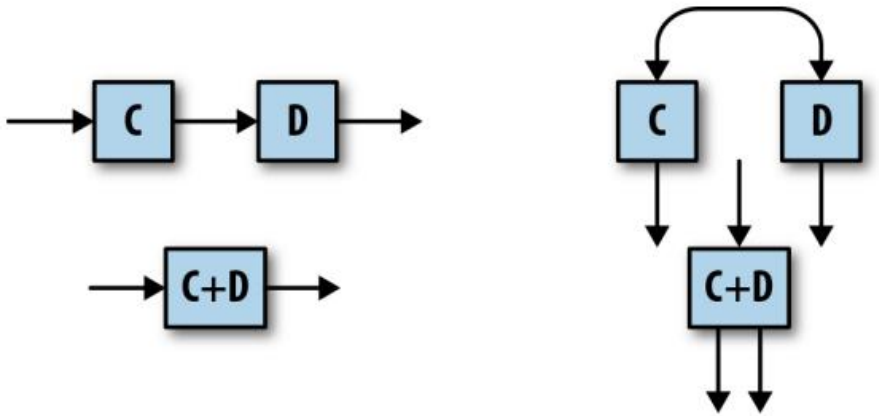
的端到端一致性实现

上下计算节点之间使用进行数据传输。

在对数据进行重复处理中，没有选择在上下游之间约定来实现去重，而是为每一条进入系统的分配了唯一的，每个计算节点维护一个已经处理过的集合，当收到新时，通过集合查找去重。

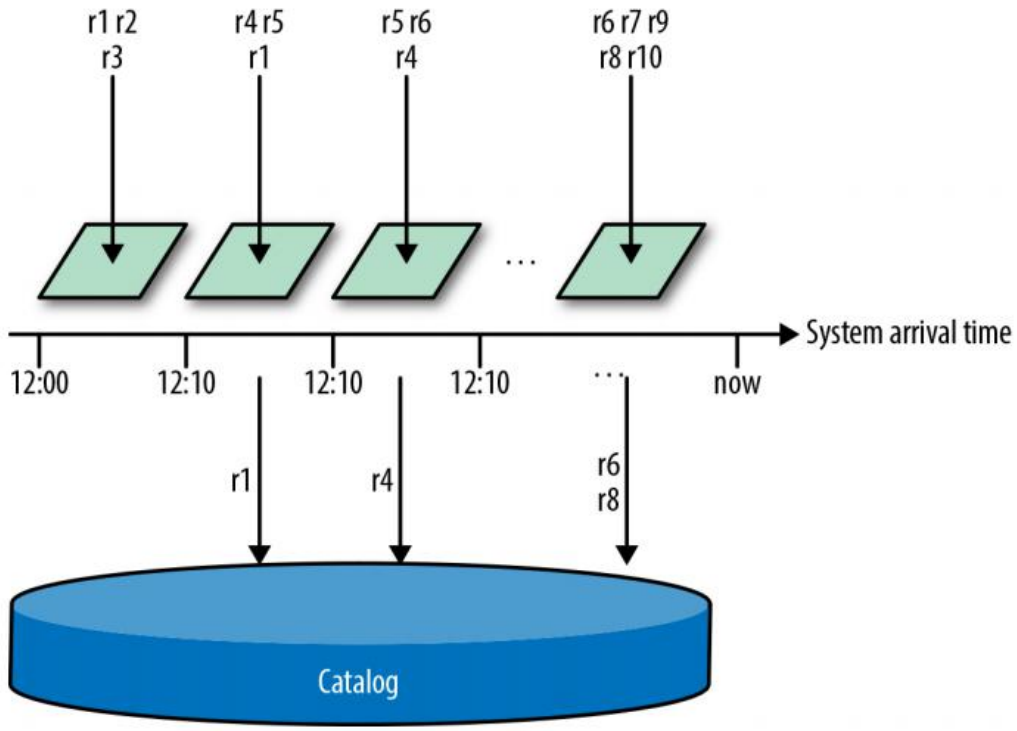
显然维护这个集合的开销是巨大的，所以使用了
- 图优化
- 过滤器

来提升性能



的端到端一致性实现

按照系统时间为每一个进入系统的消息赋予一个时间戳，然后维护多个过滤器。并当消息确定执行完成后，计算节点对布隆过滤器进行垃圾回收。



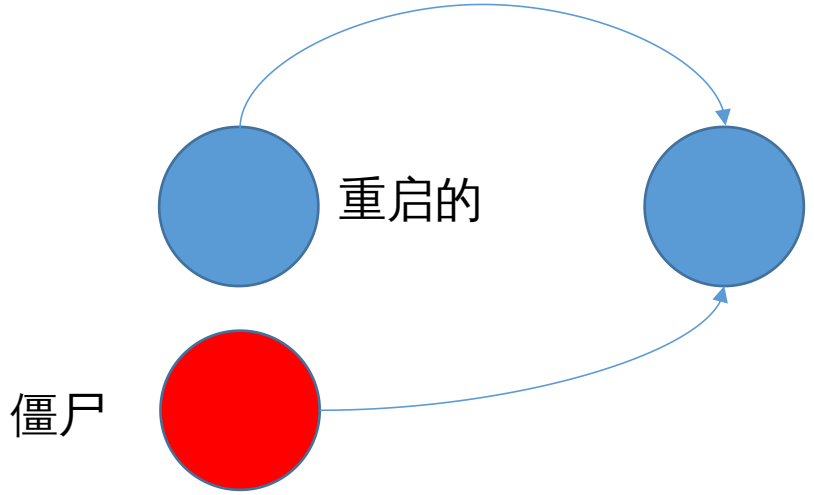
的端到端一致性实现

中通过事务的方式将中间结果，处理消息的，状态保存到后端的中，此时还需要注意僵尸计算节点的处理。

僵尸节点发到下游处理节点的被重复处理机制过滤。

但是僵尸节点保存状态、中间结果、到中时，相当于特殊的下游节点，而缺少去重机制

通过保证旧的写者的写操作不会生效，这个可以理解为的号，在重启后，必须**要读先写**，把一个更高的存入，阻止僵尸节点的写入。

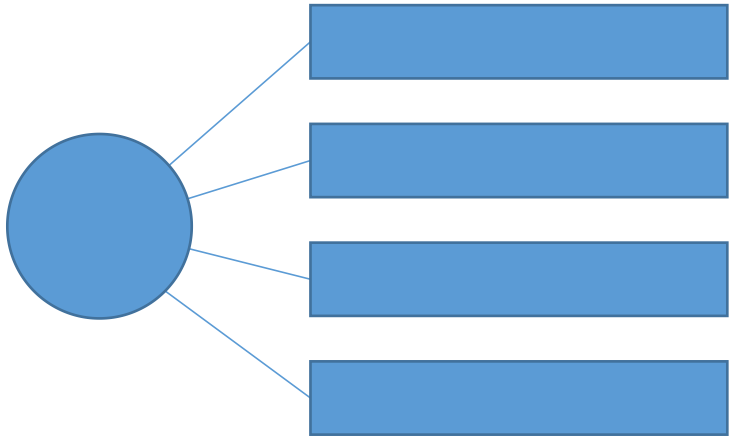


的端到端一致性实现

在介绍之前，我们首先了解一些消息队列的基础概念。

通过主题来订阅或推送数据，每个由多个构成
在向中发送数据时可以指定数据存储的位置，同时中的数据可以通过一个确定的来访问（流处理的进度）。

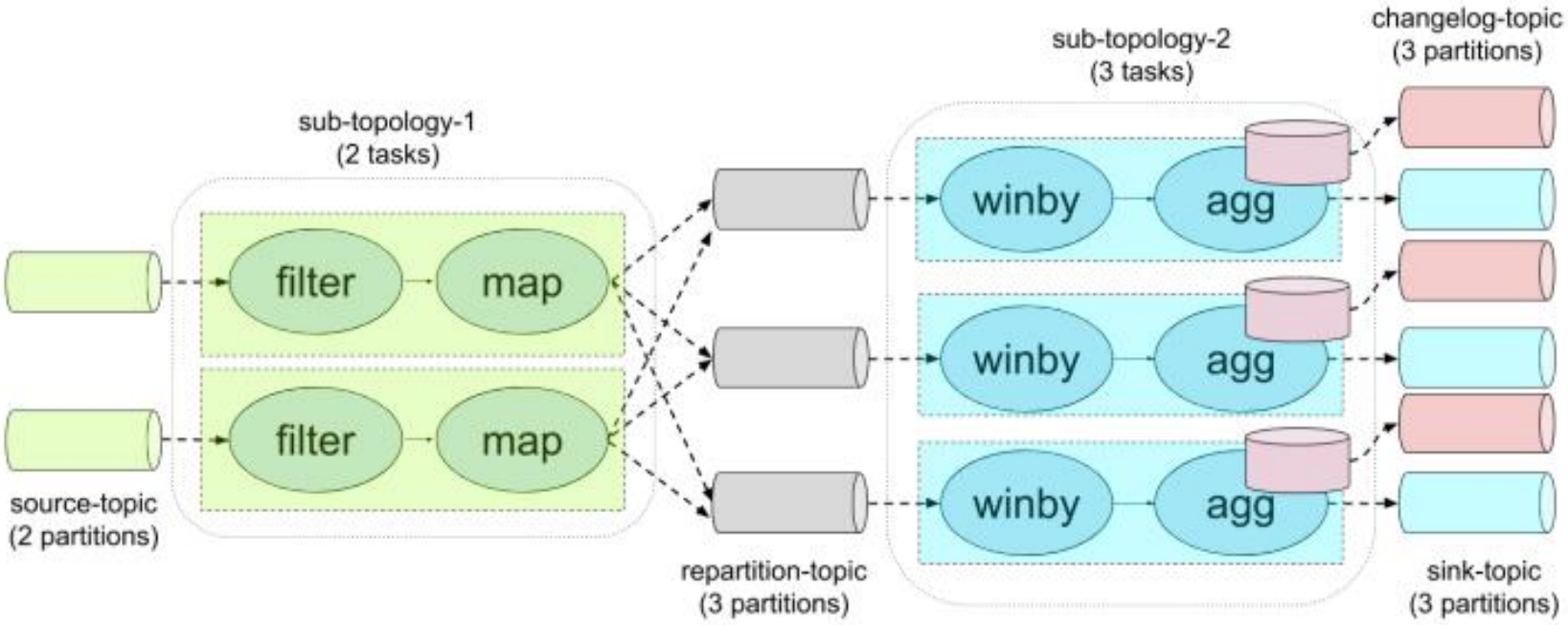
提供了事务保证，在事务开始后，向一个或者多个中的写数据要么后成功可见，要么失败，不会被消费者见到



的端到端一致性实现

然后介绍的基本架构，每个计算节点从一个的中数据，计算完成后将数据进下一个。

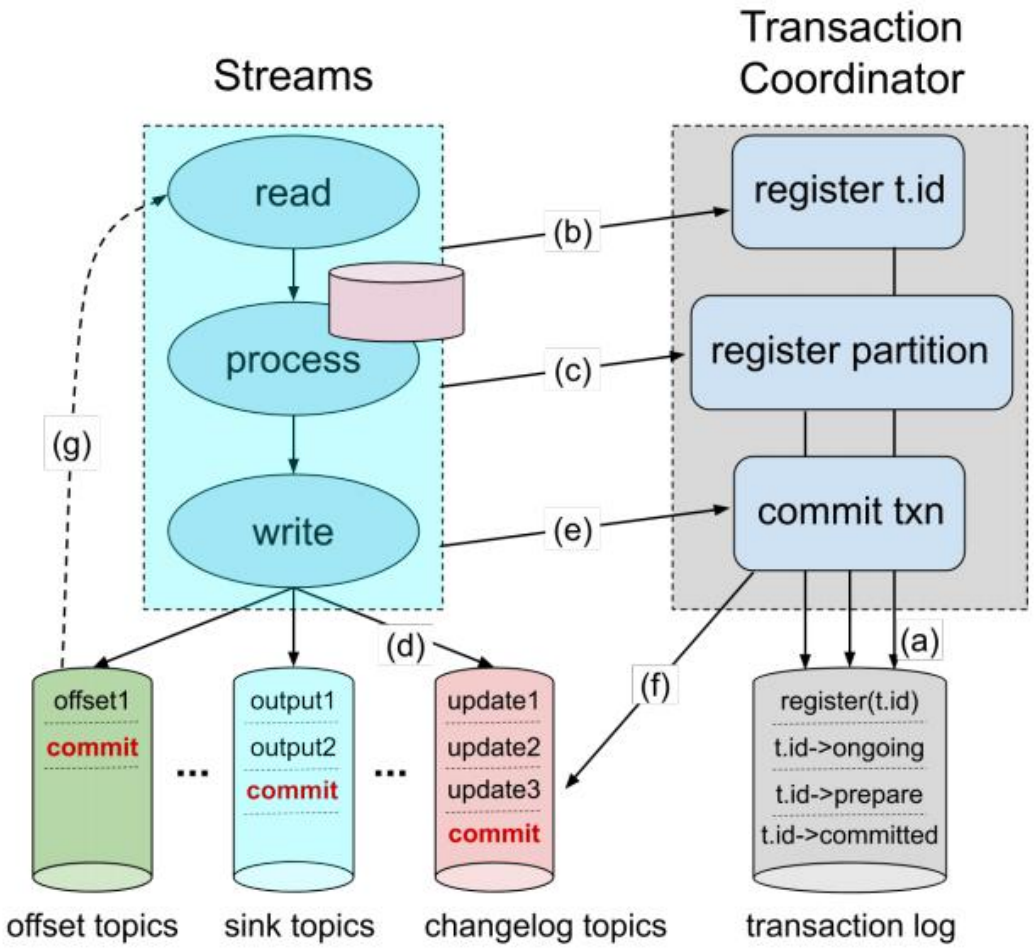
内部维护着和和，用来维护中间结果，和处理进度。



的端到端一致性实现

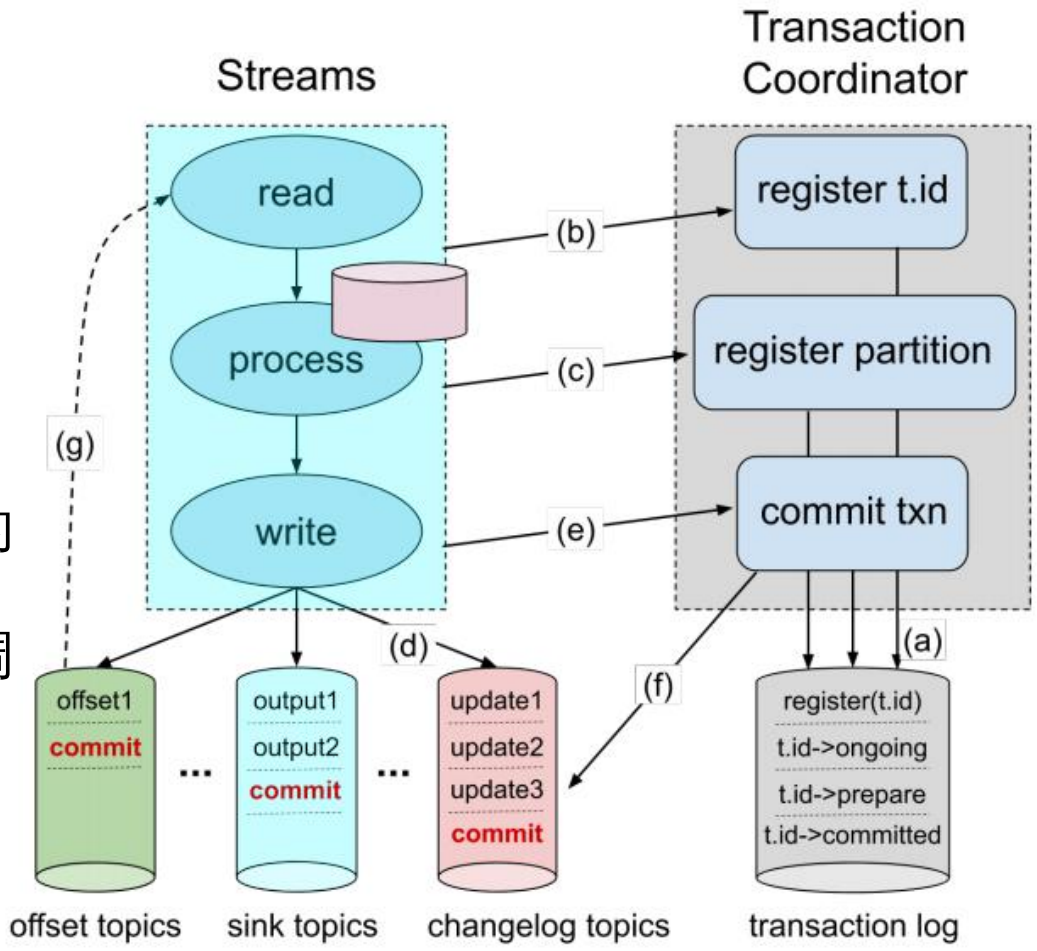
然后介绍如何通过事务保证中间结果、进度、的原子写。

事务与普通事务相比，最大的不同是可以同时写多个不同的，更像是一个过程。



的端到端一致性实现

协调者事务的读写者，维护一些事务的元信息，负责事务的分配，处理。
计算节点注册一个协调者写
当要写的某个时，先向协调者注册
向中写入数据
写操作结束后，向协调者发送提交请求（中的预提交），协调者写入
协调者向每个中写一个，表示事务完成，协调者写入



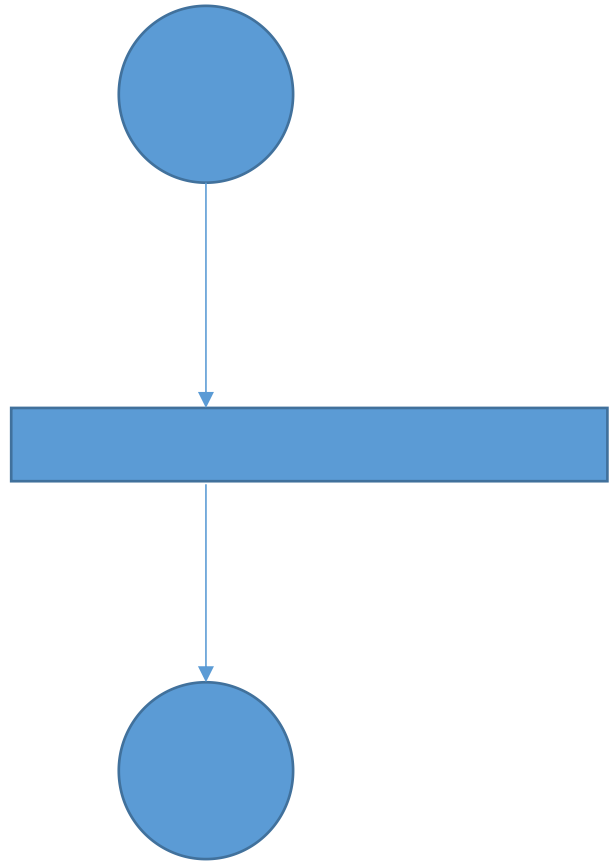
的端到端一致性实现

去重策略

的下游计算节点读取中间的数据作为输入，只要保存处理进度，就不会收到重复数据。

所以要保证上游计算节点每条数据只写一次，叫作幂等生产者。

通过事务方式写下游的，开启事务后通过当前事务中维护一个递增的来去重，如果节点，协调者会保证重启的计算节点会申请到和此节点相同的事务，从而丢弃之前的写操作，防止僵尸节点写。

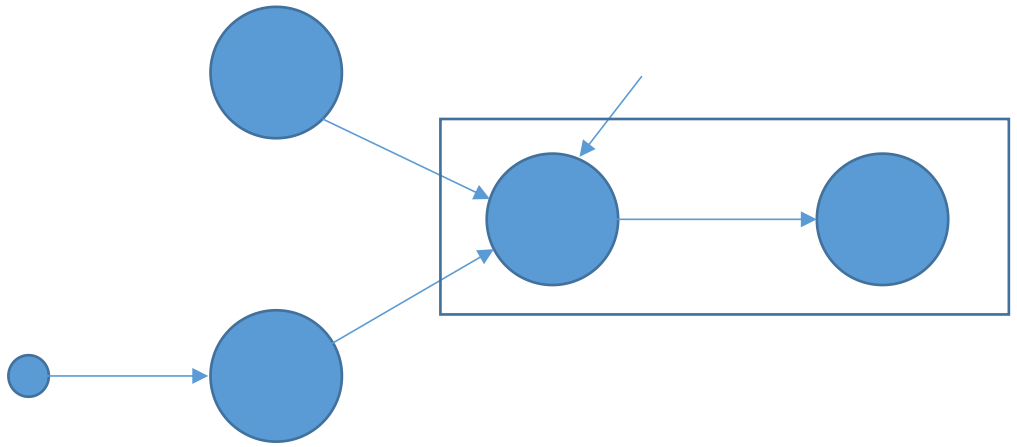


的端到端一致性实现

之前我们已经讲过和的数据处理

将每个计算节点的中间结果保存在上，这符合上述的和实现原理，不过和对一致性做了更高的保证并维护来支持流计算。

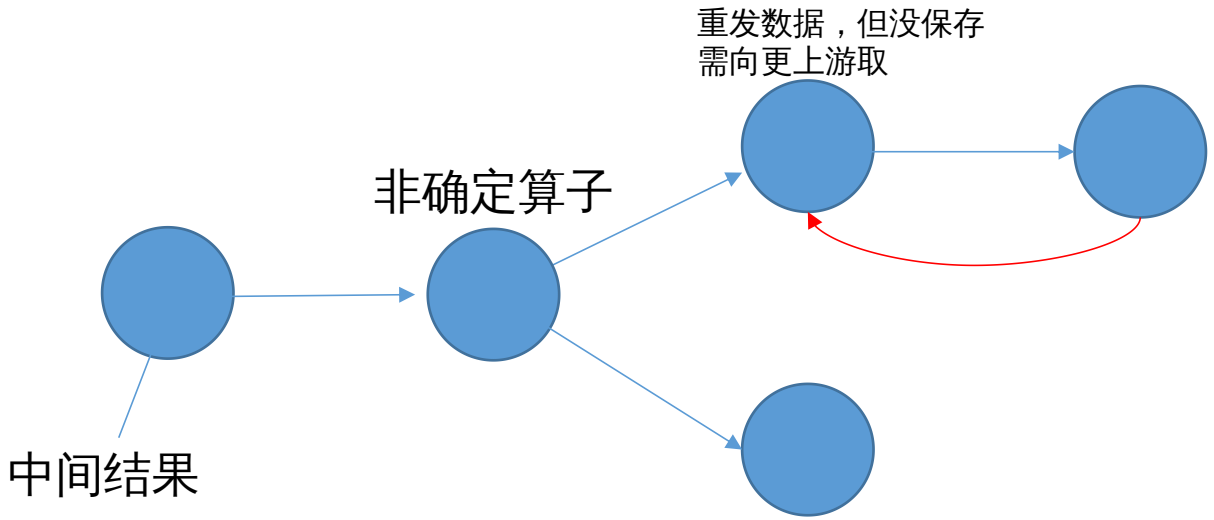
不保存每个计算节点的中间结果，而是通过血缘关系找到上游存储了中间结果的节点，再通过重算策略重放节点的输入数据，也就是说是一个存储中间结果重算的数据计算平台。



的端到端一致性实现

要改造使其可以处理流数据，首先要使离散的批数据在相同上共享一个，此时考虑下图的情况，重算数据时无法保证非确定性计算的一致性。

流处理有两个版本

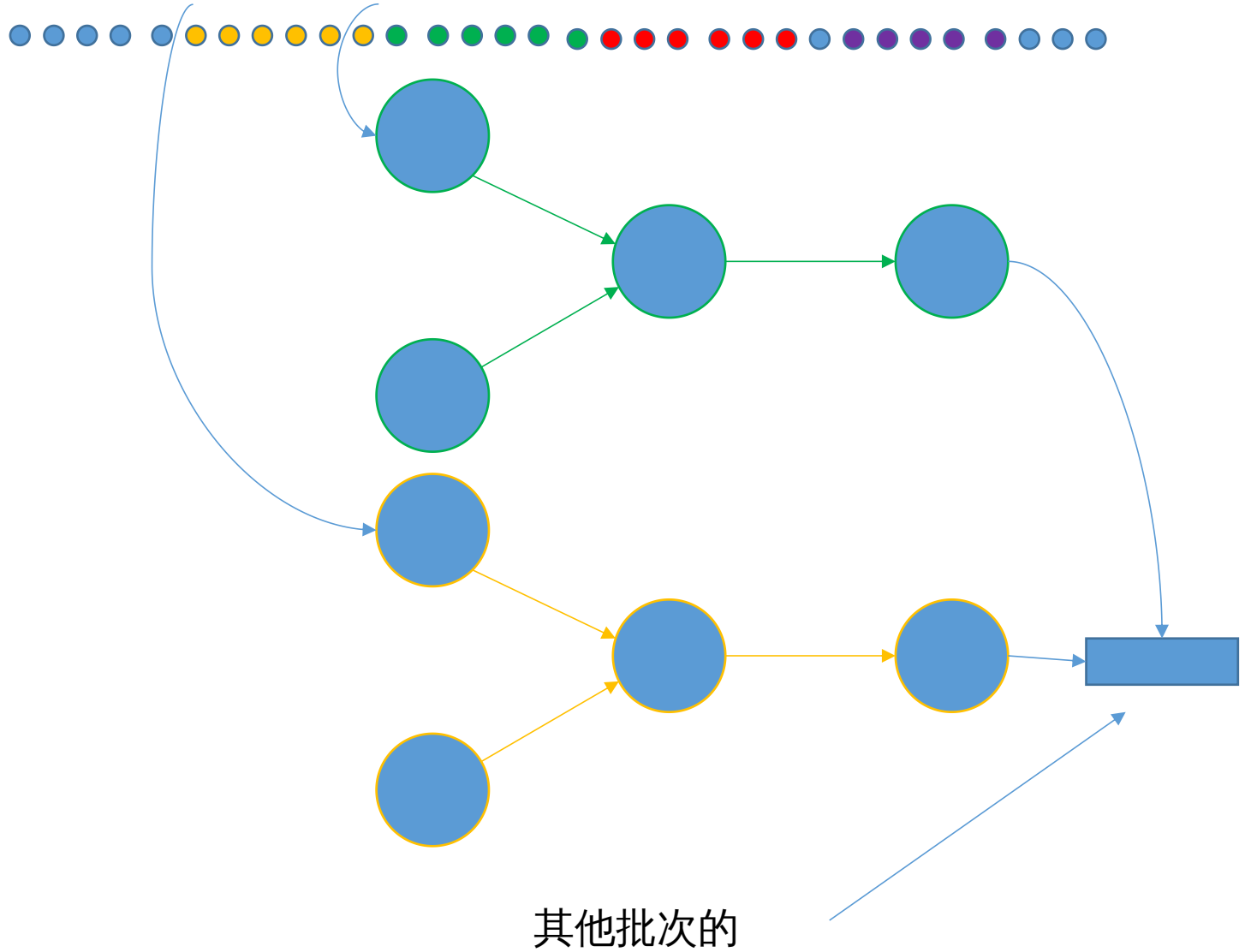


的端到端一致性实现

把运动着的流数据按照时间分成一个个小的。

然后每个微批之间独立计算，但维护共同的，实现增量计算。

为每个微批的每个数据流图中的每个都维护血缘关系，开销大，同时延迟高



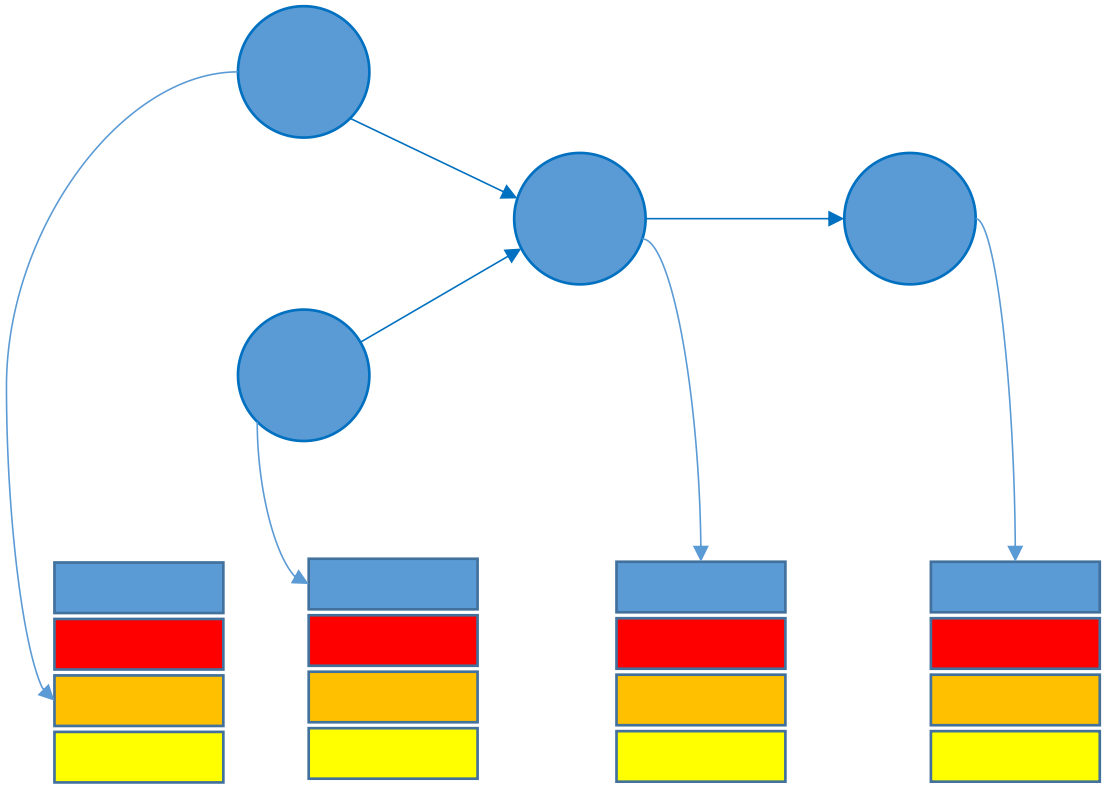
的端到端一致性实现

条数据，触发
三次计算

基于和的关系。

是随着时间推移的聚合结果，将到达的流数据一条条推入数据图，同时在每个上做增量计算，相当于维护了一个大批次。

此时部分解决了实时性问题，但同样无法应对非确定性计算



的端到端一致性实现

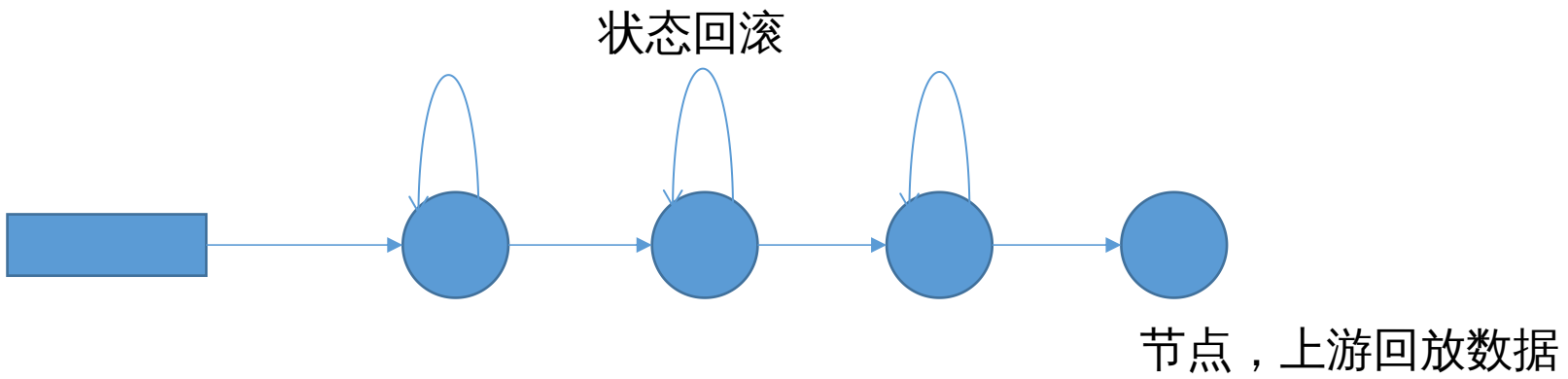
：完全依赖重算实现的流计算平台，完全不保存中间结果

首先重新观察完全依赖重算需要面对的问题

数据源需要回放数据，会产生链式反应，直到流系统输入端的队列

如果发生回放，此时所有被涉及的算子都应该回退自己的状态。

回放后重新计算，此时如果是非确定性计算，那么两次的最终结果必然不一样，此时需要保证只有一个结果可以输出到外部。



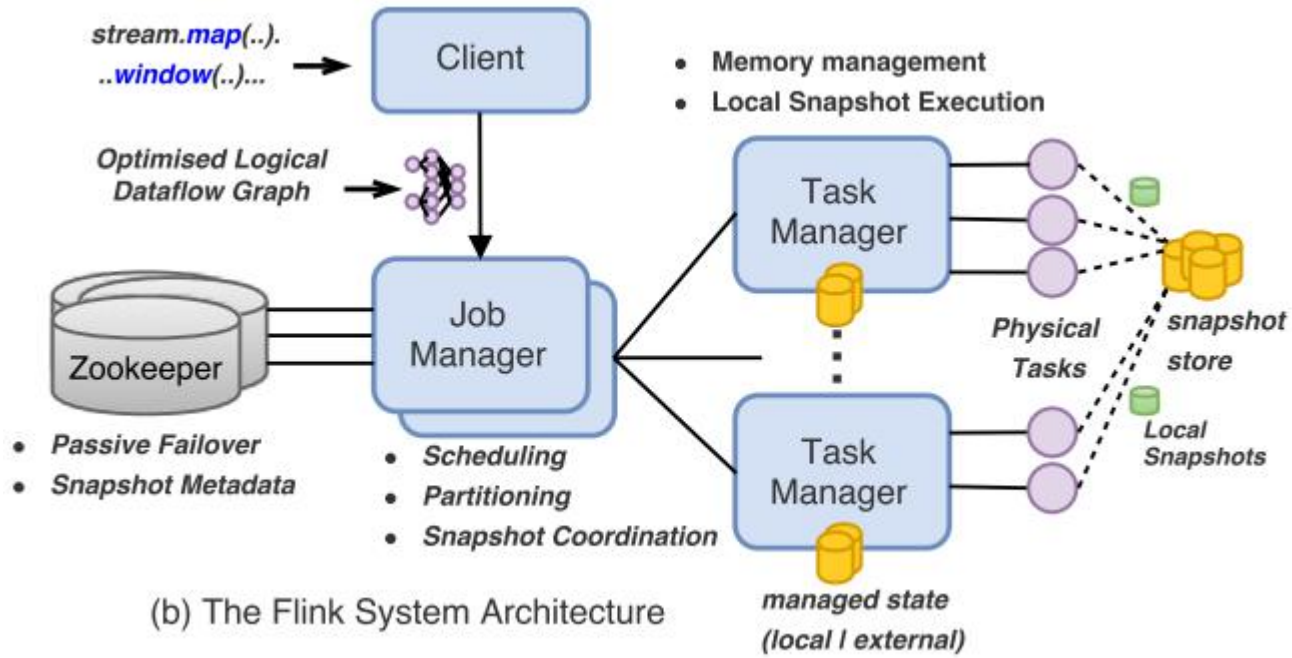
的端到端一致性实现

右图是的整体架构：

：优化数据流图，并发送给

：系统的协调者，调度任务，将执行图分割成不同的阶段，并负责实现快照

：负责实际的执行计算任务。



(b) The Flink System Architecture

的端到端一致性实现

全局一致点

考虑输入的数据为形式，则当全部处理完成并提交，且未进入系统时，此时系统的快照就是一个全局一致快照，在这里采用了类似分布式快照的方法，来异步的生成全局一致快照。

的也就是协调者每隔一段时间发出一个特殊的数据，来触发中计算节点的快照建立机制。

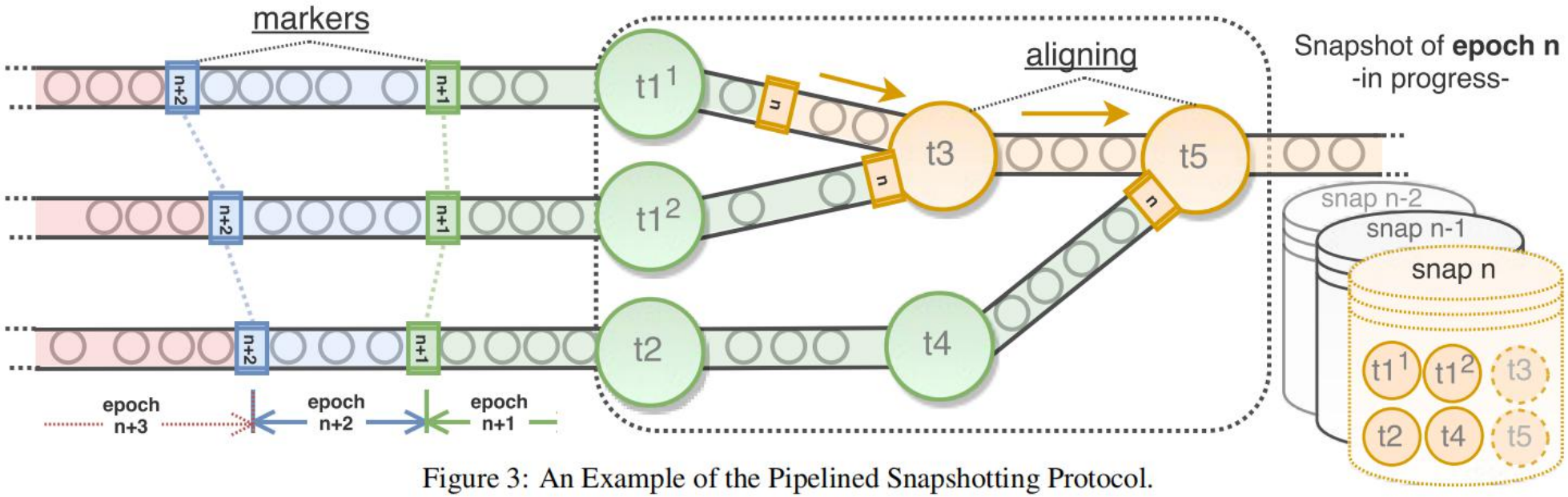


Figure 3: An Example of the Pipelined Snapshotting Protocol.

的端到端一致性实现

每个节点只需要按照如下策略

节点只使用之前的所有数据来建立自己的快照，并保存到一个外部高可用的中
只使用之前的数据计算结果并发送给下游后再转发，然后处理之后的数据。这样就可以保证节点的不会被其他批次的数据影响

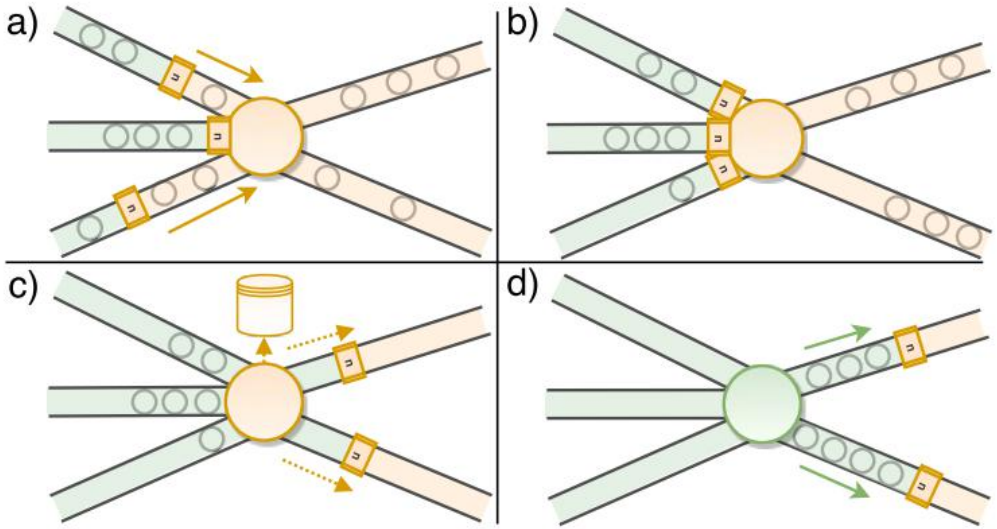
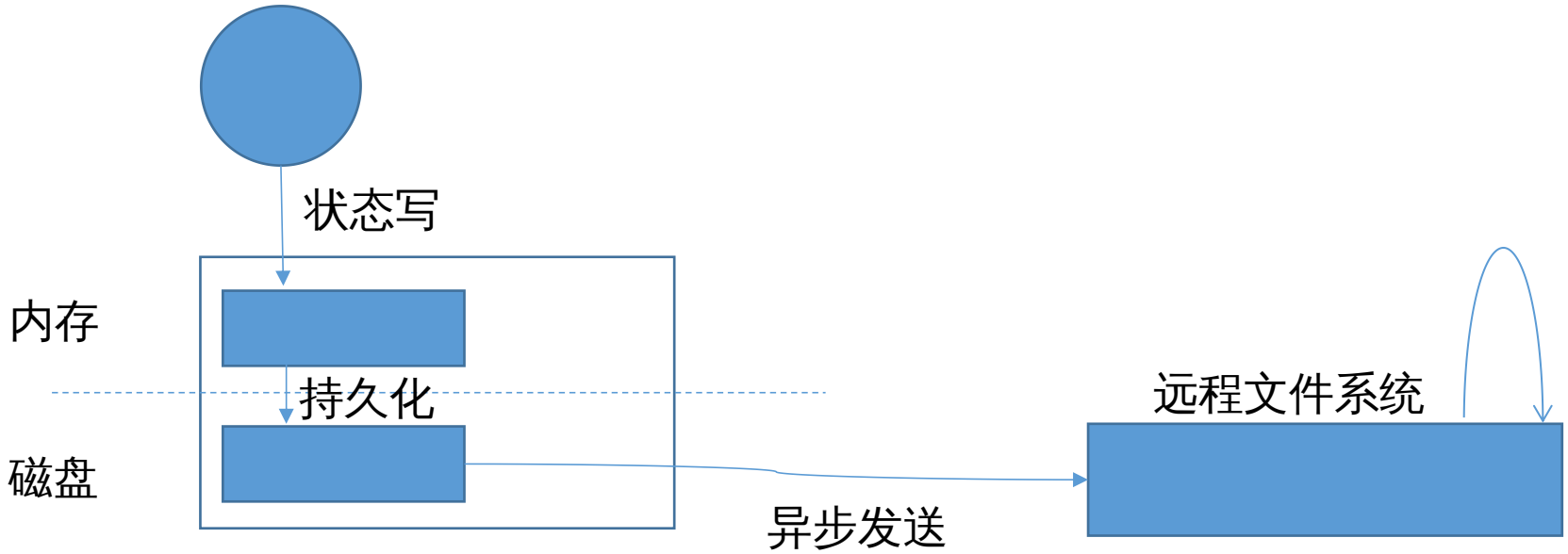


Figure 4: Alignment and Snapshotting Highlights.

的端到端一致性实现

重看的快照实现方法，快照的内容是节点的状态，也就是说在做快照时，需要阻塞输入，这显然是不合理的延时。

使用异步增量，将写入，这样只需要在做快照把中的数据写入硬盘的时候阻塞输入即可，然后把生成的异步地发送到远程文件系统达到异步增量快照

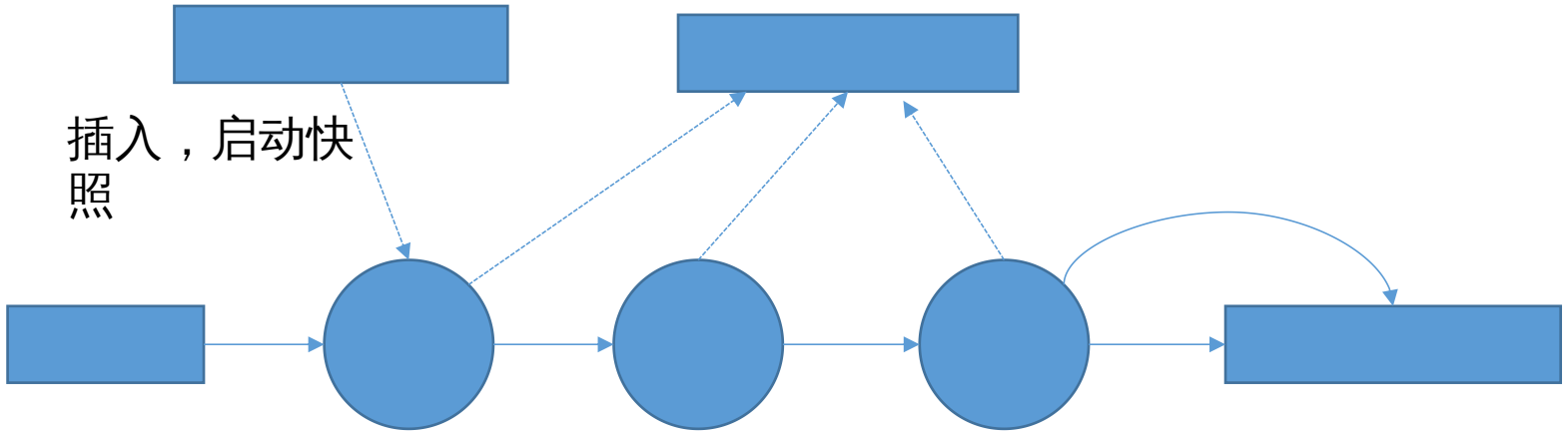


的端到端一致性实现

和都可以保证端的数据只发送一次，那么在完全重算的情况下可以实现的精确一次吗？

重看全局快照的实现，会发现全局快照也是一个类似的过程

插入相当于协调者向众节点发送预提交请求
节点收到所有并建立快照，相当于告诉我可以完成请求
所有节点完成快照后，告诉各个节点可以



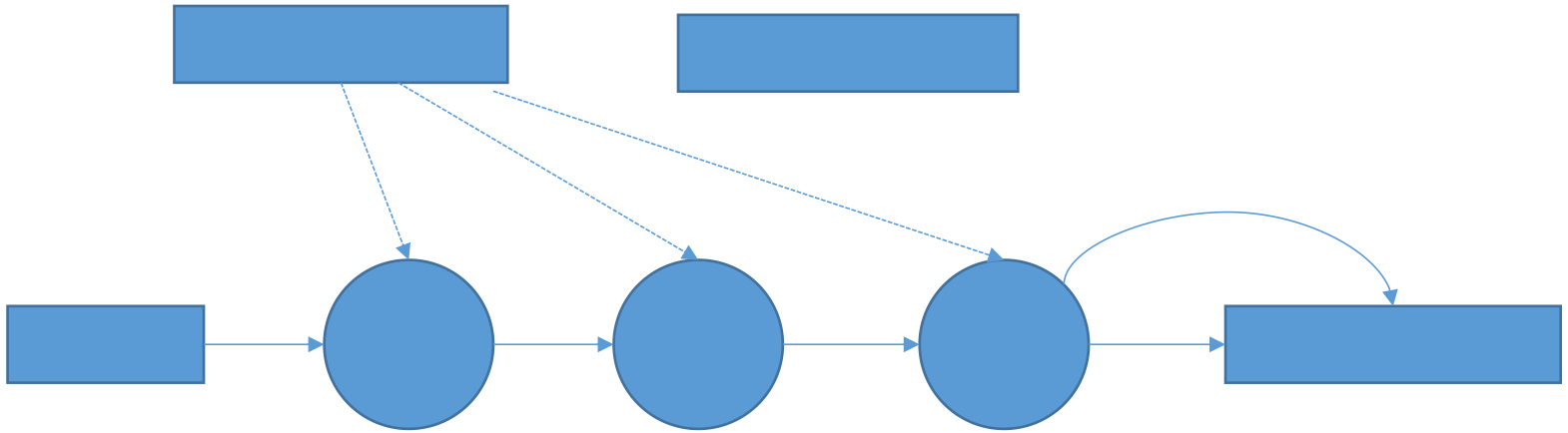
在建立快照前，收到的数据可以立刻发出吗？建立快照后呢？

的端到端一致性实现

和都可以保证端的数据只发送一次，那么在完全重算的情况下可以实现的精确一次吗？

重看全局快照的实现，会发现全局快照也是一个类似的过程

插入相当于协调者向众节点发送预提交请求
节点收到所有并建立快照，相当于告诉我可以完成请求
所有节点完成快照后，告诉各个节点可以

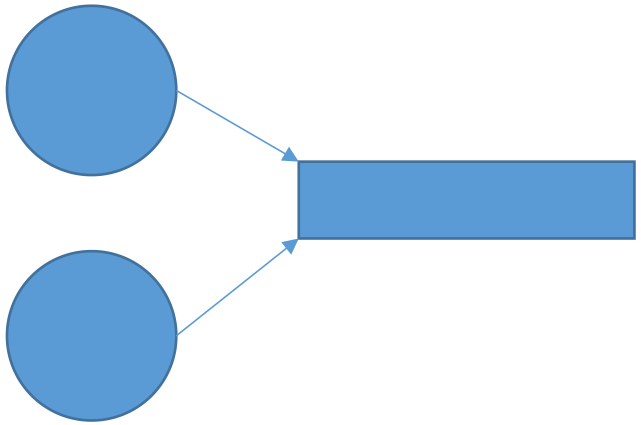


在建立快照前，收到的数据可以立刻发出吗？建立快照后呢？

的端到端一致性实现

显然的数据既不能在快照前发出，也不能再收到之前发出，只能在接收到后发出数据。

解决面对非确定计算重复输出的方法，就是在计算节点上保存结果到持久化存储，当收到时，发出数据，相当于一个微批。



的数据已经开始生成

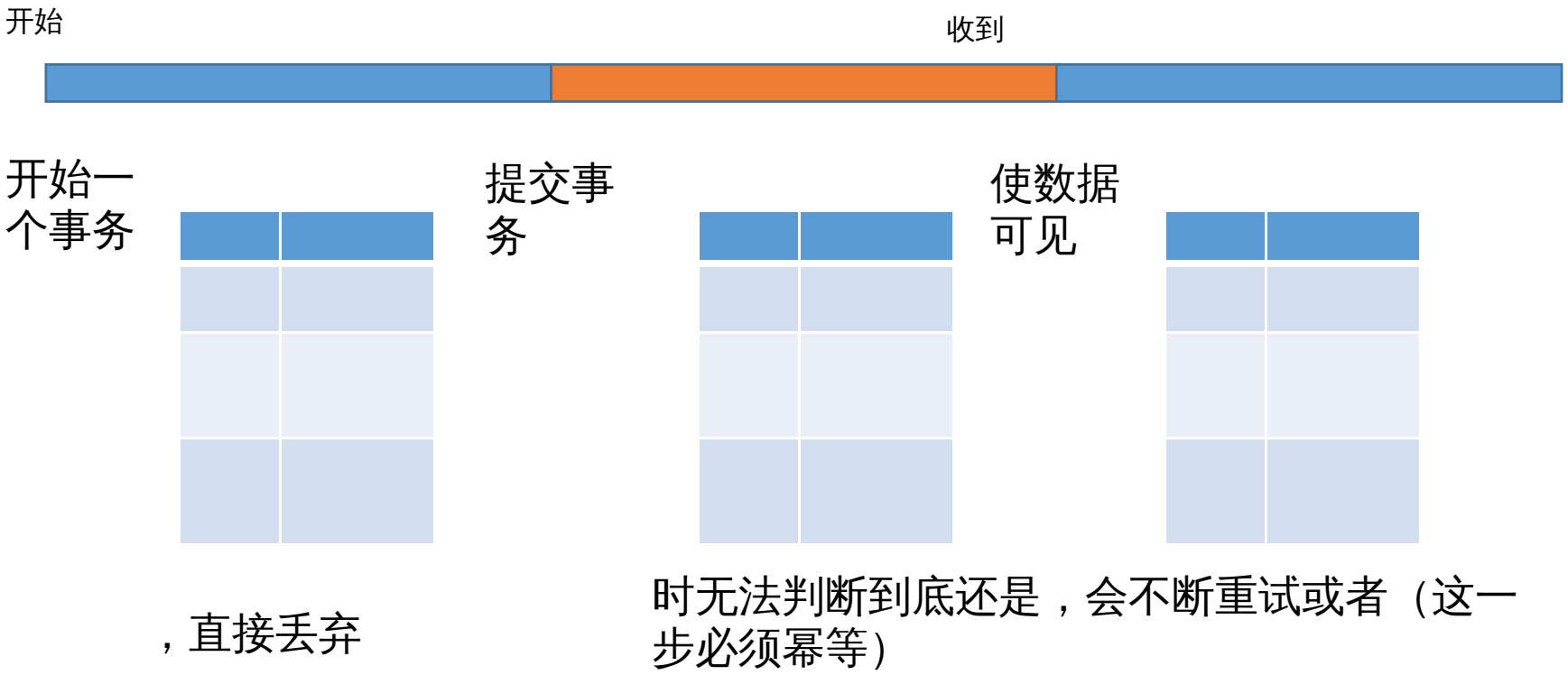
数据依然不可见，防止其他

数据可见

的端到端一致性实现

但没有使用这个方法，而是提供了一个两阶段提交的接口，让用户来保证的一致性，同时还要保证用户不会读写状态为的记录

以支持事务的后端为例。



的端到端一致性实现

使用保存中间的结果的方式支持数据回放，同时通过为所有消息分配一个唯一保证去重。已经处理过的、中间数据。状态数据通过事务存储在自家的中

同样使用保存中间结果的方式支持数据回放，所有的进度、中间数据、状态都保存在中，并通过一个幂等的生产者，保证不会产生重复数据

试图在自身的批处理框架上实现流处理，使用微批或者隐藏的大批次实现，同时在计算节点上使用中间结果（快照）重发混合的方案实现重发，不支持非确定计算时的端到端一致性。

通过全局一致点快照实现了一个完全基于重算的流处理平台，可以支持非确定性计算的端到端一致。