Distributed Big Data Stream Processing

Wenxin Li

2018-12-22

Outline

A primer on distributed stream processing

- Introduction to streams
- Stream processing systems

A brief survey on stream processing optimizations

- 7 optimization techniques will be introduced

A paper on OSDI'18

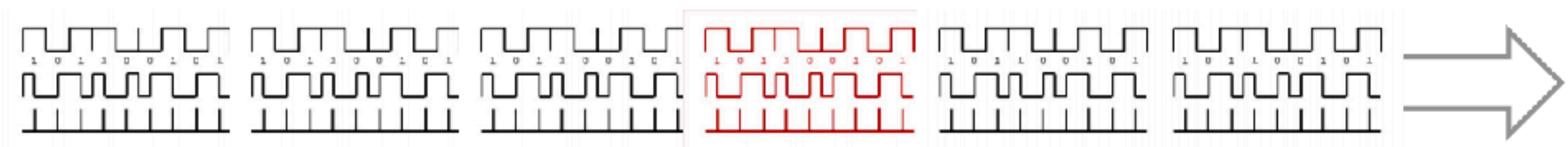
- Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflow

A primer on distributed stream processing

Streams

- Bank transactions

- Sensor data



- Cat video in tweets

Streaming applications

Application scenarios

- Ranging from anomaly traffic detection, social network analysis, stream database queries, online machine learning, …

Streaming examples

Query examples in social network analysis	
Tweet Statistics	[rate, count] of [tweet, hashtag] on [location, language, topic]
Users Analysis	[rate, count] of [tweet, hashtag, retweet] on [gender, age-group] per [location, language]
Top-k Analysis	Top-k [popular, trending] [hashtag, topic, retweet] per [language, location]
Sentiment Analysis	[aggregate, categorize] sentiment of each [hashtag, country, topic]
System Load	[rate, count] of [bandwidth usage, request] per [node, region]

Big & Fast

Social Network

- Facebook: > 845M active users, > 8B messages/day
- Twitter: > 140M active users, >340M tweets/day

Autonomous Driving

- 1GB data per minute per car

Traffic Monitoring

- High event rates: millions events / sec
- High query rates: thousands queries / sec

Distributed Processing

Apache Storm
- Used by Weather Channel, WebMD, Alibaba, Baidu etc.

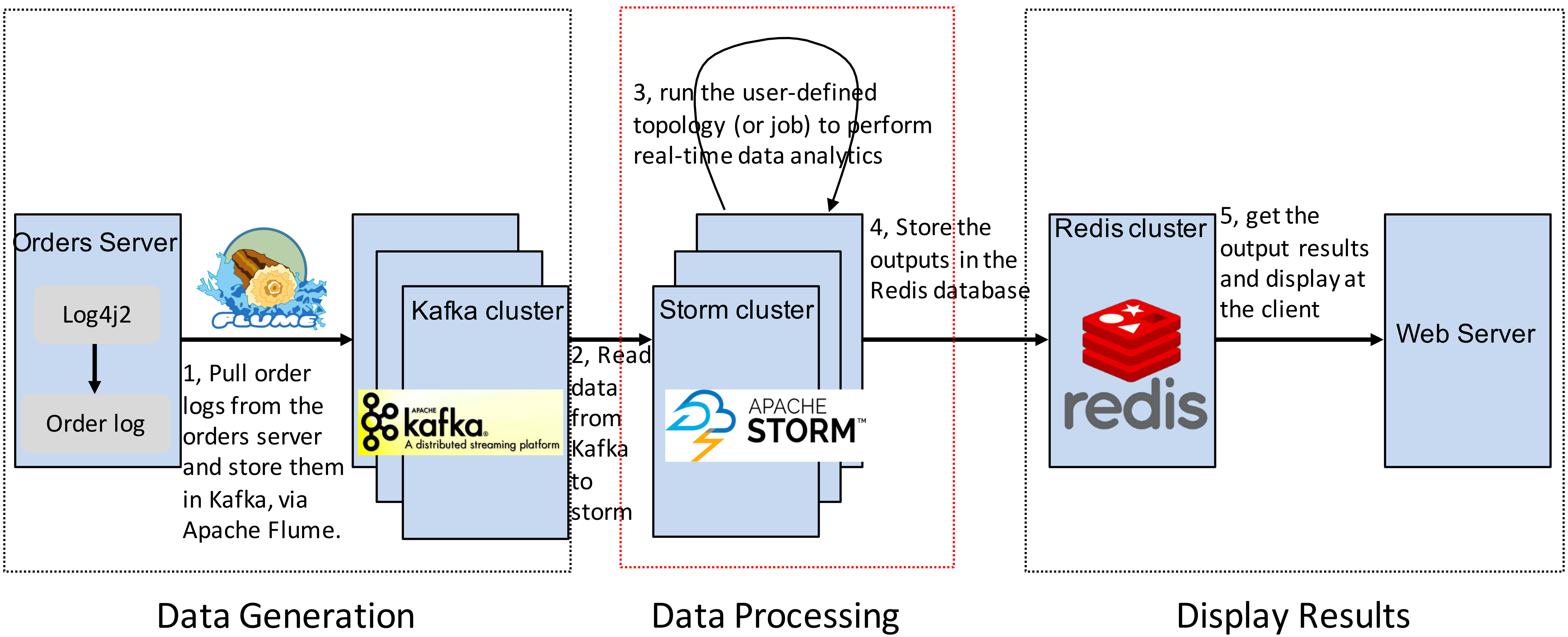
Heron
- Used by Twitter

Puma, Stylus, Swift
- Used by Facebook

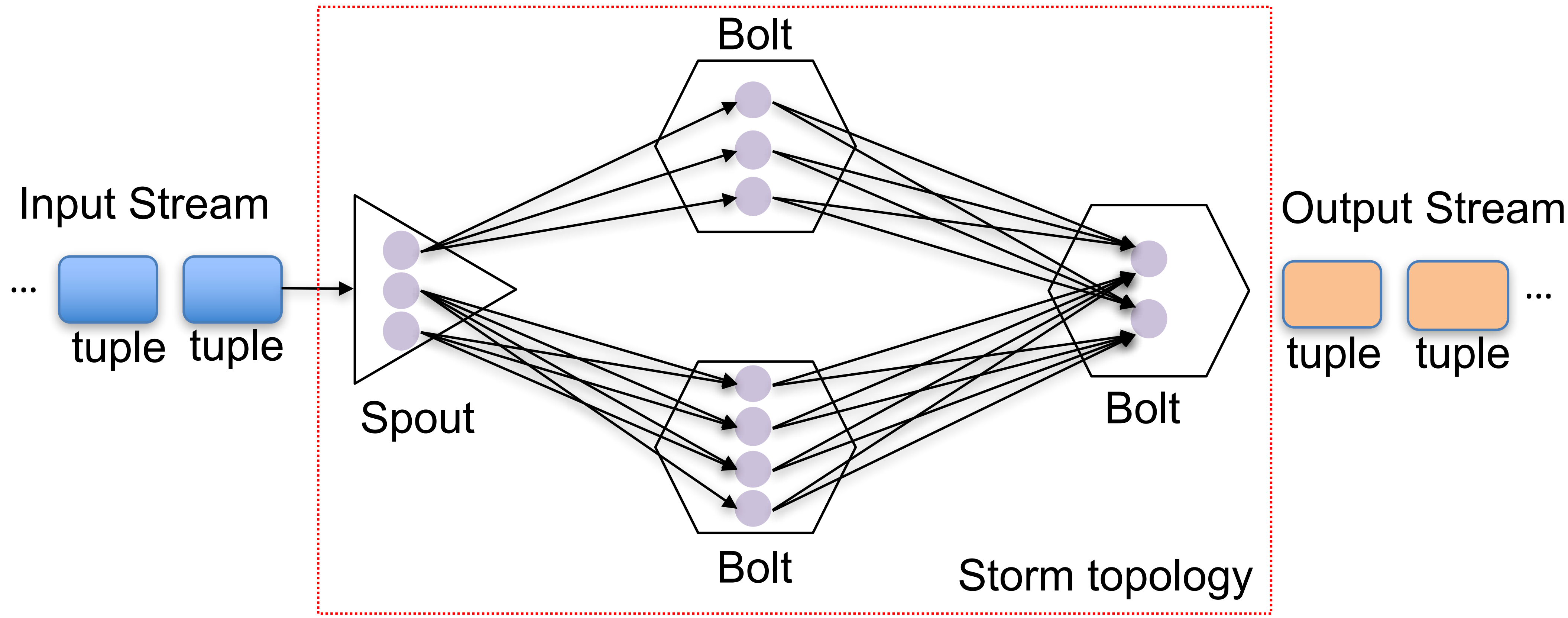
Samza
- Used by LinkIn

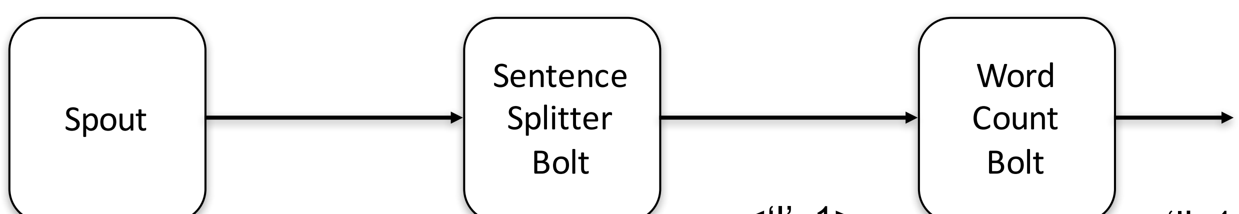
Flink
- Used by meituan

A basic architecture



Storm topology





T=1	“I want to eat”	<‘I’, 1> <‘want’, 1> <‘to’, 1> <’eat’, 1>	<‘I’, 1> <‘want’, 1> <‘to’, 1> <’eat’, 1>
T=2	“I like it”	<‘I’, 1> <‘like’, 1> <‘it’, 1>	<‘I’, 2> <‘want’, 1> <‘to’, 1> <’eat’, 1> <‘like’, 1> <‘it’, 1>

2 Requirements

Low tuple latency

- End-to-end tuple latency: the time between a tuple entering the topology from the source, to producing an output result on the sink.

High throughput:

- the number of tuples that a system can process per time unit.

A brief survey on stream processing optimizations

Overview

7 optimization techniques

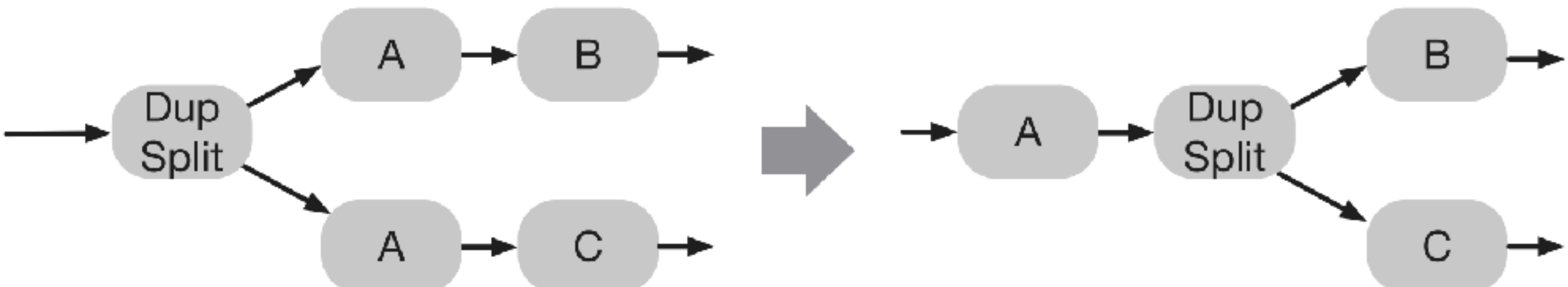
- Operator reordering
- Redundancy elimination
- Scaling
- Placement
- Load balancing
- Batching
- Load shedding

1 Operator Reordering

Move more selective operators upstream to filter data early.

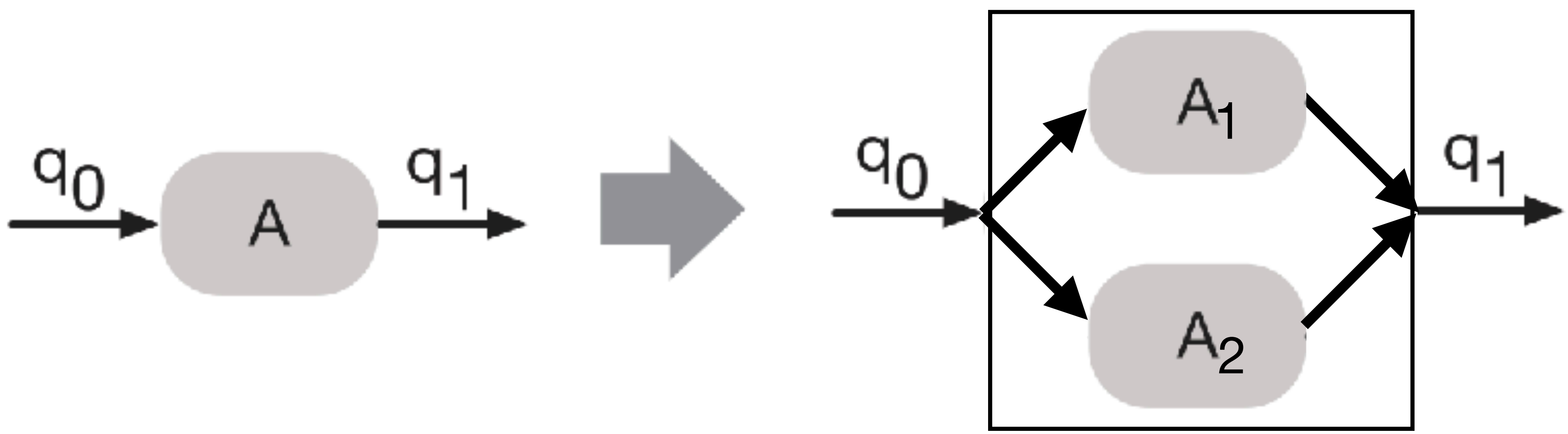
2 Redundancy Elimination

Eliminate redundant computations



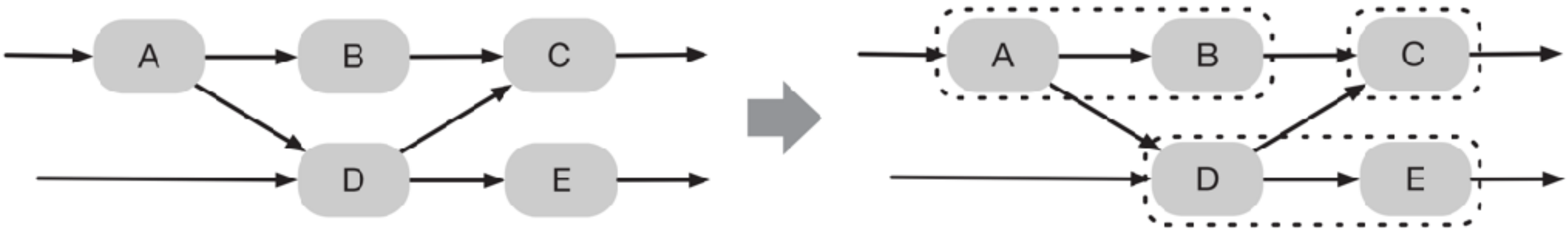
3 Scaling

Scaling the computation resource allocated to an operator



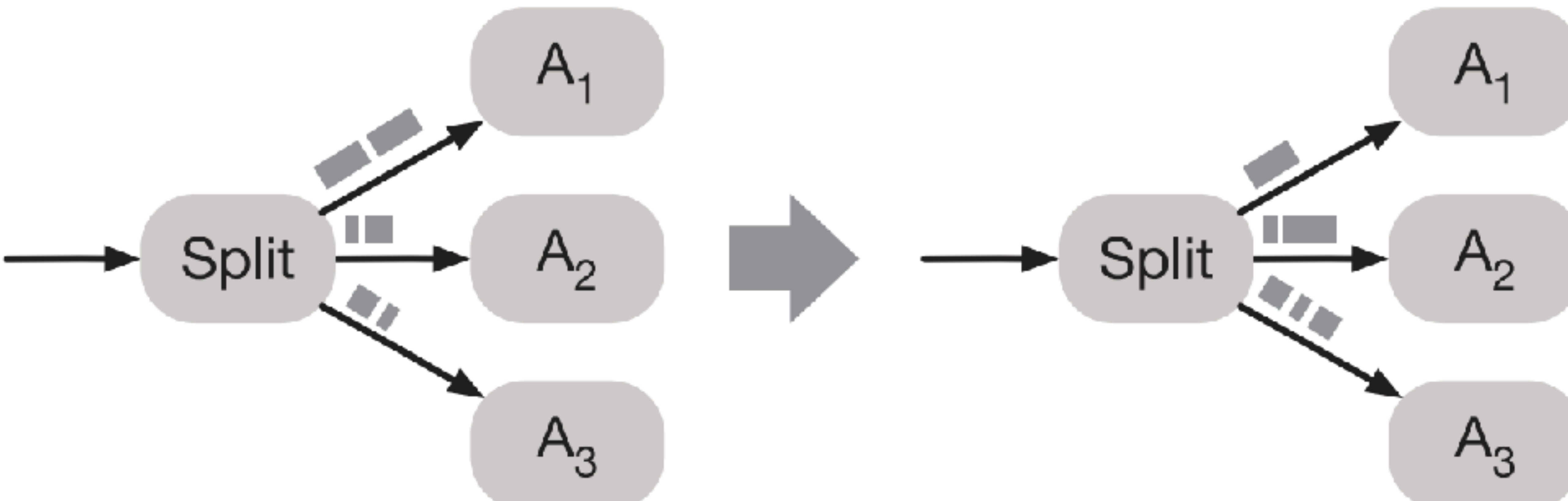
4 Placement

Assign operators to hosts and cores



5 Load balancing

Distribute workload evenly



6 Batching

Process multiple data items in a single batch

7 Load shedding

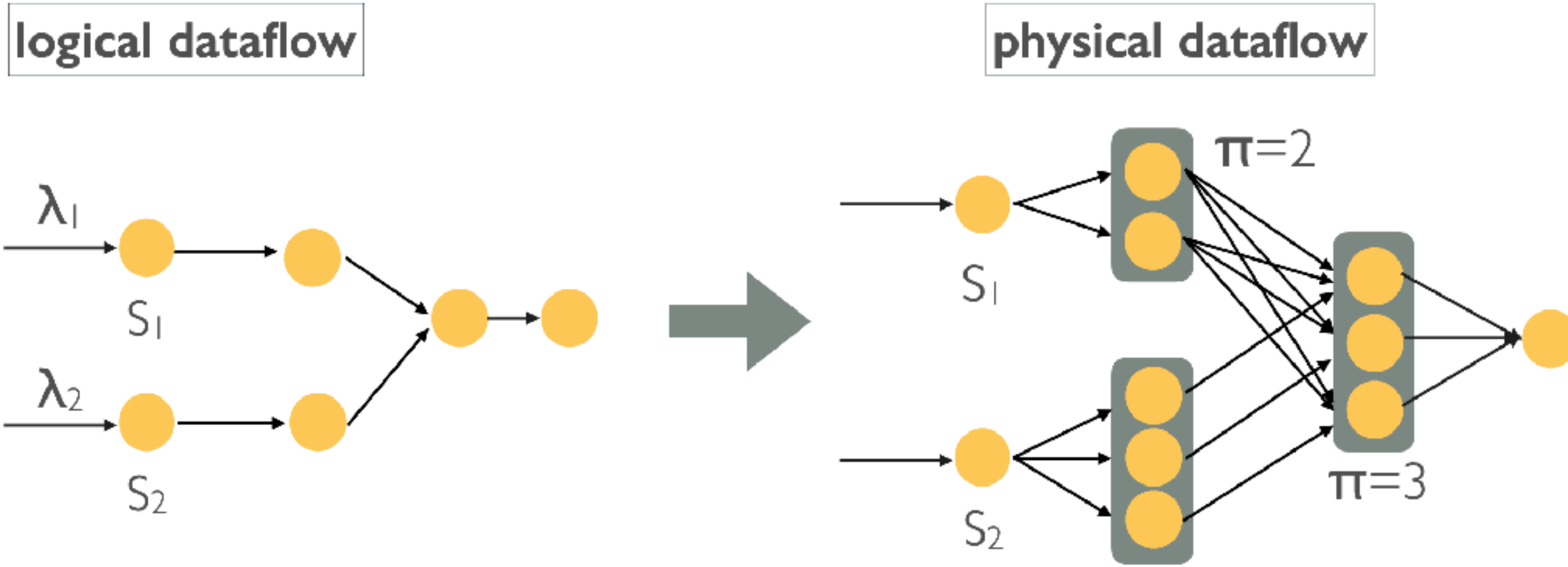
Degrade gracefully when overloaded

Three steps is all you need:

fast, accurate, automatic scaling decisions for distributed streaming dataflow

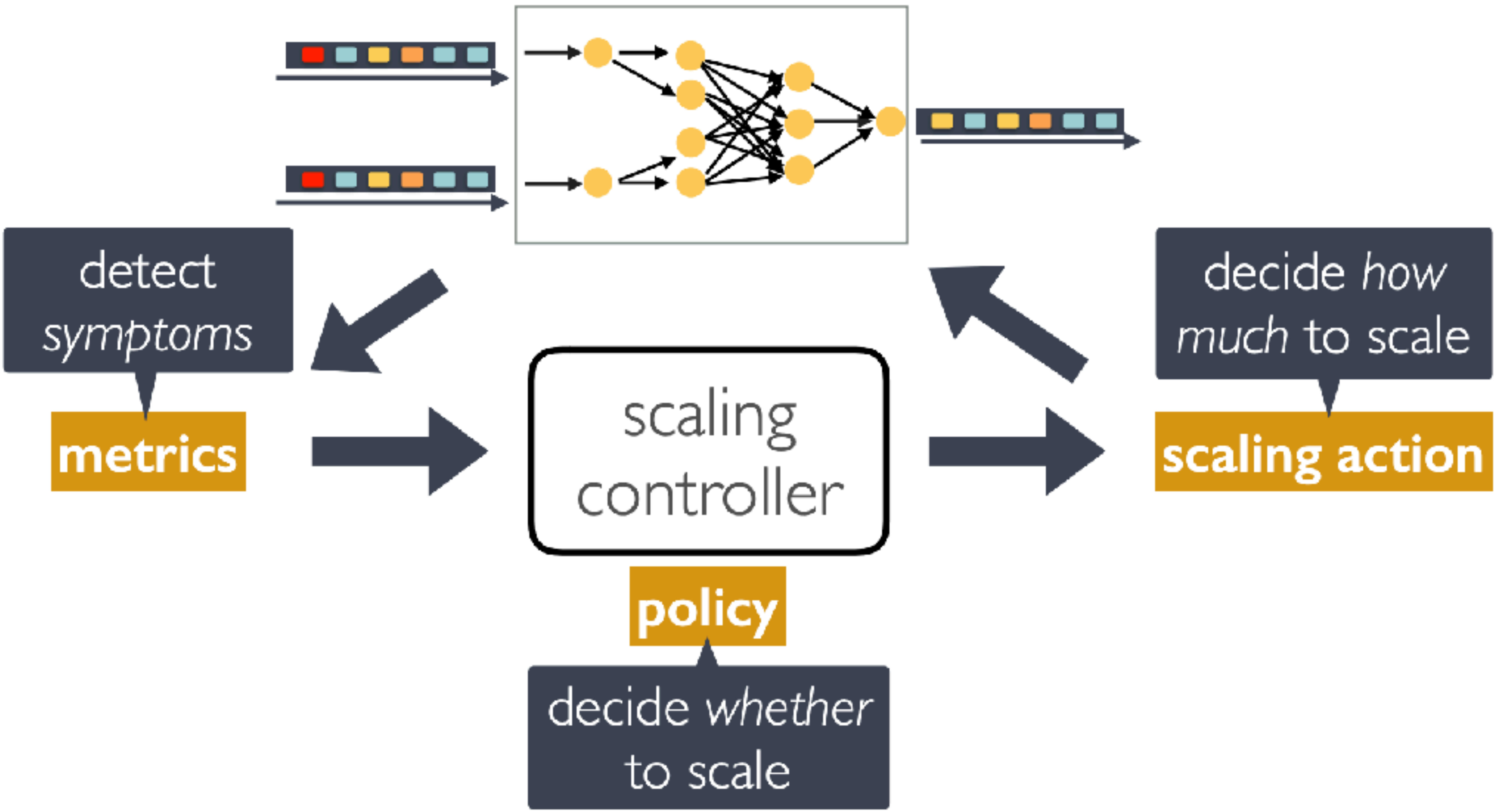
Vasiliki Kalavri, John Liagouris, Moritz Hoffmann et al.
OSDI 18

The scaling problem



Given a logical dataflow with sources $S_1, S_2, \ldots S_n$ and rates $\lambda_1, \lambda_2, \ldots \lambda_n$ identify the minimum parallelism π_i per operator i, such that the physical dataflow can sustain all source rates.

Automatic scaling



Existing work

- Borealis
- StreamCloud
- Seep
- IBM Streams
- Spark Streaming
- Google Dataflow
- Dhalion
- ...

metrics

CPU utilization
backlog, tuples/s
backpressure signal

problematic due to interference, multitenancy

✗ **oscillations**

policy

threshold and rule-based
if CPU > 80% => scale

sensitive to noise, manual, hard to tune

✗ **temporary over- and under-provisioning**

scaling action

small changes,
one operator at a time

non-predictive, speculative steps

✗ **slow convergence**

The proposed DS2 overview

metrics

~~externally observed~~

policy

~~threshold-based~~

scaling action

~~non-predictive, single-operator~~

true rates through instrumentation

dataflow dependency model

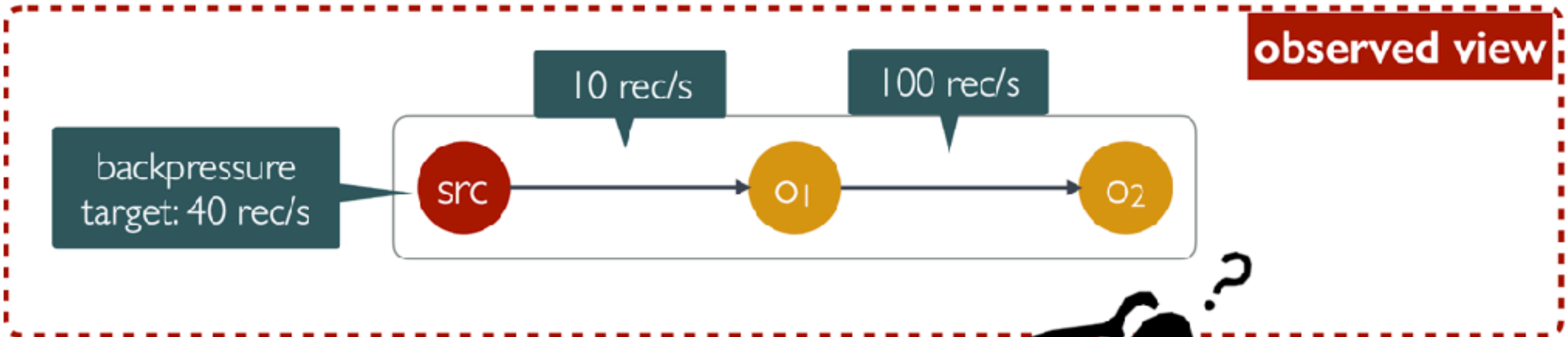
predictive, dataflow-wide actions

no oscillations

true rates as bounds to avoid over/under-shoot

fast convergence

An illustrative example

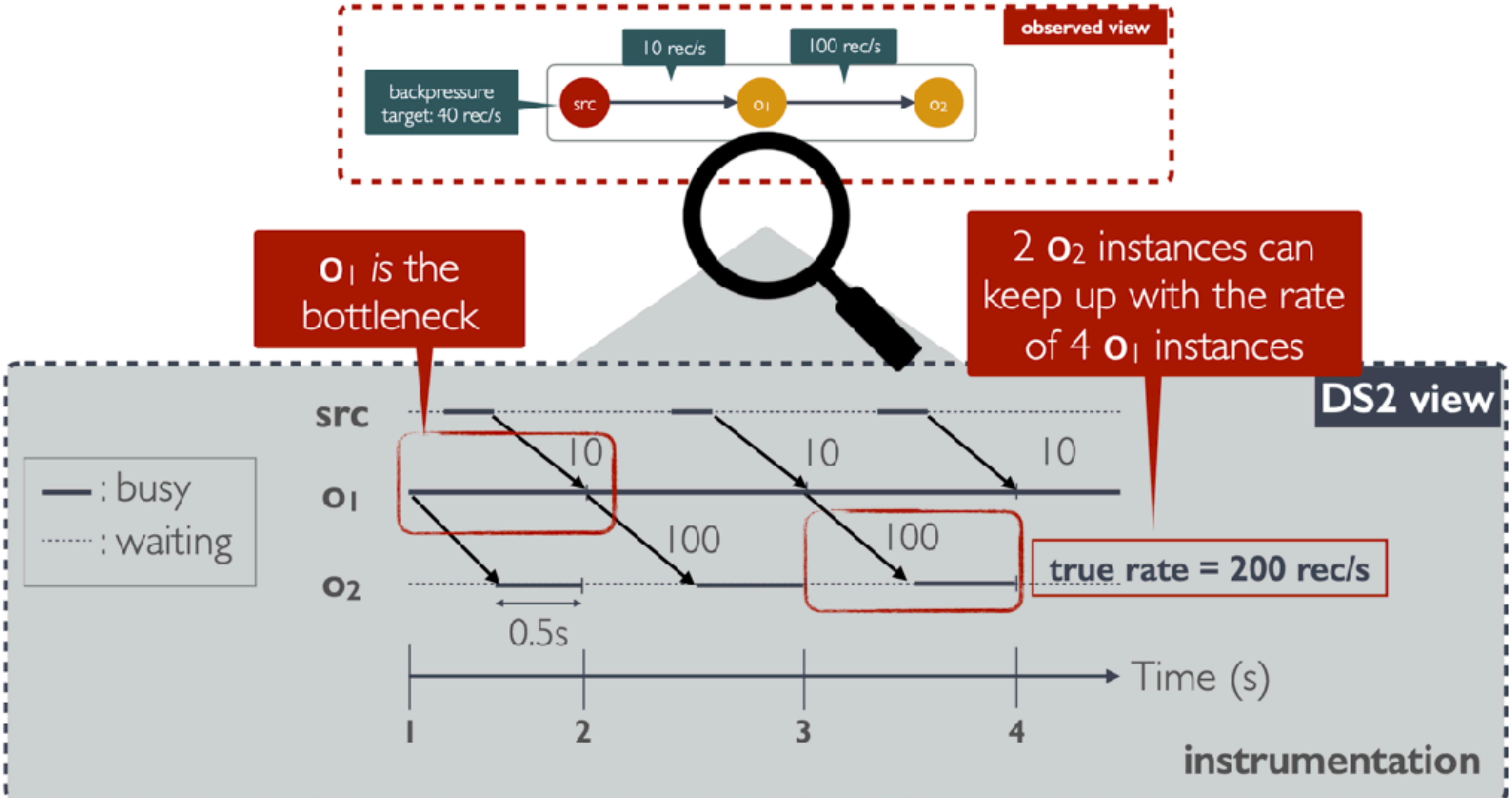


Which operator is the bottleneck?

What if we scale o_1 x 4?

How much to scale o_2?

An example


observed view
10 rec/s
100 rec/s
backpressure target: 40 rec/s
src
o1
o2
o1 is the bottleneck
2 o2 instances can keep up with the rate of 4 o1 instances
DS2 view
src
o1
o2
— : busy
······ : waiting
10
10
10
100
100
true rate = 200 rec/s
0.5s
Time (s)
1
2
3
4
instrumentation

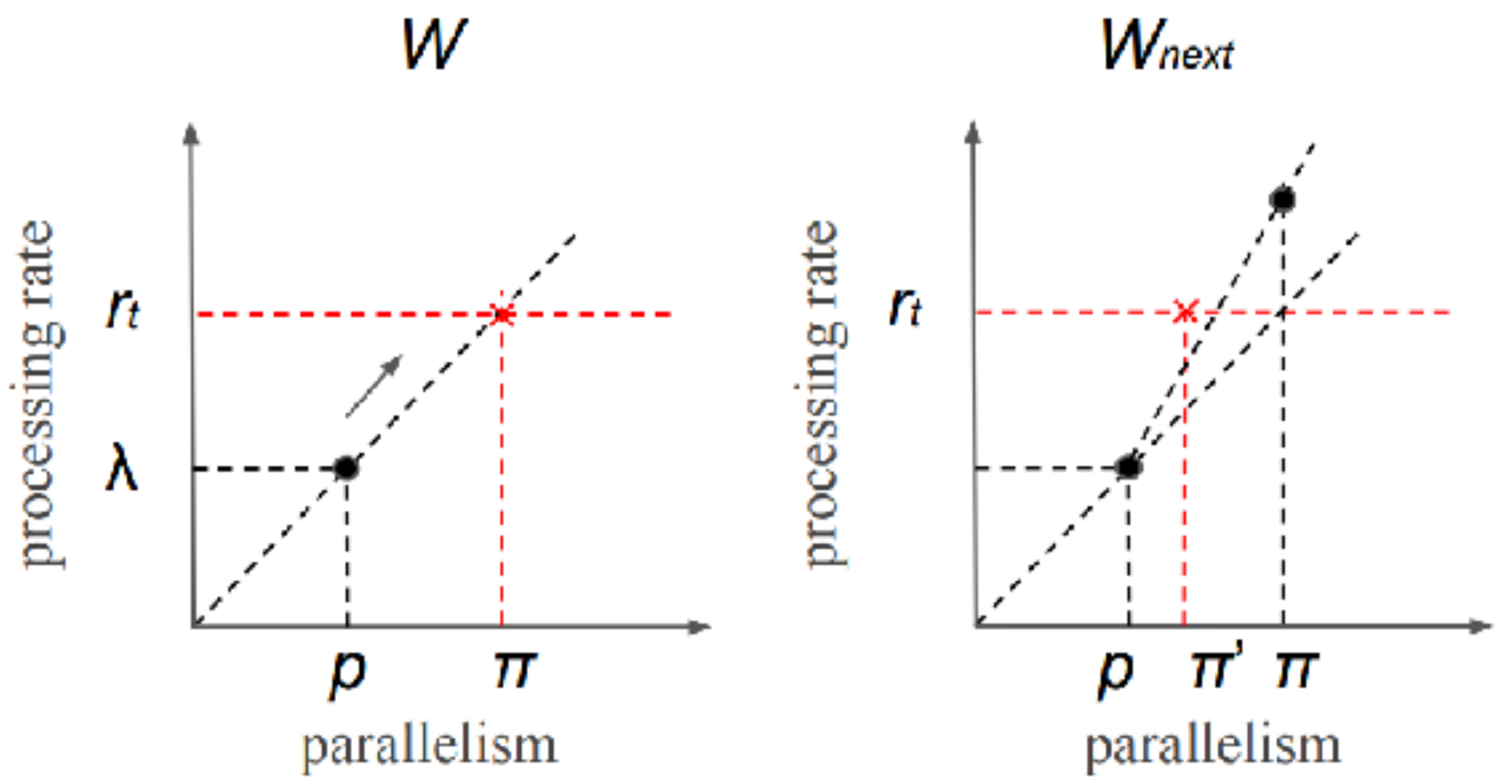
The DS2 model

<u>Useful time</u>: The time spent by an operator instance in **deserialization**, **processing**, and **serialization** activities.

<u>True processing (resp. output) rate</u>: The number of **records** an operator instance can process (resp. output) **per unit of useful time**.

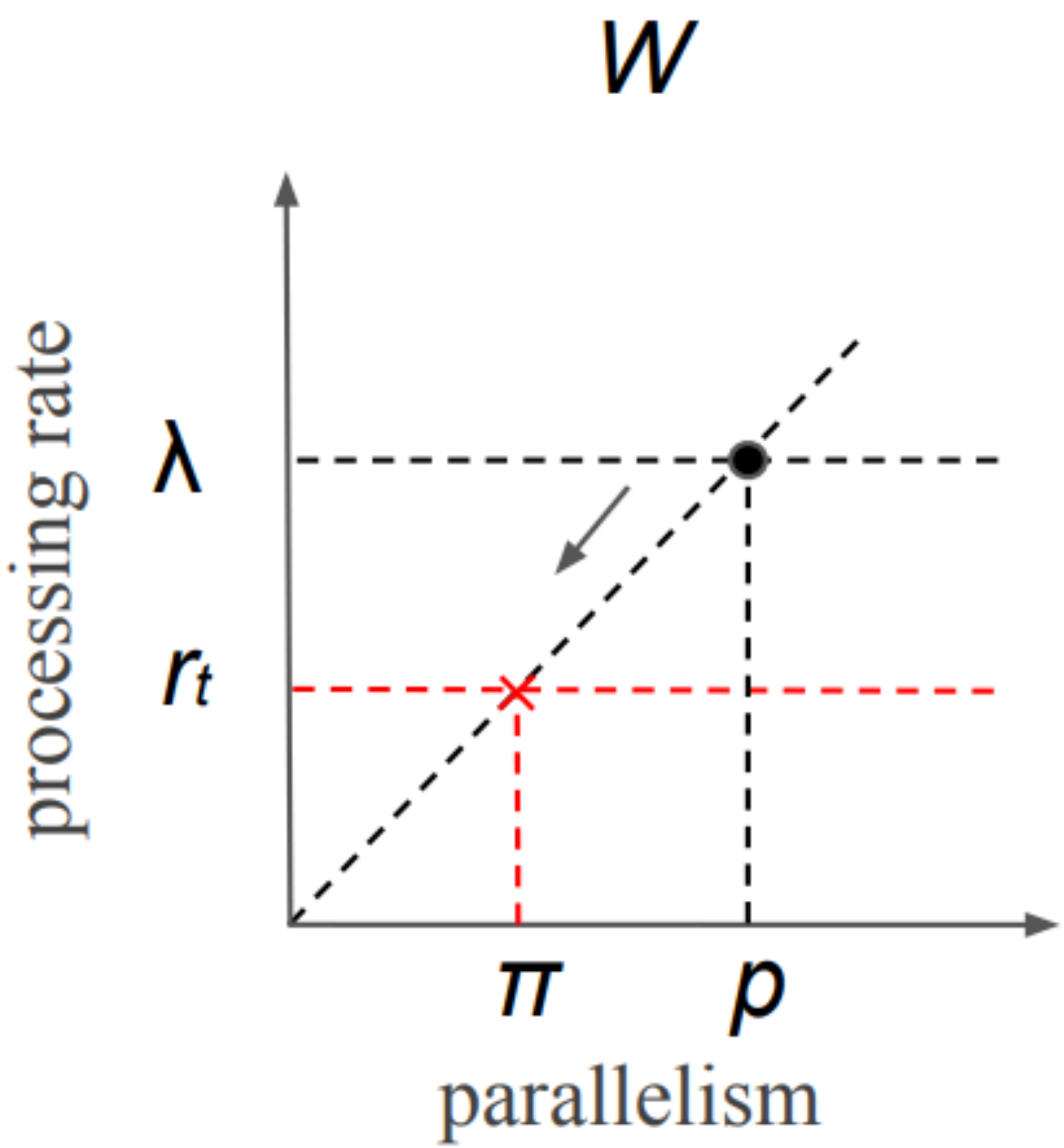
$$\textbf{Optimal parallelism for } o_i : \frac{\text{aggregated true output rate of upstream ops}}{\text{average true processing rate of } o_i}$$

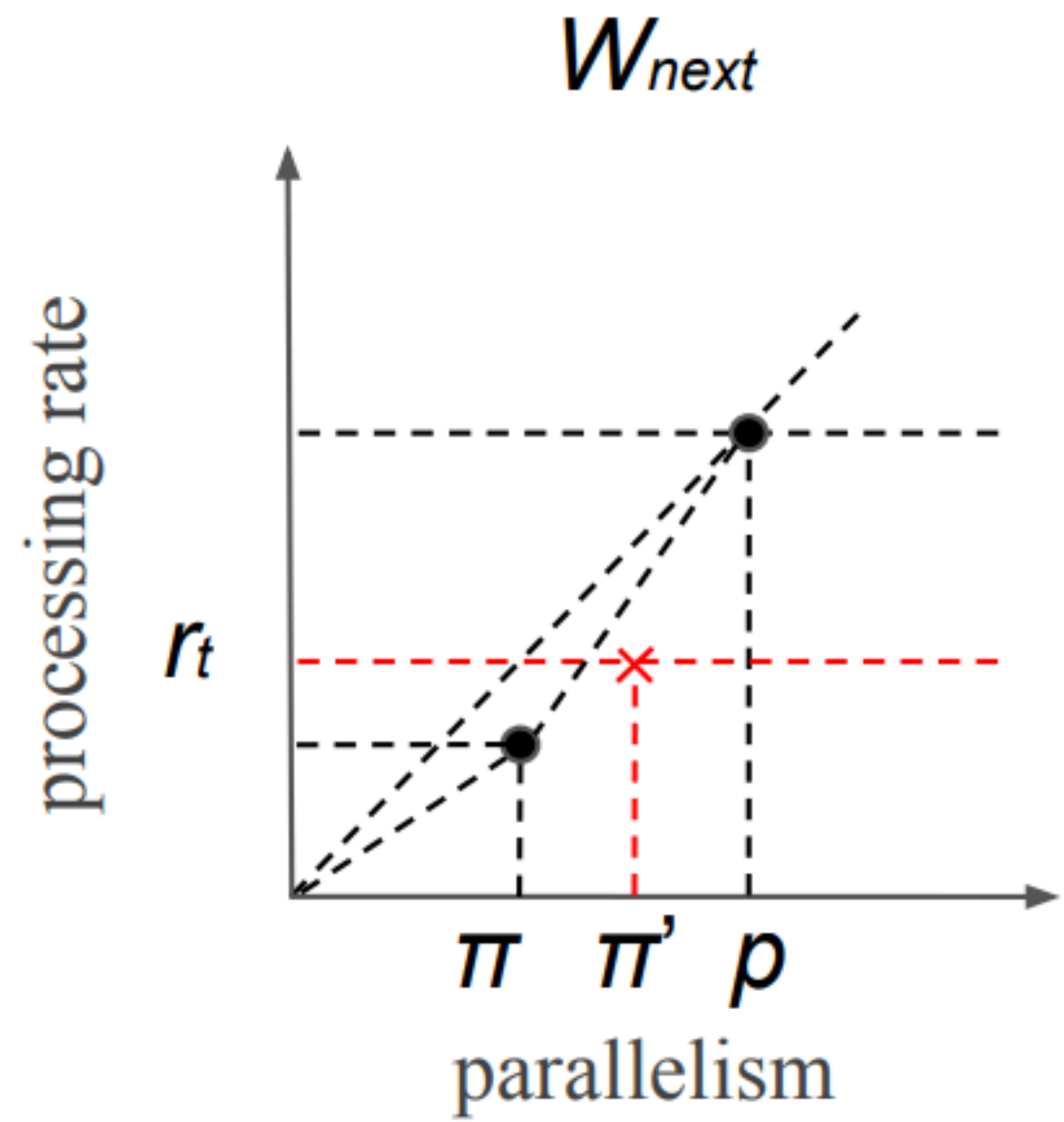
Scale up/down behavior



(a) No overshoot when scaling up

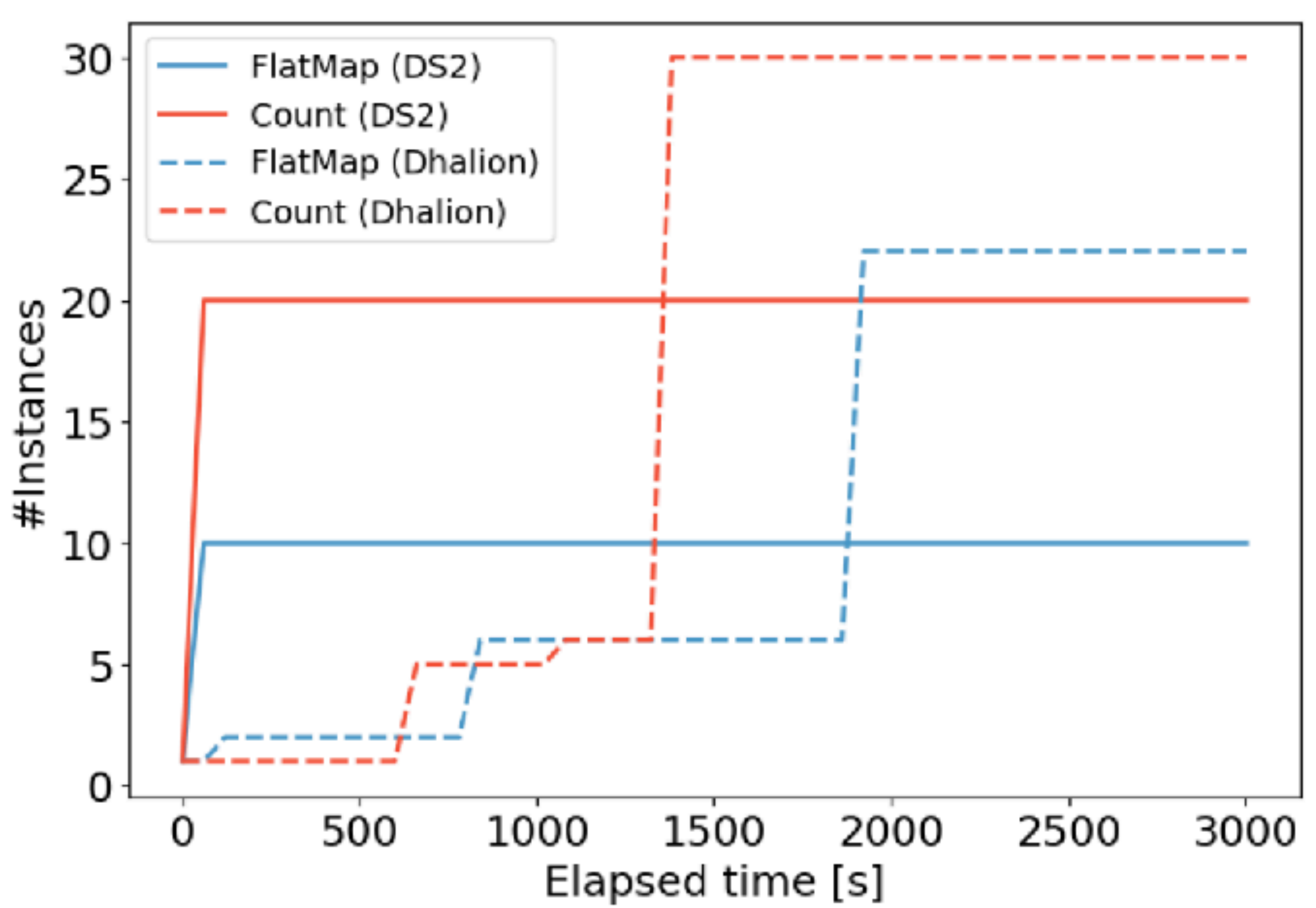
Scale up/down behavior





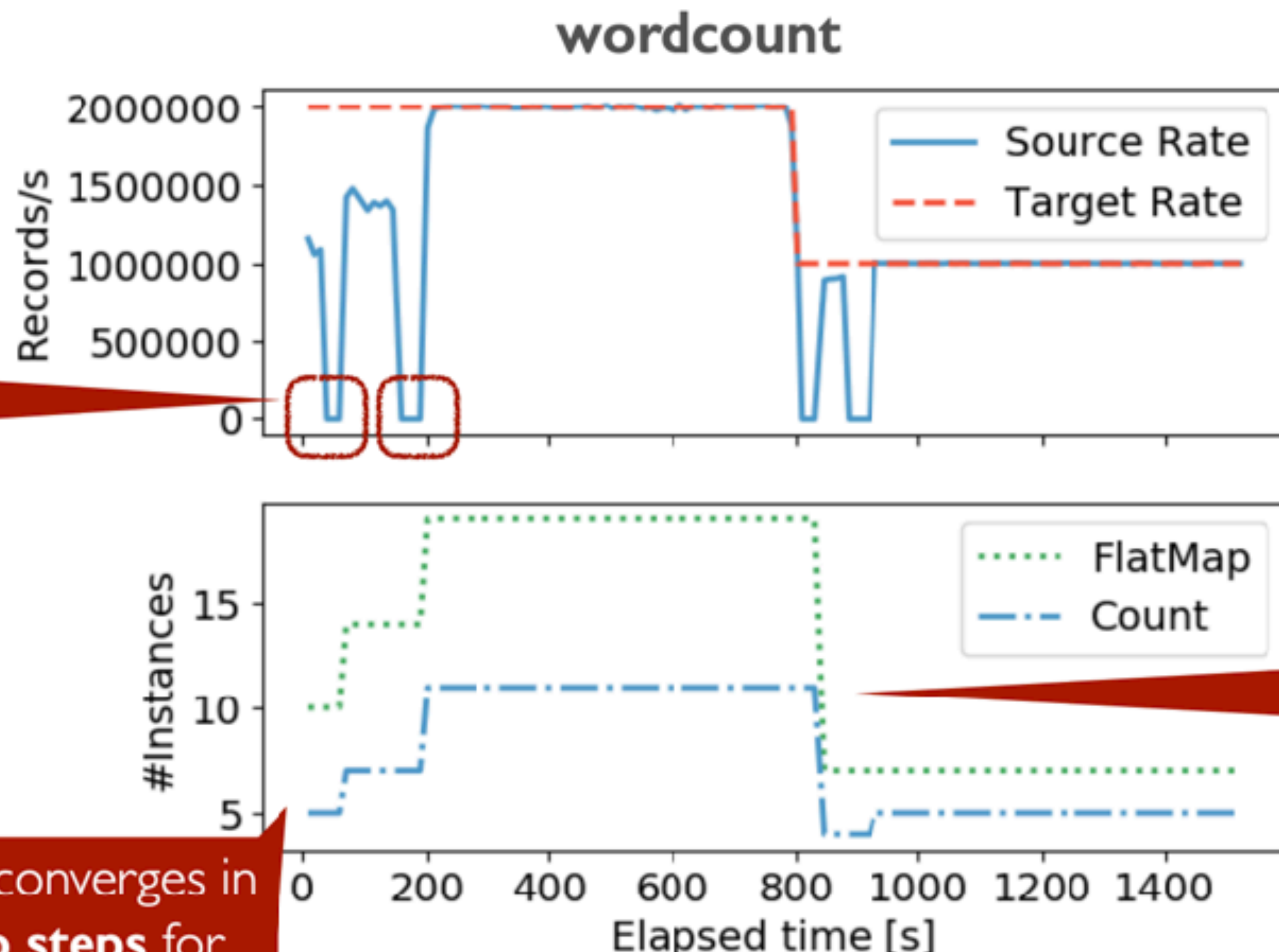
Evaluation

DS2 vs. Dhalion on Heron using a word count dataflow



Evaluation

DS2 on Apache Flink



Apache Flink savepoint and reconfiguration takes ~**30s**

DS2 reacts **3s** after the target rate has changed

DS2 converges in **two steps** for both operators

Target rate: 2.000.000 rec/s

Evaluation

Convergence-NEXMARK

at most **3 steps**

initial parallelism	Q1: flatmap	Q2: filter	Q3: incremental join	Q5: tumbling window join	Q8: sliding window	Q11: session window
8 =>	12 => **16**	11 => 13 => **14**	16 => **20**	14 => 15 => **16**	**10**	12 => 22 => **28**
12 =>	**16**	**14**	18 => **20**	**16**	**10**	22 => **28**
16 =>	**16**	12 => **14**	**20**	**16**	8 => **10**	26 => **28**
20 =>	**16**	13 => **14**	**20**	14 => **16**	8 => **10**	**28**
24 =>	**16**	**14**	**20**	14 => **16**	8 => **10**	**28**
28 =>	**16**	**14**	**20**	13 => **16**	8 => **10**	**28**

scale-up

scale-down

a single step for many queries and initial configurations

=> : scaling action

Thanks!