Apache Mahout!

What’s Next?

About Trevor

Trevor Grant

Committer - Apache Mahout

Contributor - Apache Streams-incubating, Apache Flink, Apache Zeppelin

Open Source Technical Evangelist, IBM

@rawkintrevo

trevor.d.grant@gmail.com

http://www.rawkintrevo.org

Overview

- **Intro**
- Getting to Know Mahout
- Rolling our own distributed algorithms
- How it works...
- Future: What’s Coming …
- Conclusion and Questions

Data Science Vs. Data Engineers

- Data Scientists: Complex algos- use R/Python
- Data Engineers: Implement those algos in distributed forms
- Wasteful division, gap is narrowing...

Overview

- Intro
- **Getting to Know Mahout**
    - What is Mahout?
    - What is Mahout Samsara?
    - Mahout Samsara DSL basics
- Rolling our own algorithms
- How it works...
- Future: What’s Coming …
- Conclusion and Questions

Introduction to Apache Mahout

Apache Mahout is an environment for creating scalable, performant, machine-learning applications

Apache Mahout provides:

- A simple and extensible programming environment and framework for building scalable algorithms
- A variety of pre-made algorithms for Scala + Apache Spark, Apache Flink, H2O
- Samsara, a vector math experimentation environment with R-like syntax which works at scale
- Notebook Integration packaged with Apache Zeppelin

Getting Apache Mahout

Mahout is a *Scala/Java Library*

Mahout Spark-Shell (CLI) available in Mahout Binaries Download
OR include Mahout in compiled Java/Scala JARs
https://mahout.apache.org/general/downloads.html

OR
Apache Zeppelin 0.7,
`$python scripts/mahout/add_mahout_interpreters.py`
creates %sparkMahout and %flinkMahout interpreters
(Requires Spark 1.6 build of Zeppelin (default is Spark 2.0))

Competitive Positioning

- SparkML / MLLib - Bag of Algorithms- Good for Demos, difficult to extend.
- Apache SystemML Incubating - Requires its own scripting language, Spark only execution
- Tensor Flow - Requires own cluster (at best can piggy-back off of Spark) - Python Based
- Sklearn / R - Single Node Bound
    - SparkR - PySpark: Not actually full sklearn / R- only API for using standard SparkML functions (common misconception)

Competitive Positioning: A Python Analogy

- Spark Dataframes -> Pandas
- SparkML -> (a weak version of) sklearn
- Mahout -> numpy

Recent Work on the Project

- Mahout-Samsara vector-math DSL released in August 2015 in 0.11.0 release
- New Mahout Book - ‘Apache Mahout: Beyond MapReduce’ by Dmitriy Lyubimov and Andrew Palumbo - Feb 2016
- Support for Apache Flink as a backend execution engine added in 0.12.0 release - April 2016
- Integration with Apache Zeppelin Notebooks - Nov 2016
- In Progress: Support for Native backend and ability to run algorithms in hybrid distributed - CPU/GPU environments
- Soon: Spark 2.0 Support

An ‘Ideal’ Language for Distributed Math/DS

- Mathematically Expressive
- Based in another language
- Distributed yet engine independent
- Algebraic Optimizer

Mahout-Samsara: Mathematically Expressive

Mahout-Samsara is an easy-to-use domain-specific language (DSL) for large-scale machine learning on distributed systems like Apache Spark/Flink

- Uses **Scala** as programming/scripting environment
- **algebraic expression optimizer** for distributed linear algebra
  - provides a translation layer to distributed engines
  - Support for Spark, Flink DataSets and H2O
- **System-agnostic, R-like DSL**; actual formula from **(d)spca:**

$$G = BB^T - C - C^T + \xi^T \xi s_q^T s_q$$

```
val G = B %*% B.t - C - C.t + (ksi dot ksi) * (s_q cross s_q)
```

Mahout-Samsara: Scala Based

Benefits of being based in another (popular) language.

- IDE Supported (e.g. IntelliJ / Eclipse)
- Leveraging other features / packages of host language
- Compile into (back end independent) JARs, import into other programs/workflows

Mahout-Samsara: Distributed Engine Agnostic

```
%flinkMahout                                                    READY

// Imports and creating the distributed context, similar but not exactly the same ////////////////////////////////////
////////
import org.apache.flink.api.scala._
import org.apache.mahout.math.drm._
import org.apache.mahout.math.drm.RLikeDrmOps._
import org.apache.mahout.flinkbindings._
import org.apache.mahout.math._
import scalabindings._
import RLikeOps._


implicit val ctx = new FlinkDistributedContext(benv)

// CODE IS EXACTLY THE SAME FROM HERE ON - R-Like DSL ///////////////////////////////////////////////////////////////////
/////////////////

val drmData = drmParallelize(dense(
  (2, 2, 10.5, 10, 29.509541),  // Apple Cinnamon Cheerios
  (1, 2, 12,   12, 18.042851),  // Cap'n'Crunch
  (1, 1, 12,   13, 22.736446),  // Cocoa Puffs
  (2, 1, 11,   13, 32.207582),  // Froot Loops
  (1, 2, 12,   11, 21.871292),  // Honey Graham Ohs
  (2, 1, 16,   8,  36.187559),  // Wheaties Honey Gold
  (6, 2, 17,   1,  50.764999),  // Cheerios
  (3, 2, 13,   7,  40.400208),  // Clusters
  (3, 3, 13,   4,  45.811716)), numPartitions = 2)

drmData.collect(::, 0 until 4)

val drmX = drmData(::, 0 until 4)
val y = drmData.collect(::, 4)
val drmXtX = drmX.t %*% drmX
val drmXty = drmX.t %*% y


val XtX = drmXtX.collect
val Xty = drmXty.collect(::, 0)
val beta = solve(XtX, Xty)
```

```
%sparkMahout                                                    READY

// Imports and creating the distributed context, similar but not exactly the same ////////////////////////////////////
///////

import org.apache.mahout.math._
import org.apache.mahout.math.scalabindings._
import org.apache.mahout.math.drm._
import org.apache.mahout.math.scalabindings.RLikeOps._
import org.apache.mahout.math.drm.RLikeDrmOps._
import org.apache.mahout.sparkbindings._

implicit val sdc: org.apache.mahout.sparkbindings.SparkDistributedContext = sc2sdc(sc)


// CODE IS EXACTLY THE SAME FROM HERE ON - R-Like DSL ///////////////////////////////////////////////////////////////////
////////////////

val drmData = drmParallelize(dense(
  (2, 2, 10.5, 10, 29.509541),  // Apple Cinnamon Cheerios
  (1, 2, 12,   12, 18.042851),  // Cap'n'Crunch
  (1, 1, 12,   13, 22.736446),  // Cocoa Puffs
  (2, 1, 11,   13, 32.207582),  // Froot Loops
  (1, 2, 12,   11, 21.871292),  // Honey Graham Ohs
  (2, 1, 16,   8,  36.187559),  // Wheaties Honey Gold
  (6, 2, 17,   1,  50.764999),  // Cheerios
  (3, 2, 13,   7,  40.400208),  // Clusters
  (3, 3, 13,   4,  45.811716)), numPartitions = 2)

drmData.collect(::, 0 until 4)

val drmX = drmData(::, 0 until 4)
val y = drmData.collect(::, 4)
val drmXtX = drmX.t %*% drmX
val drmXty = drmX.t %*% y


val XtX = drmXtX.collect
val Xty = drmXty.collect(::, 0)
val beta = solve(XtX, Xty)
```

An ‘Ideal’ Language for Distributed Math/DS

- ✓ Mathematically Expressive
- ✓ Based in another language
- ✓ Distributed yet engine independent
- ✓ Algebraic Optimizer (Will Show this in a bit)

Mahout-Samsara Language Guide / Overview

In Core Algebra Reference
Distributed Algebraic Reference

Samsara: Data Types

- Scalar real values

```
val x = 2.367
```

- In-memory vectors
  - dense
  - 2 types of sparse

```
val v = dvec(1, 0, 5)
val w = svec((0 -> 1)::(2 -> 5)::Nil)
```

- In-memory matrices
  - sparse and dense
  - a number of specialized matrices

```
val A = dense((1, 0, 5),
              (2, 1, 4),
              (4, 3, 1))
```

Samsara: Data Types- DRMs

Distributed Row Matrices (DRM)

- huge matrix, partitioned by rows
- lives in the main memory of the cluster
- provides small set of parallelized operations
- lazily evaluated operation execution

```
val drmA = drmDfsRead(...)
```

Features (1)

Matrix, vector, scalar operators: in-memory, distributed

```
drmA %*% drmB
A %*% x
A.t %*% drmB
A * B
```

Slicing operators

```
A(5 until 20, 3 until 40)
A(5, ::); A(5, 5)
x(a to b)
```

Assignments (in-memory only)

```
A(5, ::) := x
A *= B
A -=: B; 1 /:= x
```

Vector-specific

```
x dot y; x cross y
```

Features (2)

Summaries

```
A.nrow; x.length; A.colSums;
B.rowMeans;
A.norm
```

Solving linear systems

```
val x = solve(A, b)
```

In-memory decompositions

```
val (inMemQ, inMemR) = qr(inMemM)
val ch = chol(inMemM)
val (inMemV, d) = eigen(inMemM)
val (inMemU, inMemV, s) =
svd(inMemM)
```

Features (3)

Distributed decompositions

```
val (drmQ, inMemR) = thinQR(drmA)
val (drmU, drmV, s) =
  dssvd(drmA, k = 50, q = 1)
```

Caching of DRMs

```
val drmA_cached = drmA.checkpoint()
drmA_cached.uncache()
```

Unary Operators

In-Core

```
mahout> val mxA = dense((1,2,3),(3,4,5))
mxA: org.apache.mahout.math.DenseMatrix =
{
 0 =>	{0:1.0,1:2.0,2:3.0}
 1 =>	{0:3.0,1:4.0,2:5.0}
}

mahout> mlog(mxA)
res2: org.apache.mahout.math.Matrix =
{
 0 =>	{1:0.6931471805599453,2:1.0986122886681098}
 1 =>	{0:1.0986122886681098,1:1.3862943611198906,2:1.6094379124341003}
}

mahout> msignum(mxA)
res3: org.apache.mahout.math.Matrix =
{
 0 =>	{0:1.0,1:1.0,2:1.0}
 1 =>	{0:1.0,1:1.0,2:1.0}}
```

In-Core (Contd)

```
// add some negative numbers in
mahout> val mxB = dense((-1,2,-3),(-3,4,-5))
mxB: org.apache.mahout.math.DenseMatrix =
{
 0 =>    {0:-1.0,1:2.0,2:-3.0}
 1 =>    {0:-3.0,1:4.0,2:-5.0}
}

mahout> msignum(mxB)
res7: org.apache.mahout.math.Matrix =
{
 0 =>    {0:-1.0,1:1.0,2:-1.0}
 1 =>    {0:-1.0,1:1.0,2:-1.0}
}
```

Distributed Row Matrix (DRM)

```
mahout> val drmA = drmParallelize(mxA)

mahout> dlog(drmA).collect

res10: org.apache.mahout.math.Matrix =
{
 0 =>    {1:0.6931471805599453,2:1.0986122886681098}
 1 =>    {0:1.0986122886681098,1:1.3862943611198906,2:1.6094379124341003}
}
```

Overview

- Intro
- Getting to Know Mahout
- **Rolling our own algorithms**
  - OLS and Time Series Example
  - Eigenfaces Example
  - Visualization
- How it works...
- Future: What’s Coming …
- Conclusion and Questions

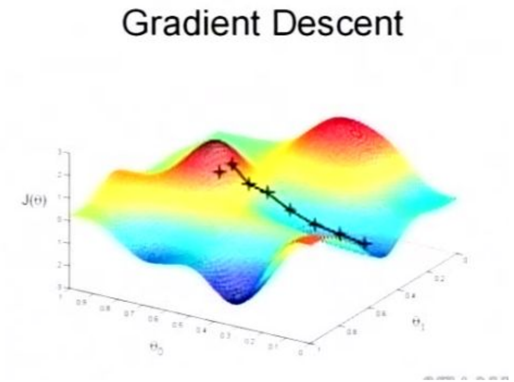
Ordinary Least Squares Regression , Serial Correlation, and Corrective Procedures

Motivation for Stochastic Gradient Descent

Useful method to approximate closed form solution.

1. Guess a solution
2. Check error
3. Slightly nudge parameters of solution
4. Repeat 2 and 3 for specified number of iterations or until error is acceptable

E.g. “walking down the error slope”



Useful when impossible to calculate closed form solution (closed form generally preferred method).

Statistician loses statistical inference of parameter estimates (standard errors) etc.

Motivation for Apache Mahout

Stochastic Gradient Descent: Popular solving method implemented in-

- Apache Spark - SparkML, MLLib
- Apache Flink - FlinkML
- Sklearn
- etc.

Good trick for approximating solutions when data set was too big…

In Apache Mahout we solve closed form equations.

Linear Regression

- Let's predict customer rating of a cereal brand based on its ingredients

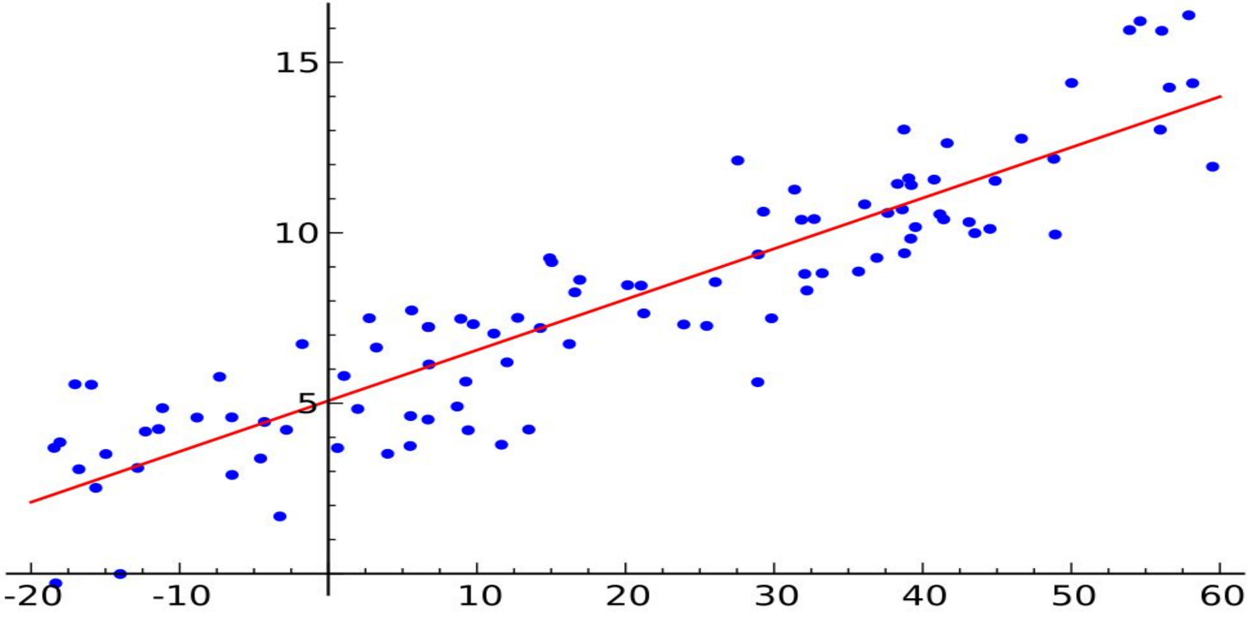
  X = weights of **ingredients**

  y = **customer rating**

- Assumption: **target variable y** generated by linear combination of **feature matrix X** with **parameter vector β**, plus **noise ε**

$$y = X\beta + \varepsilon$$

- Goal: find **estimate of the parameter vector β** that explains the data well


15
10
5
-20
-10
10
20
30
40
50
60

Data Ingestion

- Usually: load dataset as DRM from a distributed filesystem:

```
val drmData = drmDfsRead(...)
```

- Simple dataset for our example:

```
val drmData = drmParallelize(dense(
    (2, 2, 10.5, 10, 29.509541),    // Apple Cinnamon Cheerios
    (1, 2, 12,   12, 18.042851),    // Cap'n'Crunch
    (1, 1, 12,   13, 22.736446),    // Cocoa Puffs
    (2, 1, 11,   13, 32.207582),    // Froot Loops
    (1, 2, 12,   11, 21.871292),    // Honey Graham Ohs
    (3, 3, 13,   4,  45.811716)),   // Great Grains Pecan
  numPartitions = 2)
```

Data Preparation

- Cereals example: target variable y is **customer rating**, weights of **ingredients** are features X

- extract X as DRM by slicing,
fetch y as in-core vector

```
val drmX = drmData(::, 0 until 4)

val y = drmData.collect(::, 4)
```

drmX				y
2	2	10.5	10	29.509541
1	2	12	12	18.042851
1	1	12	13	22.736446
2	1	11	13	32.207582
1	2	12	11	21.871292
2	1	16	8	36.187559
6	2	17	1	50.764999
3	2	13	7	40.400208
3	3	13	4	45.811716

Estimating β

- **Ordinary Least Squares**: minimizes the sum of residual squares between true target variable and prediction of target variable
- Closed-form expression for estimation of *β* as

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

- Computing $\mathbf{X^T X}$ and $\mathbf{X^T y}$ is as simple as:

```
val drmXtX = drmX.t %*% drmX
val drmXty = drmX.t %*% y
```

Estimating β

- Solve the following linear system to get least-squares estimate of ß

$$X^T X \hat{\beta} = X^T y$$

- Fetch X^TX and X^Ty onto the driver and use an in-core solver
  - –assumes X^TX fits into memory
  - –use analogous to R's *solve()* function

```
val XtX = drmXtX.collect
val Xty = drmXty.collect(::, 0)

val betaHat = solve(XtX, Xty)
```

Estimating β

- Solve the following linear system to get least-squares estimate of ß

$$X^T X \hat{\beta} = X^T y$$

- Fetch X^TX and X^Ty onto the driver and use an in-core solver
  - –assumes X^TX fits into memory
  - –use analogous to R's *solve()* function

```
val XtX = drmXtX.collect
val Xty = drmXty.collect(::, 0)

val betaHat = solve(XtX, Xty)
```

→ We have implemented distributed linear regression!

Goodness of fit

- Prediction of the target variable is simple matrix-vector multiplication

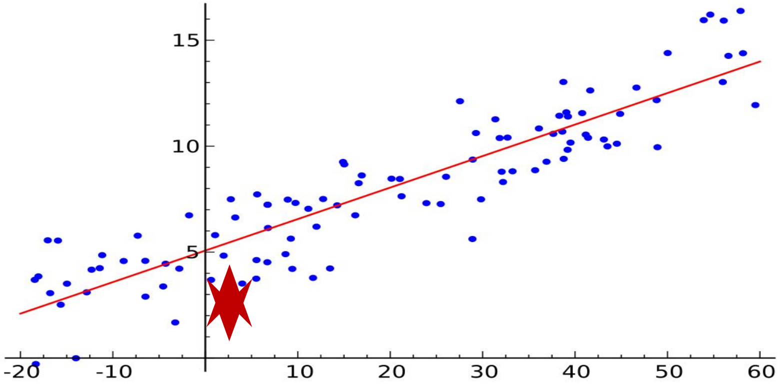
$$\hat{y} = X\hat{\beta}$$

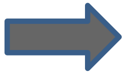
- Check L2 norm of the difference between true target variable and our prediction

```
val yHat = (drmX %*% betaHat).collect(::, 0)
(y - yHat).norm(2)
```

Adding a bias term

- **Bias term** left out so far
  - constant factor added to the model, “shifts the line vertically”
- **Common trick** is to add a column of ones to the feature matrix
  - bias term will be learned automatically





Adding a bias term

- How do we add a new column to a DRM?
  - → **mapBlock()** allows for custom modifications of the matrix

```
val drmXwithBiasColumn = drmX.mapBlock(ncol = drmX.ncol + 1) {
  case(keys, block) =>
    // create a new block with an additional column
    val blockWithBiasCol = block.like(block.nrow, block.ncol+1)
    // copy data from current block into the new block
    blockWithBiasCol(::, 0 until block.ncol) := block
    // last column consists of ones
    blockWithBiasColumn(::, block.ncol) := 1

    keys -> blockWithBiasColumn
  }
```

Adding a bias term (easy way)

```
val drmXwithBiasColumn = drmX cbind 1
```

Serial Correlation (Autocorrelation)

https://en.wikipedia.org/wiki/Autocorrelation

Autocorrelation, also known as **serial correlation**, is the correlation of a signal with itself at different points in time.

In OLS- Autocorrelation of the errors:

- No Longer Best Linear Unbiased Estimators (BLUE)
- Estimators (Betas) are not Biased
- Standard Errors are underestimated (t-scores over estimated)
  - E.g. Things seem significant which may not be
- Traditional Test is the Durbin-Watson Statistics

Durbin Watson Test Statistic

The Durbin Watson Test Statistic is used to test for autocorrelation among the error terms of a linear regression, a situation commonly faced in time series analysis.

$$d = \frac{\sum_{t=2}^{T}(e_t - e_{t-1})^2}{\sum_{t=1}^{T} e_t^2}$$

Durbin Watson Test Statistic

We can create our own Durbin Watson function

$$d = \frac{\sum_{t=2}^{T}(e_t - e_{t-1})^2}{\sum_{t=1}^{T} e_t^2}$$

```
def durbinWatson(error: DrmLike[Int]) = {
    //https://en.wikipedia.org/wiki/Durbin%E2%80%93Watson_statistic
    /*
    To test for positive autocorrelation at significance α, the test statistic d
    is compared to lower and upper critical values (dL,α and dU,α):
        If d < dL,α, there is statistical evidence that the error terms are positively autocorrelated.
        If d > dU,α, there is no statistical evidence that the error terms are positively autocorrelated.
        If dL,α < d < dU,α, the test is inconclusive.
    */
    val e = error(1 until error.nrow.toInt, 0 until 1)
    val e_t_1 = error(0 until error.nrow.toInt - 1, 0 until 1)
    val numerator = (e - e_t_1).assign(Functions.SQUARE).colSums
    val denominator = error.assign(Functions.SQUARE).colSums
    numerator / denominator
}
```

Cochrane-Orcutt Procedure

The Cochrane-Orcutt estimation adjusts for autocorrelation in the error term, e.g. “tries to correct the betas” for autocorrelation.

If the Durbin Watson Test Statistic indicates there is autocorrelation then the parameter estimates standard errors (t-score) are biased and must be corrected.

We seek new betas that produce a model which does not suffer from autocorrelation

In short this is achieved by:

$$\varepsilon_t = \rho\varepsilon_{t-1} + e_t$$

$$y_t - \rho y_{t-1} = \alpha(1-\rho) + \beta(X_t - \rho X_{t-1}) + e_t$$

Cochrane-Orcutt Procedure

```
def cochraneOrcutt( drmX: DrmLike[Int],
                    drmY: DrmLike[Int]) = {
  // https://en.wikipedia.org/wiki/Cochrane%E2%80%93Orcutt_estimation

  val drmX_lag1 = drmX(0 until error.nrow.toInt - 1, 0 until 1)
  val drmY_lag1 = drmY((0 until error.nrow.toInt - 1, 0 until 1))

  var beta = ols(drmX, drmY).checkpoint()
  val yFitted = (drmX %*% beta).checkpoint()
  val error = yFitted - drmY

  val rho = ols(error(1 until error.nrow.toInt, 0 until 1),
                error(0 until error.nrow.toInt - 1, 0 until 1)).get(0,0)
  //  e = rho*e_t-1

  val drmYprime = drmY(1 until error.nrow.toInt, 0 until 1) - drmY_lag1 * rho
  val drmXprime = drmX(1 until error.nrow.toInt, 0 until 1) - drmX_lag1 * rho

  var betaPrime = ols(drmXprime, drmYprime)
  val b0 = betaPrime(0,0) / (1 - rho)
  betaPrime(0,0) = b0
  betaPrime
}
```

$$\varepsilon_t = \rho\varepsilon_{t-1} + e_t$$

$$y_t - \rho y_{t-1} = \alpha(1-\rho) + \beta(X_t - \rho X_{t-1}) + e_t$$

Eigenfaces Demo

Mahout Decomposition Algorithms

The goal of the Mahout project is to enable users to ‘roll their own’ algorithms. That said, there are a few decomposition algorithms we provide to make that experience much smoother. Namely,

- Distributed Stochastic Singular Value Decomposition
- Distributed Stochastic Principal Component Analysis
- Distributed thinQR

Utilizing Scala: Why we don’t want a new scripting language

We transform images of 250x250 pixels into vectors of 62500 pixels

- E.g. Vector(0) = Pixel(0,0), Vector(1)= Pixel(0,1), ...

We now have a matrix where each row is the image of a person, each column represents a pixel position.

- Size: n x 62500 (n images in the database)

Create DRM of Vectorized Images

```
%sparkMahout.spark

import com.sksamuel.scrimage._
import com.sksamuel.scrimage.filter.GrayscaleFilter

val imagesRDD = sc.binaryFiles("/tmp/lfw-deepfunneled/*/*.jpg")
                  .map(o => (new DenseVector( Image.apply(o._2.toArray)
                                                  .filter(GrayscaleFilter)
                                                  .pixels
                                                  .map(p => p.toInt.toDouble / 10000000)),
                             o._1.split("/").last.split("_").slice(0,2).mkString(" ")) )
                   .zipWithIndex

val preDRM:DrmRdd[Int] = imagesRDD.map(o => (o._2.toInt, o._1._1))

val imagesDRM = drmWrap(rdd= preDRM).par(min = 16 * 4).checkpoint()

println(s"Dataset: ${imagesDRM.nrow} images, ${imagesDRM.ncol} pixels per image")
```

Eigenfaces Problem

We want a small set of images with which we can represent all of the other images in the database as a linear combination or composite of these images. We will call these images the ***eigenimages***.

- Much more compact to store than full database of images
- More ‘real-world’ than deep learning- adding additional faces.

(Distributed Stochastic) Singular Value Decomposition

https://en.wikipedia.org/wiki/Singular_value_decomposition

A method for decomposing a matrix into two matrices of eigen components.

Consider

- $\mathbf{M}$ our matrix of images
- We decompose $\mathbf{M}$ into two new matrices $\mathbf{V}$ and $\mathbf{U}$ where
  - $\mathbf{V}$ our matrix of eigenfaces where each column is a face and each row is a pixel position
  - $\mathbf{U}$ the linear combination of eigenfaces required to reconstruct each image in our training set where each row represents a linear combination required and each column is the corresponding eigenface

DSSVD and Eigenfaces

drmV or **V** is a (distributed) 62500 x 100 matrix where each column is a vectorized image, an eigen face. All images in the database can be represented as linear combinations of these eigenfaces
drmU or **U** is a (distributed) 361 x 100 matrix where each row contains a vector which is the linear combination required to reconstruct the image in the database.

Example:
The images of Aaron Peirsol has a corresponding vector u_0 in the **U** matrix of
{0:-0.007753677974867283,1:-0.04616733990335669,2:0.05441271912990047,3:0.04202143397506453,4:-5.067714488158395E-4,5:0.019155792619583792,6:0.029928492096054582,7:0.08127902253760039,8:0.0022055265810789412,9:-0.056219225182572136,...

Which is to say that the image of Aaron Peirsol can be represented as the following linear combination of eigenfaces in **V**:

$$AaronPeirsol.jpg = -0.008 \cdot eigenface_0 + -0.046 \cdot eigenface_1 + 0.054 \cdot eigenface_2 + 0.042 \cdot eigenface_3 + -0.001 \cdot eigenface_4 + 0.019 \cdot eigenface_5 + \ldots$$

Or in a more concise matrix notation:

$$AaronPeirsol.jpg = \mathbf{V} \cdot u_0$$

Recognizing New Folks

Deep learning is the sexy thing everyone loves to talk about, but has some major hang ups in this problem

- Slow to train
- If new faces are introduced, must retrain

Consider each image in our database is represented by a linear combination of the eigenfaces

Then each person can be represented by average of the linear combinations that make up their photos

So when a new photo comes in we:

1. Decompose it into a linear combination of our eigenfaces
2. Check against each person in the database to see who it is a closest match to
3. (Optional) if it is not close enough to anyone in our database, we recognize this as a new person and add them to the database.

Follow on questions:

- Why 20 eigenfaces?
- How do we measure the distance / decide what is ‘too far’ and create a new face?
- Error function?

Mahout: It’s up to you! *Roll-your-own algorithms*

Visualizing: Apache Zeppelin Integration

Mahout+Zeppelin integration relies on Zeppelin feature known as **ResourcePools**

In short this method consists of:

1. Do analysis in Apache Mahout
2. Sample Distributed Matrix (it’s probably to big to plot? If not sample 100%)
3. Write results to tab-separated string (e.g. a string that looks like a tsv file)
4. Put string in ResourcePool
5. Fetch string from ResourcePool in R/Python
6. Read into DataFrame as you would a tsv file
7. Leverage R/Python plotting libraries

Visualizing: Apache Zeppelin Integration

```
%flinkMahout                                                    FINISHED

val mxRnd = Matrices.symmetricUniformView(5000, 2, 1234)
val drmRand = drmParallelize(mxRnd)


val drmSin = drmRand.mapBlock() {case (keys, block) =>
  val blockB = block.like()
  for (i <- 0 until block.nrow) {
    blockB(i, 0) = block(i, 0)
    blockB(i, 1) = Math.sin((block(i, 0) * 8))
  }
  keys -> blockB
}

resourcePool.put("flinkSinDrm", drm.drmSampleToTSV(drmSin, 2))
```

```
mxRnd: org.apache.mahout.math.Matrix =
{
 0 =>   {0:0.4586377101191827,1:0.07261898163580698}
 1 =>   {0:0.48977896201757654,1:0.2695201068510176}
 2 =>   {0:0.33215452109376786,1:0.2148377346657124}
 3 =>   {0:0.4497098649240723,1:0.4331127334380502}
 4 =>   {0:-0.03782634247193647,1:-0.32353833540588983}
 5 =>   {0:0.15137106418749705,1:0.422446220403861}
 6 =>   {0:0.2714115385692545,1:-0.4495233989067956}
 7 =>   {0:0.02468155133492185,1:0.49474128114887833}
 8 =>   {0:-0.2269662536373416,1:-0.14808249195411455}
 9 =>   {0:0.050870692759856756,1:-0.4797329808849356}
... }
drmRand: org.apache.mahout.math.drm.CheckpointedDrm[Int] = org.apache.mahout.flinkbindings.drm.CheckpointedFlinkDrm@865
cef9d
drmSin: org.apache.mahout.math.drm.DrmLike[Int] = OpMapBlock(org.apache.mahout.flinkbindings.drm.CheckpointedFlinkDrm@8
65cef9d,<function1>,-1,-1,true)
```

Took 12 sec. Last updated by anonymous at November 11 2016, 11:46:12 AM.

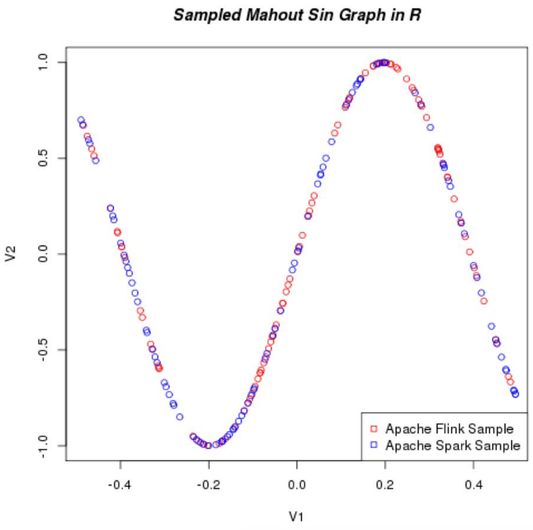
```
%sparkMahout                                                    FINISHED

val mxRnd = Matrices.symmetricUniformView(5000, 2, 1234)
val drmRand = drmParallelize(mxRnd)


val drmSin = drmRand.mapBlock() {case (keys, block) =>
  val blockB = block.like()
  for (i <- 0 until block.nrow) {
    blockB(i, 0) = block(i, 0)
    blockB(i, 1) = Math.sin((block(i, 0) * 8))
  }
  keys -> blockB
}

z.put("sparkSinDrm", org.apache.mahout.math.drm.drmSampleToTSV(drmSin, 2))
```

```
mxRnd: org.apache.mahout.math.Matrix =
{
 0 =>   {0:0.4586377101191827,1:0.07261898163580698}
 1 =>   {0:0.48977896201757654,1:0.2695201068510176}
 2 =>   {0:0.33215452109376786,1:0.2148377346657124}
 3 =>   {0:0.4497098649240723,1:0.4331127334380502}
 4 =>   {0:-0.03782634247193647,1:-0.32353833540588983}
 5 =>   {0:0.15137106418749705,1:0.422446220403861}
 6 =>   {0:0.2714115385692545,1:-0.4495233989067956}
 7 =>   {0:0.02468155133492185,1:0.49474128114887833}
 8 =>   {0:-0.2269662536373416,1:-0.14808249195411455}
 9 =>   {0:0.050870692759856756,1:-0.4797329808849356}
... }
drmRand: org.apache.mahout.math.drm.CheckpointedDrm[Int] = org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@241
5f15c
drmSin: org.apache.mahout.math.drm.DrmLike[Int] = OpMapBlock(org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@2
415f15c,<function1>,-1,-1,true)
```

Took 12 sec. Last updated by anonymous at November 11 2016, 11:45:22 AM.

Visualizing: Apache Zeppelin Integration

```
%sparkMahout.r {"imageWidth": "600px"}

install.packages("ggplot2", lib="~/R/packages", "http://cran.uk.r-project.org")
library("ggplot2", lib.loc="~/R/packages")

flinkSinStr = z.get("flinkSinDrm")
sparkSinStr = z.get("sparkSinDrm")

flinkData <- read.table(text= flinkSinStr, sep="\t", header=FALSE)
sparkData <- read.table(text= sparkSinStr, sep="\t", header=FALSE)

plot(flinkData,  col="red")
# Graph trucks with red dashed line and square points
points(sparkData, col="blue")

# Create a title with a red, bold/italic font
title(main="Sampled Mahout Sin Graph in R", col.main="black", font.main=4)

legend("bottomright", c("Apache Flink Sample", "Apache Spark Sample"), col= c("red", "blue"), pch= c(22, 22))
```



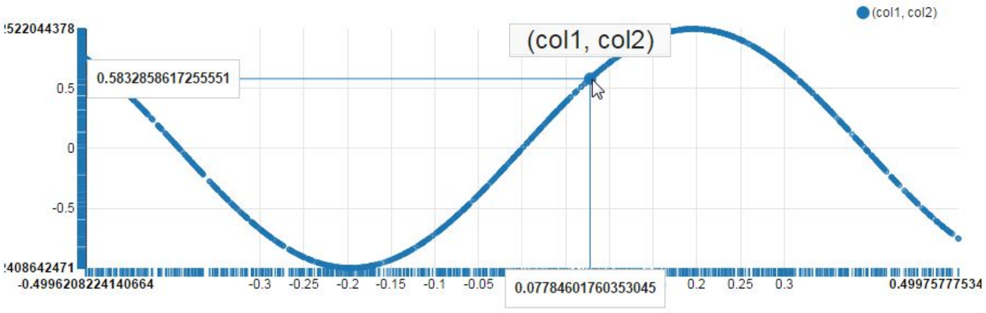
Visualizing: Apache Zeppelin Integration

Tablify Matrix Using Zepplin + Angular

FINISHED

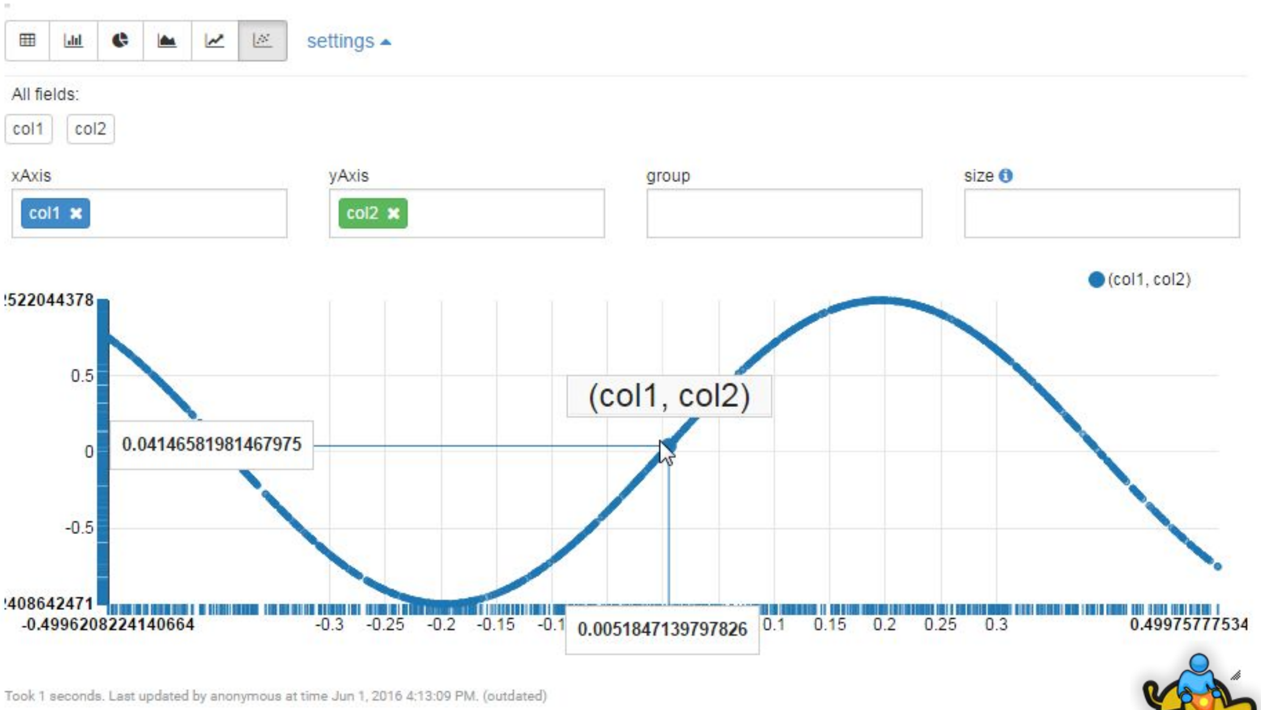
```
%spark

var str = ""
//println("matrix collected")
for (i <- 0 until mPlotMatrix.numRows()) {
    //println("i: " + i.toString)
    for (j <- 0 until mPlotMatrix.numCols()) {
        str += mPlotMatrix(i, j)
        if (j <= mPlotMatrix.numCols() - 2) { str += "\t" }
    }
    str += "\n"
  }

val tableStr = "col1\tcol2\n"+ str
println("%table\n"+tableStr)
```

settings ▾



Took 1 seconds. Last updated by anonymous at time Jun 1, 2016 4:09:07 PM. (outdated)

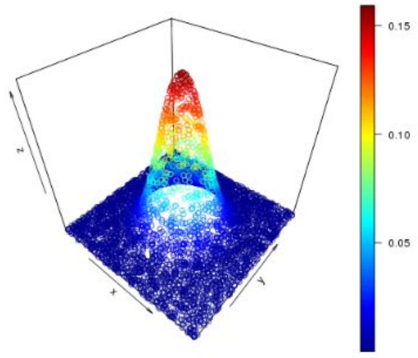
Visualizing: Apache Zeppelin Integration



Visualizing: Apache Zeppelin Integration

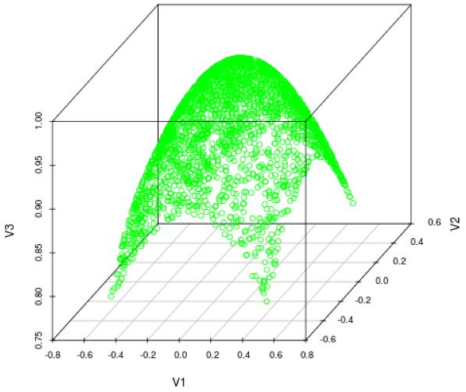
```
%r {"imageWidth": "400px"}
library("plot3D")
dfStr = z.get("mahout5000Table")
data <- read.table(text= dfStr, sep="\t", header=TRUE)
colnames(data)
points3D(data$col1, data$col2, data$col3)
```

"col1 "col2 "col3



Took 1 seconds. Last updated by anonymous at time Jun 1, 2016 3:53:14 PM. (outdated)

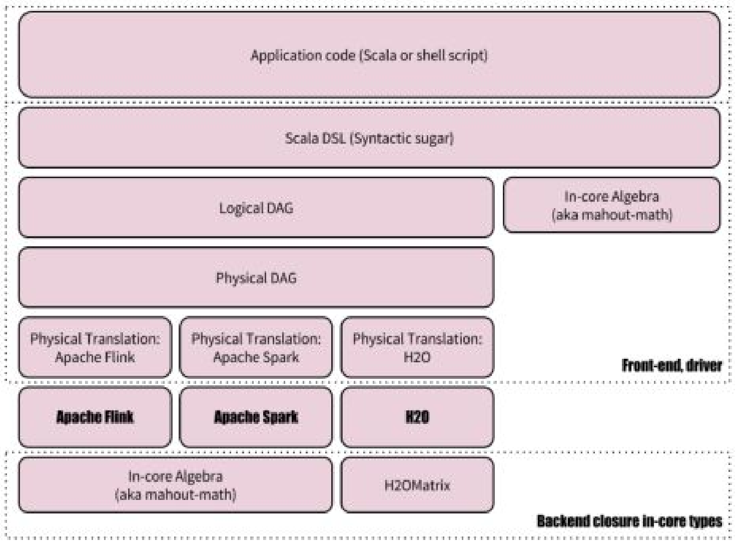
```
%spark.r {"imageWidth": "400px"}

library(scatterplot3d)


flinkGaussStr = z.get("flinkGaussDrm")
flinkData <- read.table(text= flinkGaussStr, sep="\t", header=FALSE)

scatterplot3d(flinkData, color="green")
```



Overview

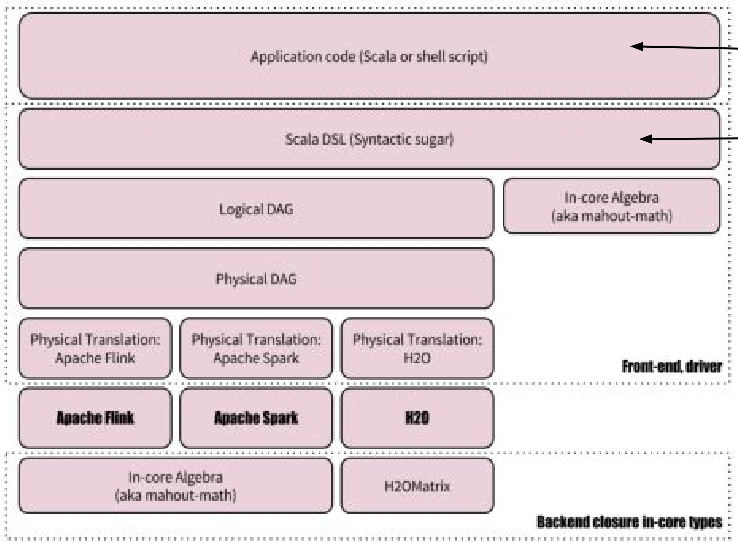
- Intro
- Getting to Know Mahout
- Rolling our own algorithms
- **How it works...**
- Future: What’s Coming …
- Conclusion and Questions

The Mahout Stack (Current)



Reproduced with permission from 'Apache Mahout: Beyond MapReduce'

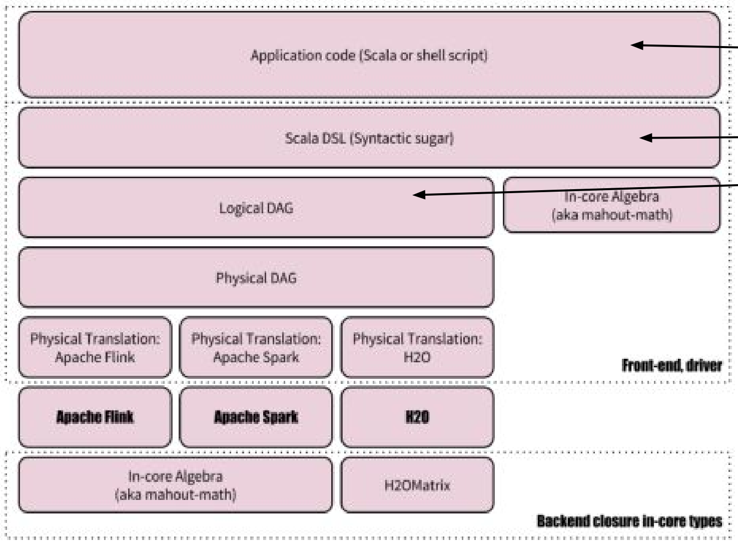
The Mahout Stack (Current)



- Regular Scala / Engine Specific Code / etc
- Mahout R-Like Code

Reproduced with permission from 'Apache Mahout: Beyond MapReduce'

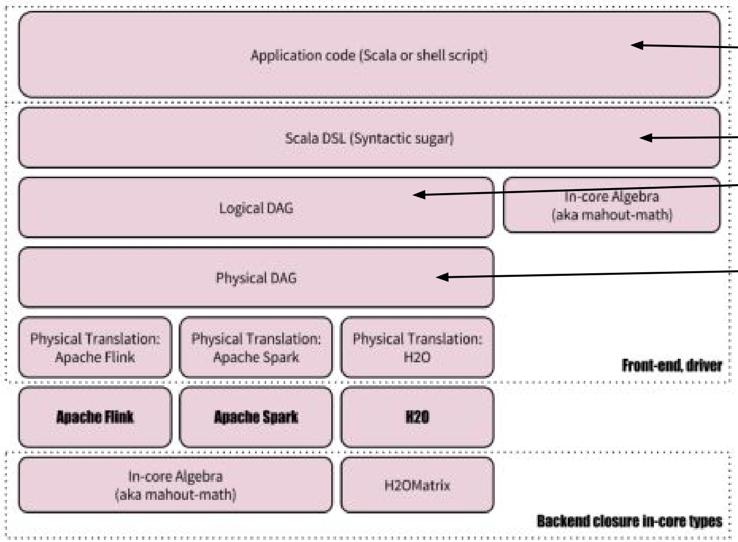
The Mahout Stack (Current)



- Regular Scala / Engine Specific Code / etc
- Mahout R-Like Code
- Convert into Abstract operations such as AtA, AtB

Reproduced with permission from 'Apache Mahout: Beyond MapReduce'

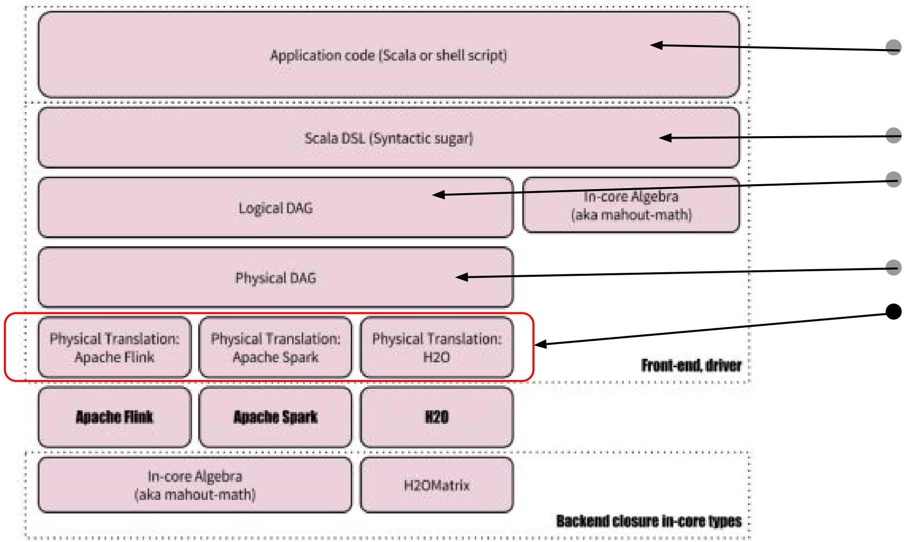
The Mahout Stack (Current)



- Regular Scala / Engine Specific Code / etc
- Mahout R-Like Code
- Convert into Abstract operations such as AtA, AtB
- Remove Redundancies (3 passes, on DAG not data)

Reproduced with permission from 'Apache Mahout: Beyond MapReduce'

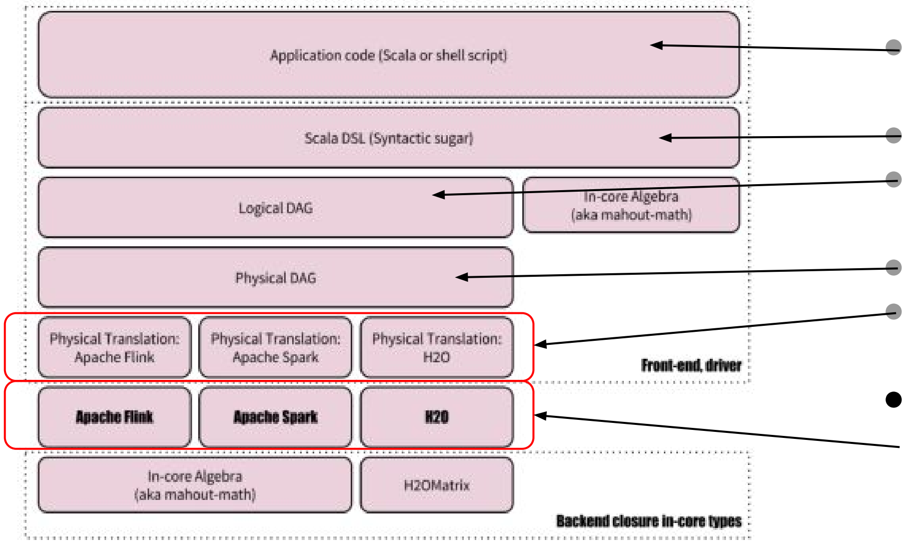
The Mahout Stack (Current)



- Regular Scala / Engine Specific Code / etc
- Mahout R-Like Code
- Convert into Abstract operations such as AtA, AtB
- Remove Redundancies
- Implement operations in engine specific distributed manner (fat vs skinny)

Reproduced with permission from 'Apache Mahout: Beyond MapReduce'

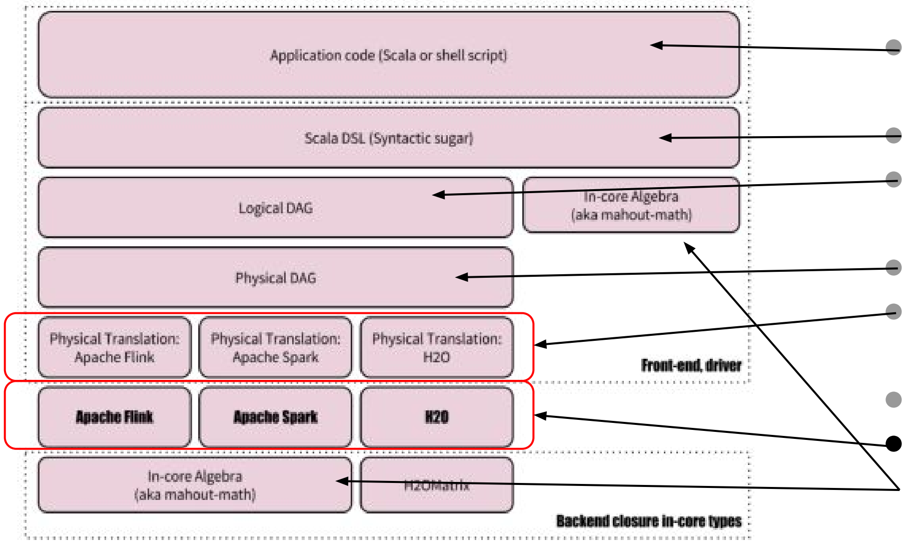
The Mahout Stack (Current)



- Regular Scala / Engine Specific Code / etc
- Mahout R-Like Code
- Convert into Abstract operations such as AtA, AtB
- Remove Redundancies
- Implement operations in engine specific distributed manner
- Distribute with Engine

Reproduced with permission from ‘Apache Mahout: Beyond MapReduce’

The Mahout Stack (Current)



- Regular Scala / Engine Specific Code / etc
- Mahout R-Like Code
- Convert into Abstract operations such as AtA, AtB
- Remove Redundancies
- Implement operations in engine specific distributed manner
- Distribute with Engine
- Mahout in Core Algebra

Reproduced with permission from 'Apache Mahout: Beyond MapReduce'

Runtime and Optimizations

- Execution is deferred, user composes logical operators

```
val drmC = drmA.t %*% drmA
```

- Computation triggers optimization and execution

```
drmI.dfsWrite(path)
val inMemV =(drmU %*% drmM).collect
```

- Optimization factors
  - Size of operands
  - Orientation of operands
  - Partitioning
  - Shared computational paths

Example with matrix multiplication:

- 5 physical operators for drmA %*% drmB
- 2 operators for drmA %*% inMemA
- 1 operator for drm A %*% x
- 1 operator for x %*% drmA

Runtime and Optimizations

- Common computational paths ((A + B)’ %*% (A + B) -> self-square(A + B)
- Tracking identically partitioned sets (“zip” vs. “join” judgements)
- Tracking data deficiencies (missing or duplicate rows)
  - automatic fixes
- Algebraic cost reducing rewrites (Expr t) t -> Expr
- Unary operator fusion dlog(X * X) -> elementwise-apply [x => log(x * x)]
- Elements of cost based optimizations (“slim” vs. “wide”)
- Product parallelism decisions
- Explicit and implicit optimization barriers
  - control the scope of optimization

Optimization Example

- Computation of $A^T A$ in example

```
val C = A.t %*% A
```

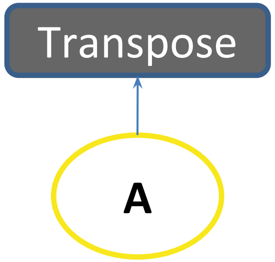
Optimization Example

- Computation of $A^T A$ in example

```
val C = A.t %*% A
```

- **Naïve execution**

1st pass: transpose A
(requires repartitioning of A)



Optimization Example

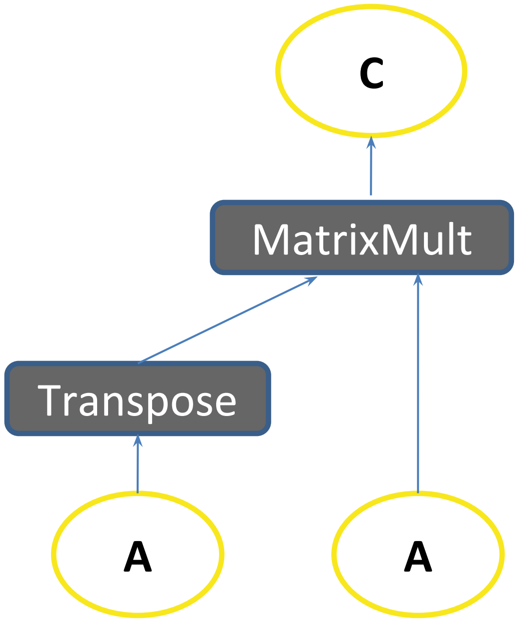
- Computation of $A^T A$ in example

```
val C = A.t %*% A
```

- Naïve execution

1st pass: transpose A
(requires repartitioning of A)

2nd pass: multiply result with A
(expensive, potentially requires repartitioning again)



Optimization Example

Computation of $A^T A$ in example

```
val C = A.t %*% A
```

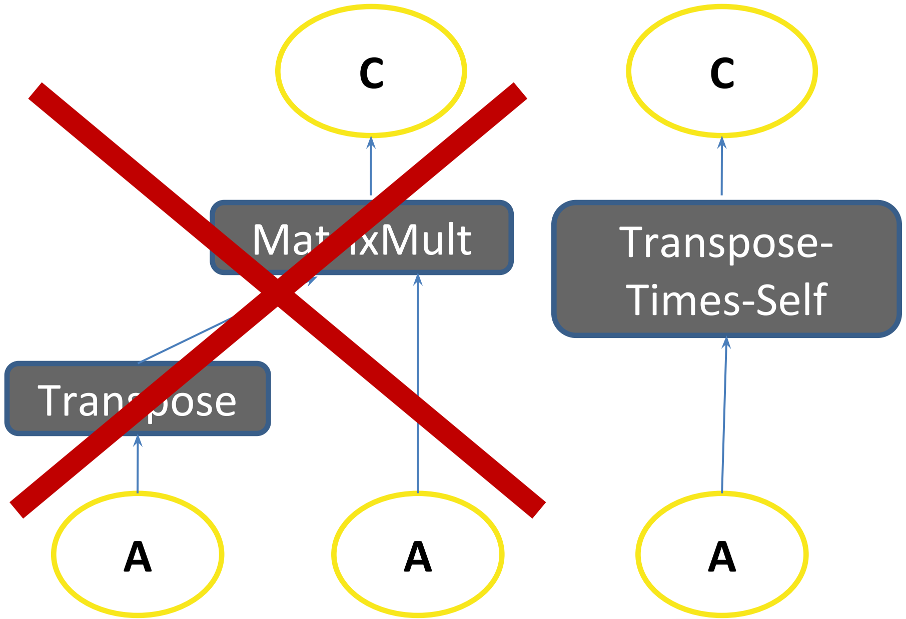
Naïve execution

~~1st pass: transpose A~~
~~(requires repartitioning of A)~~

~~2nd pass: multiply result with A~~
~~(expensive, potentially requires~~
~~repartitioning again)~~

Logical optimization

Optimizer rewrites plan to use specialized logical operator for *Transpose-Times-Self* matrix multiplication



Transpose-Times-Self

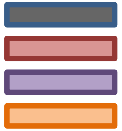
- Mahout Samsara computes $A^T A$ via **row-outer-product** formulation
  - executes in a single pass over row-partitioned A

$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$

Transpose-Times-Self

- Mahout Samsara computes A^TA via **row-outer-product** formulation
  - executes in a single pass over row-partitioned A

$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$

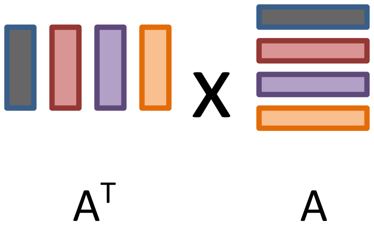


A

Transpose-Times-Self

- Mahout Samsara computes A^TA via **row-outer-product** formulation
  - executes in a single pass over row-partitioned A

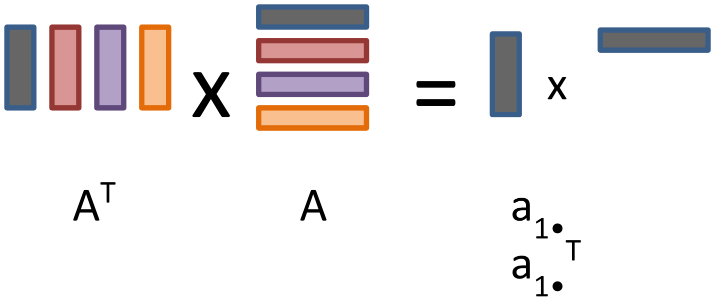
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$



Transpose-Times-Self

- Mahout Samsara computes A^TA via **row-outer-product** formulation
  - executes in a single pass over row-partitioned A

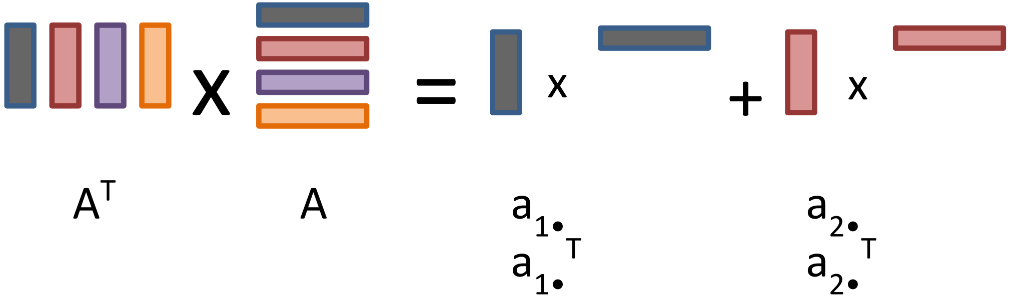
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$



Transpose-Times-Self

- Mahout Samsara computes A^TA via **row-outer-product** formulation
  - –executes in a single pass over row-partitioned A

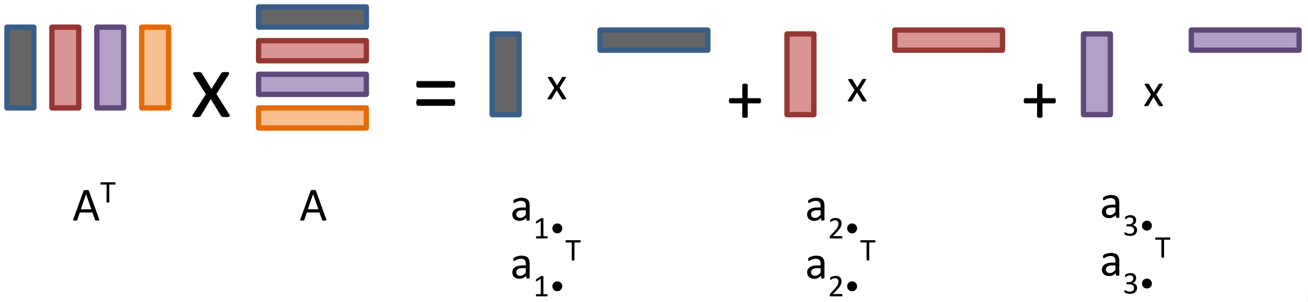
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$



Transpose-Times-Self

- Mahout Samsara computes A^TA via **row-outer-product** formulation
  - executes in a single pass over row-partitioned A

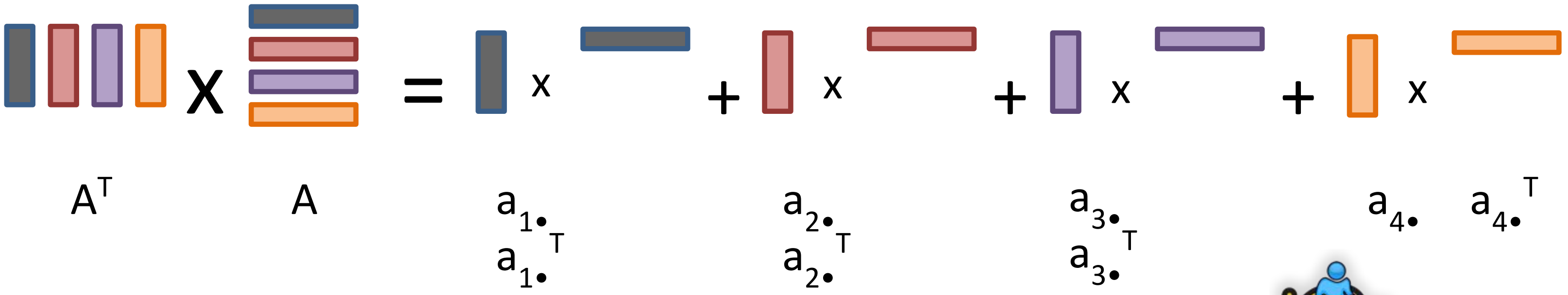
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^{\ T}$$



Transpose-Times-Self

- Mahout Samsara computes A^TA via **row-outer-product** formulation
  - executes in a single pass over row-partitioned A

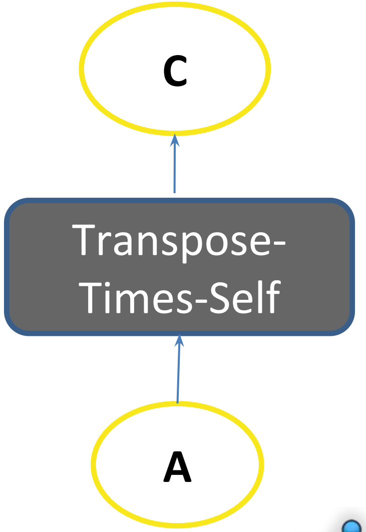
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$



Physical operators for the distributed computation of $A^T A$

Physical Operators for Transpose-Times-Self

- Two physical operators (concrete implementations) available for *Transpose-Times-Self* operation
  - standard operator ***AtA***
  - operator ***AtA_slim***, specialized implementation for tall & skinny matrices

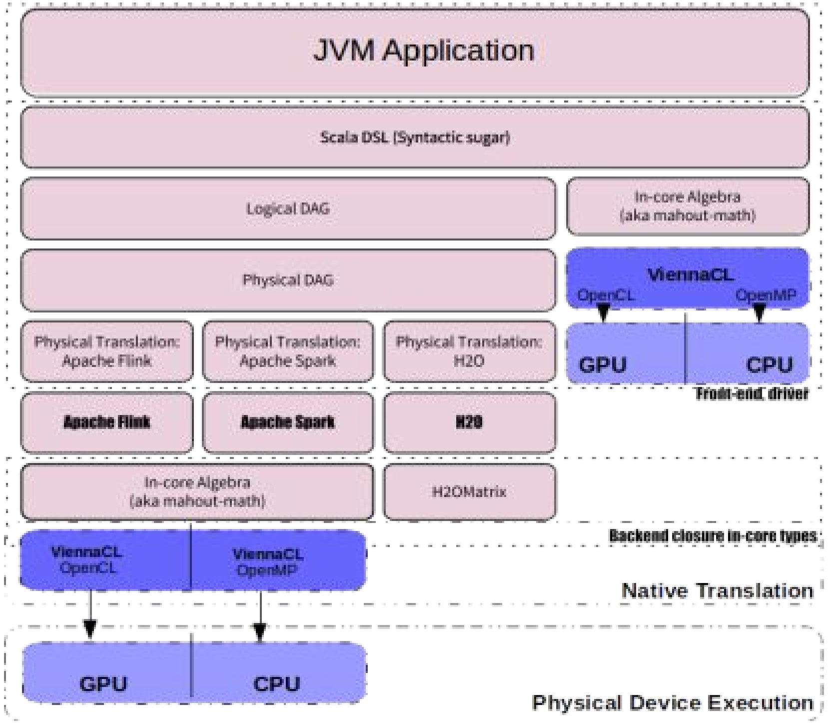


- Optimizer must choose
  - currently: depends on user-defined threshold for number of columns
  - ideally: cost based decision, dependent on estimates of intermediate result sizes

Overview

- Intro
- Getting to Know Mahout
- Rolling our own algorithms
- How it works...
- **Future: What's Coming …**
    - Soon: Native, JVM -> (GPU/CPU)
    - More prebuilt algos/pipeline ready
- Conclusion and Questions

The Mahout Stack + Native (Future)

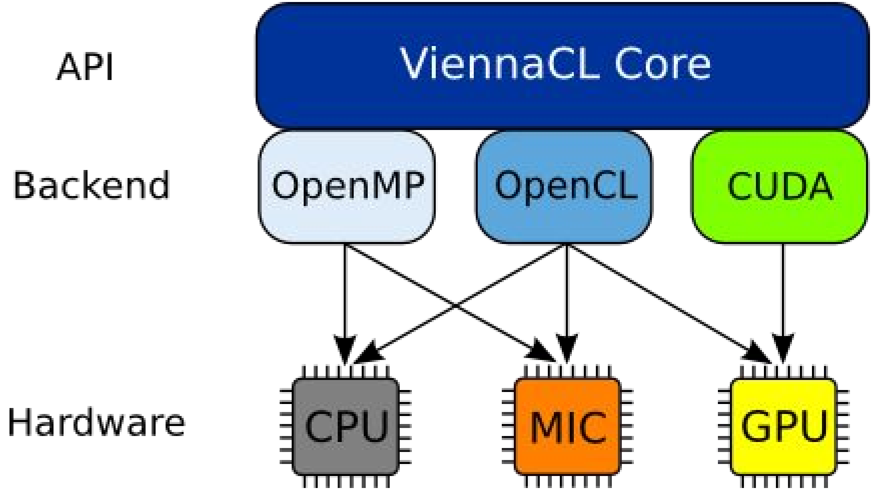


Current work: A plugin to break mahout out of the restrictions of the JVM by implementing native operations for in-core matrices using ViennaCL. This Provides GPU and multi-threaded native CPU Acceleration for DRM operations on the back end of a cluster.

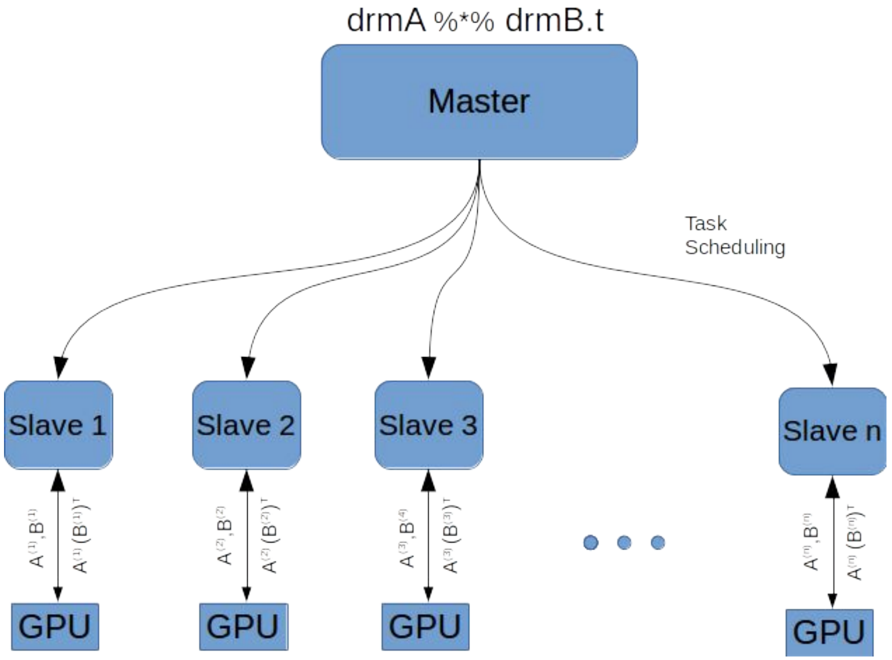
Native OpenMP and GPU Acceleration will not require any change in syntax.

mahout

Why ViennaCL



GPU Compute



With GPU Integration, Mahout syntax will not change at all.

Consistent API and New Algorithms

Creating a Framework for new Algorithms
https://github.com/apache/mahout/pull/246

R and Mahout in days of old: “Loose bag of algorithms”

Consistent API

- Approach of sklearn, adopted by SparkML
- Makes ‘Pipelines’ easier.

Mahout approach somewhere in the middle

Consistent API and New Algorithms

Several Types of Models

- Regressor: Many inputs -> one continuous output
  - OLS
  - GLM
- Transformer: Many inputs -> many outputs
  - Feature Normalizer
  - Neural Networks
- Classifier: Many inputs -> choose a classification
  - K-Means

Consistent API and New Algorithms

Goal: Consistent interface

Methods such as ‘.fit’ ‘.predict’ and ‘.transform’ where possible

Similar to sklearn interface

Parameter space that can be searched similar to sklearn `gridCVSearch`

Consistent API and New Algorithms

Many ‘prototype’ models part of pull request

Transformers:

- Mean center (used in Eigenfaces Demo)
- Feature Normalizer

Regressors:

- Ordinary Least Squares

We’re still trying to figure out where fitness tests, ALS, and procedures like Cochrane-Orcutt belong, Our Algorithm API is still [WIP].

Hoping to integrate with Spark Pipelines

- Mahout algos as part of SparkML pipeline and vice versa

Overview

- Intro
- Getting to Know Mahout
- Rolling our own algorithms
- How it works ...
- Future: What’s Coming …
- **Conclusion and Questions**

What's Next for Apache Mahout and You?

1. Ask us some questions now.
2. Go home and start using Mahout:
   a. Include in your compiled jars OR
   b. Download Zeppelin and play in notebook
3. Join the mailing lists
4. Get the book
5. Contribute!
   a. All Volunteers
   b. Seattle

Credits

Anand Avati	- Research Student, Stanford University
Andrew Musselman	- Committer
Andrew Palumbo	- Committer
Dmitriy Lyubimov	- Committer
Nathan Halko	- Special Thanks
Pat Ferrel	- Committer
Sebastian Schelter	- Committer
Shannon Quinn	- Committer
Suneel Marthi	- Committer
Trevor Grant	- Committer

Pointers

Apache Mahout documentation on Samsara

- http://mahout.apache.org/users/environment/in-core-reference.html
- https://mahout.apache.org/users/environment/out-of-core-reference.html

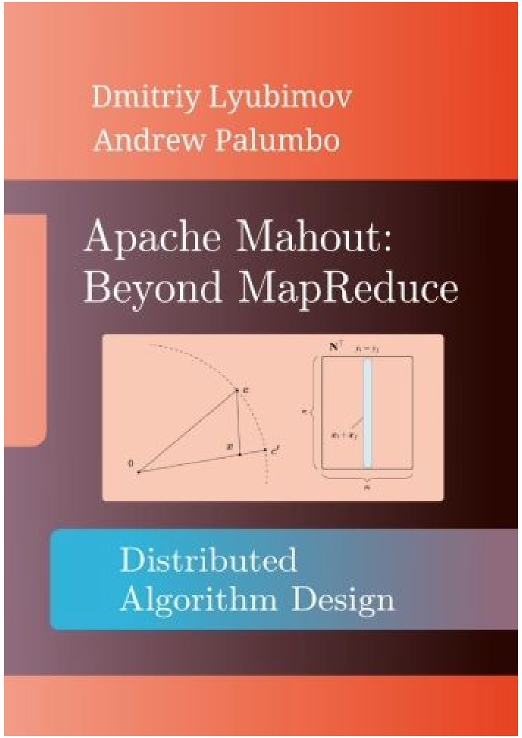
Mahout Committer, Dmitriy Lyubimov's Blog

- http://www.weatheringthroughtechdays.com/2015/04/mahout-010x-first-mahout-release-as.html

Mahout Commiter, Trevor Grant's Blogs

- https://trevorgrant.org/2016/05/19/visualizing-apache-mahout-in-r-via-apache-zeppelin-incubating/
- https://trevorgrant.org/2016/10/13/deep-magic-volume2-absurdly-large-ols-with-apache-mahout/
- https://trevorgrant.org/2016/11/10/deep-magic-volume-3-eigenfaces/

Apache Mahout: Beyond Map Reduce



https://www.amazon.com/Apache-Mahout-MapReduce-Dmitriy-Lyubimov/dp/1523775785

Bonus Material

“But I don’t have a 5-node YARN cluster to play with…”

Yes you do.

https://trevorgrant.org/2016/10/13/deep-magic-volume2-absurdly-large-ols-with-apache-mahout/

Free Trial: www.bluemix.net

Convenience Scripts for Deploying Zeppelin+Mahout (and others)
https://github.com/rawkintrevo/bluemix-extra-services

Contact Us!

dev@mahout.apache.org

user@mahout.apache.org

https://mahout.apache.org/general/mailing-lists,-irc-and-archives.html

Twitter: @ApacheMahout