**Stream Processing: What We'll Cover**

- Why Stream Processing
- Storm
Stream Processing Challenge

- Large amounts of data => Need for real-time views of data
  - Social network trends, e.g., Twitter real-time search
  - Website statistics, e.g., Google Analytics
  - Intrusion detection systems, e.g., in most datacenters

- Process large amounts of data
  - With latencies of few seconds
  - With high throughput
• Batch Processing $\Rightarrow$ Need to wait for entire computation on large dataset to complete
• Not intended for long-running stream-processing
Apache Project

https://storm.incubator.apache.org/

Highly active JVM project

Multiple languages supported via API
  • Python, Ruby, etc.

Used by over 30 companies including
  • Twitter: For personalization, search
  • Flipboard: For generating custom feeds
  • Weather Channel, WebMD, etc.
STORM COMPONENTS

- Tuples
- Streams
- Spouts
- Bolts
- Topologies
An ordered list of elements

E.g., `<tweeter, tweet>`
  - E.g., `<“Miley Cyrus”, “Hey! Here’s my new song!”>`
  - E.g., `<“Justin Bieber”, “Hey! Here’s MY new song!”>`

E.g., `<URL, clicker-IP, date, time>`
  - E.g., `<coursera.org, 101.201.301.401, 4/4/2014, 10:35:40>`
  - E.g., `<coursera.org, 901.801.701.601, 4/4/2014, 10:35:42>`
**Sequence of tuples**
- Potentially unbounded in number of tuples

**Social network example:**
- <“Miley Cyrus”, “Hey! Here’s my new song!”>,
  <“Justin Bieber”, “Hey! Here’s MY new song!”>,
  <“Rolling Stones”, “Hey! Here’s my old song that’s still a super-hit!”>, ...

**Website example:**
- <coursera.org, 101.201.301.401, 4/4/2014, 10:35:40>, <coursera.org, 901.801.701.601, 4/4/2014, 10:35:42>, ...
• A Storm entity (process) that is a source of streams
• Often reads from a crawler or DB
- A Storm entity (process) that
  - Processes input streams
  - Outputs more streams for other bolts
A directed graph of spouts and bolts (and output bolts)
Corresponds to a Storm “application”
Can have cycles if the application requires it
BOLTS come in many Flavors

• Operations that can be performed
  • Filter: forward only tuples which satisfy a condition
  • Joins: When receiving two streams A and B, output all pairs (A,B) which satisfy a condition
  • Apply/transform: Modify each tuple according to a function
  • And many others

• But bolts need to process a lot of data
  • Need to make them fast
PARALLELIZING BOLTS

- Have multiple processes ("tasks") constitute a bolt
- Incoming streams split among the tasks
- Typically each incoming tuple goes to one task in the bolt
  - Decided by "Grouping strategy"
- Three types of grouping are popular
Grouping

- **Shuffle Grouping**
  - Streams are distributed randomly to the bolt’s tasks
  - Randomly but consistently – use a hash function! (Remember consistent hashing from P2P systems?)

- **Fields Grouping**
  - Group a stream by a subset of its fields
  - E.g., All tweets where twitter username starts with [A-M,a-m,0-4] goes to task 1, and all tweets starting with [N-Z,n-z,5-9] go to task 2

- **All Grouping**
  - All tasks of bolt receive all input tuples
  - Useful for joins
**Storm Cluster**

- **Master node**
  - Runs a daemon called *Nimbus*
  - Responsible for
    - Distributing code around cluster
    - Assigning tasks to machines
    - Monitoring for failures of machines

- **Worker node**
  - Runs on a machine (server)
  - Runs a daemon called *Supervisor*
  - Listens for work assigned to its machines

- **Zookeeper**
  - Coordinates Nimbus and Supervisors communication
  - All state of Supervisor and Nimbus is kept here
Failures

- A tuple is considered failed when its topology (graph) of resulting tuples fails to be fully processed within a specified timeout.

- **Anchoring:** Anchor an output to one or more input tuples.
  - Failure of one tuple causes one or more tuples to be replayed.
API For Fault-Tolerance (OutputCollector)

- **Emit**(tuple, output)
  - Emits an output tuple, perhaps anchored on an input tuple (first argument)
- **Ack**(tuple)
  - Acknowledge that you (bolt) finished processing a tuple
- **Fail**(tuple)
  - Immediately fail the spout tuple at the root of tuple topology if there is an exception from the database, etc.
- **Must remember to ack/fail each tuple**
  - Each tuple consumes memory. Failure to do so results in memory leaks.
• Processing data in real-time a big requirement today
• Storm
  • And other sister systems, e.g., Spark Streaming
• Parallelism
• Application topologies
• Fault-tolerance
Graph Processing: What We’ll Cover

- Distributed Graph Processing
- Google’s Pregel system
  - Inspiration for many newer graph processing systems: Piccolo, Giraph, GraphLab, PowerGraph, LFGraph, X-Stream, etc.
Lots of Graphs

• Large graphs are all around us
  • Internet Graph: vertices are routers/switches and edges are links
  • World Wide Web: vertices are webpages, and edges are URL links on a webpage pointing to another webpage
    • Called “Directed” graph as edges are uni-directional
  • Social graphs: Facebook, Twitter, LinkedIn
  • Biological graphs: DNA interaction graphs, ecosystem graphs, etc.
Graph Processing Operations

• Need to derive properties from these graphs
• Need to summarize these graphs into statistics
  • E.g., find shortest paths between pairs of vertices
    • Internet (for routing)
    • LinkedIn (degrees of separation)
• E.g., do matching
  • Dating graphs in match.com (for better dates)
• PageRank
  • Web Graphs
  • Google search, Bing search, Yahoo search: all rely on this
• And many (many) other examples!
Why Hard?

• Because these graphs are large!
  • Human social network has 100s Millions of vertices and Billions of edges
  • WWW has Millions of vertices and edges
• Hard to store the entire graph on one server and process it
  • Slow on one server (even if beefy!)
• Use distributed cluster/cloud!
Typical Graph Processing Application

- Works in *iterations*
- Each vertex assigned a *value*
- In each iteration, each vertex:
  1. Gathers values from its immediate neighbors (vertices who join it directly with an edge). E.g., @A: B→A, C→A, D→A, ...
  2. Does some computation using its own value and its neighbors values.
  3. Updates its new value and sends it out to its neighboring vertices. E.g., A→B, C, D, E
- Graph processing terminates after: i) fixed iterations, or ii) vertices stop changing values
Multi-stage Hadoop
• Each stage == 1 graph iteration
• Assign vertex ids as keys in the reduce phase

Well-known

At the end of every stage, transfer all vertices over network (to neighbor vertices)
  All vertex values written to HDFS (file system)
  Very slow!

Hadoop/MapReduce to the Rescue?
"Think like a vertex"

Originally by Valiant (1990)

Basic Distributed Graph Processing

- “Think like a vertex”
- Assign each vertex to one server
- Each server thus gets a subset of vertices
- In each iteration, each server performs **Gather-Apply-Scatter** for all its assigned vertices
  - Gather: get all neighboring vertices’ values
  - Apply: compute own new value from own old value and gathered neighbors’ values
  - Scatter: send own new value to neighboring vertices
Assigning Vertices

• How to decide which server a given vertex is assigned to?
• Different options
  • **Hash-based**: Hash(vertex id) modulo number of servers
    • Remember consistent hashing from P2P systems?!
  • **Locality-based**: Assign vertices with more neighbors to the same server as its neighbors
    • Reduces server to server communication volume after each iteration
    • Need to be careful: some “intelligent” locality-based schemes may take up a lot of upfront time and may not give sufficient benefits!
Pregel System by Google

- Pregel uses the master/worker model
  - Master (one server)
    - Maintains list of worker servers
    - Monitors workers; restarts them on failure
    - Provides Web-UI monitoring tool of job progress
  - Worker (rest of the servers)
    - Processes its vertices
    - Communicates with the other workers
- Persistent data is stored as files on a distributed storage system (such as GFS or BigTable)
- Temporary data is stored on local disk
1. Many copies of the program begin executing on a cluster
2. The master assigns a partition of input (vertices) to each worker
   • Each worker loads the vertices and marks them as *active*
3. The master instructs each worker to perform a iteration
   • Each worker loops through its active vertices & computes for each vertex
   • Messages can be sent whenever, but need to be delivered before the end of the iteration (i.e., the barrier)
   • When all workers reach iteration barrier, master starts next iteration
4. Computation halts when, in some iteration: no vertices are active and when no messages are in transit
5. Master instructs each worker to save its portion of the graph
Fault-Tolerance in Pregel

- **Checkpointing**
  - Periodically, master instructs the workers to save state of their partitions to persistent storage
    - e.g., Vertex values, edge values, incoming messages

- **Failure detection**
  - Using periodic “ping” messages from master → worker

- **Recovery**
  - The master reassigns graph partitions to the currently available workers
  - The workers all reload their partition state from most recent available checkpoint
How Fast Is It?

• Shortest paths from one vertex to all vertices
  • SSSP: “Single Source Shortest Path”
• On 1 Billion vertex graph (tree)
  • 50 workers: 180 seconds
  • 800 workers: 20 seconds
• 50 B vertices on 800 workers: 700 seconds (~12 minutes)
• Pretty Fast!
SUMMARY: GRAPH PROCESSING

- Lots of (large) graphs around us
- Need to process these
- MapReduce not a good match
- Distributed Graph Processing systems: Pregel by Google
- Many follow-up systems
  - Piccolo, Giraph: Pregel-like
  - GraphLab, PowerGraph, LFGraph, X-Stream: more advanced