Big Data Processing

COS 418: *Distributed Systems*
Lecture 21

Michael Freedman

Data-Parallel Computation

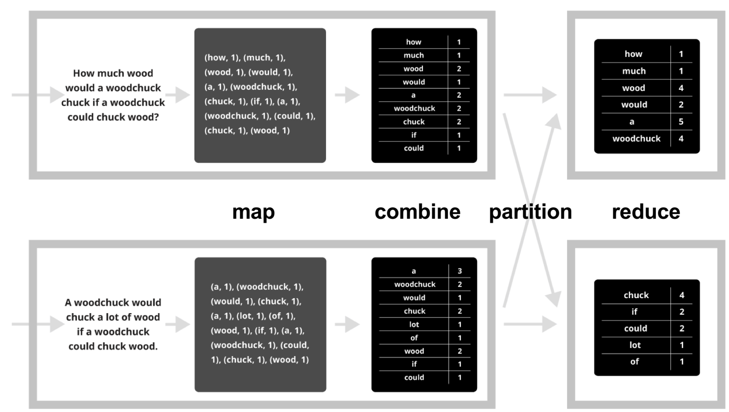
Ex: Word count using partial aggregation

1. Compute word counts from individual files

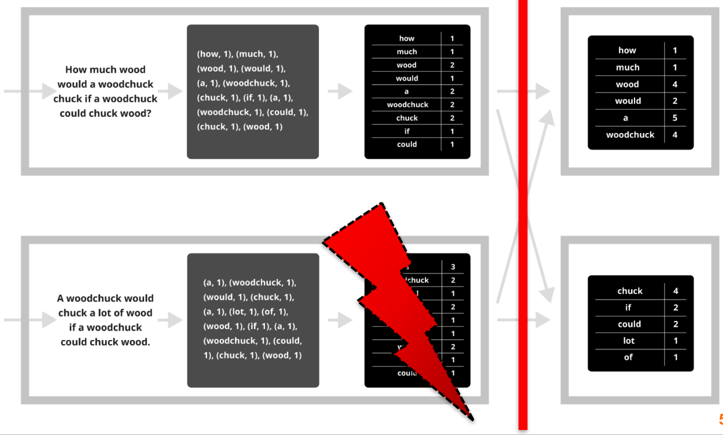
2. Then merge intermediate output

3. Compute word count on merged outputs

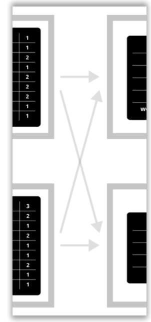
Putting it together…



Synchronization Barrier



Fault Tolerance in MapReduce



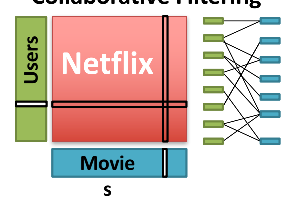
- Map worker writes intermediate output to local disk, separated by partitioning. Once completed, tells master node.
- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data
- Note:
  - "All-to-all" shuffle b/w mappers and reducers
  - Written to disk ("materialized") b/w *each* stage

Graph-Parallel Computation

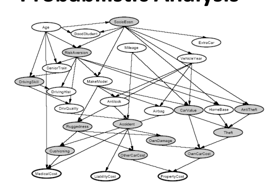
Graphs are Everywhere

Social Network

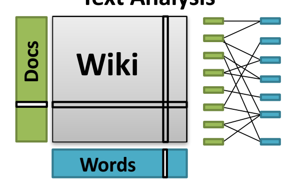
Collaborative Filtering



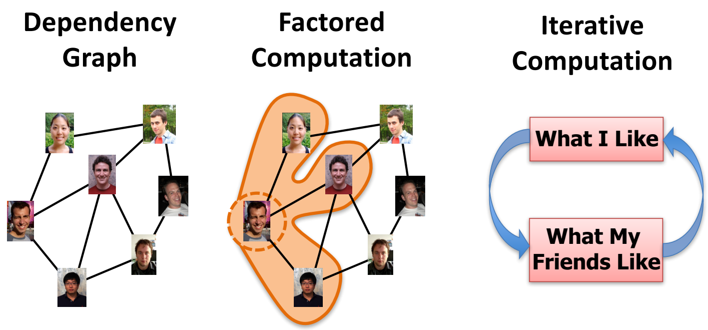
Probabilistic Analysis



Text Analysis

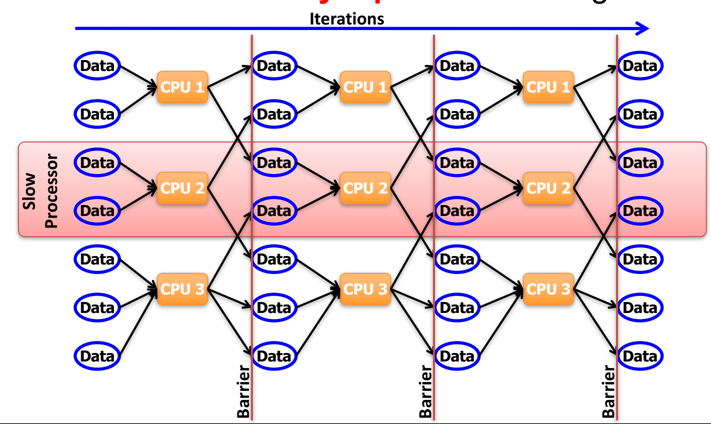


Properties of Graph Parallel Algorithms



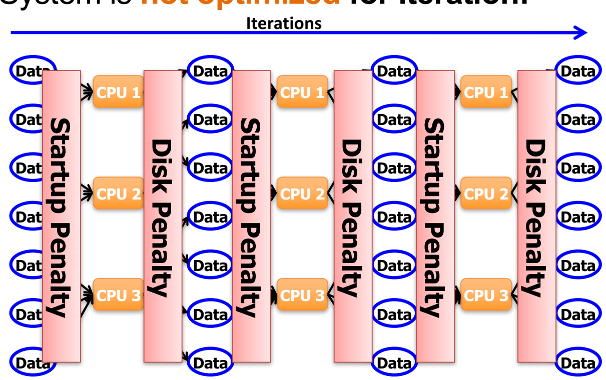
Iterative Algorithms

- MR **doesn't efficiently express** **iterative** algorithms:

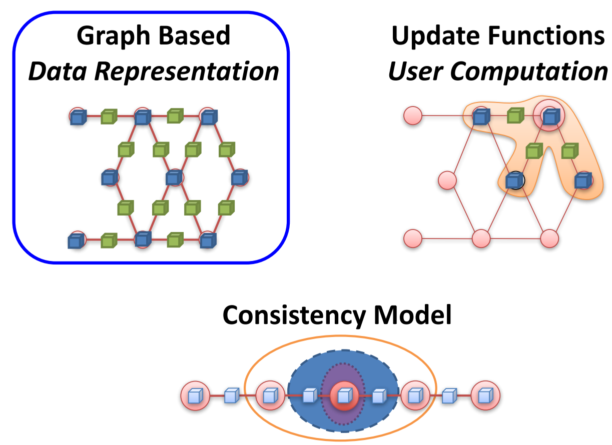


MapAbuse: Iterative MapReduce

- System is **not optimized** **for iteration:**



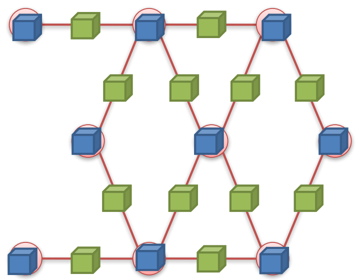
The GraphLab Framework



12

Data Graph

Data is associated with both vertices and edges



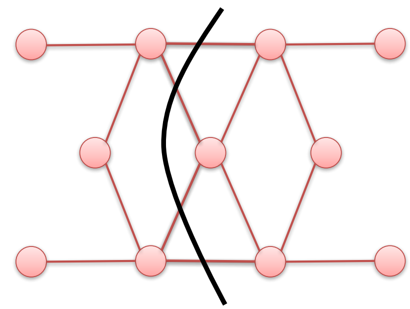
Graph:
- **Social Network**

Vertex Data:
- **User profile**
- **Current interests estimates**

Edge Data:
- **Relationship (friend, classmate, relative)**

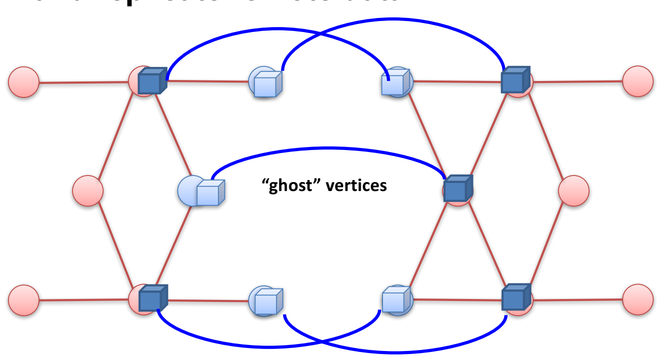
Distributed Data Graph

Partition the graph across multiple machines:



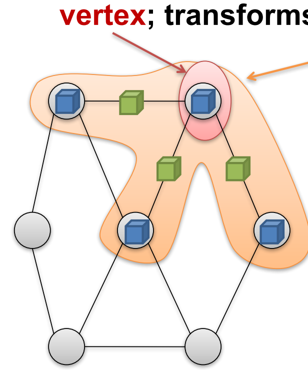
Distributed Data Graph

- ***Ghost vertices*** **maintain adjacency structure and replicate remote data.**



Update Function

A user-defined program, applied to a vertex; transforms data in scope of vertex



Pagerank (scope) {

// Update the current vertex data

vertex.PageRank = α

ForEach inPage:

vertex.PageRank += $(1-\alpha)\times$ inPage.PageRank

// Reschedule Neighbors if needed

if vertex.PageRank changes then

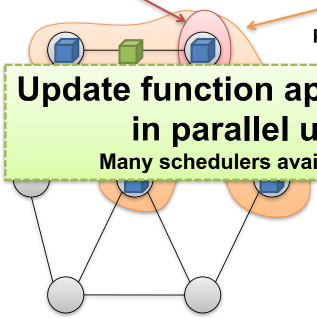
reschedule_all_neighbors;

}

Selectively triggers computation at neighbors

Update Function

A user-defined program, applied to a vertex; transforms data in scope of vertex



```
Pagerank (scope) {
  // Update the current vertex data
```

Update function applied (asynchronously) in parallel until convergence

Many schedulers available to prioritize computation

```
  // Reschedule Neighbors if needed
  if vertex.PageRank changes then
      reschedule_all_neighbors;
}
```

Selectively triggers computation at neighbors

17

How to handle machine failure?

- *What when machines fail?*
  How do we **provide fault tolerance?**

- Strawman scheme:
  Synchronous snapshot checkpointing

  1. Stop the world
  2. Write each machines' state to disk

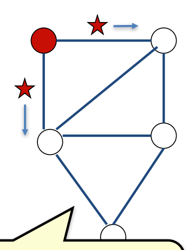
Chandy-Lamport checkpointing

Step 1. Atomically, one *initiator:*

1. Turns red
2. Records its own state
3. Sends *marker* to neighbors

Step 2. On receiving marker.
non-red node atomically:

1. Turns red,
2. Records its own state,
3. Sends markers along all outgoing channels

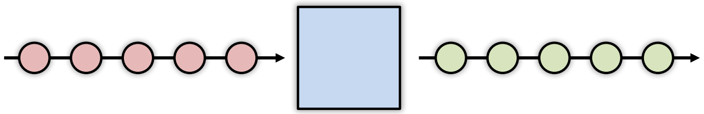


First-in, first-out channels between nodes

Stream Processing

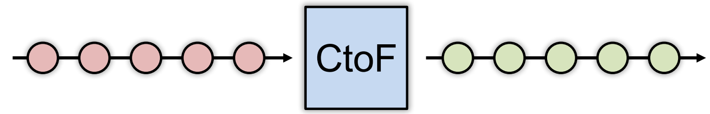
20

Simple stream processing



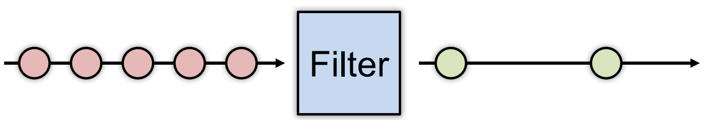
- Single node
  - Read data from socket
  - Process
  - Write output

Examples: Stateless conversion



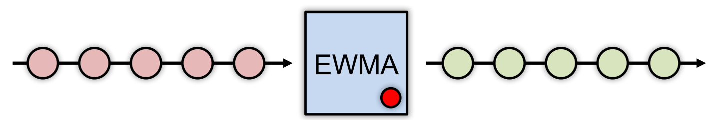
- Convert Celsius temperature to Fahrenheit
  - Stateless operation: **emit** (input * 9 / 5) + 32

Examples: Stateless filtering



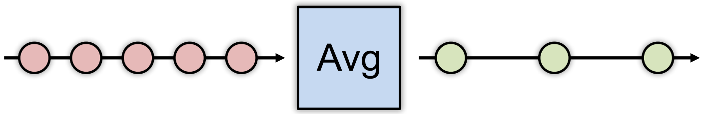
- Function can filter inputs
  - if (input > threshold) { **emit** input }

Examples: Stateful conversion

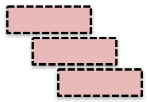


- Compute EWMA of Fahrenheit temperature
  - new_temp = α * (CtoF(input)) + (1- α) * last_temp
  - last_temp = new_temp
  - **emit** new_temp

Examples: Aggregation (stateful)

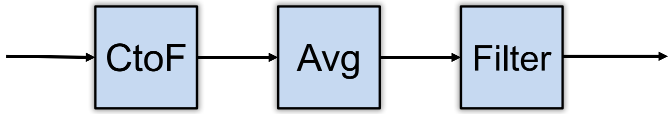


- E.g., Average value per window
  - Window can be # elements (10) or time (1s)
  - Windows can be disjoint (every 5s)
  - Windows can be "tumbling" (5s window every 1s)



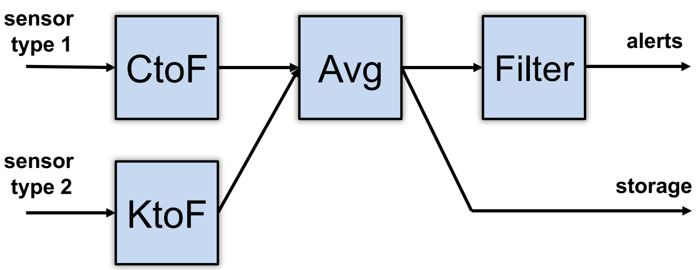
25

Stream processing as chain



26

Stream processing as directed graph



27

Enter "BIG DATA"

28

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

29

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
```

30

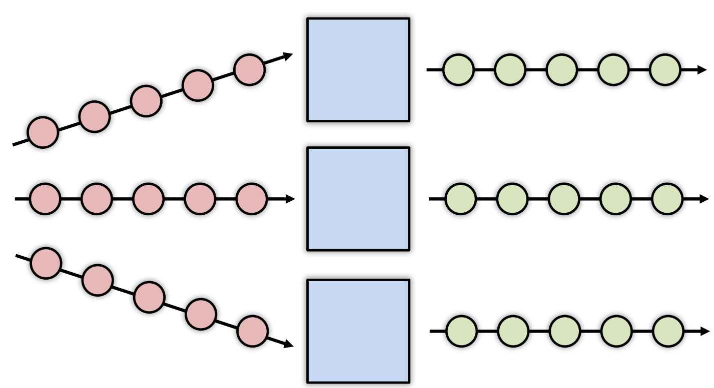
Scale "up"

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.

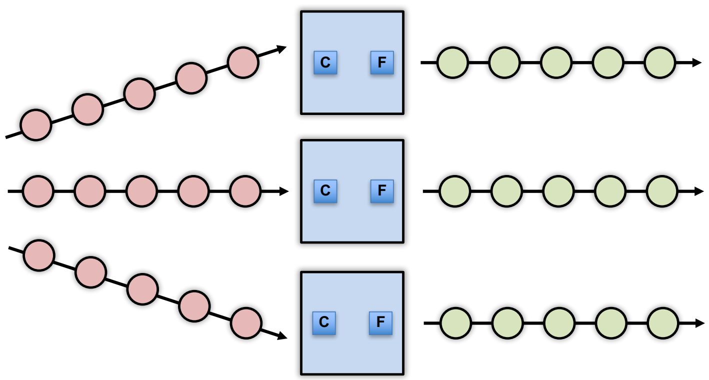
31

Scale "out"



32

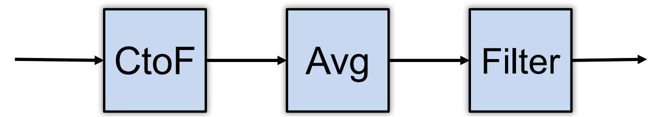
Stateless operations: trivially parallelized



33

State complicates parallelization

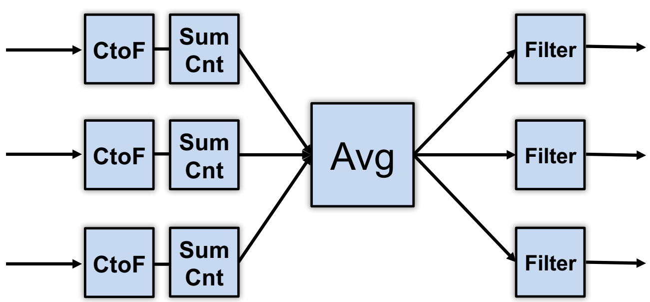
- Aggregations:
  - Need to join results across parallel computations



34

State complicates parallelization

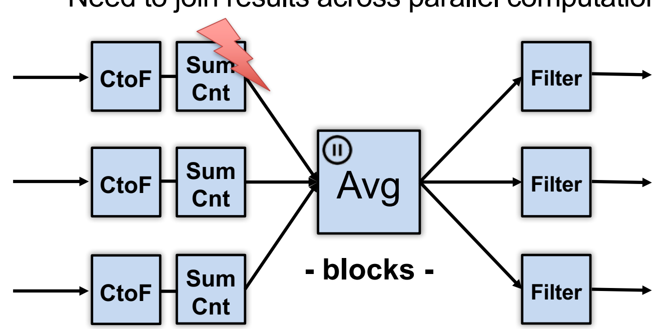
- Aggregations:
  - Need to join results across parallel computations



35

Parallelization complicates fault-tolerance

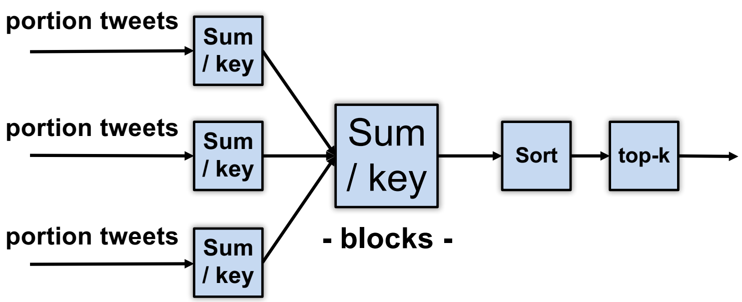
- Aggregations:
  - Need to join results across parallel computations



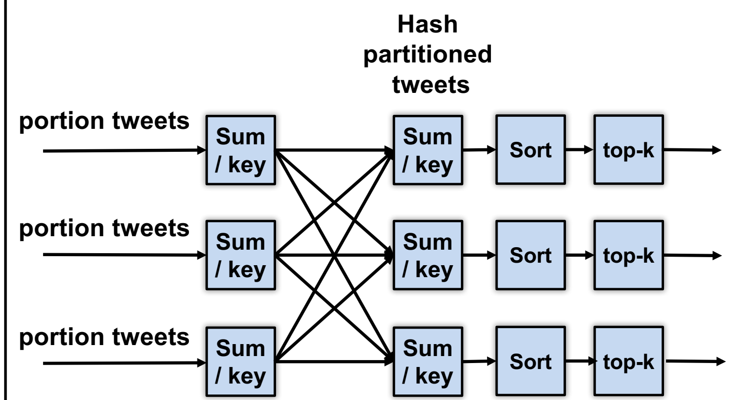
36

Can parallelize joins

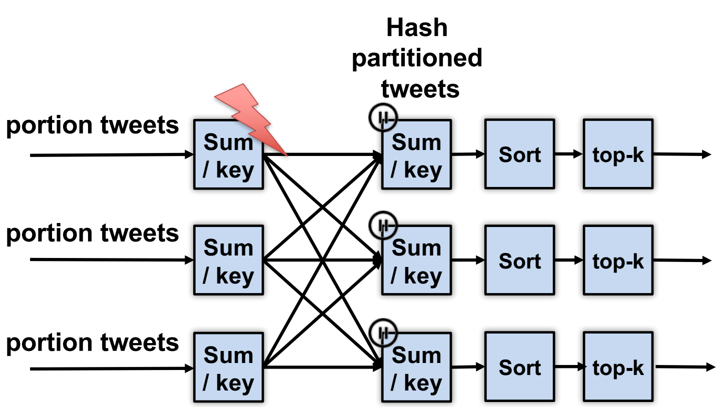
- Compute trending keywords
  - E.g.,



Can parallelize joins



Parallelization complicates fault-tolerance

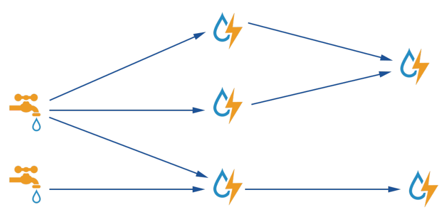


A Tale of Four Frameworks

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

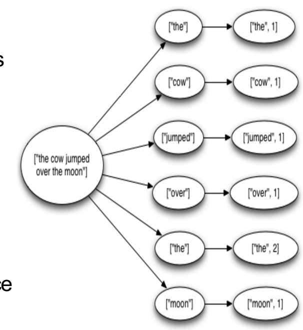
Apache Storm

- 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



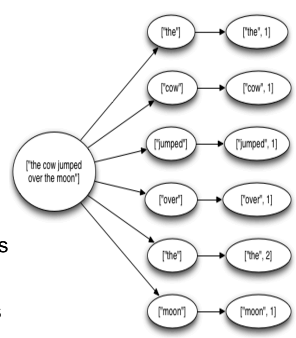
Fault tolerance via record acknowledgement (Apache Storm -- at least once semantics)

- 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 (spouts) of data when complete; otherwise, they resend tuple.
- Challenge: “at least once” means:
  - Operators (bolts) can receive tuple > once
  - Replay can be out-of-order
  - ... application needs to handle.



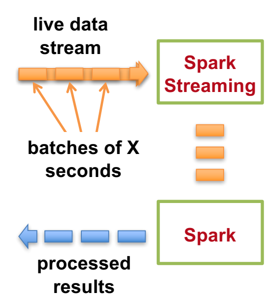
Fault tolerance via record acknowledgement (Apache Storm -- at least once semantics)

- 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.



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”



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)

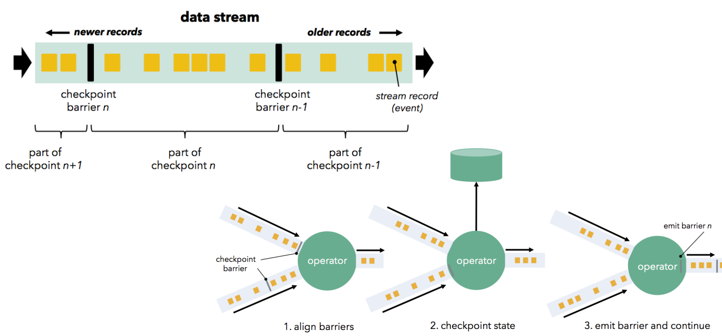
- Computation is long-running DAG of continuous operators
- For each intermediate record at operator
  - Create commit record including input record, state update, and derived downstream records generated
  - Write commit record to transactional log / DB
- On failure, replay log to
  - Restore a consistent 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



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”**