

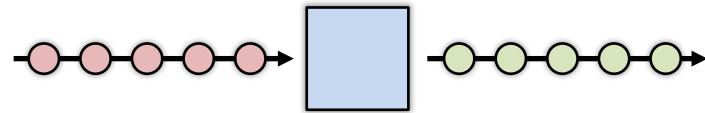
Stream Processing



COS 418: *Distributed Systems*
Lecture 22

Michael Freedman

Simple stream processing



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

2

Examples: Stateless conversion



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

3

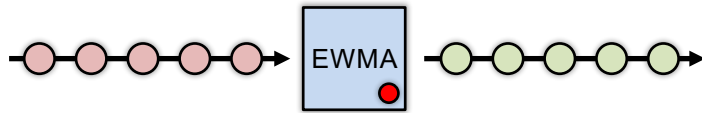
Examples: Stateless filtering



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

4

Examples: Stateful conversion





- Compute EWMA of Fahrenheit temperature
 - $\text{new_temp} = \alpha * (\text{CtoF}(\text{input})) + (1 - \alpha) * \text{last_temp}$
 - $\text{last_temp} = \text{new_temp}$
 - **emit** new_temp

5

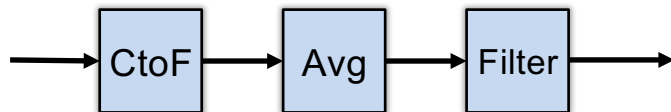
Examples: Aggregation (stateful)



- 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) 

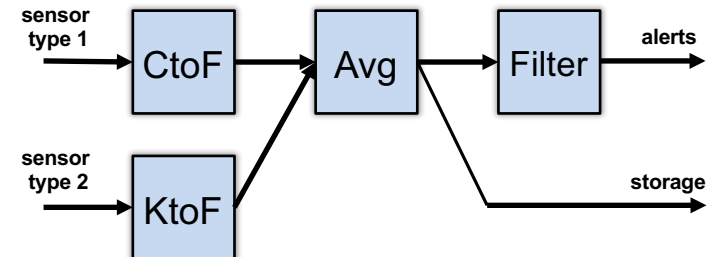
6

Stream processing as chain



7

Stream processing as directed graph



8

Enter “BIG DATA”

9

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

10

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
```

11

Scale “up”

Tuple-by-Tuple

Lower Latency
Lower Throughput

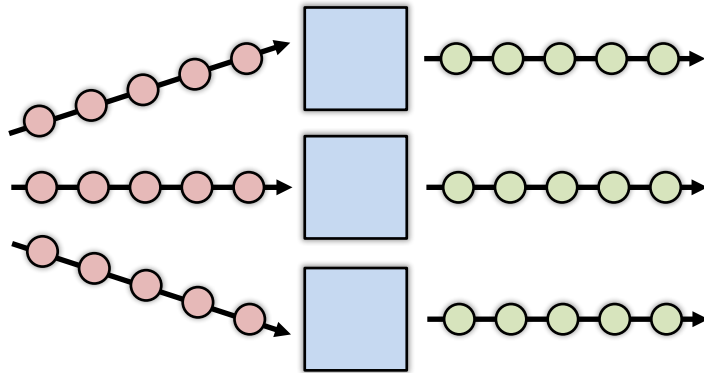
Micro-batch

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.

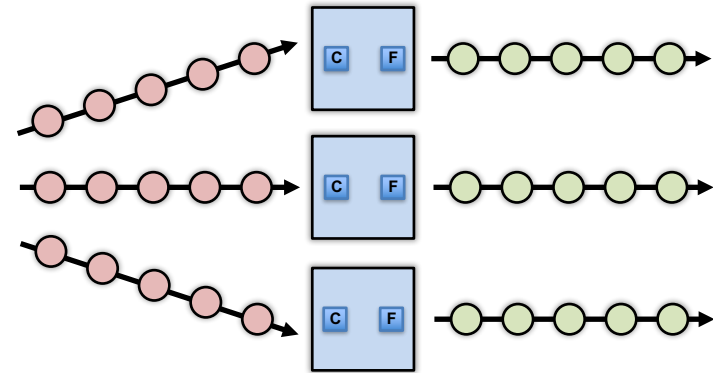
12

Scale “out”



13

Stateless operations: trivially parallelized



14

State complicates parallelization

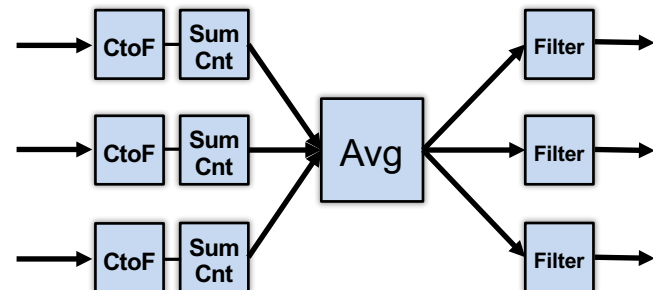
- Aggregations:
 - Need to join results across parallel computations



15

State complicates parallelization

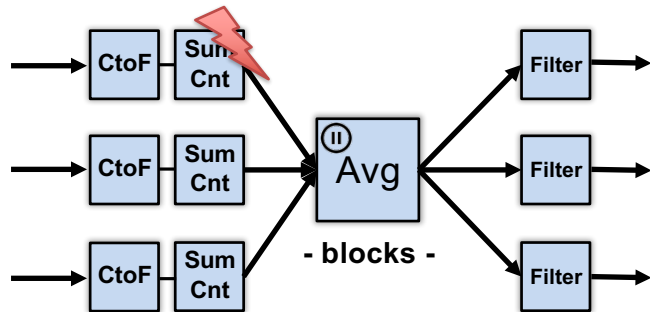
- Aggregations:
 - Need to join results across parallel computations



16

Parallelization complicates fault-tolerance

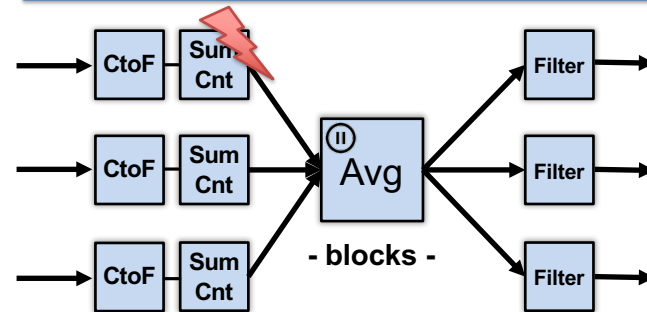
- Aggregations:
 - Need to join results across parallel computations



17

Parallelization complicates fault-tolerance

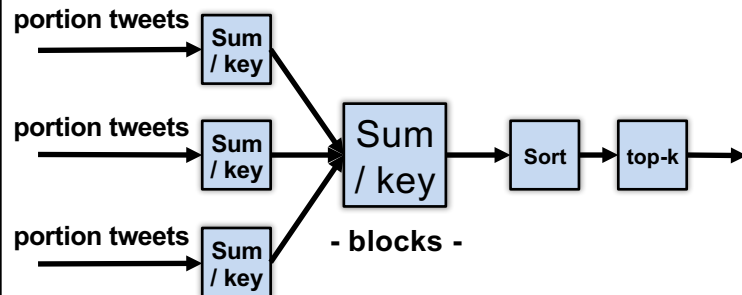
Can we ensure exactly-once semantics?



18

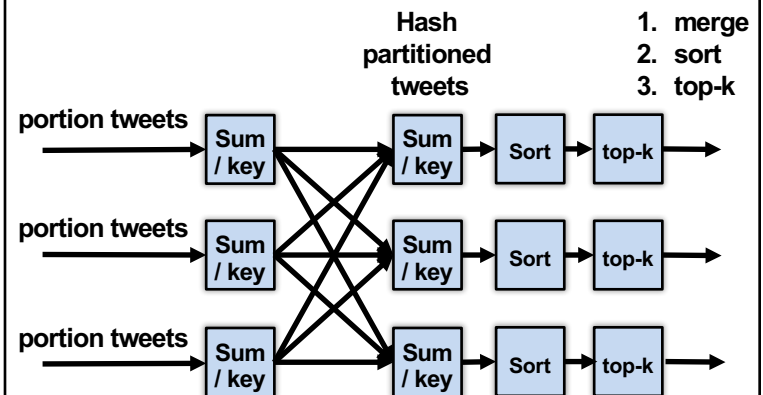
Can parallelize joins

- Compute trending keywords
 - E.g.,



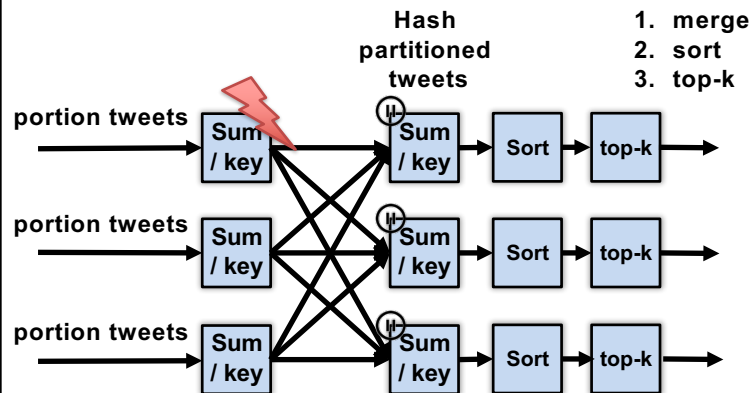
19

Can parallelize joins



20

Parallelization complicates fault-tolerance



21

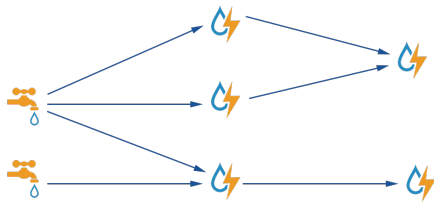
A Tale of Four Frameworks

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

22

Apache Storm

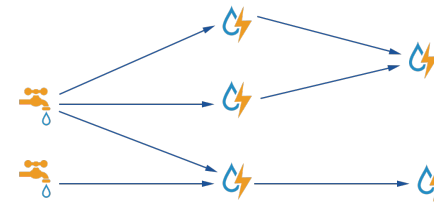
- Architectural components
 - Data: streams of tuples, e.g., Tweet = <Author, Msg, Time>
 - Sources of data: “spouts”
 - Operators to process data: “bolts”
 - Topology: Directed graph of spouts & bolts



23

Apache Storm: Parallelization

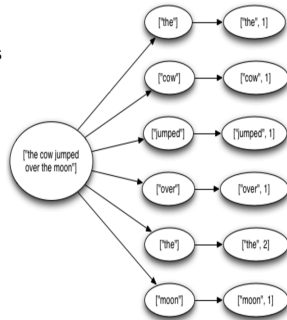
- Multiple processes (tasks) run per bolt
- Incoming streams split among tasks
 - **Shuffle Grouping:** Round-robin distribute tuples to tasks
 - **Fields Grouping:** Partitioned by key / field
 - **All Grouping:** All tasks receive all tuples (e.g., for joins)



24

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

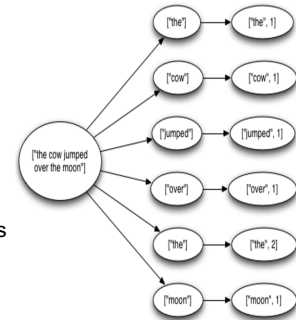
- Goal: Ensure each input "fully processed"
- Approach: DAG / tree edge tracking
 - Record edges that get created as tuple is processed.
 - Wait for all edges to be marked done
 - Inform source (spout) of data when complete; otherwise, they resend tuple.
- Challenge: "at least once" means:
 - Bolts can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



25

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

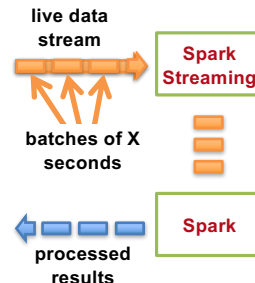
- Spout assigns new unique ID to each tuple
- When bolt "emits" dependent tuple, it informs system of dependency (new edge)
- When a bolt finishes processing tuple, it calls ACK (or can FAIL)
- Acker tasks:
 - Keep track of all emitted edges and receive ACK/FAIL messages from bolts.
 - When messages received about all edges in graph, inform originating spout
- Spout garbage collects tuple or retransmits
- Note: Best effort delivery by not generating dependency on downstream tuples.



26

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
 - Akin to in-memory MapReduce
- Emit each micro-batch result
 - RDD = "Resilient Distributed Data"



27

Apache Spark Streaming: Dataflow-oriented programming

```
# Create a local StreamingContext with batch interval of 1 second
ssc = StreamingContext(sc, 1)
# Create a DStream that reads from network socket
lines = ssc.socketTextStream("localhost", 9999)

words = lines.flatMap(lambda line: line.split(" ")) # Split each line into words

# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

wordCounts.pprint()

ssc.start() # Start the computation
ssc.awaitTermination() # Wait for the computation to terminate
```

28

Apache Spark Streaming: Dataflow-oriented programming

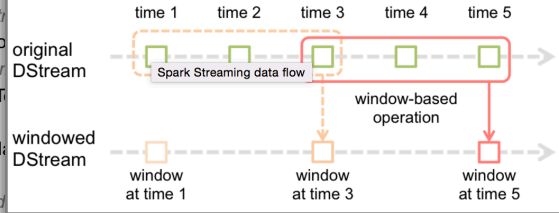
```
# Create a local StreamingContext with resiliency parameter 'Resiliency_Enabled'
ssc = StreamingContext(Seconds(10), Resiliency_Enabled)
# Create a DStream representing the input socket stream
lines = ssc.socketTextStream("localhost", 8080)

words = lines.flatMap(splitters.words)

# Count each word in the stream
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKeyAndWindow( lambda x, y: x + y,
                                         lambda x, y: x - y, 3, 2)

wordCounts.pprint()

ssc.start() # Start the computation
ssc.awaitTermination() # Wait for the computation to terminate
```



29

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 upstream dependencies starting at checkpoint (downstream might filter)

30

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

31

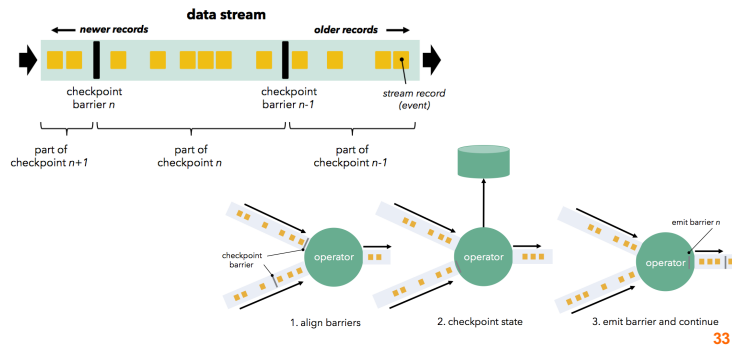
Fault Tolerance via distributed snapshots (Apache Flink)

- 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

32

Fault Tolerance via distributed snapshots (Apache Flink)

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



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”**

34

Wednesday lecture

Cluster Scheduling

35