流计算引擎弹性扩展综述

汇报人：王仁杰

分布式存储与计算实验室

2022.04.29

目录

01 流系统扩展概述

01 流处理系统概述概述

- 逻辑执行计划是一个有向无环图
- 节点对应数据处理
- 边对应数据通道
- 最终每个节点由分布在物理机上的多个实例执行

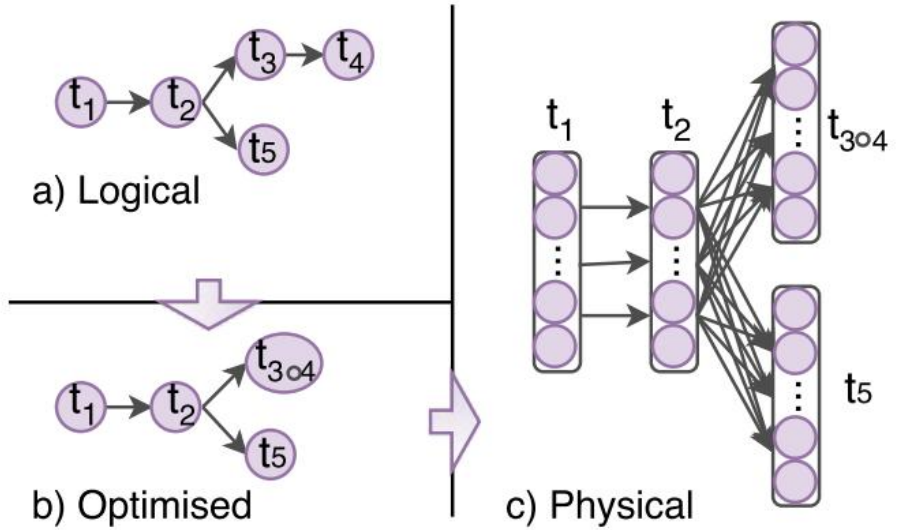
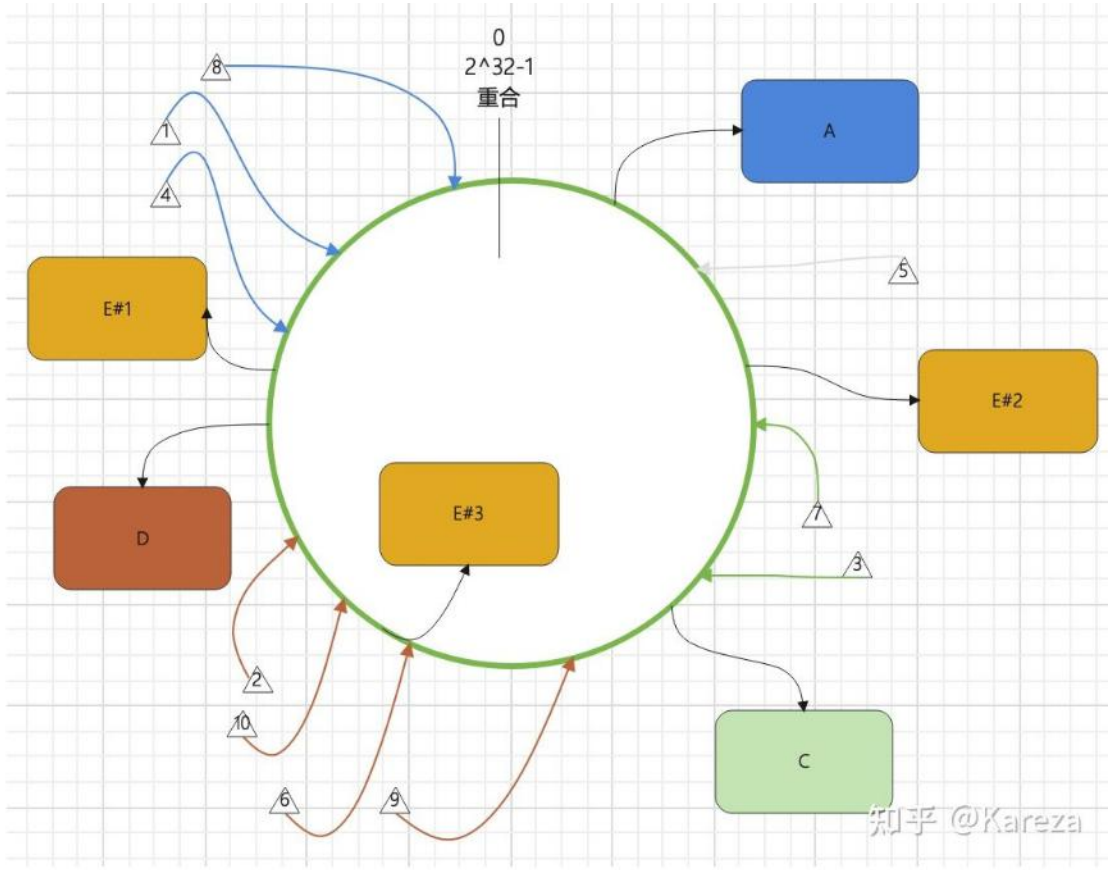


Figure 2: Dataflow Graph Representation Examples.

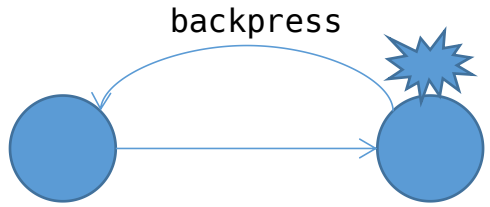
01 流处理系统概述概述

为了实现负载均衡，每个算子实例处理的key往往通过一致性hash+虚拟节点的方式实现



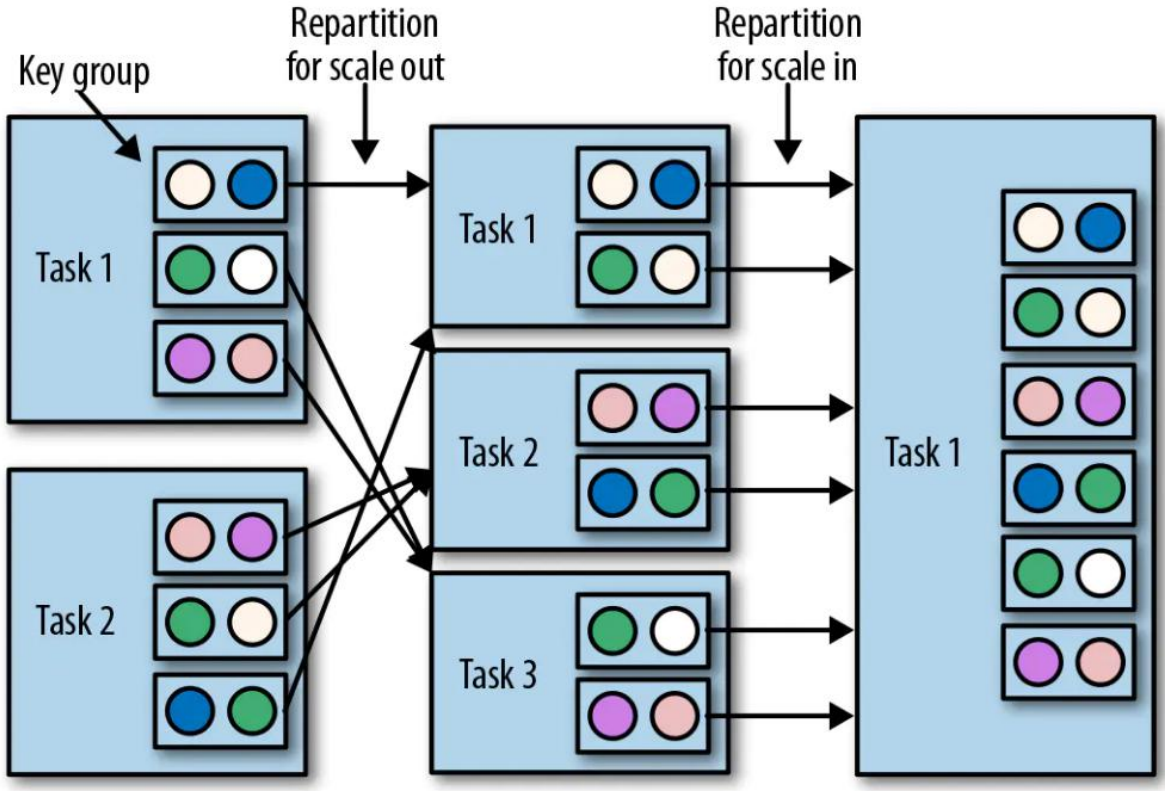
在物理计划执行时，一般通过一致性hash+虚拟节点的方式实现负载均衡

01 流处理系统概述概述



下游算子负载过高，输入队列溢出，导致背压，从而拖慢整个处理

- 流处理任务往往是一个long-running任务
- 流系统面对的输入数据大小波动大
- 需要对算子的实例进行scale-out或者scale-in
- 弹性的流处理系统保证系统对不同的负载的性能和延迟。



01 流处理系统概述概述

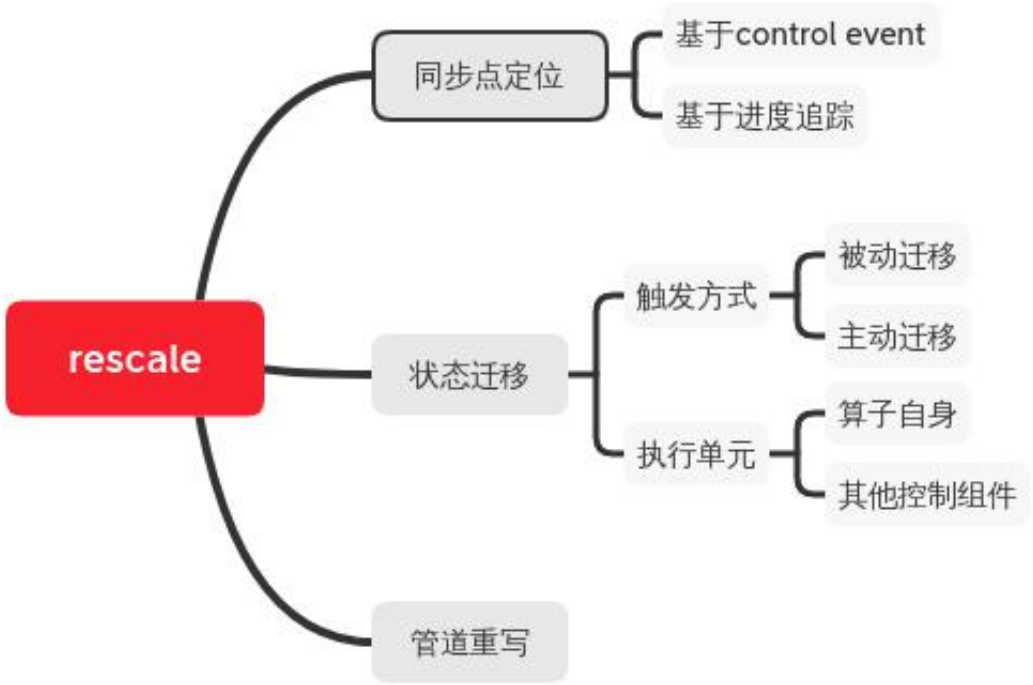
分布式存储与计算实验室

流处理系统重配置过程，主要面对以下三个问题：

1．如何定位一个同步点，使得同步点前的数据流使用旧配置，同步点后的数据流使用新配置

2．状态迁移，随着算子实例个数的变化，对应实例维护的状态也需要相应的移动

3．算子路由通道重写，重配置后，需要修改受影响的算子实例的路由表



02 ／ stop-and-restart 方式

02 stop-and-restart 方式

传统的流处理系统，如Flink，Spark stream、struct stream使用的重配置方法，实现简单。

暂停正在运行的流系统，根据新配置生成物理执行计划，对算子的状态进行迁移并重写算子的路由通道，最后重启流系统，对数据进行处理。

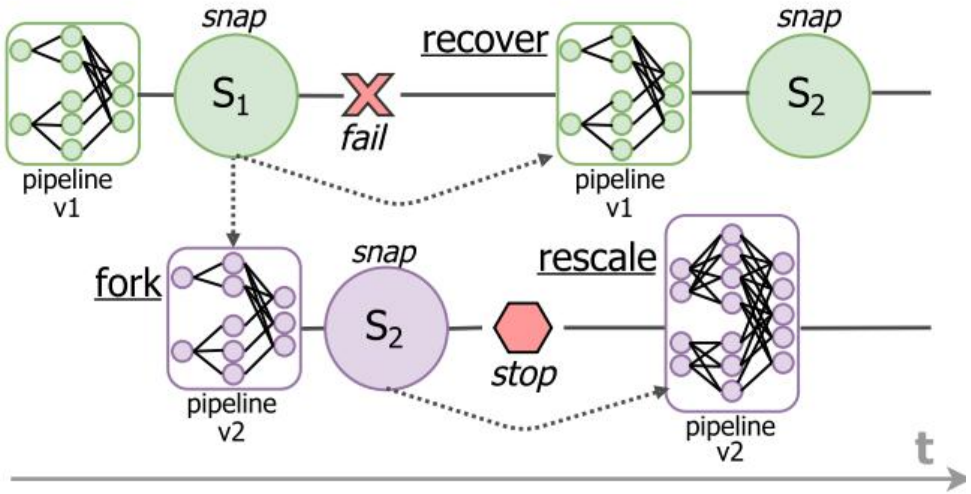


Figure 6: Snapshot usage examples.

02 stop-and-restart 方式

分布式存储与计算实验室

Flink的实现方式基于全局一致快照

Flink的协调者通过最近的快照实现状态的迁移，并生成新的物理执行计划，然后重启。

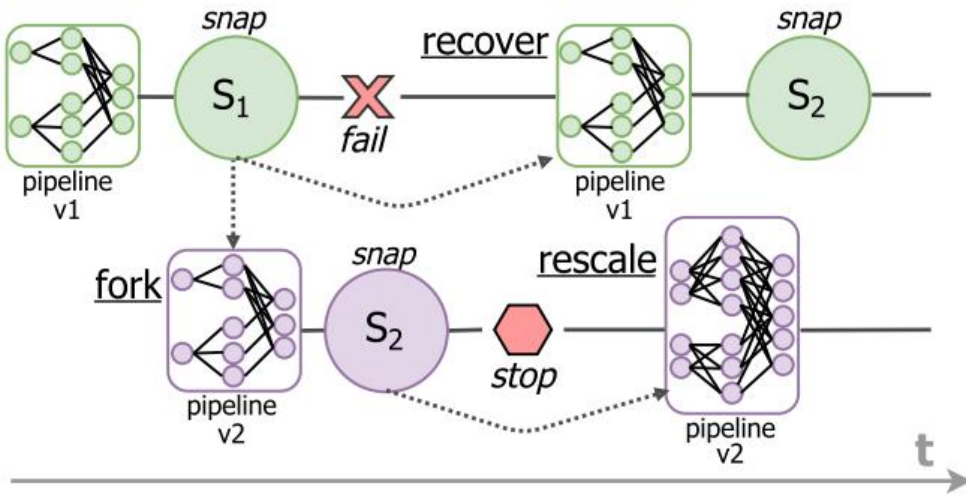


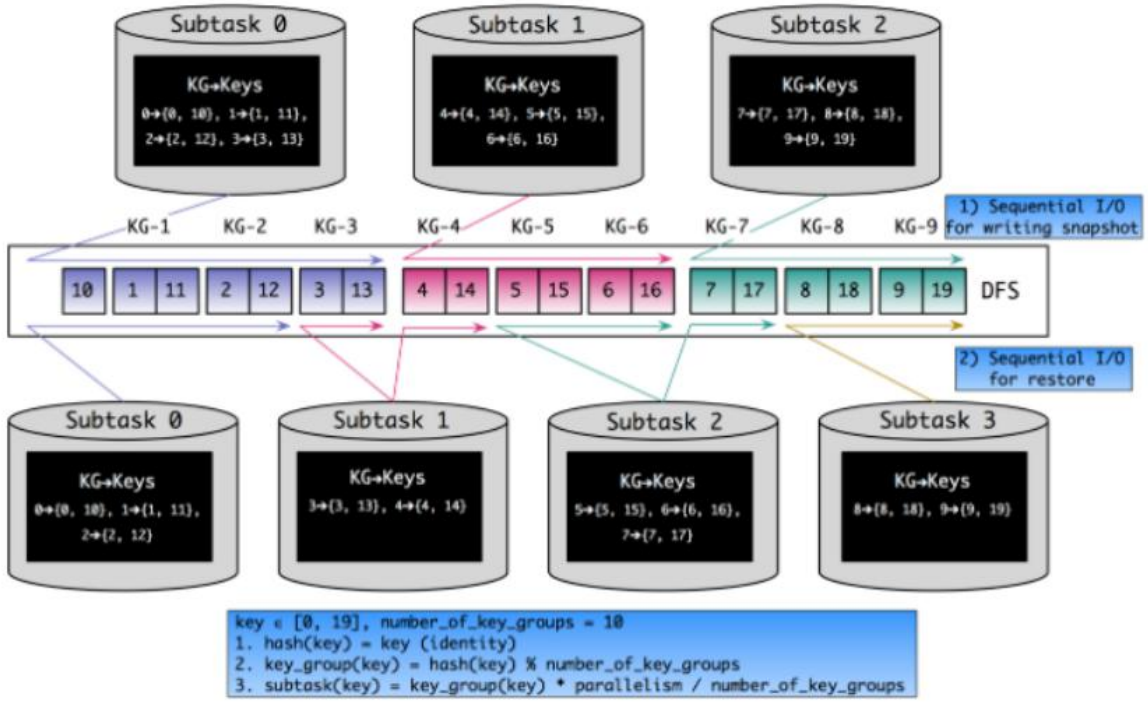
Figure 6: Snapshot usage examples.

02 stop-and-restart 方式

Flink为了减少rescale后的状态迁移规模，通过keyGroup和keyGroupRange来组织每个key对应的状态。

key的状态通过hash分为最大并行度个group
每个实例通过类似一致性hash的方式，获得自己的keyGroupRange

Flink在rescale的时候重新计算实例的keyGroupRange。




```
int start = ((operatorIndex * maxParallelism + parallelism - 1) / parallelism);
int end = ((operatorIndex + 1) * maxParallelism - 1) / parallelism;
```

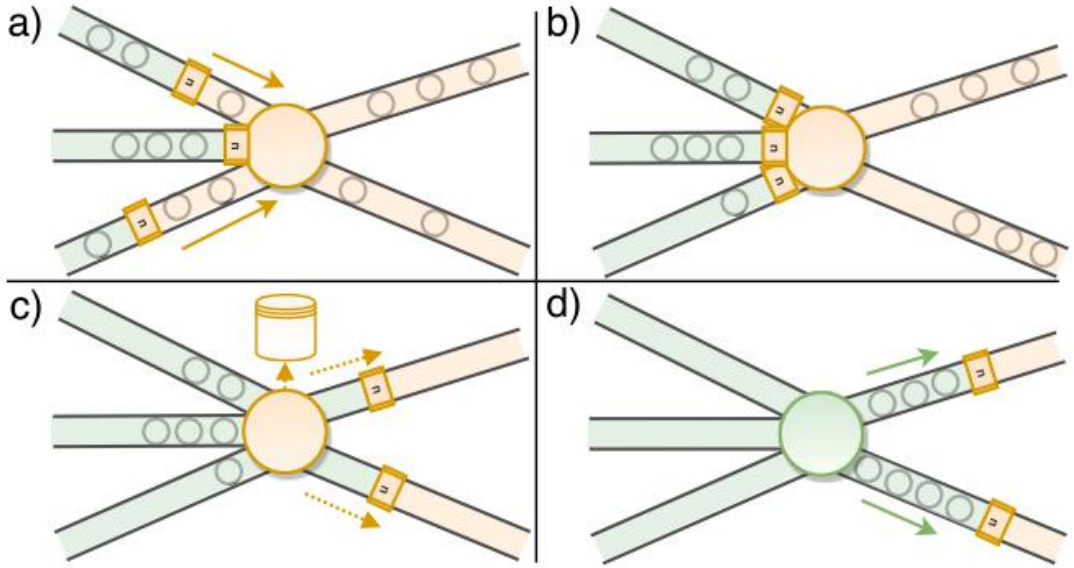
03 ／ partly-stop-and-restart 方式

03 partly-stop-and-restart 方式

对于大部分rescale而言，只会更改系统中一部算子的实例个数，没必要完全暂停数据流。

chi通过control event的方式实现rescale

control event在单个算子上的处理过程类似Flink的checkpoint event。

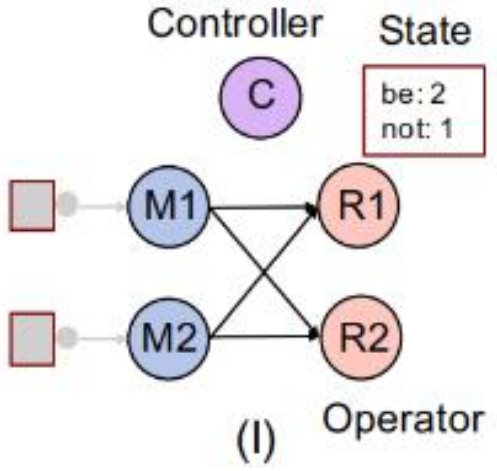


03 partly-stop-and-restart 方式

chi通过control operator来监视整个dataflow。

当触发某些控制决定，如checkpoint，rescale时

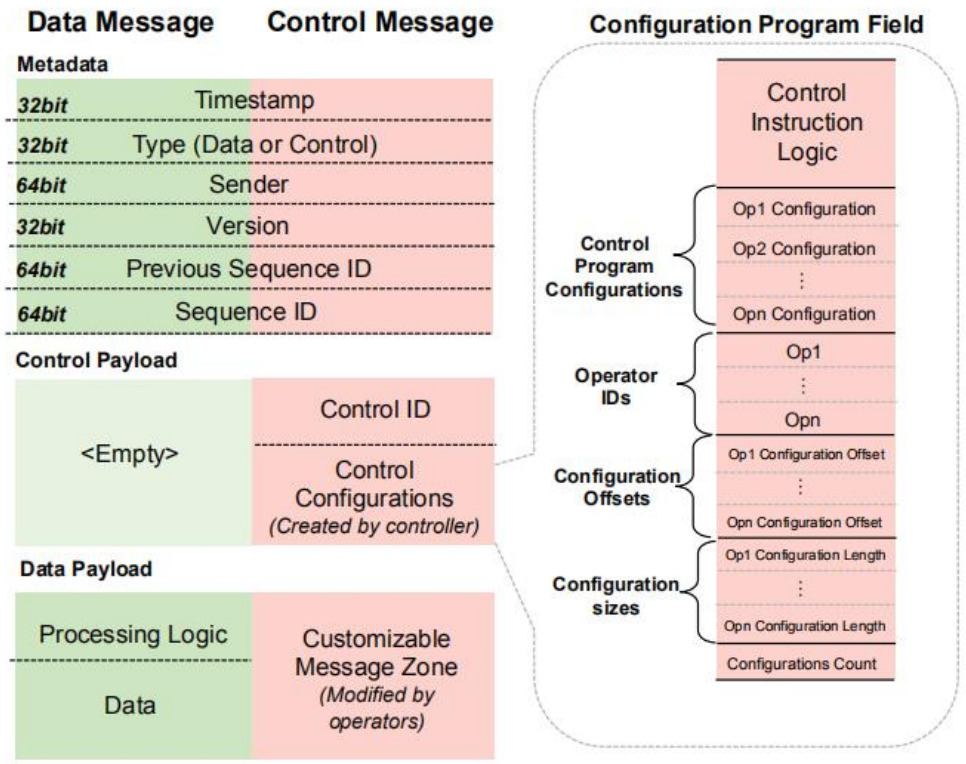
初始化一个control event并广播给所有的Source operator，event流过dataflow ，最终sink operator将control event返回给control operator实现整个控制流程。



03 partly-stop-and-restart 方式

分布式存储与计算实验室

chi的控制过程对用户完全透明，并且可以通过在control msg上附加代码的方式控制算子实例的运行。



partly-stop-and-restart 方式

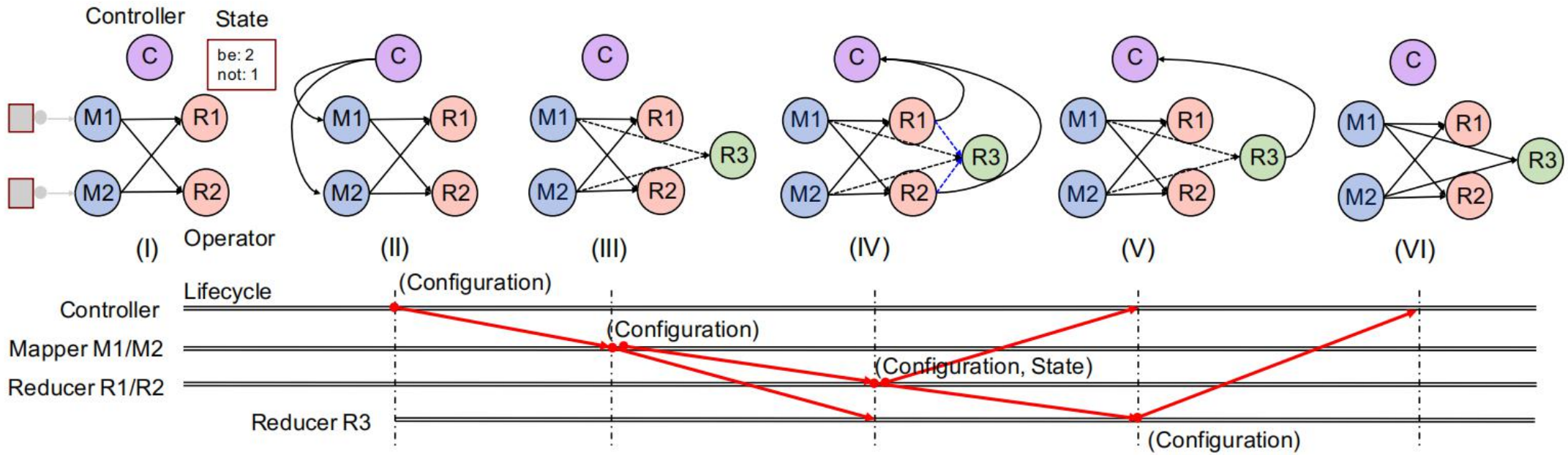


Figure 2: Scaling-out control in action where the user is interested in changing the number of reducers in Example 1

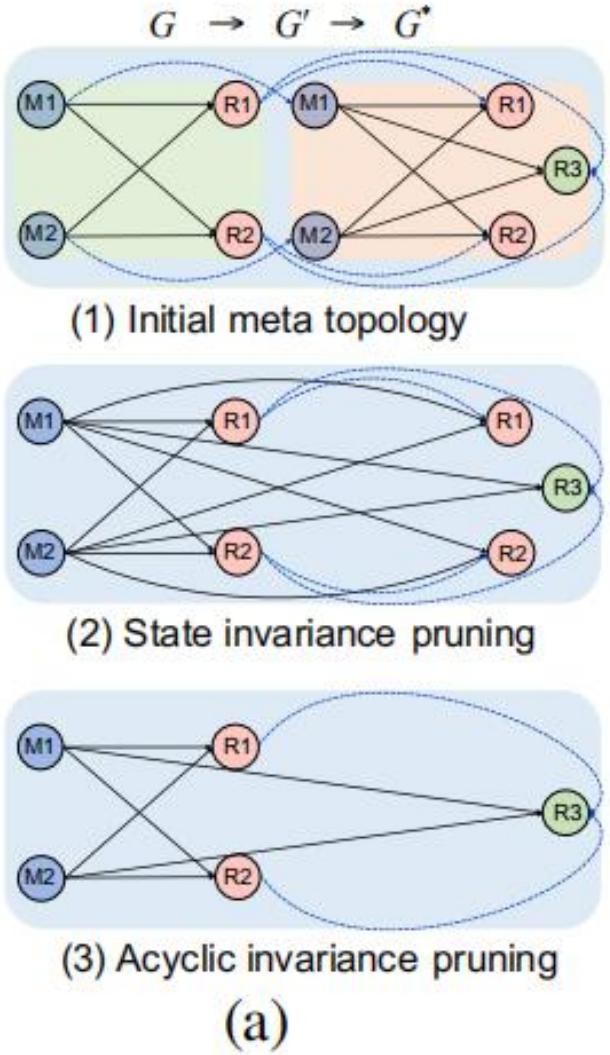
chi的完整控制过程如上图

03

partly-stop-and-restart 方式

为了实现从就dataflow G到新dataflow的G*的修改，control opreator需要通过下面两条原则生成配置修改计划。

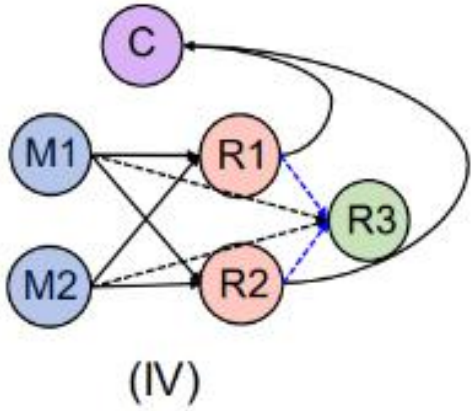
1.当算子状态不变时，复用算子实例。
2.在保证图上无圈的情况下折叠算子实例



03 partly-stop-and-restart 方式

chi的rescale与Flink的checkpoint非常相似，那么为什么Flink没有使用这种rescale方式呢？

Flink算子的实例之间缺少通道来传输状态，并且算子实例缺少runtime支持，如state change能力支持。



(IV)

03 partly-stop-and-restart 方式

chi在Flare上实现，
dizzle是spark的一个分支，对流处理做了优化。

可以看到chi在两个吞吐量和延时上都拥有更好的表现。

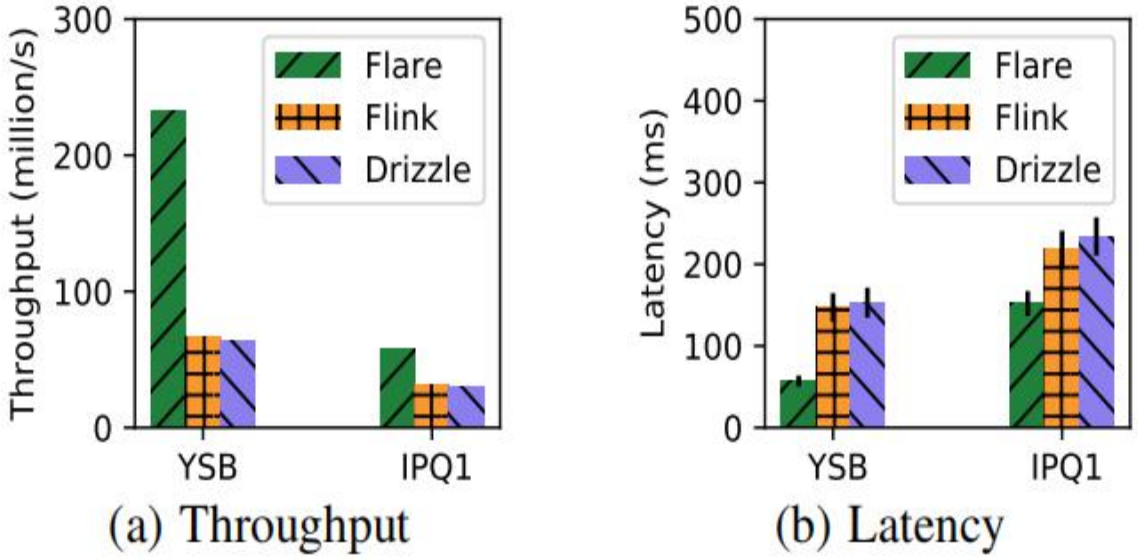
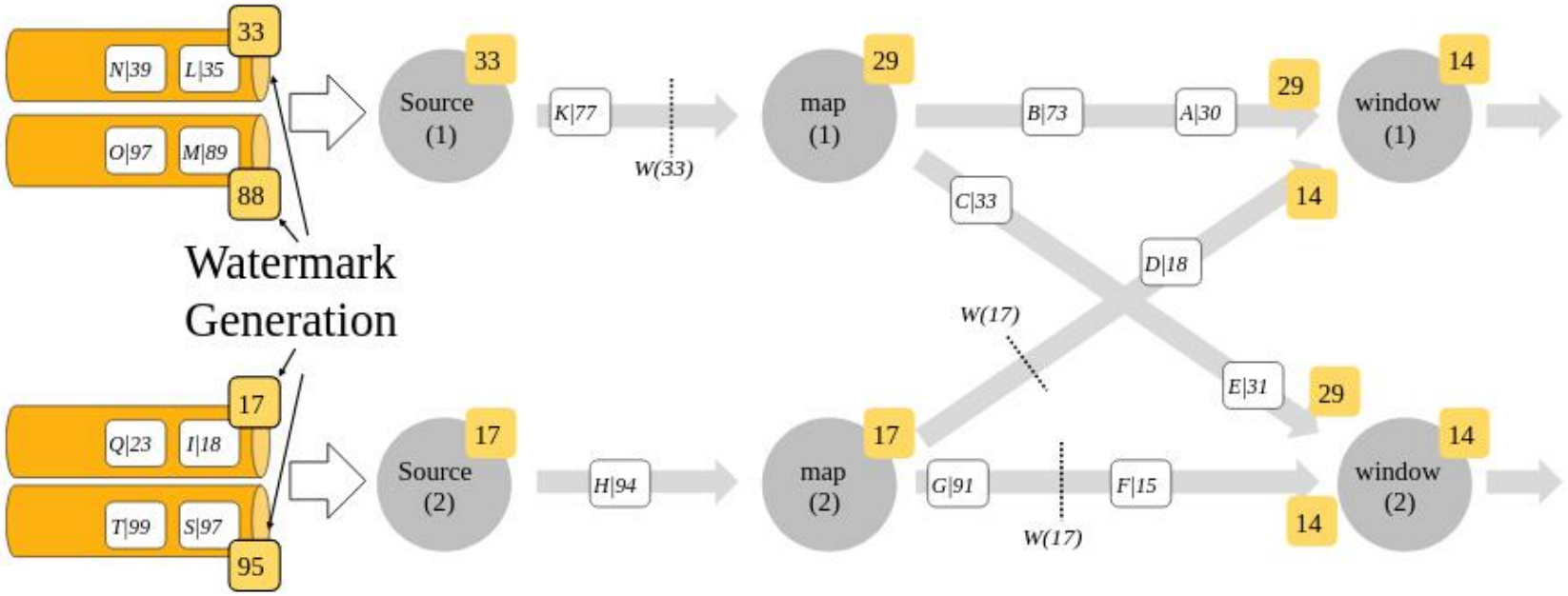


Figure 7: Flare's throughput and latency against Flink and Drizzle for the YSB and IPQ1 workload

04 ／ 基于处理进度跟踪的细粒度方式

04 处理进度跟踪



流处理系统为了确定数据的完整性，需要对进度进行跟踪
在Flink等有向无环dataflow类型的流处理引擎中，追踪进度的屏障标志，叫做watermarker，表示算子不会收到小于watermarker的记录。

04 处理进度跟踪

Timely dataflow通过逻辑时间戳和全局时间监控，实现了一个在复杂有向图上的进度追踪机制。

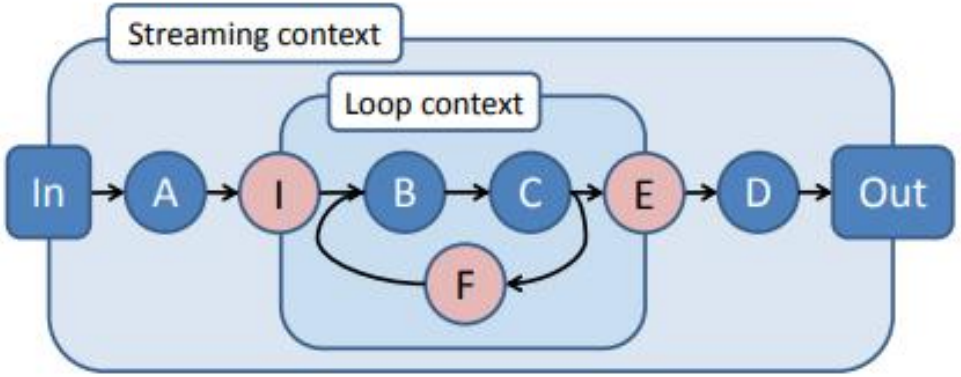


Figure 3: This simple timely dataflow graph (§2.1) shows how a loop context nests within the top-level streaming context.

04 处理进度跟踪

每个顶点v上需要实现两个回调函数
v.onRecv(edge,msg,time)
v.onNotify(time)

同时在回调函数的上下文中，可能会调用两个系统提供的方法。
this.SendBy(edge,msg,time)
this.NotifyAt(time)

```
class DistinctCount<S,T> : Vertex<T>
{
  Dictionary<T, Dictionary<S,int>> counts;
  void OnRecv(Edge e, S msg, T time)
  {
    if (!counts.ContainsKey(time)) {
      counts[time] = new Dictionary<S,int>();
      this.NotifyAt(time);
    }

    if (!counts[time].ContainsKey(msg)) {
      counts[time][msg] = 0;
      this.SendBy(output1, msg, time);
    }

    counts[time][msg]++;
  }

  void OnNotify(T time)
  {
    foreach (var pair in counts[time])
      this.SendBy(output2, pair, time);
    counts.Remove(time);
  }
}
```

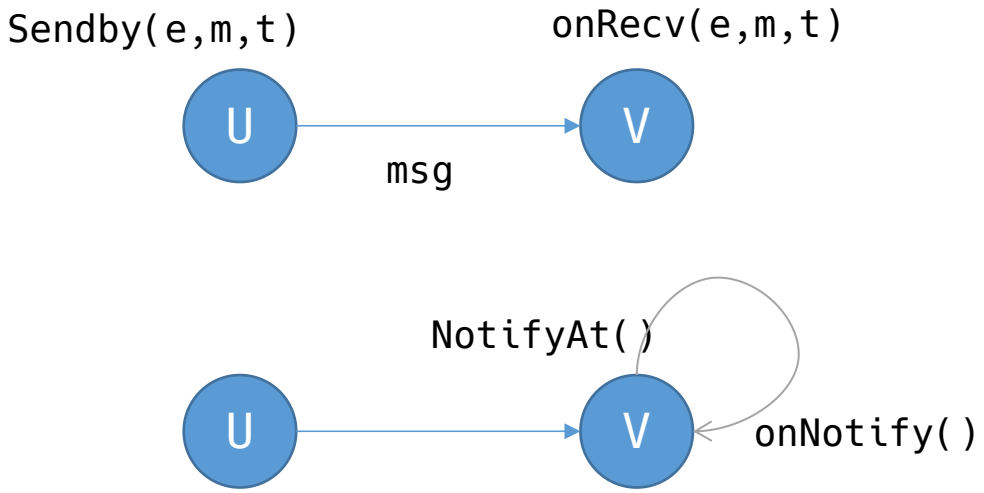
wordcount的一个例子，在output1上输出不同的msg，在output2上输出count结果

04 处理进度跟踪

几个函数的关系如下：

u.SendBy(e,m,t)会导致v.onRecv(e,m,t)
v.NotifyAt(time)可能会导致onNotify(t)

当onNotify(t)被触发时，表示不会有<=t的tuple会到达顶点。




```
class DistinctCount<S,T> : Vertex<T>
{
  Dictionary<T, Dictionary<S,int>> counts;
  void OnRecv(Edge e, S msg, T time)
  {
    if (!counts.ContainsKey(time)) {
      counts[time] = new Dictionary<S,int>();
      this.NotifyAt(time);
    }

    if (!counts[time].ContainsKey(msg)) {
      counts[time][msg] = 0;
      this.SendBy(output1, msg, time);
    }

    counts[time][msg]++;
  }

  void OnNotify(T time)
  {
    foreach (var pair in counts[time])
      this.SendBy(output2, pair, time);
    counts.Remove(time);
  }
}
```

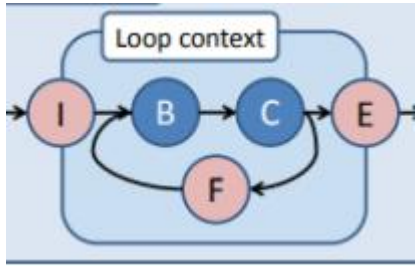
wordcount的一个例子，在output1上输出不同的msg，在output2上输出count结果

04 处理进度跟踪

Timely dataflow的逻辑时间戳如右图所示。

其中e在数据进入系统时由用户赋予。向量c初始化为[],在右表的情况下发生修改。

$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$



Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

定义：

当且仅当$e_1 \le e_2$ 且 $\vec{c_1} \le \vec{c_2}$（根据字典序比较）时，$t_1 \le t_2$

04 处理进度跟踪

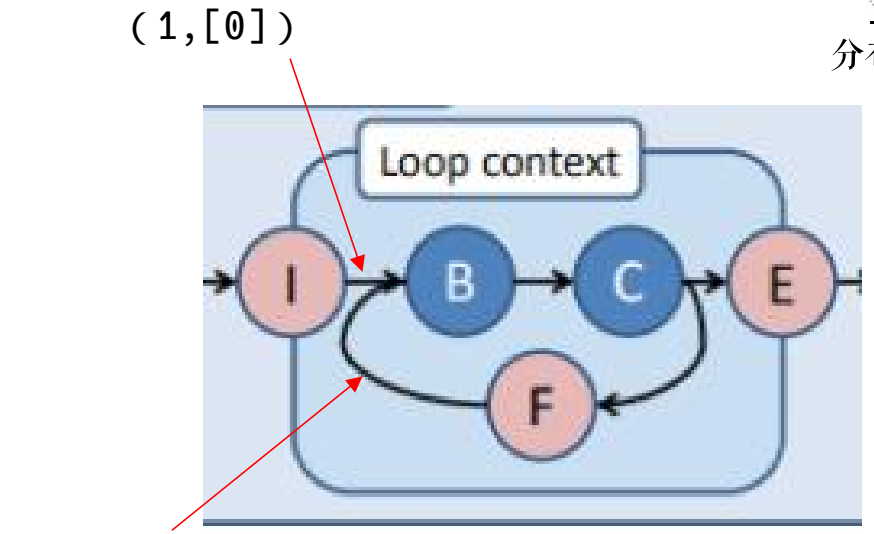
分布式存储与计算实验室

根据时间戳的偏序关系，系统可以得出，如果对两个事件（t1，l1）和（t2，l2），其中l1和l2是边或者顶点。

如果在l1和l2中存在一条路径[l1……l2]使得，t1在这条路径上修改后的时间戳<=t2,则

（t1，l1）could-result-in （t2，l2）

Timely dataflow 通过寻找l1和l2之间的最短路径，来判断两个事件之间是否存在could-result-in关系。



(1,[0])could result in(1,[2])

Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

当且仅当$e_1 \le e_2$ 且 $c_1^{\rightarrow} \le c_2^{\rightarrow}$（根据字典序比较）时，$t_1 \le t_2$

04 处理进度跟踪

系统实现了一个单线程的进度跟踪控制实现

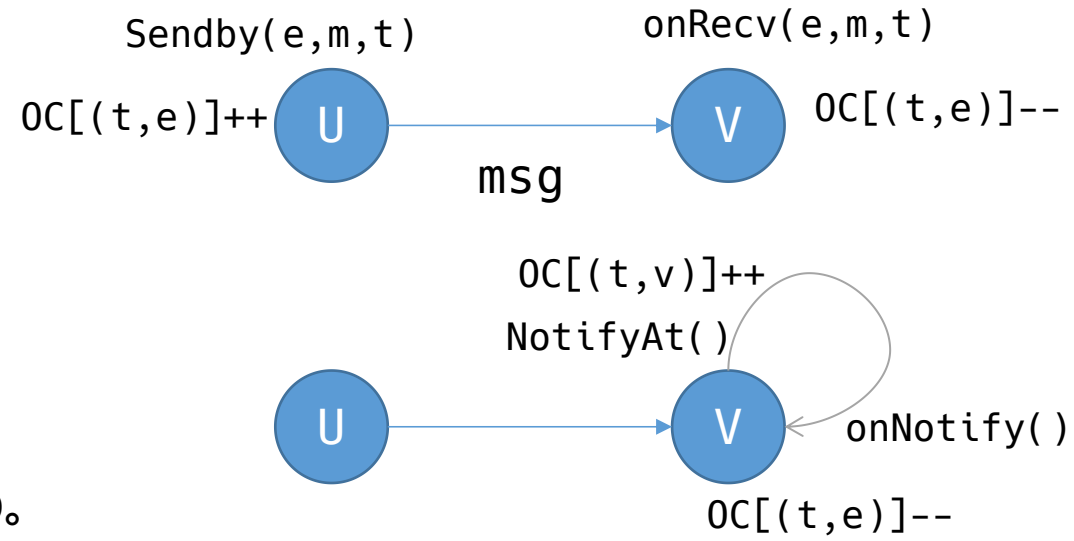
每个事件都可以表示为如下的二元组形式：

$$\text{Pointstamp} : (t \in \text{Timestamp}, \overbrace{l \in \text{Edge} \cup \text{Vertex}}^{\text{location}})$$

系统维护两个类型为[Pointstamp]int 的map。

一个叫做OC（occurrence count）活跃计数map，表示事件上活跃记录的个数。

一个叫做PC（precursor count）先驱计数map，表示could-result-in该事件的活跃记录个数。



Operation	Update
$v.\text{SENDBY}(e,m,t)$	$\text{OC}[(t,e)] \leftarrow \text{OC}[(t,e)] + 1$
$v.\text{ONRECV}(e,m,t)$	$\text{OC}[(t,e)] \leftarrow \text{OC}[(t,e)] - 1$
$v.\text{NOTIFYAT}(t)$	$\text{OC}[(t,v)] \leftarrow \text{OC}[(t,v)] + 1$
$v.\text{ONNOTIFY}(t)$	$\text{OC}[(t,v)] \leftarrow \text{OC}[(t,v)] - 1$

04 处理进度跟踪

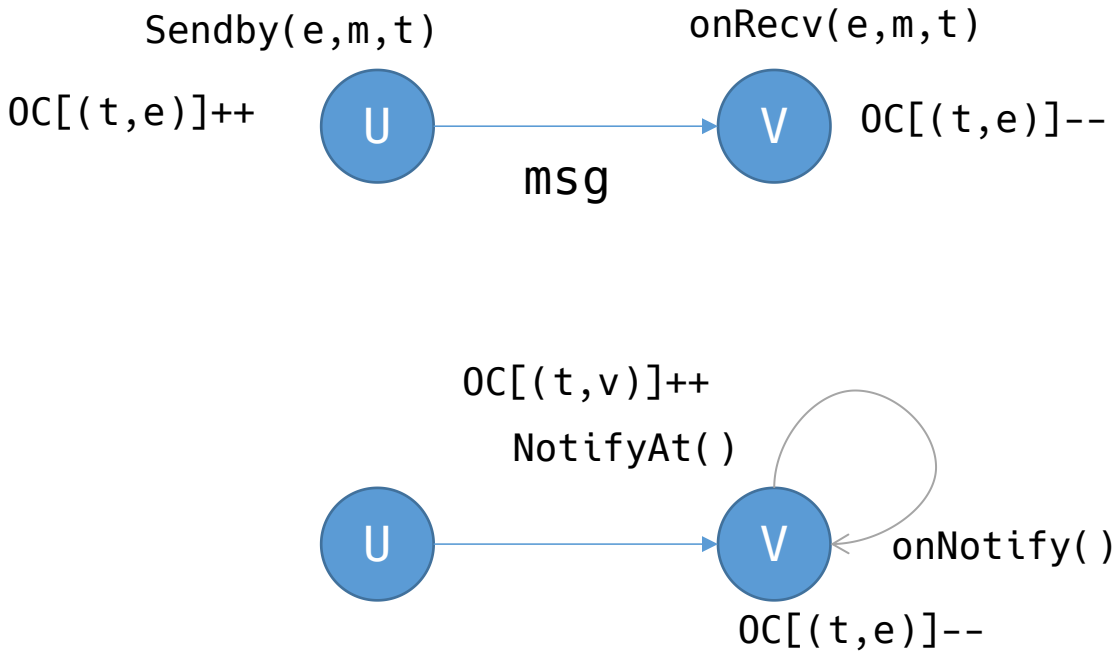
分布式存储与计算实验室

对于PC（先驱计数）而言：

当一个事件进入活跃（OC: 0->1）时，对应事件的PC被初始化为可以could-result-in该事件的活跃事件个数，并且将该事件could-result-in的事件对应的PC加一。

当一个事件不活跃（OC:==0）时，所有被该事件的could-result-in的事件对应的PC减一。

显然，当一个事件对应的PC计数等于0时，表示该事件是一个屏障，此位置不会收到could-result-in他的事件。

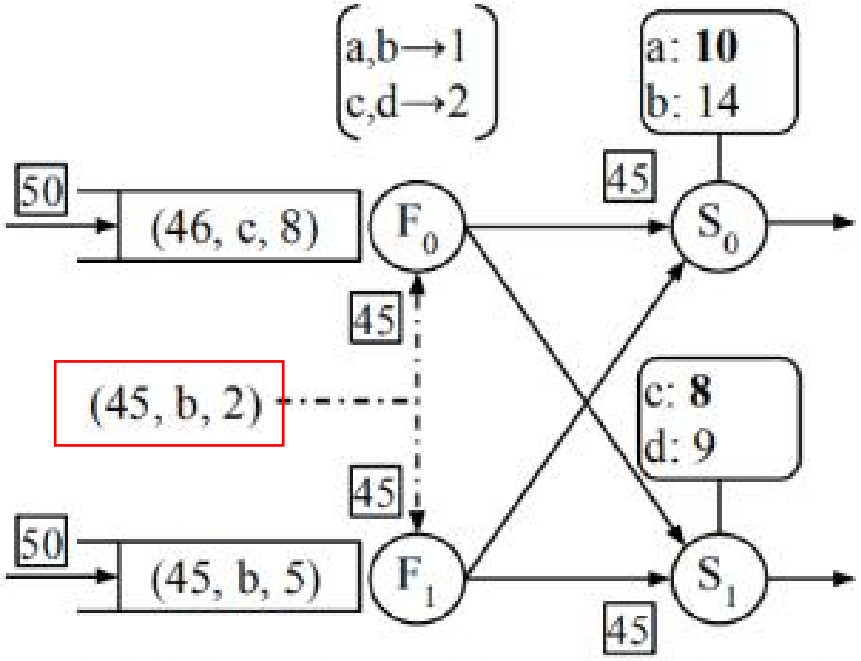


04 megaphone

Megaphone是一个建立在timely dataflow上的弹性流处理平台。

核心思想是，通过带时间戳的控制事件和进度屏障将完整的数据流分为两部分，在算子的屏障到达控制事件的时间时，进行状态迁移和管道重写。

控制事件可以表示为：
（time，key，worker）的三元组形式。



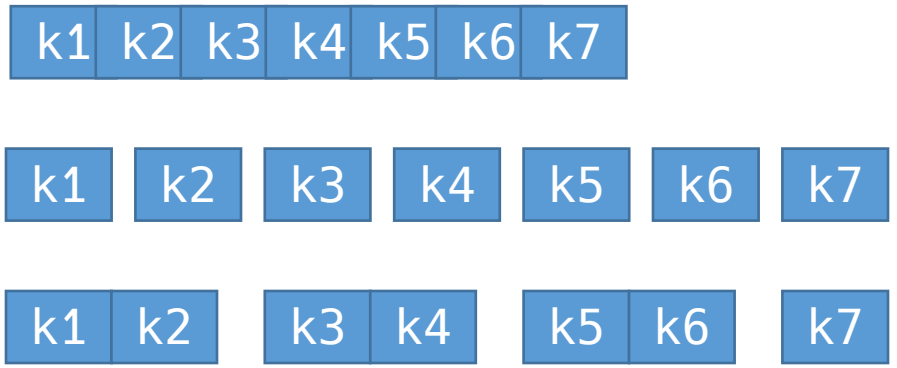
(b) Receiving a configuration update

04 megaphone

通过进度追踪屏障，用户可以自定义新配置的迁移粒度，megaphone提供了三种方式：

1. 所有状态一次迁移，此时实现类似chi的部分暂停并重启方式
2. 流式迁移，用户每次迁移一个key的状态，当迁移完成后，进行下一个key的状态迁移
3. 批量迁移，一次迁移一批状态。

三种方式中，一次迁移的吞吐量最高，流式迁移对延时的影响最低。

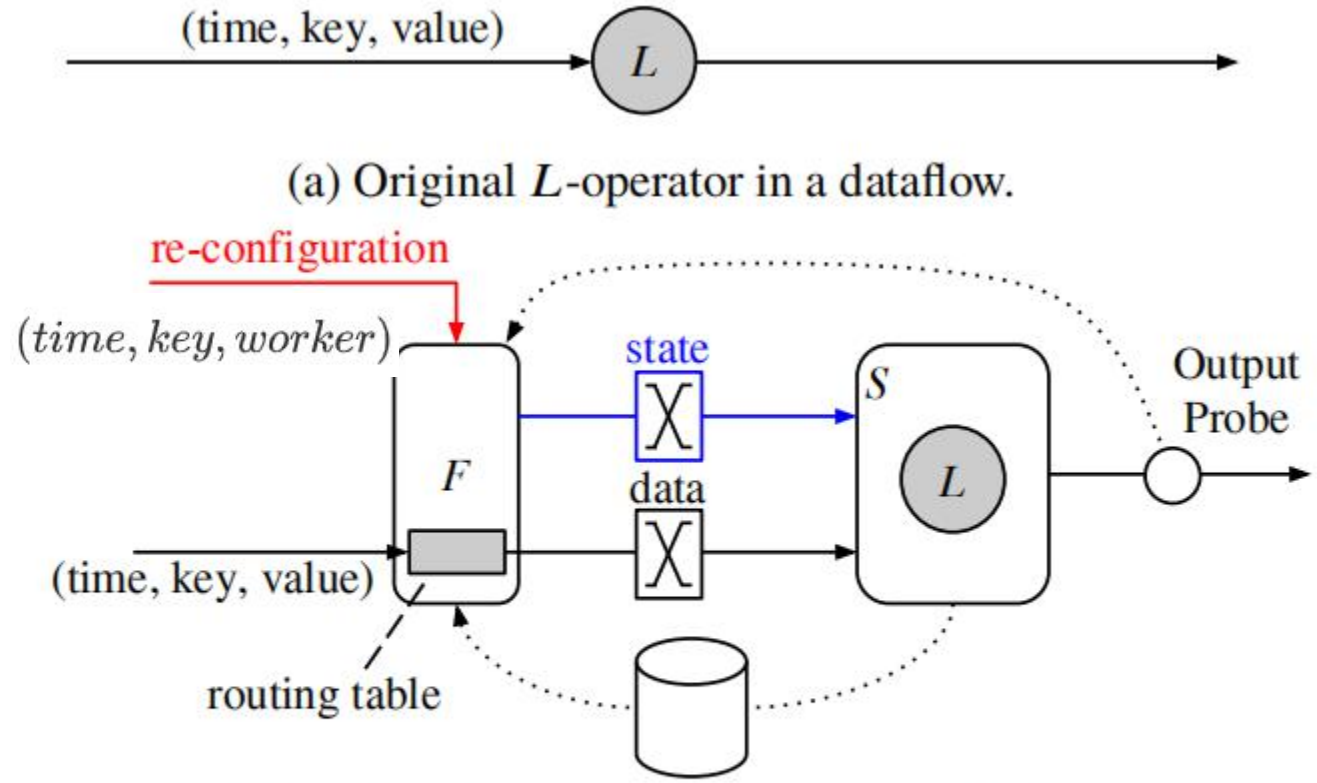


04 megaphone

系统为每个可以迁移的算子额外建立F和S两个算子来负责状态迁移。

F将数据流和配置更新流作为额外的输入，并输出data（暂时阻塞的数据流）和state

S将收到的data和state应用到L的实例上

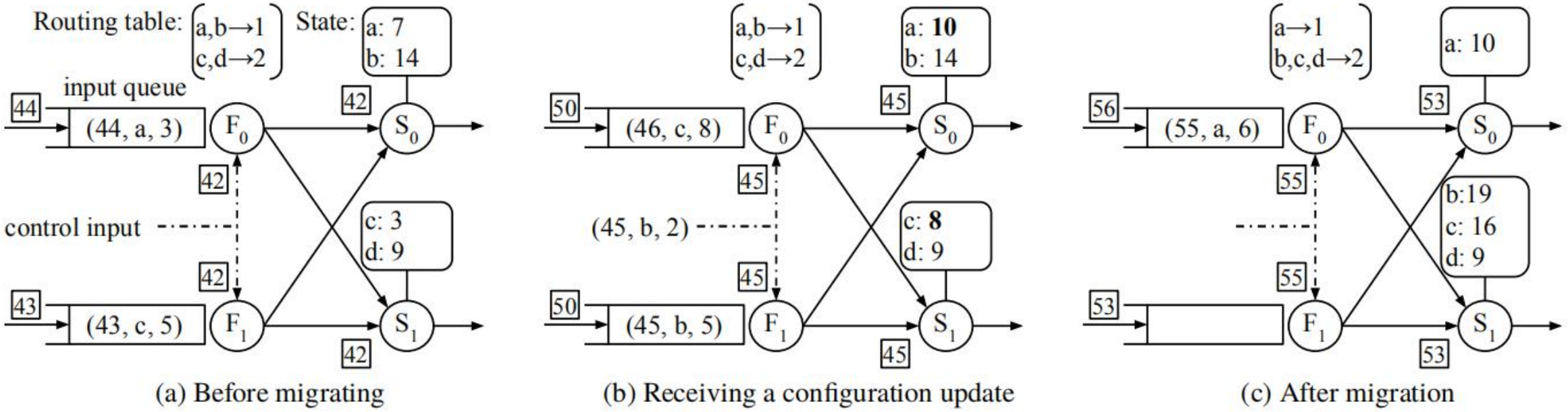


(a) Original L-operator in a dataflow.

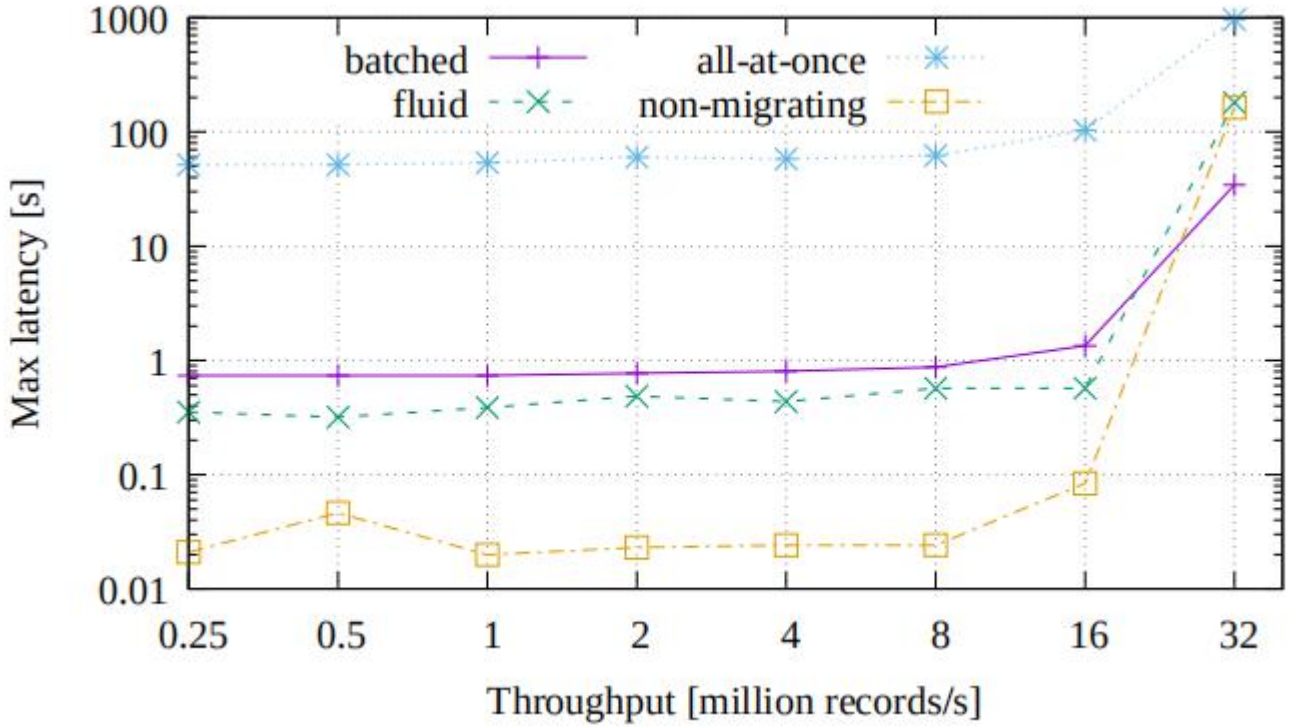
(b) Megaphone's operator structure in a dataflow.

Figure 3: Overview of Megaphone's migration mechanism

04 megaphone



上图是一个迁移示例，将key b从S0迁移到S1


1000
100
10
1
0.1
0.01
Max latency [s]
batched
fluid
all-at-once
non-migrating
0.25
0.5
1
2
4
8
16
32
Throughput [million records/s]

不同策略的表现结果

05 ／ 扩展过程中的一些优化

05 扩展过程中的一些优化

当状态非常大时，如何在重配置过程中保证数据的高效迁移

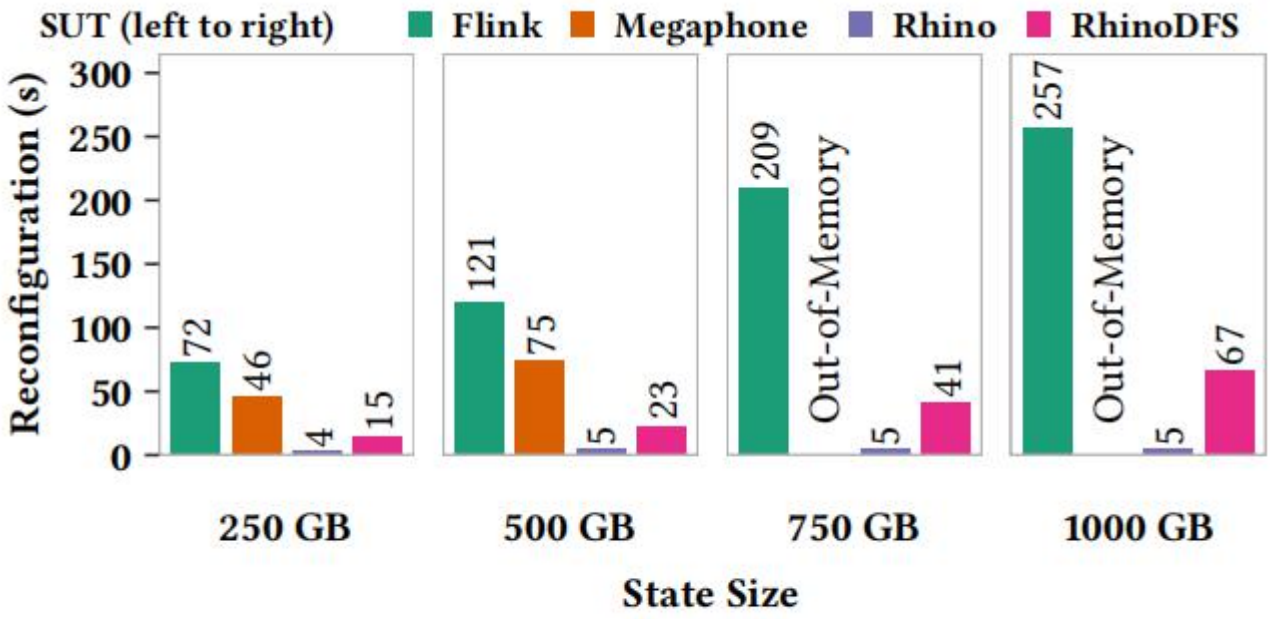


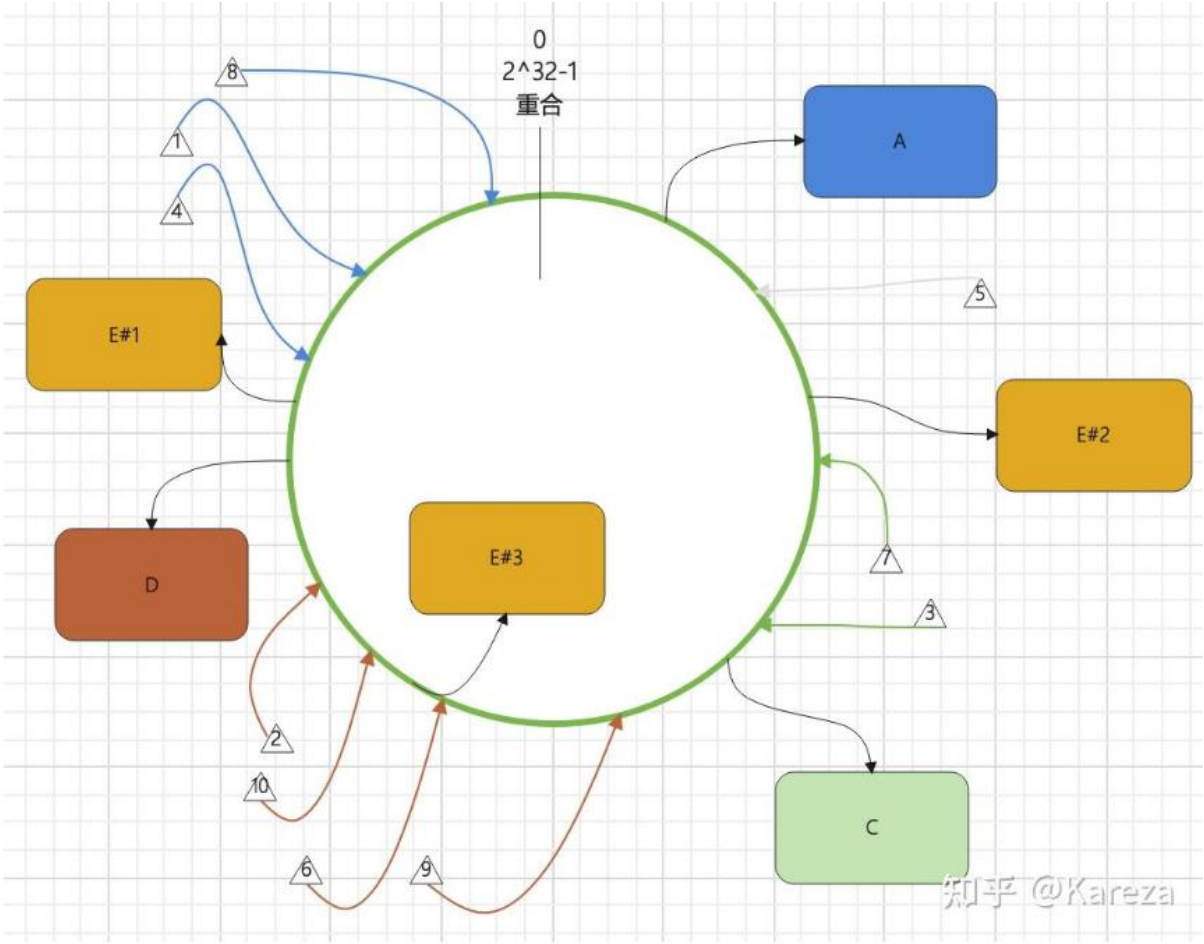
Figure 1: Time spent to reconfigure the execution of NBQ8.

05 Rhino

分布式存储与计算实验室

Rhino建立在Flink上，通过以下技术实现大规模状态的迁移

1. 主动的状态迁移，通过副本的方式，将状态副本分布到其他worker上
2. 为了减少状态的大小，异步的复制状态的增量快照，在迁移时，只迁移最后的一个快照
3. 通过一个带虚拟节点的一致性hash实现了更好的状态迁移和负载均衡



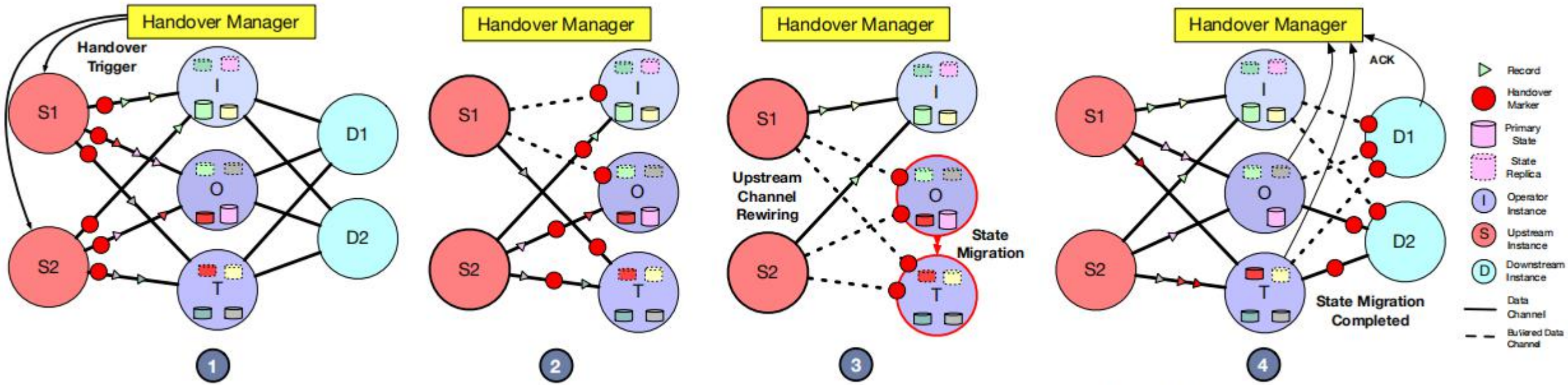


Figure 2: Steps (left to right) of the Handover Protocol of Rhino.

Rhino实现了Flink算子对控制事件的响应，通过外部的监视平台，将控制事件传入dataflow后，算子响应控制，完成配置修改。

05 Rhino

Flink以DFS为外部快照保存地点时，快照以block为单位保存。

Rhino通过以state为单位保存快照，使系统在rescale时可以尽可能的减少网络开销。

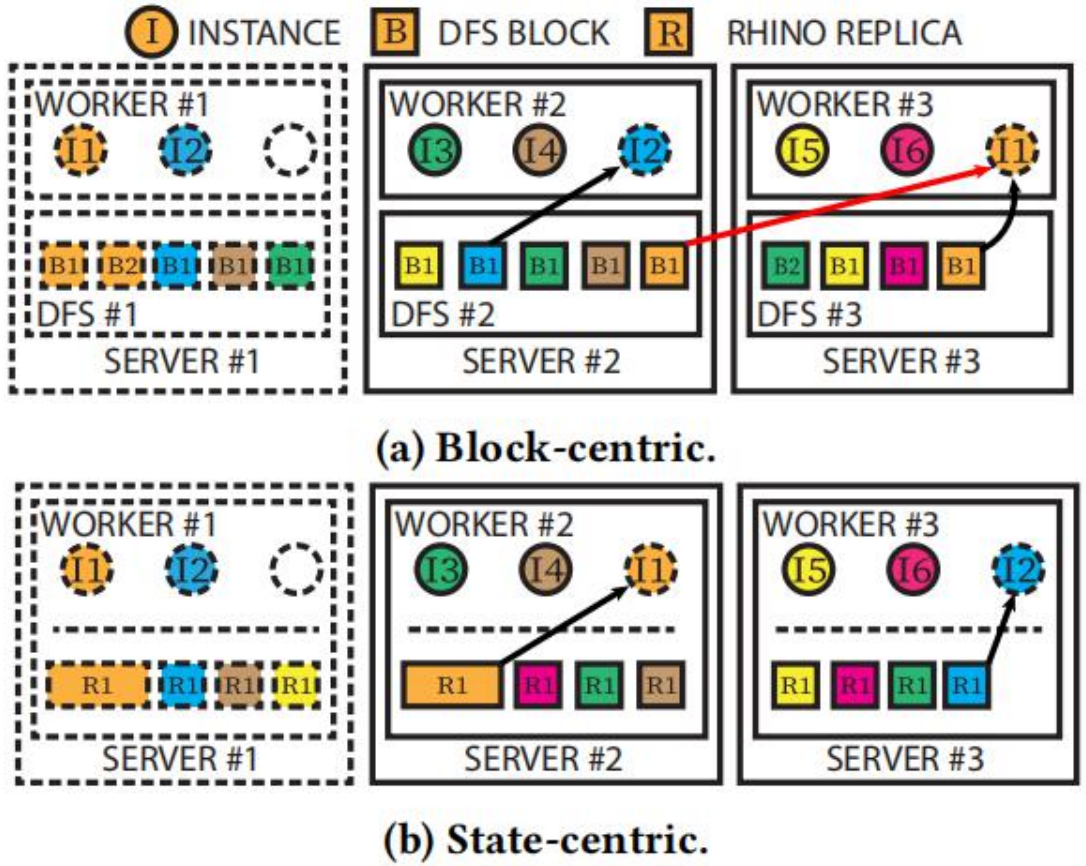
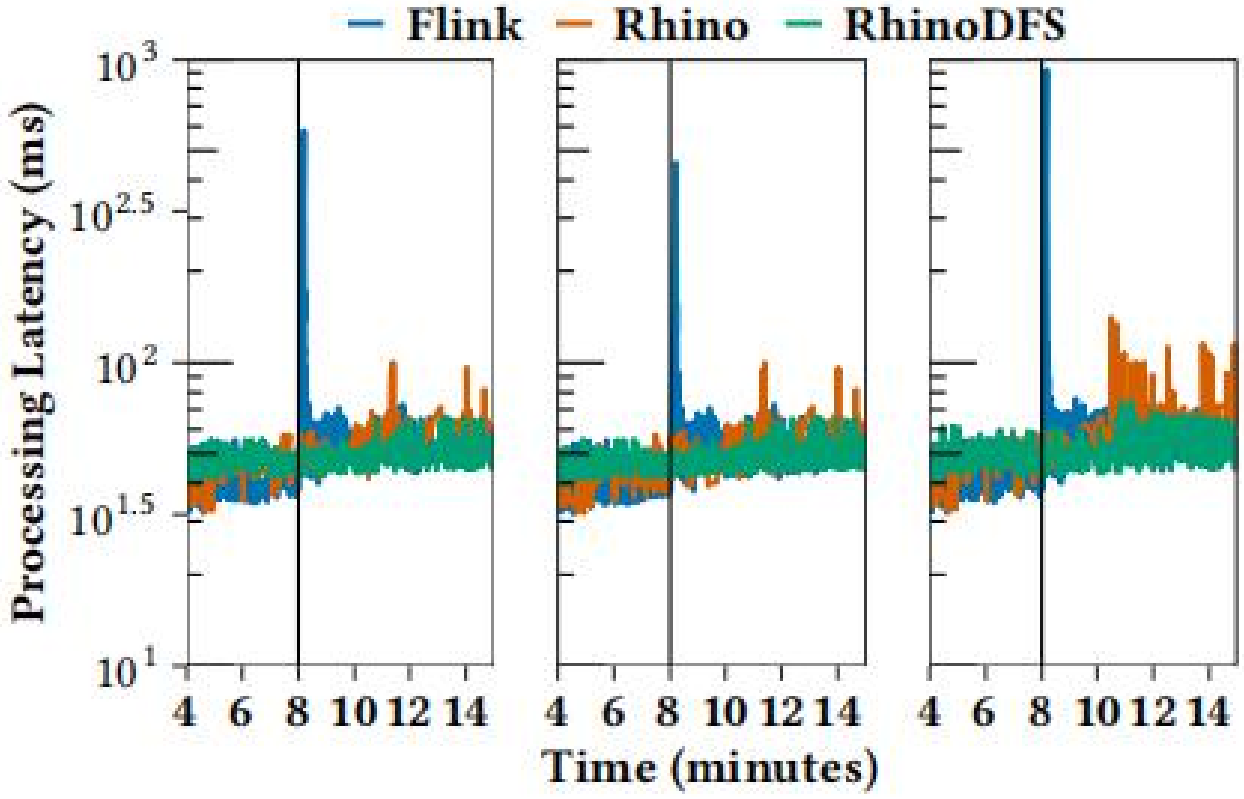


Figure 3: Block-centric vs. state-centric replication. Black and red arrows indicate local and remote fetching, respectively.

05 Rhino

右图展示了在Rhino在大规模状态的迁移的相比Flink更加稳定。



06 总结

06 总结

	思路	优点	缺点
Flink	1. 暂停数据流处理 2. 协调者重新生成物理执行计划后从最近的快照处重启	1. 实现简单	1. 吞吐量和延迟表现都比较差
chi	1. 通过控制事件找到同步点 2. 部分暂停数据流处理	1. 用户可以自定义控制事件 2. 同一算子的不同实例间直接进行状态迁移	1. 配置修改一次完成，开销较大
megaphone	1. 通过进度追踪机制进行状态迁移 2. 细粒度的配置变更	1. 细粒度配置更新，对系统的性能的影响较低	1. 依赖流系统提供的进度追踪机制 2. 通过F和S算子实现变更，维护开销较大
Rhino	1. 通过控制事件找到同步点 2. 主动迁移状态 3. 通过一致性hash和虚拟节点提高rescale效率	1. 在大规模状态迁移时，系统的性能波动低 2. 以state为中心的副本复制减少了状态迁移开销。	1. 依赖调度算法保证，新实例需要的状态可以在本地上读取

06 文献

分布式存储与计算实验室

[01] - Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: A Scalable and Pro_x0002_grammable Control Plane for Distributed Stream Processing Systems. VLDB (2018).

[02] - Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In ACM SOSP.

[03] - Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, and Timothy Roscoe. 2019. Megaphone: Latency-conscious State Mi_x0002_gration for Distributed Streaming Dataflows. VLDB (2019).

[04] - Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20).