Streaming

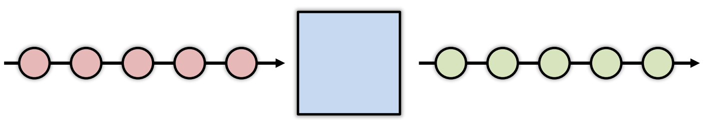
COS 518: Advanced Computer Systems
Lecture 11

Michael Freedman

What is streaming?

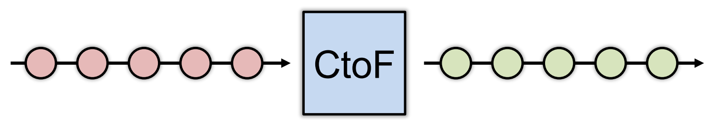
- Fast data!
- Fast processing!
- Lots of data!

Simple stream processing



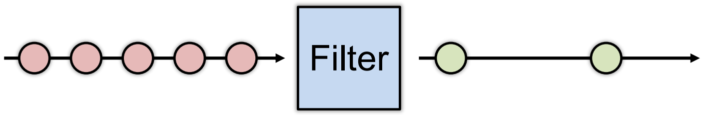
- Single node
  - Read data from socket
  - Process
  - Write output

Examples: Stateless conversion



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

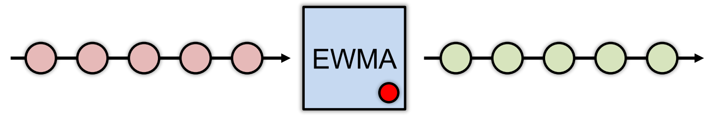
Examples: Stateless filtering



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

5

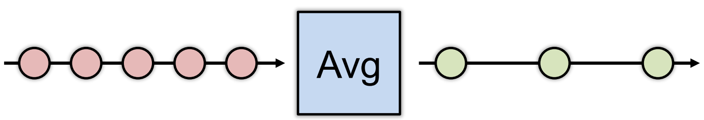
Examples: Stateful conversion



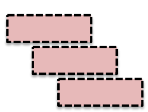
- Compute EWMA of Fahrenheit temperature
  - new_temp = α * (CtoF(input)) + (1- α) * last_temp
  - last_temp = new_temp
  - **emit** new_temp

6

Examples: Aggregation (stateful)



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



7

Enter “BIG DATA”

8

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

9

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
```

10

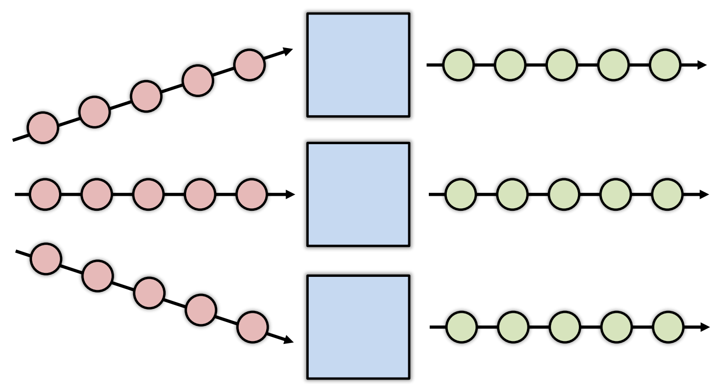
Scale "up"

Tuple-by-Tuple	Micro-batch
Lower Latency	Higher Latency
Lower Throughput	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.

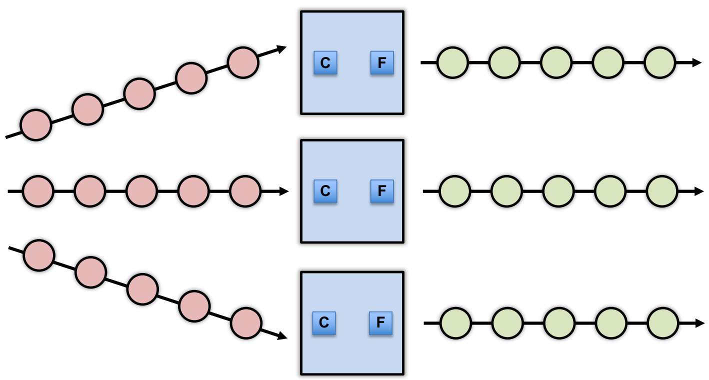
11

Scale "out"



12

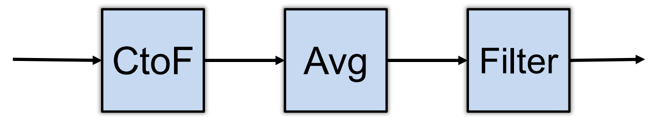
Stateless operations: trivially parallelized



13

State complicates parallelization

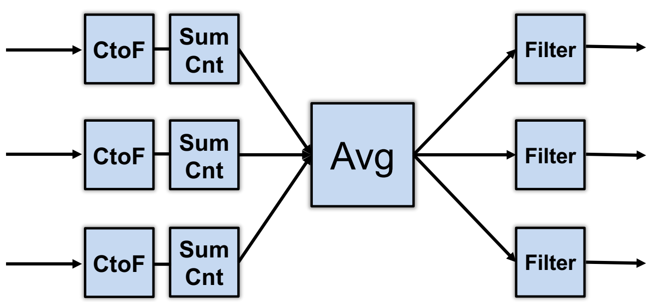
- Aggregations:
  – Need to join results across parallel computations



14

State complicates parallelization

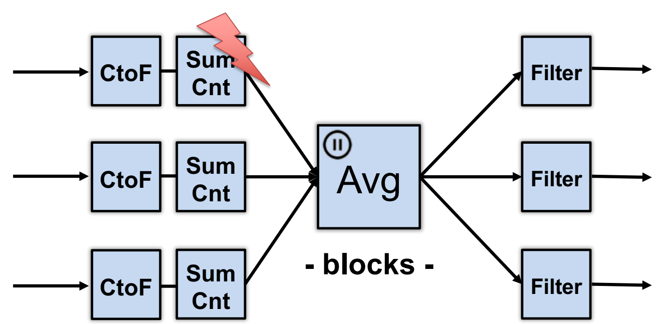
- Aggregations:
  – Need to join results across parallel computations



15

Parallelization complicates fault-tolerance

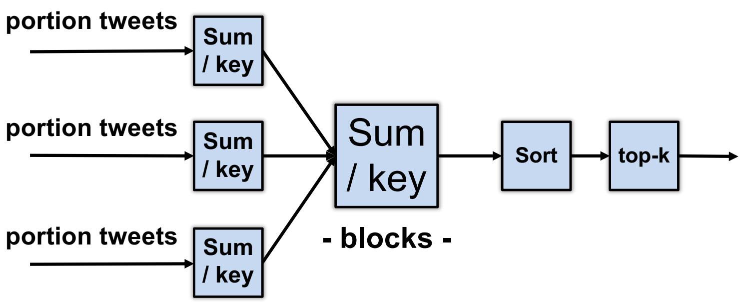
- Aggregations:
  – Need to join results across parallel computations



16

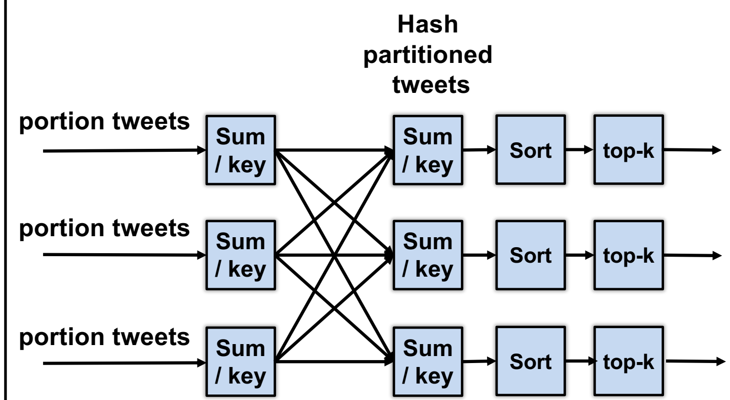
Can parallelize joins

- Compute trending keywords
  - E.g.,



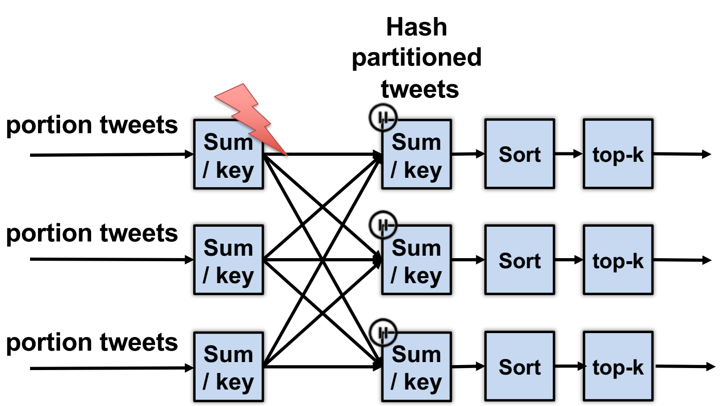
17

Can parallelize joins



18

Parallelization complicates fault-tolerance



19

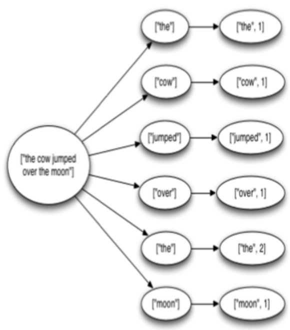
A Tale of Four Frameworks

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

20

Fault tolerance via record acknowledgement (Apache Storm -- at least once semantics)

- Goal: Ensure each input "fully processed"
- Approach: DAG / tree edge tracking
  - Record edges created as tuple processed
  - Wait for all edges to be marked done
  - Inform source of data when complete; otherwise, they resend tuple.
- Challenge: "at least once" means:
  - Operators can receive tuple > once
  - Replay can be out-of-order
  - ... application needs to handle.

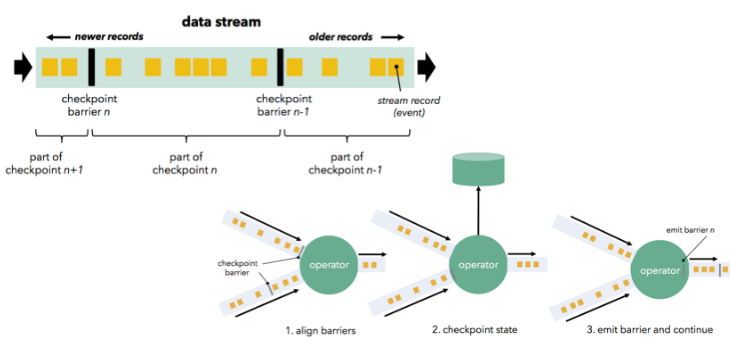


Fault Tolerance via distributed snapshots (Apache Flink)

- 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



But another big issue:

Streaming = unbounded data

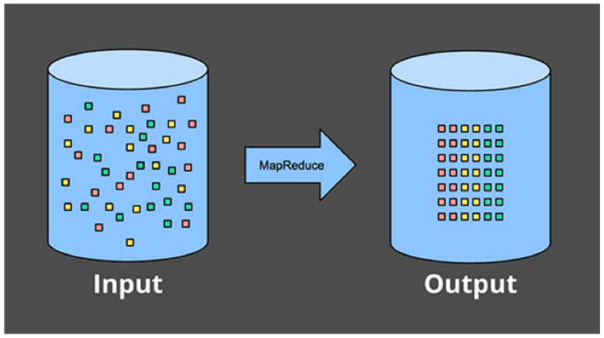
(Batch = bounded data)

Three major challenges

- **Consistency**: historically, streaming systems were created to decrease latency and made many sacrifices (e.g., at-most-once processing)
- **Throughput vs. latency**: typically a trade-off
- **Time**: new challenge

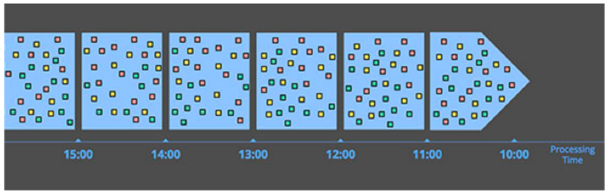
We've covered consistency in a lot of detail, let's investigate time

Our lives used to be easy…



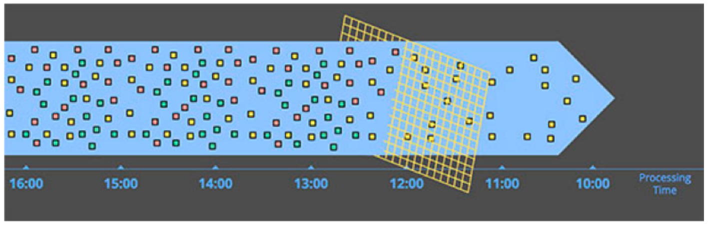
New Concerns

- Once data is unbounded, new concerns:
  - Sufficient capacity so processing speed >= arrival velocity (on average)
  - Support for handling out-of-order data
- Easiest thing to do:



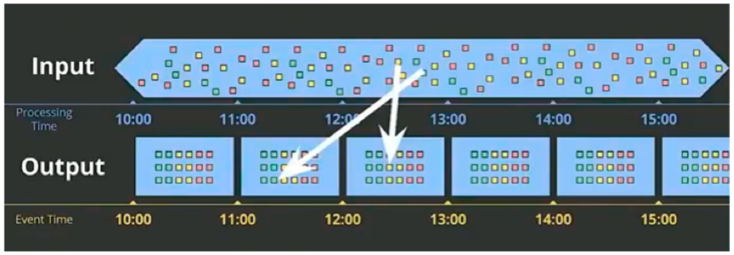
Windowing by processing time is great

- Easy to implement and verify correctness
- Great for applications like filtering or monitoring

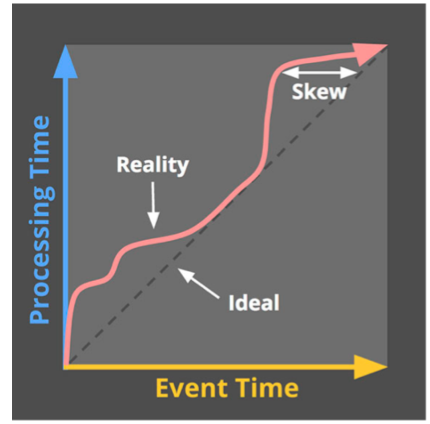


What if care about *when* events happen?

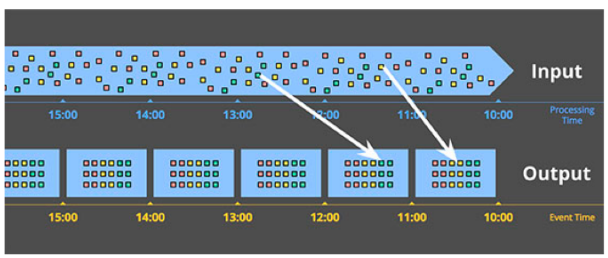
- If we associate event times, then items could now come out-of-order! (why?)



Time creates new wounds



This would be nice



But not the case, so we need tools

- **Windows**: how should we group together data?
- **Watermarks**: how can we mark when the last piece of data in some window has arrived?
- **Triggers**: how can we initiate an early result?
- **Accumulators**: what do we do with the results (correct, modified, or retracted)?

All topics covered in next week's readings!