Other Distributed Frameworks

Shannon Quinn
Distinction

1. General Compute Engines
   - Hadoop
2. User-facing APIs
   - Cascading
   - Scalding
Alternative Frameworks

1. Apache Mahout
2. Apache Giraph
3. GraphLab
4. Apache Storm
5. Apache Tez
6. Apache Flink
Alternative Frameworks

1. Apache Mahout
2. Apache Giraph
3. GraphLab
4. Apache Storm
5. Apache Tez
6. Apache Flink
Apache Mahout

- A Tale of Two Frameworks

1. Distributed machine learning on Hadoop
   - 0.1 to 0.9

2. "Samsara"
   - New in 0.10+
Machine learning on Hadoop

• Born out of the Apache Lucene project
• Built on Hadoop (all in Java)
• Pragmatic machine learning at scale
1: Recommendation

Here's a daily sample of items recommended for you. Click here to see all recommendations.

- Programming in Python 3: A Case Study (Paperback) by Mark Summerfield
  - Price: $26.68
  - Rating: 4 stars
  - Fix this recommendation

- Algorithm Design (Hardcover) by Jon Kleinberg
  - Price: $103.24
  - Fix this recommendation

- Econometrics (Hardcover) by Fumio Hayashi
  - Price: $74.55
  - Fix this recommendation

- Mathematical Statistics, Basic Principles and Applications (Paperback) by Peter J. Bickel
  - Price: $68.94
  - Fix this recommendation

Who to follow - Refresh - View all

Demetri Martin @DemetriMartin
Followed by waitwait and others
Follow

Josh Gates @JoshuaGates
Followed by David Blue and others
Follow

American Red Cross @RedCr... X
Followed by James Cotton and others
Follow

Critically-acclaimed Goofy Comedies

Your taste preferences created this row.

Comedies
Goofy
Critically-acclaimed.

- Young Frankenstein
- Shaolin Soccer
- Tom Hanks BIG
- CJ Gregg Tests Your Math
2: Classification
3: Clustering
Other MapReduce algorithms

• Dimensionality reduction
  – Lanczos
  – SSVD
  – LDA

• Regression
  – Logistic
  – Linear
  – Random Forest

• Evolutionary algorithms
Mahout-Samsara

- Programming “environment” for distributed machine learning
- R-like syntax
- Interactive shell (like Spark)
- Under-the-hood algebraic optimizer
- **Engine-agnostic**
  - Spark
  - H2O
  - Flink
  - ?
\[ G = BB^T - C - C^T + s_qs_q^T \xi^T \xi \]

\[
\text{val } g = bt.t \times bt - c - c.t + (s_q \times s_q) \times (xi \cdot xi) \]
Mahout

• 3 main components

- Engine-agnostic environment for building scalable ML algorithms (Samsara)
- Engine-specific algorithms (Spark, H2O)
- Legacy MapReduce algorithms
Mahout

• v0.10.0 released April 11 (as in, 5 days ago!)

• 0.10.1
  – More base linear algebra functionality

• 0.11.0
  – Compatible with Spark 1.3

• 1.0
  – ?
# Mahout features by engine

<table>
<thead>
<tr>
<th>Mahout Math-Scala Core Library and Scala DSL</th>
<th>Single Machine</th>
<th>MapReduce</th>
<th>Spark</th>
<th>H2O</th>
<th>Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mahout Distributed BLAS. Distributed Row Matrix API with R and Matlab like operators. Distributed ALS, SPCA, SSVD, thin-QR. Similarity Analysis.</td>
<td>x</td>
<td>x</td>
<td>in development</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Mahout Interactive Shell

Interactive REPL shell for Spark optimized Mahout DSL

### Dimensionality Reduction

*note: most scala-based dimensionality reduction algorithms are available through the Mahout Math-Scala Core Library for all engines*

<table>
<thead>
<tr>
<th>Singular Value Decomposition</th>
<th>Single Machine</th>
<th>MapReduce</th>
<th>Spark</th>
<th>H2O</th>
<th>Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lanczos Algorithm</th>
<th>Single Machine</th>
<th>MapReduce</th>
<th>Spark</th>
<th>H2O</th>
<th>Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>deprecated</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stochastic SVD</th>
<th>Single Machine</th>
<th>MapReduce</th>
<th>Spark</th>
<th>H2O</th>
<th>Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCA (via Stochastic SVD)</th>
<th>Single Machine</th>
<th>MapReduce</th>
<th>Spark</th>
<th>H2O</th>
<th>Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QR Decomposition</th>
<th>Single Machine</th>
<th>MapReduce</th>
<th>Spark</th>
<th>H2O</th>
<th>Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Mahout features by engine

### Collaborative Filtering

<table>
<thead>
<tr>
<th>Feature</th>
<th>Spark Engine</th>
<th>Hadoop Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-Based Collaborative Filtering</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Item-Based Collaborative Filtering</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Matrix Factorization with ALS</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Matrix Factorization with ALS on Implicit Feedback</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Weighted Matrix Factorization, SVD++</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

### Classification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Spark Engine</th>
<th>Hadoop Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression - trained via SGD</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Naive Bayes / Complementary Naive Bayes</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Random Forest</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Hidden Markov Models</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Multilayer Perceptron</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>
Mahout features by engine

- No engine-agnostic clustering algorithms yet
  - Still the domain of legacy MapReduce
- H2O and especially Flink still highly experimental
Alternative Frameworks

1. Apache Mahout
2. Apache Giraph
3. GraphLab
4. Apache Storm
5. Apache Tez
6. Apache Flink
Apache Giraph

- Vertex-centric alternative to Hadoop
  - Runs on Hadoop
Giraph

• “...an iterative graph processing system built for high scalability.”
• Bulk-synchronous Parallel (BSP) model of distributed computation
Bulk-synchronous Parallel

- Vertex-centric model
Giraph terminology

• Superstep
  – Sequence of iterations
  – Each “active” vertex invokes a compute() method
    • receives messages sent to the vertex in the previous superstep,
    • computes using the messages, and the vertex and outgoing edge values, which may result in modifications to the values, and
    • may send messages to other vertices.
Shortest path

• Example compute() method

```java
public void compute(Iterable<DoubleWritable> messages) {
    double minDist = Double.MAX_VALUE;
    for (DoubleWritable message : messages) {
        minDist = Math.min(minDist, message.get());
    }
    if (minDist < getValue().get()) {
        setValue(new DoubleWritable(minDist));
        for (Edge<LongWritable, FloatWritable> edge : getEdges()) {
            double distance = minDist + edge.getValue().get();
            sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
        }
    }
    voteToHalt();
}
```
Giraph terminology

• Barrier
  – The messages sent in any current superstep get delivered to the destination vertices only in the next superstep
  – Vertices start computing the next superstep after every vertex has completed computing the current superstep
Alternative Frameworks

1. Apache Mahout
2. Apache Giraph
3. GraphLab
4. Apache Storm
5. Apache Tez
6. Apache Flink
GraphLab / Dato

• Began as a PhD thesis at Carnegie Mellon University
• Like Mahout, a Tale of Two Frameworks

1. GraphLab 1.0, 2.0
   – Vertex-centric alternative to Hadoop for graph analytics (a la Apache Giraph)

2. Dato, GraphLab Create
   – ???
   – SaaS: front-facing Python API for interacting with [presumably] C++ backend on AWS
GraphLab: the early years

• Envisioned as a vertex-centric alternative to Hadoop and, in particular, Mahout
• Built in C++
• Liked to compare apples and oranges…
GraphLab to Dato

• Data Engineering
  – Extraction, transformation
  – Visualization
• Data Intelligence
  – Recommendation
  – Clustering
  – Classification
• Deployment
  – Creating services
Dato data structures

• SArray
  – An immutable, homogeneously typed array object backed by persistent storage. SArray is scaled to hold data that are much larger than the machine’s main memory. It fully supports missing values and random access. The data backing an SArray is located on the same machine as the GraphLab Server process. Each column in an SFrame is an SArray.

• SFrames
  – A tabular, columnmutable dataframe object that can scale to big data. The data in SFrame is stored column-wise on the GraphLab Server side, and is stored on persistent storage (e.g. disk) to avoid being constrained by memory size. Each column in an SFrame is a sizeimmutable SArray, but SFrames are mutable in that columns can be added and subtracted with ease. An SFrame essentially acts as an ordered dict of SArrays.

• SGraph
  – A scalable graph data structure. The SGraph data structure allows arbitrary dictionary attributes on vertices and edges, provides flexible vertex and edge query functions, and seamless transformation to and from SFrame.
GraphLab Create

• “Five-line recommender”

```python
import graphlab as gl
url = 'http://s3.amazonaws.com/dato-datasets/movie_ratings/training_data.csv'
data = gl.SFrame.read_csv(url, column_type_hints={"rating":int})
data.show()
model = gl.recommender.create(data, user_id="user", item_id="movie", target="rating")
results = model.recommend(users=None, k=5)
```

<table>
<thead>
<tr>
<th>user</th>
<th>movie</th>
<th>score</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marco Smith</td>
<td>Sex and the City: Season 6: Part 2</td>
<td>5.53567966735</td>
<td>1</td>
</tr>
<tr>
<td>Marco Smith</td>
<td>Sex and the City: Season 4</td>
<td>5.21305689132</td>
<td>2</td>
</tr>
<tr>
<td>Marco Smith</td>
<td>Sex and the City: Season 5</td>
<td>5.1795544318</td>
<td>3</td>
</tr>
<tr>
<td>Marco Smith</td>
<td>Sex and the City: Season 6: Part 1</td>
<td>5.15331420219</td>
<td>4</td>
</tr>
<tr>
<td>Marco Smith</td>
<td>friends: Season 4, High Noon</td>
<td>5.01739624059</td>
<td>5</td>
</tr>
<tr>
<td>Zion Smith</td>
<td>The Treasure of the Sierra Madre</td>
<td>3.5382391355</td>
<td>2</td>
</tr>
<tr>
<td>Zion Smith</td>
<td>Dial M for Murder</td>
<td>3.44573309934</td>
<td>3</td>
</tr>
<tr>
<td>Zion Smith</td>
<td>Band of Brothers</td>
<td>3.43282082593</td>
<td>4</td>
</tr>
<tr>
<td>Zion Smith</td>
<td>The Dish</td>
<td>3.40498876488</td>
<td>5</td>
</tr>
</tbody>
</table>
Alternative Frameworks

1. Apache Mahout
2. Apache Giraph
3. GraphLab
4. Apache Storm
5. Apache Tez
6. Apache Flink
Apache Storm

• “...doing for realtime processing what Hadoop did for batch processing.”
• Distributed realtime computation system
• Reliably process unbounded streams of data
• Common use cases
  – Realtime analytics
  – Online learning
  – Distributed RPC
  – [Your use case here]
Storm terminology

• Spouts
  – Source of streaming data
  – Kestrel, RabbitMQ, Kafka, JMS, databases (brokers)
  – Twitter Streaming API

• Bolts
  – Processes input streams to produce output streams
  – Functions, filters, joins, aggregations

• Topologies
  – Network of sprouts (vertices) and bolts (edges)
  – Arbitrarily complex multi-stage streaming operation
  – Run indefinitely once deployed
Storm

Topology

data

functions
Storm

```
Spout  <---  Bolt

“emit random number < 100”  ---  “multiply by 2”

(74)  ---  (148)
```
**Spout**

```java
1    public void nextTuple() {
2        final Random rand = new Random(); // normally this should be an instance field
3        int nextRandomNumber = rand.nextInt(100);
4        collector.emit(new Values(nextRandomNumber)); // auto-boxing
5    }
```

**Bolt**

```java
1    @Override
2    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3        this.collector = collector;
4    }
5
6    @Override
7    public void execute(Tuple tuple) {
8        Integer inputNumber = tuple.getInteger(0);
9        collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
10       collector.ack(tuple);
11    }
12
13    @Override
14    public void declareOutputFields(OutputFieldsDeclarer declarer) {
15        declarer.declare(new Fields("doubled-number"));
16    }
```
Performance

- 1M 100-byte messages per second per node
- Storm automatically restarts workers that fail
  - Workers which cannot be restarted on the original node are restarted on different nodes
  - Nimbus and Supervisor
- Guarantees each tuple will be fully processed
Highly configurable

• Usable with [virtually] any language
• Thrift definition for defining topologies
  – Thrift is language-agnostic, so topologies are as well
• So are spouts and bolts!
  – Non-JVM languages communicate over JSON protocols
  – Adapters available for Ruby, Python, JavaScript, and Perl
Alternative Frameworks

1. Apache Mahout
2. Apache Giraph
3. GraphLab
4. Apache Storm
5. Apache Tez
6. Apache Flink
Apache Tez

• “...aimed at building an application framework which allows for a complex directed-acyclic-graph [DAG] of tasks for processing data.”
• Distributed execution framework
• Express computation as a data flow graph
• Built on Hadoop’s YARN
The software stack
Apache Tez

- Separates application logic from parallel execution, resource allocation, and fault tolerance

<table>
<thead>
<tr>
<th>App</th>
<th>Tez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom application logic</td>
<td>Distributed parallel execution</td>
</tr>
<tr>
<td>Custom data format</td>
<td>Negotiating resources from the Hadoop framework</td>
</tr>
<tr>
<td>Custom data transfer technology</td>
<td>Fault tolerance and recovery</td>
</tr>
<tr>
<td></td>
<td>Horizontal scalability</td>
</tr>
<tr>
<td></td>
<td>Resource elasticity</td>
</tr>
<tr>
<td></td>
<td>Shared library of ready-to-use components</td>
</tr>
<tr>
<td></td>
<td>Built-in performance optimizations</td>
</tr>
<tr>
<td></td>
<td>Security</td>
</tr>
</tbody>
</table>
Workflow optimization

- Workflows that previously required multiple MR passes can be done in only one
Directed acyclic execution

- Vertices are data transformations
- Edges are data movement
// Define DAG
DAG dag = new DAG();

// Define Vertex
Vertex Map1 = new Vertex(Processor.class);

// Define Edge
Edge edge = Edge(Map1, Reduce1, SCATTER_GATHER, PERSISTED, SEQUENTIAL, Output.class, Input.class);

// Connect them
dag.addVertex(Map1).addEdge(edge)....
### Hive:

#### Broadcast Join

```sql
SELECT ss.ss_item_sk, ss.ss_quantity, avg_price, inv.inv_quantity_on_hand
FROM (select avg(ss_sold_price) as avg_price, ss_item_sk, ss_quantity_sk from store_sales
      group by ss_item_sk) ss
JOIN inventory inv
ON (inv.inv_item_sk = ss.ss_item_sk);
```

### Comparison

<table>
<thead>
<tr>
<th>Hive – MR</th>
<th>Hive – Tez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Sales scan. Group by and aggregation.</td>
<td>Store Sales scan. Group by and aggregation reduce size of this input.</td>
</tr>
<tr>
<td>Inventory and Store Sales (aggr.) output scan and shuffle join.</td>
<td>Inventory scan and Join</td>
</tr>
</tbody>
</table>

Broadcast edge
Comparison

- Read/write barrier between successive computations
- Overhead of launching a new job
- `map()` thread at the start of every job

- Engine has a global picture of the workflow
Alternative Frameworks

1. Apache Mahout
2. Apache Giraph
3. GraphLab
4. Apache Storm
5. Apache Tez
6. Apache Flink
Apache Flink

• [formerly StratoSphere]
• “Fast and reliable large-scale data processing engine”
• Incubation in April 2014, TLP in December 2014
Selling points

• Fast
  – In-memory computations (like Spark)
  – Integrates iterative processing
Selling points

• Reliable and scalable
  – Designed to keep working when memory runs out
  – Contains its own memory management, serialization, and type inference frameworks
Selling points

• Ease of use
  – Very few configuration options required
  – Infers most of the configuration itself
Ease of use

• No memory thresholds to configure
  – Flink manages its own memory
• Requires no network configuration
  – Only needs slave information
• Needs no configured serializers
  – Flink handles this internally
• Programs automatically adjust to data type
  – Flink’s internals dynamically choose execution strategies
Architecture
Software stack
Flink engine
### Flink engine

<table>
<thead>
<tr>
<th>API</th>
<th>MapReduce on k/v pairs</th>
<th>k/v pair Readers/ Writers</th>
<th>Transformations on k/v pair collections</th>
<th>Iterative transformations on collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>MapReduce</td>
<td>DAG</td>
<td>RDD</td>
<td>Cyclic dataflows</td>
</tr>
<tr>
<td>Optimization</td>
<td>none</td>
<td>none</td>
<td>Optimization of SQL queries</td>
<td>Optimization in all APIs</td>
</tr>
<tr>
<td>Execution</td>
<td>Batch sorting</td>
<td>Batch sorting and partitioning</td>
<td>Batch with memory pinning</td>
<td>Stream with out-of-core algorithms</td>
</tr>
</tbody>
</table>
Flink engine

- On-the-fly program optimization

```java
DataSet<Tuple...> large = env.readCsv(...);
DataSet<Tuple...> medium = env.readCsv(...);
DataSet<Tuple...> small = env.readCsv(...);

DataSet<Tuple...> joined1 = large.join(medium).where(3).equals(1)
    .with(new JoinFunction() { ... });

DataSet<Tuple...> joined2 = small.join(joined1).where(0).equals(2)
    .with(new JoinFunction() { ... });

DataSet<Tuple...> result = joined2.groupBy(3).aggregate(MAX, 2);
```

Possible execution

1) Partitioned hash-join
2) Broadcast hash-join
3) Grouping /Aggregation reuses the partitioning from step (1) → No shuffle!!!
WordCount!

- Uses Scala, just like Spark

```scala
case class Word (word: String, frequency: Int)

val env = ExecutionEnvironment.getExecutionEnvironment()

val lines = env.readTextFile(...)

lines
  .flatMap {line => line.split(" ").map(word => Word(word,1))}
  .groupBy("word").sum("frequency")
  .print()

env.execute()
```
Flink API

- map, flatMap, filter, groupBy, reduce, reduceGroup, aggregate, join, coGroup, cross, project, distinct, union, iterate, iterateDelta...
- All Hadoop InputFormats supported
- Windowing functions for streaming data
- Counters, accumulators, broadcast variables
- Local standalone mode for testing/debugging
Flink philosophy

• Developers made a concerted effort to hide internals from Flink users

• The Good
  – Anyone who has had OutOfMemoryExceptions in Spark will probably agree this is a very good thing

• The Bad
  – Execution model is much more complicated than Hadoop or Spark
Flink internals

- Programs are *not* executed eagerly

- Flink compiles program to an “execution plan”
  - Essentially a pipeline, rather than a staged or batched execution
Iterative processing on Flink

```java
DataSet<Page> pages = ...
DataSet<Neighborhood> edges = ...
DataSet<Page> oldRanks = pages; DataSet<Page> newRanks;

for (i = 0; i < maxIterations; i++) {
    newRanks = update(oldRanks, edges)
    oldRanks = newRanks
}

DataSet<Page> result = newRanks;
```

```java
DataSet<Page> update (DataSet<Page> ranks, DataSet<Neighborhood> adjacency) {
    return oldRanks
    .join(adjacency)
    .where("id").equalTo("id")
    .with ((page, adj, out) -> {
        for (long n : adj.neighbors)
            out.collect(new Page(n, df * page.rank / adj.neighbors.length))
    })
    .groupBy("id")
    .reduce ((a, b) -> new Page(a.id, a.rank + b.rank ));
```
Iterative processing

• Hadoop, Spark, etc
  – Iterate by unrolling: loop submits one job per iteration
  – Data reuse by caching in memory and/or disk
Iterate natively [with delta]
**Flink summary**

- Flink decouples API from execution
  - Same program can be executed in many different ways
  - Ideally users do not care about this
- Pipelined execution, native iterations, program optimizer, serialized data manipulation
- Equivalent or better performance to Spark
Resources

• Apache Mahout
  – http://mahout.apache.org/users/sparkbindings/home.html

• Apache Giraph
  – http://www.slideshare.net/ClaudioMartella/giraph-at-hadoop-summit-2014

• GraphLab / Dato
  – https://dato.com/

• Apache Storm
  – http://www.slideshare.net/miguno/apache-storm-09-basic-training-verisign

• Apache Tez
  – http://www.slideshare.net/Hadoop_Summit/w-1205phall1saha

• Apache Flink
  – https://flink.apache.org/material.html