Big Data II: Stream Processing and Coordination
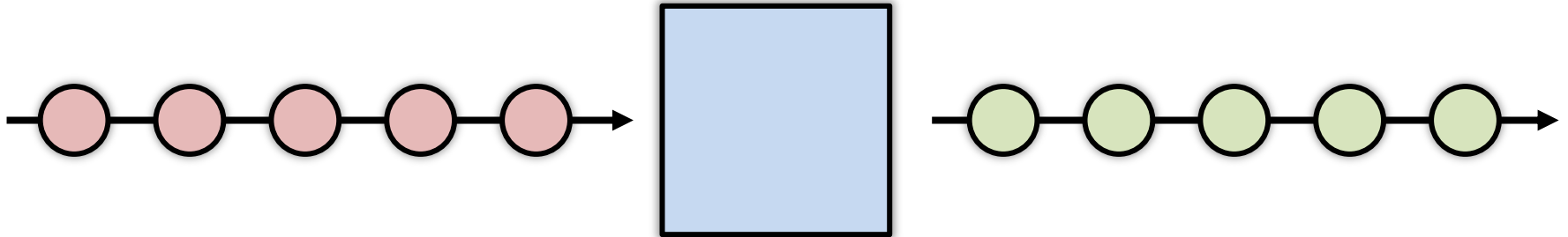
جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

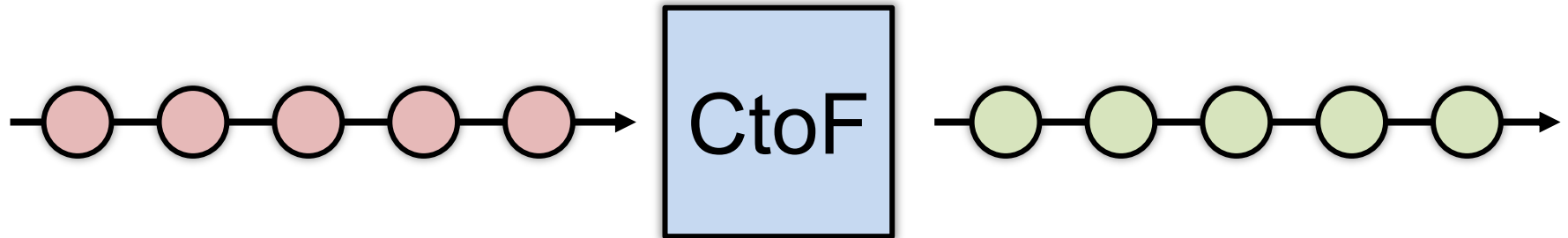CS 240: Computing Systems and Concurrency
Lecture 24

Marco Canini
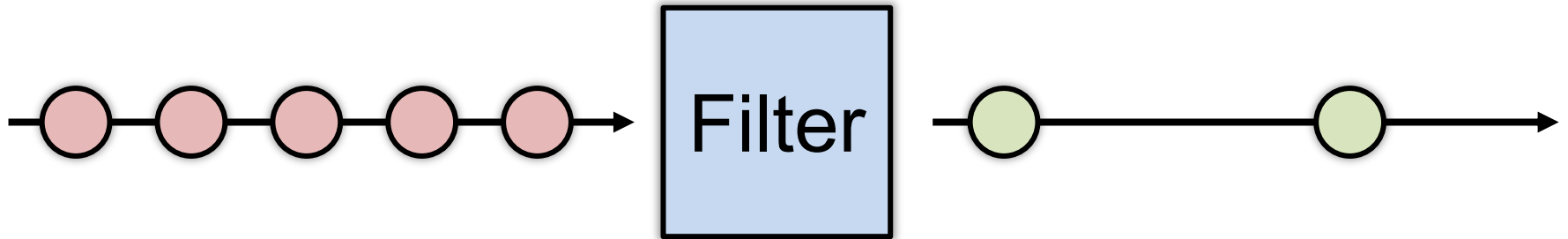
Simple stream processing

- Single node

  - Read data from socket

  - Process

  - Write output

Examples: Stateless conversion

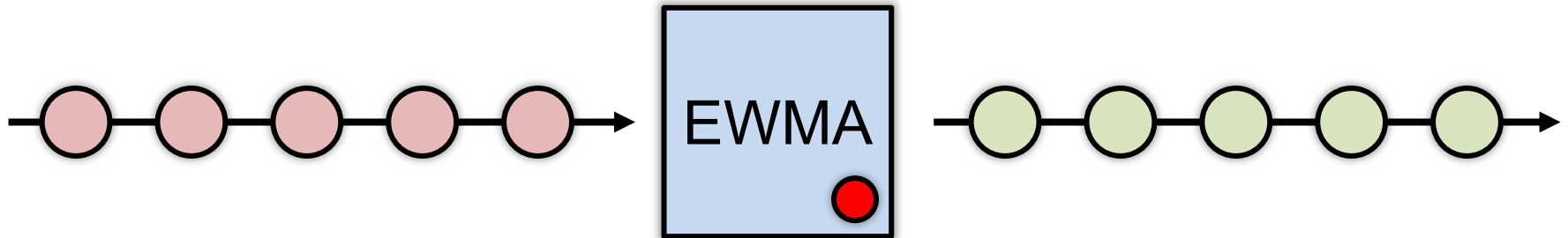- Convert Celsius temperature to Fahrenheit

    – Stateless operation: **emit** (input * 9 / 5) + 32

Examples: Stateless filtering

Filter

- 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)
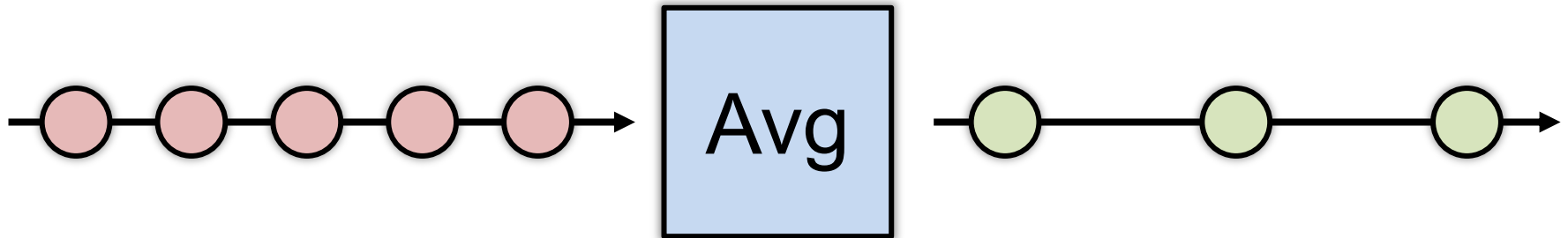
- E.g., Average value per window

  – Window can be # elements (10) or time (1s)

  – Windows can be fixed (every 5s)

  – Windows can be "sliding" (5s window every 1s)
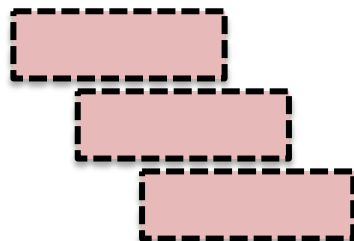
Stream processing as chain
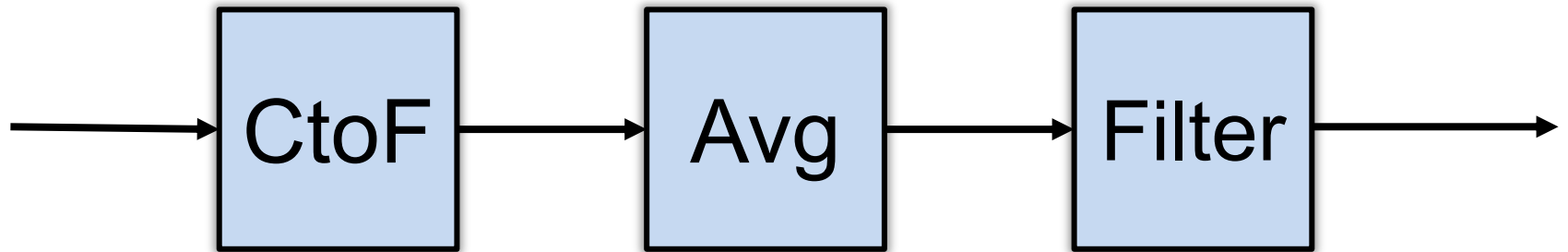
Stream processing as directed graph

sensor type 1 → CtoF → Avg → Filter → alerts

sensor type 2 → KtoF → Avg

Avg → storage
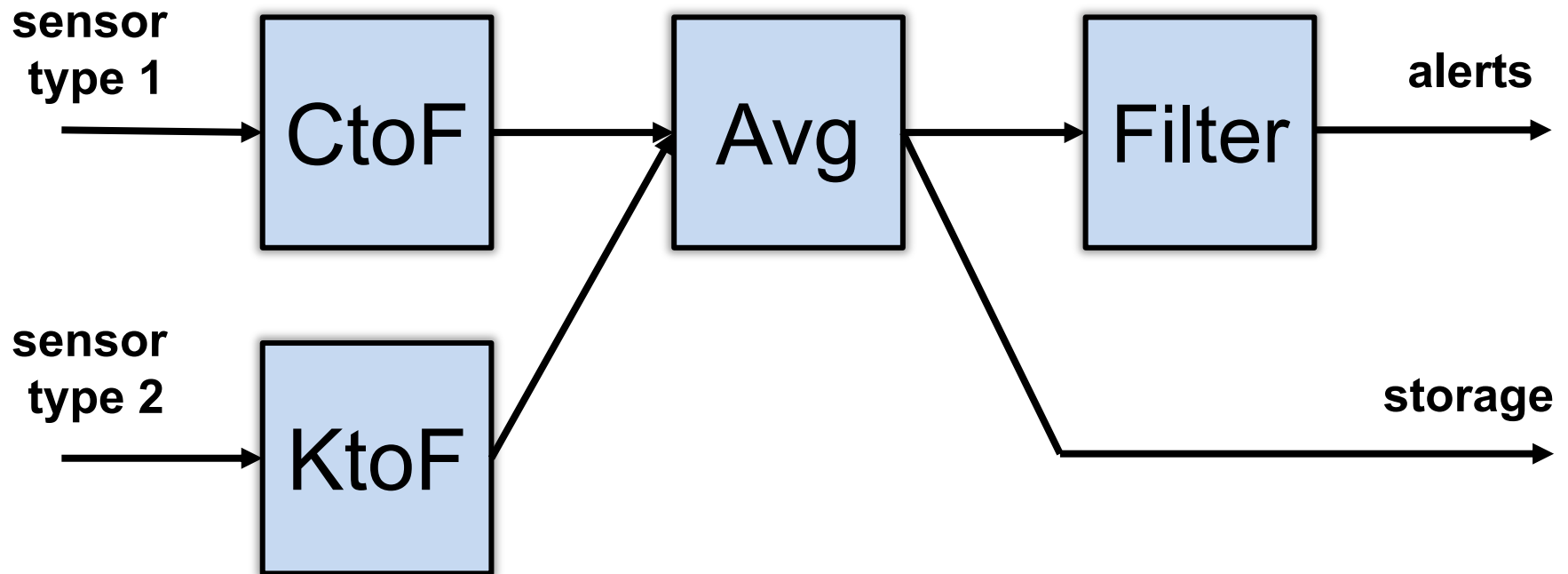
Enter "BIG DATA"

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"

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"
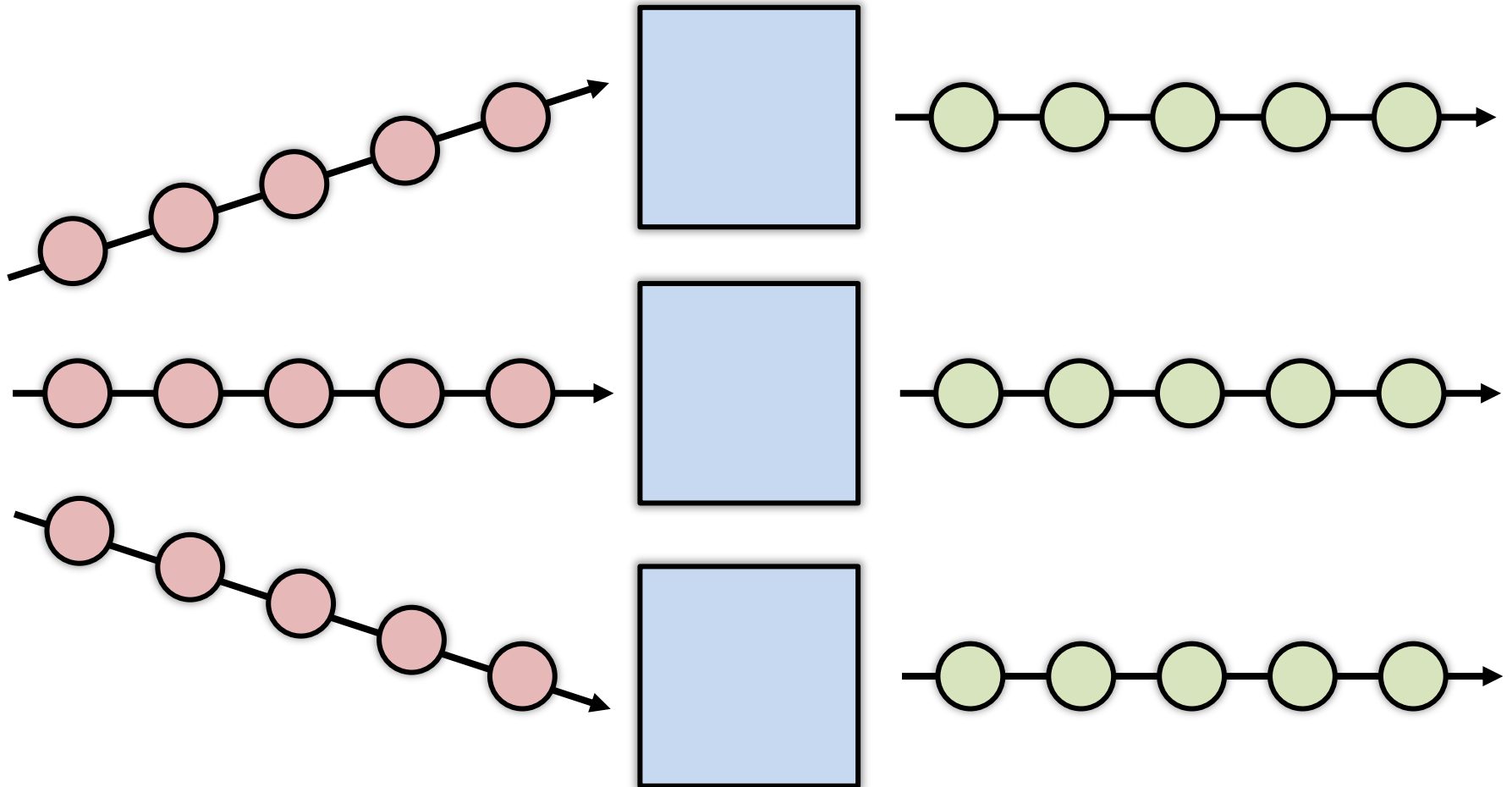
Stateless operations: trivially parallelized
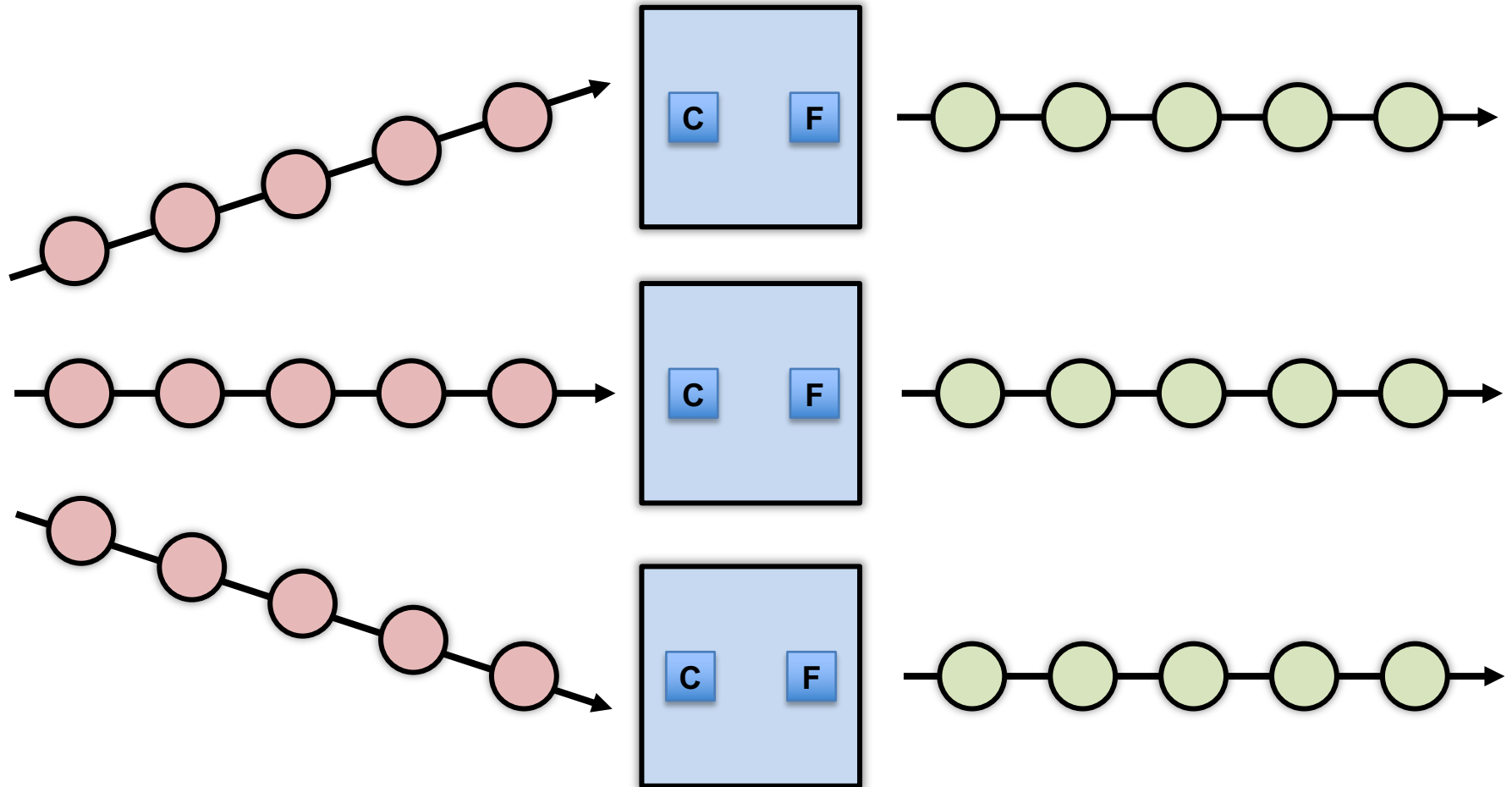
State complicates parallelization

- Aggregations:
  - Need to join results across parallel computations
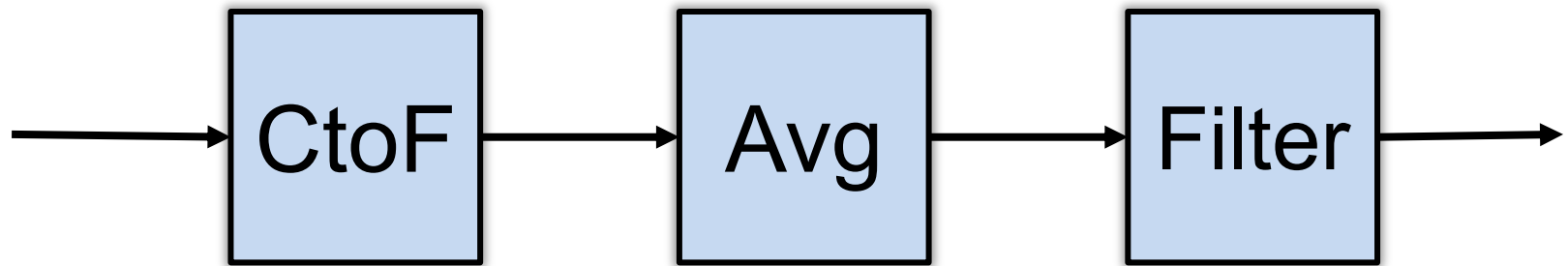
State complicates parallelization

- Aggregations:
  - Need to join results across parallel computations
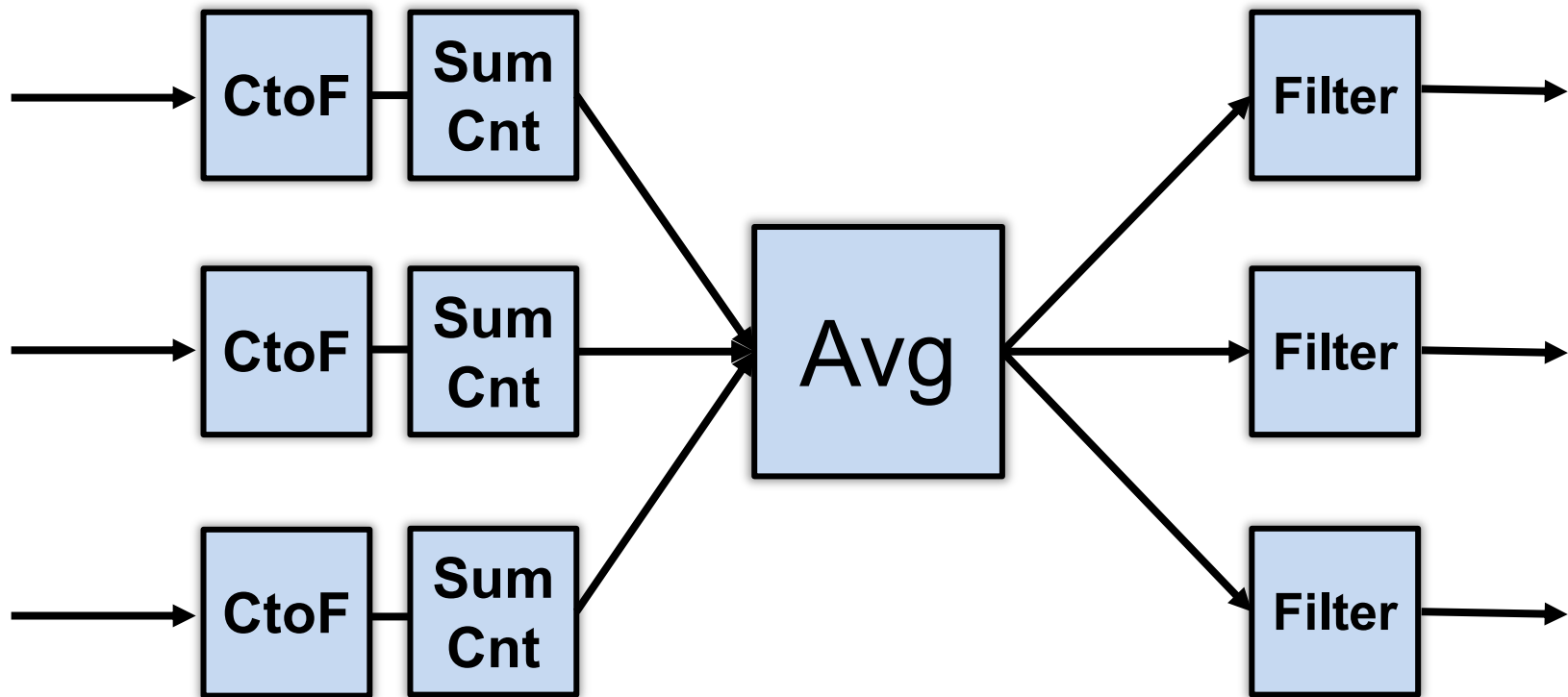
Parallelization complicates fault-tolerance

- Aggregations:
  - Need to join results across parallel computations
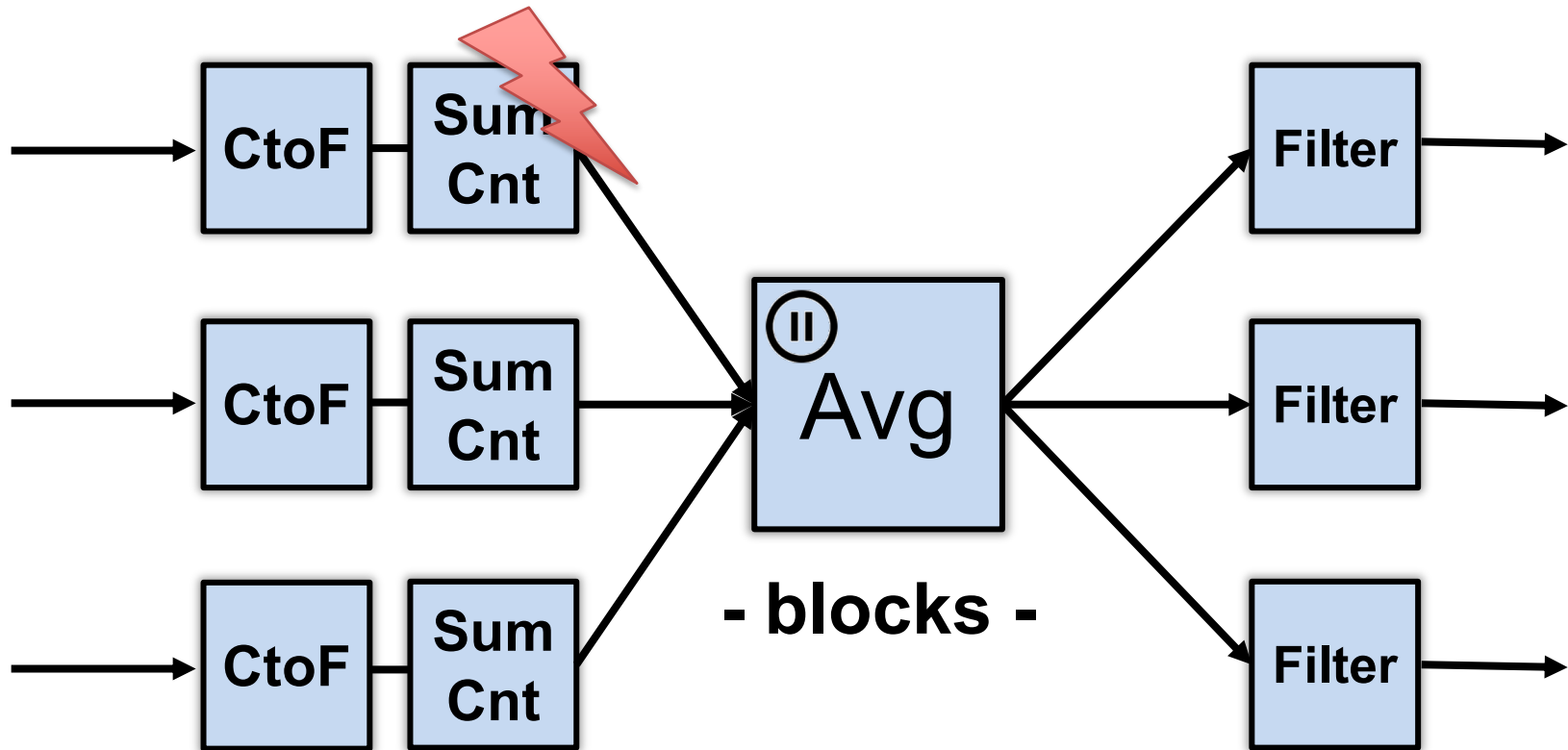
Can parallelize joins

- Compute trending keywords
  - E.g.,

portion tweets → **Sum / key**

portion tweets → **Sum / key** → **Sum / key** → **Sort** → **top-k** →

portion tweets → **Sum / key**

- blocks -

Can parallelize joins

Hash partitioned tweets

1. merge
2. sort
3. top-k

portion tweets → Sum / key → Sum / key → Sort → top-k →

portion tweets → Sum / key → Sum / key → Sort → top-k →

portion tweets → Sum / key → Sum / key → Sort → top-k →
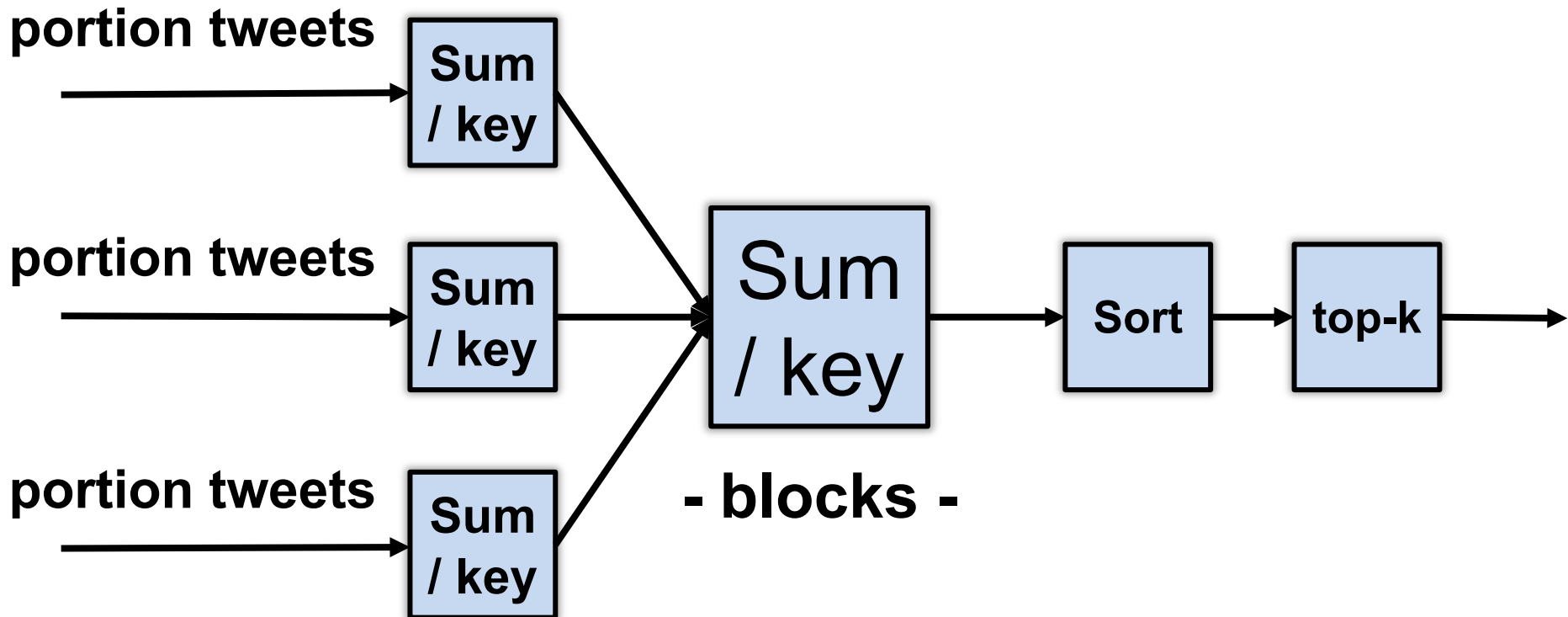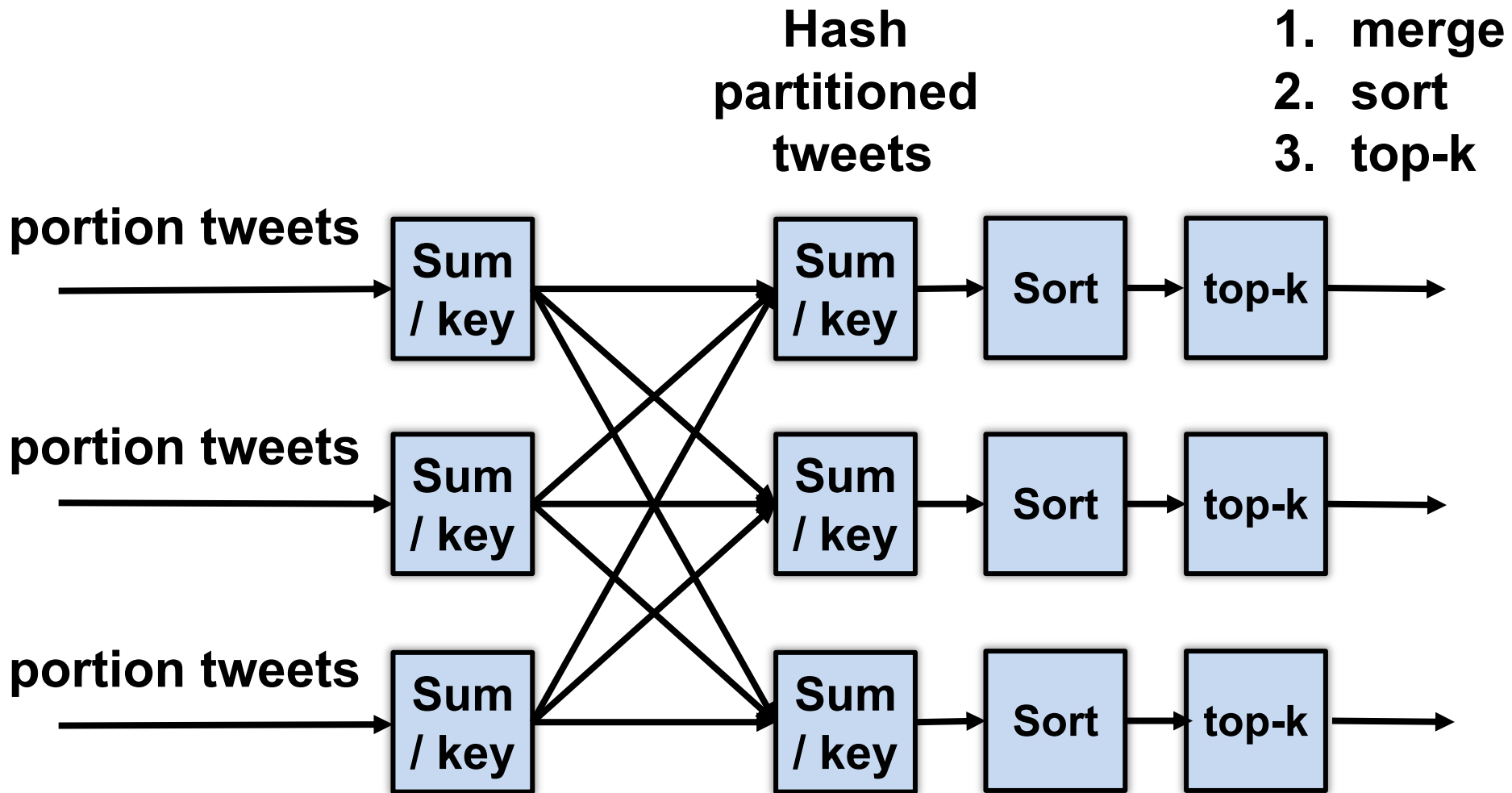
Parallelization complicates fault-tolerance
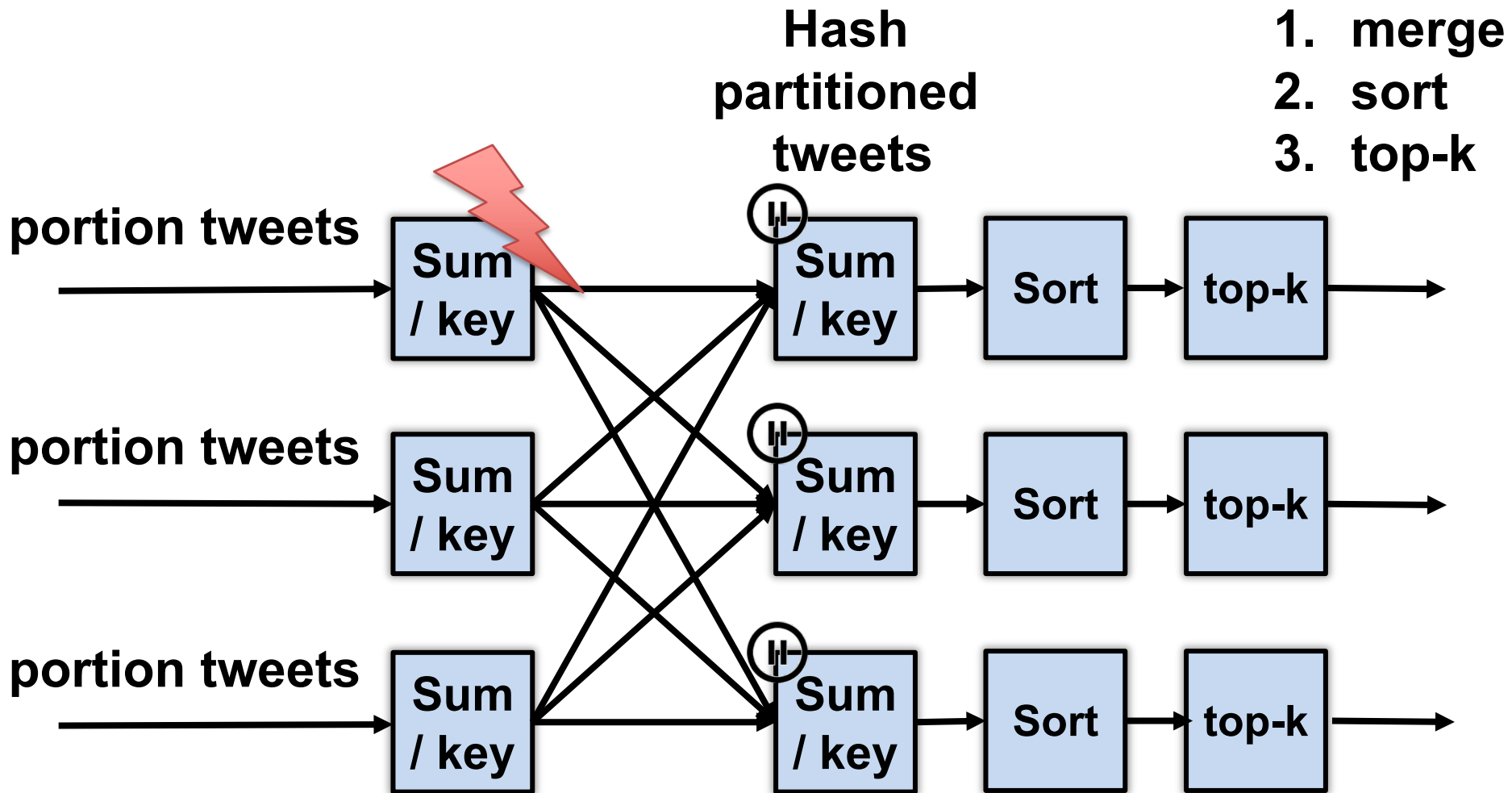
A Tale of Four Frameworks

1. Record acknowledgement (Storm)

2. Micro-batches (Spark Streaming, Storm Trident)

3. Transactional updates (Google Cloud dataflow)

4. Distributed snapshots (Flink)

Apache Storm

- Architectural components
  - Data: streams of tuples, e.g., Tweet = <Author, Msg, Time>
  - Sources of data: "spouts"
  - Operators to process data: "bolts"
  - Topology: Directed graph of spouts & bolts
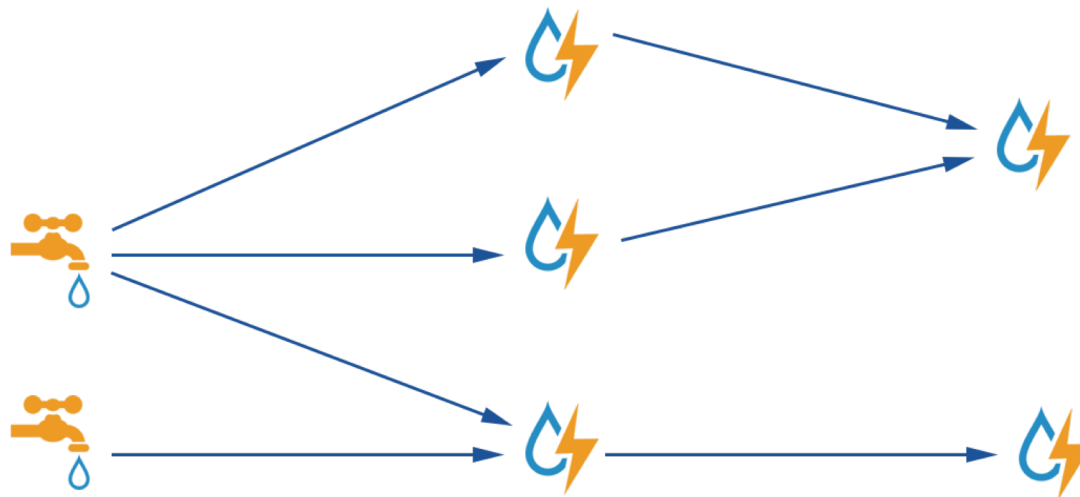
Apache Storm: Parallelization

- Multiple processes (tasks) run per bolt

- Incoming streams split among tasks

  - **Shuffle Grouping:** Round-robin distribute tuples to tasks

  - **Fields Grouping:** Partitioned by key / field

  - **All Grouping:** All tasks receive all tuples (e.g., for joins)
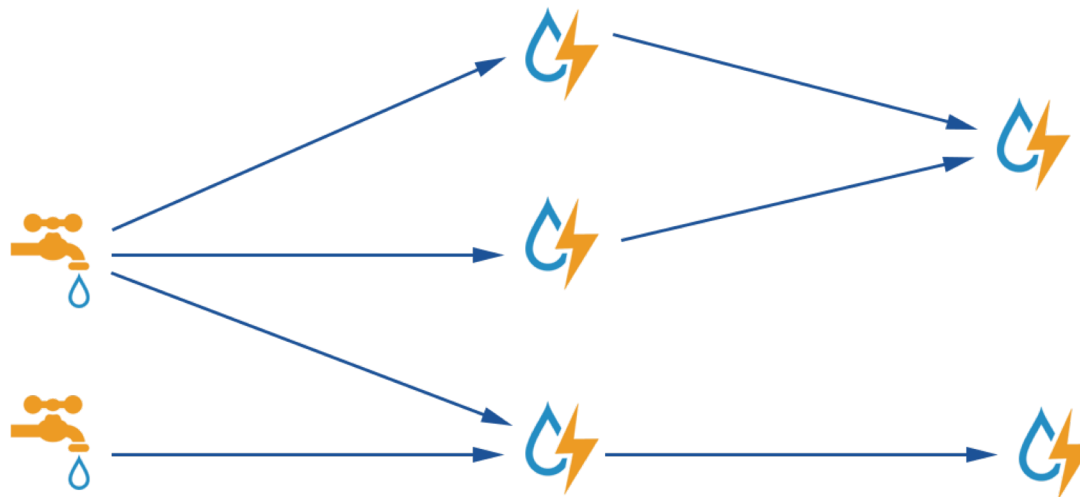
Fault tolerance via record acknowledgement (Apache Storm – at least once semantics)

- Goal: Ensure each input "fully processed"

- Approach: DAG / tree edge tracking

    – Record edges that get created as tuple is processed

    – Wait for all edges to be marked done

    – Inform source (spout) of data when complete; otherwise, they resend tuple

- Challenge: "at least once" means:

    – Bolts can receive tuple > once

    – Replay can be out-of-order

    – ... application needs to handle
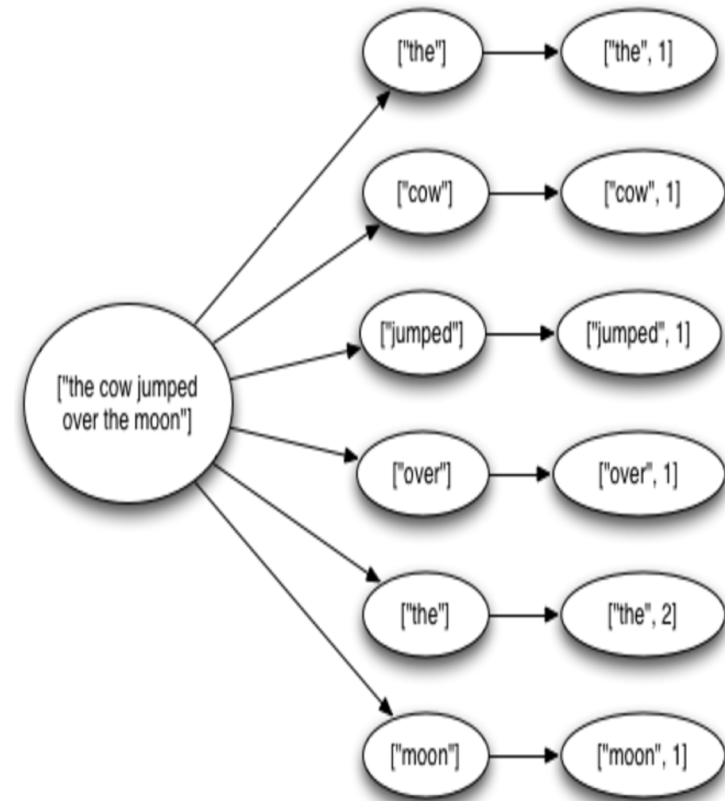
Apache Spark Streaming:
Discretized Stream Processing

- Split stream into series of small, atomic batch jobs (each of X seconds)

- Process each individual batch using Spark "batch" framework

- Emit each micro-batch result

  - RDD = "Resilient Distributed Data"

live data stream

Spark Streaming

batches of X seconds

Spark

processed results
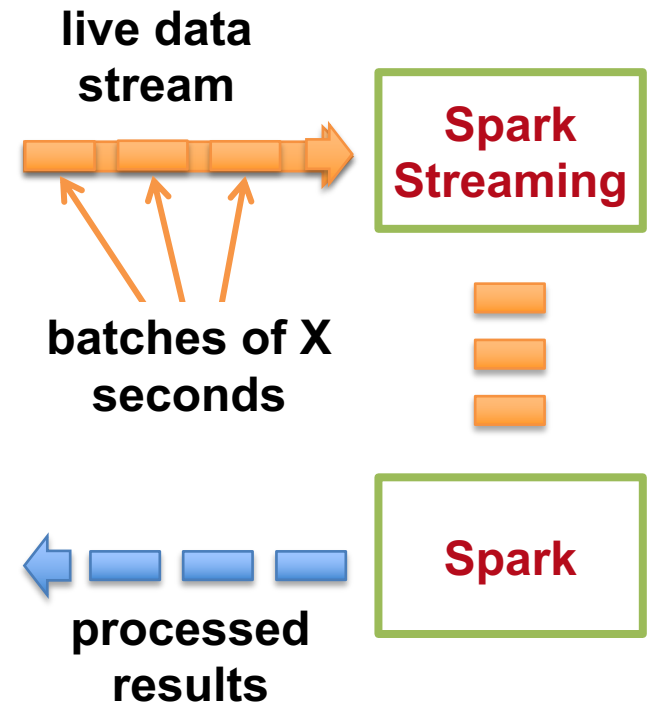
Fault tolerance via micro batches
(Apache Spark Streaming, Storm Trident)

- Can build on batch frameworks (Spark) and tuple-by-tuple (Storm)
  - Tradeoff between throughput (higher) and latency (higher)

- Each micro-batch may succeed or fail
  - Original inputs are replicated (memory, disk)
  - At failure, latest micro-batch can be simply recomputed (trickier if stateful)

- DAG is a pipeline of transformations from micro-batch to micro-batch
  - Lineage info in each RDD specifies how generated from other RDDs

- To support failure recovery:
  - Occasionally checkpoints RDDs (state) by replicating to other nodes
  - To recover: another worker (1) gets last checkpoint, (2) determines upstream dependencies, then (3) starts recomputing using those usptream dependencies starting at checkpoint (downstream might filter)

Fault Tolerance via transactional updates (Google Cloud Dataflow)

- Computation is long-running DAG of continuous operators

- For each intermediate record at operator

  – Create commit record including input record, state update, and derived downstream records generated

  – Write commit record to transactional log / DB

- On failure, replay log to

  – Restore a consistent state of the computation

  – Replay lost records (further downstream might filter)

- Requires: High-throughput writes to distributed store

Fault Tolerance via distributed snapshots (Apache Flink)

- Rather than log each record for each operator, take system-wide snapshots

- Snapshotting:
  - Determine consistent snapshot of system-wide state (includes in-flight records and operator state)
  - Store state in durable storage

- Recover:
  - Restoring latest snapshot from durable storage
  - Rewinding the stream source to snapshot point, and replay inputs

- Algorithm is based on Chandy-Lamport distributed snapshots, but also captures stream topology

Fault Tolerance via distributed snapshots (Apache Flink)

- Use markers (barriers) in the input data stream to tell downstream operators when to consistently snapshot

data stream

newer records ← → older records

checkpoint barrier n   checkpoint barrier n-1   stream record (event)

part of checkpoint n+1   part of checkpoint n   part of checkpoint n-1

checkpoint barrier

operator   operator   operator

emit barrier n

1. align barriers   2. checkpoint state   3. emit barrier and continue
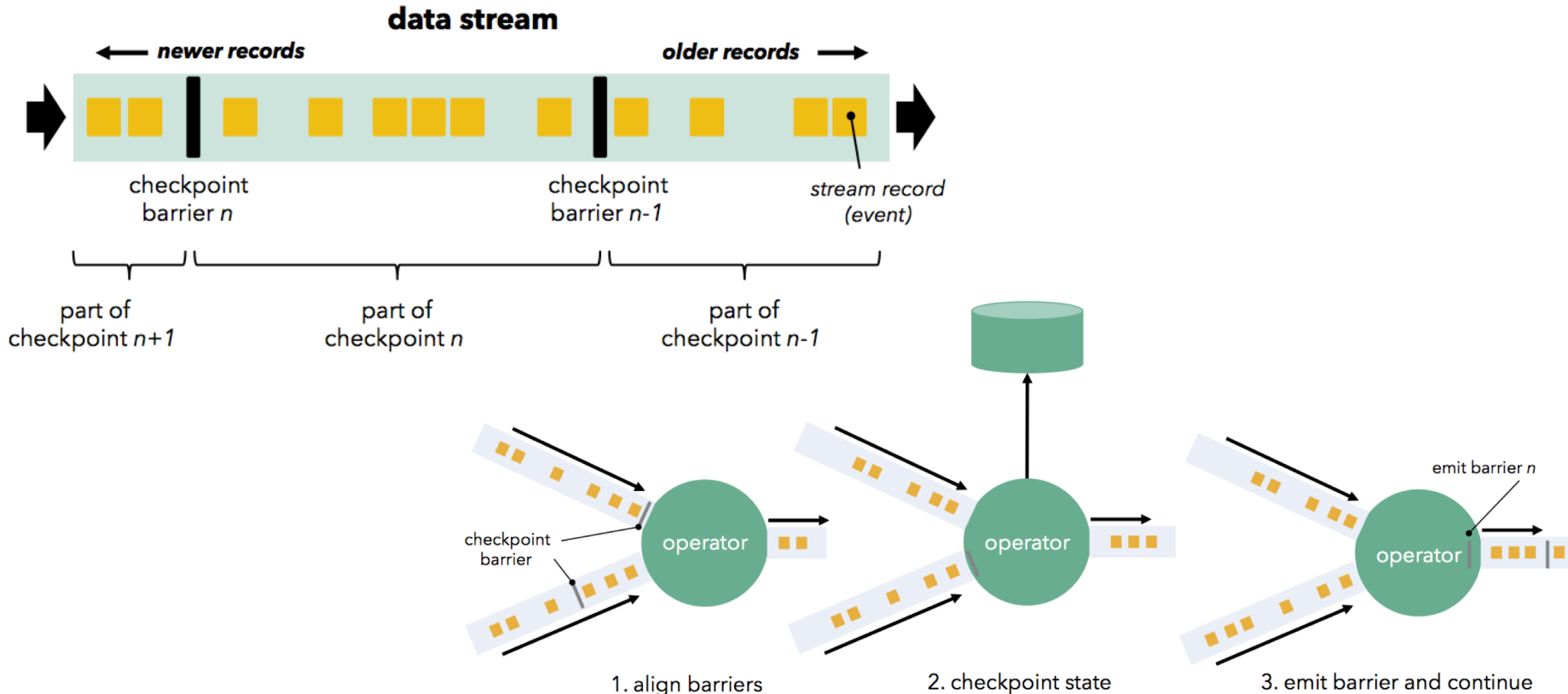
Optimizing stream processing

- Keeping system performant:

  - Careful optimizations of DAG

  - Scheduling: Choice of parallelization, use of resources

  - Where to place computation

  - …

- Often, many queries and systems using same cluster concurrently: **"Multi-tenancy"**

Coordination

Practical consensus

Needs of distributed apps

- Lots of apps need various coordination primitives
  - Leader election
  - Group membership
  - Locks
  - Leases

- Common requirement is consensus but we'd like to avoid duplication
  - Duplicating is bad and duplicating poorly even worse
  - Maintenance?

How do we go about coordination?

- One approach

    – For each coordination primitive build a specific service

- Some recent examples

    – Chubby, Google [*Burrows et al, USENIX OSDI, 2006*]

        - Lock service

    – Centrifuge, Microsoft [*Adya et al, USENIX NSDI, 2010*]

        - Lease service

How do we go about coordination?

- Alternative approach

  – A coordination service

  – Develop a set of lower level primitives (i.e., an API) that can be used to implement higher-level coordination services

  – Use the coordination service API across many applications

- Example: Apache Zookeeper

ZooKeeper

- A "Coordination Kernel"

  - Provides a file system abstraction and API that enables realizing several coordination primitives

    - Group membership
    - Leader election
    - Locks
    - Queueing
    - Barriers
    - Status monitoring

Data model

- In brief, it's a file system with a simplified API

- Only whole file reads and writes

  - No appends, inserts, partial reads

- Files are znodes; organized in hierarchical namespace

- Payload not designed for application data storage but for application metadata storage

- Znodes also have associated version counters and some metadata (e.g., flags)

Semantics

- CAP perspective: Zookeeper is CP
  - It guarantees consistency
  - May sacrifice availability under system partitions
    - strict quorum based replication for writes
- Consistency (safety)
  - *FIFO client order:* all client requests are executed in order sent by client
    - Matters for asynchronous calls
  - *Linearizable writes:* all writes are linearizable
  - *Serializable reads:* reads can be served locally by any server, which may have a stale value

Types of znodes

- Regular znodes
  - May have children
  - Explicitly deleted by clients
- Ephemeral znodes
  - May not have children
  - Disappear when deleted or when creator terminates
    - Session termination can be deliberate or due to failure
- Sequential flag
  - Property of regular znodes
  - Children have strictly increasing integer appended to their names

Client API

- *create(znode, data, flags)*
  - *Flags denote the type of the znode:*
    - *REGULAR, EPHEMERAL, SEQUENTIAL*
  - *znode must be addressed by giving a full path in all operations (e.g., '/app1/foo/bar')*
  - *returns znode path*

- *delete(znode, version)*
  - *Deletes the znode if the version is equal to the actual version of the znode*
  - *set version = -1 to omit the conditional check (applies to other operations as well)*

Client API (cont'd)

- *exists(znode, watch)*

  – *Returns true if the znode exists, false otherwise*

  – *watch flag enables a client to set a watch on the znode*

  – *watch is a subscription to receive an information from the Zookeeper when this znode is changed*

  – *NB: a watch may be set even if a znode does not exist*

    - *The client will be then informed when a znode is created*

- *getData(znode, watch)*

  – *Returns data stored at this znode*

  – *watch is not set unless znode exists*

Client API (cont'd)

- *setData(znode, data, version)*
  - *Rewrites znode with data, if version is the current version number of the znode*
  - *version = -1 applies here as well to omit the condition check and to force setData*

- *getChildren(znode, watch)*
  - *Returns the set of children znodes of the znode*

- *sync()*
  - *Waits for all updates pending at the start of the operation to be propagated to the Zookeeper server that the client is connected to*

Some examples

Implementing consensus

- Propose(v)

  create("/c/proposal-", "v", SEQUENTIAL)

- Decide()

  C = getChildren("/c")

  Select znode *z in C* with smallest sequence number

  v' = getData(*z*)

  Decide v'

Simple configuration management

- Clients initialized with the name of znode

  - E.g., "/config"

```
config = getData("/config", TRUE)

while (true)

    wait for watch notification on "/config"

    config = getData("/config", TRUE)
```

Note: A client may miss some configuration, but it will always "refresh" when it realizes the configuration is stale

Group membership

- Idea: leverage ephemeral znodes

- Fix a znode "/group"

- Assume every process (client) is initialized with its own unique name and ID

  – What to do if there are no unique names?

joinGroup()

```
create("/group/" + name, [address,port], EPHEMERAL)
```

getMembers()

```
getChildren("/group", false)
```

Set to true to get notified about membership changes

A simple lock

Lock(filename)

```
1: create(filename, "", EPHEMERAL)

2: if create is successful

3:     return         //have lock

4: else

5:     getData(filename,TRUE)

6:     wait for filename watch

7:     goto 1:
```

Release(filename)

```
delete(filename)
```

Problems?

- Herd effect

  – If many clients wait for the lock they will all try to get it as soon as it is released

- Only implements exclusive locking

Simple Lock without Herd Effect

```
Lock(filename)

1:   myLock = create(filename + "/lock-", "", EPHEMERAL & SEQUENTIAL)

2:   C = getChildren(filename, false)

3:   if myLock is the lowest znode in C then return

4:   else

5:       precLock = znode in C ordered just before myLock

6:       if exists(precLock, true)

7:           wait for precLock watch

8:           goto 2:


Release(filename)

    delete(myLock)
```

Read/Write Locks

- The previous lock solves herd effect but makes reads block other reads

- How to do it such that reads always get the lock unless there is a concurrent write?

Read/Write Locks

Write Lock(filename)

1:   myLock = create(filename + "/write-", "", EPHEMERAL & SEQUENTIAL)

[...]   // same as simple lock w/o herd effect

Read Lock(filename)

1:   myLock = create(filename + "/read-", "", EPHEMERAL & SEQUENTIAL)

2:   C = getChildren(filename, false)

3:   if no **write** znodes lower than myLock in C then return

4:   else

5:        precLock = **write** znode in C ordered just before myLock

6:        if exists(precLock, true)

7:            wait for precLock watch

8:            goto 3:

Release(filename)

    delete(myLock)

A brief look inside

Zookeeper components
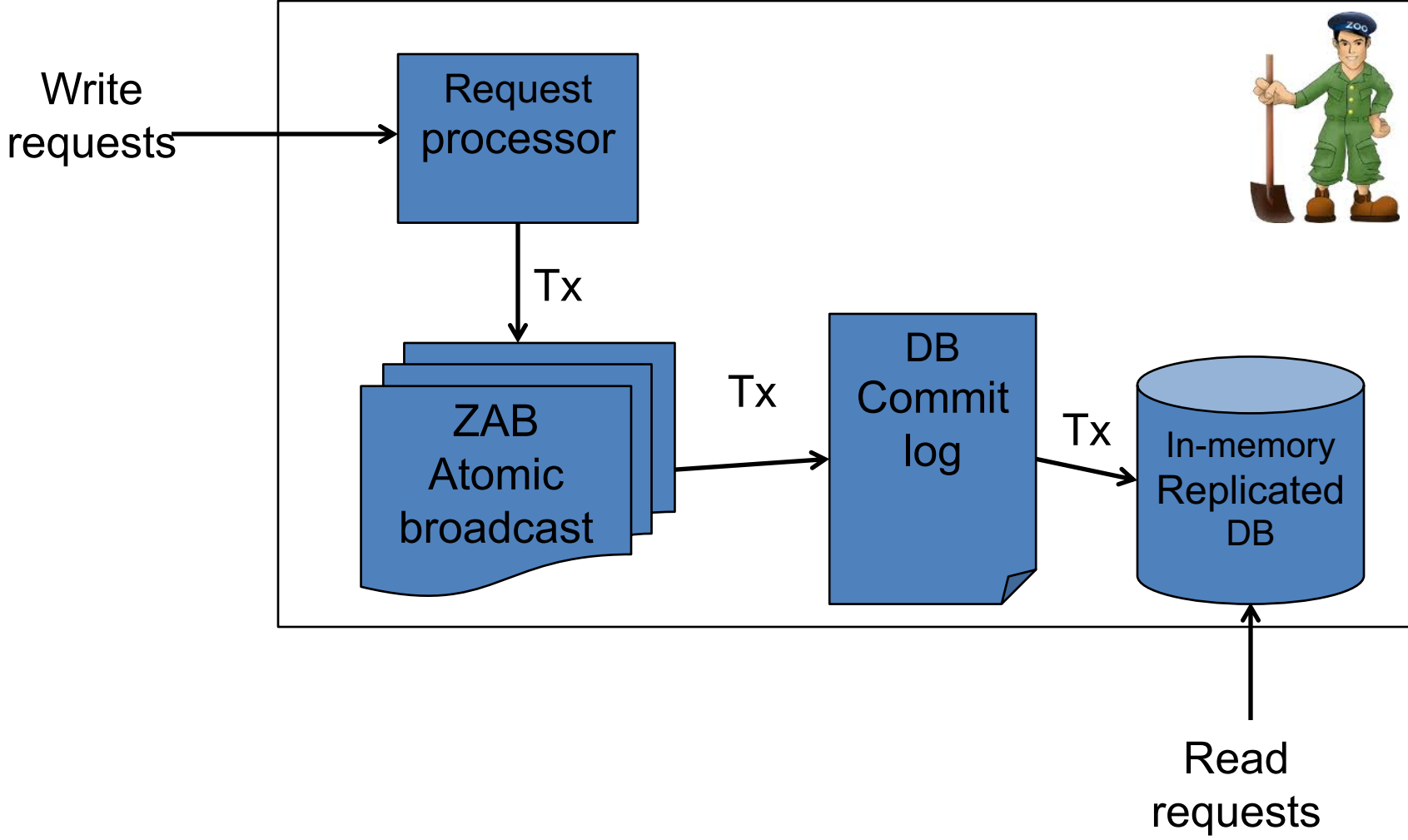
Zookeeper DB

- Fully replicated
  - To be contrasted with partitioning/placement in storage systems

- Each server has a copy of in-memory DB
  - Store the entire znode tree
  - Default max 1 MB per znode (configurable)

- Crash-recovery model
  - Commit log
  - + periodic snapshots of the database

ZAB: a very brief overview

- Used to totally order write requests

  – Relies on a quorum of servers ($f+1$ out of $2f+1$)

- ZAB internally elects leader replica

- Zookeeper adopts this notion of a leader

  – Other servers are followers

- All writes are sent by followers to the leader

  – Leader sequences the requests and invokes ZAB atomic broadcast

Request processor

- Upon receiving a write request

  - Leader calculates in what state system will be after the write is applied

  - Transforms the operation in a transactional update

- Transactional updates are then processed by ZAB, DB

  - Guarantees idempotency of updates to the DB originating from the same operation

- Idempotency important as ZAB may redeliver a message

That's all
Hope you enjoyed CS 240

Final lab (w/ bonus): Dec 4, 1PM

Review session: Dec 5, in class

Final exam: Dec 9, 9AM-12PM, Bldg 9: Lecture Hall 2 (room 2325)