Muppet: MapReduce-Style Processing of Fast Data

Amila Suriarachchi
Big Data Group

Motivation

- MapReduce has has emerged as a popular method to process big data
- However MapReduce is not suitable to process fast data
- Is it possible to write a MapReduce-Style framework to achieve low latency and high scalability?

Motivating Applications

- An application that monitors Foursquare – checkin stream to count the number of checkins per retailer
- Twitter Firehose to detect hot topics as they occur
- An application maintains a reputation score for each Twitter user as users tweet
- All these applications perform stream computations

Why MapReduce is not suitable?

- MapReduce runs on a static snapshot of a data set, while stream computations proceed over an evolving data stream
- MapReduce computation has a start and a finish. Stream computation never end
- In case of a failure, MapReduce jobs can be restarted. But stream system should cope with failures without dragging too far.

Requirements of new framework

- Should be easy to program. Should retain the familiar Map and Reduce model
- Should manage dynamic data structures as first class citizens
- Low latency for near real time processing
- Should scale up in with the commodity hardware

MapUpdate

- MapUpdate operates on data streams. Map and Update functions should define on data streams
- Streams never ends. Updaters use slates to summarize data so far
- Not just a mapper and updater but many of them in a workflow that consumes streams

Events and Streams

- Event is a tuple <sid, ts, k, v>
  - sid – Stream ID
  - ts – Time Stamp
  - k – key
  - v – value (binary value)

Map Function

- map(event) → event*
- Subscribes to one or more streams
- Receive events ordered by time stamp
- Process input streams and emit new events to one or more streams

Map Sample

```
public void map(PerformerUtilities submitter,
                String stream, byte[] key, byte[] event)
{
    String checkin = new String(event, charset);
    String venue = getVenue(checkin);

    String retailer = null;
    if (walmart.matcher(venue).matches()) {
        retailer = "Walmart";
    } else if (samsclub.matcher(venue).matches()) {
        retailer = "Sam's Club";
    }

    if (retailer != null) {
        try {
            submitter.publish("S_2",
                retailer.getBytes(charset), event);
        } catch(Exception e) {
            logger.error("Could not publish event: "+
                e.toString());
        }
    }
}
```

Update Function

- update(event, slate) → event*
- One slate for each key
- Receive data from multiple streams, process them update slate and emits new events

Update Sample

```
public void update(PerformerUtilities submitter,
                   String stream, byte[] key, byte[] event,
                   byte[] slate)
{
    int count = 0;
    try {
        if (slate != null)
            count =
                Integer.parseInt(new String(slate, charset));
    } catch (NumberFormatException e) {
        count = 0;
    }
    ++count;
    byte[] updatedSlate =
        Integer.toString(count).getBytes(charset);
    submitter.replaceSlate(updatedSlate);
}
```

Map Update Application

- Map Update application is a work flow of Map and Update functions
- Work flow is modeled as a directed graph (cycles allowed)
  - Map and update functions as Nodes
  - Streams as Edges
- Use a configuration file to define the flow

Sample Applications

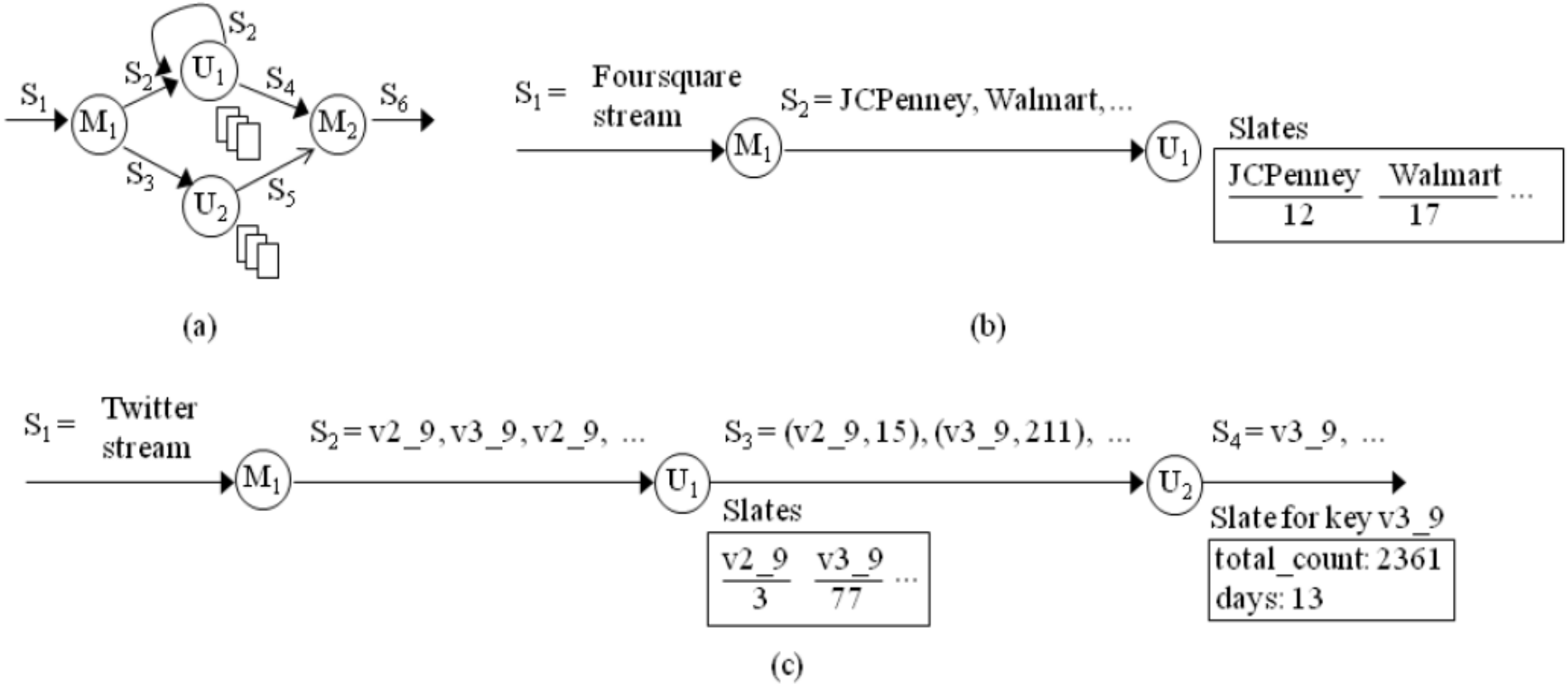


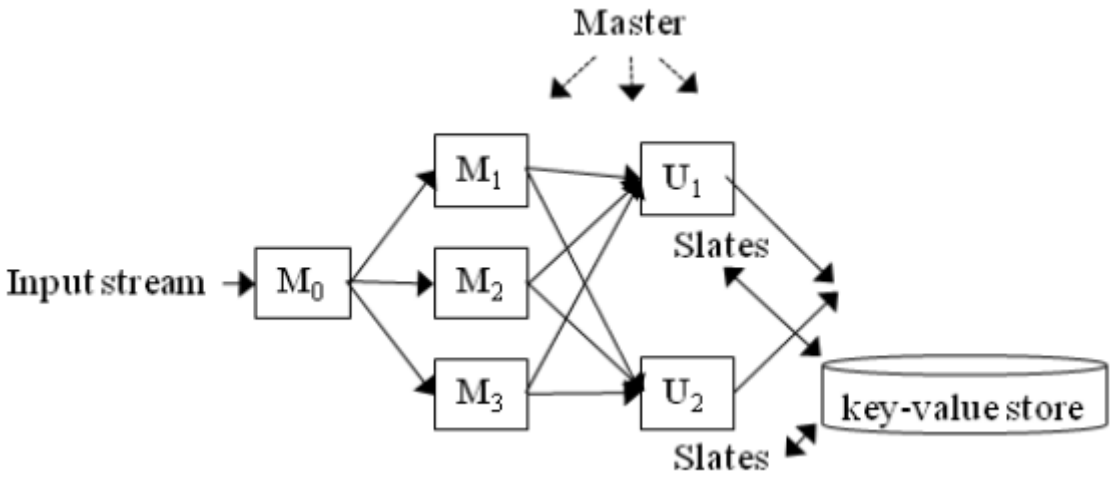
Figure 1: Example MapUpdate applications

Muppet System 1.0

- Distributed Execution
- Managing slates
- Handling Failures
- Reading slates

Distributed Execution

- Each machine runs a Worker which executes either a mapper or updater
- Use Hash function to map key to updater to avoid master



Managing Slates

- Uses a key – value store to
  - Avoid memory out grow
  - Help resuming, restarting, or recovering the application from crash
  - Query the slates
- Use Cassandara on SSD (solid state flash memory storage) as the key-value store

Handling Failures

- Machine Crash
  - When a Node detect a failure it notify that to master and master notify it to all other nodes. Failed node removed from hash ring
- Queue Overflow
  - If a workers Queue is full it decline the event
  - Sending process can either drop the event or direct to an overflow stream

Reading Slates

- Uses a small HTTP server in each node to retrieve state from slates
- URL contains the updater and key
- Retrieve from updater nodes to get update copy

Muppet 2.0

- Written in Java and scala
- Each worker is now a thread that can execute any map or update function
- All threads share same map and update code
- All states are kept in a single central slate cache
- Allow two workers to process same key

Experience and ongoing extensions

- Limiting Slate Sizes
- Changing the Number of Machines on the Fly
  - How to redistribute the load
- Handling hot spots
- Placing Mappers and Updaters
- Bulk reading of states

Questions?