Apache Mahout!?

What's Next?!

Overview

- **Intro**
- Getting to Know Mahout
- Rolling our own distributed algorithms
- Future: What's Coming …
- Conclusion and Questions

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

Overview

- Intro
- **Getting to Know Mahout**
  - What is Mahout?
  - What is Mahout Samsara?
  - Mahout Samsara DSL basics
- Rolling our own algorithms
- 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

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

An ‘Ideal’ Language for Distributed Math/DS

- Mathematically Expressive
- Based in another language
- Backend independent
- Algebraic Optimizer

Mahout-Samsara

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)
```

System Agnostic R-Like DSL

```
%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)
```

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
- Solving linear systems
- In-memory decompositions

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

val x = solve(A, b)

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

Features (3)

- Distributed decompositions
- Caching of DRMs

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

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**
  - Mahout Decomposition Algorithms
  - OLS and Time Series Example
  - Zeppelin Visualization
  - Eigenfaces Demo
- Future: What's Coming …
- Conclusion and Questions

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

Ordinary Least Squares

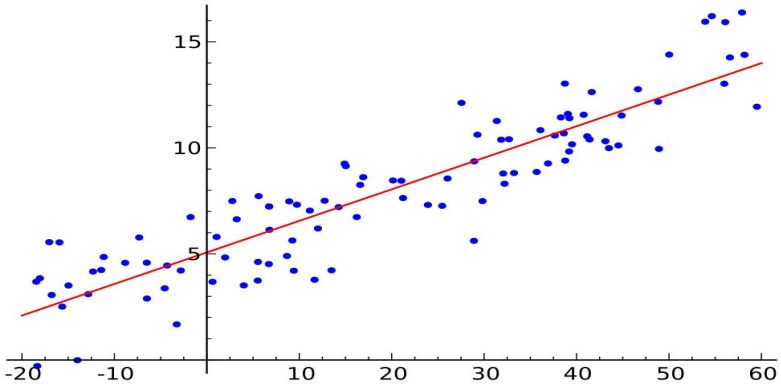
Linear Regression

- 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



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 *ß* 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 $X^T X$ and $X^T y$ onto the driver and use an in-core solver
  - assumes $X^T X$ 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 $X^T X$ and $X^T y$ onto the driver and use an in-memory solver
  - assumes $X^T X$ 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

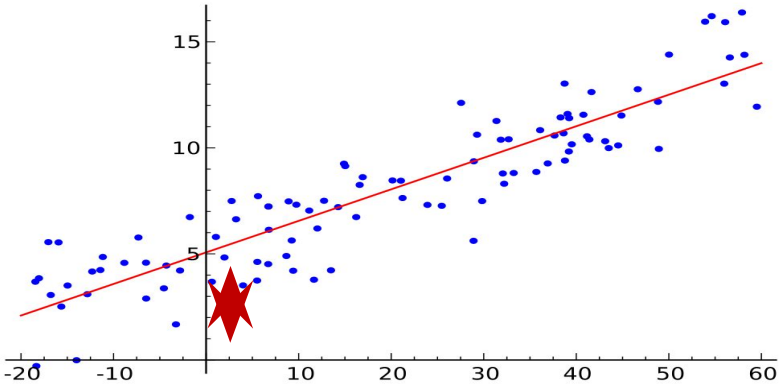
$$\hat{y} = X\,\hat{\beta}$$

- 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
   }
```

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 (betas) 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
}
```

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

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

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. Write results to tab-separated string (e.g. a string that looks like a tsv file)
3. Put string in ResourcePool
4. Fetch string from ResourcePool in R/Python
5. Read into DataFrame as you would a tsv file
6. Leverage R/Python plotting libraries

Visualizing: Apache Zeppelin Integration

```
%flinkMahout

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))
```

FINISHED

```
mxRnd: org.apache.mahout.math.Matrix =
{
 0 =>   {0:0.4586377101191827,1:0.072618981635806 98}
 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 =>   {0:0.02468155133492185,1:0.4947412811488783 3}
 8 =>   {0:-0.2269662536373416,1:-0.14808249195411455}
 9 =>   {0:0.0508706927598567 56,1:-0.4797329808849356}
... }
drmRand: org.apache.mahout.math.drm.CheckpointedDrm[Int] = org.apache.mahout.flinkbindings.drm.CheckpointedFlinkDrm@865cef9d
drmSin: org.apache.mahout.math.drm.DrmLike[Int] = OpMapBlock(org.apache.mahout.flinkbindings.drm.CheckpointedFlinkDrm@865cef9d,<function1>,-1,-1,true)
```

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

```
%sparkMahout

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))
```

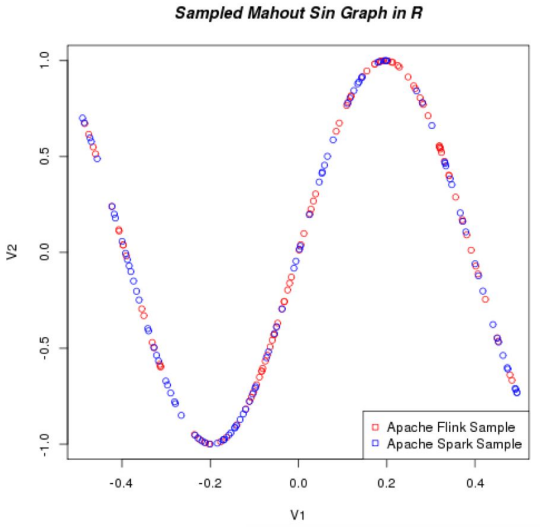
FINISHED

```
mxRnd: org.apache.mahout.math.Matrix =
{
 0 =>   {0:0.4586377101191827,1:0.072618981635806 98}
 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 =>   {0:0.02468155133492185,1:0.4947412811488783 3}
 8 =>   {0:-0.2269662536373416,1:-0.14808249195411455}
 9 =>   {0:0.0508706927598567 56,1:-0.4797329808849356}
... }
drmRand: org.apache.mahout.math.drm.CheckpointedDrm[Int] = org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@2415f15c
drmSin: org.apache.mahout.math.drm.DrmLike[Int] = OpMapBlock(org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@2415f15c,<function1>,-1,-1,true)
```

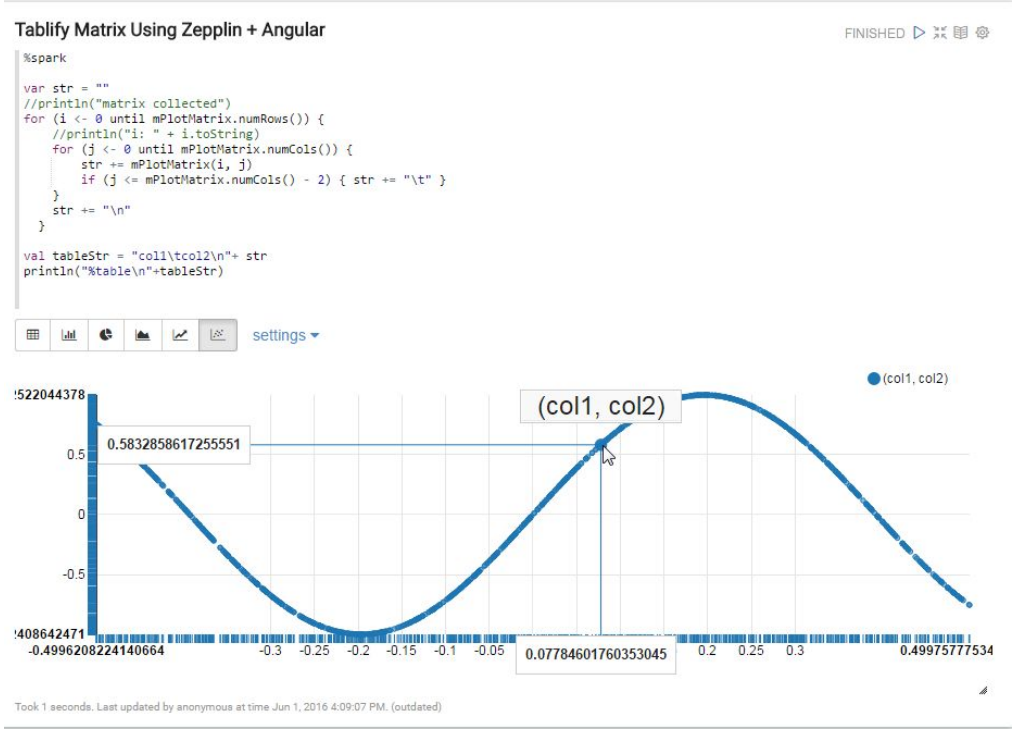
Took 12 sec. Last updated by anonymous at November 11 2016, 11:45:22 AM.

Visualizing: Apache Zeppelin Integration

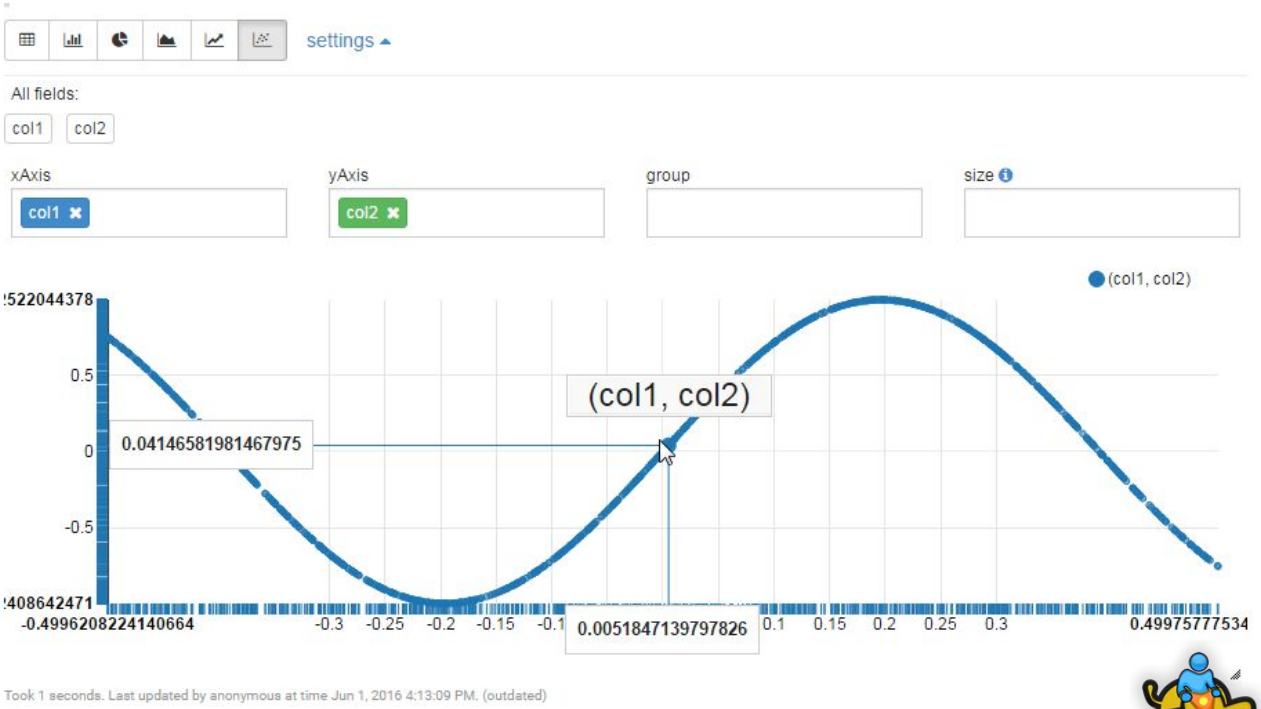
```
%sparkMahout.r {"imageWidth": "600px"}

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



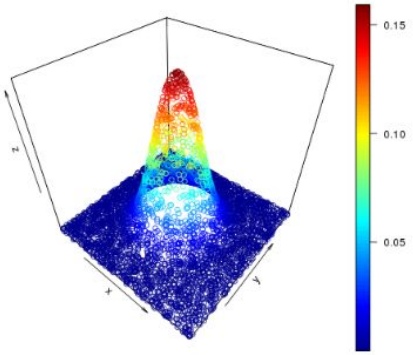
Visualizing: Apache Zeppelin Integration



Visualizing: Apache Zeppelin Integration

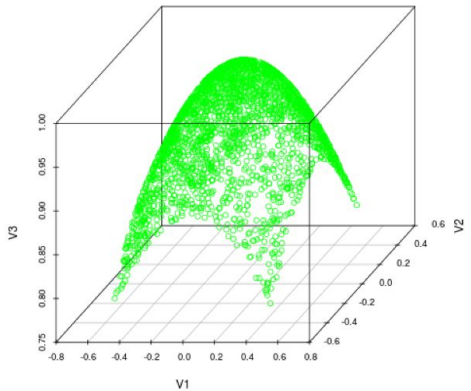
```
%r {"imageWidth": "400px"}
library("plot3D")
dfStr = z.get("mahout5000Table")
data <- read.table(text= dfStr, sep="\t", header=TRUE)
colnames(data)
points3D(data$col1, data$col2, data$col3)
```

"col1 "col2 "col3

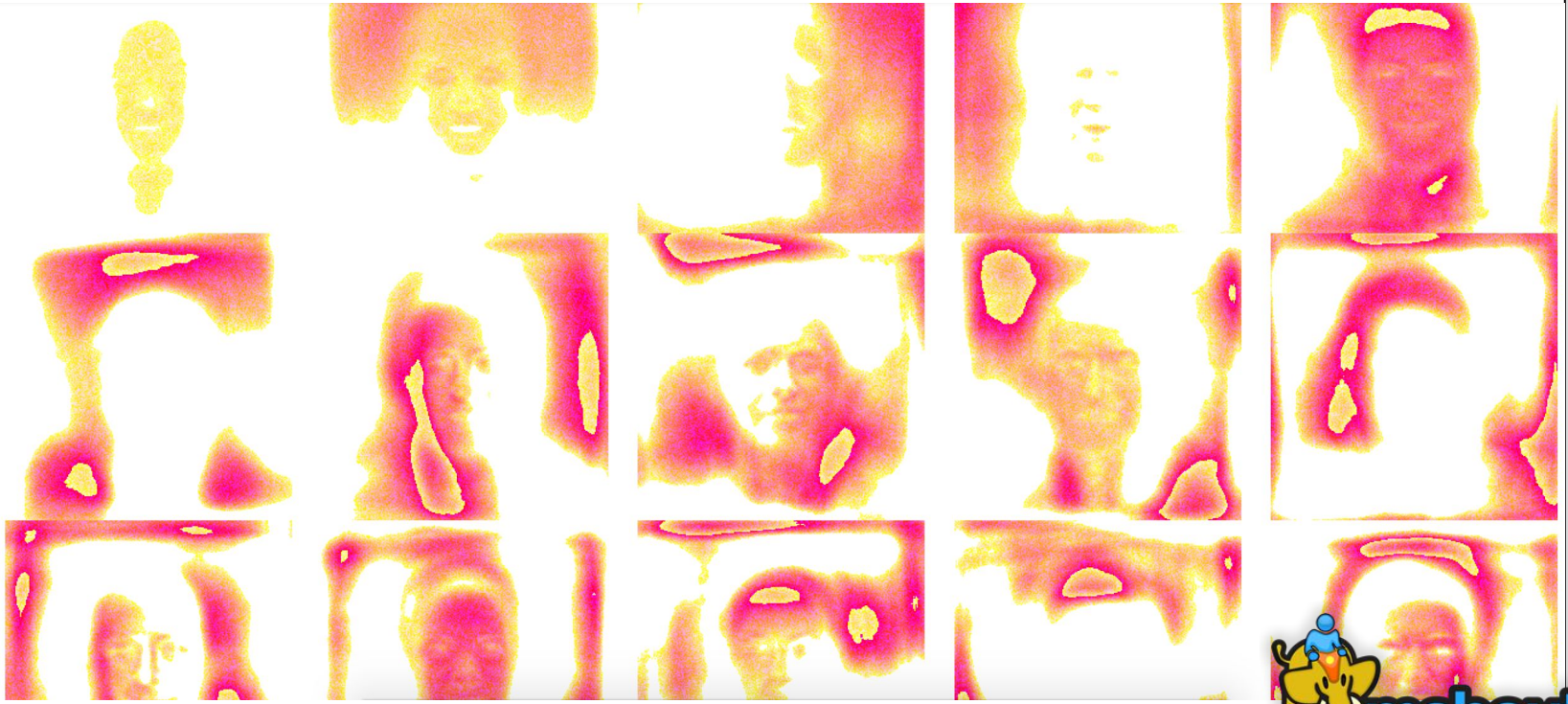


Took 1 seconds. Last updated by anonymous at time Jun 1, 2016 3:53:14 PM. (outdated)

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

library(scatterplot3d)


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

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



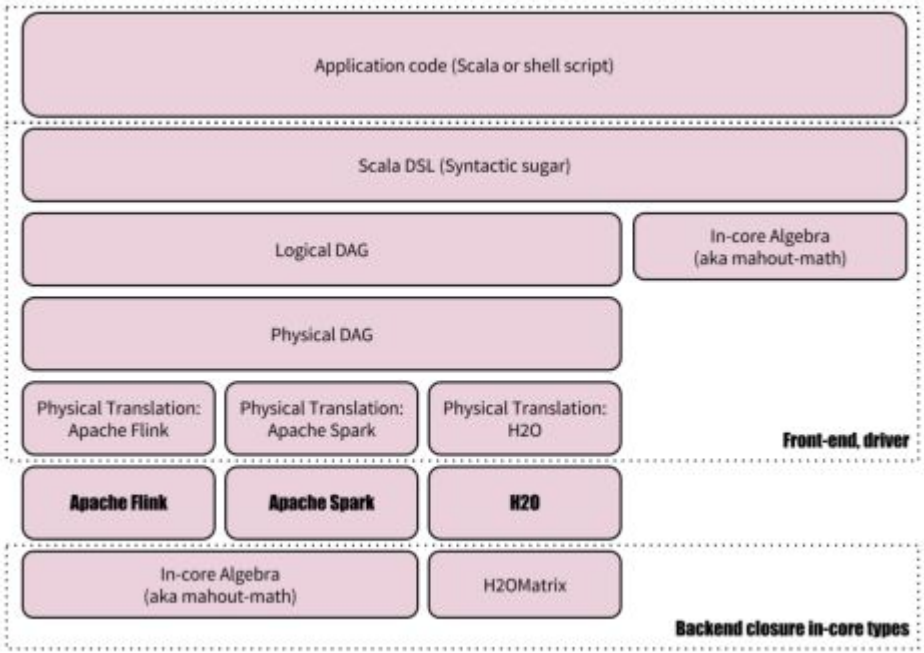
Eigenfaces Demo



Overview

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

The Mahout Stack (Current)



Reproduced with permission from 'Apache Mahout: Beyond MapReduce'

mahout

Runtime & Optimization

- Execution is deferred, user composes logical operators

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

- Computational actions implicitly trigger optimization (= selection of physical plan) and execution

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

- Optimization factors: size of operands, orientation of operands, partitioning, sharing of computational paths

- e. g.: 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 & Optimization (contd.)

- 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^TA in example

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

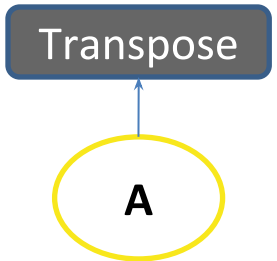
Optimization Example

- Computation of A^TA in example

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

- **Naïve execution**

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



Optimization Example

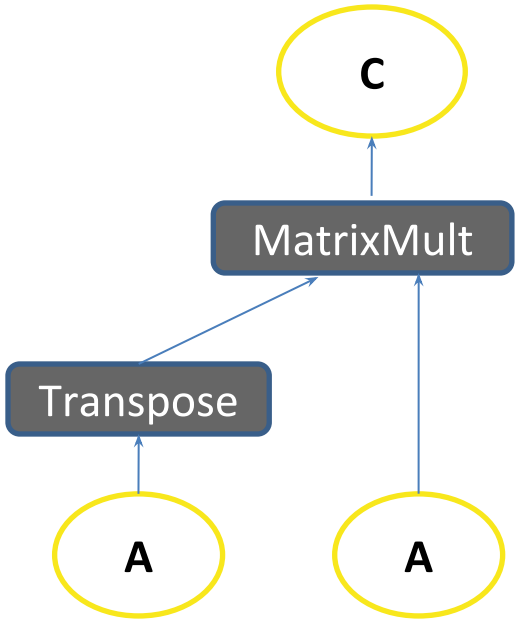
- Computation of A^TA in example

```
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^TA in example

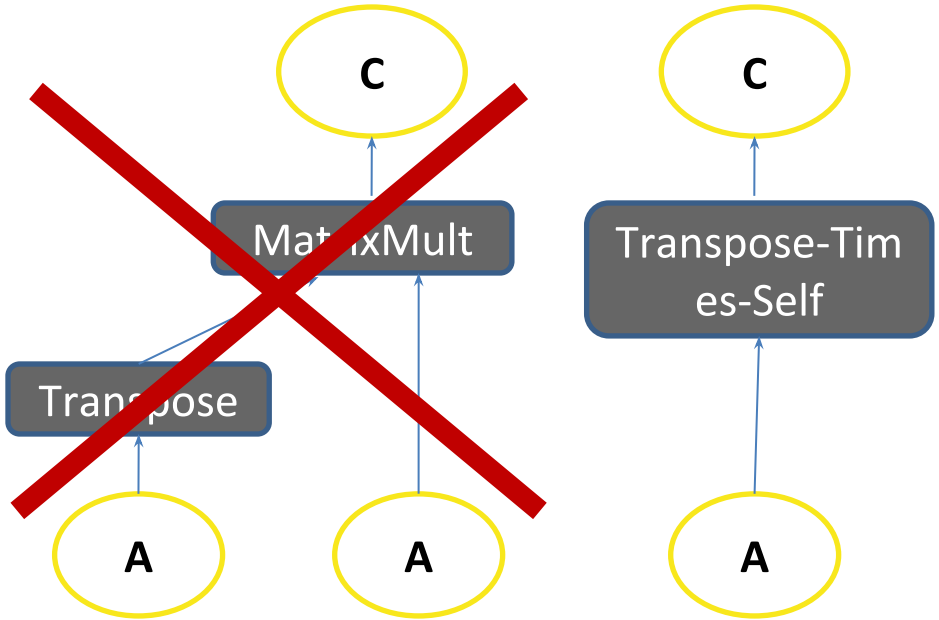
```
val C = A.t %*% A
```

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

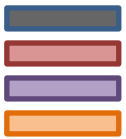
- Mahout Samsara computes $A^T A$ via **row-outer-product** formulation
  - executes in a single pass over row-partitioned A

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

Transpose-Times-Self

- 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

- 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^T A

Transpose-Times-Self

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

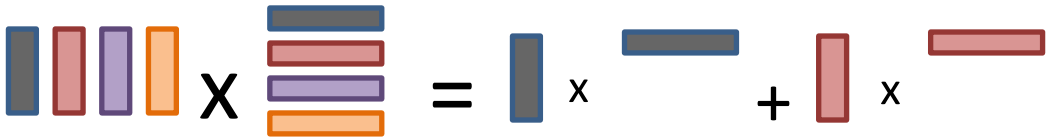
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$

A^T A $a_{1\bullet}$ $a_{1\bullet}^T$

Transpose-Times-Self

- 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^T A $a_{1\bullet}$ $a_{1\bullet}^T$ $a_{2\bullet}$ $a_{2\bullet}^T$

Transpose-Times-Self

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

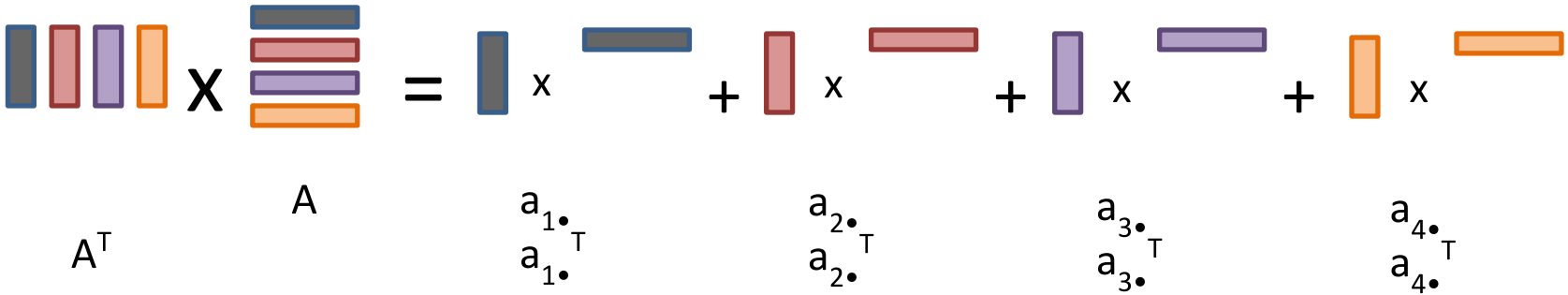
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$

A^T A $a_{1\bullet}$ $a_{1\bullet}^T$ $a_{2\bullet}$ $a_{2\bullet}^T$ $a_{3\bullet}$ $a_{3\bullet}^T$

Transpose-Times-Self

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

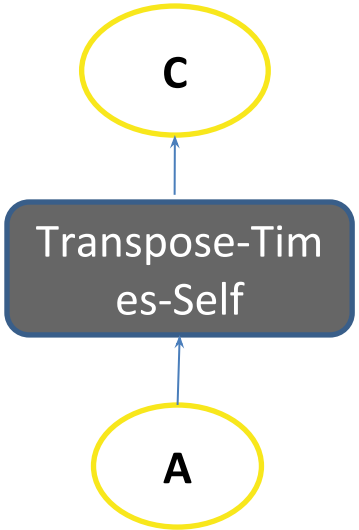
$$A^T A = \sum_{i=0}^{m} a_{i\bullet} a_{i\bullet}^T$$



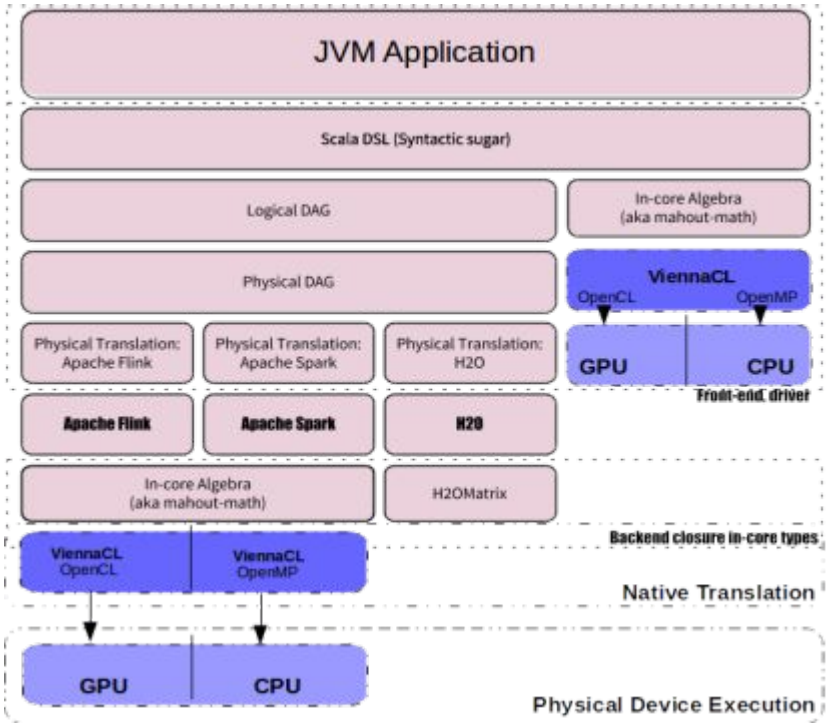
Physical operators for the distributed computation of A^TA

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

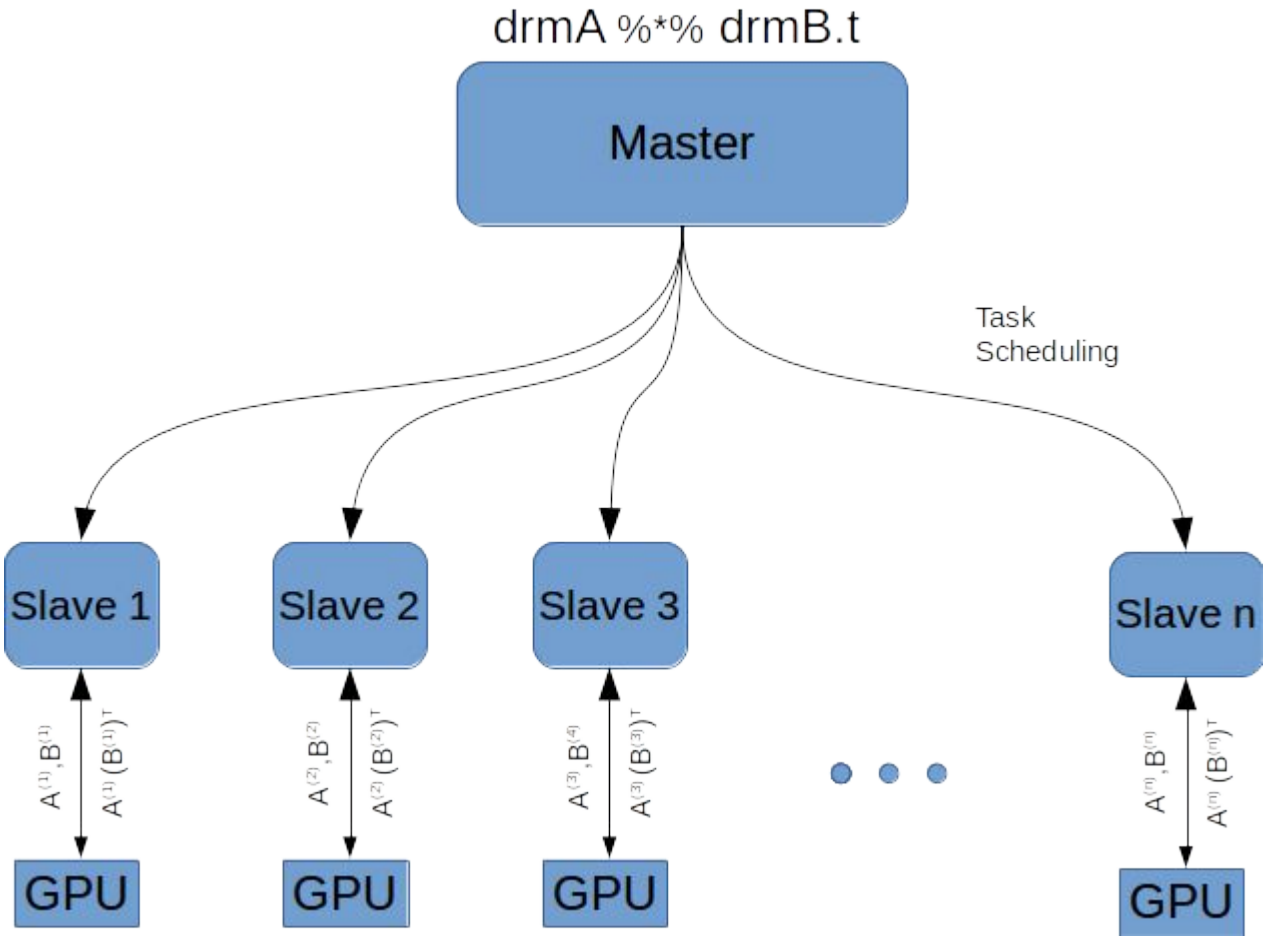


The Mahout Stack + Native (Future)



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.



With GPU Integration, the 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 ‘ol: “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
- Pipelines: A series of models
- Ensembles: Output of several models combined into one

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
- Future: What's Coming …
- **Conclusion and Questions**

What's Next for Apache Mahout and You?

1. Ask me 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!

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
Trevor Grant - Committer

Pointers

Apache Mahout has extensive documentation on Samsara

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

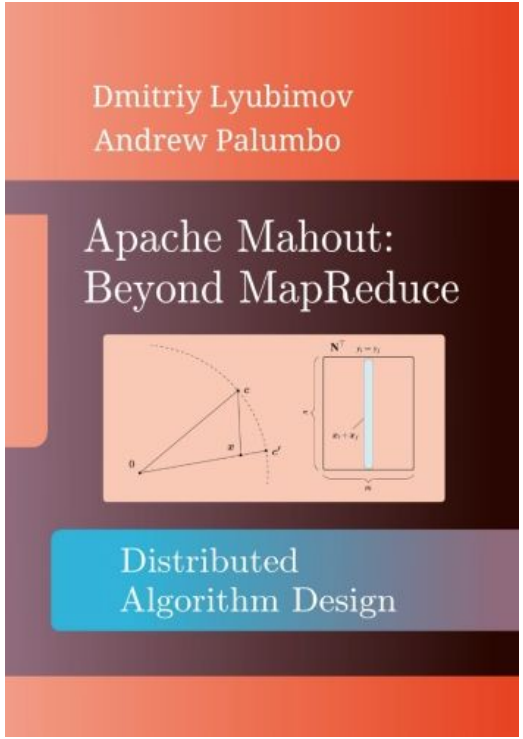
Mahout Committer, Dmitriy Lyubimov's Blog

- 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



mahout

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

Twitter: @ApacheMahout