ZOOKEEPER

Coordinating your cluster

What is ZooKeeper?

- It basically keeps track of information that must be synchronized across your cluster
  - *Which node is the master?*
  - *What tasks are assigned to which workers?*
  - *Which workers are currently available?*
- It's a tool that applications can use to recover from partial failures in your cluster.
- An integral part of HBase, High-Availability (HA) MapReduce, Drill, Storm, Solr, and much more

Failure modes

- Master crashes, needs to fail over to a backup
- Worker crashes – its work needs to be redistributed
- Network trouble – part of your cluster can't see the rest of it

"Primitive" operations in a distributed system

- Master election
  - *One node registers itself as a master, and holds a "lock" on that data*
  - *Other nodes cannot become master until that lock is released*
  - *Only one node allowed to hold the lock at a time*
- Crash detection
  - *"Ephemeral" data on a node's availability automatically goes away if the node disconnects, or fails to refresh itself after some time-out period.*
- Group management
- Metadata
  - *List of outstanding tasks, task assignments*

But ZooKeeper's API is not about these primitives.

- Instead they have built a more general purpose system that makes it easy for applications to implement them.

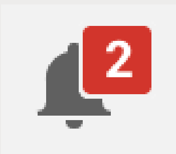
Zookeeper's API

- Really a little distributed file system
  - *With strong consistency guarantees*
  - *Replace the concept of "file" with "znode" and you've pretty much got it*
- Here's the ZooKeeper API:
  - *Create, delete, exists, setData, getData, getChildren*

```
/
├── /master      "master1.foobar.com:2223"
└── /workers
    ├── worker-1  "worker-2.foobar.com:2225"
    └── worker-2  "worker-5.foobar.com:2225"
```

Notifications

- A client can register for notifications on a znode
  - *Avoids continuous polling*
  - *Example: register for notification on /master - if it goes away, try to take over as the new master.*



Persistent and ephemeral znodes

- Persistent znodes remain stored until explicitly deleted
  - *i.e., assignment of tasks to workers must persist even if master crashes*
- Ephemeral znodes go away if the client that created it crashes or loses its connection to ZooKeeper
  - *i.e., if the master crashes, it should release its lock on the znode that indicates which node is the master!*

ZooKeeper Architecture

Master
ZK client

Worker
ZK client

Worker
ZK client

Worker
ZK client

Zookeeper server
Zookeeper server
Zookeeper server
Zookeeper server
Zookeeper server

Zookeeper ensemble

Clients have a list of Zookeeper servers to connect to.

ZooKeeper quorums

Zookeeper server
Zookeeper server
Zookeeper server
Zookeeper server
Zookeeper server

Zookeeper ensemble

ZooKeeper quorums

Zookeeper server
Zookeeper server
Zookeeper server
Zookeeper server
Zookeeper server

Zookeeper ensemble

Sounds a lot like how MongoDB works

Let’s play with the ZooKeeper.

OOZIE

Orchestrating your Hadoop jobs

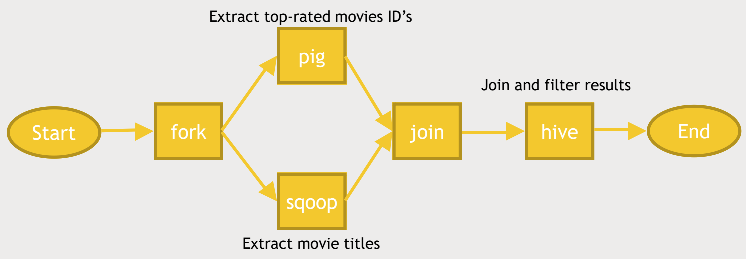
What is Oozie?

- Burmese for “elephant keeper”
- A system for running and scheduling Hadoop tasks

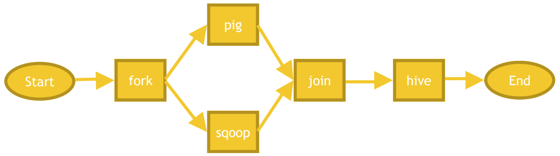
Workflows

- A multi-stage Hadoop job
  - *Might chain together MapReduce, Hive, Pig, sqoop, and distcp tasks*
  - *Other systems available via add-ons (like Spark)*
- A workflow is a Directed Acyclic Graph of actions
  - *Specified via XML*
  - *So, you can run actions that don’t depend on each other in parallel.*

Workflow example



Workflow XML structure




```xml
<?xml version="1.0" encoding="UTF-8"?>
<workflow-app xmlns="uri:oozie:workflow:0.2" name="top-movies">
    <start to="fork-node"/>

    <fork name="fork-node">
        <path start="sqoop-node" />
        <path start="pig-node" />
    </fork>

    <action name="sqoop-node">
        <sqoop xmlns="uri:oozie:sqoop-action:0.2">

        … sqoop configuration here …

        </sqoop>
        <ok to="joining"/>
        <error to="fail"/>
    </action>

    <action name="pig-node">
        <pig>

         … pig configuration here …

        </pig>
        <ok to="joining"/>
        <error to="fail"/>
    </action>

    <join name="joining" to="hive-node"/>

    <action name="hive-node">
        <hive xmlns="uri:oozie:hive-action:0.2">

        … hive configuration here …

        </hive>
        <ok to="end"/>
        <error to="fail"/>
    </action>

    <kill name="fail">
        <message>Sqoop failed, error
message[${wf:errorMessage(wf:lastErrorNode())}]</message>
    </kill>
    <end name="end"/>
</workflow-app>
```

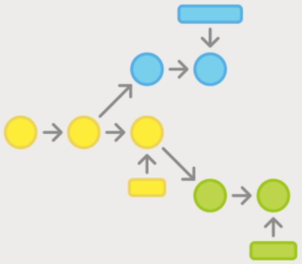
Steps to set up a workflow in Oozie

- Make sure each action works on its own
- Make a directory in HDFS for your job
- Create your workflow.xml file and put it in your HDFS folder
- Create job.properties defining any variables your workflow.xml needs
  - *This goes on your local filesystem where you'll launch the job from*
  - *You could also set these properties within your XML.*

```
nameNode=hdfs://sandbox.hortonworks.com:8020
jobTracker=http://sandbox.hortonworks.com:8050
queueName=default
oozie.use.system.libpath=true
oozie.wf.application.path=${nameNode}/user/maria_dev
```

Running a workflow with Oozie

- oozie job --oozie http://localhost:11000/oozie -config /home/maria_dev/job.properties -run
- Monitor progress at http://127.0.0.1:11000/oozie



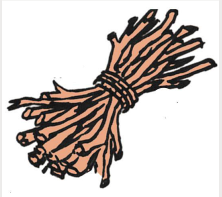
Oozie Coordinators

- Schedules workflow execution
- Launches workflows based on a given start time and frequency
- Will also wait for required input data to become available
- Run in exactly the same way as a workflow

```
<coordinator-app xmlns = "uri:oozie:coordinator:0.2" name = “sample coordinator" frequency = "5 * * * *" start = "2016-00-18T01:00Z" end = "2025-12-31T00:00Z"" timezone = "America/Los_Angeles">
        <controls>
                <timeout>1</timeout>
                 <concurrency>1</concurrency>
                <execution>FIFO</execution>
                <throttle>1</throttle>
        </controls>
        <action>
                <workflow>
                        <app-path>pathof_workflow_xml/workflow.xml</app-path>
                </workflow>
        </action>
</coordinator-app>
```

Oozie bundles

- New in Oozie 3.0
- A bundle is a collection of coordinators that can be managed together
- Example: you may have a bunch of coordinators for processing log data in various ways
  - *By grouping them in a bundle, you could suspend them all if there were some problem with log collection*



Let’s set up a simple workflow in Oozie.

- We’ll get movielens back into MySQL if it’s not still there
- Write a Hive script to find all movies released before 1940
- Set up an Oozie workflow that uses sqoop to extract movie information from MySQL, then analyze it with Hive

ZEPPELIN

A Notebook Interface to your Big Data

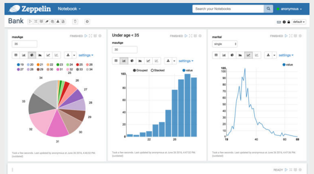
What is Zeppelin?

Apache Zeppelin

- If you're familiar with iPython notebooks – it's like that
  - *Lets you interactively run scripts / code against your data*
  - *Can interleave with nicely formatted notes*
  - *Can share notebooks with others on your cluster*
- If you're not familiar with iPython notebooks – well, you kind of just have to see it.

Apache Spark integration

- Can run Spark code interactively (like you can in the Spark shell)
  - *This speeds up your development cycle*
  - *And allows easy experimentation and exploration of your big data*
- Can execute SQL queries directly against SparkSQL
- Query results may be visualized in charts and graphs
- Makes Spark feel more like a data science tool!

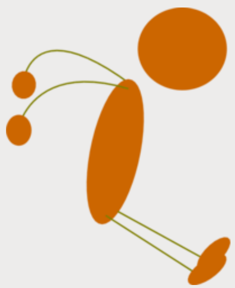


Zeppelin can do much more than Spark

ALLUXIO
Google BigQuery
cassandra
elasticsearch
Flink
APACHE GEODE
APACHE HBASE
hadoop HDFS
HIVE
Apache Ignite
LENS
LIVY
PostgreSQL
python
R
Scalding
TAJO

It'll make more sense if we just play with it.

- Zeppelin comes pre-installed on Hortonworks Data Platform
- So let's jump in



HUE

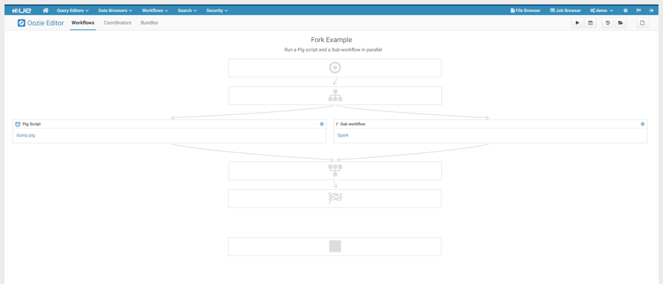
Hadoop User Experience

A Tale of Two Distros

- Hortonworks
  - *Ambari used for management and query / files UI*
  - *Zeppelin used for notebook*
- Cloudera
  - *Hue used for query / files UI and notebook*
  - *Cloudera Manager used for management*
- Hue is Cloudera's Ambari – sort of.

Cool things about Hue

- Oozie editor
  - *Also Spark, Pig, Hive, HBase, HDFS, Sqoop*
- Built-in notebooks



It *is* open source

- Not an Apache project; maintained by Cloudera
- It can be installed on a Hortonworks distribution if you try hard enough!

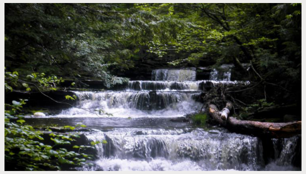
There's a live demo

- Let's play with it.

STREAMING WITH KAFKA

Publish/Subscribe Messaging with Kafka

What is streaming?



- So far we've really just talked about processing historical, existing big data
  - *Sitting on HDFS*
  - *Sitting in a database*
- But how does new data get into your cluster? Especially if it's "Big data"?
  - *New log entries from your web servers*
  - *New sensor data from your IoT system*
  - *New stock trades*
- Streaming lets you publish this data, in real time, to your cluster.
  - *And you can even process it in real time as it comes in!*

Two problems

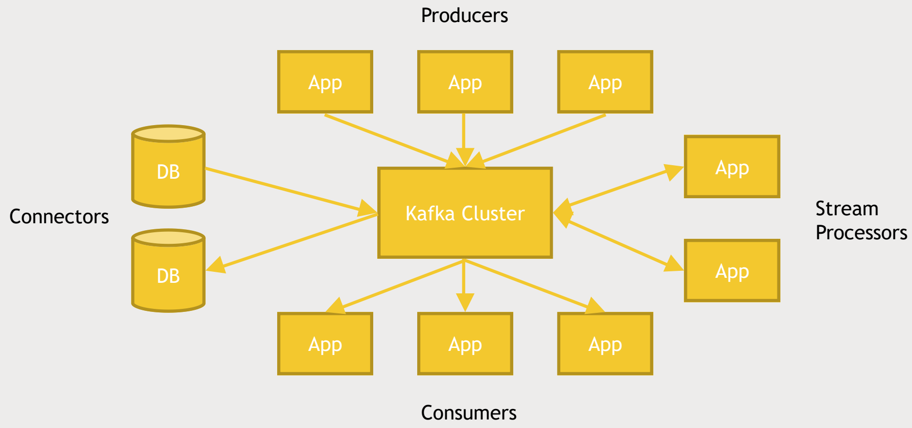
- How to get data from many different sources flowing into your cluster
- Processing it when it gets there
- First, let's focus on the first problem

Enter Kafka

kafka

- Kafka is a general-purpose **publish/subscribe messaging system**
- Kafka servers store all incoming messages from *publishers* for some period of time, and *publishes* them to a stream of data called a *topic*.
- Kafka *consumers* subscribe to one or more topics, and receive data as it's published
- A stream / topic can have many different consumers, all with their own position in the stream maintained
- It's not just for Hadoop

Kafka architecture



How Kafka scales

- Kafka itself may be distributed among many processes on many servers
  - *Will distribute the storage of stream data as well*
- Consumers may also be distributed
  - *Consumers of the same group will have messages distributed amongst them*
  - *Consumers of different groups will get their own copy of each message*

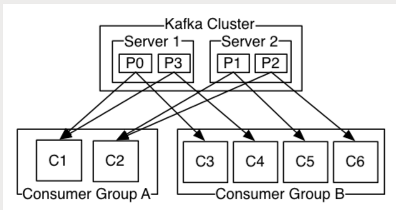


Image: kafka.apache.org

Let's play

- Start Kafka on our sandbox
- Set up a topic
  - *Publish some data to it, and watch it get consumed*
- Set up a file connector
  - *Monitor a log file and publish additions to it*

FLUME

More fun with data streaming

What is Flume?

- Another way to stream data into your cluster
- Made from the start with Hadoop in mind
  - *Built-in sinks for HDFS and Hbase*
- Originally made to handle log aggregation

Anatomy of a Flume Agent and Flow

Components of an agent

- Source
  - *Where data is coming from*
  - *Can optionally have Channel Selectors and Interceptors*
- Channel
  - *How the data is transferred (via memory or files)*
- Sink
  - *Where the data is going*
  - *Can be organized into Sink Groups*
  - *A sink can connect to only one channel*
    - Channel is notified to delete a message once the sink processes it.

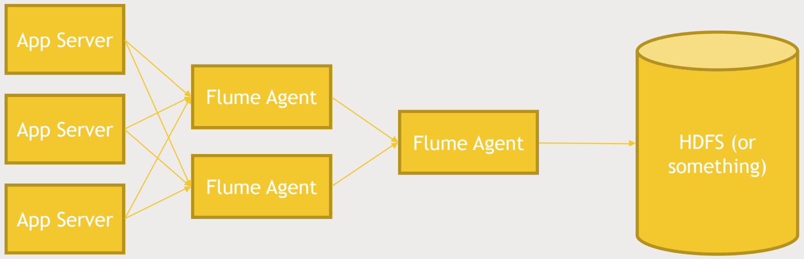
Built-in Source Types

- Spooling directory
- Avro
- Kafka
- Exec
- Thrift
- Netcat
- HTTP
- Custom
- And more!

Built-in Sink Types

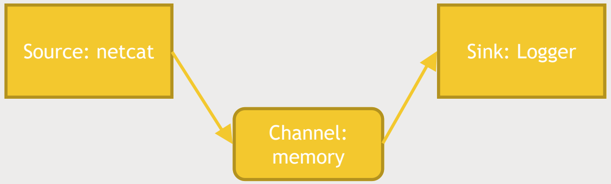
- HDFS
- Hive
- HBase
- Avro
- Thrift
- Elasticsearch
- Kafka
- Custom
- And more!

Using Avro, agents can connect to other agents as well

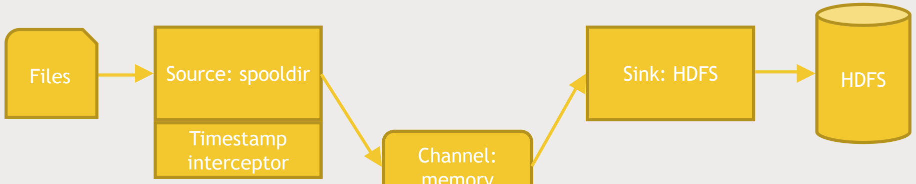


Think of Flume as a buffer between your data and your cluster.

Let’s play: Simple flow



Let’s play: log spool to HDFS



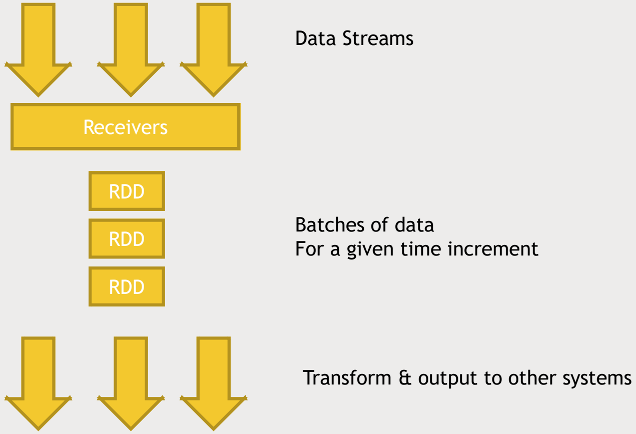
SPARK STREAMING

Processing continuous streams of data in near-real-time

Why Spark Streaming?

- “Big data” never stops!
- Analyze data streams in real time, instead of in huge batch jobs daily
- Analyzing streams of web log data to react to user behavior
- Analyze streams of real-time sensor data for “Internet of Things” stuff

Spark Streaming: High Level

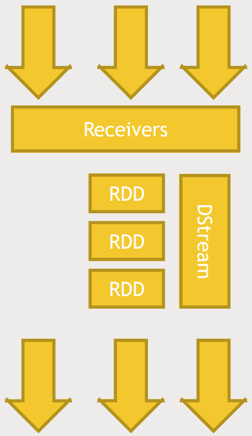


This work can be distributed

- Processing of RDD's can happen in parallel on different worker nodes

DStreams (Discretized Streams)

- Generates the RDD's for each time step, and can produce output at each time step.
- Can be transformed and acted on in much the same way as RDD's
- Or you can access their underlying RDD's if you need them.



Common stateless transformations on DStreams

- Map
- Flatmap
- Fliter
- reduceByKey

Stateful data

- You can also maintain a long-lived state on a Dstream
- For example – running totals, broken down by keys
- Another example: aggregating session data in web activity

WINDOWING

Windowed Transformations

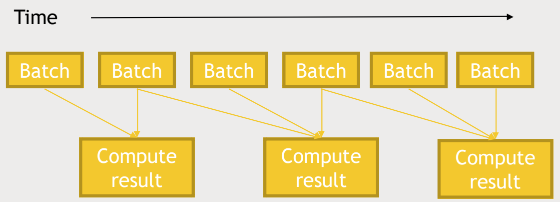
- Allow you to compute results across a longer time period than your batch interval
- Example: top-sellers from the past hour
  - *You might process data every one second (the batch interval)*
  - *But maintain a window of one hour*
- The window "slides" as time goes on, to represent batches within the window interval

Batch interval vs. slide interval vs. window interval

- The batch interval is how often data is captured into a Dstream
- The slide interval is how often a windowed transformation is computed
- The window interval is how far back in time the windowed transformation goes

Example

- Each batch contains one second of data (the batch interval)
- We set up a window interval of 3 seconds and a slide interval of 2 seconds



Windowed transformations: code

- The batch interval is set up with your SparkContext:

  ssc = StreamingContext(sc, 1)

- You can use reduceByWindow() or reduceByKeyAndWindow() to aggregate data across a longer period of time!

  hashtagCounts = hashtagKeyValues.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y : x - y, 300, 1)

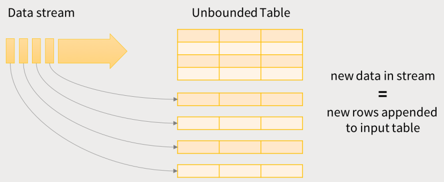
STRUCTURED STREAMING

What is structured streaming?

- A new, higher-level API for streaming structured data
  - *Available in Spark 2.0 and 2.1 as an experimental release*
  - *But it's the future.*
- Uses DataSets
  - *Like a DataFrame, but with more explicit type information*
  - *A DataFrame is really a DataSet[Row]*

Imagine a DataFrame that never ends

- New data just keeps getting appended to it
- Your continuous application keeps querying updated data as it comes in



Data stream as an unbounded Input Table

Advantages of Structured Streaming

- Streaming code looks a lot like the equivalent non-streaming code
- Structured data allows Spark to represent data more efficiently
- SQL-style queries allow for query optimization opportunities - and even better performance.
- Interoperability with other Spark components based on DataSets
  - *MLLib is also moving toward DataSets as its primary API.*
- DataSets in general is the direction Spark is moving

Once you have a SparkSession, you can stream data, query it, and write out the results.

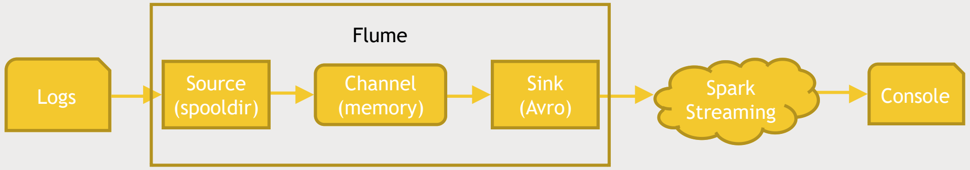
2 lines of code to stream in structured JSON log data, count up "action" values for each hour, and write the results to a database.

```
val inputDF = spark.readStream.json("s3://logs")
inputDF.groupBy($"action", window($"time", "1 hour")).count()
      .writeStream.format("jdbc").start("jdbc:mysql//...")
```

LET'S PLAY

Spark Streaming with Flume

- We'll set up Flume to use a spooldir source as before
- But use an Avro sink to connect it to our Spark Streaming job!
  - *Use a window to aggregate how often each unique URL appears from our access log.*
- Using Avro in this manner is a "push" mechanism to Spark Streaming
  - *You can also "pull" data by using a custom sink for Spark Streaming*



APACHE STORM

Real-time stream processing

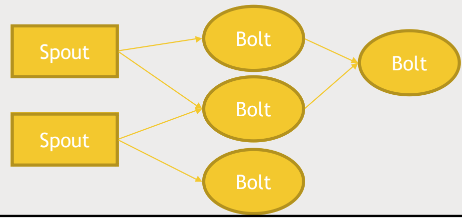
What is Storm?

APACHE STORM™

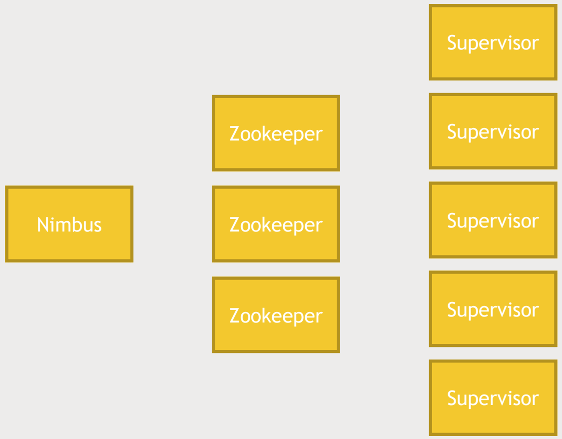
- Another framework for processing continuous streams of data on a cluster
  - *Can run on top of YARN (like Spark)*
- Works on individual events, not micro-batches (like Spark Streaming does)
  - *If you need sub-second latency, Storm is for you*

Storm terminology

- A *stream* consists of *tuples* that flow through…
- *Spouts* that are sources of stream data (Kafka, Twitter, etc.)
- *Bolts* that process stream data as it's received
  - *Transform, aggregate, write to databases / HDFS*
- A *topology* is a graph of spouts and bolts that process your stream



Storm architecture



Developing Storm applications

- Usually done with Java
  - *Although bolts may be directed through scripts in other languages*
- Storm Core
  - *The lower-level API for Storm*
  - *"At-least-once" semantics*
- Trident
  - *Higher-level API for Storm*
  - *"Exactly once" semantics*
- Storm runs your applications "forever" once submitted - until you explicitly stop them

Storm vs. Spark Streaming

- There's something to be said for having the rest of Spark at your disposal
- But if you need truly real-time processing (sub-second) of events as they come in, Storm's your choice
- Core Storm offers "tumbling windows" in addition to "sliding windows"
- Kafka + Storm seems to be a pretty popular combination

Let's Play

- We'll run the WordCount topology example and examine it.

FLINK

Another data stream framework!

What is Flink?

Flink

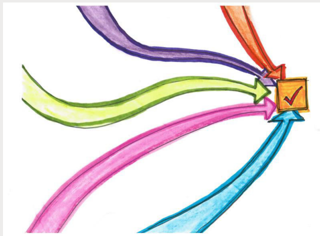
- German for quick and nimble
- Another stream processing engine – most similar to Storm
- Can run on standalone cluster, or on top of YARN or Mesos
- Highly scalable (1000's of nodes)
- Fault-tolerant
  - *Can survive failures while still guaranteeing exactly-once processing*
  - *Uses "state snapshots" to achieve this*
- Up & coming quickly

Flink vs. Spark Streaming vs. Storm

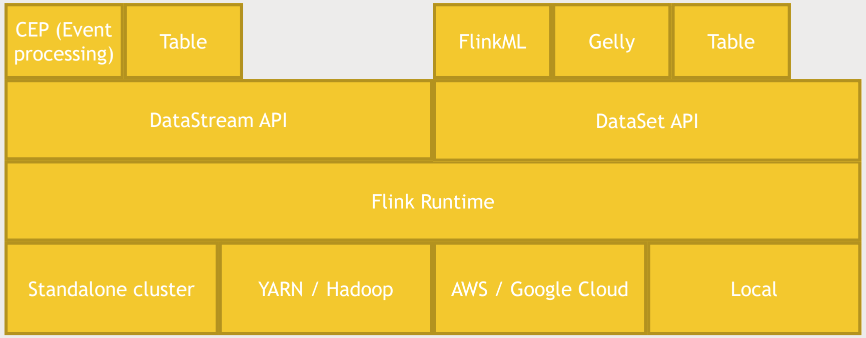
- Flink's faster than Storm
- Flink offers "real streaming" like Storm (but if you're using Trident with Storm, you're using micro-batches)
- Flink offers a higher-level API like Trident or Spark, but while still doing real-time streaming
- Flink has good Scala support, like Spark Streaming
- Flink has an ecosystem of its own, like Spark
- Flink can process data based on event times, not when data was received
  - *Impressive windowing system*
  - *This plus real-time streaming and exactly-once semantics is important for financial applications*
- But it's the youngest of the technologies

All three are converging it seems

- Spark Streaming's "Structured Streaming" paves the way for real event-based streaming in Spark
- Becomes more a question of what fits best in your existing environment



Flink architecture



Connectors

- HDFS
- Cassandra
- Kafka
- Others
  - *Elasticsearch, NiFi, Redis, RabbitMQ*

Let's Fiddle with Flink

THE BEST OF THE REST

Other relevant technologies, in brief

Impala

- Cloudera’s alternative to Hive
- Massively parallel SQL engine on Hadoop
- Impala’s always running, so you avoid the start-up costs when starting a Hive query
  - *Made for BI-style queries*
- Bottom line: Impala’s often faster than Hive, but Hive offers more versatility
- Consider using Impala instead of Hive if you’re using Cloudera

Accumulo

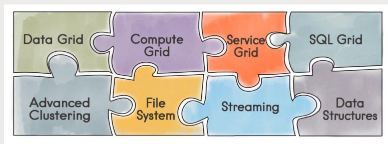
- Another BigTable clone (like HBase)
- But offers a better security model
  - *Cell-based access control*
- And server-side programming
- Consider it for your NoSQL needs if you have complex security requirements
  - *But make sure the systems that need to read this data can talk to it.*

Redis

- A distributed in-memory data store (like memcache)
- But it’s more than a cache!
- Good support for storing data structures
- Can persist data to disk
- Can be used as a data store and not just a cache
- Popular caching layer for web apps

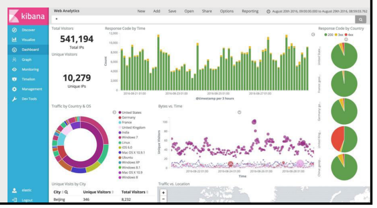
Ingite

- An “in-memory data fabric”
- Think of it as an alternative to Redis
- But it’s closer to a database
  - *ACID guarantees*
  - *SQL support*
  - *But it’s all done in-memory*



Elasticsearch

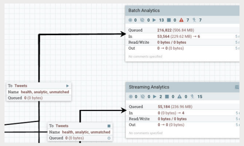
- A distributed document search and analytics engine
- Really popular
  - *Wikipedia, The Guardian, Stack Overflow, many more*
- Can handle things like real-time search-as-you-type
- When paired with Kibana, great for interactive exploration
- Amazon offers an Elasticsearch Service



Kinesis (and the AWS ecosystem)

- Amazon Kinesis is basically the AWS version of Kafka
- Amazon has a whole ecosystem of its own
  - *Elastic MapReduce (EMR)*
  - *S3*
  - *Elasticsearch Service / CloudSearch*
  - *DynamoDB*
  - *Amazon RDS*
  - *ElastiCache*
  - *AI / Machine Learning services*
- EMR in particular is an easy way to spin up a Hadoop cluster on demand

Apache NiFi



- Directed graphs of data routing
  - *Can connect to Kafka, HDFS, Hive*
- Web UI for designing complex systems
- Often seen in the context of IoT sensors, and managing their data
- Relevant in that it can be a streaming data source you'll see

Falcon

- A "data governance engine" that sits on top of Oozie
- Included in Hortonworks
- Like NiFi, it allows construction of data processing graphs
- But it's really meant to organize the flow of your data within Hadoop

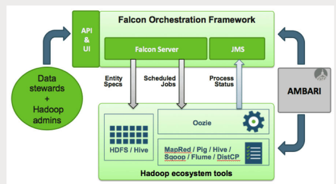


Image: falcon.apache.org

Apache Slider

- Deployment tool for apps on a YARN cluster
- Allows monitoring of your apps
- Allows growing or shrinking your deployment as it's running
- Manages mixed configurations
- Start / stop applications on your cluster
- Incubating

And many more…

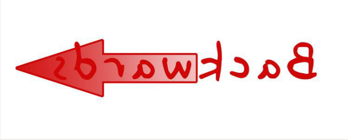
- Is your head spinning yet?

HADOOP ARCHITECTURE DESIGN

Putting the pieces together

Working backwards

- Start with the end user's needs, not from where your data is coming from
  - *Sometimes you need to meet in the middle*
- What sort of access patterns do you anticipate from your end users?
  - *Analytical queries that span large date ranges?*
  - *Huge amounts of small transactions for very specific rows of data?*
  - *Both?*
- What availability do these users demand?
- What consistency do these users demand?

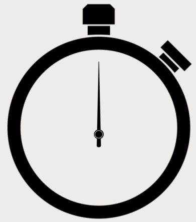


Thinking about requirements

- Just how big is your big data?
  - *Do you really need a cluster?*
- How much internal infrastructure and expertise is available?
  - *Should you use AWS or something similar?*
  - *Do systems you already know fit the bill?*
- What about data retention?
  - *Do you need to keep data around forever, for auditing?*
  - *Or do you need to purge it often, for privacy?*
- What about security?
  - *Check with Legal*

More requirements to understand

- Latency
  - *How quickly do end users need to get a response?*
    - Milliseconds? Then something like HBase or Cassandra will be needed
- Timeliness
  - *Can queries be based on day-old data? Minute-old?*
    - Oozie-scheduled jobs in Hive / Pig / Spark etc may cut it
  - *Or must it be near-real-time?*
    - Use Spark Streaming / Storm / Flink with Kafka or Flume



Judicious future-proofing

- Once you decide where to store your "big data", moving it will be really difficult later on
  - *Think carefully before choosing proprietary solutions or cloud-based storage*
- Will business analysts want your data in addition to end users (or vice versa?)

Cheat to win

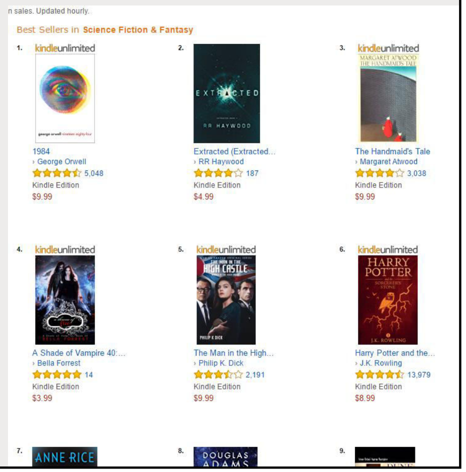
- Does your organization have existing components you can use?
  - *Don't build a new data warehouse if you already have one!*
  - *Rebuilding existing technology always has negative business value*
- What's the least amount of infrastructure you need to build?
  - *Import existing data with Sqoop etc. if you can*
  - *If relaxing a "requirement" saves lots of time and money - at least ask*

EXAMPLE: TOP SELLERS

Designing a system to keep track of top-selling items

What we want to build

- A system to track and display the top 10 best-selling items on an e-commerce website



What are our requirements? Work backwards!

- There are millions of end-users, generating thousands of queries per second
  - *It MUST be fast - page latency is important*
  - *So, we need some distributed NoSQL solution*
  - *Access pattern is simple: "Give me the current top N sellers in category X"*
- Hourly updates probably good enough (consistency not hugely important)
- Must be highly available (customers don't like broken websites)
- So – we want partition-tolerance and availability more than consistency

Sounds like Cassandra

cassandra

But how does data get into Cassandra?

- Spark can talk to Cassandra…
- And Spark Streaming can add things up over windows

OK, how does data get into Spark Streaming?

- Kafka or Flume - either works
- Flume is purpose-built for HDFS, which so far we haven't said we need
- But Flume is also purpose-built for log ingestion, so it may be a good choice
  - *Log4j interceptor on the servers that process purchases?*

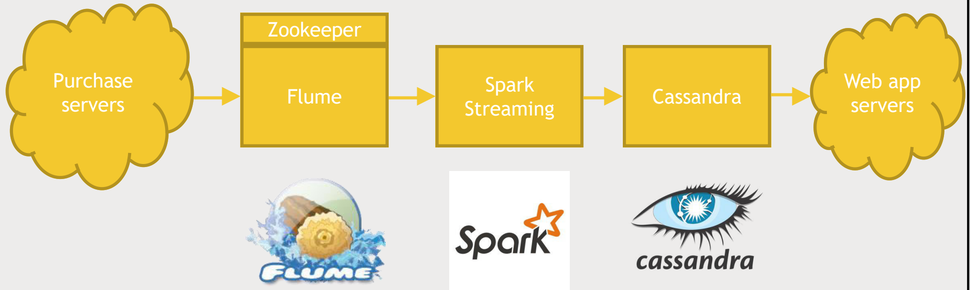
Don't forget about security

- Purchase data is sensitive - get a security review
  - *Blasting around raw logs that include PII* is probably a really bad idea*
  - *Strip out data you don't need at the source*
- Security considerations may even force you into a totally different design
  - *Instead of ingesting logs as they are generated, some intermediate database or publisher may be involved where PII is scrubbed*

*Personally Identifiable Information

So, something like this might work:

Interestingly, you *could* build this without Hadoop at all



But there’s more than one way to do it.

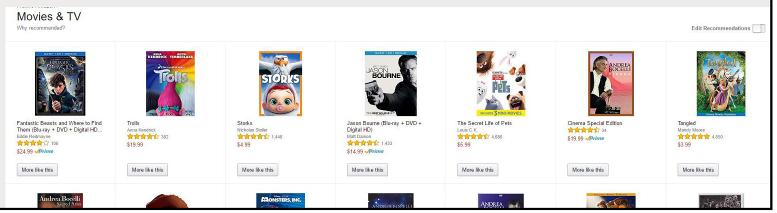
- Maybe you have an existing purchase database
  - *Instead of streaming, hourly batch jobs would also meet your requirements*
  - *Use Sqoop + Spark -> Cassandra perhaps?*
- Maybe you have in-house expertise to leverage
  - *Using Hbase, MongoDB, or even Redis instead of Cassandra would probably be OK.*
  - *Using Kafka instead of Flume - totally OK.*
- Do people need this data for analytical purposes too?
  - *Might consider storing on HDFS in addition to Cassandra.*

EXAMPLE: MOVIE RECOMMENDATIONS

Other movies you may like…

Working backwards

- Users want to discover movies they haven’t yet seen that they might enjoy
- Their own behavior (ratings, purchases, views) are probably the best predictors
- As before, availability and partition-tolerance are important. Consistency not so much.



Cassandra’s our first choice

- But any NoSQL approach would do these days

How do movie recommendations get into Cassandra?

- We need to do machine learning
  - *Spark MLLib*
  - *Flink could also be an alternative.*
- Timeliness requirements need to be thought out
  - *Real-time ML is a tall order - do you really need recommendations based on the rating you just left?*
  - *That kinda would be nice.*

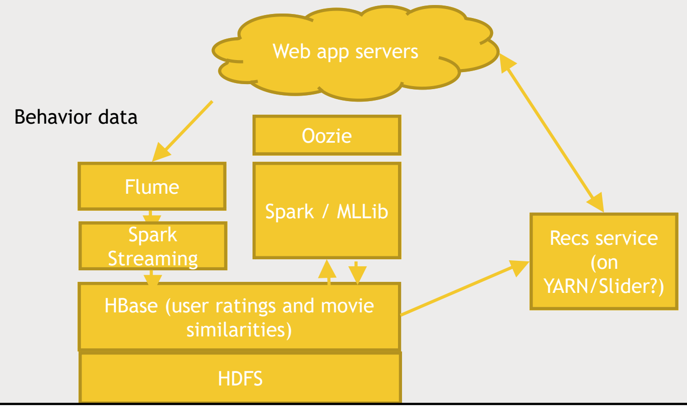
Creative thinking

- Pre-computing recommendations for every user
  - *Isn't timely*
  - *Wastes resources*
- Item-based collaborative filtering
  - *Store movies similar to other movies (these relationships don't change quickly)*
  - *At runtime, recommend movies similar to ones you've liked (based on real-time behavior data)*
- So we need something that can quickly look up movies similar to ones you've liked at scale
  - *Could reside within web app, but probably want your own service for this*
- We also need to quickly get at your past ratings /views /etc.

OK Then.

- So we'll have some web service to create recommendations on demand
- It'll talk to a fast NoSQL data store with movie similarities data
- And it also needs your past ratings / purchases /etc.
- Movie similarities (which are expensive) can be updated infrequently, based on log data with views / ratings / etc.

Something like this might work.



EXERCISE: DESIGN WEB ANALYTICS

Track number of sessions per day on a website

Your mission…

- You work for a big website
- Some manager wants a graph of total number of sessions per day
- And for some reason they don't want to use an existing service for this!

Requirements

- Only run daily based on previous day's activity
- Sessions are defined as traffic from same IP address within a sliding one hour window
  - *Hint: Spark Streaming etc. can handle "stateful" data like this.*
- Let's assume your existing web logs do not have session data in them
- Data is only used for analytic purposes, internally

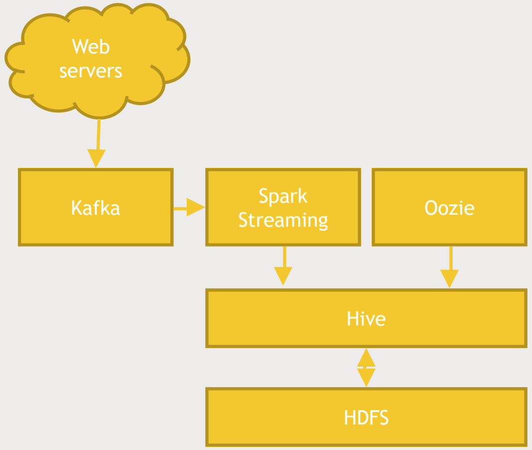
How would you do it?

- Things to consider:
  - *A daily SQL query run automatically is all you really need*
  - *But this query needs some table that contains session data*
    - And that will need to be built up throughout the day

EXERCISE: (A) SOLUTION

One way to solve the daily session count problem.

One way to do it.



There's no "right answer."

- And, it depends on a lot of things
  - *Have an existing sessions database that's updated daily? Just use sqoop to get at it*
  - *In fact, then you might not even need Hive / HDFS.*

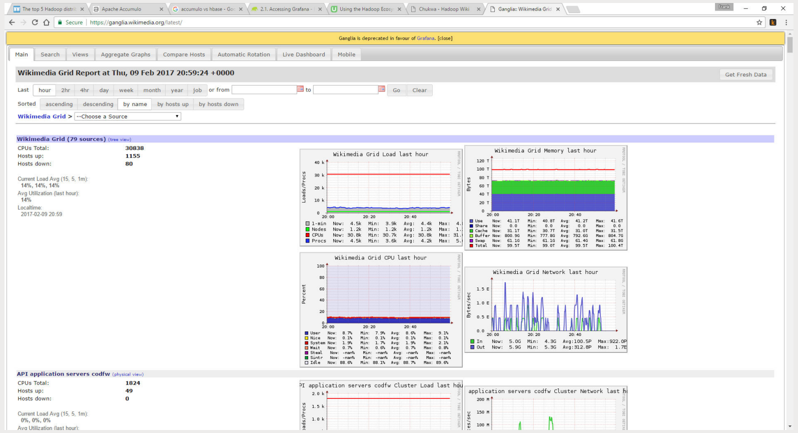
OTHER ADMINISTRATIVE TECHNOLOGIES

Some older systems you might hear about.

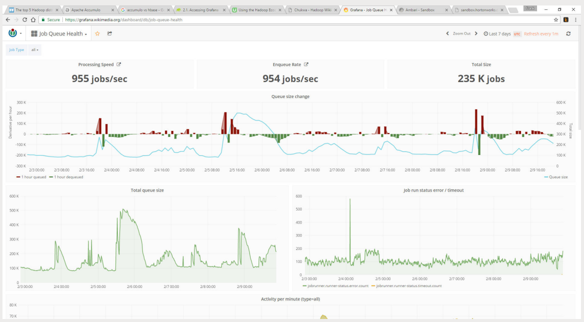
Ganglia

- Distributed monitoring system
  - *Developed by UC Berkeley*
  - *Originally widely used by universities*
  - *Wikimedia / Wikipedia used to use it.*
- Largely supplanted by Ambari / Cloudera Manager / Grafana

It ain't pretty



Compare that to Grafana / Ambari



Ganglia is dead

- Last updated in 2008
- Website is largely broken
- But you might encounter it in really old systems.

Chukwa

- System for collecting and analyzing logs from your Hadoop cluster\
- Initially adopted by NetFlix
- Largely supplanted by Flume and Kafka
  - *Both are more general purpose*
  - *Faster*
  - *Easier to set up*
  - *More reliable*

Chukwa is dead

- Hasn't changed since 2010
- Website is largely broken
- No usage to speak of today