Big Data Processing

CS 240: Computing Systems and Concurrency

Lecture 23

Marco Canini

Data-Parallel Computation

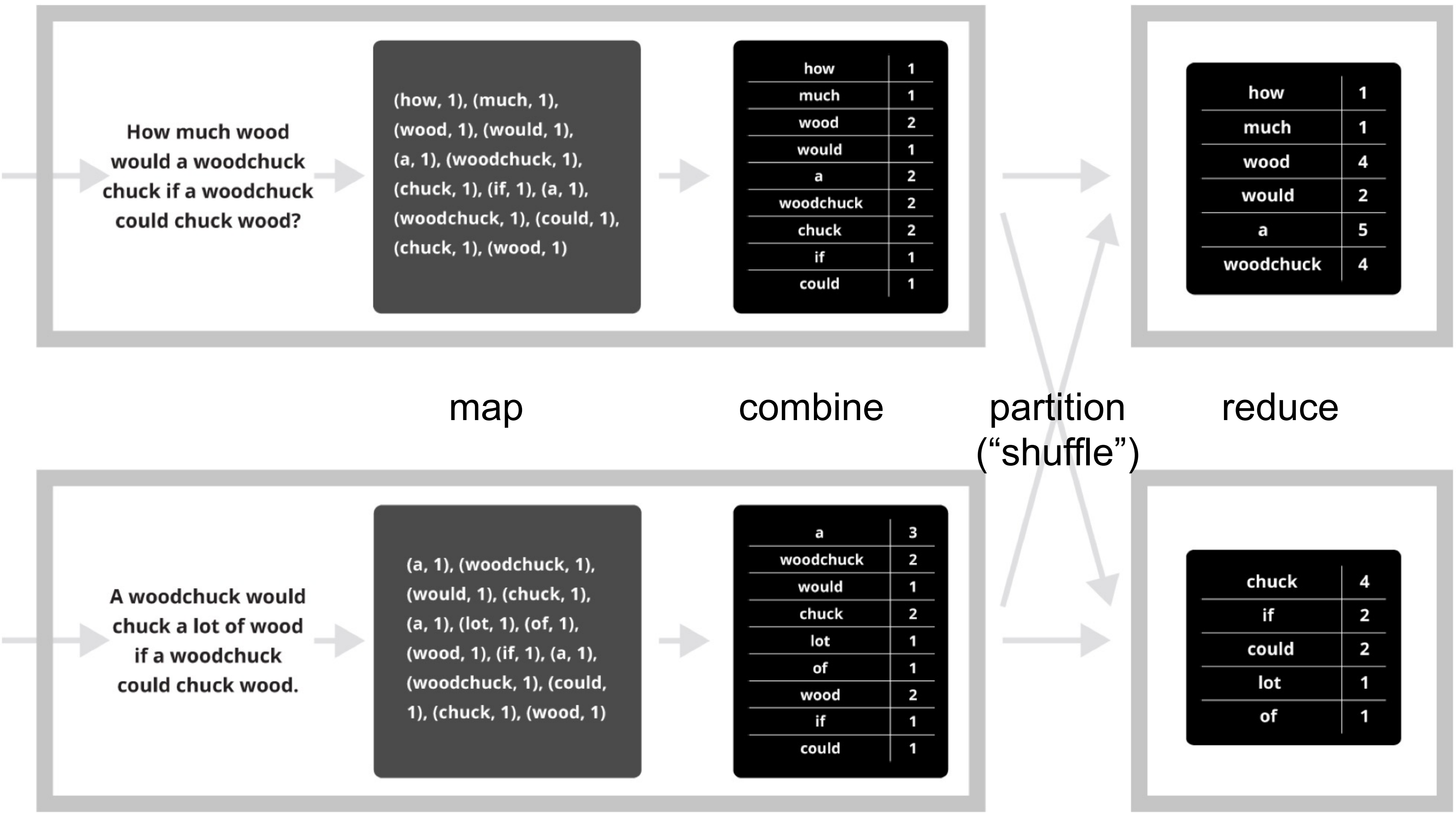
Ex: Word count using partial aggregation

1. Compute word counts from individual files

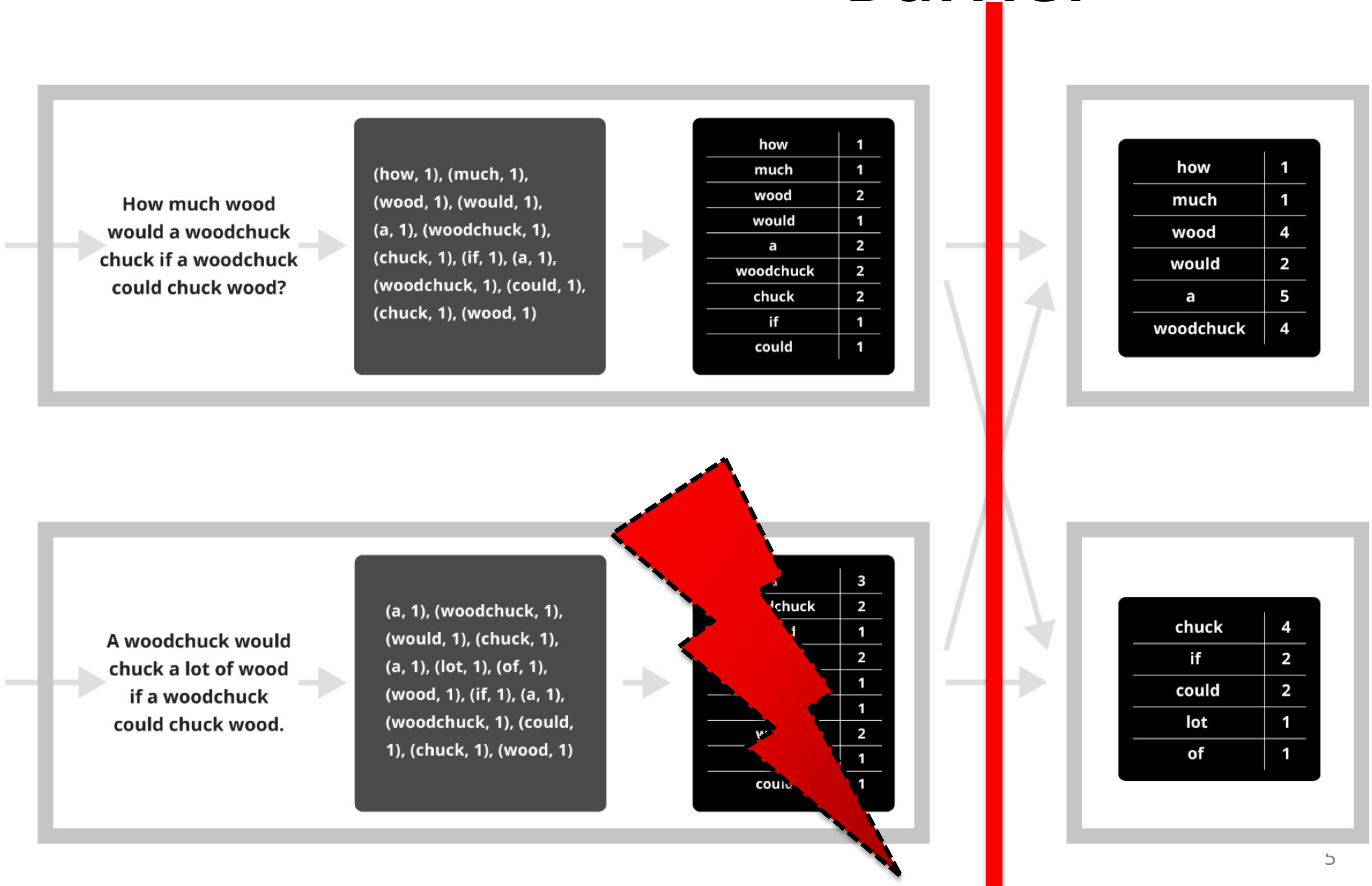
2. Then merge intermediate output

3. Compute word count on merged outputs

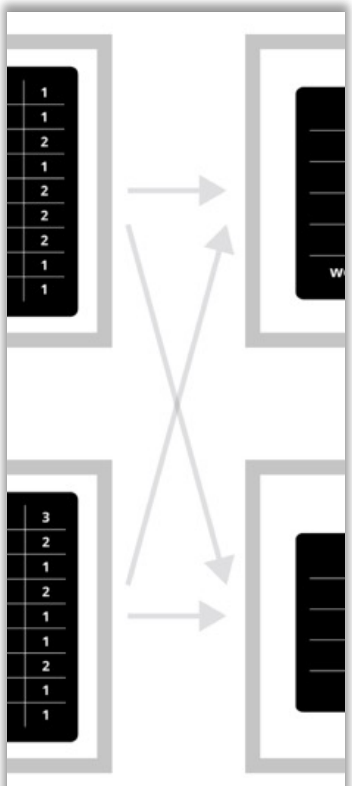
Putting it together…



Synchronization Barrier



Fault Tolerance in MapReduce



- Map worker writes intermediate output to local disk, separated by partitioning. Once completed, tells master node

- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data

- Note:
  - "All-to-all" shuffle b/w mappers and reducers
  - Written to disk ("materialized") b/w *each* stage

Generality vs Specialization

General Systems

- Can be used for many different applications
- Jack of all trades, master of none
  - Pay a generality penalty
- Once a specific application, or class of applications becomes sufficiently important, time to build specialized systems

MapReduce is a General System

- Can express large computations on large data; enables fault tolerant, parallel computation

- Fault tolerance is an inefficient fit for many applications

- Parallel programming model (map, reduce) within synchronous rounds is an inefficient fit for many applications

MapReduce for Google’s Index

- Flagship application in original MapReduce paper

- *Q: What is inefficient about MapReduce for computing web indexes?*
  - “MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency.”

- Index moved to Percolator in ~2010 [OSDI ‘10]
  - Incrementally process updates to index
  - Uses OCC to apply updates
  - 50% reduction in average age of documents

MapReduce for Iterative Computations

- Iterative computations: compute on the same data as we update it
  - e.g., PageRank
  - e.g., Logistic regression

- *Q: What is inefficient about MapReduce for these?*
  - Writing data to disk between all iterations is slow

- Many systems designed for iterative computations, most notable is Apache Spark
  - Key idea 1: Keep data in memory once loaded
  - Key idea 2: Provide fault tolerance via *lineage* (record ops)

MapReduce for Stream Processing

- Stream processing: Continuously process an infinite stream of incoming events
  - e.g., estimating traffic conditions from GPS data
  - e.g., identify trending hashtags on twitter
  - e.g., detect fraudulent ad-clicks

- *Q: What is inefficient about MapReduce for these?*

Stream Processing Systems

- Many stream processing systems as well, typical structure:
  - Definite computation ahead of time
  - Setup machines to run specific parts of computation and pass data around (topology)
  - Stream data into topology
  - Repeat forever
  - Trickiest part: fault tolerance!

- Notably systems and their fault tolerance
  - Apache/Twitter Storm: Record acknowledgment
  - Spark Streaming: Micro-batches
  - Google Cloud dataflow: transactional updates
  - Apache Flink: Distributed snapshot

- Specialization is much faster, e.g., click-fraud detection at Microsoft
  - Batch-processing system: 6 hours
  - w/ StreamScope[NSDI '16]: 20 minute average

In-Memory Data-Parallel Computation

Spark: Resilient Distributed Datasets

- Let’s think of just having a big block of RAM, partitioned across machines…
  - And a series of operators that can be executed in parallel across the different partitions

- That’s basically Spark
  - A distributed memory abstraction that is both fault-tolerant and efficient

Spark: Resilient Distributed Datasets

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)
  - They are called **Resilient Distributed Datasets** (RDDs)

- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

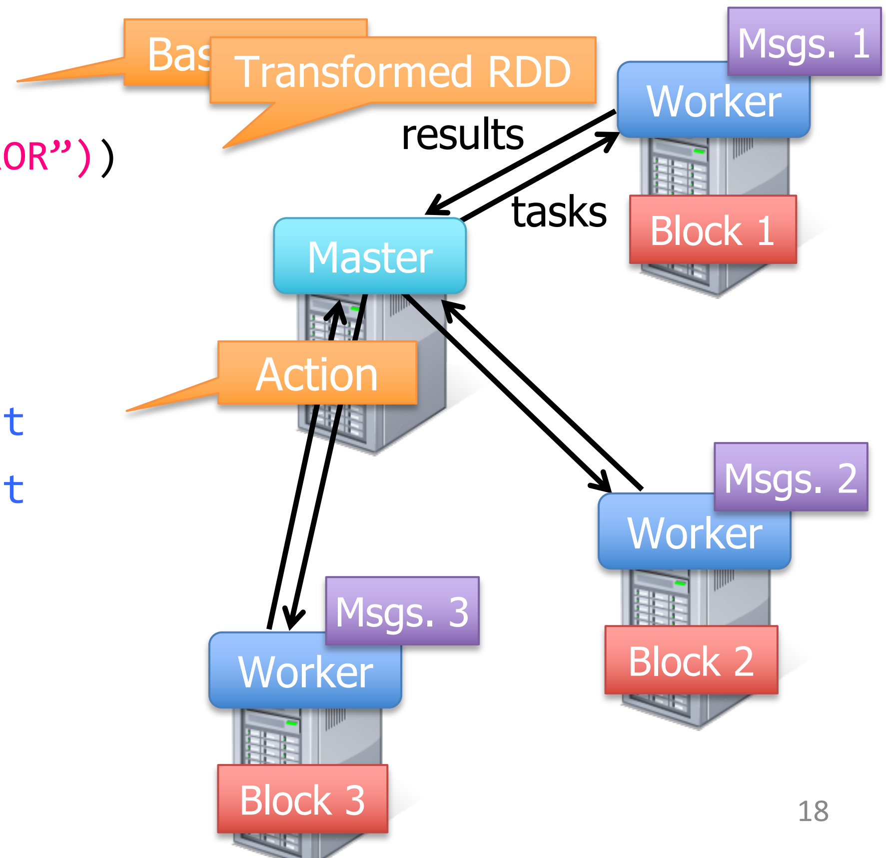
Spark Programming Interface

- Language-integrated API in Scala (+ Python)
- Usable interactively via Spark shell
- Provides:
  - Resilient distributed datasets (RDDs)
  - Operations on RDDs: deterministic *transformations* (build new RDDs), *actions* (compute and output results)
  - Control of each RDD's *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)

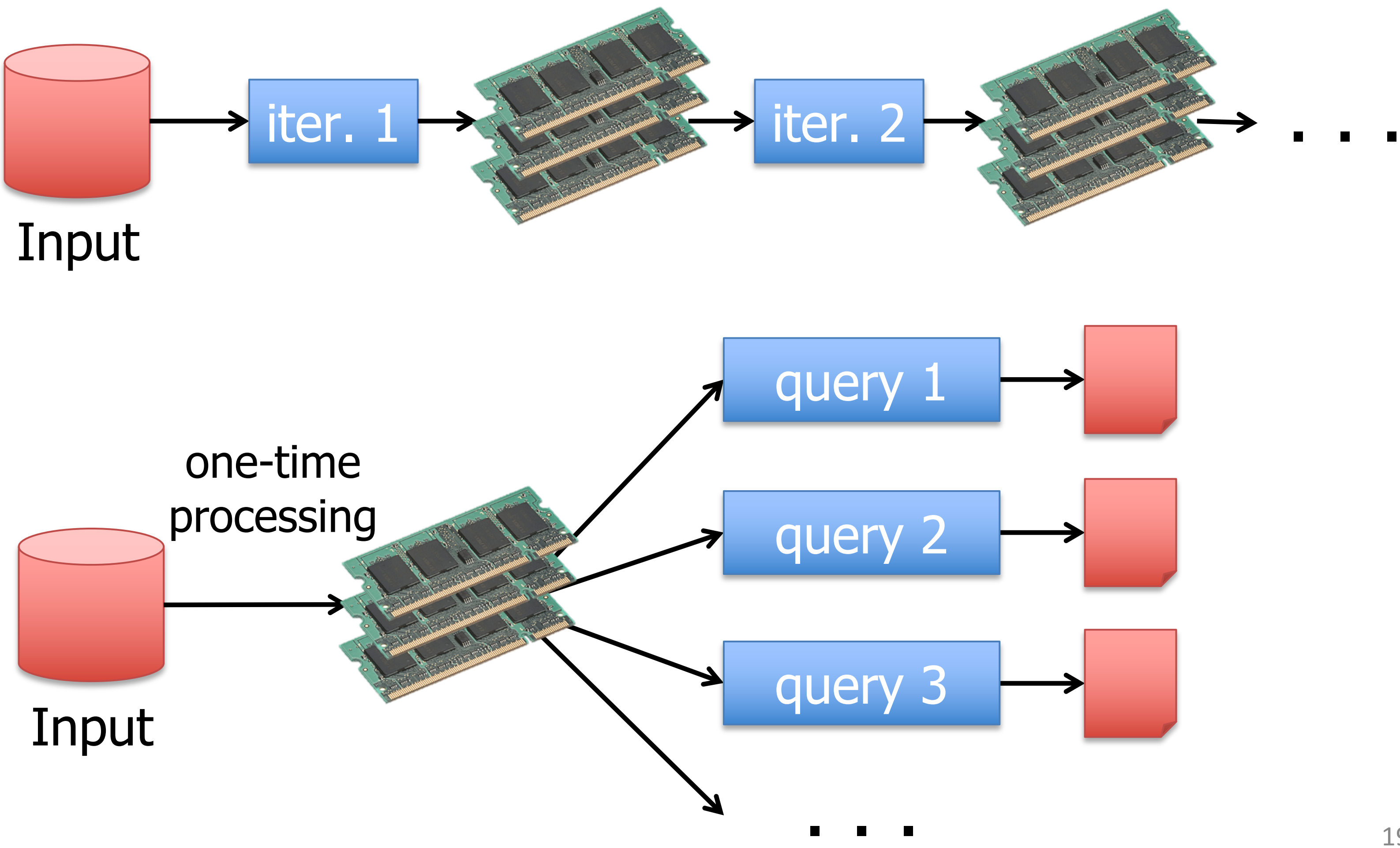
Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

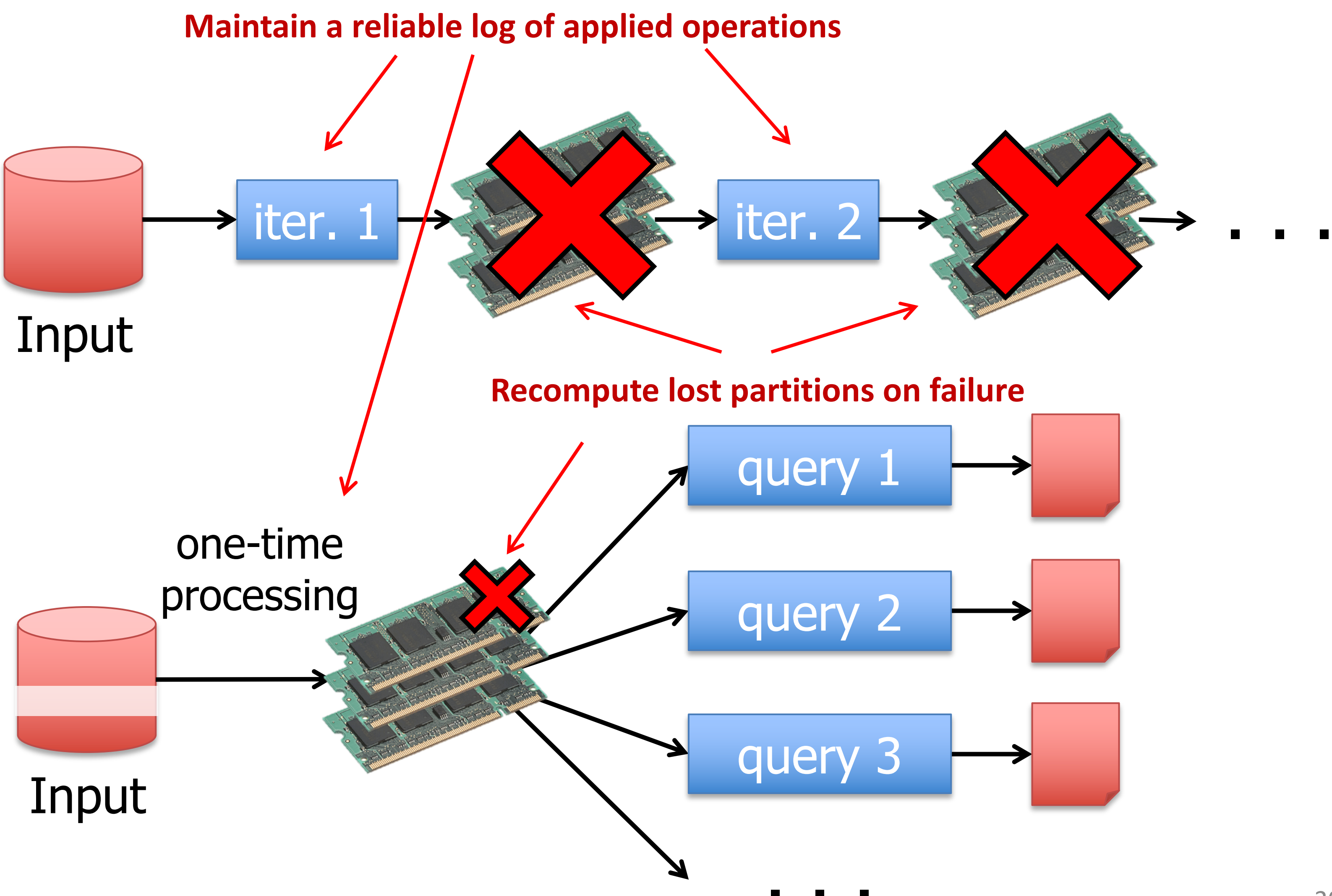
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```



In-Memory Data Sharing



Efficient Fault Recovery via Lineage



Generality of RDDs

- Despite their restrictions, RDDs can express many parallel algorithms
  - These naturally *apply the same operation to many items*
- Unify many programming models
  - *Data flow models:* MapReduce, Dryad, SQL, …
  - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …
- Support *new apps* that these models don't
- Enables apps to efficiently *intermix* these models

Spark Operations

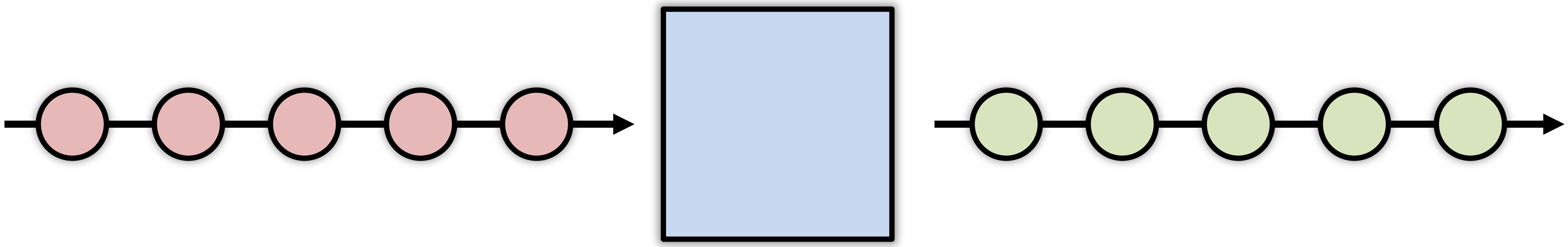
Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey take	

Spark Summary

- Global aggregate computations that produce program state
  - compute the count() of an RDD, compute the max diff, etc.
- Loops!
  - Spark makes it much easier to do multi-stage MapReduce
- Built-in abstractions for some other common operations like joins
- See also Apache Flink for a flexible big data platform

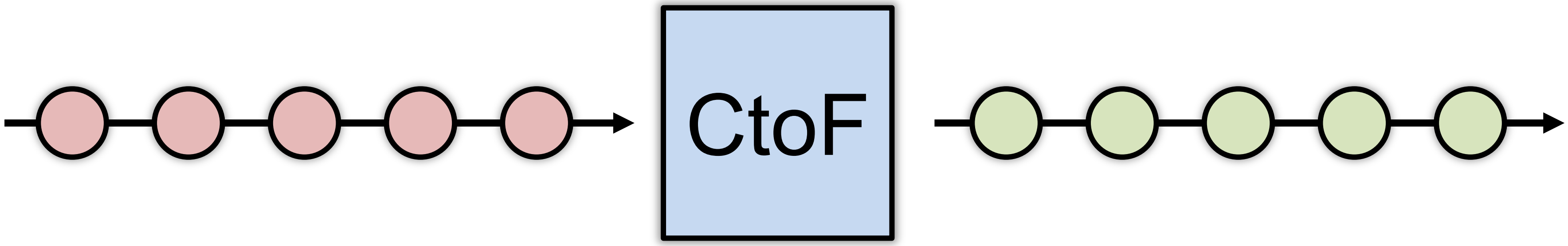
Stream Processing

Simple stream processing



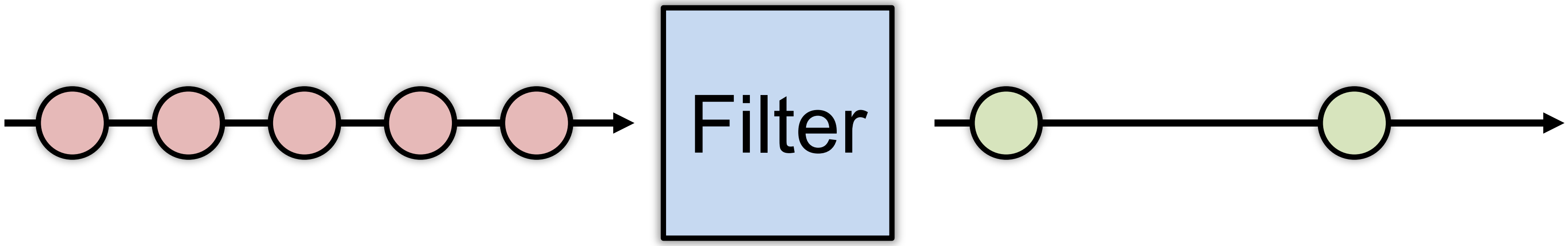
- Single node/process
  - Read data from input source (e.g., network socket)
  - Process
  - Write output

Examples: Stateless conversion



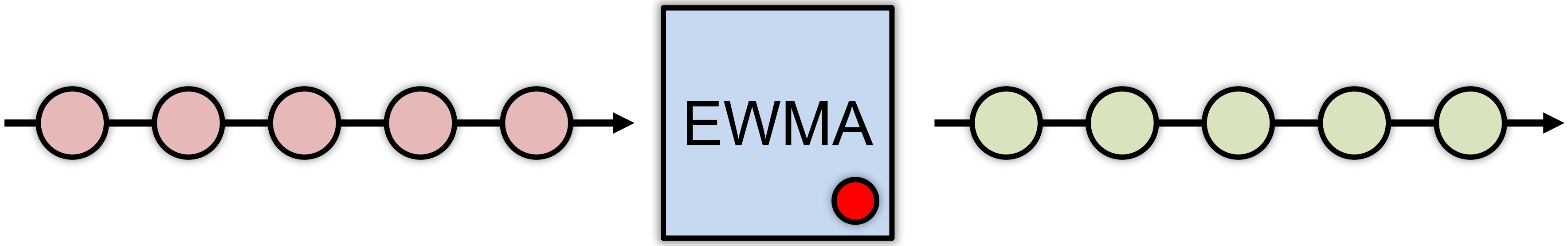
- Convert Celsius temperature to Fahrenheit
  - Stateless operation: **emit** (input * 9 / 5) + 32

Examples: Stateless filtering



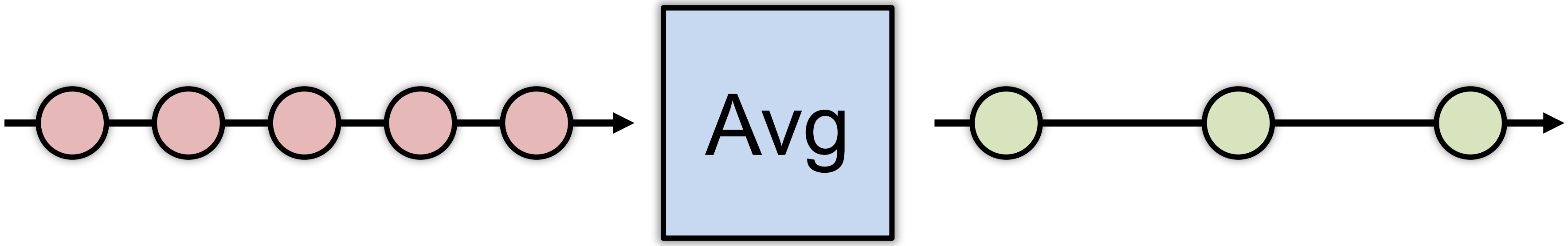
- Function can filter inputs
  - if (input > threshold) { **emit** input }

Examples: Stateful conversion

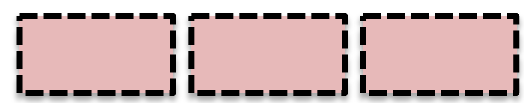


- Compute EWMA of Fahrenheit temperature
  - new_temp = α * (CtoF(input)) + (1- α) * last_temp
  - last_temp = new_temp
  - **emit** new_temp

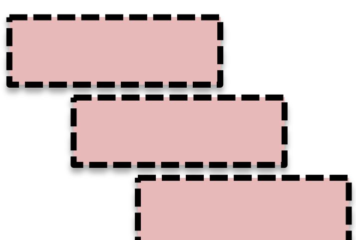
Examples: Aggregation (stateful)



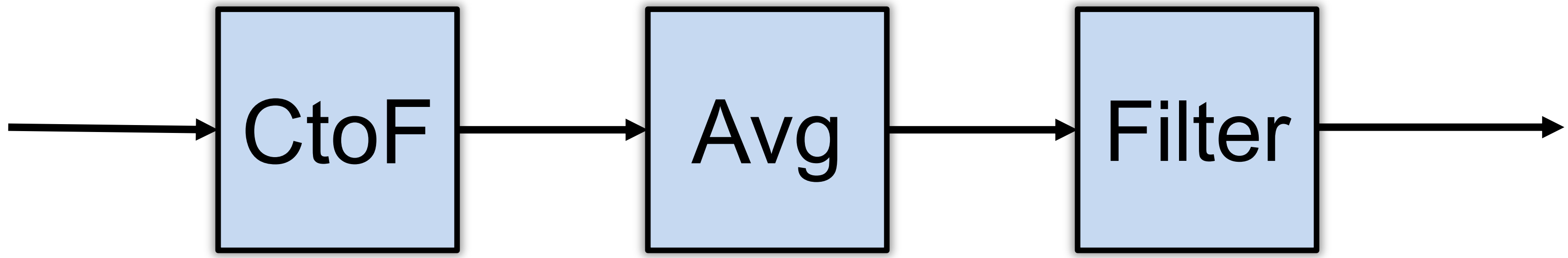
- E.g., Average value per window
  - Window can be # elements (10) or time (1s)
  - Windows can be fixed (every 5s)



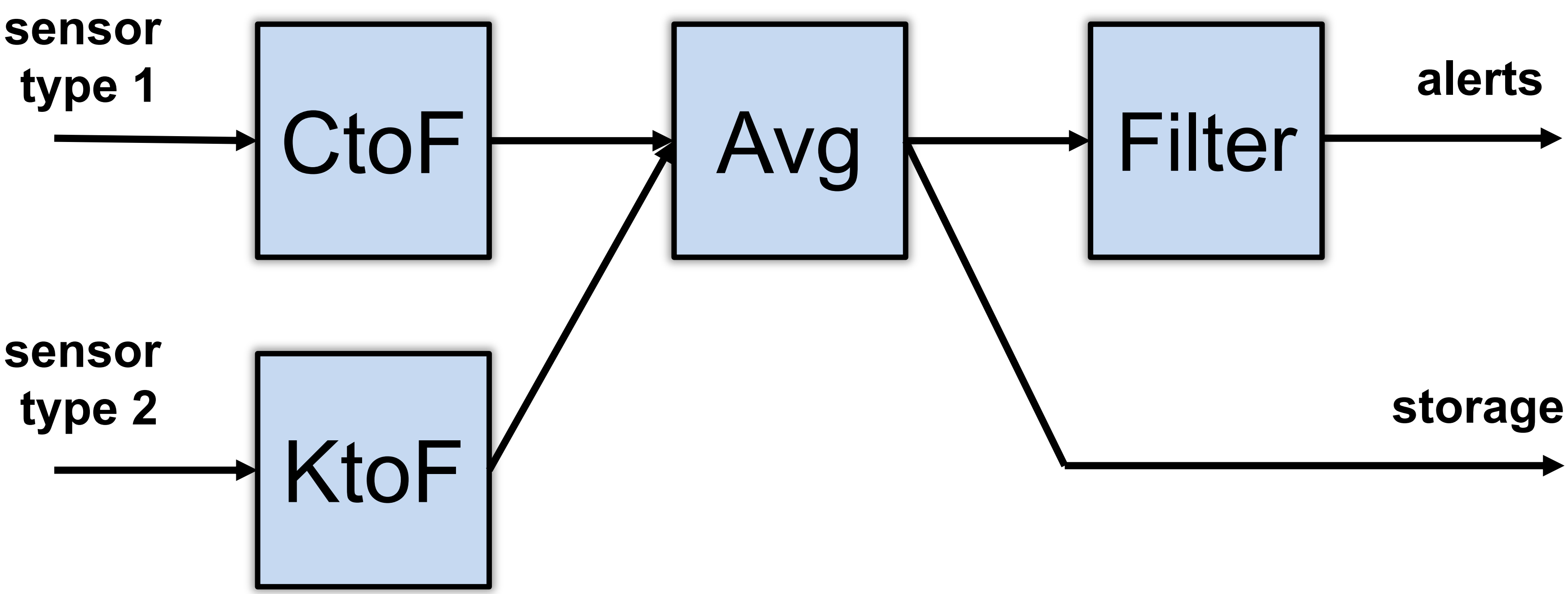
  - Windows can be “sliding” (5s window every 1s)



Stream processing as chain



Stream processing as directed graph



The challenge of stream processing

- Large amounts of data to process in real time
- Examples
  - Social network trends (#trending)
  - Intrusion detection systems (networks, datacenters)
  - Sensors: Detect earthquakes by correlating vibrations of millions of smartphones
  - Fraud detection
    - Visa: 2000 txn / sec on average, peak ~47,000 / sec

Scale “up”: batching

Tuple-by-Tuple

```
input ← read
if (input > threshold)  {
     emit input
}
```

Micro-batch

```
inputs ← read
out = []
for input in inputs {
     if (input > threshold) {
          out.append(input)
     }
}
emit out
```

Scale “up”

Tuple-by-Tuple

Lower Latency

Lower Throughput

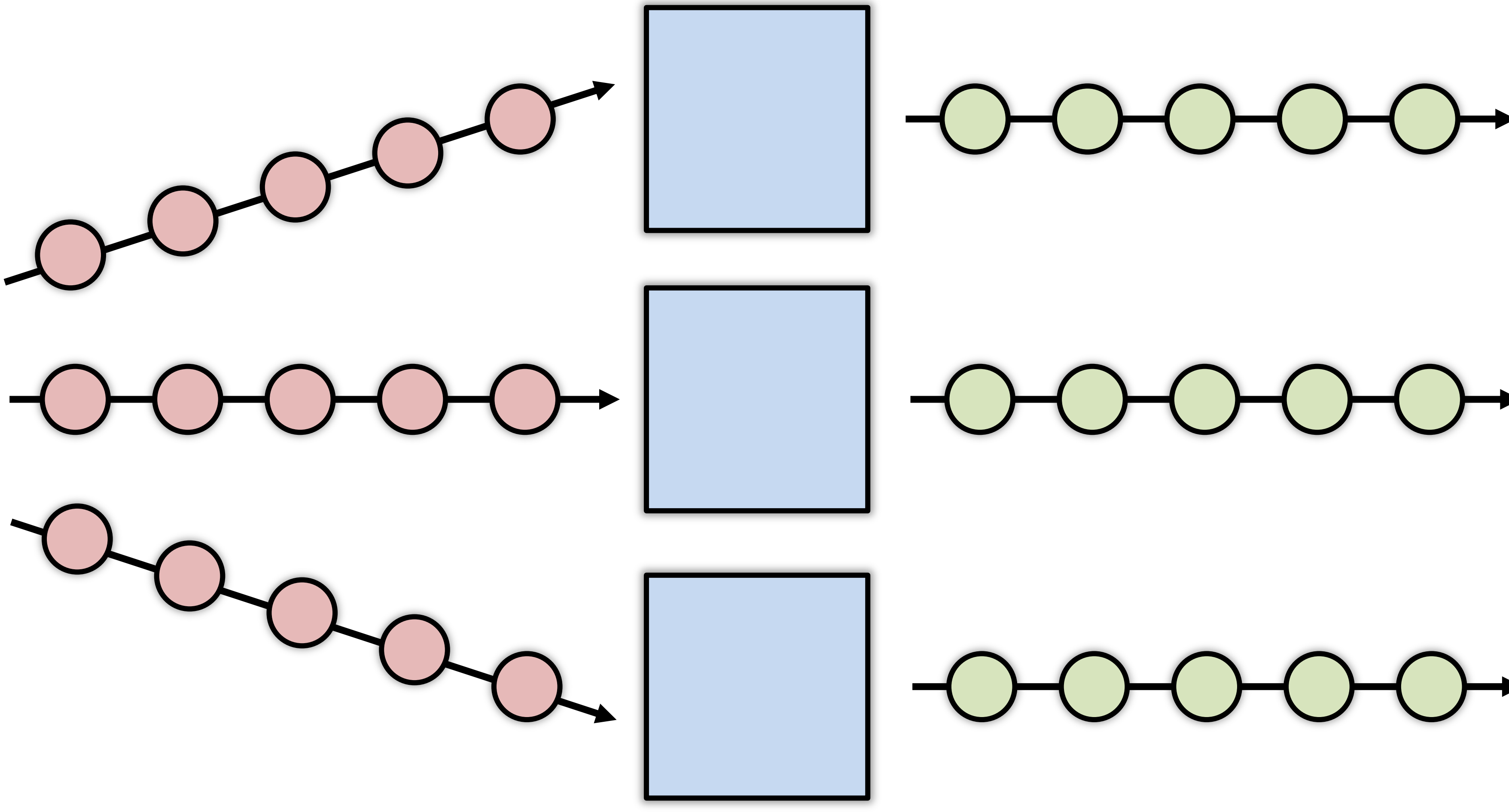
Micro-batch

Higher Latency

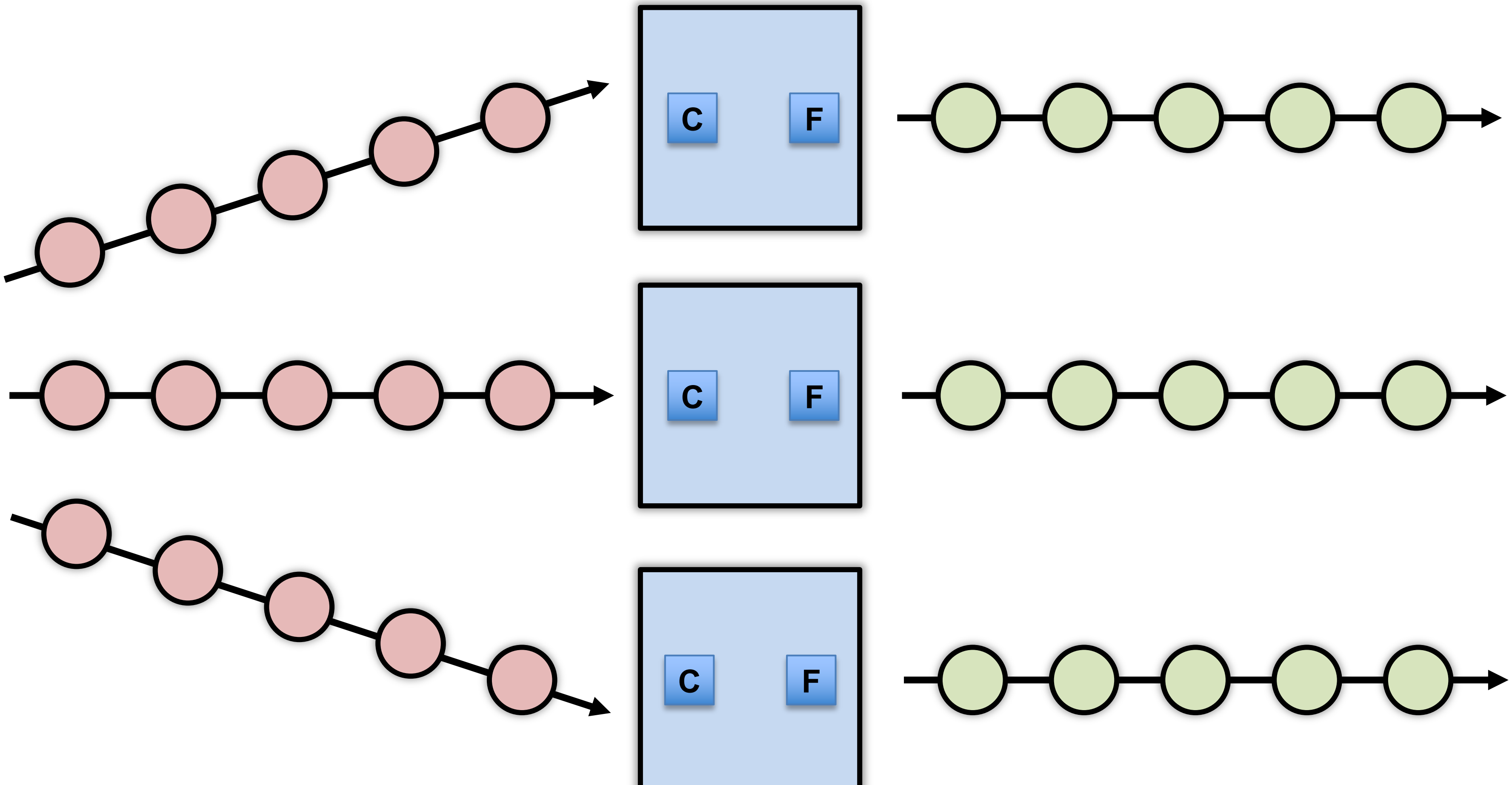
Higher Throughput

Why? Each read/write is an system call into kernel. More cycles performing kernel/application transitions (context switches), less actually spent processing data.

Scale “out”

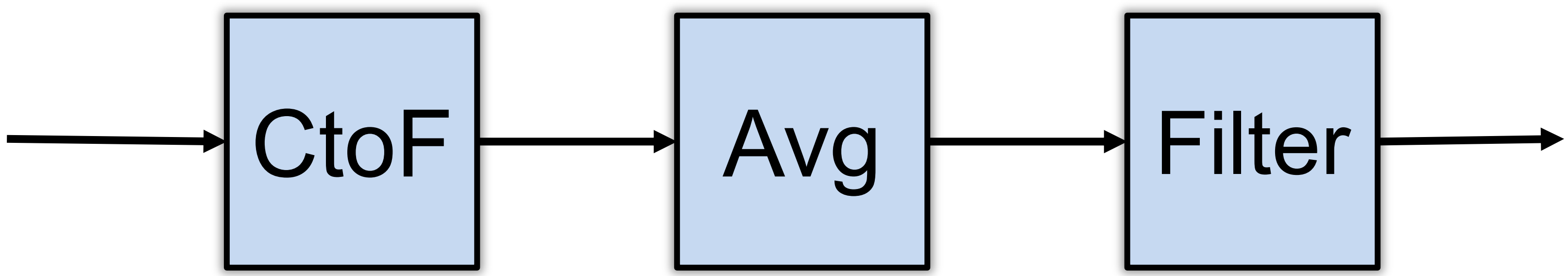


Stateless operations: trivially parallelized



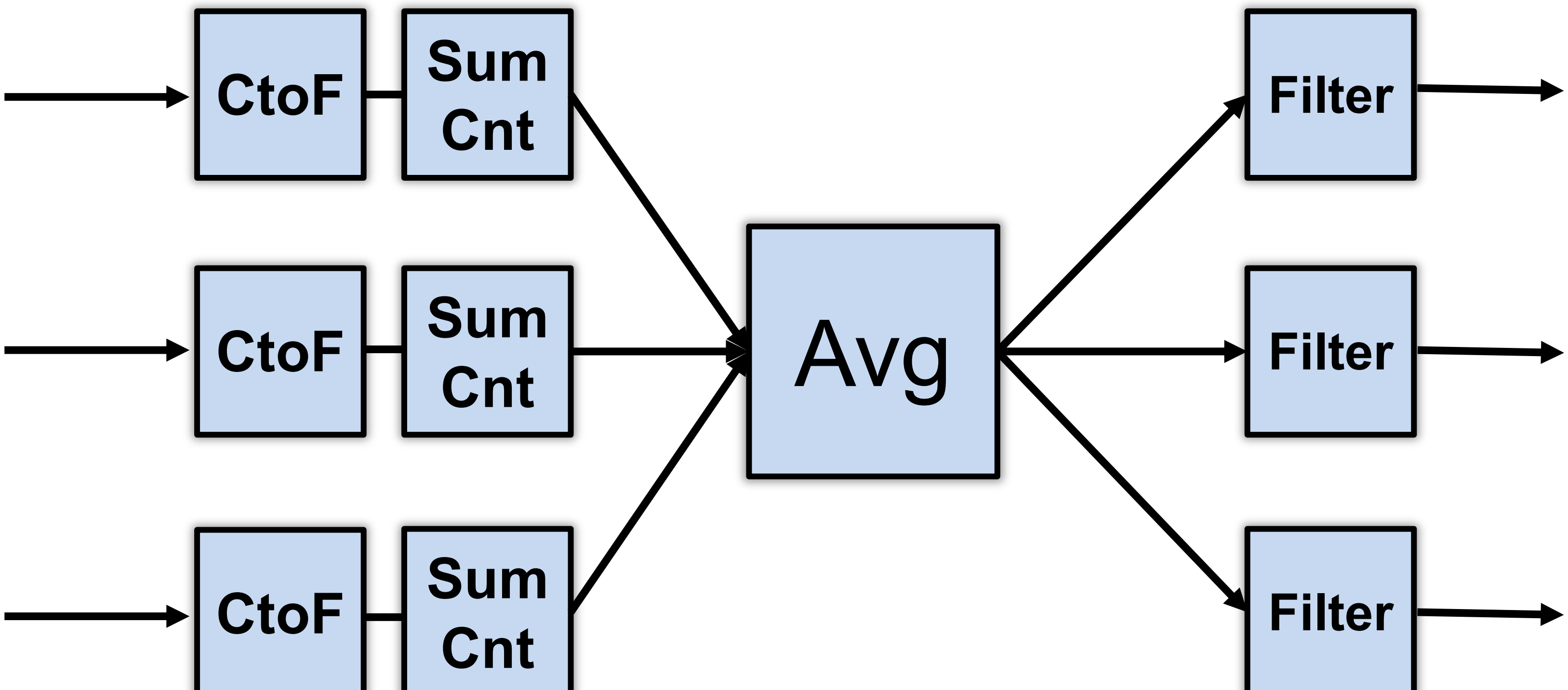
State complicates parallelization

- Aggregations:
  - Need to join results across parallel computations



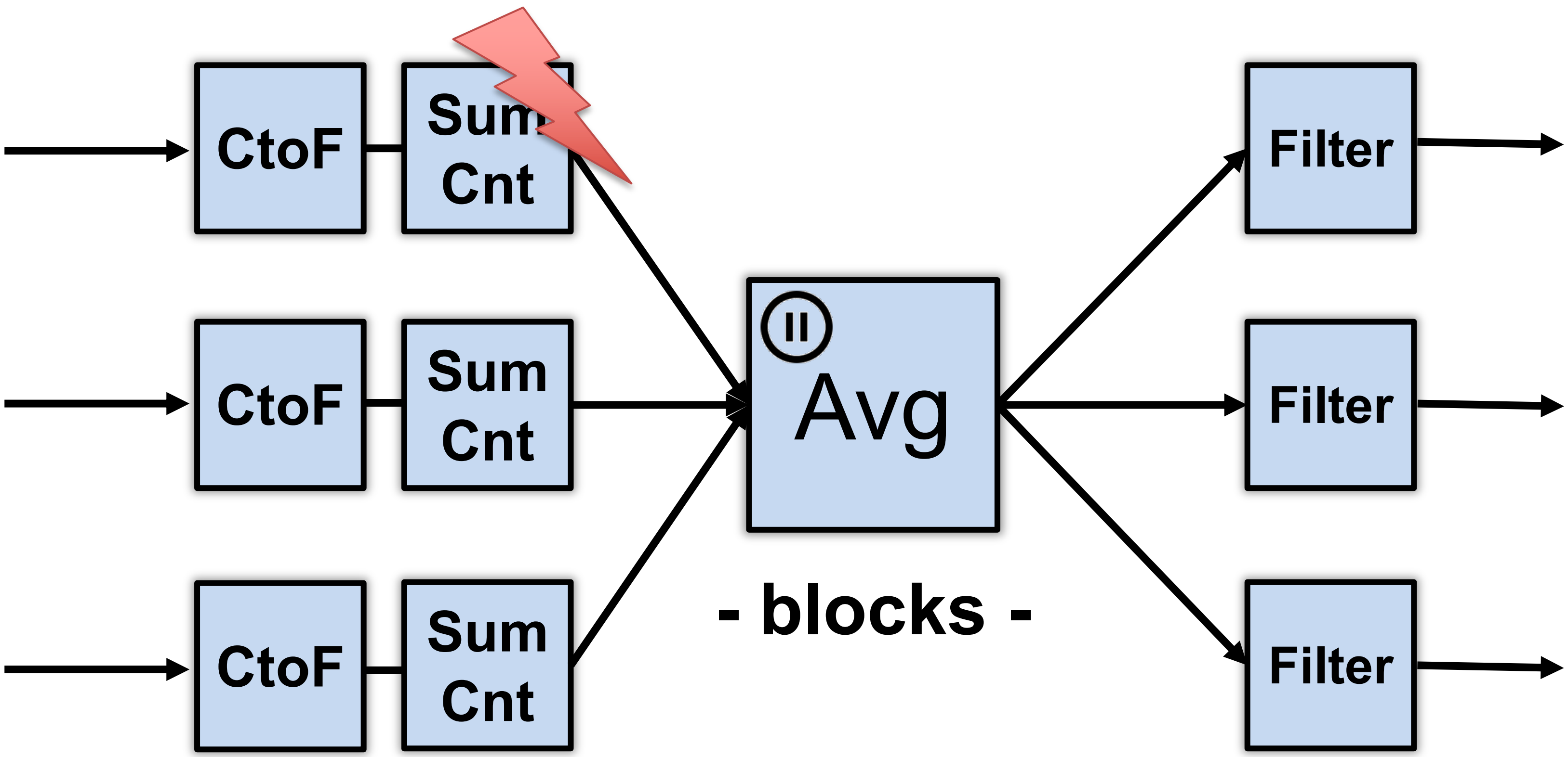
State complicates parallelization

- Aggregations:
  - Need to join results across parallel computations



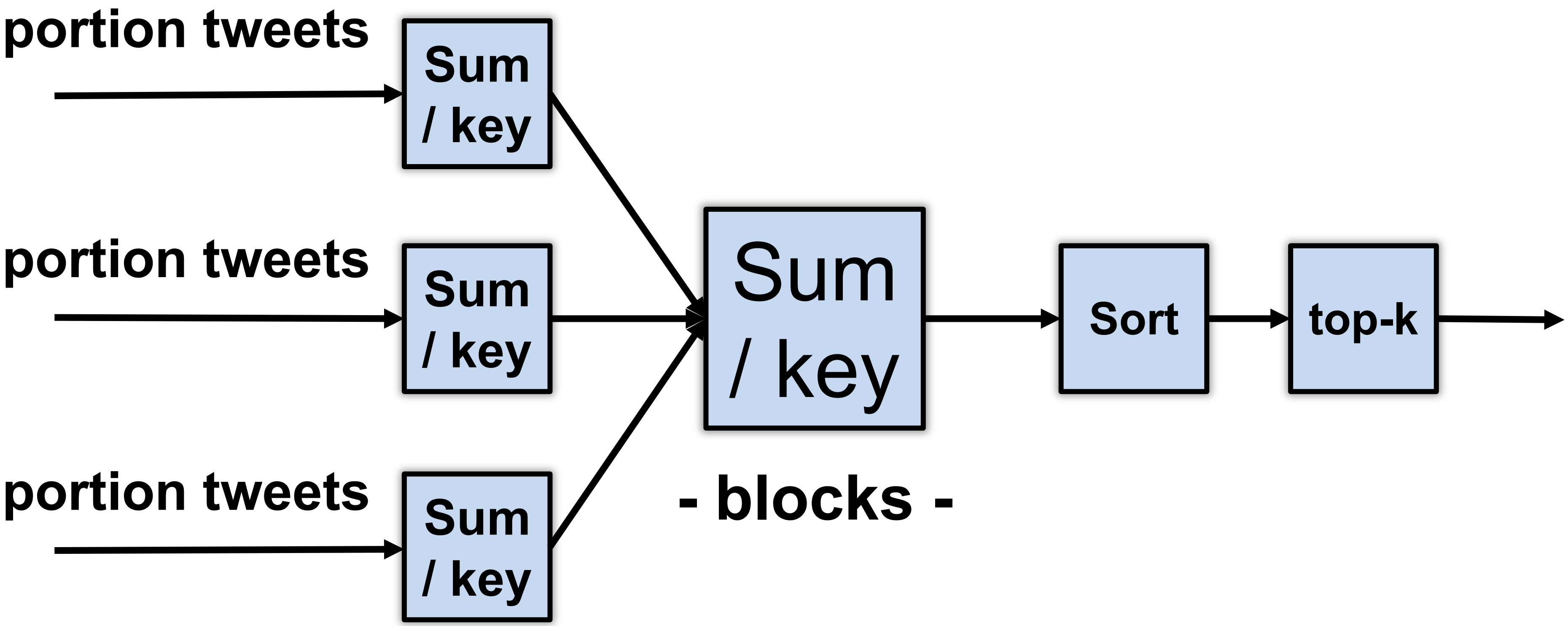
Parallelization complicates fault-tolerance

- Aggregations:
  - Need to join results across parallel computations

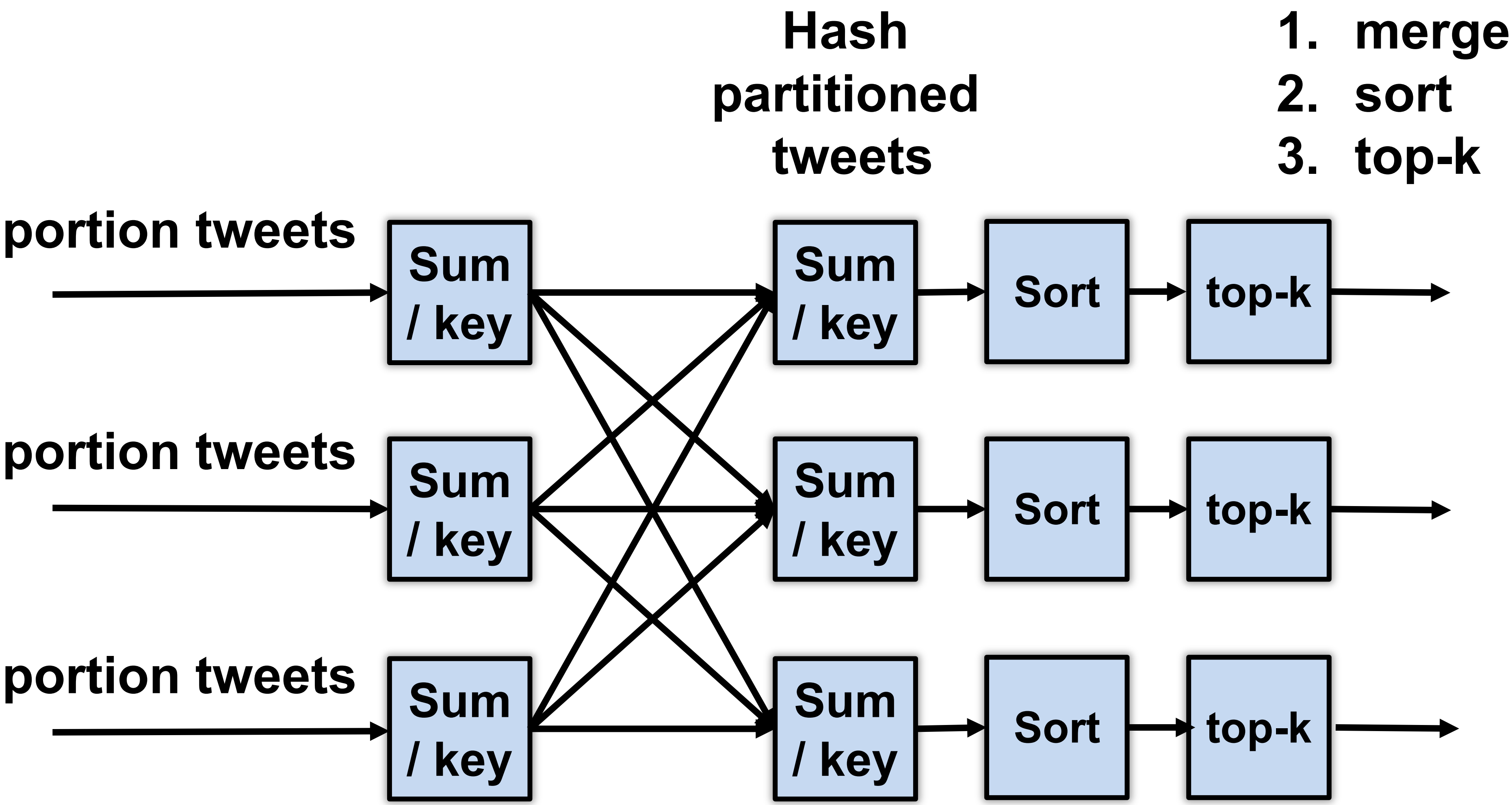


Can parallelize joins

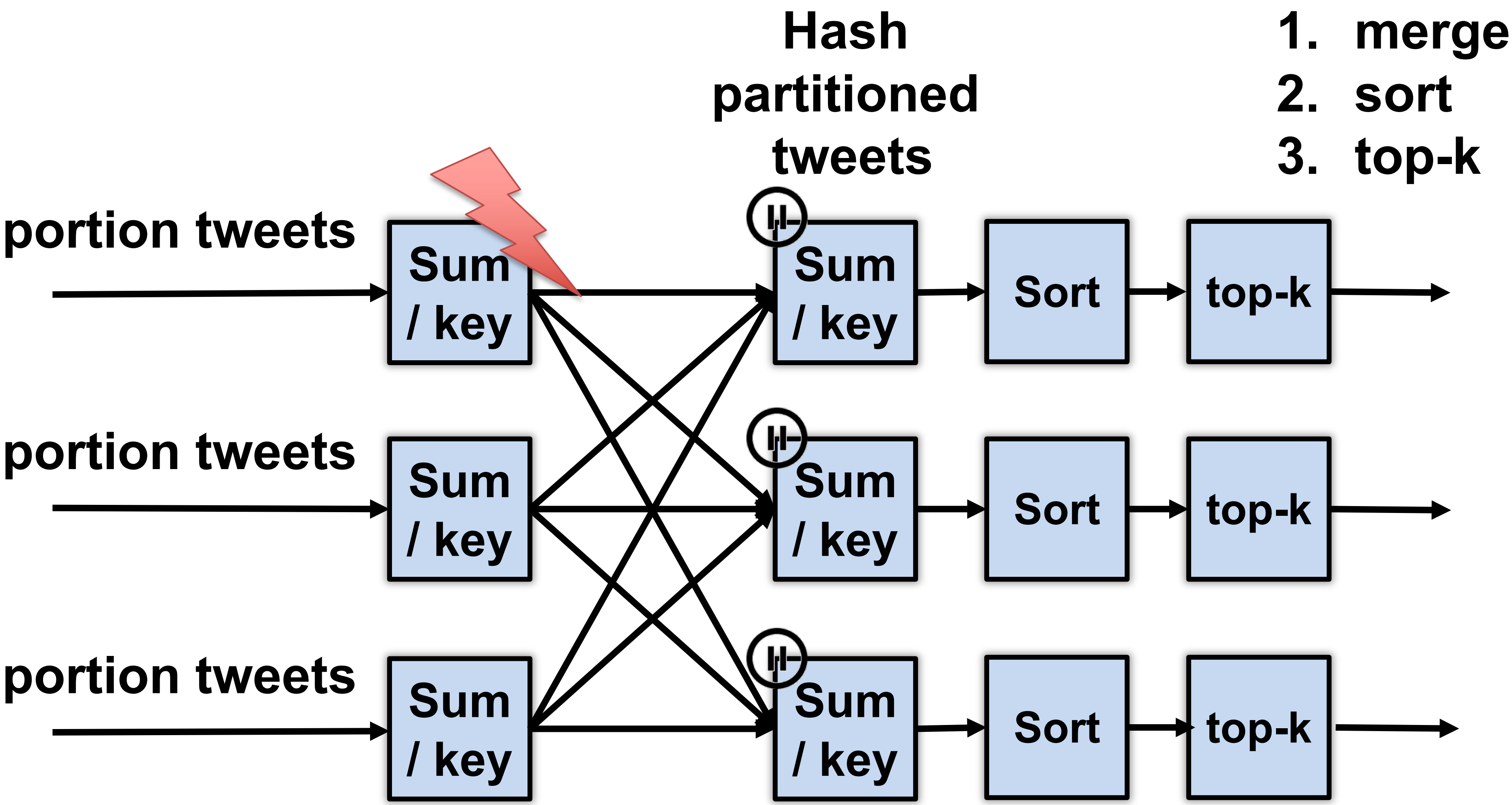
- Compute trending keywords
  - E.g.,



Can parallelize joins



Parallelization complicates fault-tolerance



Popular Streaming Frameworks

Various fault tolerance mechanisms:

1. Record acknowledgement (Storm)
2. Micro-batches (Spark Streaming, Storm Trident)
3. Transactional updates (Google Cloud dataflow)
4. Distributed snapshots (Flink)

Popular Streaming Frameworks

1. **Record acknowledgement (Storm)**
   - At least once semantics
   - Ensure each input “fully processed”
   - Track every processed tuple over the DAG, propagate ACKs upwards to the input source of data
   - Cons: Apps need to deal with duplicate or out-of-order tuples
2. Micro-batches (Spark Streaming, Storm Trident)
3. Transactional updates (Google Cloud dataflow)
4. Distributed snapshots (Flink)

Popular Streaming Frameworks

1. Record acknowledgement (Storm)
2. **Micro-batches (Spark Streaming, Storm Trident)**
   – Each micro-batch may succeed or fail
   – On failure, recompute the micro-batch
   – Use lineage to track dependencies
   – Checkpoint state to support failure recovery
3. Transactional updates (Google Cloud dataflow)
4. Distributed snapshots (Flink)

Popular Streaming Frameworks

1. Record acknowledgement (Storm)
2. Micro-batches (Spark Streaming, Storm Trident)
3. **Transactional updates (Google Cloud dataflow)**
   – Treat every processed record as a transaction, committed upon processing
   – On failure, replay the log to restore a consistent state and replay lost records
4. Distributed snapshots (Flink)

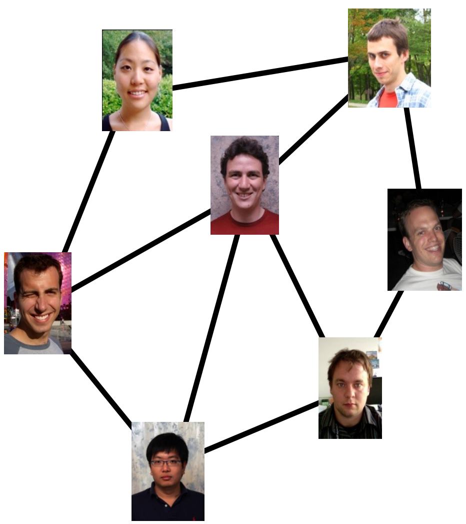
Popular Streaming Frameworks

1. Record acknowledgement (Storm)
2. Micro-batches (Spark Streaming, Storm Trident)
3. Transactional updates (Google Cloud dataflow)
4. **Distributed snapshots (Flink)**
   – Take system-wide consistent snapshot (algo is a variation of Chandy-Lamport)
   – Snapshot periodically
   – On failure, recover the latest snapshot and rewind the stream source to snapshot point, then replay inputs

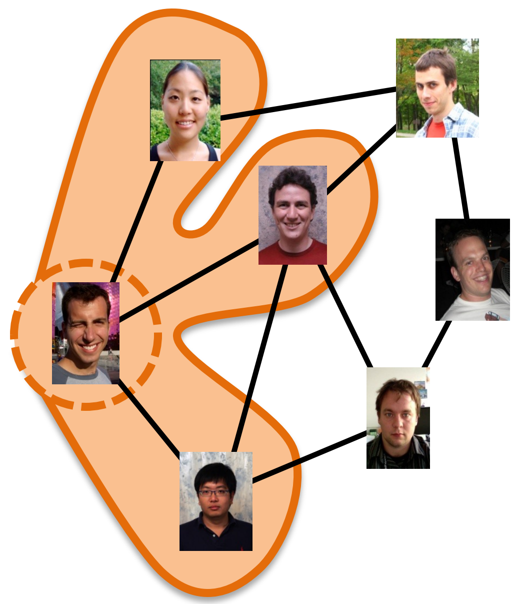
Graph-Parallel Computation

Properties of Graph Parallel Algorithms

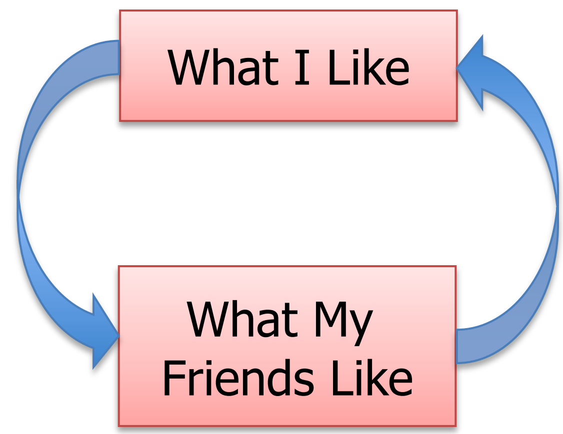
Dependency Graph



Factored Computation



Iterative Computation



ML Tasks Beyond Data-Parallelism

Data-Parallel | Graph-Parallel

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

?

Graphical Models
Gibbs Sampling
Belief Propagation
Variational Opt.

Semi-Supervised Learning
Label Propagation
CoEM

Collaborative Filtering
Tensor Factorization

Graph Analysis
PageRank
Triangle Counting

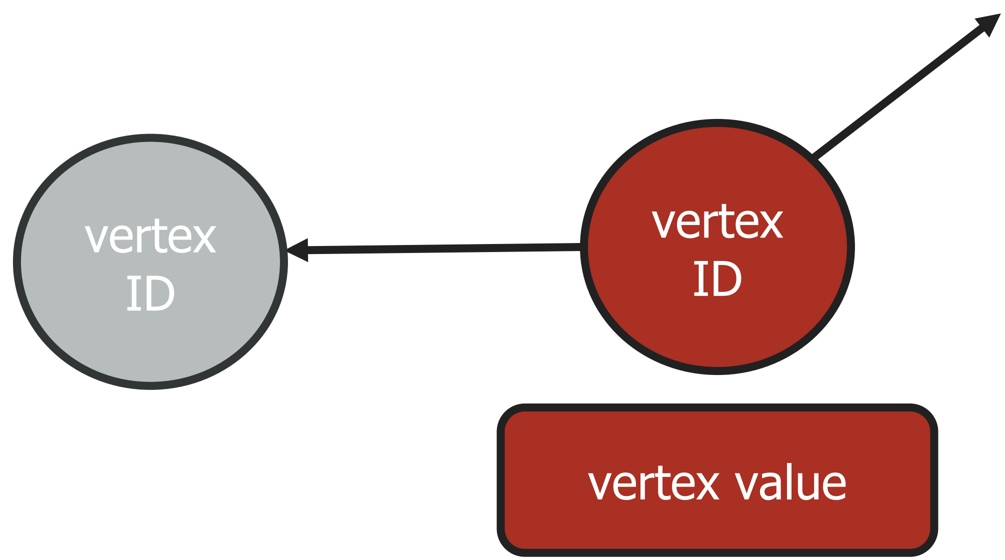
Pregel: Bulk Synchronous Parallel

Let's slightly rethink the MapReduce model for processing **graphs**

- Vertices
- "Edges" are really messages

Compare to MapReduce keys → values?

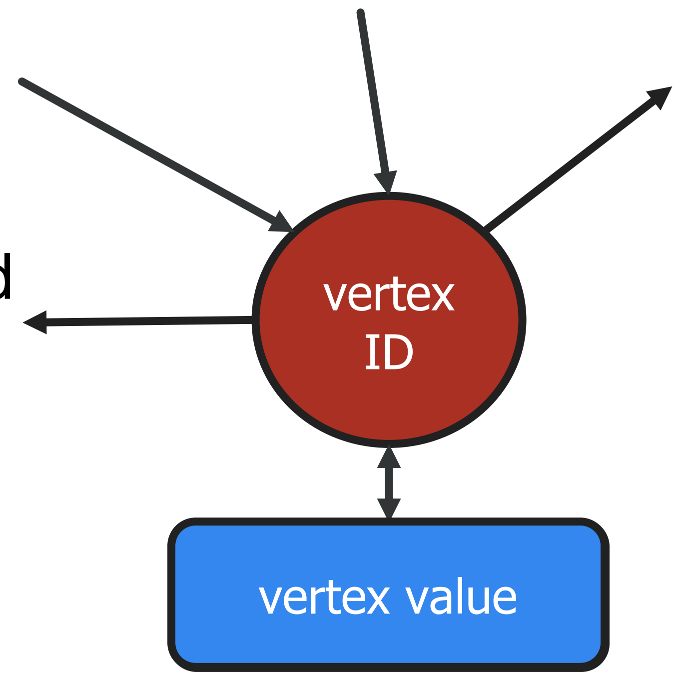
"Think like a vertex"



The Basic Pregel Execution Model

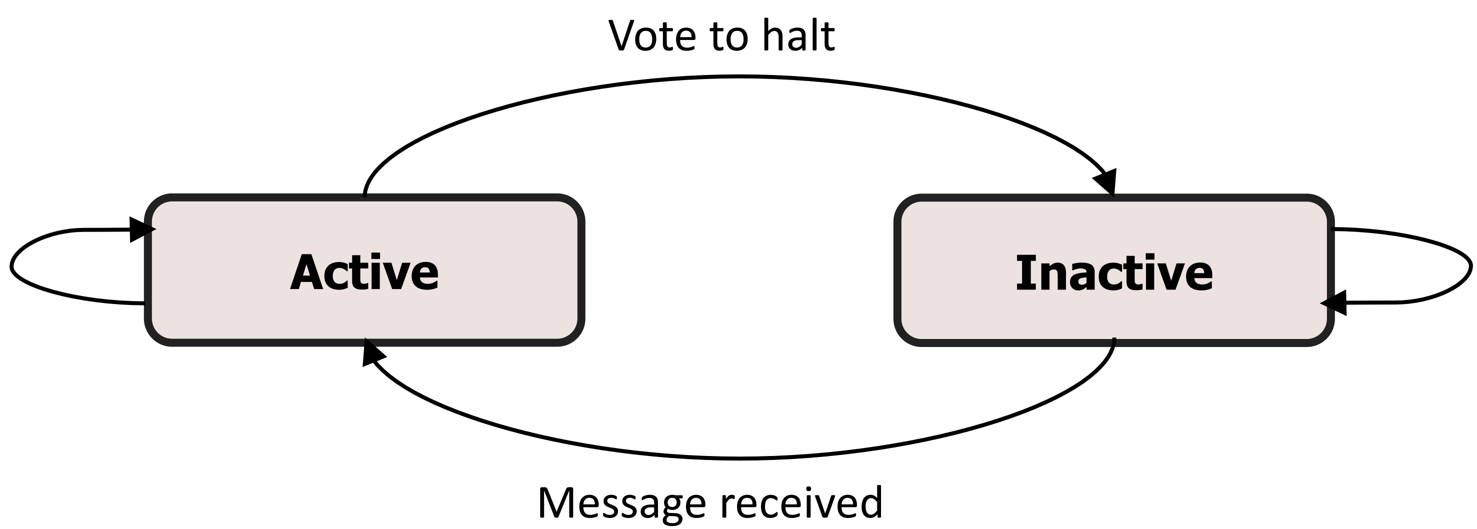
A sequence of *supersteps*, for each vertex V
At superstep S:

- Compute in parallel at each V
  - Read messages sent to V in superstep S-1
  - Update value / state
  - Optionally change topology
- Send messages
- Synchronization
  - Wait till all communication is finished



Termination Test

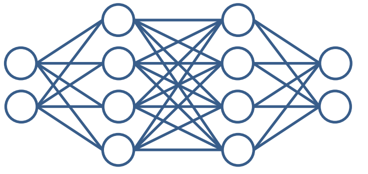
- Based on every vertex voting to halt
  – Once a vertex deactivates itself it does no further work unless triggered externally by receiving a message
- Algorithm terminates when all vertices are simultaneously inactive



Distributed Machine Learning

Machine learning (ML)

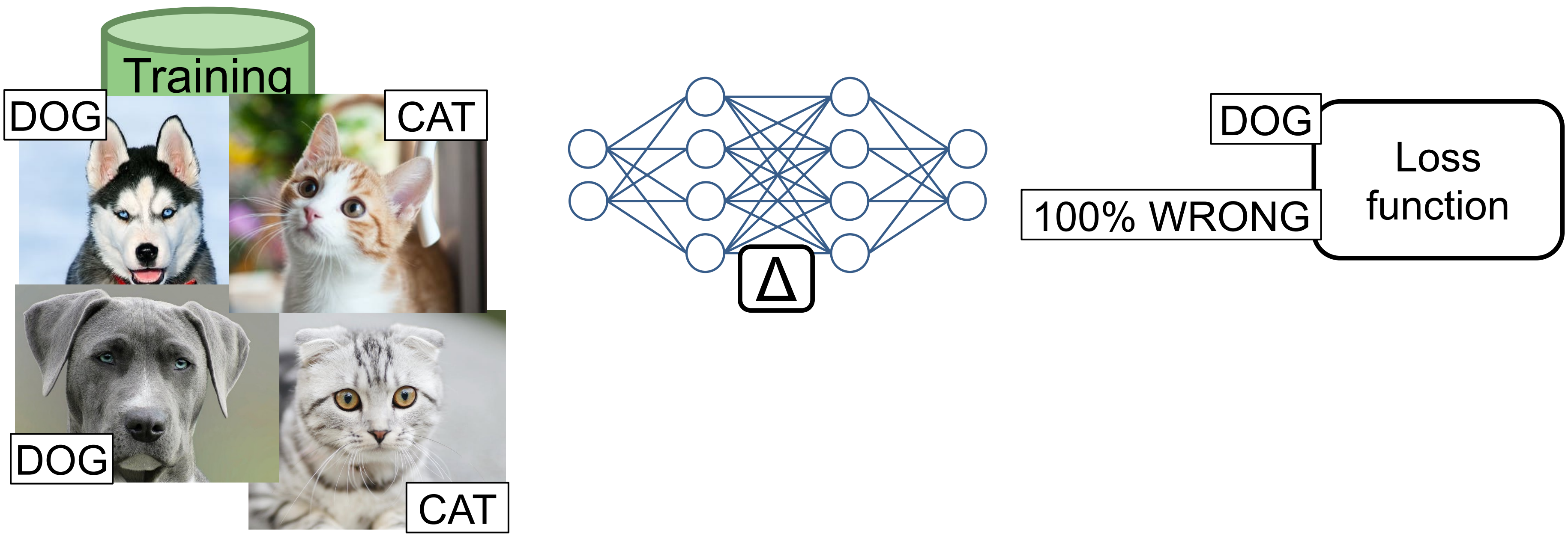
ML algorithms can improve automatically through experience (data)



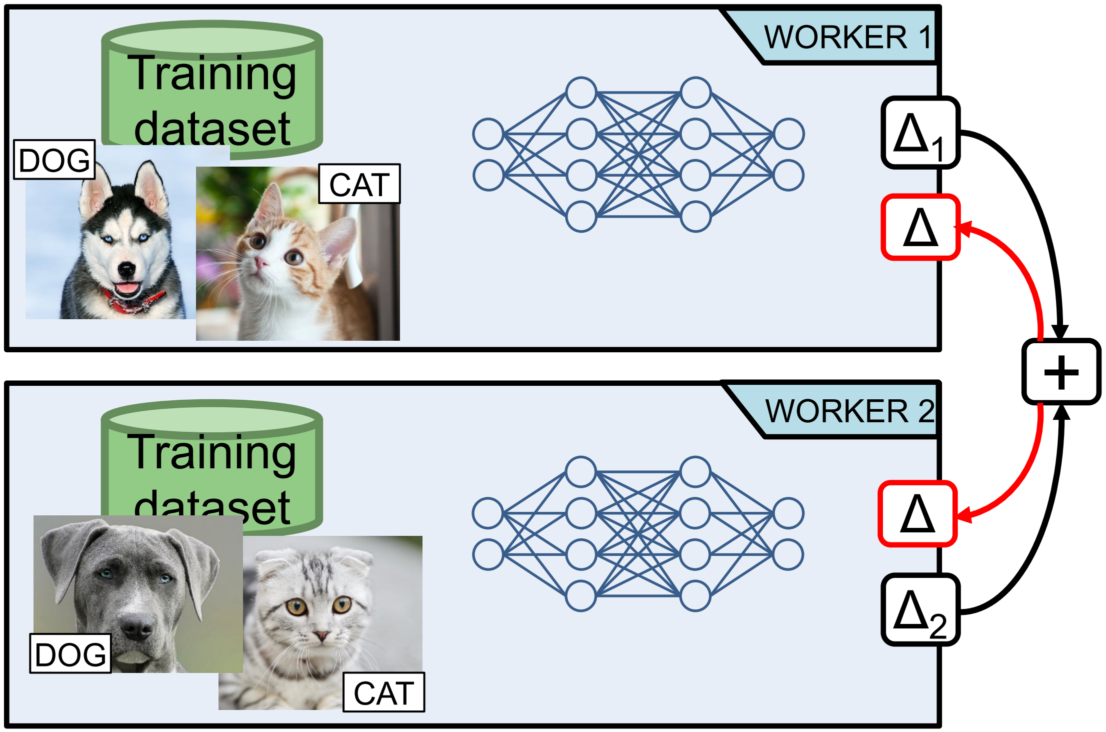
- Most common approaches
  - **Supervised learning**: train the model first, then use it
  - **Unsupervised learning**: the model learns by itself
  - **Reinforcement learning (RL)**: model learns while doing

Training	**Inference (or model serving)**
Feed the ML model data, so that it can learn how to make decisions	ML model in use, to process live data

ML training



Distributed ML training
Data parallel



Mini-batch

Amount of data processed by a single worker during 1 iteration

Global batch

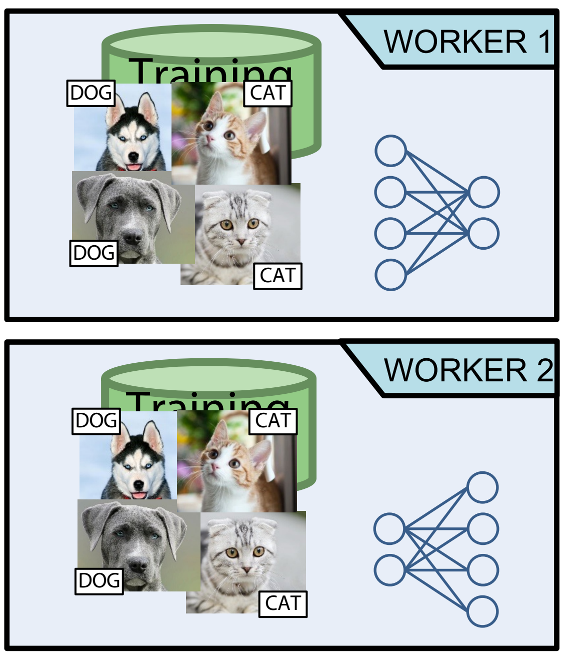
Amount of data processed by all workers during 1 iteration

Stochastic gradient descent (SGD)

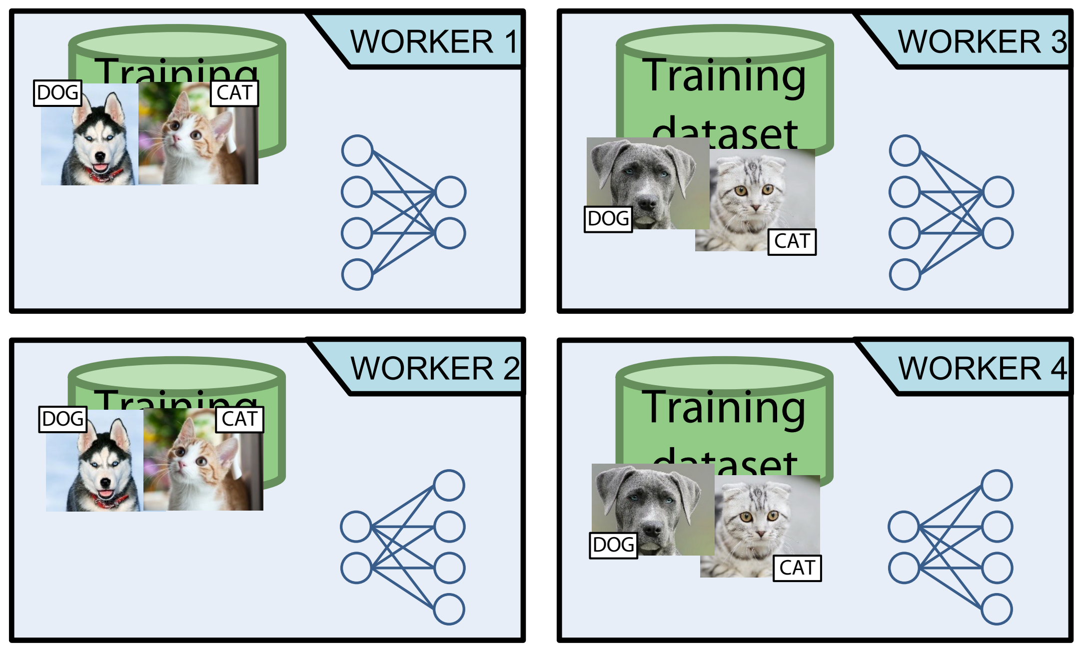
$$\omega := \omega - \frac{\eta}{n} \sum_{i=1}^{n} \nabla Q_i(\omega)$$

Distributed ML training
Model parallel or hybrid

Model parallel



Hybrid model-data parallel



Weak scaling and strong scaling

Weak scaling

- Fixed local batch size per-worker fixed
- More workers can process a larger global batch in one iteration
- Same iteration time, fewer iterations
- Same data transfers at each iteration
- Time to accuracy does not scale linearly with the number of workers

Strong scaling

- Fixed global batch size
- With more workers, the local batch size per-worker decreases
- Reduced iteration time (for computation)
- Same data transfers at each iteration
- More frequent synchronizations among workers (**more network traffic**)

When the network is the bottleneck

- Compute accelerators performance improvements have so far outpaced network bandwidth increases
- Newer, larger DNN models spend more time on communication