CS 591 K1: Data Stream Processing and Analytics

Spring 2020

4/09: Flow control and load shedding

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Keeping up with the producers

- Producers can generate events in a higher rate than the rate consumers can process events.

Keeping up with the producers

- Producers can generate events in a higher rate than the rate consumers can process events.
- What happens if consumers cannot keep up with the event rate?

Keeping up with the producers

- Producers can generate events in a higher rate than the rate consumers can process events.
- What happens if consumers cannot keep up with the event rate?
    - drop messages

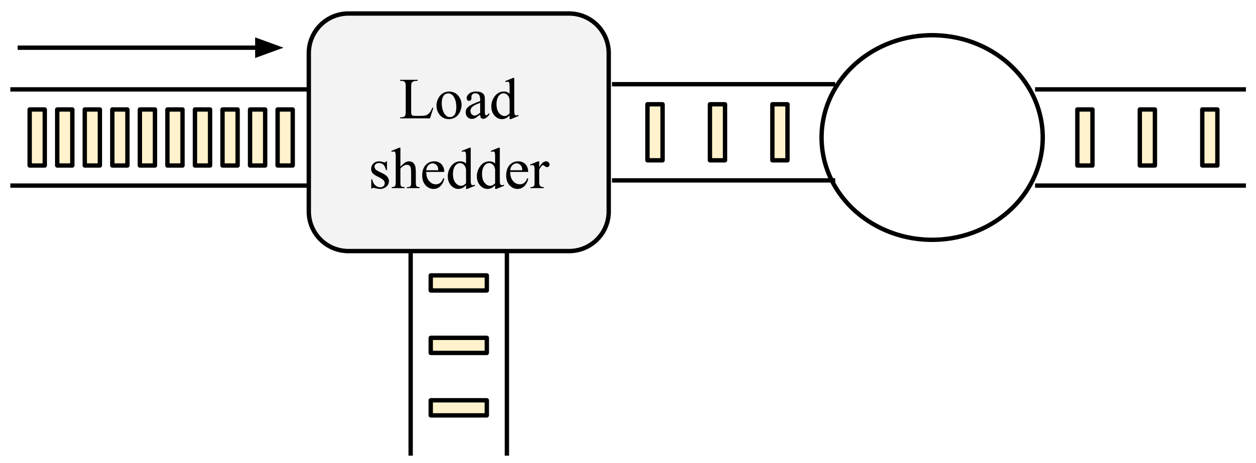
Keeping up with the producers

- Producers can generate events in a higher rate than the rate consumers can process events.
- What happens if consumers cannot keep up with the event rate?
  - drop messages
  - buffer messages in a queue: what if the queue grows larger than available memory?

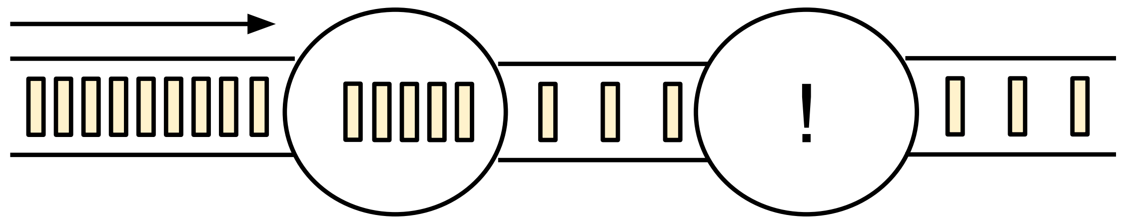
Keeping up with the producers

- Producers can generate events in a higher rate than the rate consumers can process events.
- What happens if consumers cannot keep up with the event rate?
  - drop messages
  - buffer messages in a queue: what if the queue grows larger than available memory?
  - block the producer (back-pressure, flow control)

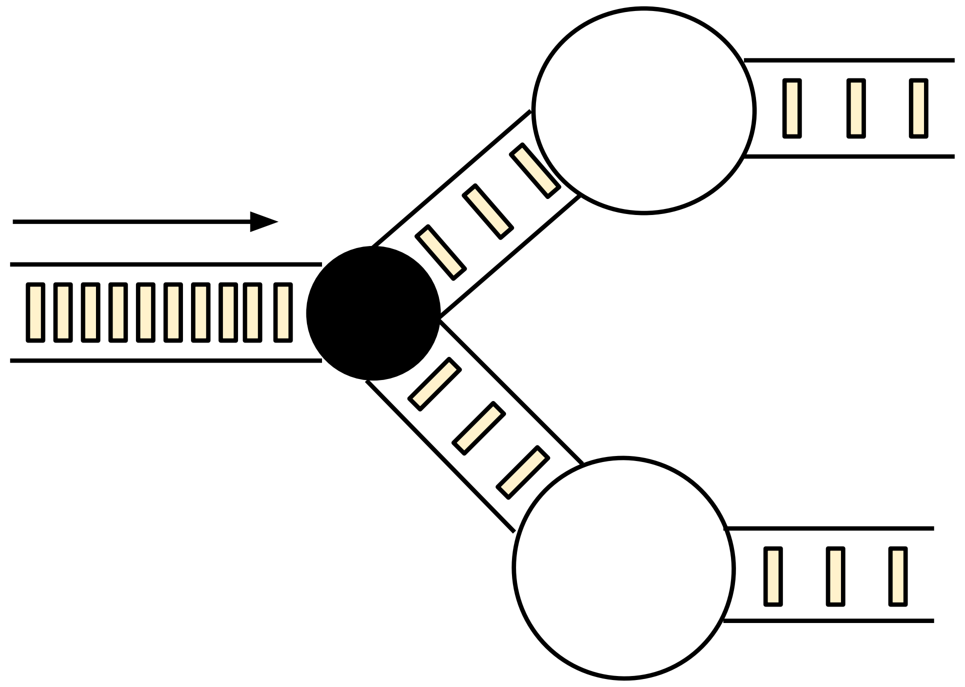
Load management approaches



(a) Load shedding



(b) Back-pressure



(c) Elasticity

Selectively drop records:

- Temporarily trades-off result accuracy for sustainable performance.
- Suitable for applications with strict latency constraints that can tolerate approximate results.

Slow down the flow of data:

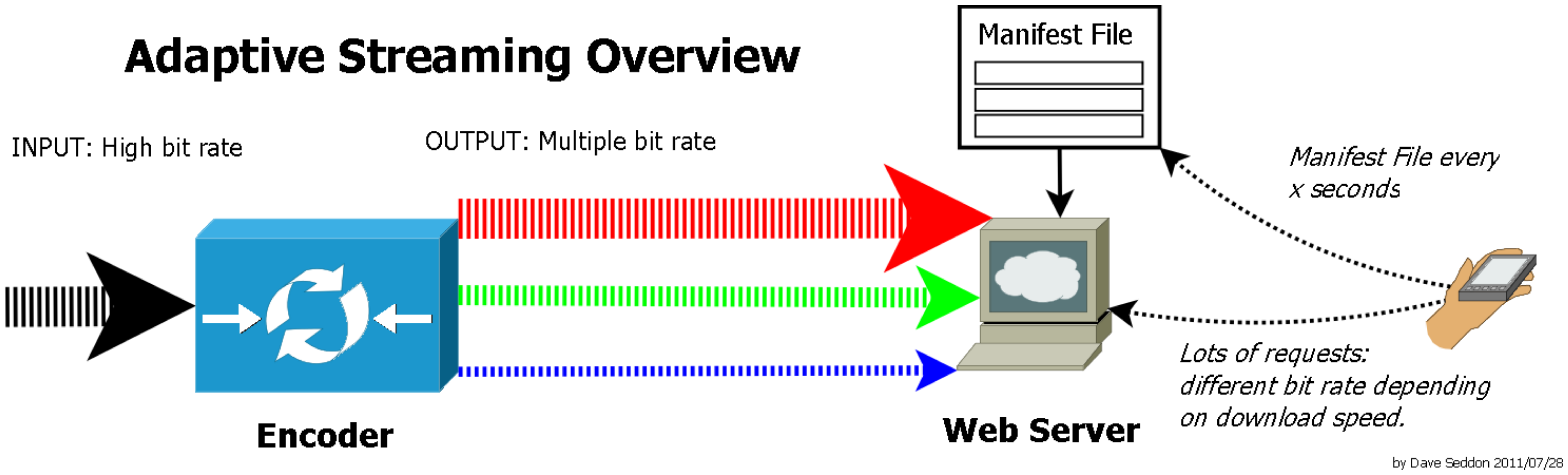
- The system buffers excess data for later processing, once input rates stabilize.
- Requires a persistent input source.
- Suitable for transient load increase.

Scale resource allocation:

- Addresses the case of increased load and additionally ensures no resources are left idle when the input load decreases.

Load shedding

- Load shedding is the process of **discarding data** when input rates increase beyond system capacity.
- Load shedding techniques operate in a dynamic fashion: the system detects an overload situation during runtime and selectively drops tuples according to a QoS specification.
- Similar to **congestion control** or video streaming in a lower quality.



https://commons.wikimedia.org/wiki/File:Adaptive_streaming_overview_daseddon_2011_07_28.png

Load shedding as an optimization problem

N: query network

I: set of input streams with known arrival rates

C: system processing capacity

H: headroom factor, i.e. a conservative estimate of the percentage of resources required by the system at steady state

Load(N(I)): the load as a fraction of the total capacity C that network N(I) presents

U_{acc}: the aggregate utility

Find a new network N' such that

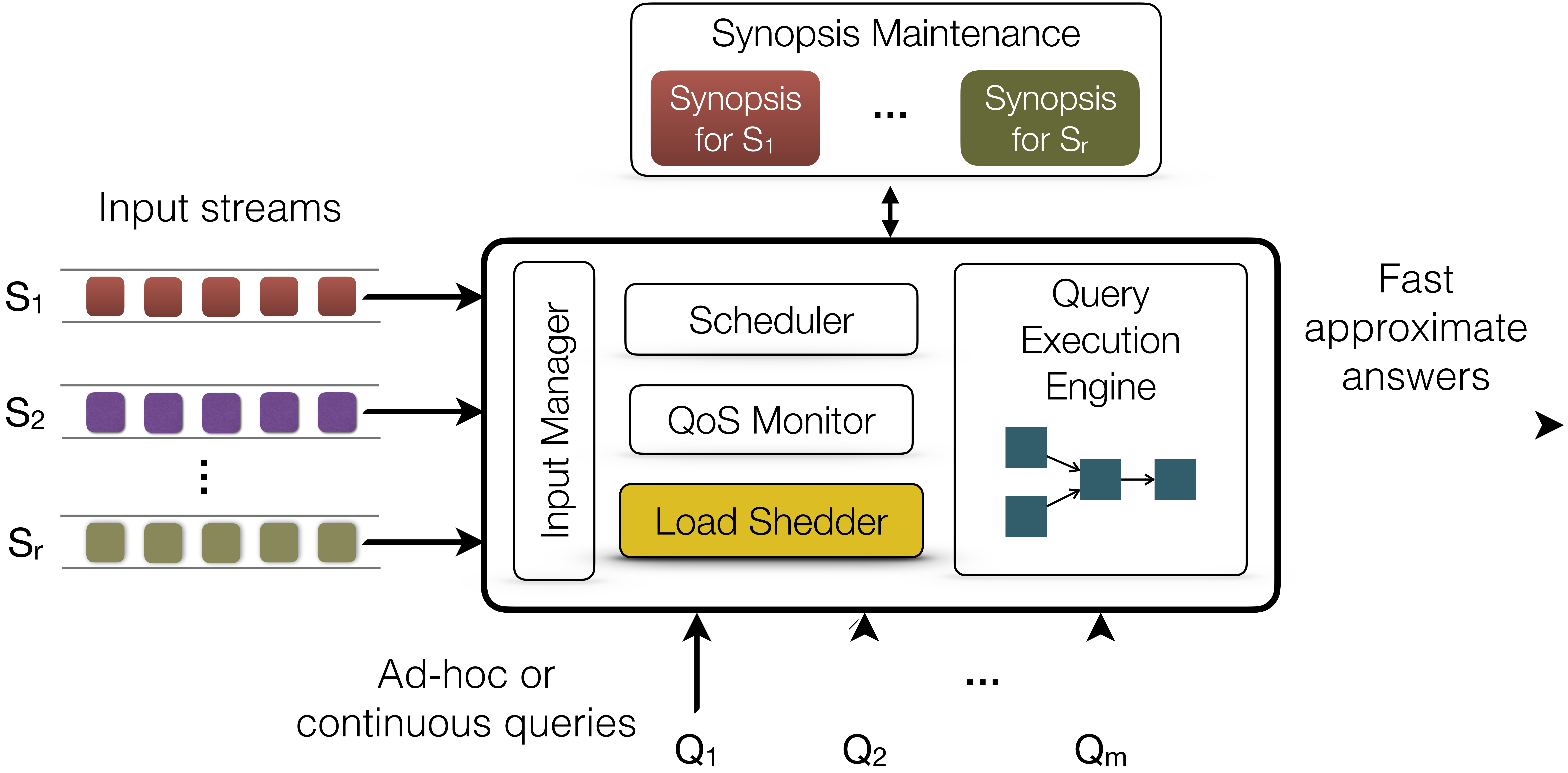
Load(N’(I))< H x C and

U_{acc}(N(I)) - U_{acc}(N'I)) is minimized

Implementation

- Load shedding is commonly implemented by a standalone component integrated with the stream processor
- The **load shedder** continuously monitors input rates or other system metrics and can access information about the running query plan
  - It detects overload and decides what actions to take in order to maintain acceptable latency and minimize result quality degradation.

DSMS with load shedder



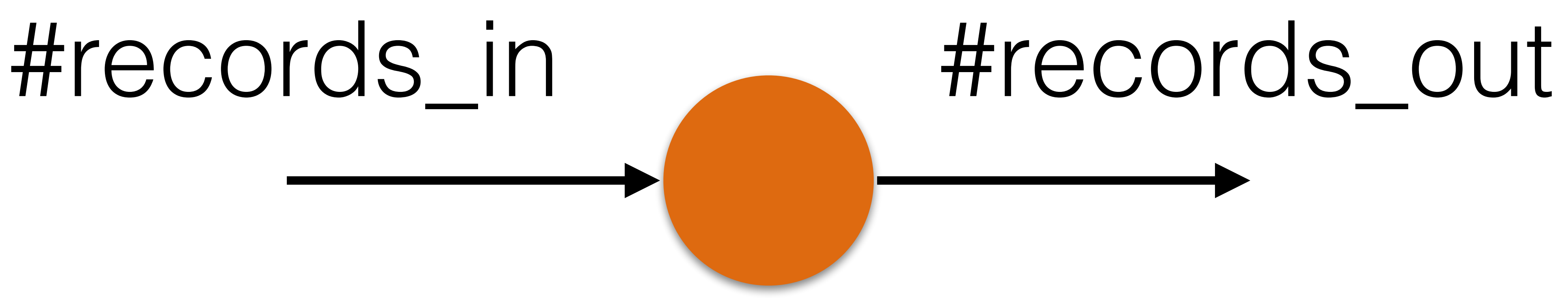
Load shedding decisions

- When to shed load?
  - detect overload quickly to avoid latency increase
  - monitor input rates
- Where in the query plan?
  - dropping at the sources vs. dropping at bottleneck operators
- How much load to shed?
  - enough for the system to keep-up
- Which tuples to drop?
  - improve latency to an acceptable level
  - cause only minimal results quality degradation

Detecting overload

- When to shed load? An incorrectly triggered shedding action can cause unnecessary result degradation!
- Load shedding components rely on **statistics** gathered during execution:
  - A statistics manager module monitors processing and input rates and periodically estimates operator selectivities.
  - The load shedder assigns a cost, $\mathbf{c_i}$, in cycles per tuple, and a selectivity, $\mathbf{s_i}$, to each operator **i**.
  - The statistics manager collects metrics and estimates those parameters either continuously or by running the system for a designated period of time, prior to regular query execution.

Estimating cost and selectivity



- Selectivity: how many records does the operator produce per record in its input?
  - map: 1 in 1 out
  - filter: 1 in, 1 or 0 out
  - flatMap, join: 1 in 0, 1, or more out
- Cost: how many records can an operator process in a unit of time?

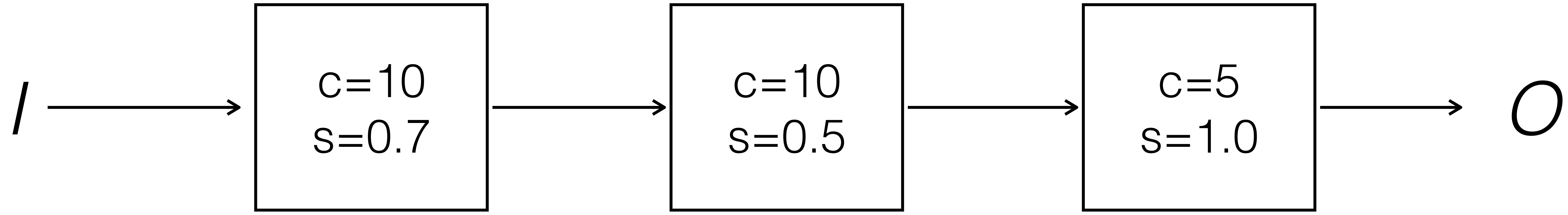
Overload detection (II)

Load coefficient for input *I*:

$$L = \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} s_j \right) \times c_i$$

Total load over *m* inputs:

$$L_T = \sum_{i=1}^{m} L \times r_i$$



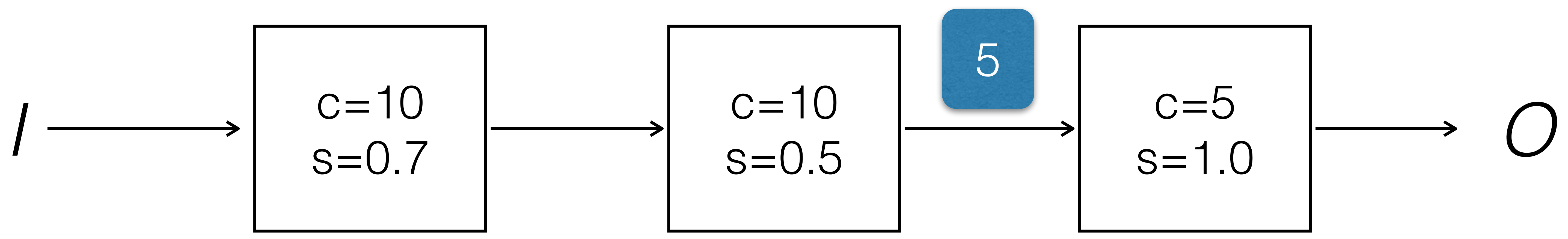
Overload detection (II)

Load coefficient for input *I*:

$$L = \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} s_j\right) \times c_i$$

Total load over *m* inputs:

$$L_T = \sum_{i=1}^{m} L \times r_i$$



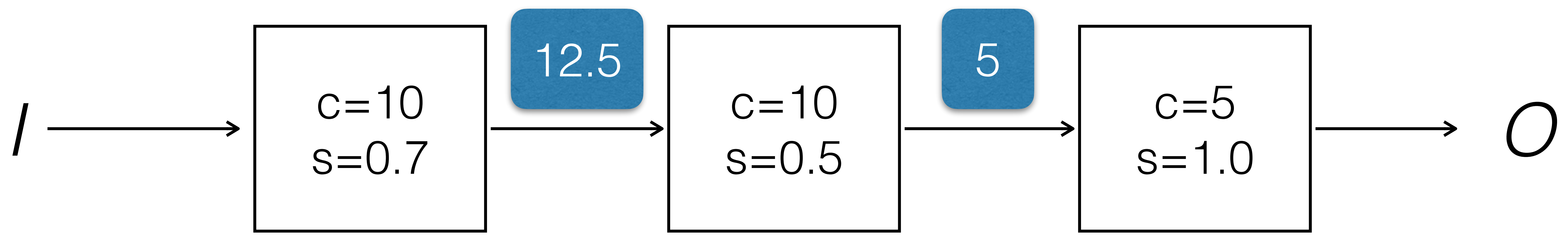
Overload detection (II)

Load coefficient for input *I*:

$$L = \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} s_j \right) \times c_i$$

Total load over *m* inputs:

$$L_T = \sum_{i=1}^{m} L \times r_i$$



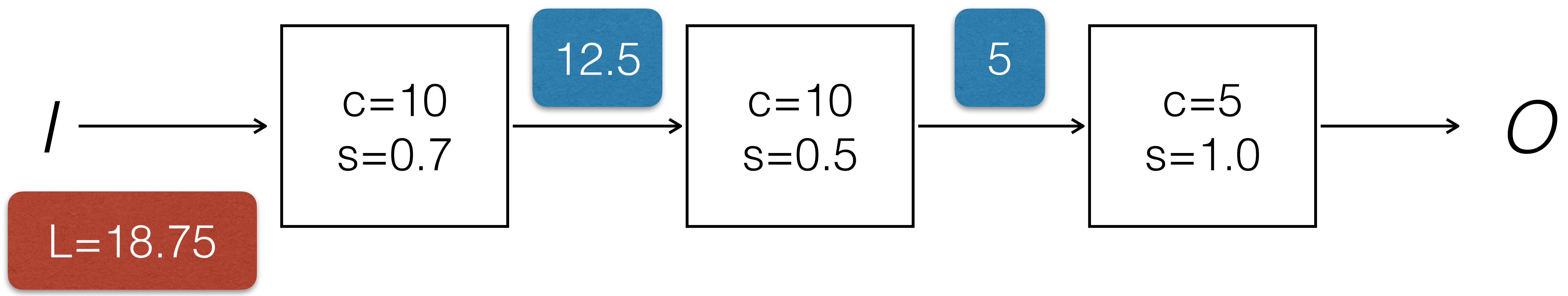
Overload detection (II)

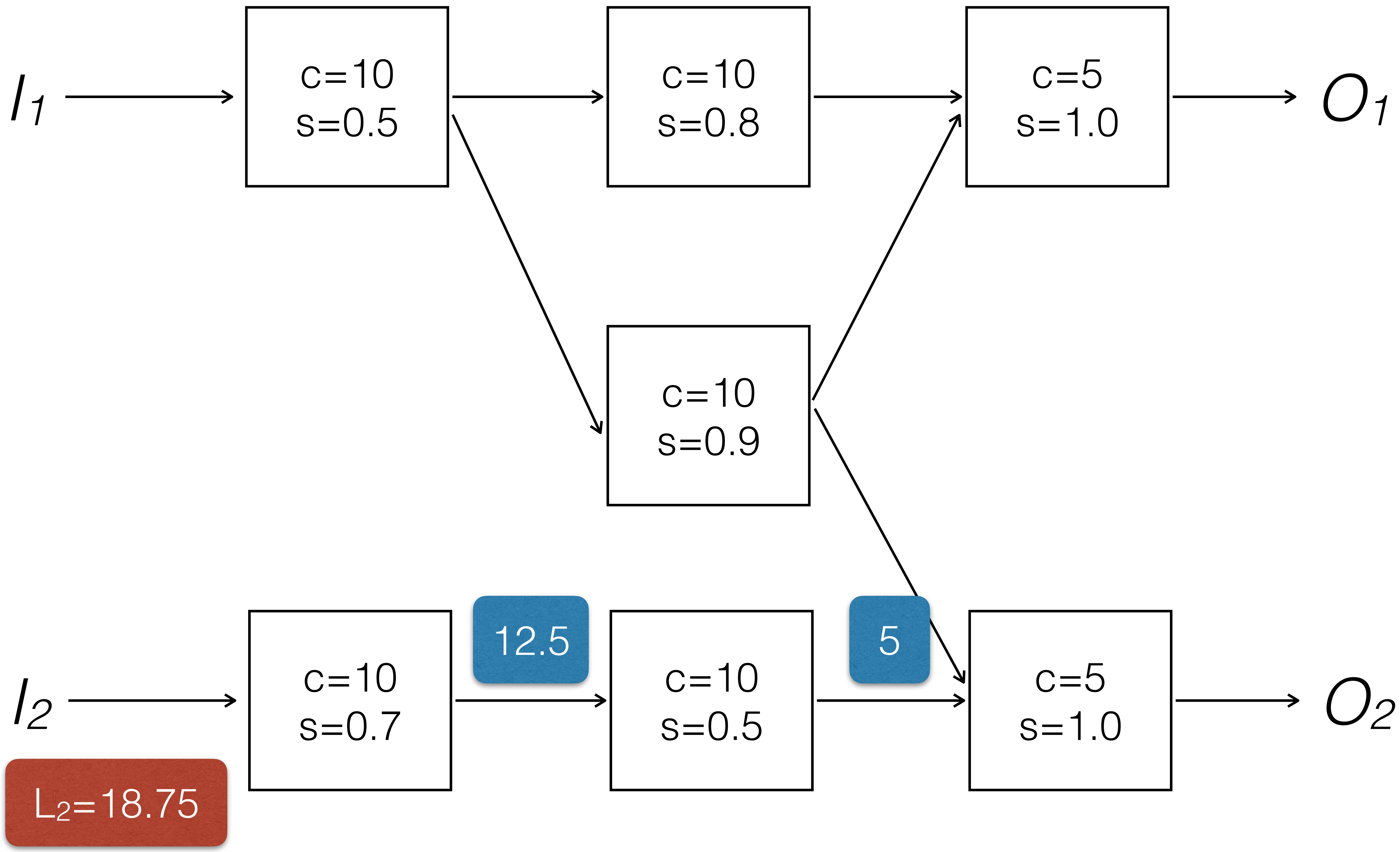
Load coefficient for input *I*:

$$L = \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} s_j \right) \times c_i$$

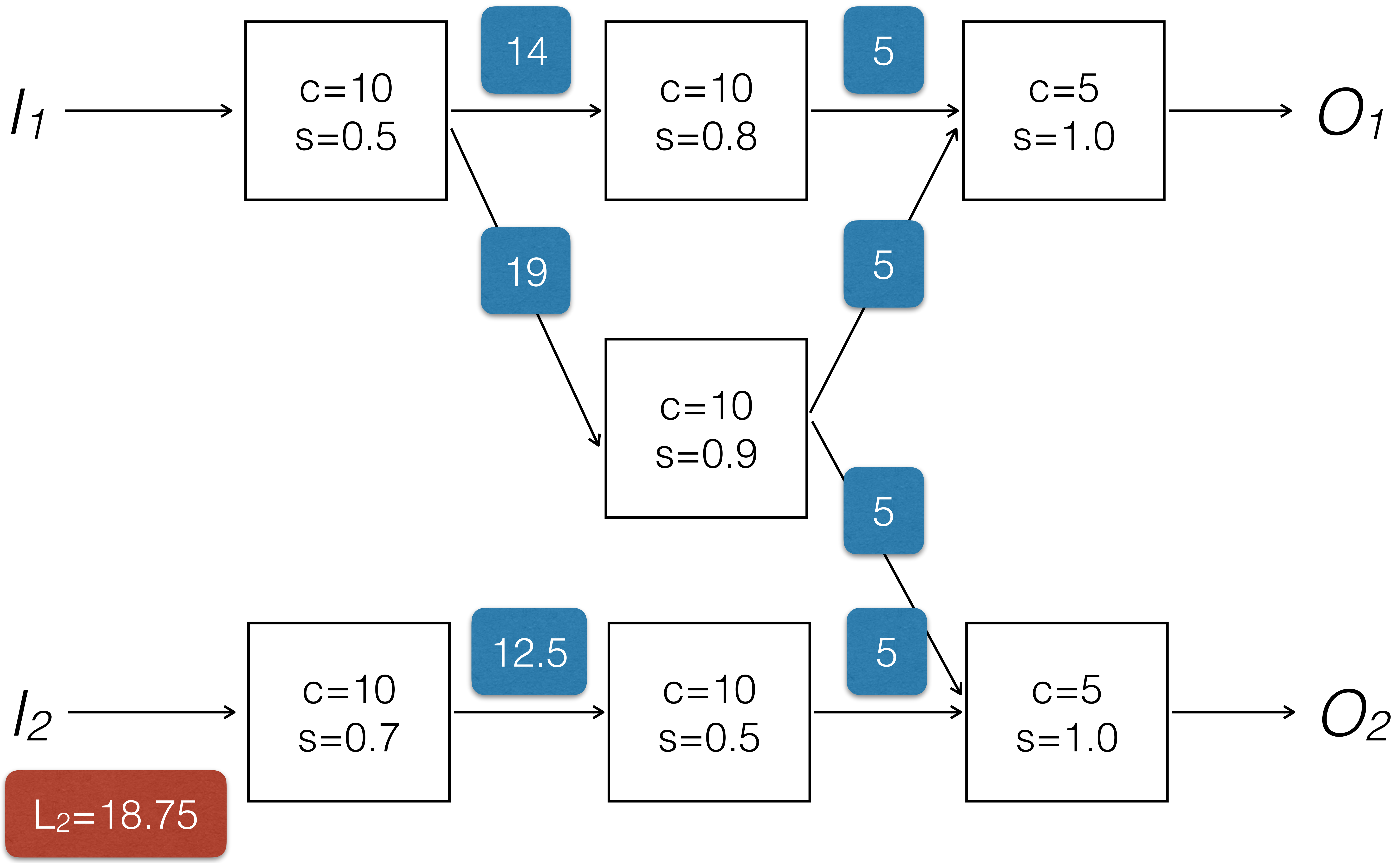
Total load over *m* inputs:

$$L_T = \sum_{i=1}^{m} L \times r_i$$

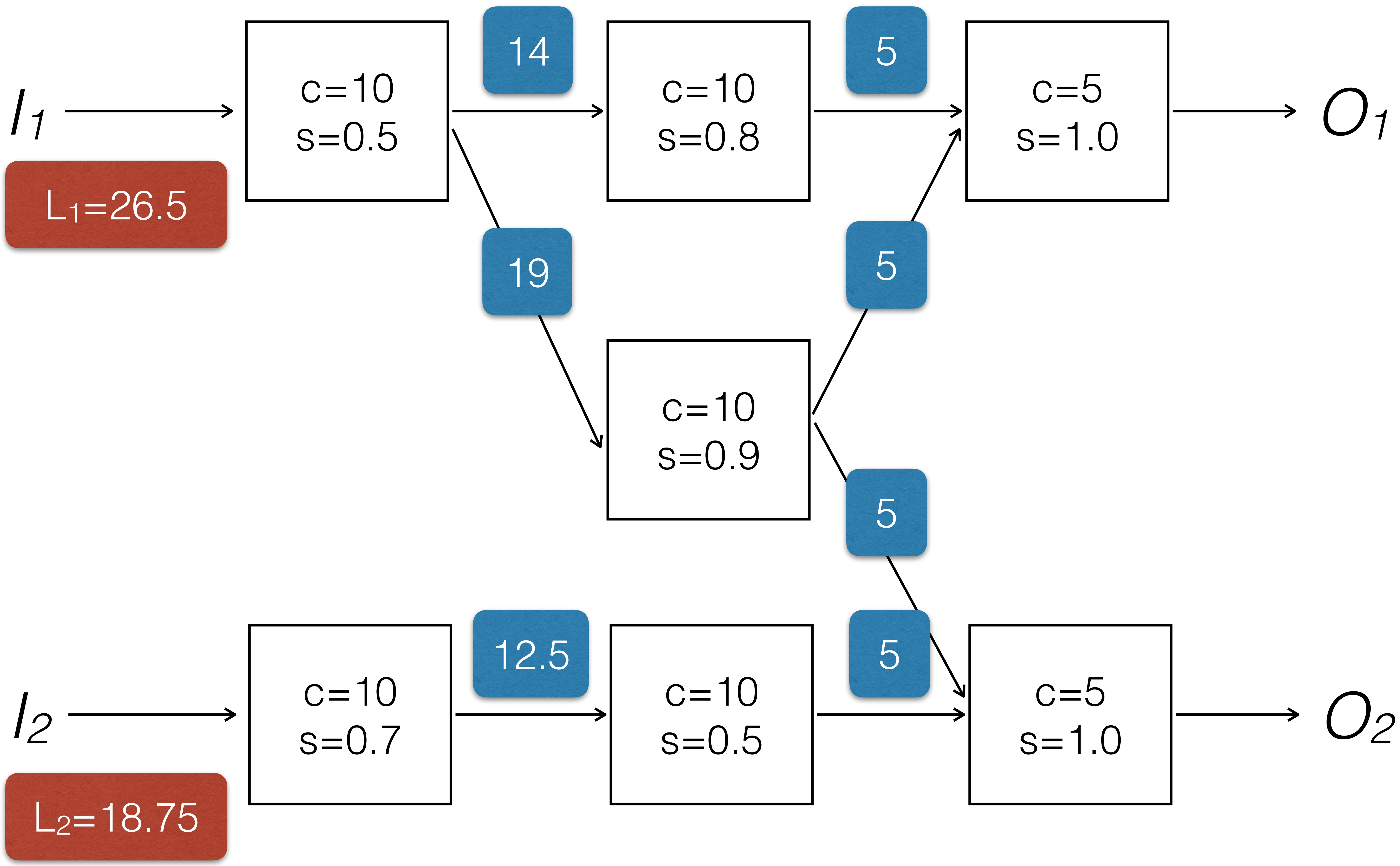




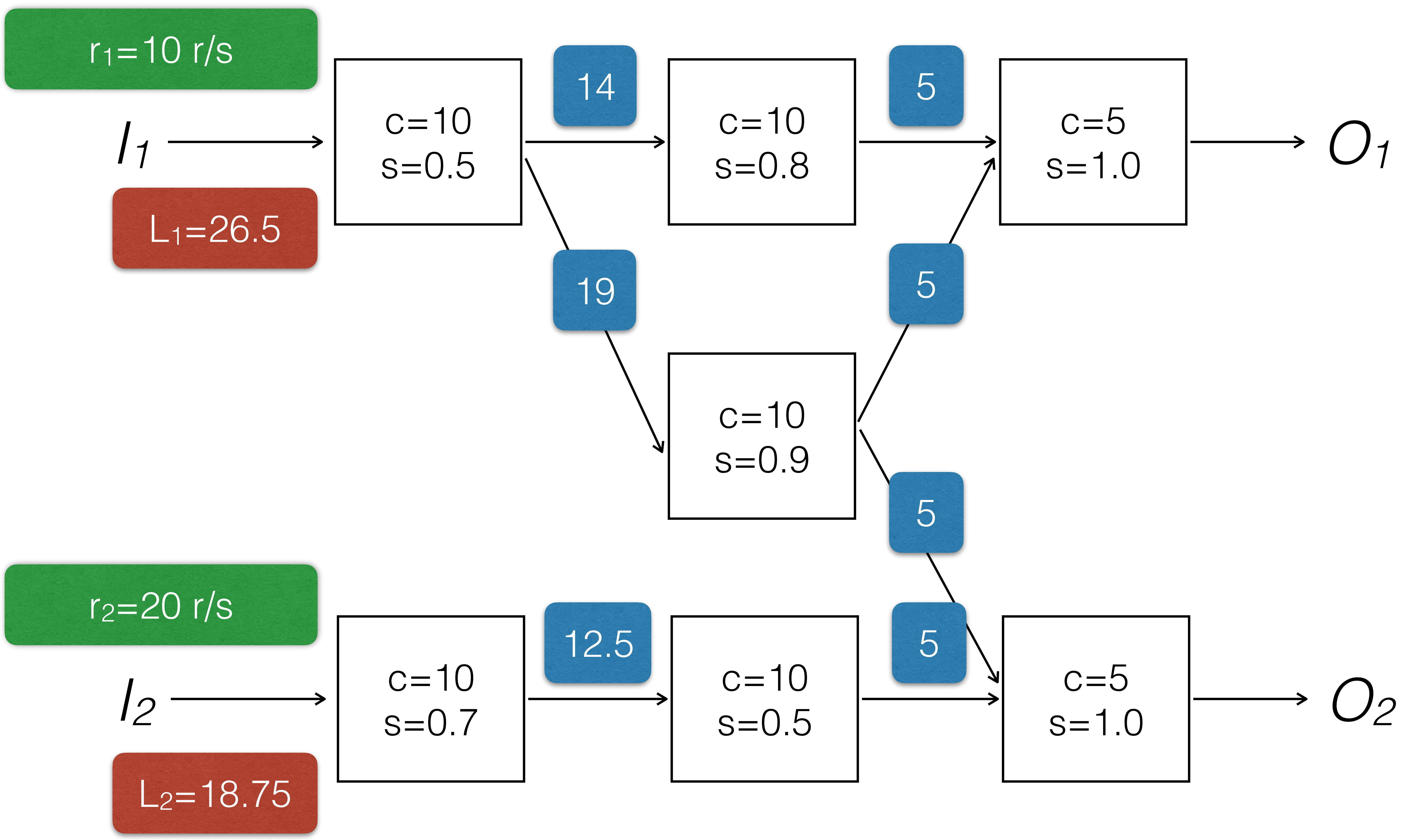
I1
c=10
s=0.5
c=10
s=0.8
c=5
s=1.0
O1
c=10
s=0.9
I2
c=10
s=0.7
12.5
c=10
s=0.5
5
c=5
s=1.0
O2
L2=18.75

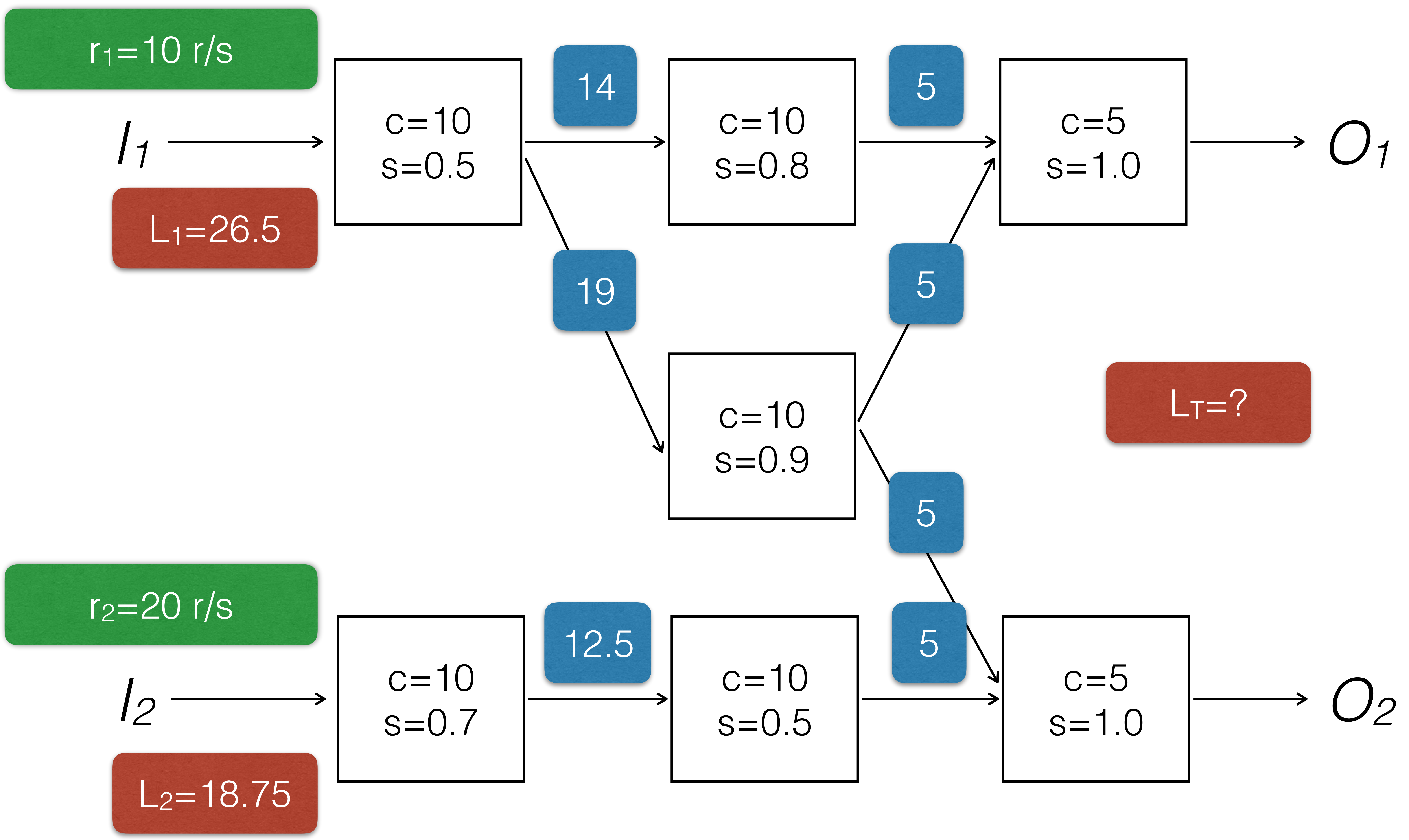


I1
c=10
s=0.5
14
c=10
s=0.8
5
c=5
s=1.0
O1
19
5
c=10
s=0.9
5
I2
c=10
s=0.7
12.5
c=10
s=0.5
5
c=5
s=1.0
O2
L2=18.75

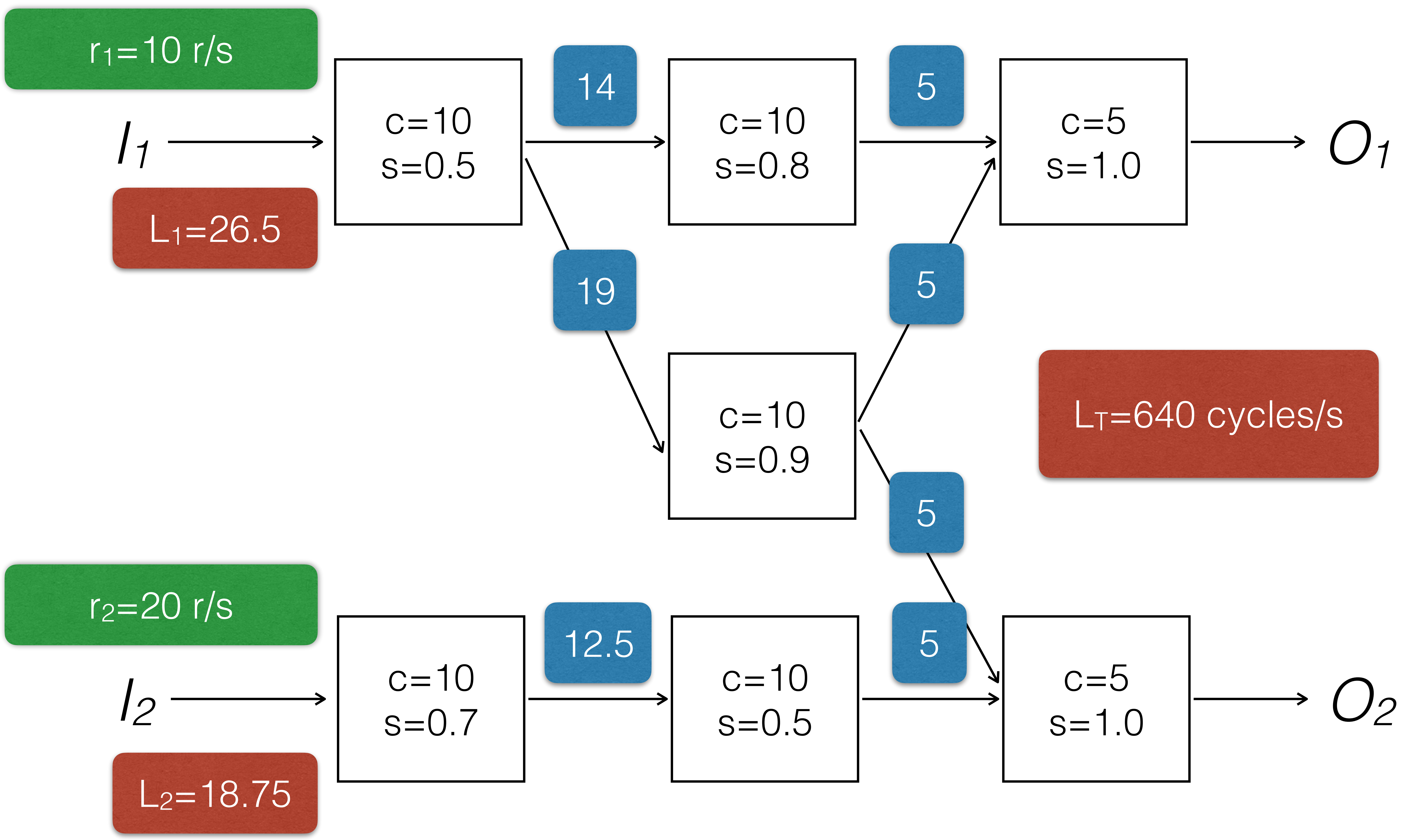


I1
L1=26.5
c=10
s=0.5
14
19
c=10
s=0.8
5
5
c=10
s=0.9
5
c=5
s=1.0
O1
I2
L2=18.75
c=10
s=0.7
12.5
c=10
s=0.5
5
c=5
s=1.0
O2


r1=10 r/s
I1
L1=26.5
c=10
s=0.5
14
c=10
s=0.8
5
c=5
s=1.0
O1
19
5
c=10
s=0.9
5
r2=20 r/s
I2
L2=18.75
c=10
s=0.7
12.5
c=10
s=0.5
5
c=5
s=1.0
O2



r1=10 r/s
I1
L1=26.5
c=10
s=0.5
14
c=10
s=0.8
5
c=5
s=1.0
O1
19
5
c=10
s=0.9
LT=?
5
r2=20 r/s
I2
L2=18.75
c=10
s=0.7
12.5
c=10
s=0.5
5
c=5
s=1.0
O2



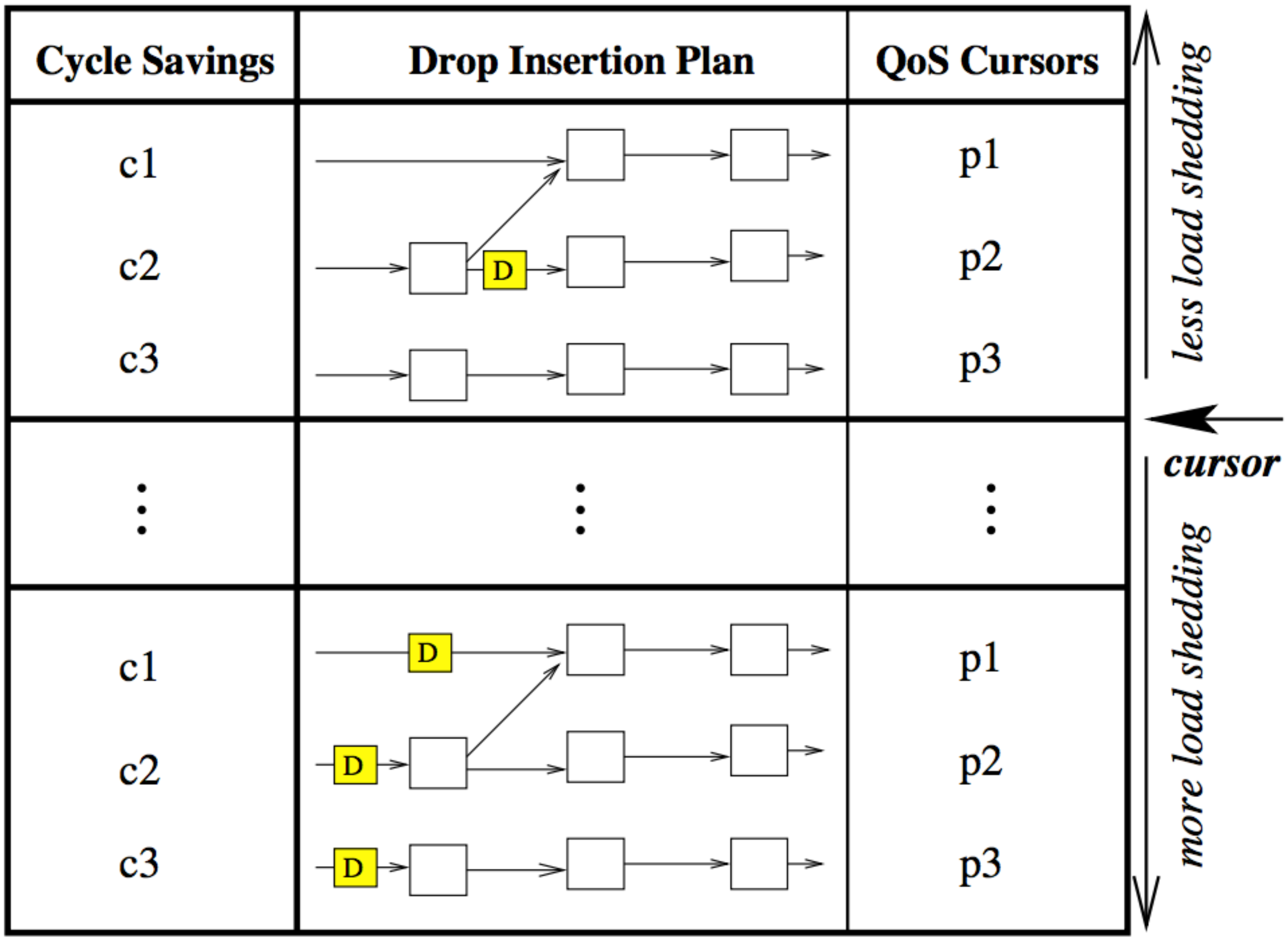
r1=10 r/s
I1
L1=26.5
c=10
s=0.5
14
19
c=10
s=0.8
5
5
c=10
s=0.9
5
c=5
s=1.0
O1
LT=640 cycles/s
r2=20 r/s
I2
L2=18.75
c=10
s=0.7
12.5
c=10
s=0.5
5
c=5
s=1.0
O2

Reacting to overload

- **Where** in the query plan to drop tuples, **which** tuples, and **how many**
- The question of where is equivalent to placing special **drop operators** in the best positions in the query plan
- Drop operators can be placed at any location in the query plan
- Dropping near the source avoids wasting work but it might affect results of multiple queries if the source is connected to multiple queries.

Load Shedding Road Map (LSRM)

- A pre-computed table that contains materialized **load shedding plans** ordered by how much load shedding they will cause.
- Each row contains a plan with
  - expected cycle savings
  - locations for drop operations
  - drop amounts
  - QoS effects (provided that tuples can be associated with a utility metric)


Cycle Savings
Drop Insertion Plan
QoS Cursors
c1
c2
c3
D
p1
p2
p3
less load shedding
cursor
more load shedding
c1
c2
c3
D
D
D
p1
p2
p3

Which tuples to drop?

- Relevant when load shedding takes into account the **semantic** importance of tuples with respect to results quality
- Drop at **random**:
  - Insert random sampling operators in the query plan, parametrized with a **sampling rate**
  - The rate defines the probability to discard a tuple and is computed based on statistics and operator selectivity
  - The optimization objective is to achieve the highest possible accuracy given the constraint that system throughput matches the data input rate
  - In the case of known aggregation functions, results can be scaled using approximate query processing techniques, where accuracy is measured in terms of relative error in the computed query answers.

Which tuples to drop?

- **Window-aware** load shedding applies shedding to entire windows instead of individual tuples
  - When discarding tuples at the sources or another point in a query with multiple window aggregations, it is unclear how shedding will affect the correctness of downstream window operators.
  - This approach **preserves window integrity** and guarantees that the results under shedding will not be approximations but a subset of the exact answers.
- **Concept-driven** load shedding measures tuple utility
  - The method selects tuples to discard by relying on the notion of a window-based concept drift.
  - The metric is defined by computing a similarity metric across windows.

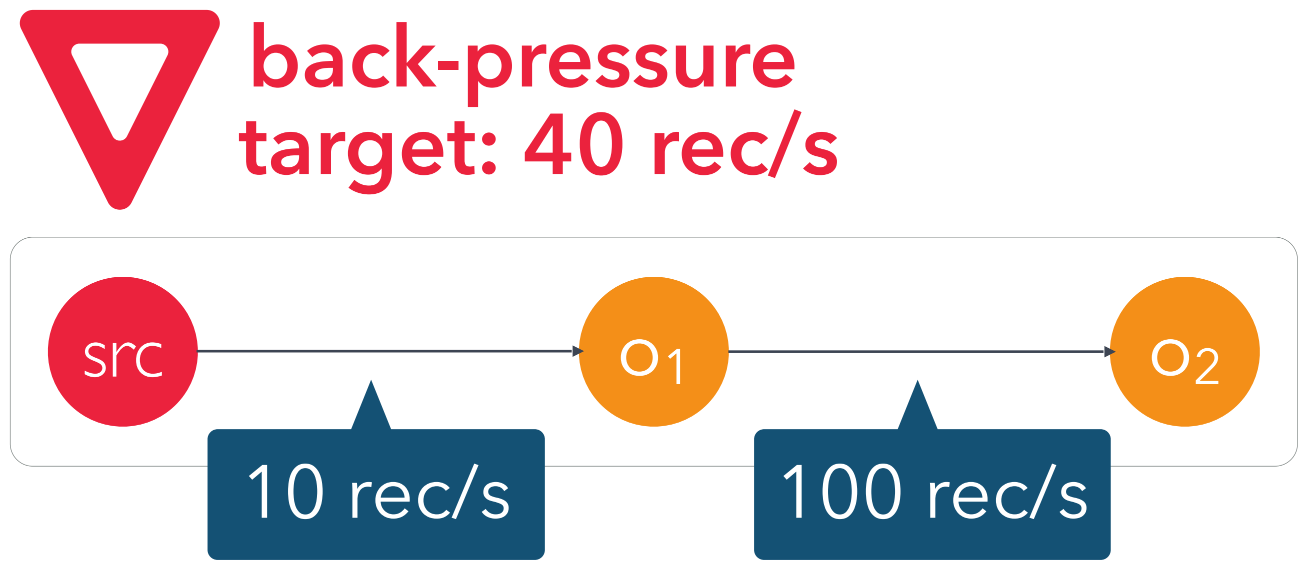
How many tuples to drop?

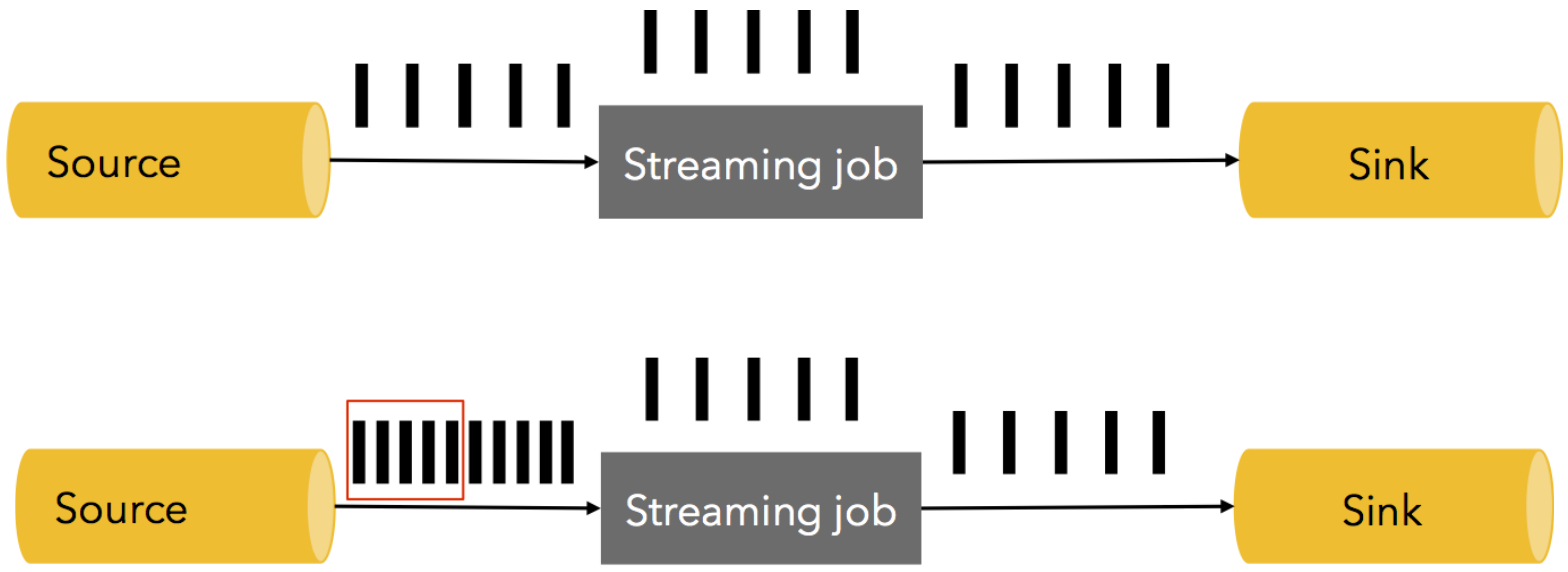
- The amount of tuples to discard strongly depends on the decisions of where and which tuples to shed.
- If input rates and processing capacity are known or easy to measure, estimates can be computed in a straight-forward manner.
- Estimations based on static operator selectivities and heuristics are unsuitable for frequent load fluctuations.
- Naive approaches can lead to system instability or unnecessary load shedding.
- In window-aware load shedding, queries need to define a batch size: an application-specific maximum tolerance to gaps.
  - This parameter indicates how many consecutive missing results the query can tolerate.

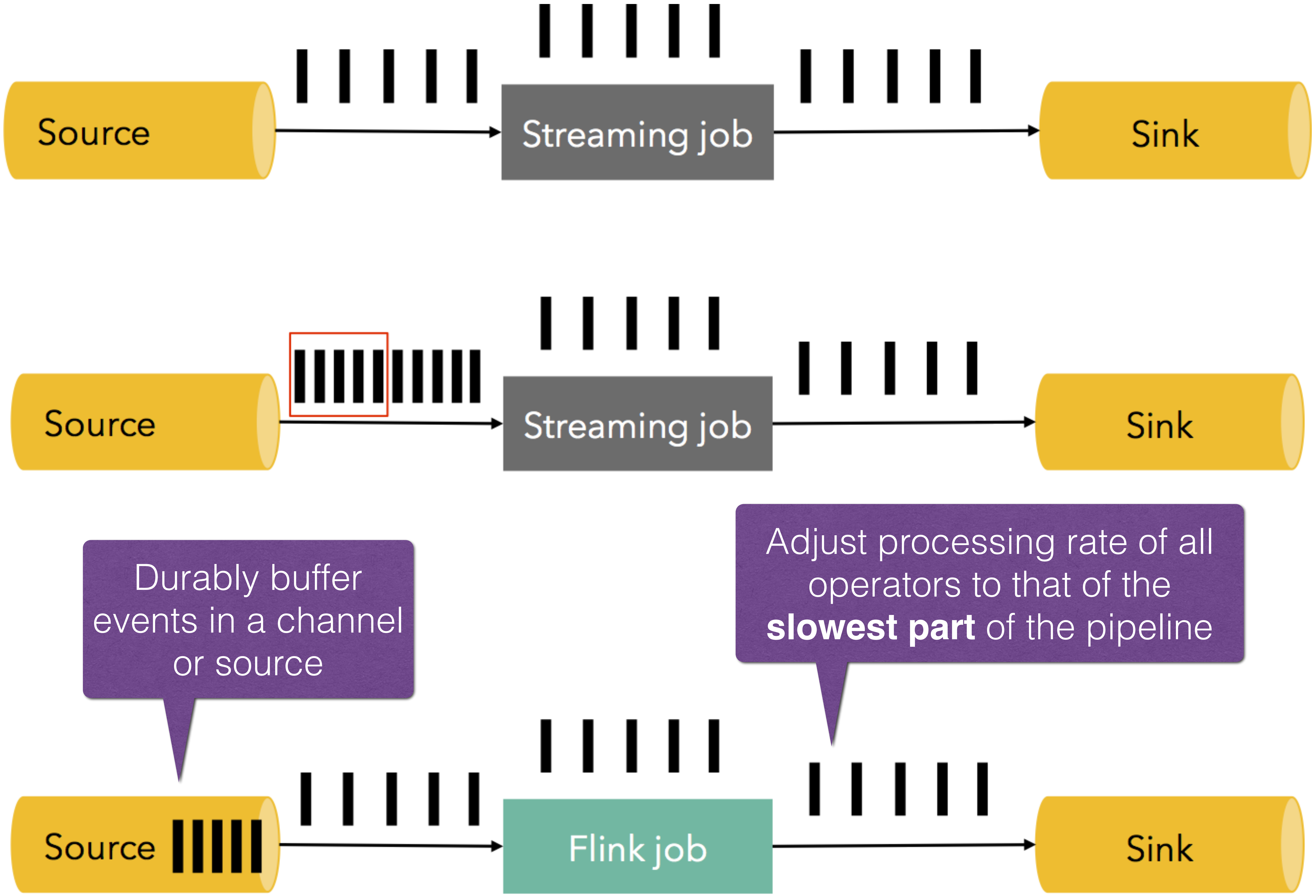
Backpressure

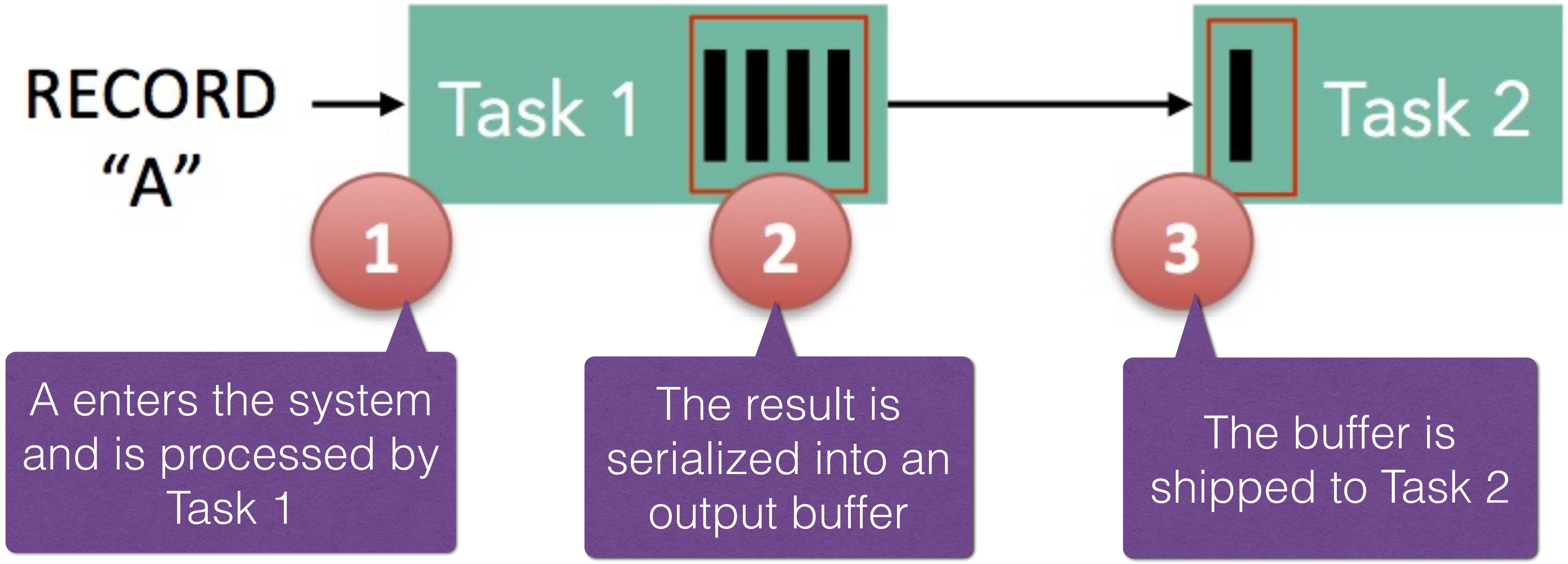
Rate control

- In a network of consumers and producers such as a streaming execution graph with multiple operators, back-pressure has the effect that **all operators slow down** to match the processing speed of **the slowest consumer**.
- If the bottleneck operator is far down the dataflow graph, back-pressure propagates to upstream operators, eventually reaching the data stream sources.
- To ensure no data loss, a persistent input message queue, such as Kafka, and enough storage is required.




Source
Streaming job
Sink
Source
Streaming job
Sink


Source
Streaming job
Sink
Source
Streaming job
Sink
Durably buffer events in a channel or source
Adjust processing rate of all operators to that of the **slowest part** of the pipeline
Source
Flink job
Sink



Progress is controlled though buffer availability

- Each produced and consumed stream have managed buffer pools with bounded capacity.
- A buffer pool is a set of buffers which are recycled after they have been consumed and can be re-used.

Rate adjustment

Local exchange: If both producer and consumer run on the same node the buffer is recycled as soon as it is consumed.

- The producer slows down according to the rate the consumer recycles buffers.

Remote exchange: If tasks run on different worker nodes, the buffer can be recycled as soon as it is on the TCP channel.

- If there is no buffer on the consumer side, reading from the TCP connection is interrupted.
- The producer uses a threshold to control how much data is *in-flight.*
- The producer is slowed down if it cannot put new data on the wire.

Remarks on buffer-based rate control

- Simple mechanism:the buffer occupancy controls the data rate automatically.
- The maximum throughput is limited by the processing rate of the slowest task.
- Parallel tasks are connected via virtual channels multiplexed over TCP connections:
  - In the presence of skew, a single overload channel can cause the slowdown of the entire dataflow… can we do better?

Credit-based flow control

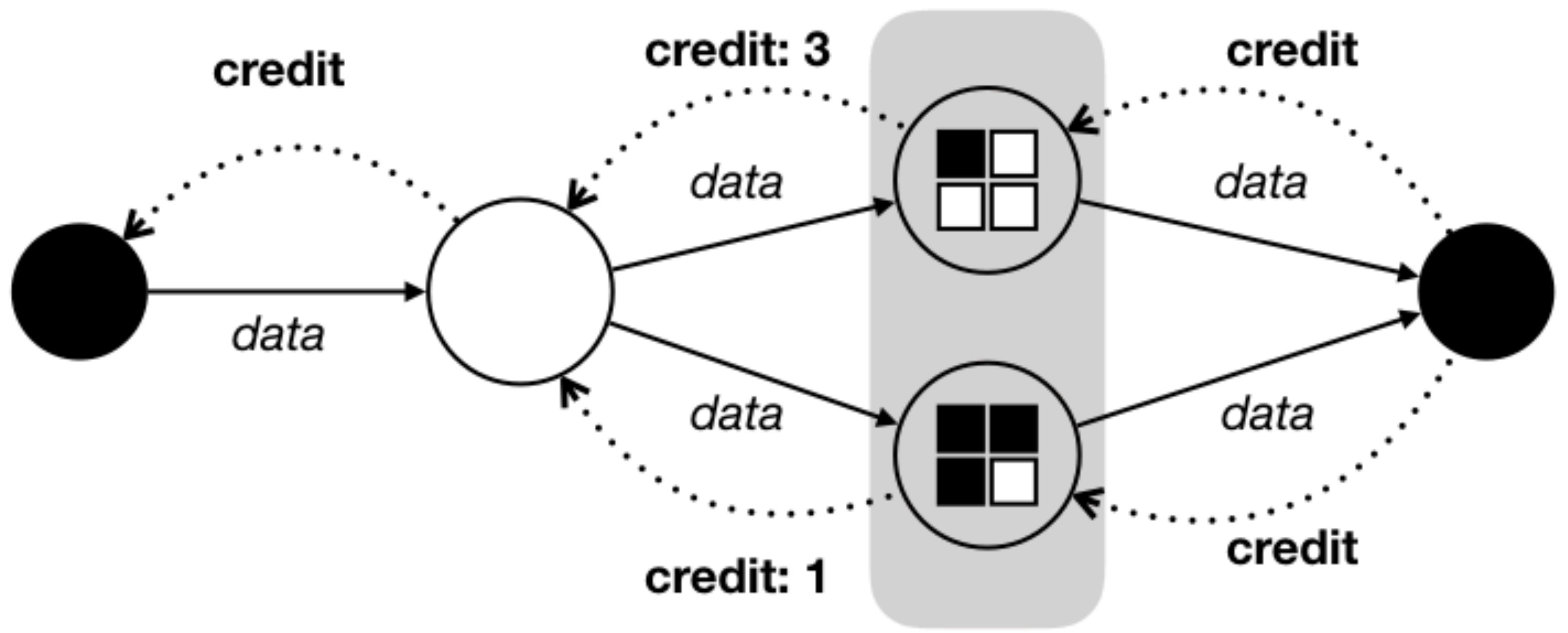
- Credit-based flow control (CFC) is a **link-by-link**, per virtual channel congestion control technique used in ATM network switches.

- To exchange data through an ATM network, each pair of endpoints first needs to establish a virtual circuit (VC) or connection.

- CFC uses a **credit system** to signal the availability of buffer space from receivers to senders.

- Senders maintain a **credit balance** for all their receivers and receivers regularly send notifications upstream containing their number of available credits.

- One credit corresponds to some amount of buffer space so that a sender can know how much data they can afford to forward downstream.



Credit-based flow control

- This classic networking technique turns out to be very useful for load management in modern, highly-parallel stream processors and is implemented in Apache Flink.

- Each task informs its senders of its buffer availability via credit messages.

- This way, senders always know whether receivers have the required capacity to handle data messages.

- When the credit of a receiver drops to zero (or a specified threshold), backpressure appears on its virtual channel.

Remarks on CFC

- Bakcpressure is inflicted on pairs of communicating tasks only
  - it does not interfere with other tasks sharing the same TCP connection.
- CFC maximizes network utilization and prevents faults caused by high congestion.
- In the presence of bursty traffic, CFC causes backpressure to build up fast and propagate along congested VCs to their sources which can be throttled.
- Essentially, CFC allows blocking excess traffic *outside the network* to protect it.
  - This is crucial in the presence of data skew where a single overloaded task could otherwise block the flow of data to all other downstream operator instances.
- On the downside, the additional credit announcement messages might increase end-to-end latency.

Lecture references

- Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. **Load shedding in a data stream manager**. (VLDB ’03)
- N. Tatbul and S. Zdonik. **Window-aware load shedding for aggregation queries over data streams**. (VLDB’06)
- N. Tatbul, U. Çetintemel, and S. Zdonik. **Staying fit: Efficient load shedding techniques for distributed stream processing**. (VLDB’07)
- N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. **Concept-driven load shedding: Reducing size and error of voluminous and variable data streams**. (IEEE Big Data ’18)
- H. T. Kung, T. Blackwell, and A. Chapman. **Credit-based flow control for atm networks: Credit update protocol, adaptive credit allocation and statistical multiplexing**. (ACM SGCOMM’94).
- https://www.ververica.com/blog/how-flink-handles-backpressure
- https://flink.apache.org/2019/06/05/flink-network-stack.html