Tagging and Processing Data in Real Time

Hari Shreedharan
Cloudera

Siddhartha Jain
Salesforce.com

Features

- Process and normalize log streams in near real time
  - regex matching
- Scale from 100k events per second to a million
  - More producers could get added in real time
  - Must scale to increasing data volumes
- Horizontal scalability and fault tolerance
  - Throwing more hardware at the app should not break the app
  - Machines/Tiers can fail and come back
- Ability to plug in existing frameworks
  - Kafka, HDFS, Elastic Search, Spark….

Goals

- Search/Analysis
  - Investigations and casual exploration
- Compute
  - Data Science, correlation and alerting
- Enrich
  - For integrating external threat feeds and internally generated profiles

Expectations

- Stream processing delay tolerance
  - Worst case < 5 minutes
- Volume of messages
  - Anywhere between 100k/sec to 1 million/sec
- Maintain common data dictionary across ingest, store and compute pipelines

Possible solution stacks

Flume:

Syslog Source

↓

File Channel

↓

Custom Interceptor

↓

Sink to HDFS

Storm:

Custom Syslog Spout

↓

Custom Bolts

↓

Custom “Save to HDFS”

Spark Streaming:

RSyslog Kafka Plugin

↓

Kafka

↓

Spark Streaming Receiver

↓

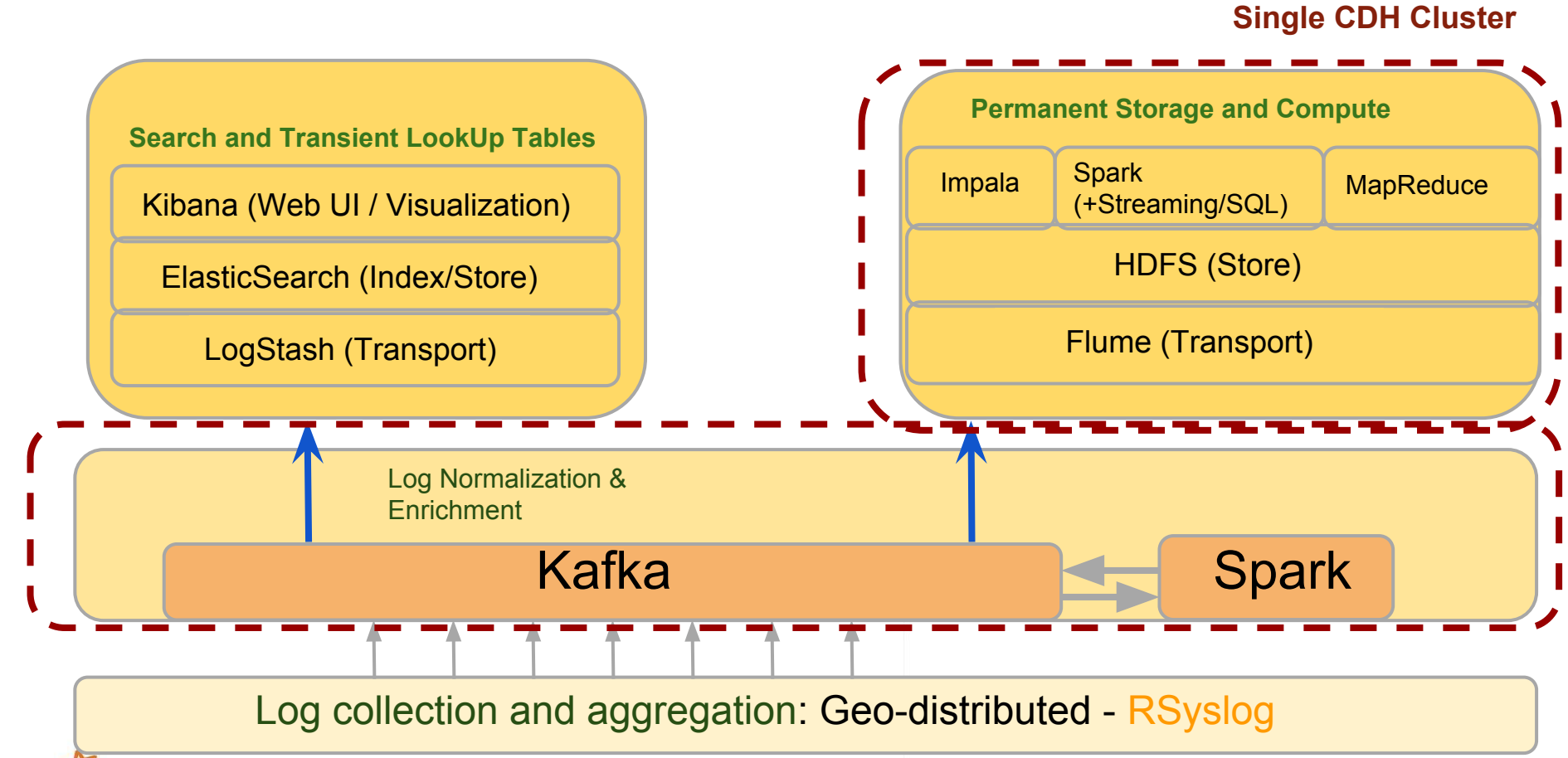
Custom Spark App

↓

Kafka

↓

Storage/Compute


Single CDH Cluster
Search and Transient LookUp Tables
Kibana (Web UI / Visualization)
ElasticSearch (Index/Store)
LogStash (Transport)
Permanent Storage and Compute
Impala
Spark (+Streaming/SQL)
MapReduce
HDFS (Store)
Flume (Transport)
Log Normalization & Enrichment
Kafka
Spark
Log collection and aggregation: Geo-distributed - RSyslog

RSyslog

```
<86>May 15 20:29:59 rh6-x64-test-template sshd[32632]: Accepted publickey for jdoe from 10.3.1.1 port 62902
ssh2
```

Kafka (topic=rawUnStructured)

```
<86>May 15 20:29:59 rh6-x64-test-template sshd[32632]: Accepted publickey for jdoe from 10.3.1.1 port 62902
ssh2
```

Spark (Apply Regex match/extract, map to JSON)

Kafka

topic=**NormalizedStructured**

```
JSON = {
   "srcIP": "10.3.1.1",
   "srcPort": "62902",
   "serviceType": "authentication",
   "regexMatch":
"sshdAcceptedSessionsLinux",
   "user": "jdoe",
   "product": "Linux",
   "@version": "1",
   "@timestamp": "2015-05-15T20:30:53.714Z"
 }
```

topic=**sshdAcceptedSessionsLinux**

```
JSON = {
   "srcIP": "10.3.9.1",
   "srcPort": "62902",
   "serviceType": "authentication",
   "regexMatch": "sshdAcceptedSessionsLinux",
   "user": "jdoe",
   "product": "Linux",
   "@version": "1",
   "@timestamp": "2015-05-15T20:30:53.714Z"
 }
```

topic=**srcIP**

1.1.1.1
2.2.2.2
4.4.4.4

topic=**User**

joe
jane
doe
john

.....

topic=**dstIP**

10.1.1.1
10.2.2.2
7.2.2.2
3.3.3.3

Production stats and lessons

- 100k EPS peak, 3 billion events/day
  - End-to-end delay of <20 seconds
- RSyslog to Kafka bottleneck
  - low producer instances - Only 2 RSyslog->Kafka Producers
- Spark specific configuration
  - Executors - 100
  - Memory - Total Yarn Memory: 201GB
  - CPU - 400 virtual threads/cores
- Parallel scheduling vs Union + Inherent-Partitioning
  - Had to use concurrent jobs (undocumented feature) until 1.1

Spark Streaming Choices

- Concurrent Jobs vs Union/Partitioning
- Kafka Read/Write choices
- Scale/Partition Kafka as per Spark cluster size

Issues

- None of the issues were because of Spark Streaming!
- Going from Spark 0.9 to 1.3
- YARN logAggregation
  - too verbose, crashing the executor nodes
- Yarn Client vs Cluster mode
  - failed at first attempt
- Too many config options, switches/knobs
  - hard to test/identify bottlenecks/bugs

More Issues

- Can’t debug issues in an IDE
- Can’t debug or identify bottlenecks with test data easily or test flows
- If something gets fixed eventually in Spark, don’t bother trying to find root cause of why it didn’t work earlier
  - You probably won’t be able to figure out in a reasonable amount of time (100s of commits per release!)

What it’s not

- All open source, no proprietary data stores or compute frameworks
- It is a platform, adaptable to any big data problem, no silver bullet or magic sauce
- Uses scalable data stores, no massive monolithic databases
- Mostly plumbing/integration work, no massive code to maintain
- Runs on commodity hardware, no custom hardware