CS 591 K1: Data Stream Processing and Analytics

Spring 2021

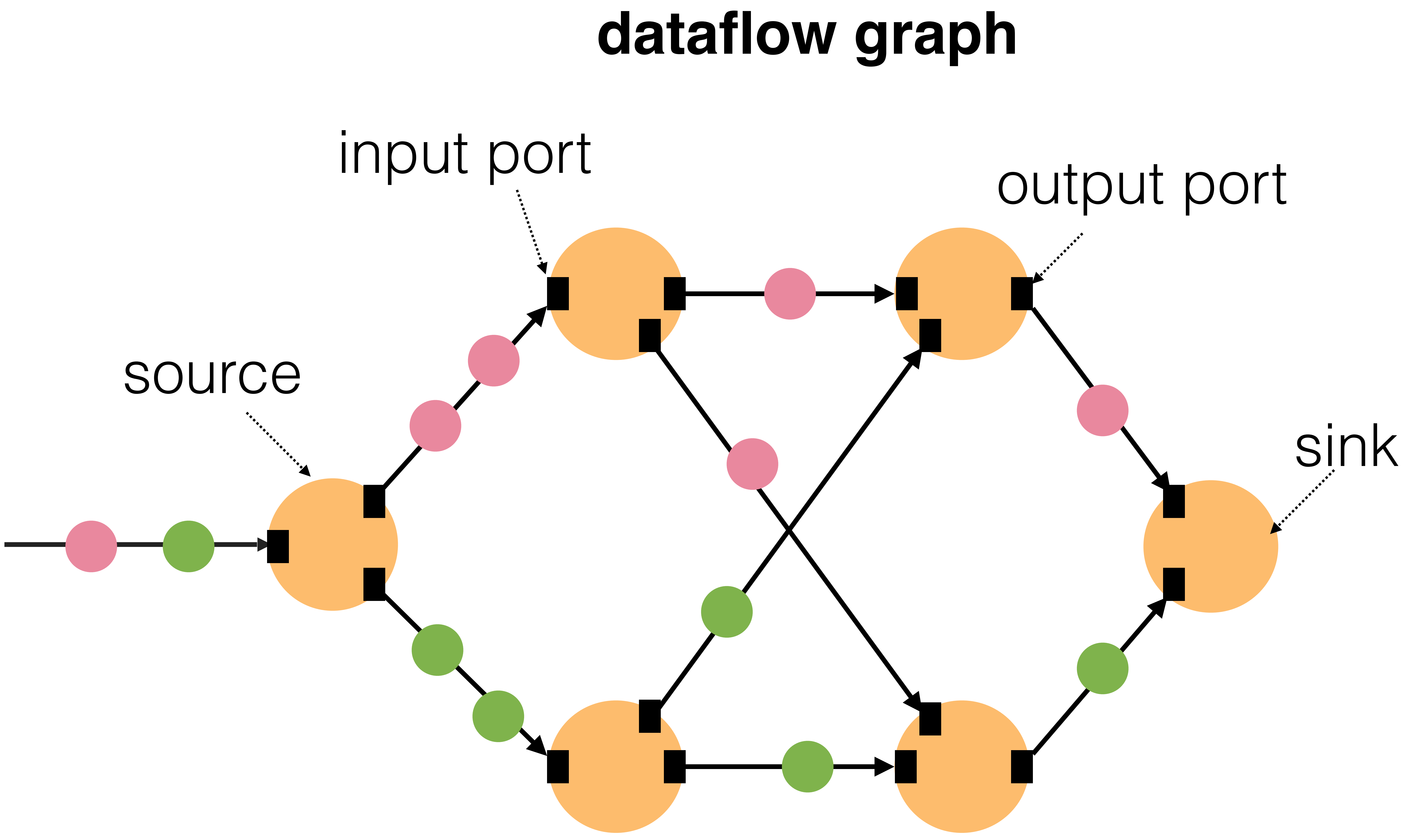
Stream processing optimizations

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Topics covered in this lecture

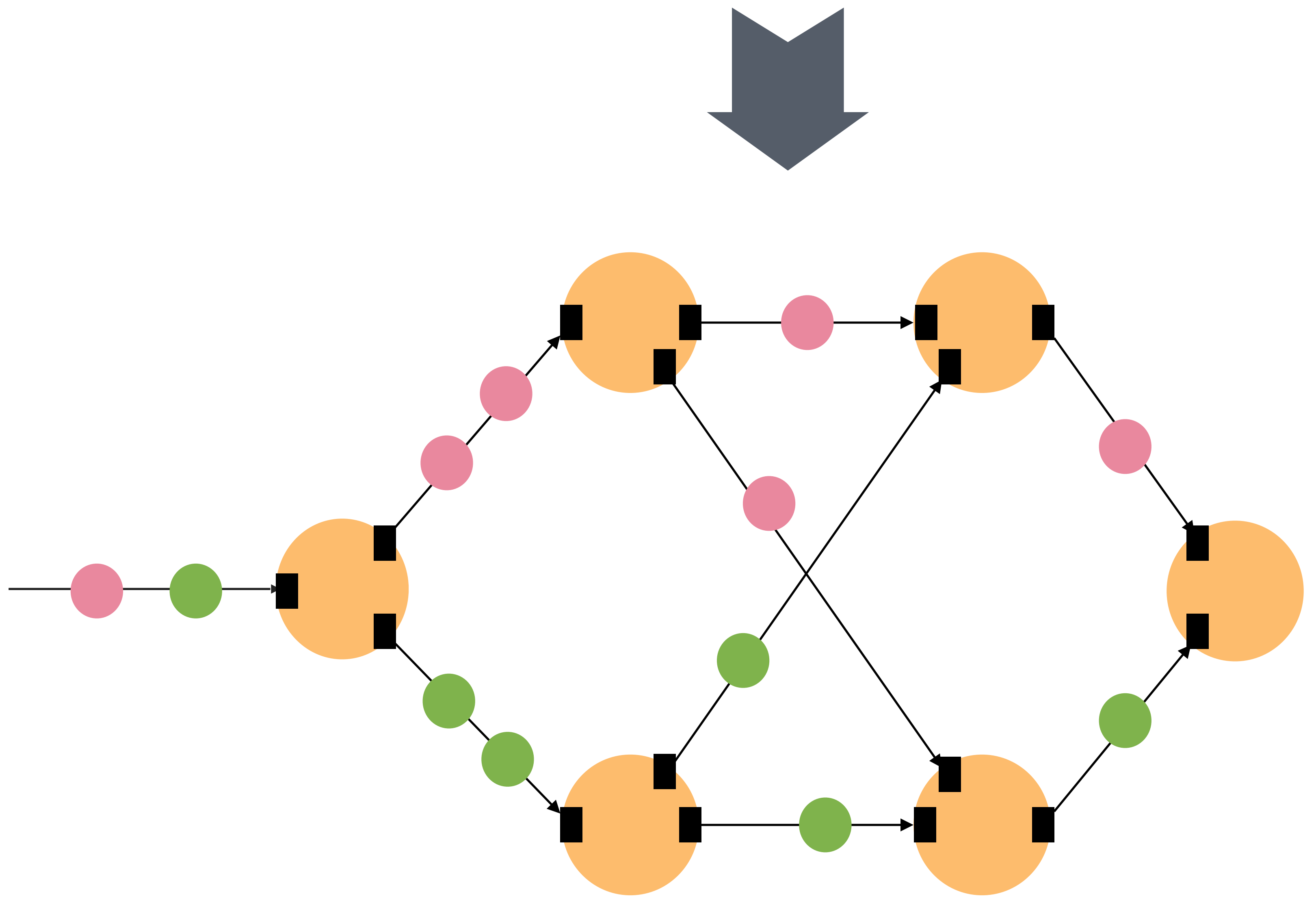
- Costs of streaming operator execution
    - state, parallelism, selectivity
- Dataflow optimizations
    - plan translation alternatives
- Runtime optimizations
    - load management, scheduling, state management
- Optimization semantics, correctness, profitability

Revisiting the basics



Revisiting the basics

A series of transformations
on streams in
Stream SQL, Scala, Python,
Rust, Java…



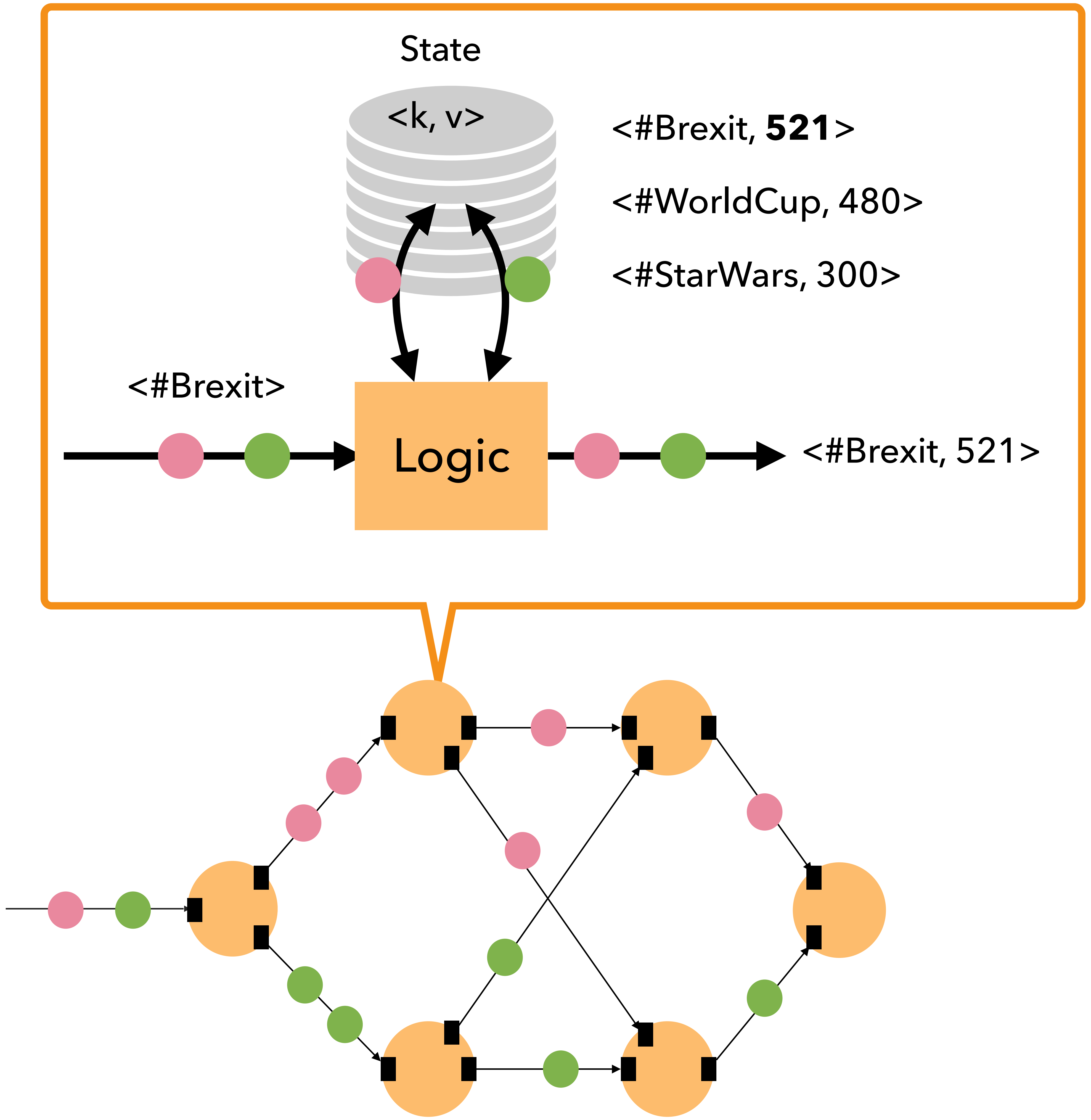
Dataflow graph

- operators are nodes, data channels are edges
- channels have FIFO semantics
- streams of data elements flow continuously along edges

Operators

- receive one or more input streams
- perform tuple-at-a-time, window, logic, pattern matching transformations
- output one or more streams of possibly different type

Stateful operators



- Stateful operators maintain state that reflect part of the stream history they have seen
  - windows, continuous aggregations, distinct…
- State is commonly partitioned by key
- State can be cleared based on watermarks or punctuations
  - window fires, post becomes inactive

Operator selectivity

- The number of output elements produced per number of input elements
  - a map operator has a selectivity of 1, i.e. it produces one output element for each input element it processes
  - an operator that tokenizes sentences into words has selectivity > 1
  - a filter operator typically has selectivity < 1

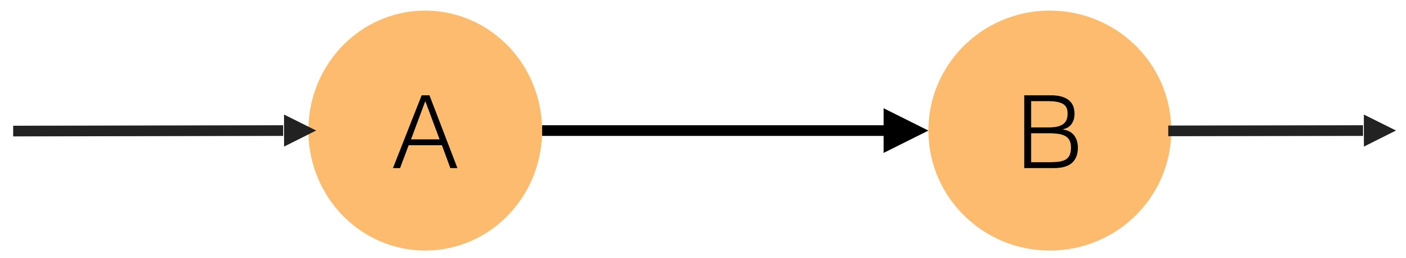
Operator selectivity

- The number of output elements produced per number of input elements
  - a map operator has a selectivity of 1, i.e. it produces one output element for each input element it processes
  - an operator that tokenizes sentences into words has selectivity > 1
  - a filter operator typically has selectivity < 1

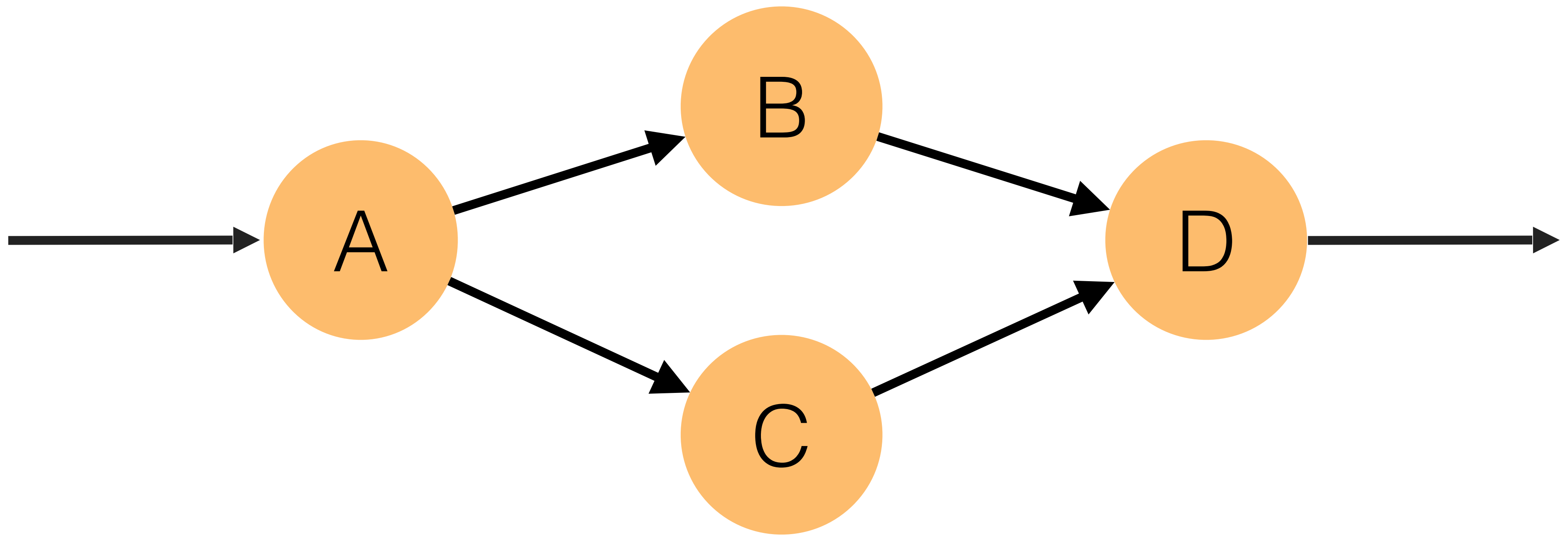
Is selectivity always known at development time?

Types of Parallelism

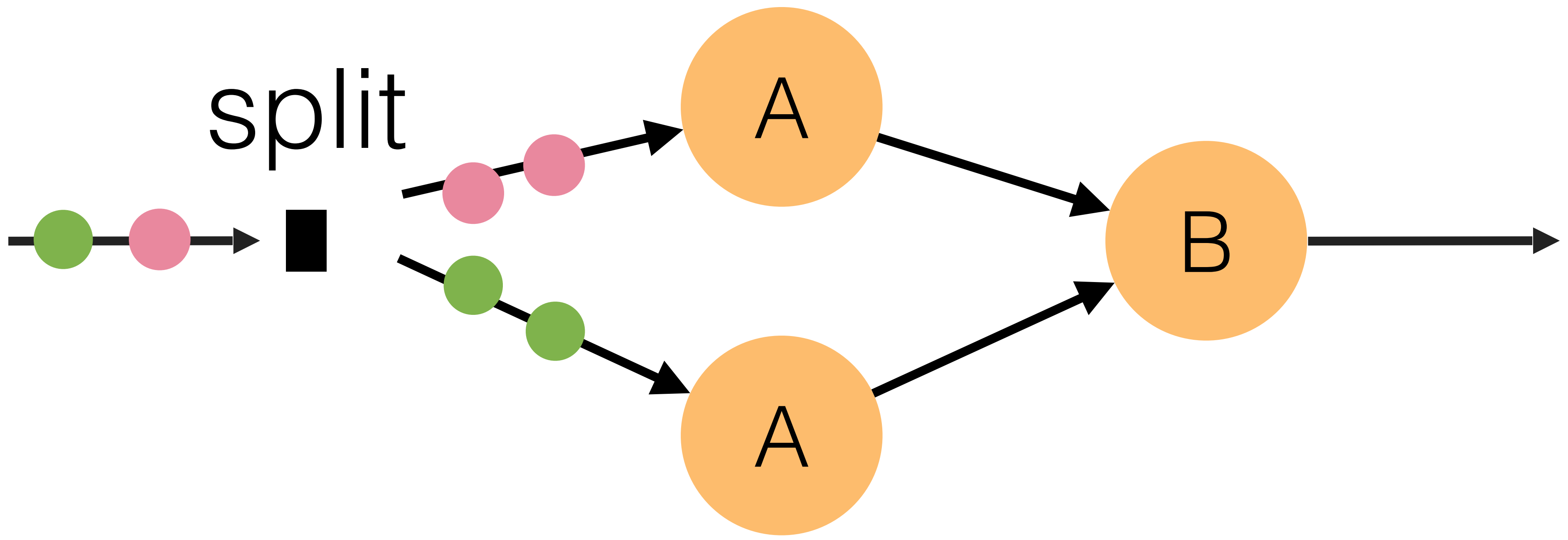
Pipeline: A || B



Task: B || C



Data: A || A



Query optimization (I)

Identify the most efficient way to execute a query

- There may exist several ways to execute a computation
  - query plans, e.g. order of operators
  - scheduling and placement decisions
  - different algorithms, e.g. hash-based vs. broadcast join
- What does performance depend on?
  - input data, intermediate data
  - operator properties
- How can we estimate the cost of different strategies?
  - before execution or during runtime

Query optimization (II)

Optimization strategies

- enumerate equivalent execution plans
- minimize intermediate data and communication

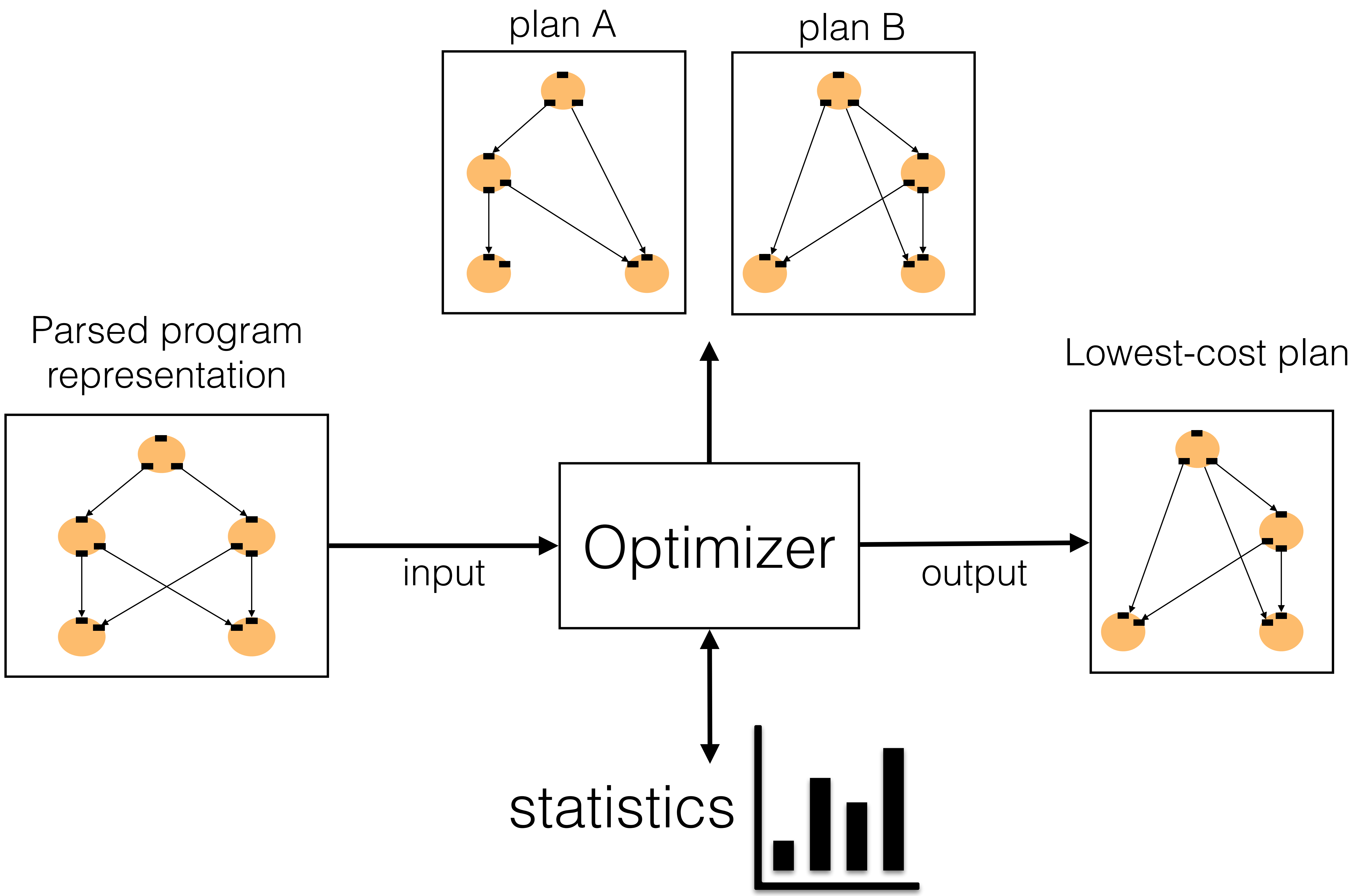
Alternatives

- data structures
- sorting vs hashing
- indexing, pre-fetching
- minimize disk access
- scheduling

Objectives

- optimize resource utilization or minimize resources
- decrease latency, increase throughput
- minimize monetary costs (if running in the cloud)

Cost-based optimization



Challenges in streaming optimization

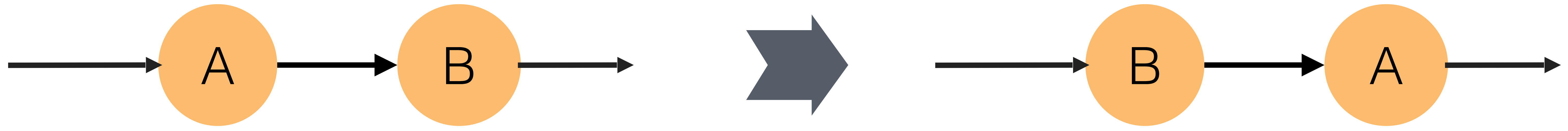
- What does *efficient* mean in the context of streaming?
  - queries run continuously
  - streams are unbounded
- In traditional ad-hoc database queries, the query plan is generated on-the-fly. Different plans can be used for two consecutive executions of the same query.
- A streaming dataflow is generated once and then scheduled for execution.
- Changing execution strategy while the query is running might be impractical.
  - state accumulation and re-partitioning
  - high-availability and low latency requirements
  - scheduling overhead

When to optimize?

- **Profitability**: under what conditions does the optimization improve performance?
    - can the decision be automatic?
- **Safety**: under what conditions does the optimization preserve correctness?
    - maintain state semantics
    - maintain result and selectivity semantics
- **Dynamism**: can the optimization be applied during runtime or does it have to be applied statically?

Catalog of Optimizations

Operator re-ordering

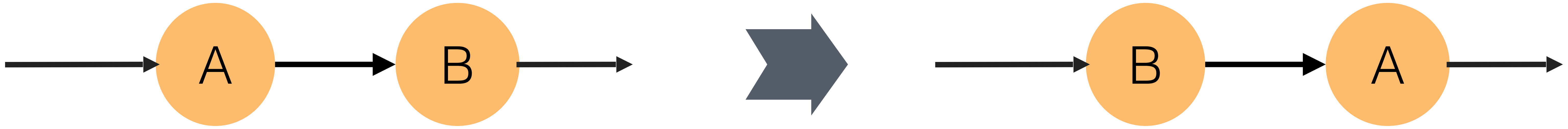


Move selective operators upstream to filter data early

Safety

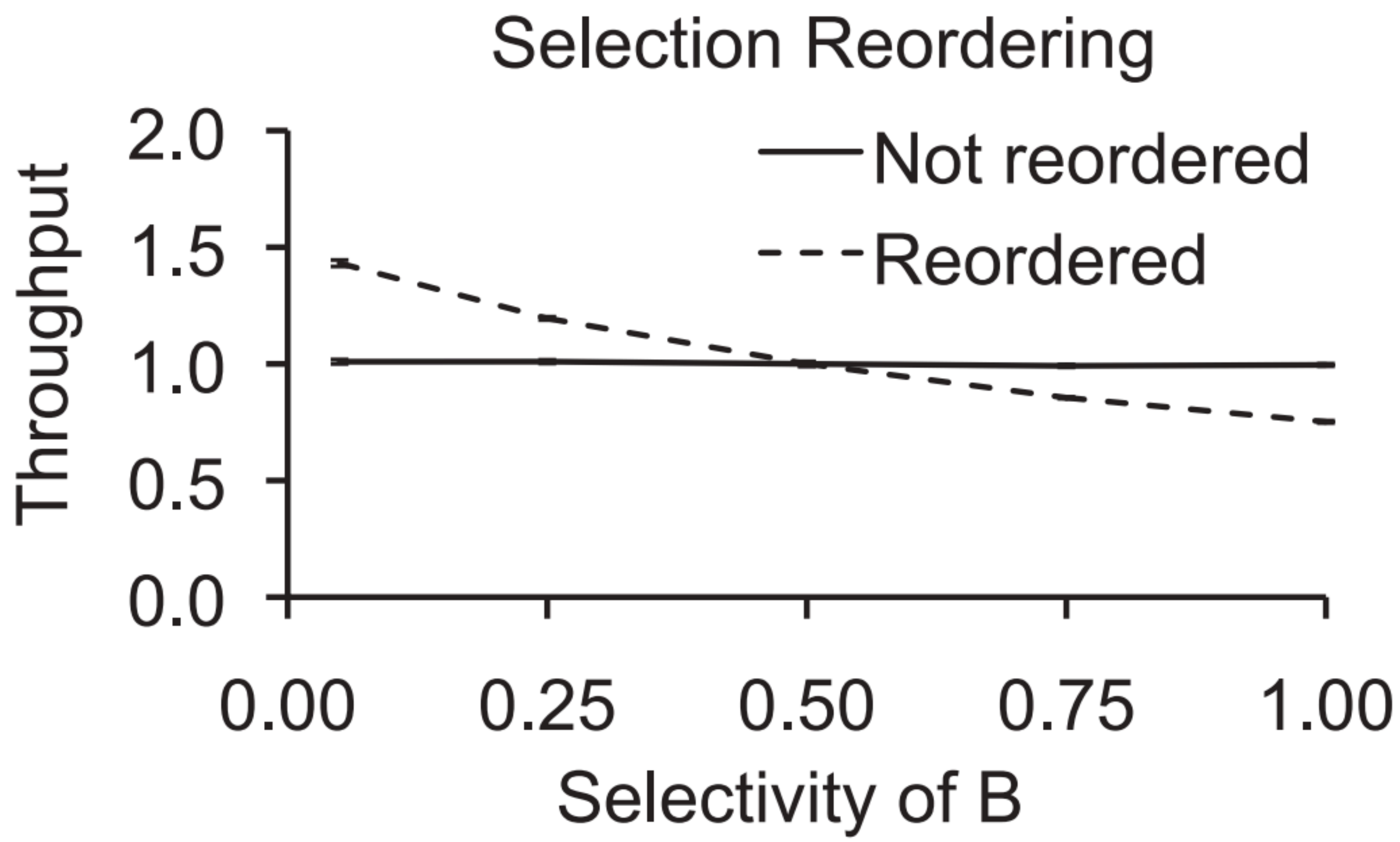
- **Attribute availability**: the set of attributes B reads from must be disjoint from the set of attributes A writes to.
- **Commutativity**: the results of applying A and then B must be the same as the result of applying B and then A.
  - holds if both operators are stateless

Operator re-ordering

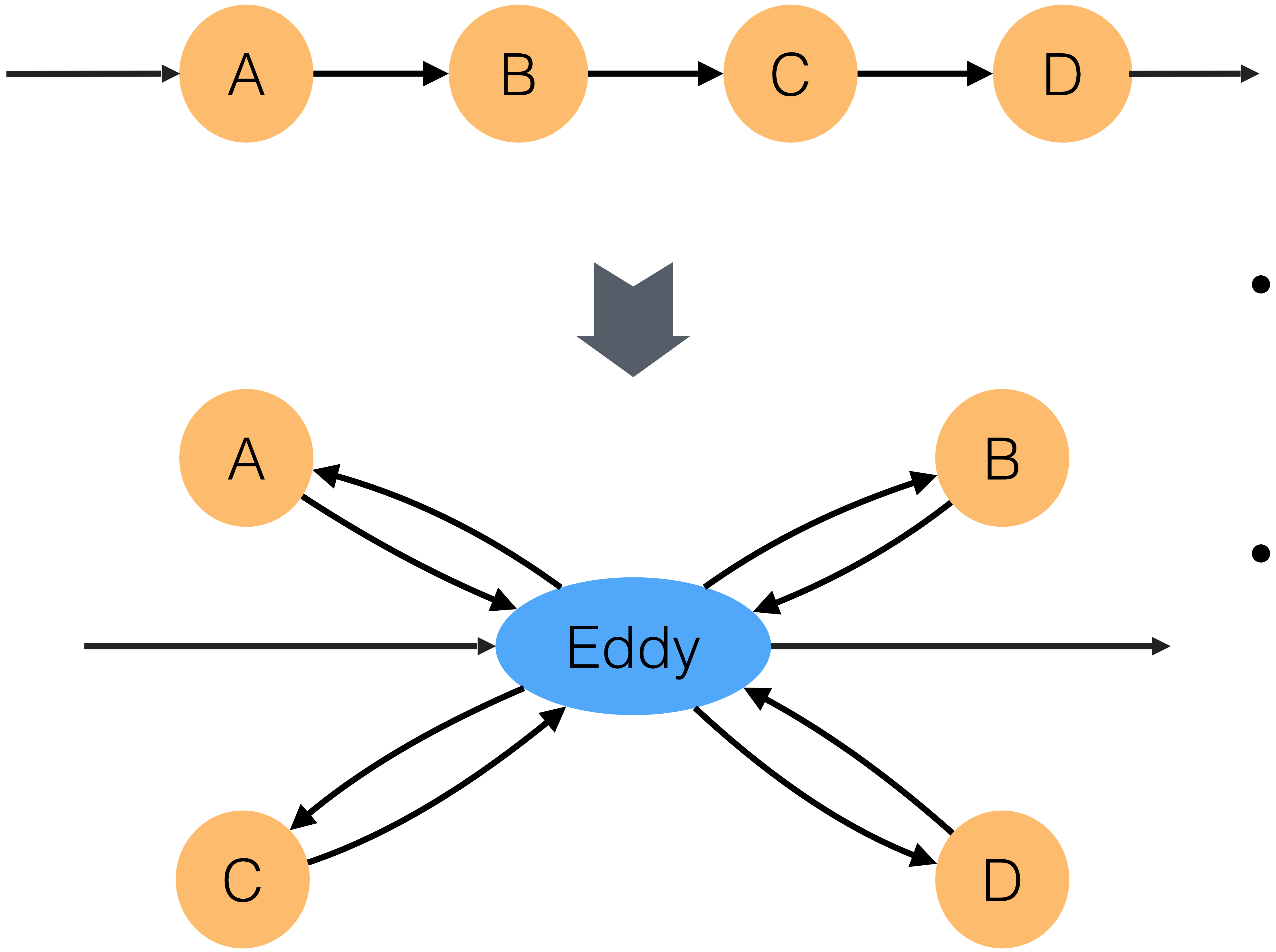


Profitability

- Selectivity of A = 0.5
- Profitable when selectivity of B < 0.5

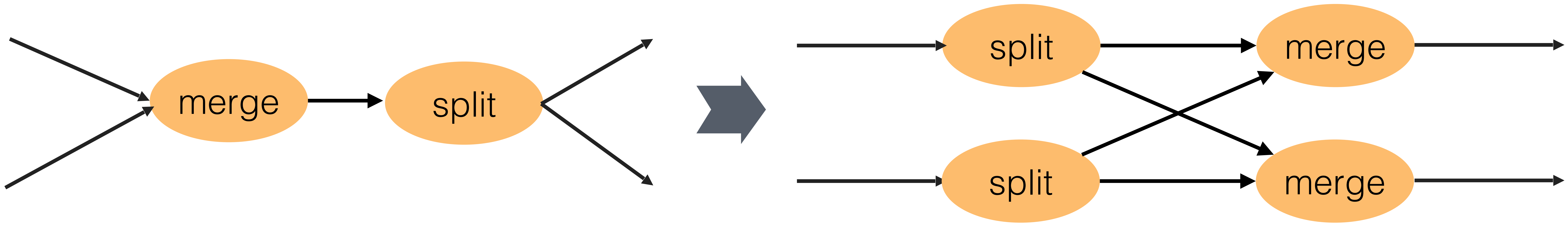


Dynamic re-ordering with Eddy



- A static graph transformation that enables re-ordering at runtime
- It dynamically routes data after measuring which ordering is the most profitable

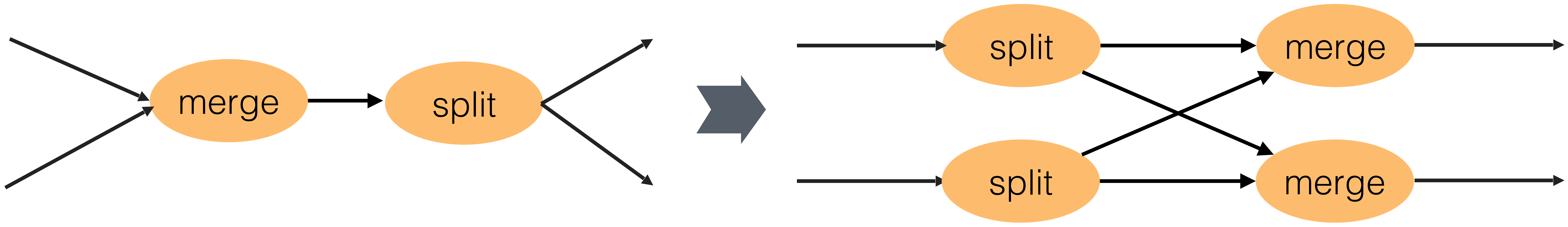
Re-ordering split and merge



Safety

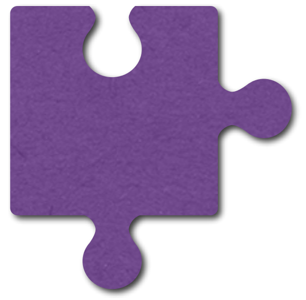
- **attribute availability**: the set of attributes B reads from must be disjoint from the set of attributes A writes to.
- **commutativity**: the results of applying A and then B must be the same as the result of applying B and then A.
  - holds if both operators are stateless

Re-ordering split and merge



Safety

- **attribute availability**: the set of attributes B reads from must be disjoint from the set of attributes A writes to.
- **commutativity**: the results of applying A and then B must be the same as the result of applying B and then A.
  - holds if both operators are stateless

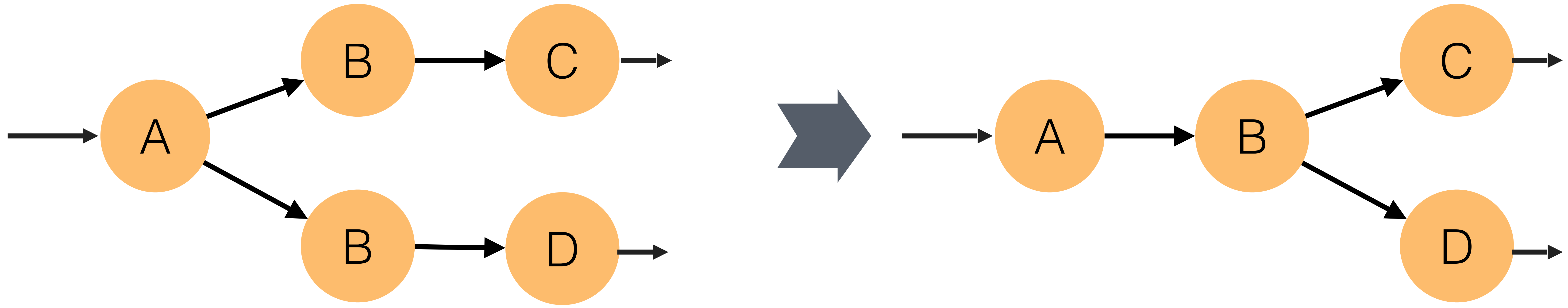


When might this be beneficial?

Algebraic re-orderings

- Use equivalence transformation rules if the language allows
  - selection operations are commutative
  - theta-join operations are commutative
  - natural joins are associative
- Move projections early to reduce data item size
- Pick join orderings to minimize the size of intermediate results
  - execute selective joins first => follow-up joins will have less work to do

Redundancy elimination

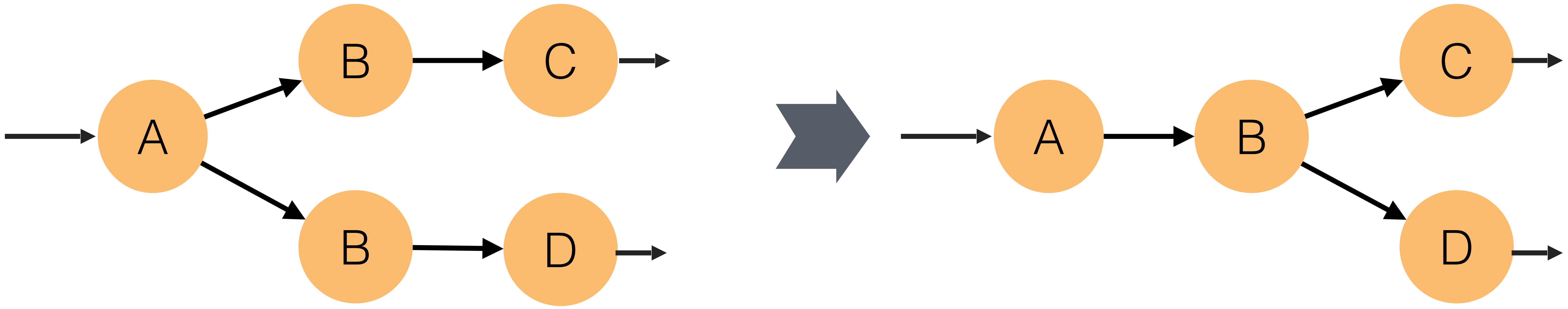


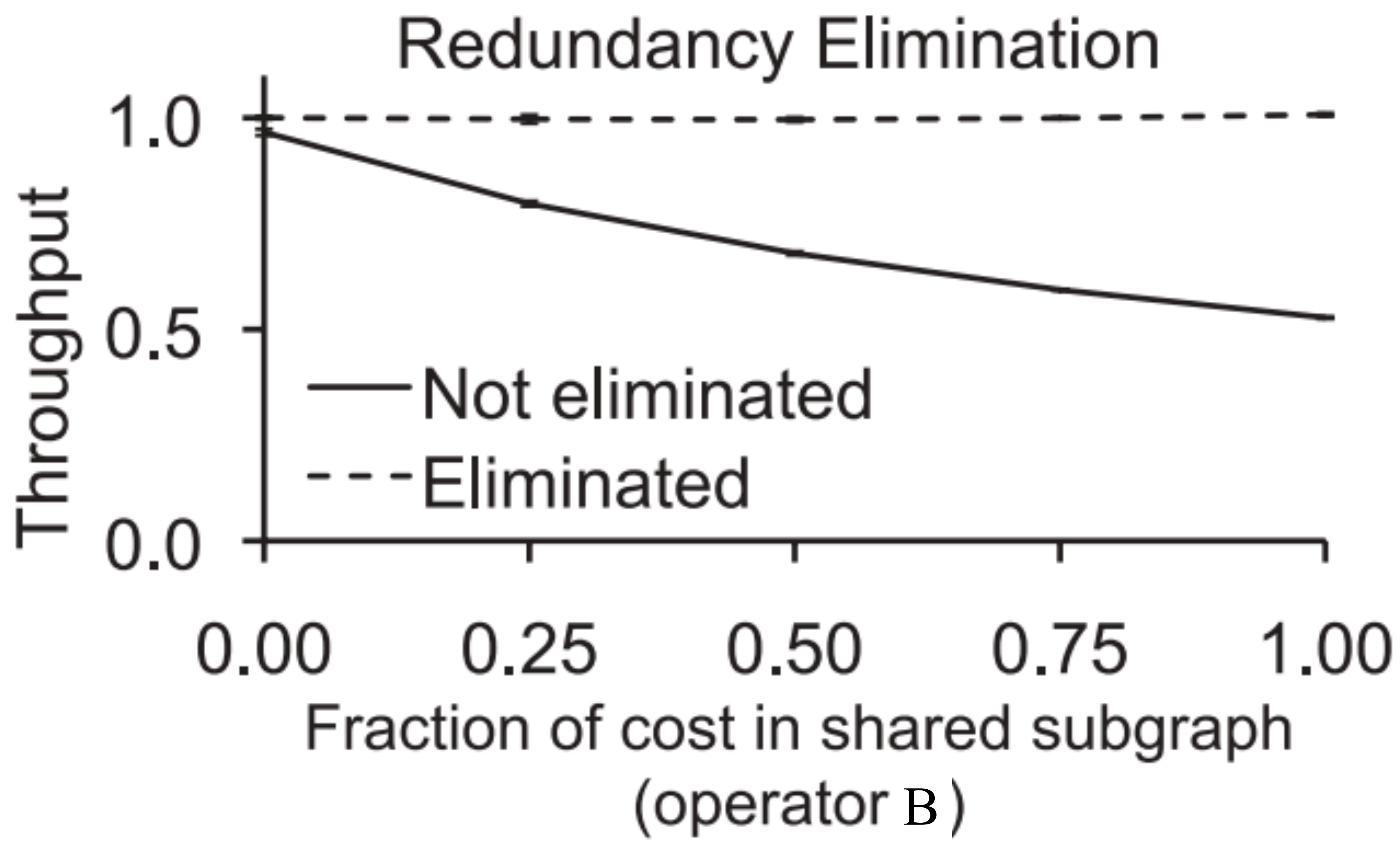
Eliminate redundant operations, aka subgraph sharing

Safety

- **Ensure same algorithm**: the redundant operators must perform an equivalent computation
- **Ensure mergeable state**: even a simple counter might differ on a combined stream vs. on separate streams

Redundancy elimination





Profitability

- Running two applications together on a single core, one with operators B and C, the other with operators B and D.

Redundancy elimination variations

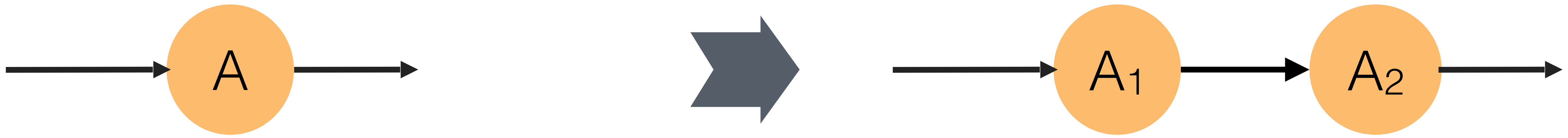
- Multi-tenancy
  - in streaming systems that build one dataflow graph for several queries
  - when applications analyze data streams from a small set of sources
- Operator elimination
  - remove a no-op, e.g. a projection that keeps all attributes
  - remove idempotent operations, e.g. two selections on the same predicate
  - remove a dead subgraph, i.e. one that never produces output

Redundancy elimination variations

- Multi-tenancy
  - in streaming systems that build one dataflow graph for several queries
  - when applications analyze data streams from a small set of sources
- Operator elimination
  - remove a no-op, e.g. a projection that keeps all attributes
  - remove idempotent operations, e.g. two selections on the same predicate
  - remove a dead subgraph, i.e. one that never produces output

How can no-op or idempotent operators appear in an application?

Operator separation



Separate operators into smaller computational steps

Safety

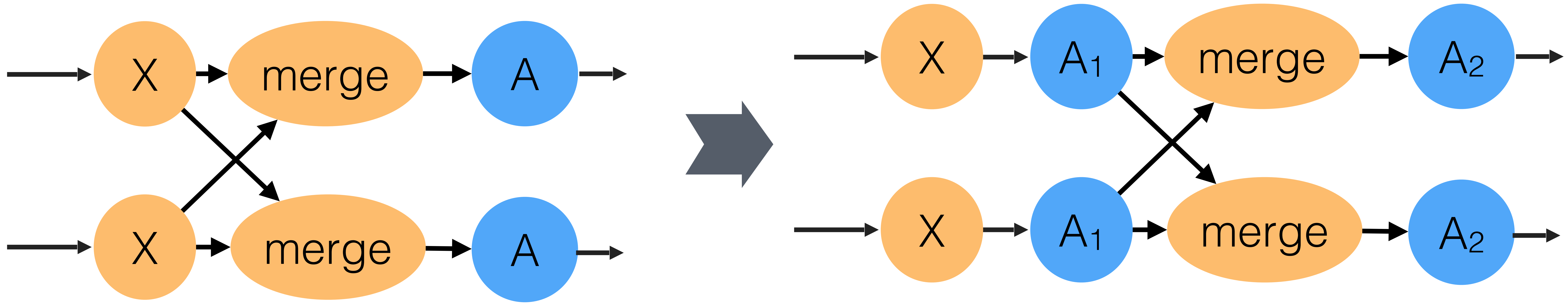
Ensure the combination of A_1, A_2 is equivalent to A: Given a stream s, make sure $A_2(A_1(s)) = A(s)$, e.g.,

- if A is a selection operator and the selection predicate uses logical conjunction
- if A is a projection on multiple attributes
- if A is an idempotent aggregation

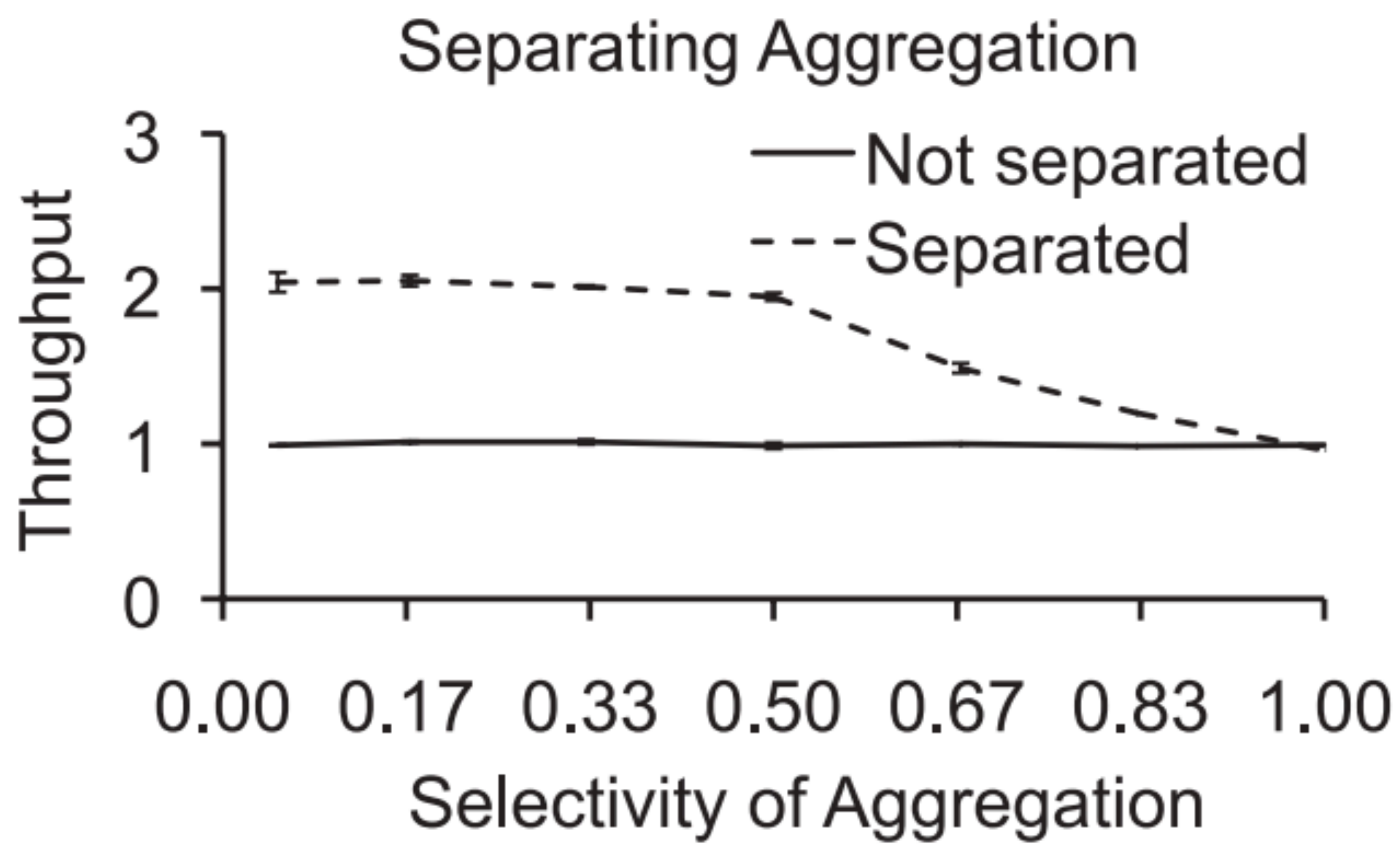
Profitability

- beneficial if it enables other optimizations, e.g. re-ordering
- if the pipeline parallelism pays off

Operator separation

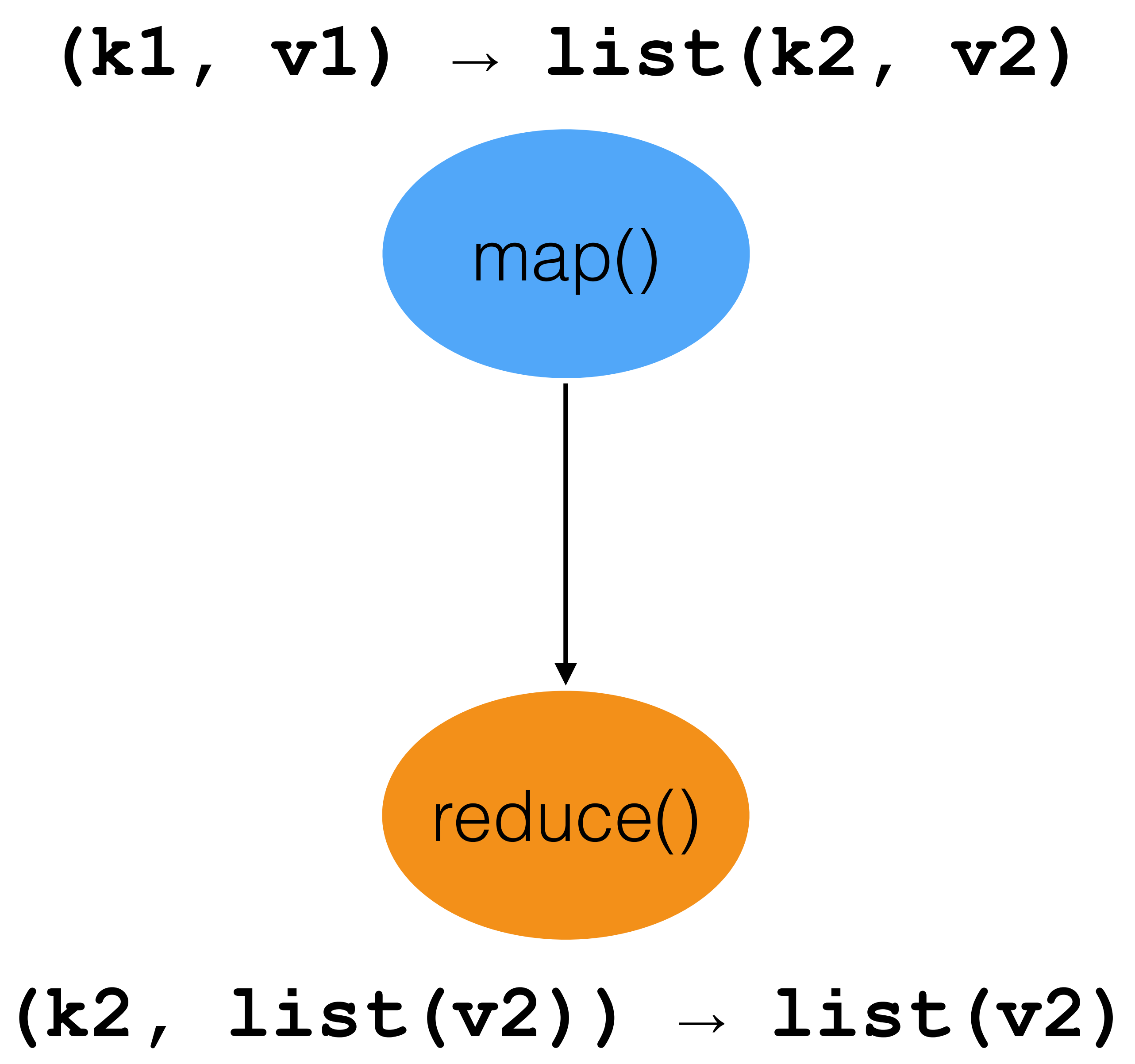


- Cost of Merge = 0.5
- Cost of A = 0.5
- Splitting A allows a pre-aggregation similar to what combiners do in MapReduce

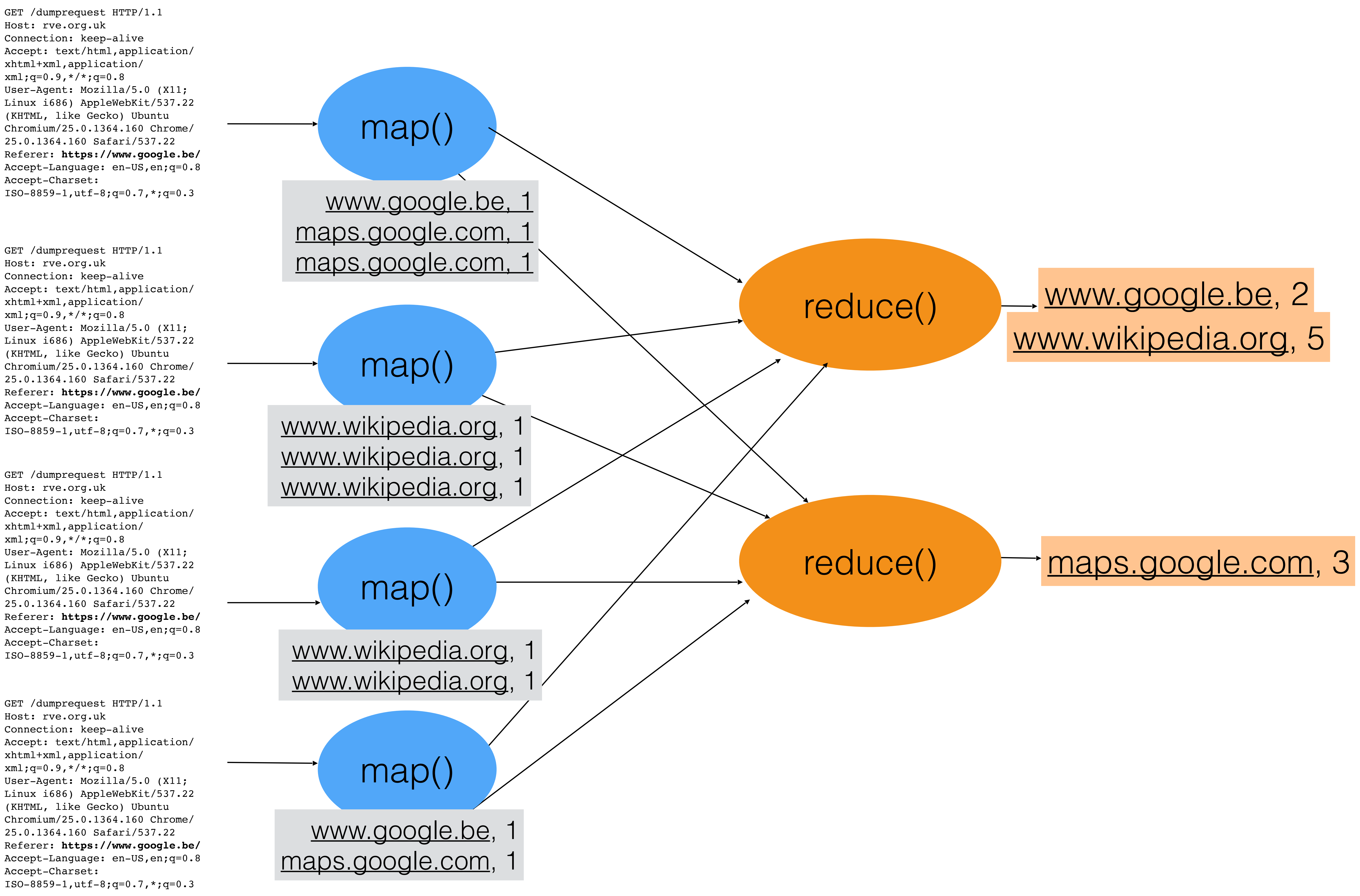


MapReduce combiners example: URL access frequency

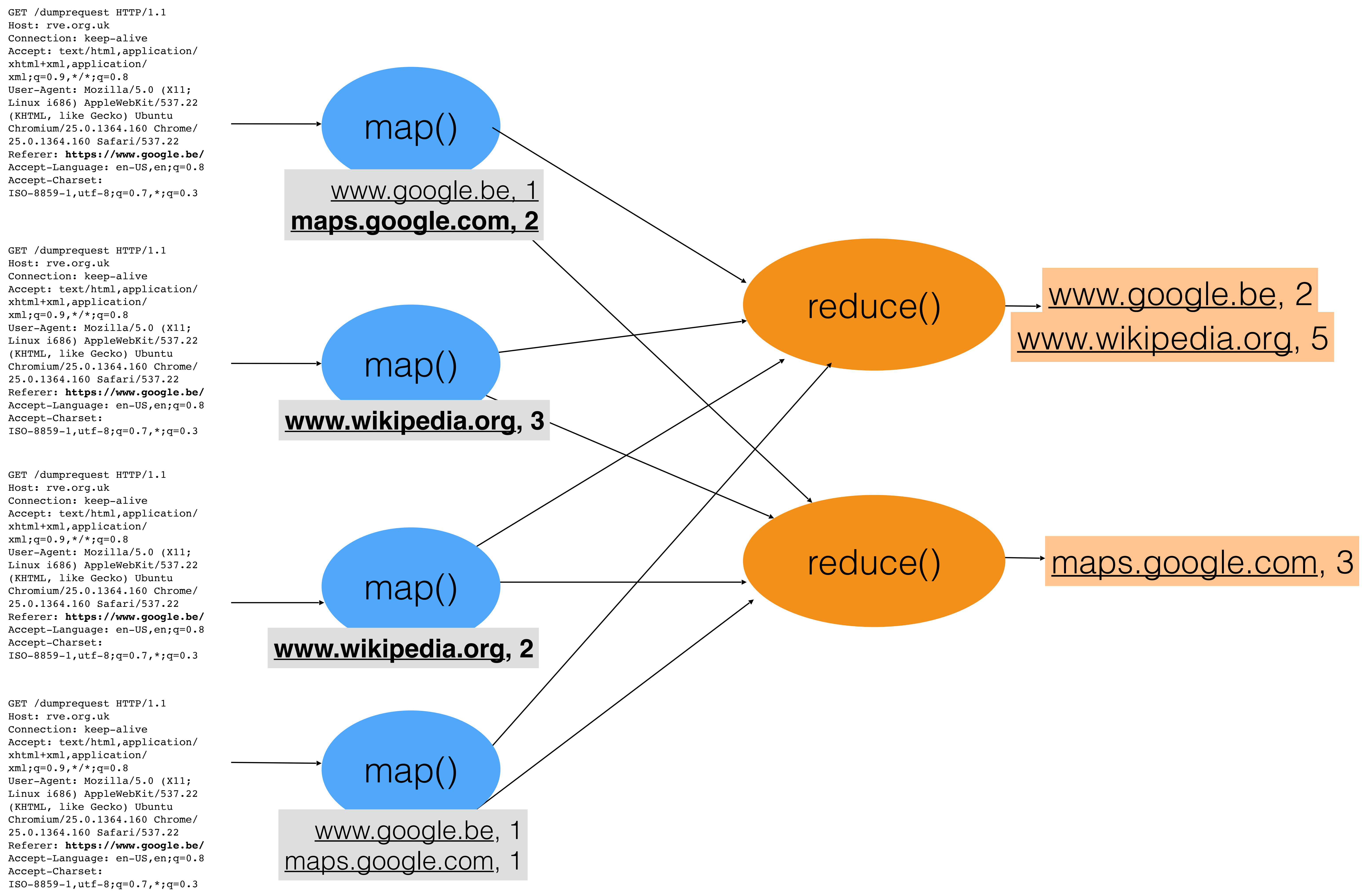
```
map(String key, String value):
  // key: document name
  // value: document contents
    for each URL u in value:
        EmitIntermediate(u, "1");

reduce(String key, Iterator values):
  // key: a URL
  // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
        Emit(key, AsString(result));
```



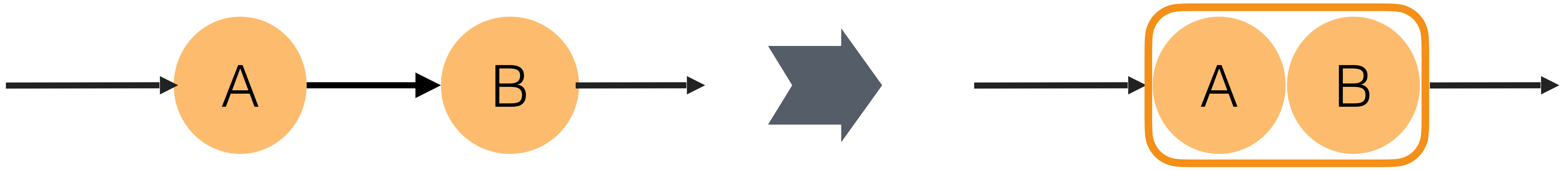
MapReduce combiners example: URL access frequency



MapReduce combiners example: URL access frequency



Operator fusion

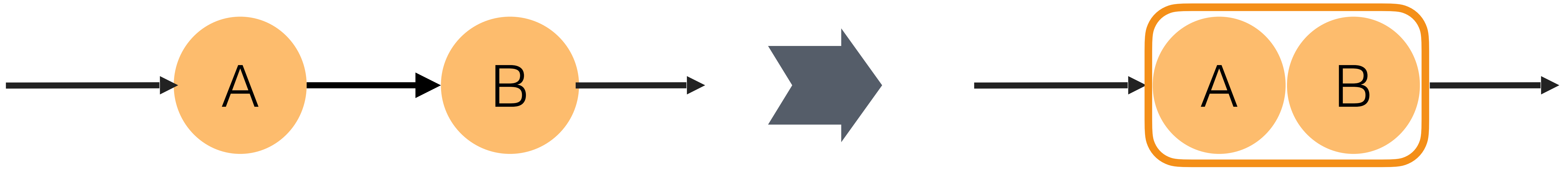


Avoid the overhead of serialization and transport

Safety

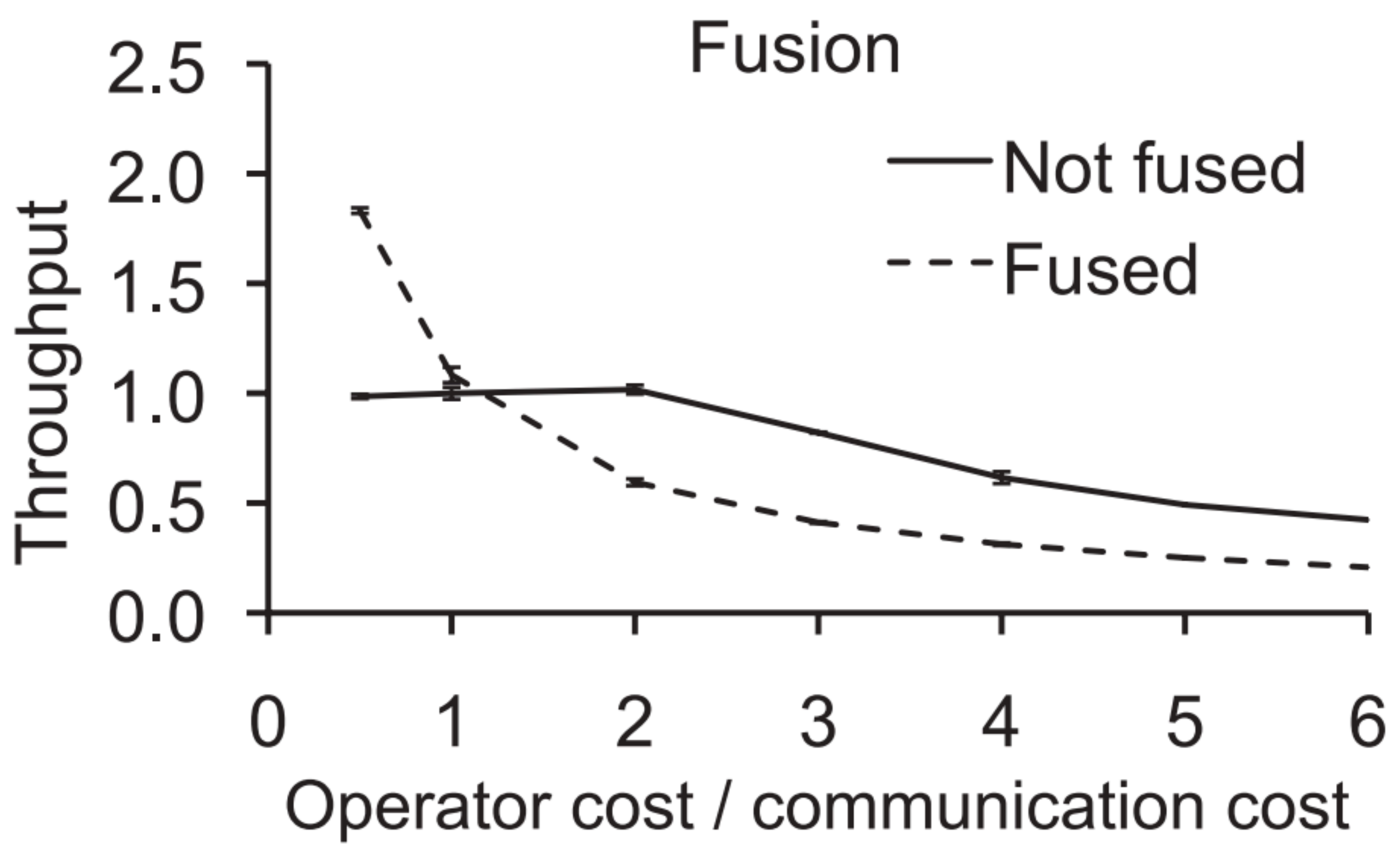
- **Ensure resource kinds**: all resources required by a fused operator should remain available.
- **Ensure resource amounts**: the total amount of resources required by the fused operator must be available on a single host.
- **Avoid infinite recursion**: caution if there exist cycles in the stream graph.

Operator fusion



Profitability

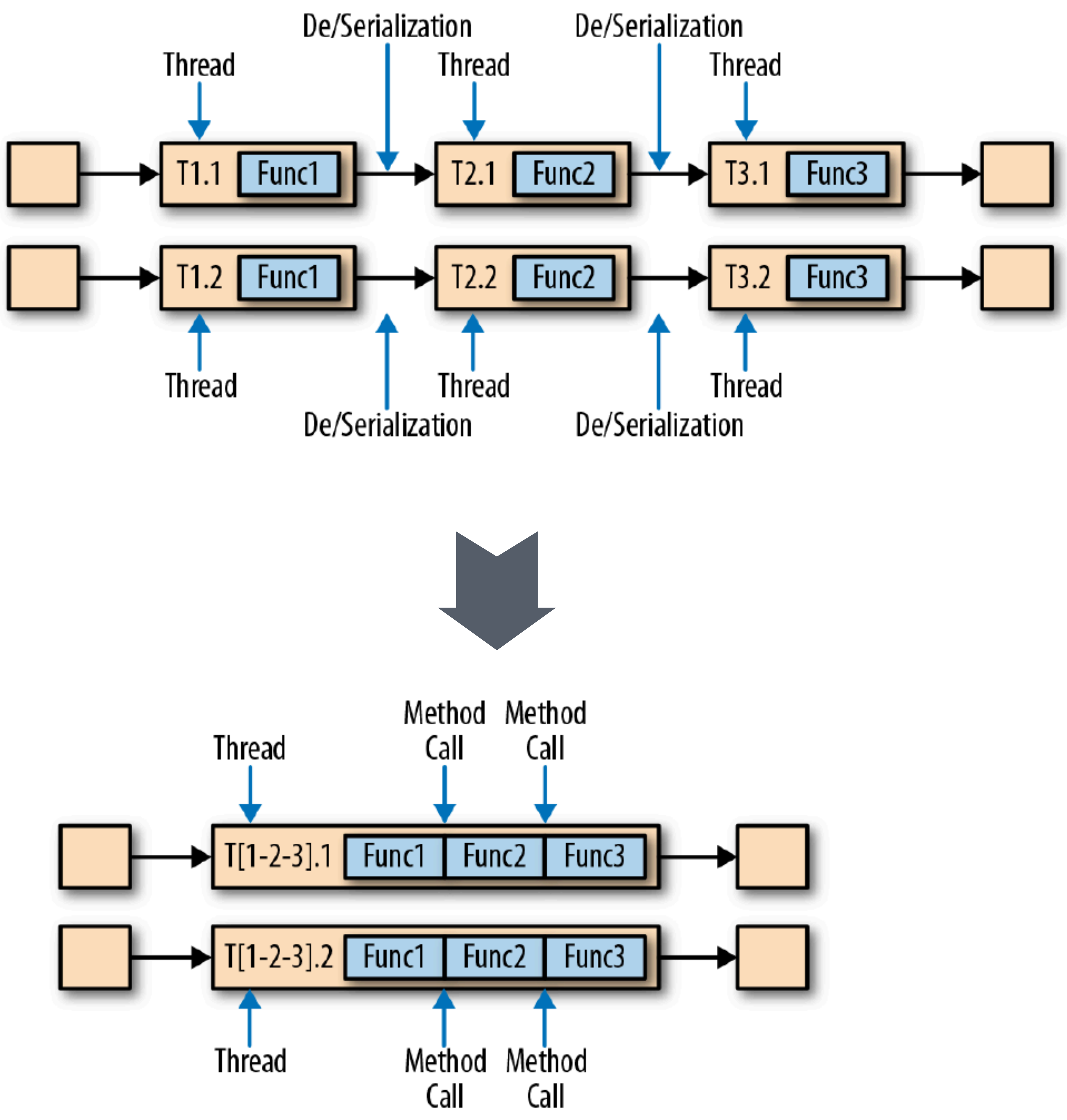
- removes pipeline parallelism but saves communication and serialization cost
- if operators are separate, throughput is bounded by either communication or processing cost
- if fused, throughput is determined by operator cost only



Synergies with scheduling and other optimizations

- Non-fused operators can run on different threads
- The optimizer can interact with the scheduler and fuse operators according to the number of available cores / threads
- Fused operators can share the address space but use separate threads of control
  - avoid communication cost without losing pipeline parallelism
  - use a shared buffer for communication
- Fused filters / projections at the source can significantly reduce I/O and intermediate results size

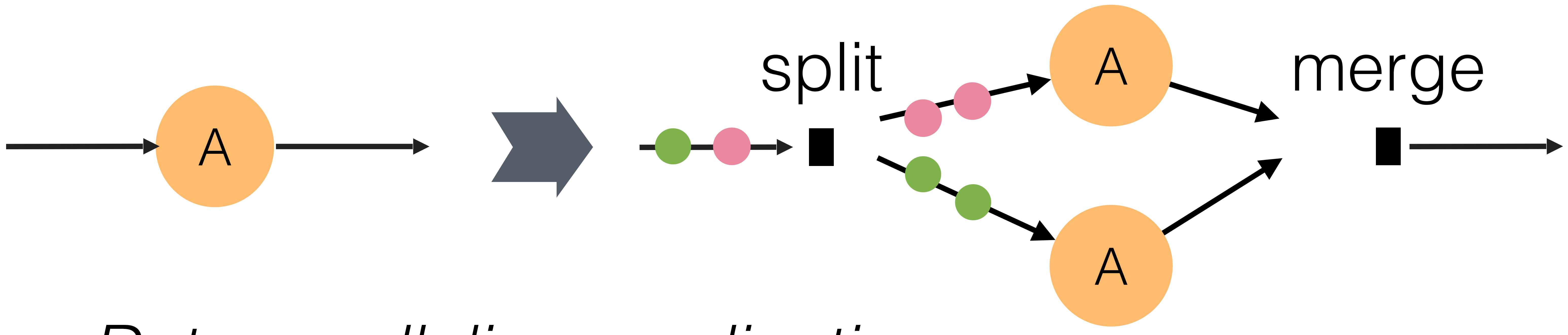
Task chaining: Fusion in Flink




```
StreamExecutionEnvironment
   .disableOperatorChaining()
```

```
val input: DataStream[X] = ...
val result: DataStream[Y] = input
 .filter(new Filter1())
 .map(new Map1())
 // disable chaining for Map2
 .map(new Map2()).disableChaining()
 .filter(new Filter2())
```

Operator fission

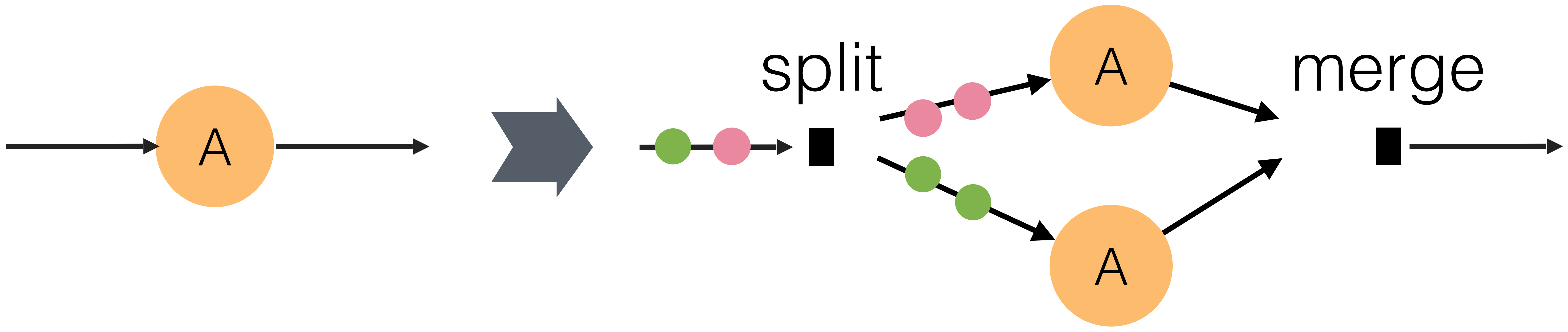


Data parallelism, replication

Safety

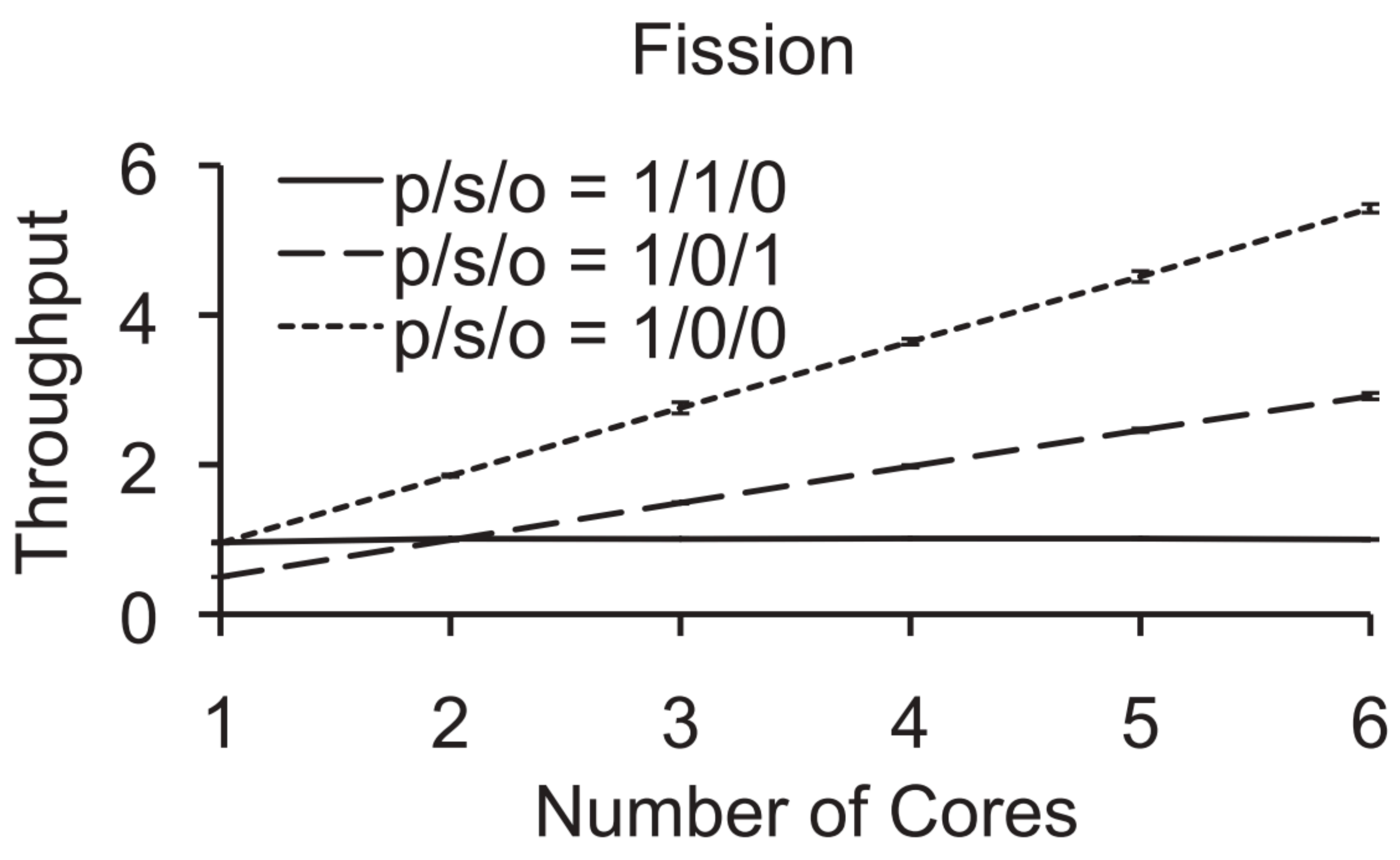
- **Ensure partitioned state**: each parallel operator maintains disjoint state based on a key attribute
- **Ensure ordering constraints**: if downstream operator expects elements in a particular order, merging should handle that
- **Avoid deadlocks**: if split cannot push data because one channel is full and merge cannot receive data because another channel is empty

Operator fission



Profitability

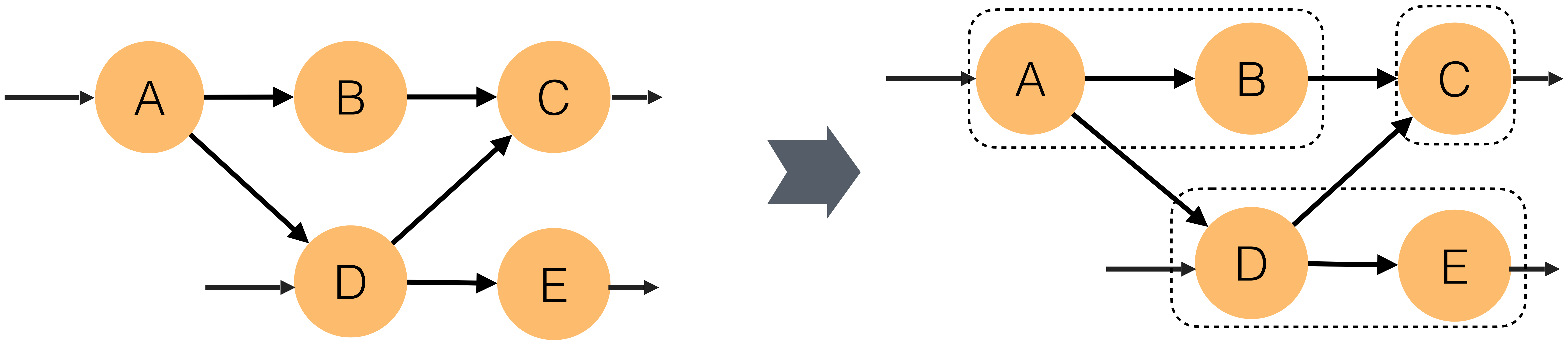
- if operator is costly enough to bring benefit when parallelized
- split incurs a routing overhead
- merge might incur overhead if ordering is required
- p/s/o: parallel/sequential/overhead



Variations and dynamism

- Fission might be preferable to pipeline and task parallelism because it balances load more evenly
- Data-parallel streaming languages enable fission by construction
- Elastic *scaling* techniques enable dynamic operator fission by adjusting the number of parallel operator instances according to data rates
  - straight-forward for stateless operators, non-trivial for stateful

Operator placement

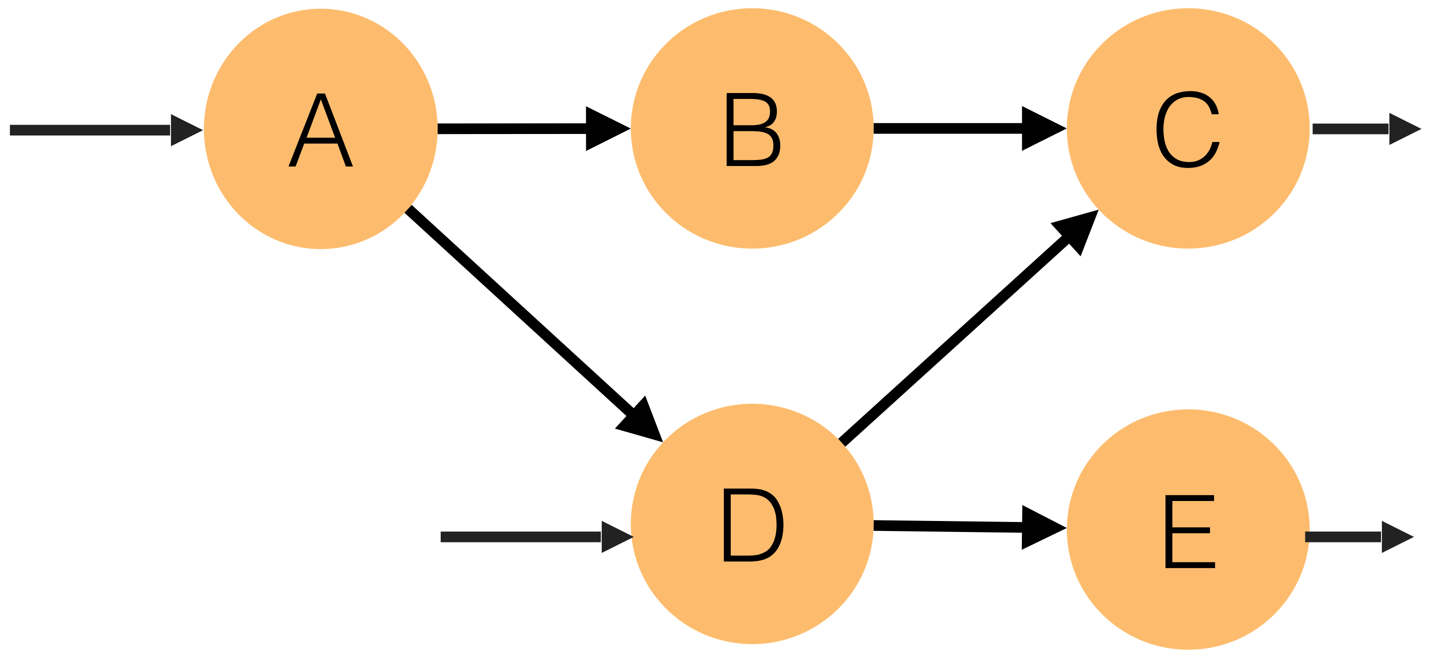


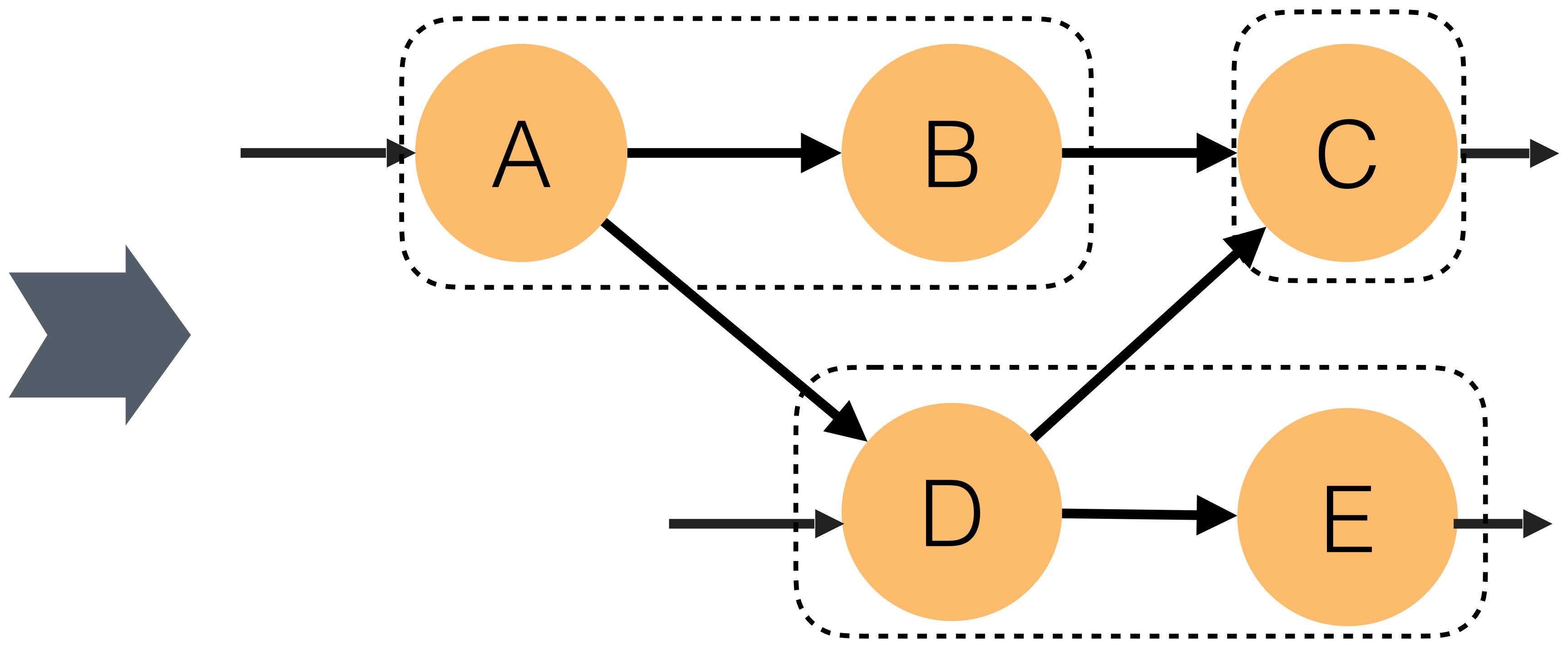
Assignment to hosts, colocation

Safety

- **Ensure resource availability**: the host must have enough resources for all assigned operators
- **Ensure security constraints**: what are the trusted hosts for each operator?
- **Ensure state migration**: if placement is dynamic and the operator is stateful, its state must be moved in a consistent manner

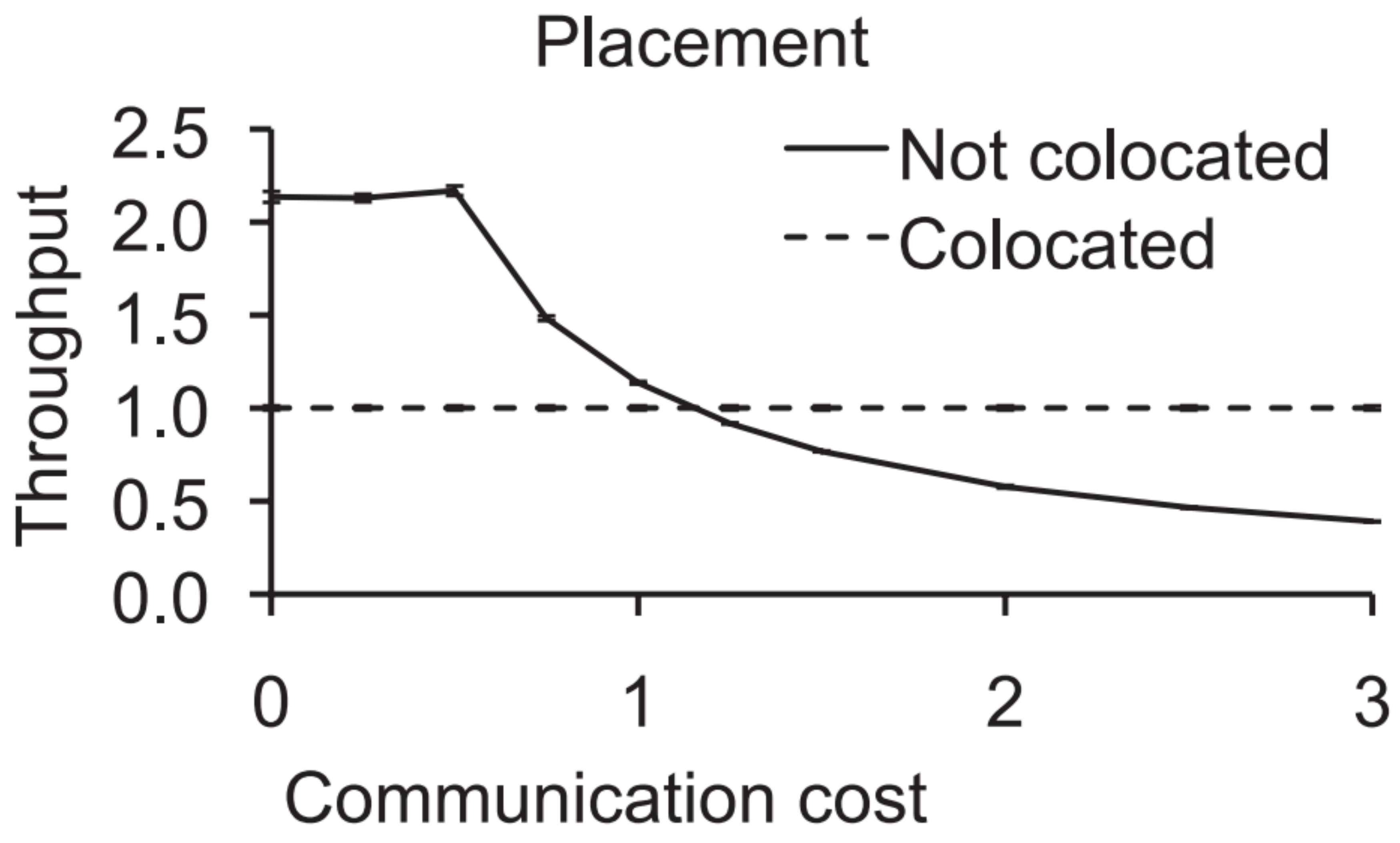
Operator placement



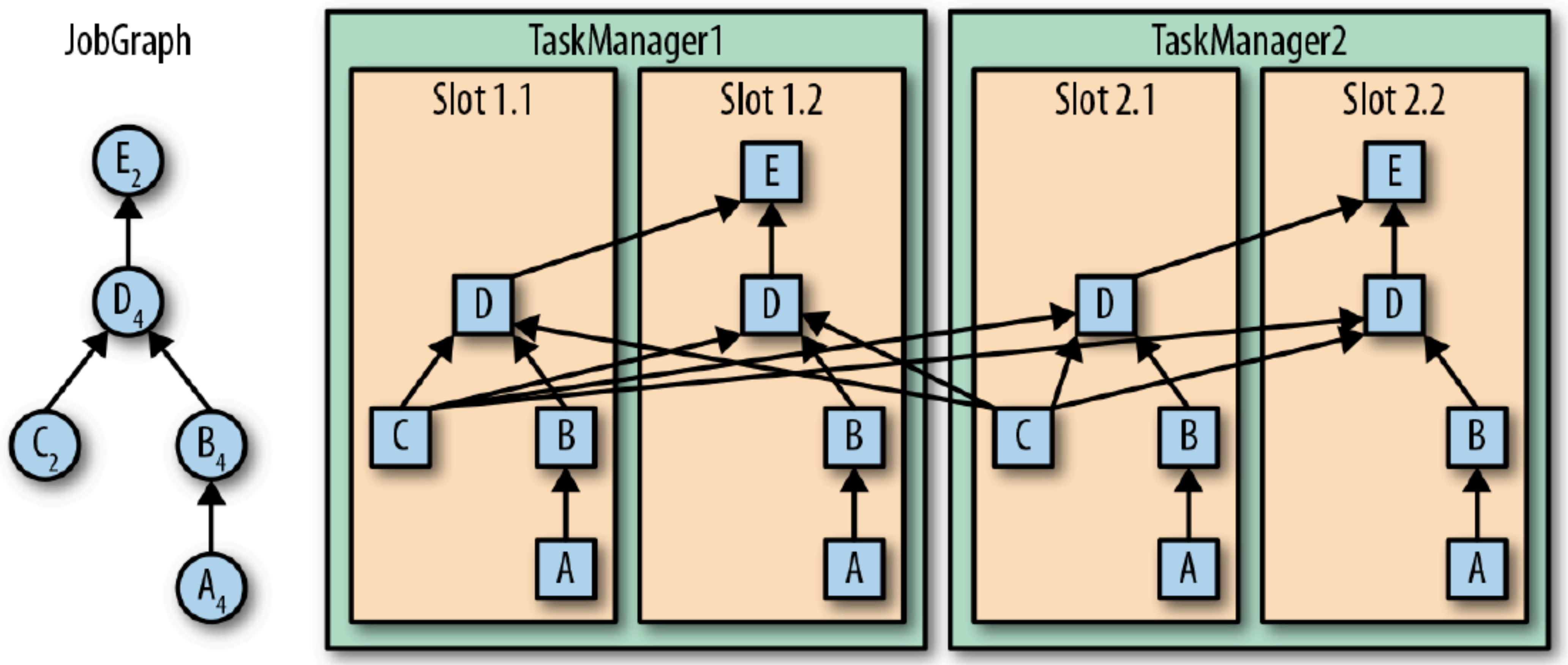


Profitability

- Trade communication cost against resource utilization
- Operators on the same host compete for resources, e.g. memory and CPU

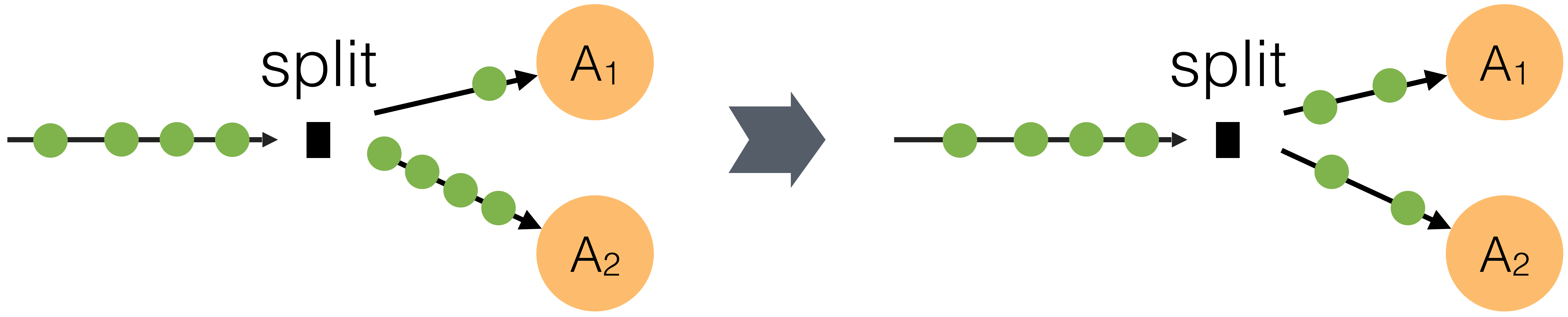


Operator placement in Flink



- A TaskManager can execute several tasks at the same time.
- It is statically configured with a certain number of **processing slots** that defines the maximum number of concurrent tasks it can execute.
- A processing slot can execute one **slice** of an application, i.e. one parallel task of each operator of the application.

Load balancing

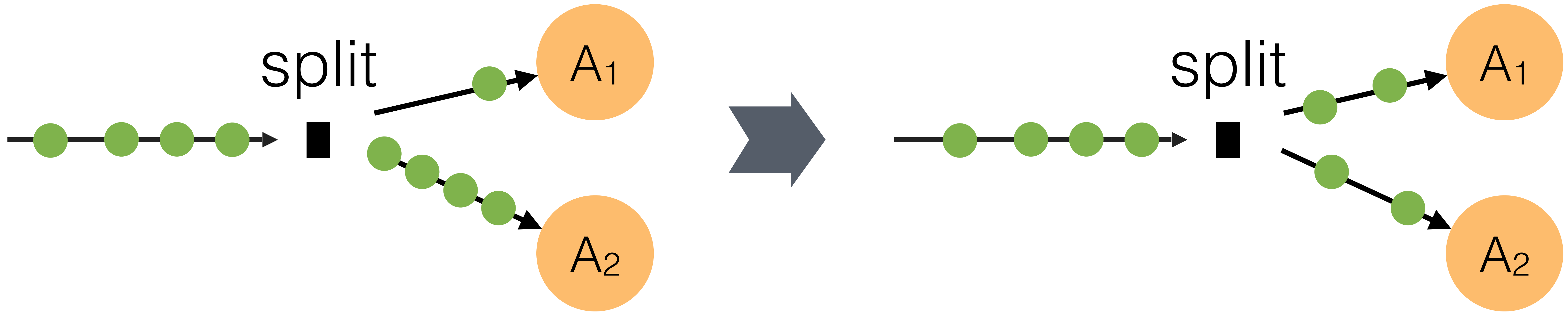


Distribute workload evenly across resources

Safety

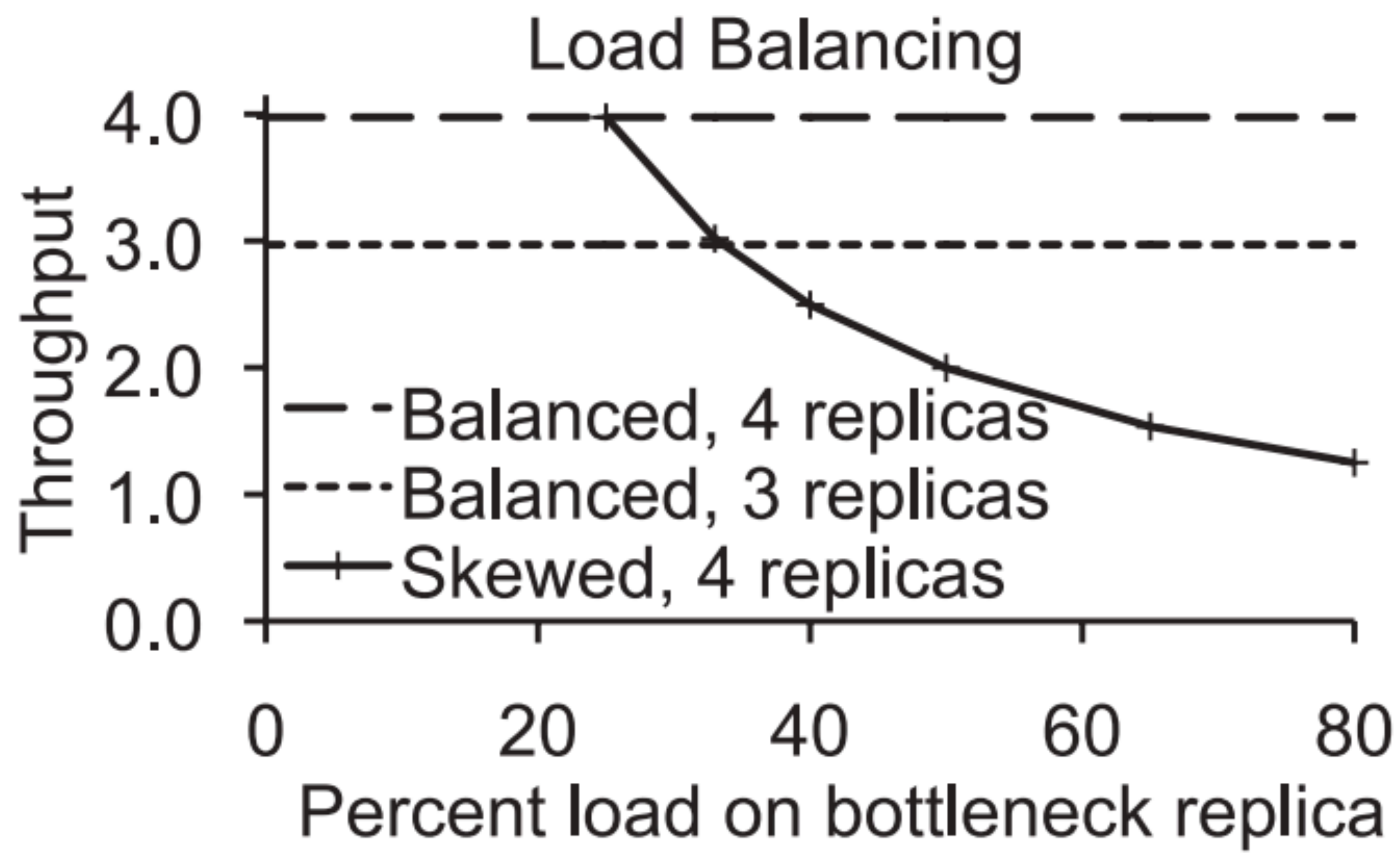
- **Avoid starvation**: every data item is eventually processed
- **Ensure each worker is qualified**: if load balancing is applied after fission, each instance must be capable of processing each item and have access to necessary state
- **Establish placement safety**: if load balancing while performing operator placement

Load balancing

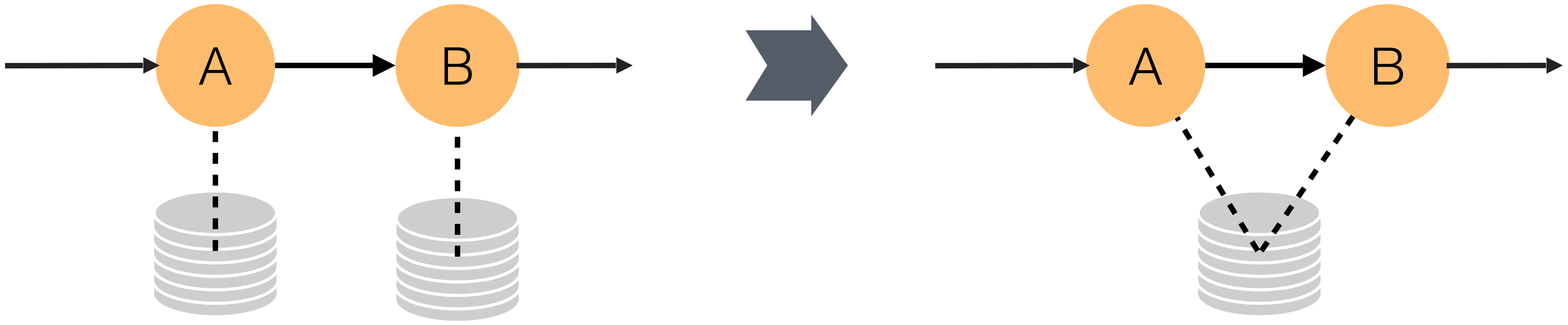


Profitability

- If it compensates for skew, e.g. when there exist popular keys
- if there is skew, throughput is bounded by the instance that receives the highest load



State sharing

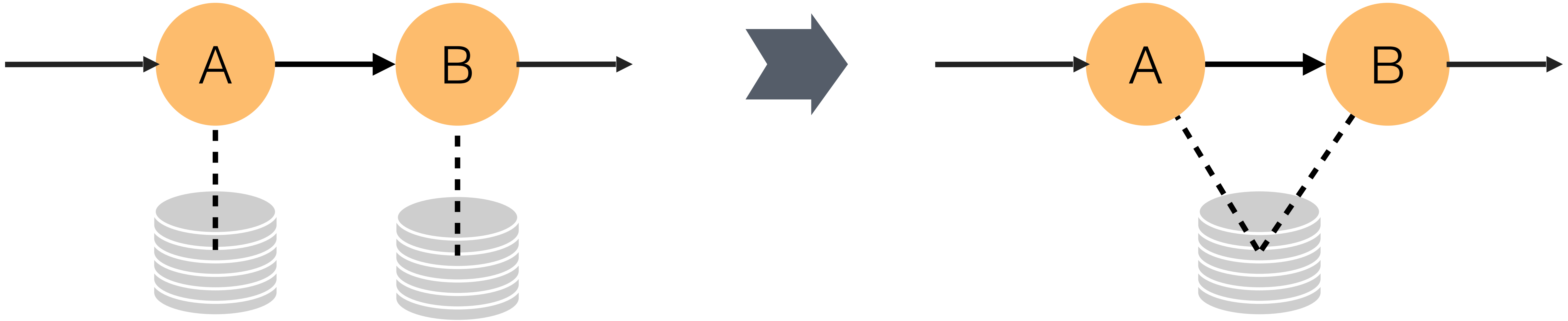


Avoid unnecessary data copies

Safety

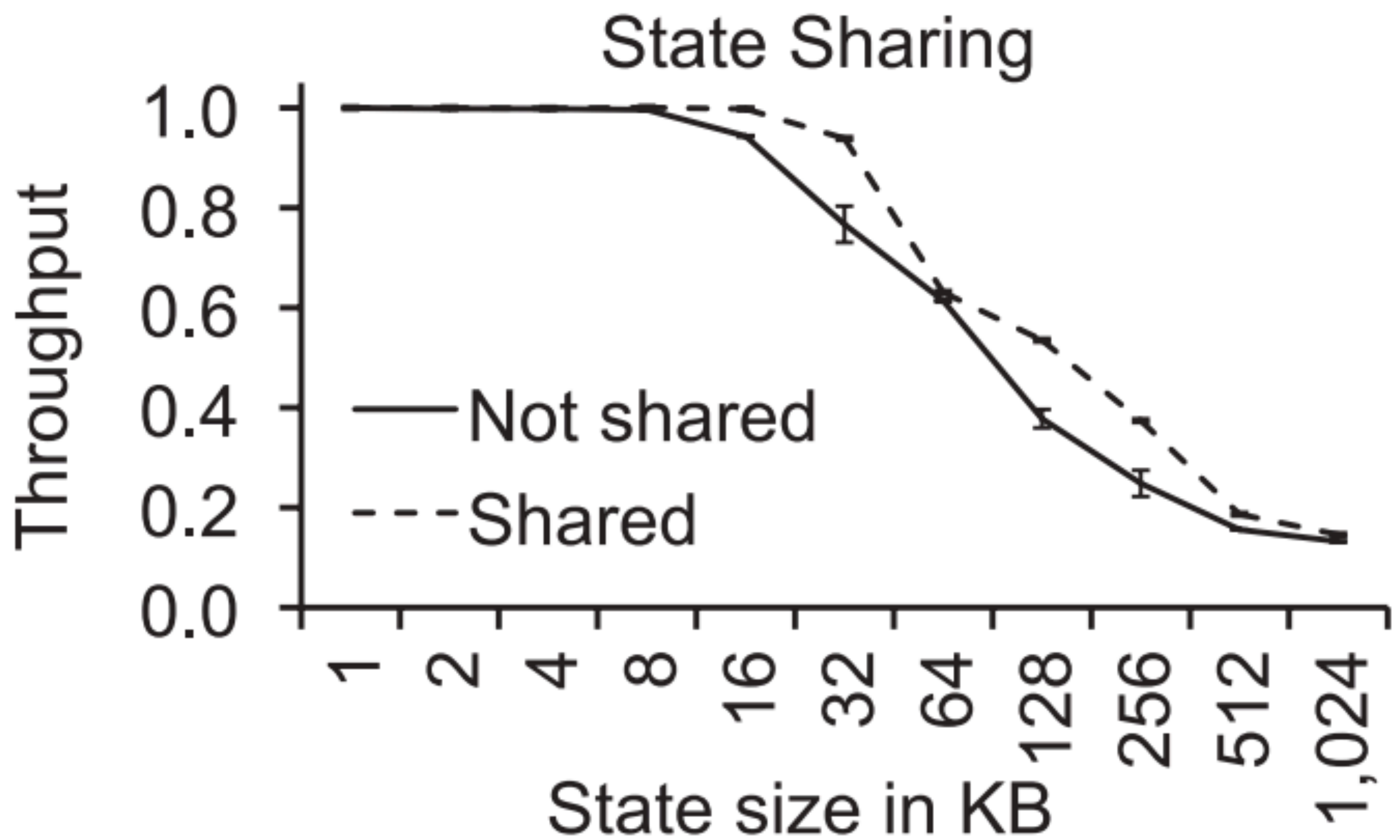
- **Ensure state visibility**: operators sharing state are commonly fused and placed on the same host.
- **Avoid race conditions**: either ensure the data is immutable or synchronize access to state.
- **Manage memory safely**: reclaiming and growing without bounds.

State sharing

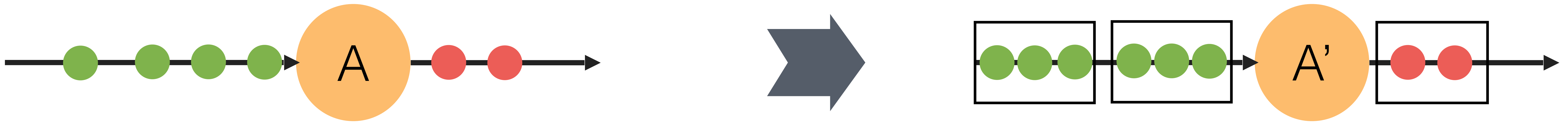


Profitability

- it reduces stalls due to cache misses or disk I/O
- fixed number of random state accesses, 32K L1 cache
- the throughput of the non-shared version degrades first



Batching

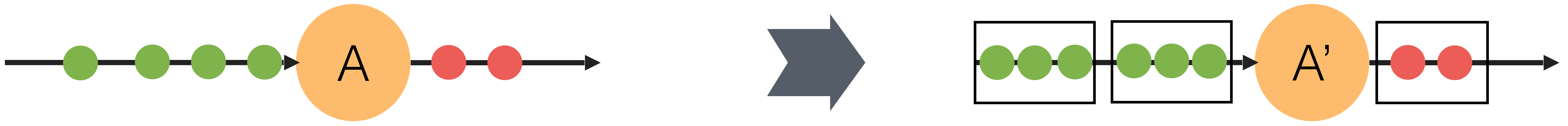


Process multiple data elements in a single batch

Safety

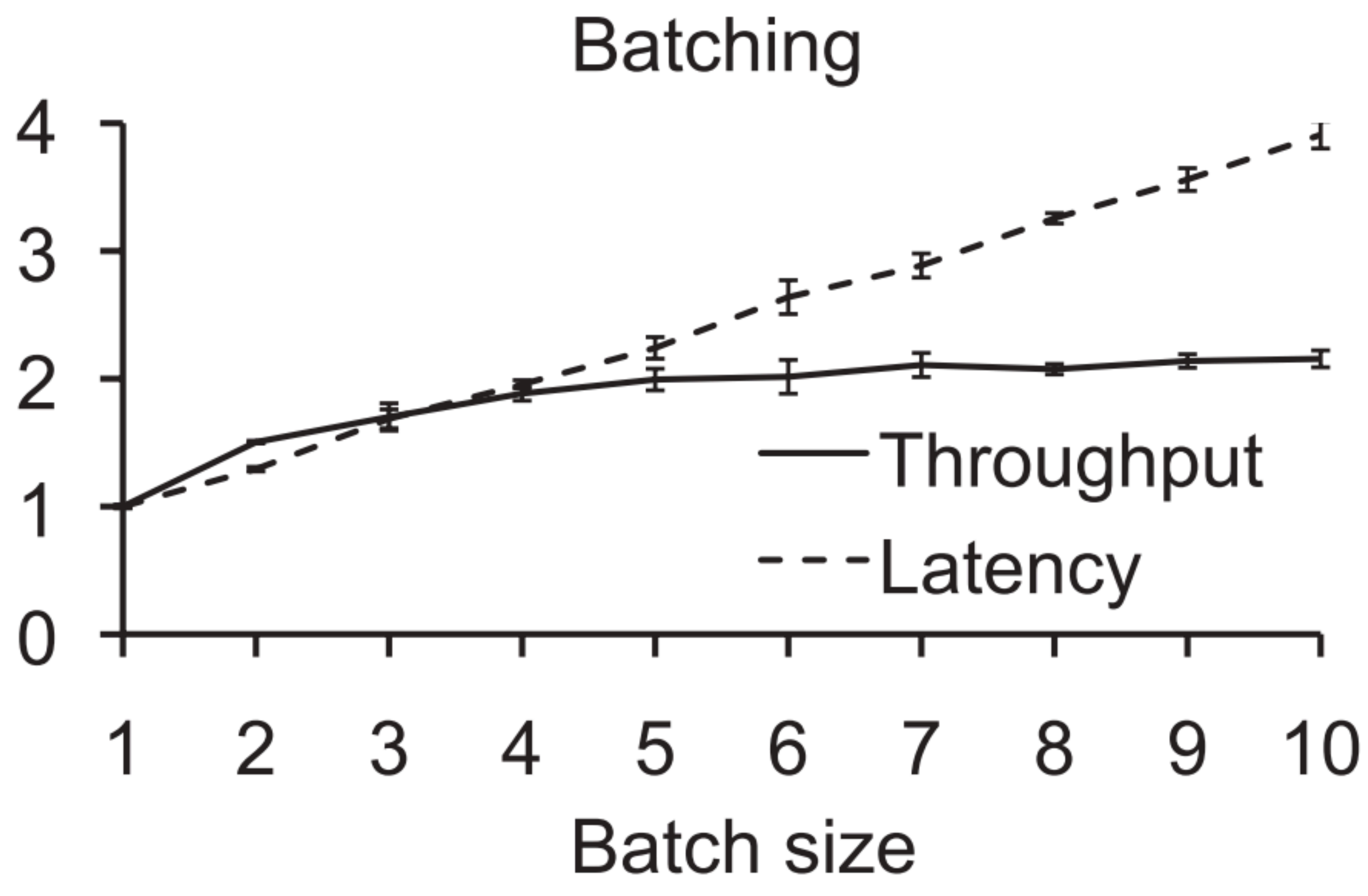
- **Avoid deadlocks**: if the dataflow graph is cyclic or if the batched operator shares a lock with an upstream operator.
- **Satisfy deadlines**: for applications with real-time constraints or QoS latency constraints.

Batching



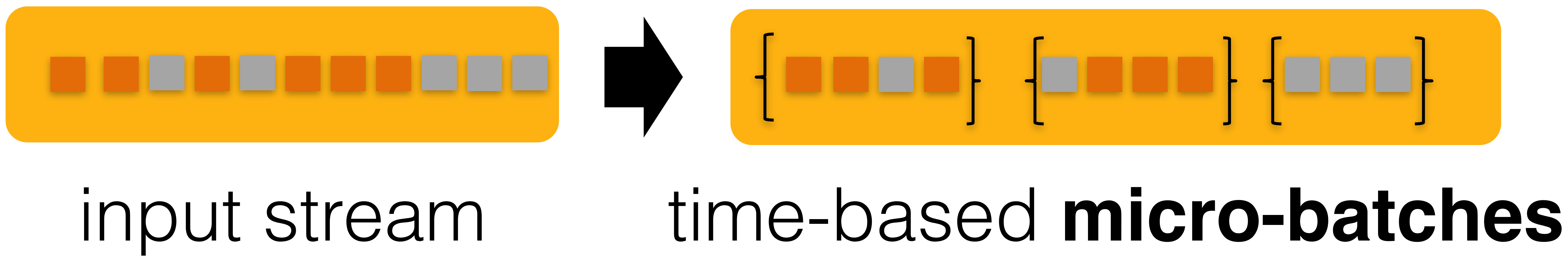
Profitability

- Batching trades throughput for latency
- It improves throughput by amortizing operator firing and communication costs over more data items
- Batching hurts latency as events can only be processed once the entire batch is complete



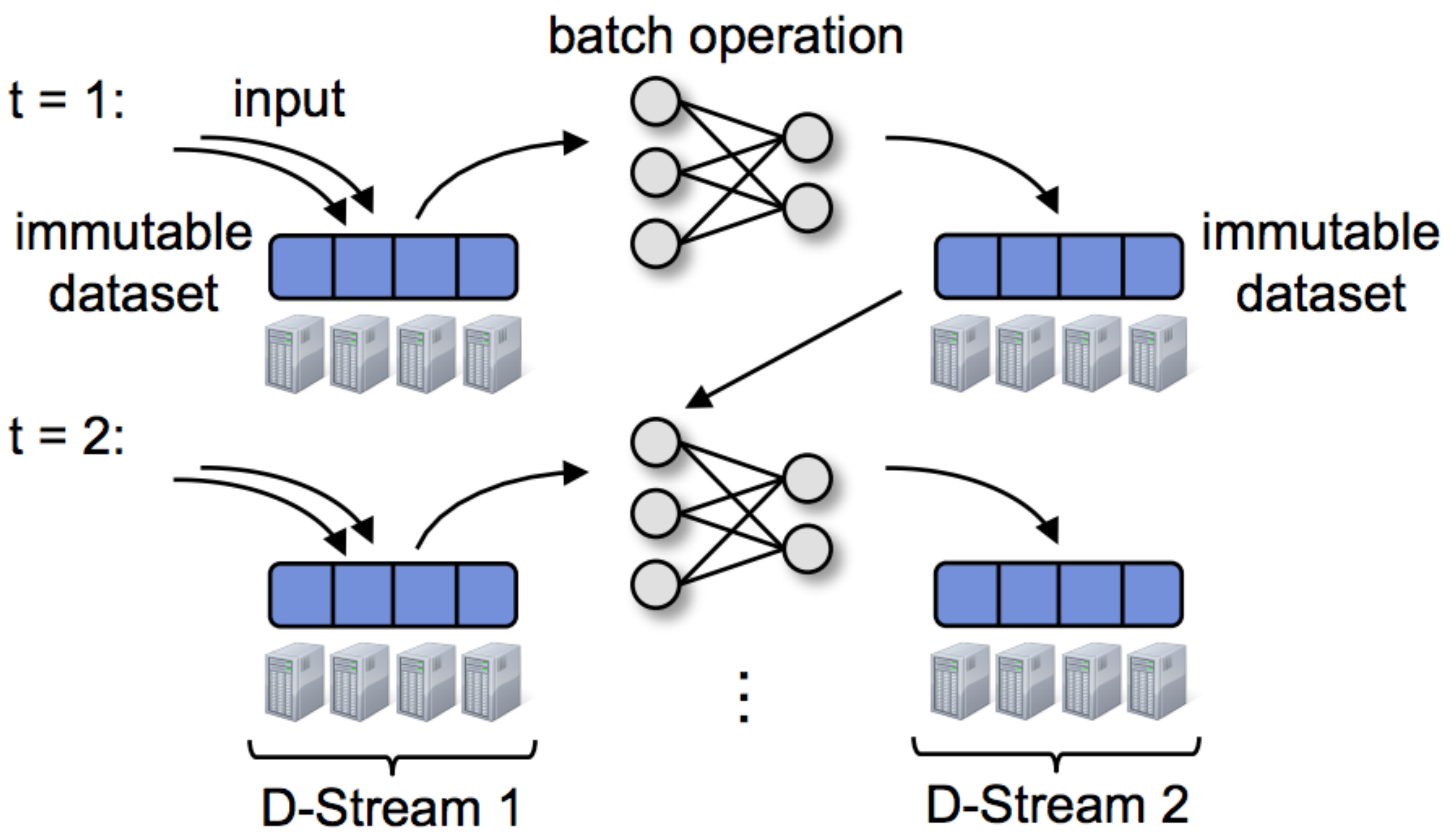
Spark Streaming

- Treat streaming computation as **a series of deterministic batch computations on small time intervals**
- Keep intermediate state **in memory**
- Use Spark's **RDDs** instead of replication
- Parallel recovery mechanism in case of failures



D-Streams

- During an ***interval***, input data received is stored using ***RDDs***
- A ***D-Stream*** is a group of such RDDs which can be processed using common operators

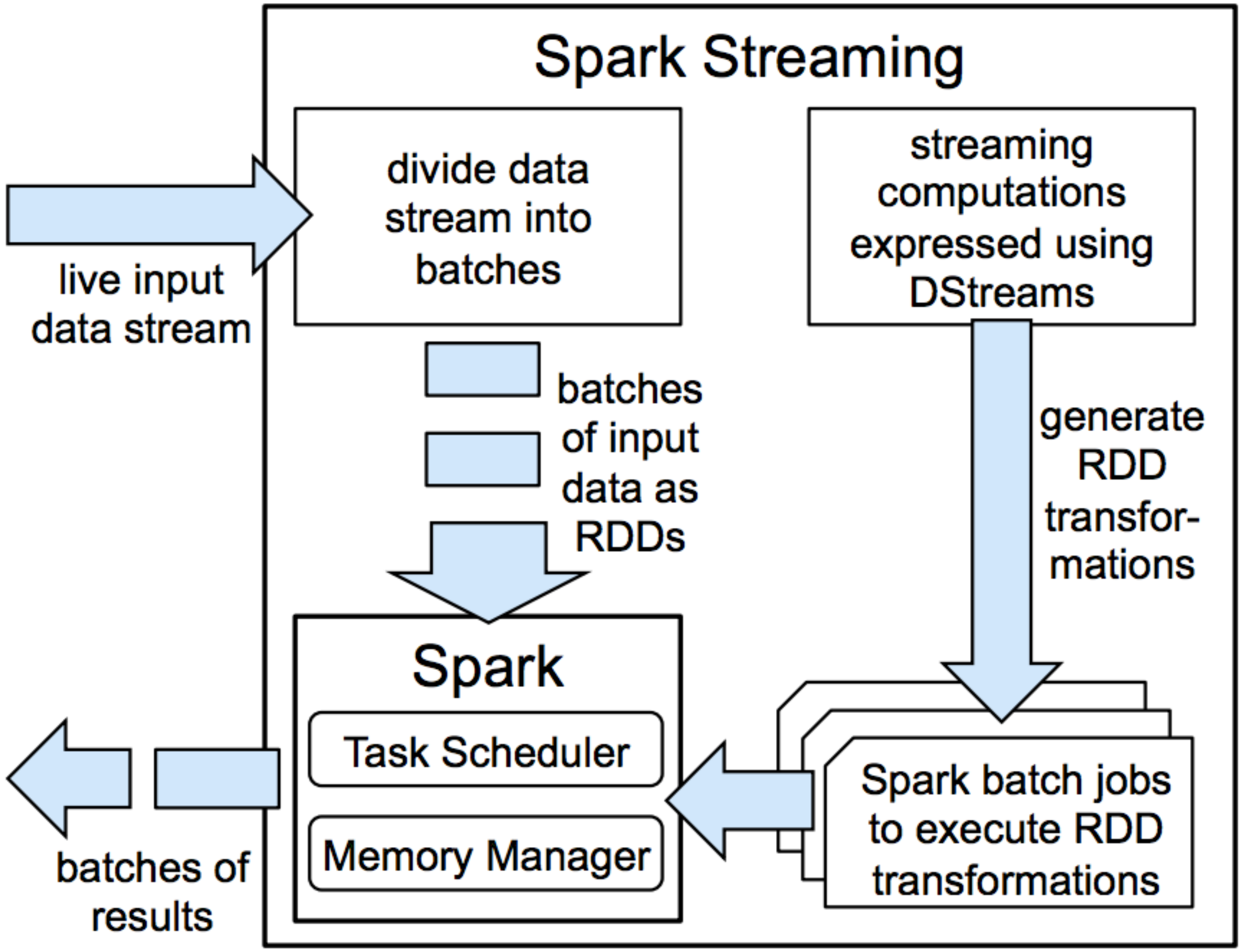


Example

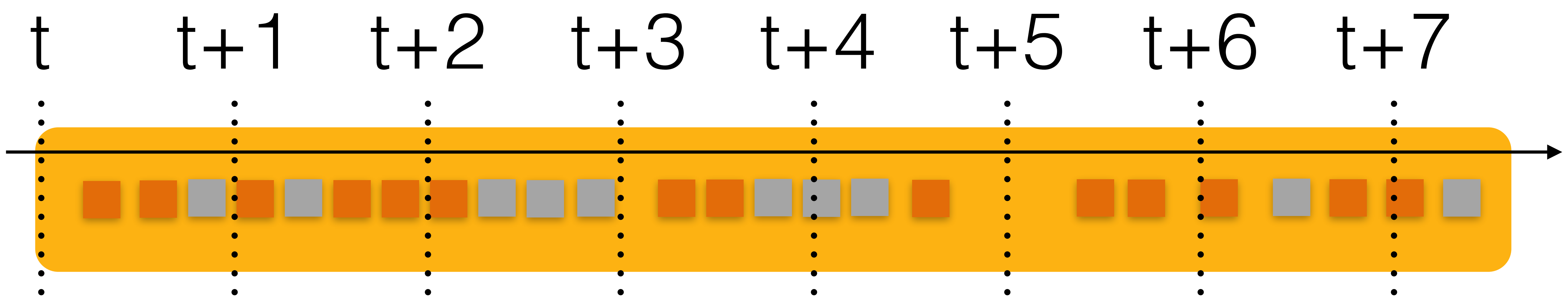
```
pageViews = readStream("http://...", "1s")

ones = pageViews.map(event => (event.url, 1)

counts = ones.runningReduce((a, b) => a + b)
```

- pageViews is a D-Stream grouped into 1s intervals
- ones is a (URL, 1) D-Stream

Streaming as a series of batch jobs

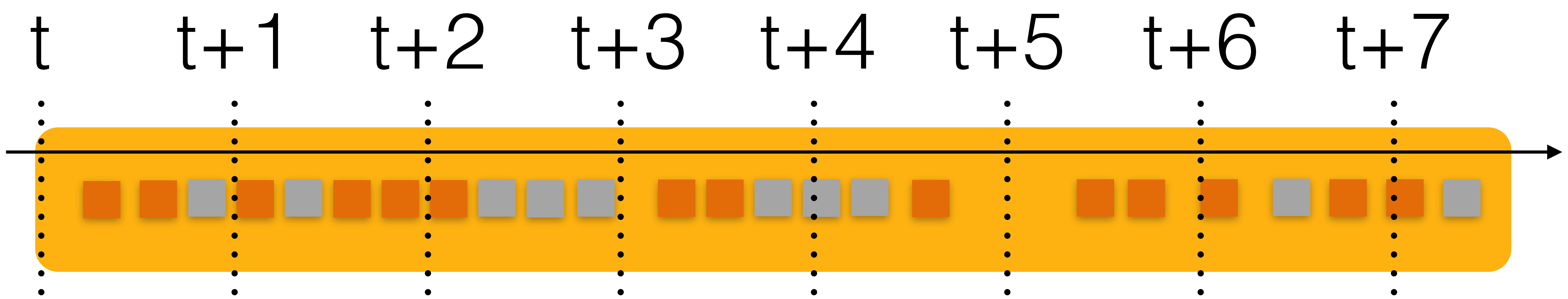


How would you compute…



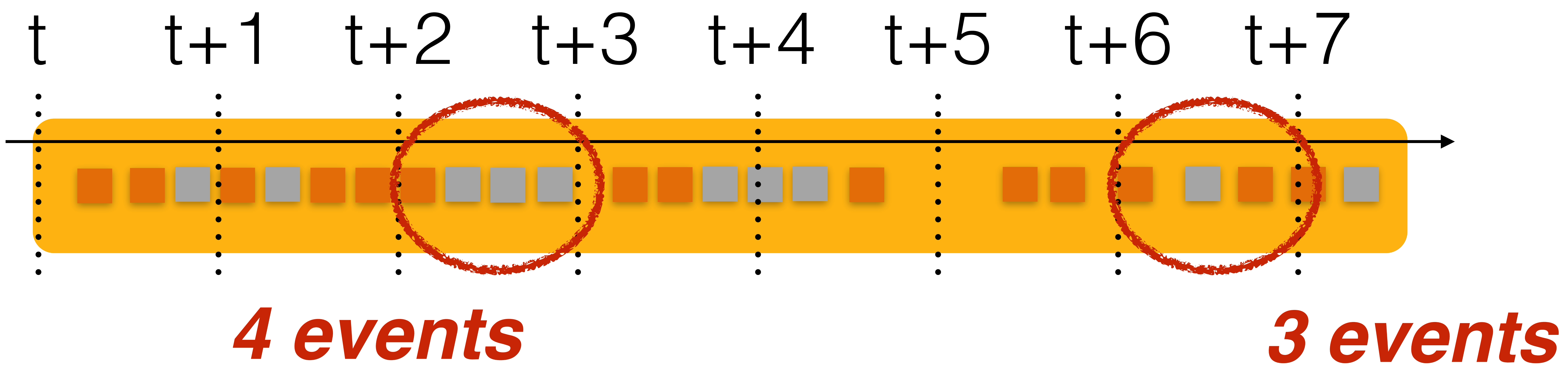
How would you compute…

- the maximum every 100 events?



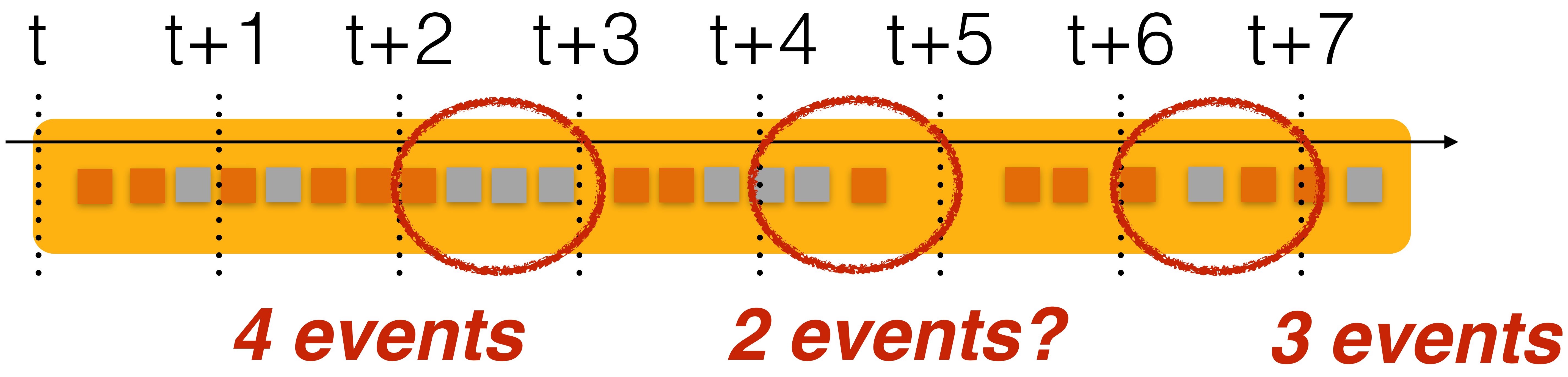
How would you compute…

- the maximum every 100 events?



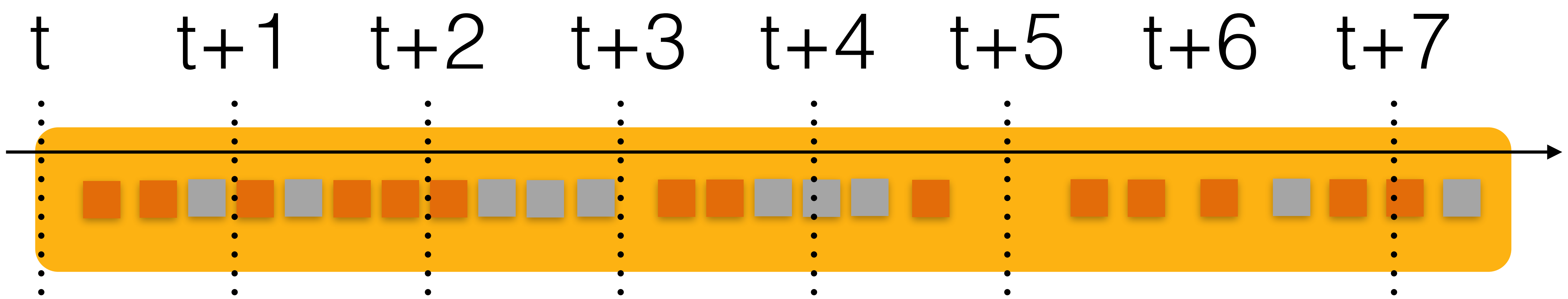
How would you compute…

- the maximum every 100 events?



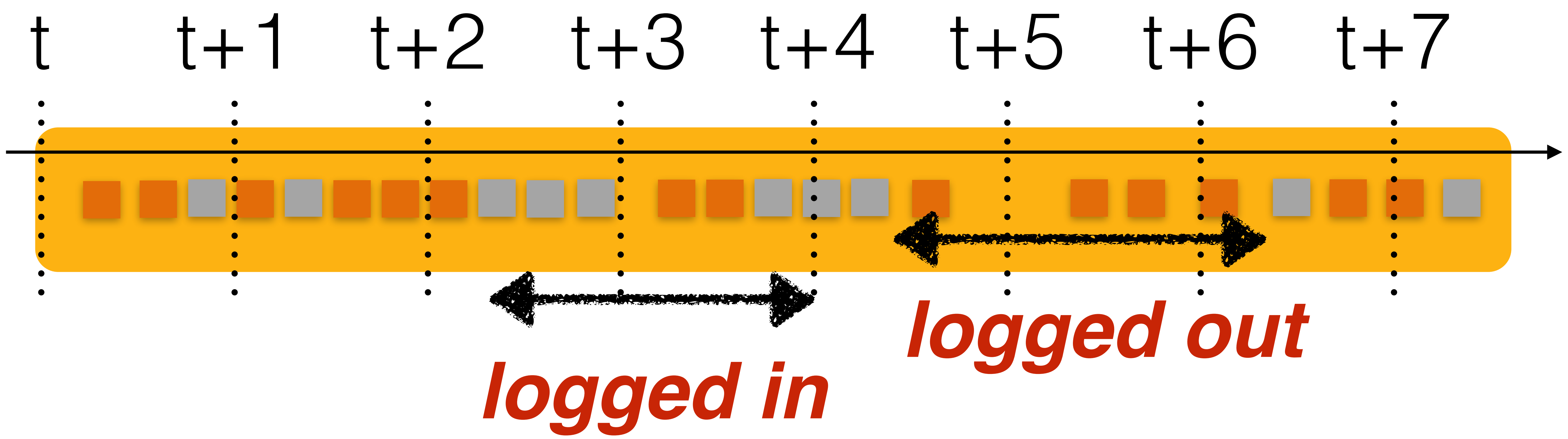
How would you compute…

- the maximum every 100 events?
- clicks per user session?



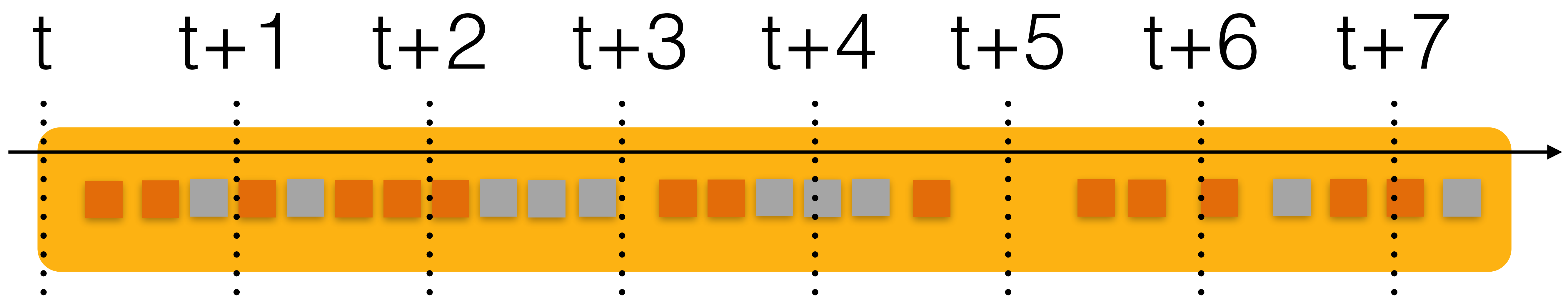
How would you compute…

- the maximum every 100 events?
- clicks per user session?

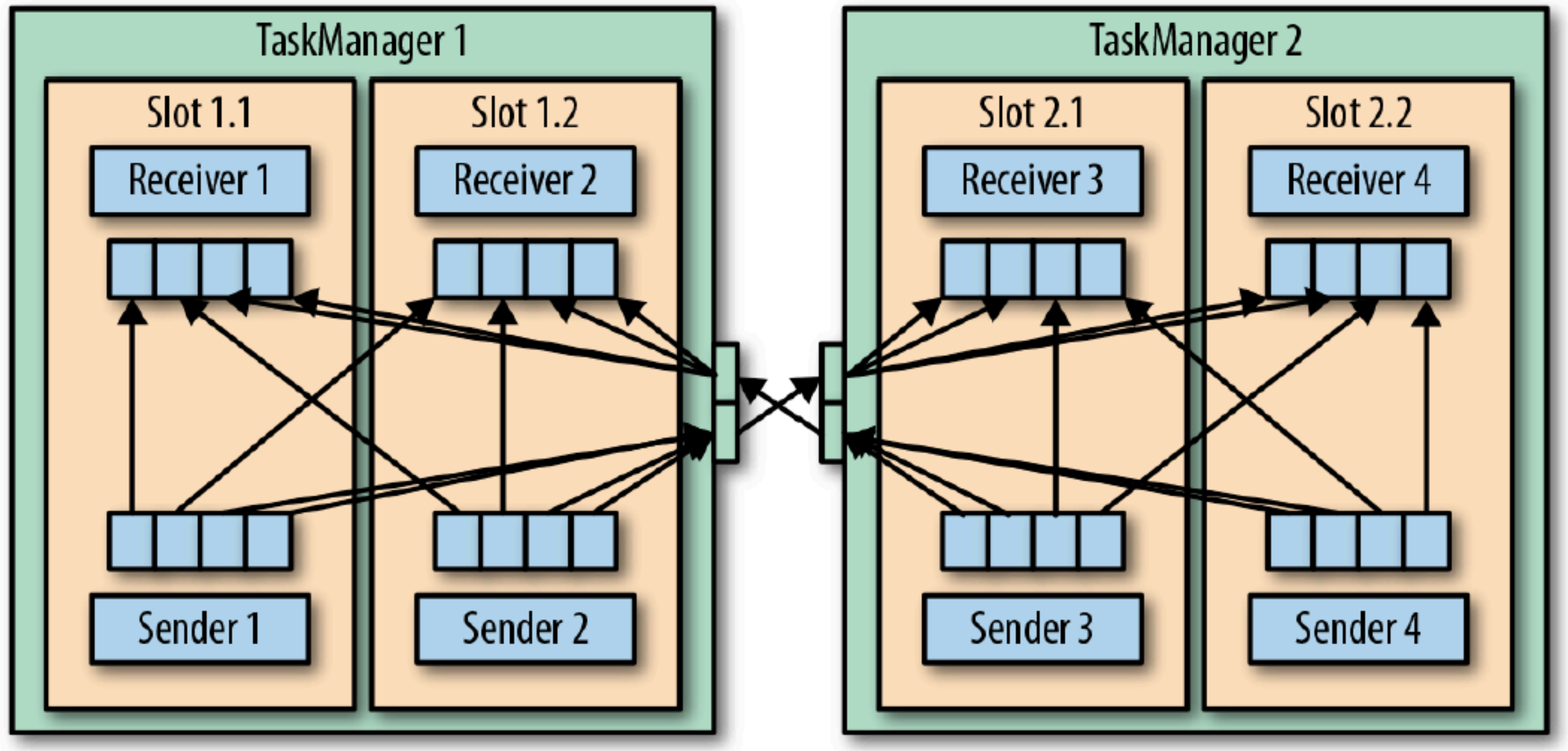


How would you compute…

- the maximum every 100 events?
- clicks per user session?
- faster than the batch size?
- alerts when patterns occur?



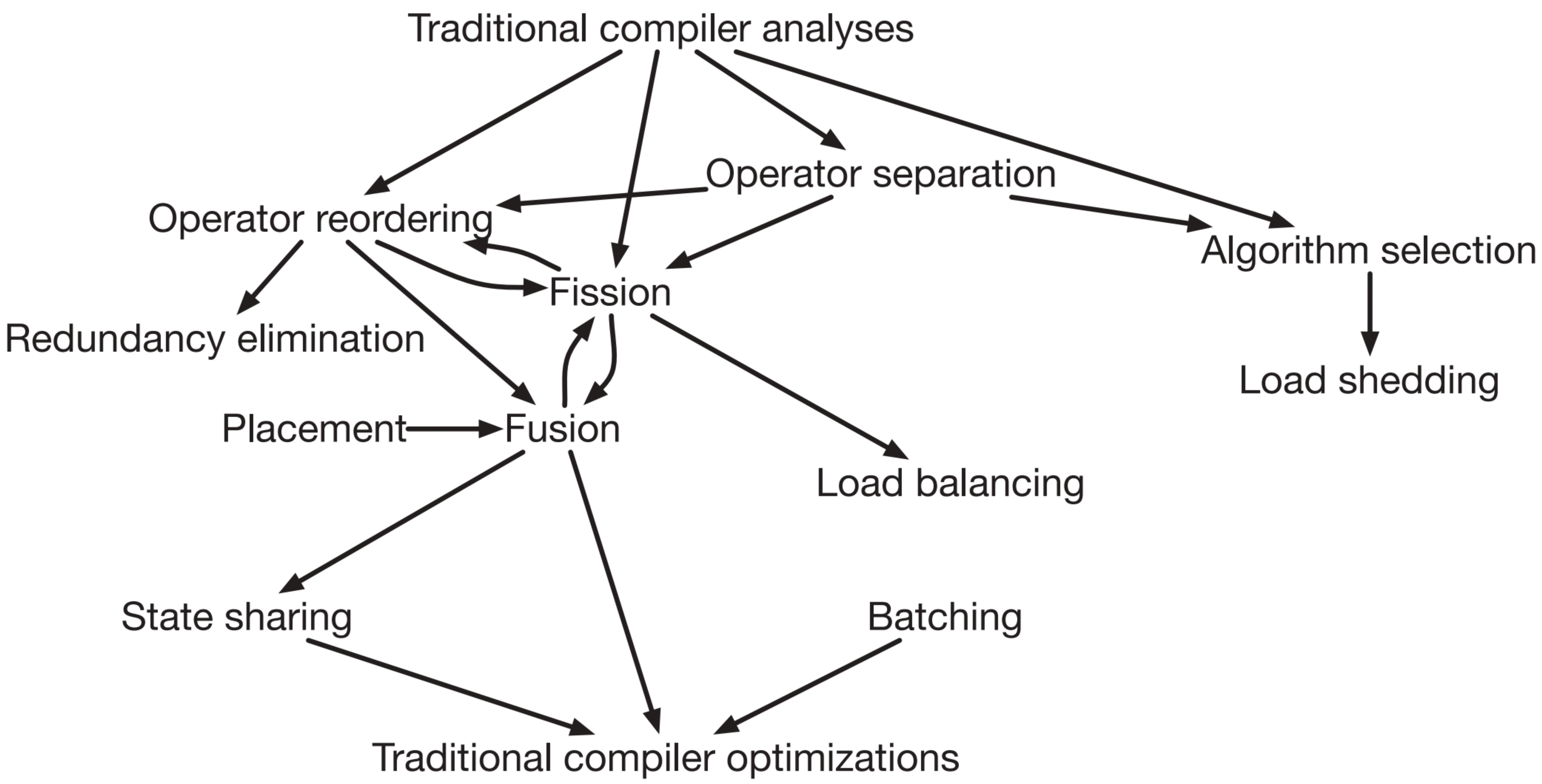
Batching in Apache Flink



- The TaskManagers ship data from sending tasks to receiving tasks.
- The network component of a TaskManager collects records **in buffers** before they are shipped, i.e., records are not shipped one by one but batched.

- TaskManagers have a pool of network buffers to send and receive data.
- If the sender and receiver run in separate processes, they communicate via permanent TCP connections.
- If they run in the same process, the sender task serializes the outgoing records into a byte buffer.
- A TaskManager needs one dedicated network buffer for each receiving task that any of its tasks need to send data to.

Interacting optimizations



Lecture references

- Martin Hirzel et. al. **A Catalog of Stream Processing Optimizations**. (ACM Computing Surveys 2014).
- Ron Avnur and Joseph M. Hellerstein. **Eddies: continuously adaptive query processing**. (SIGMOD 2000).
- Matei Zaharia et. al. **Discretized streams: fault-tolerant streaming computation at scale** (SOSP ’13).
- Fabian Hueske, and Vasiliki Kalavri. **Stream Processing with Apache Flink**. (O’Reilly Media ’19).

Further reading

- Re-ordering
  - Shivnath Babu et. al. **Adaptive Ordering of Pipelined Stream Filters**. SIGMOD 2004.
- Scheduling and placement
  - Peter R. Pietzuch et. al. **Network-Aware Operator Placement for Stream-Processing Systems**. ICDE 2006.
  - Brian Babcock et. al. **Chain : Operator Scheduling for Memory Minimization in Data Stream Systems**. SIGMOD 2003.
  - Donald Carney et. al. **Operator Scheduling in a Data Stream Manager**. VLDB 2003.
- Load balancing and skew mitigation
  - Muhammad Anis Uddin Nasir et. al. **The power of both choices: Practical load balancing for distributed stream processing engines.** ICDE 2015.
  - Nikos R. Katsipoulakis et. al. **A holistic view of stream partitioning costs**. VLDB 2017.
- Rate-based optimization
  - Statis Viglas and Jeffrey Naughton. **Rate-based Query Optimization for Streaming Information Sources**. SIGMOD 2002.