Apache Mahout's new DSL for Distributed Machine Learning

Sebastian Schelter

GOTO Berlin

11/06/2014

Overview

- Apache Mahout: Past & Future
- A DSL for Machine Learning
- Example
- Under the covers
- Distributed computation of X^TX

Overview

- ***Apache Mahout: Past & Future***
- A DSL for Machine Learning
- Example
- Under the covers
- Distributed computation of $X^T X$

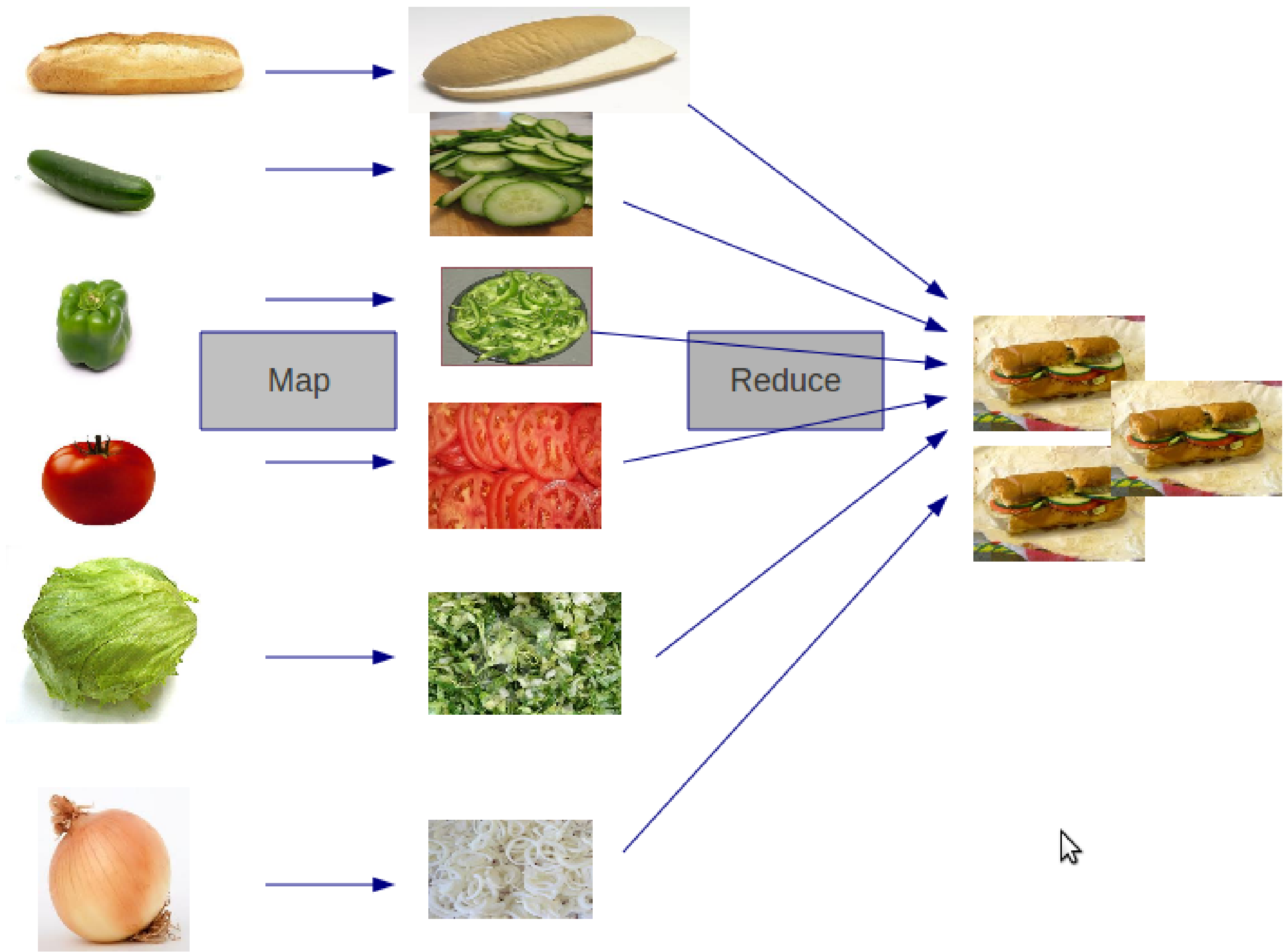
Apache Mahout: History

- library for **scalable machine learning (ML)**
- **started** six years ago **as ML on MapReduce**
- focus on popular ML problems and algorithms
  - **Collaborative Filtering**
    „find interesting items for users based on past behavior"
  - **Classification**
    „learn to categorize objects"
  - **Clustering**
    „find groups of similar objects"
  - **Dimensionality Reduction**
    „find a low-dimensional representation of the data"
- **large userbase** (e.g. Adobe, AOL, Accenture, Foursquare, Mendeley, Researchgate, Twitter)

Background: MapReduce

- simple **paradigm for distributed processing** (proposed by Google)
- user implements two functions **map** and **reduce**
- system executes program in parallel,
  scales to clusters with thousands of machines
- popular open source implementation:
  Apache Hadoop

Background: MapReduce



Apache Mahout: Problems

- **MapReduce not well suited for ML**
  - slow execution, especially for iterations
  - constrained programming model makes code hard to write, read and adjust
  - lack of declarativity
  - lots of handcoded joins necessary
- **→ Abandonment of MapReduce**
  - will reject new MapReduce implementations
  - widely used „legacy“ implementations will be maintained
- **→ „Reboot“ with a new DSL**

Overview

- Apache Mahout: Past & Future
- ***A DSL for Machine Learning***
- Example
- Under the covers
- Distributed computation of X^TX

Requirements for an ideal ML environment

1. **R/Matlab-like semantics**
   - type system that covers **linear algebra** and **statistics**

2. **Modern programming language qualities**
   - functional programming
   - object oriented programming
   - scriptable and interactive

3. **Scalability**
   - automatic distribution and parallelization with sensible performance

Requirements for an ideal ML environment

1. **R/Matlab-like semantics**
   - type system that covers **linear algebra** and **statistics**

2. **Modern programming language qualities**
   - functional programming
   - object oriented programming
   - scriptable and interactive

3. **Scalability**
   - automatic distribution and parallelization with sensible performance

Requirements for an ideal ML environment

1. **R/Matlab-like semantics**
   – type system that covers **linear algebra** and **statistics**

2. **Modern programming language qualities**
   – functional programming
   – object oriented programming
   – scriptable and interactive

3. **Scalability**
   – automatic distribution and parallelization with sensible performance

Scala DSL

- **Scala** as programming/scripting environment

- **R-like DSL** :

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

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

- **Declarativity!**
- **Algebraic expression optimizer** for distributed linear algebra
  - provides a translation layer to distributed engines
  - currently supports Apache Spark only
  - might support Apache Flink in the future

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))
```

- 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 = drmFromHDFS(...)
```

Features (1)

- Matrix, vector, scalar operators: in-memory, out-of-core

```
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
```

- Summaries

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

Features (2)

- 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)
```

- out-of-core 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()
```

Overview

- Apache Mahout: Past & Future
- A DSL for Machine Learning
- ***Example***
- Under the covers
- Distributed computation of X^TX

Cereals

Name	*protein*	*fat*	*carbo*	*sugars*	*rating*
Apple Cinnamon Cheerios	2	2	10.5	10	29.509541
Cap'n'Crunch	1	2	12	12	18.042851
Cocoa Puffs	1	1	12	13	22.736446
Froot Loops	2	1	11	13	32.207582
Honey Graham Ohs	1	2	12	11	21.871292
Wheaties Honey Gold	2	1	16	8	36.187559
Cheerios	6	2	17	1	50.764999
Clusters	3	2	13	7	40.400208
Great Grains Pecan	3	3	13	4	45.811716

http://lib.stat.cmu.edu/DASL/Datafiles/Cereals.html

Linear Regression

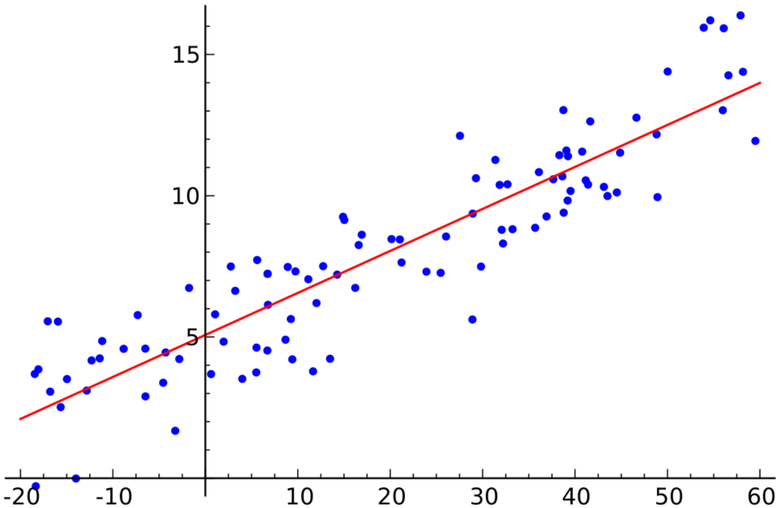
- Assumption: **target variable y** generated by linear combination of **feature matrix X** with **parameter vector β**, plus **noise ε**

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

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

- Cereals example

  X = weights of **ingredients**
  y = **customer rating**



Data Ingestion

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

```
val drmData = drmFromHdfs(...)
```

- ‚Mimick‘ a large 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
    (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)),   // 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)
```

$$
\begin{array}{c}
\begin{array}{cc} \textbf{drmX} & \qquad\qquad\qquad \textbf{y} \end{array} \\
\left(\begin{array}{cccc|c}
2 & 2 & 10.5 & 10 & 29.509541 \\
1 & 2 & 12 & 12 & 18.042851 \\
1 & 1 & 12 & 13 & 22.736446 \\
2 & 1 & 11 & 13 & 32.207582 \\
1 & 2 & 12 & 11 & 21.871292 \\
2 & 1 & 16 & 8 & 36.187559 \\
6 & 2 & 17 & 1 & 50.764999 \\
3 & 2 & 13 & 7 & 40.400208 \\
3 & 3 & 13 & 4 & 45.811716
\end{array}\right)
\end{array}
$$

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 typing the formulas:

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

val drmXty = drmX %*% 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
  - uses analogon 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
  - uses analogon 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!
(would need to add a bias term in a real implementation)

Overview

- Apache Mahout: Past & Future
- A DSL for Machine Learning
- Example
- ***Under the covers***
- Distributed computation of X^TX

Underlying systems

- currently: prototype on **Apache Spark**
  - fast and expressive cluster computing system
  - general computation graphs, in-memory primitives, rich API, interactive shell
- future: add **Apache Flink**
  - database-inspired distributed processing engine
  - emerged from research by TU Berlin, HU Berlin, HPI
  - functionality similar to Apache Spark, adds data flow optimization and efficient out-of-core execution

Runtime & Optimization

- Execution is defered, user composes logical operators

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

I.writeDrm(path);
```

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

```
val inMemV =
  (U %*% M).collect
```

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

Optimization Example

- Computation of $X^T X$ in example

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

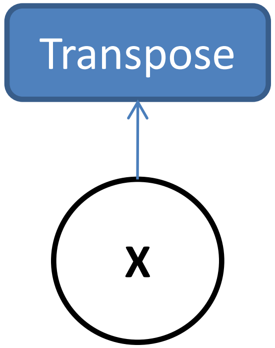
Optimization Example

- Computation of X^TX in example

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

- **Naïve execution**

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



Optimization Example

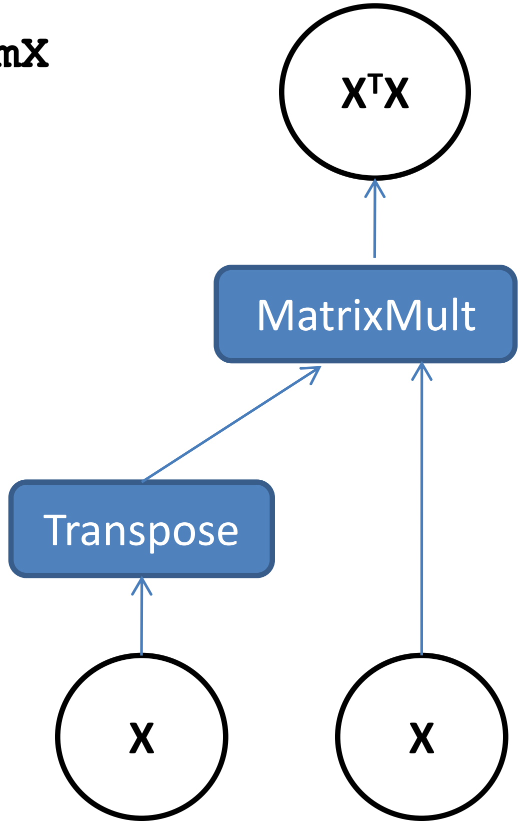
- Computation of $X^T X$ in example

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

- Naïve execution

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

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



Optimization Example

- Computation of X^TX in example

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

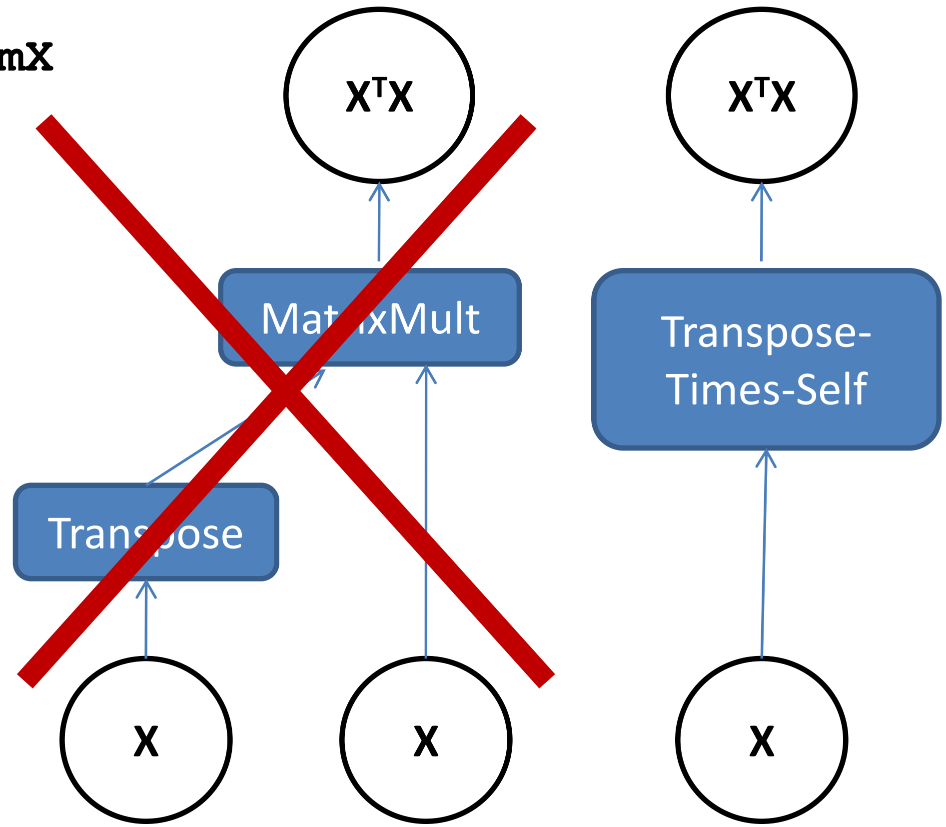
- Naïve execution

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

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

- **Logical optimization**

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



Tranpose-Times-Self

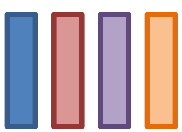
- Mahout computes X^TX via **row-outer-product** formulation
  - executes in a single pass over row-partitioned X

$$X^T X = \sum_{i=0}^{m} x_{i\bullet} x_{i\bullet}^T$$

Tranpose-Times-Self

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

$$X^T X = \sum_{i=0}^{m} x_{i\bullet} x_{i\bullet}^T$$

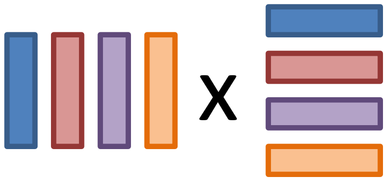


X^T

Tranpose-Times-Self

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

$$X^T X = \sum_{i=0}^{m} x_{i\bullet} x_{i\bullet}^T$$

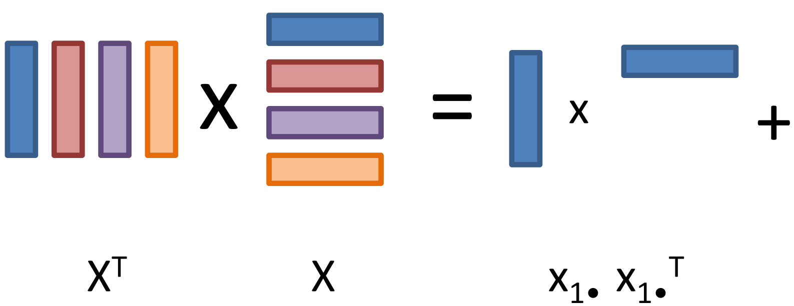


X^T X

Tranpose-Times-Self

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

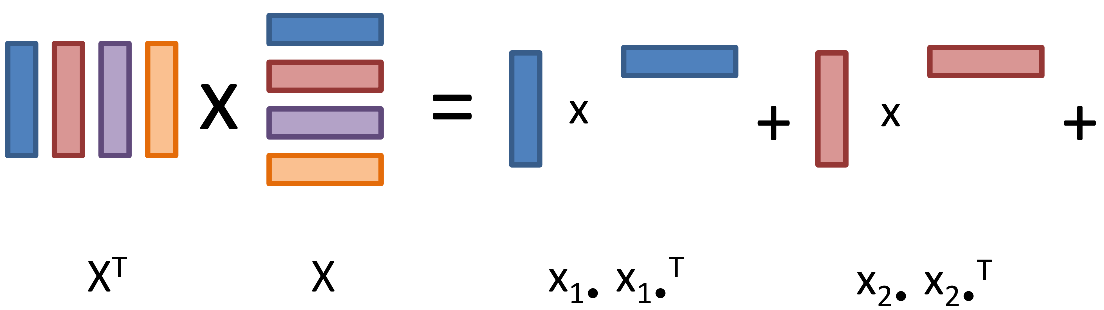
$$X^T X = \sum_{i=0}^{m} x_{i\bullet} x_{i\bullet}^T$$



Tranpose-Times-Self

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

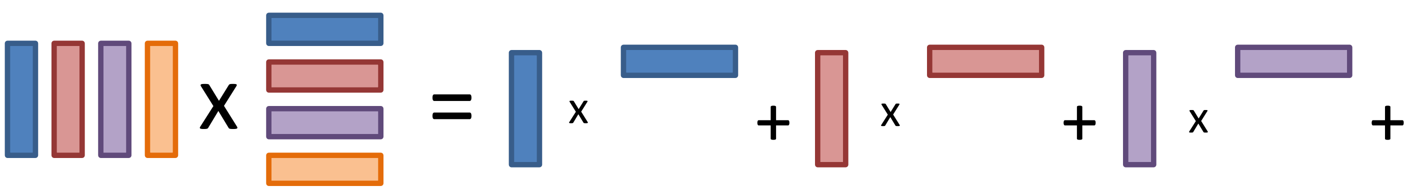
$$X^T X = \sum_{i=0}^{m} x_{i\bullet} x_{i\bullet}^T$$



Tranpose-Times-Self

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

$$X^T X = \sum_{i=0}^{m} x_{i\bullet} x_{i\bullet}^T$$

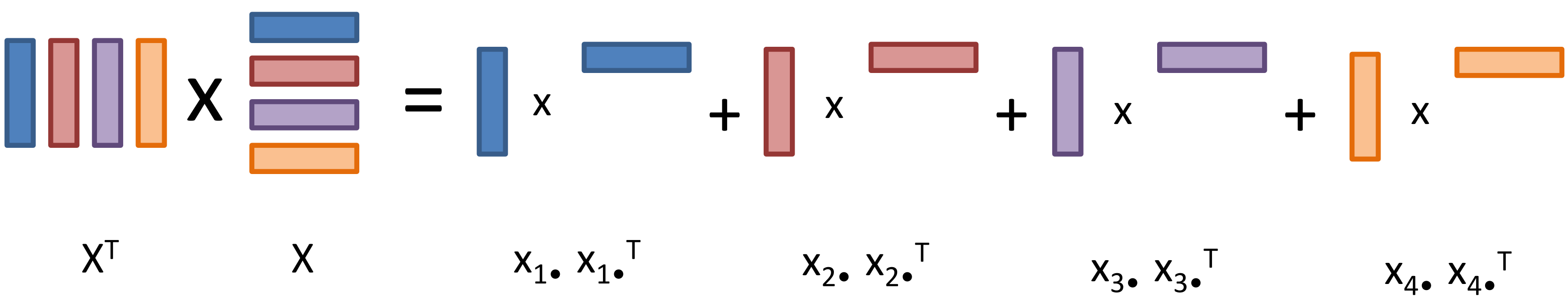


X^T X $x_{1\bullet}\, x_{1\bullet}^T$ $x_{2\bullet}\, x_{2\bullet}^T$ $x_{3\bullet}\, x_{3\bullet}^T$

Tranpose-Times-Self

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

$$X^T X = \sum_{i=0}^{m} x_{i\bullet} x_{i\bullet}^T$$



Overview

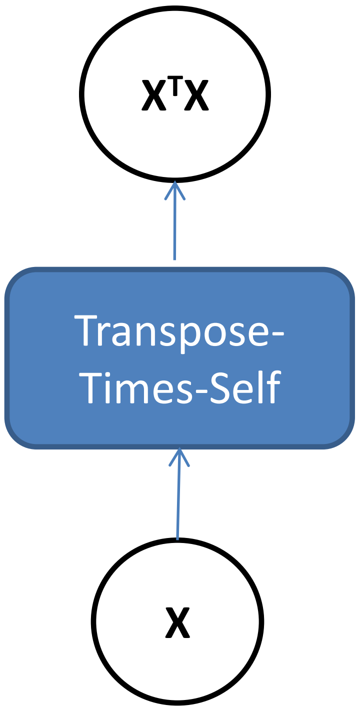
- Apache Mahout: Past & Future
- A DSL for Machine Learning
- Example
- Under the covers
- ***Distributed computation of $X^T X$***

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 (many rows, few columns)

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



Physical operator „*AtA*“

$$\begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \end{pmatrix}$$

X

Physical operator „*AtA*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

X_1

worker 1

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

Physical operator „*AtA*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

for 1st partition

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

X_1

worker 1

for 1st partition

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

Physical operator „*AtA*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

X_1

worker 1

for 1st partition

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

for 1st partition

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

Physical operator „*AtA*“

$$X = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$X_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ \boxed{1 \quad 0} & & 1 & 0 \end{pmatrix}$$

X_1

worker 1

for 1st partition

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

$$X_2 = \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

for 1st partition

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

Physical operator „*AtA*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

X_1

worker 1

for 1st partition

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

for 2nd partition

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

for 1st partition

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

for 2nd partition

Physical operator *AtA*

$$\begin{pmatrix} 1 & 1 & \boxed{1 \quad 0} \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

A_1

worker 1

for 1st partition

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

for 2nd partition

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

A

$$\begin{pmatrix} 0 & 0 & \boxed{1 \quad 1} \end{pmatrix}$$

A_2

worker 2

for 1st partition

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

for 2nd partition

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

Physical operator „*AtA*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

X_1

worker 1

for 1st partition

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

for 2nd partition

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

for 1st partition

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

for 2nd partition

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

Physical operator „*AtA*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

X_1

worker 1

for 1st partition

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

for 2nd partition

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

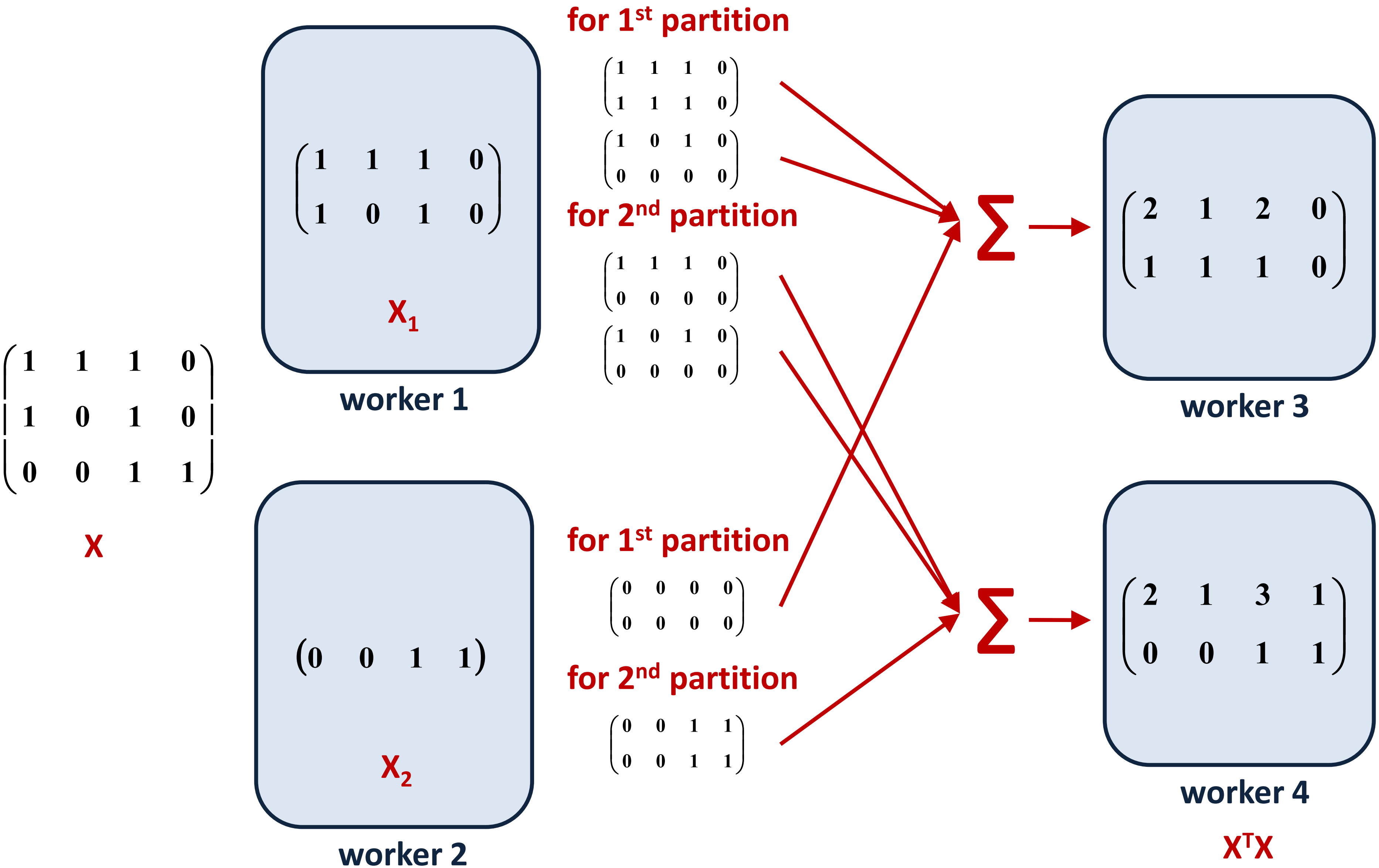
for 1st partition

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

for 2nd partition

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Physical operator „*AtA*“



Physical operator „*AtA_slim*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

Physical operator „*AtA_slim*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

X_1

worker 1

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$$

X_2

worker 2

Physical operator „*AtA_slim*“

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

X

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 2 & 1 & 2 & 0 \\ - & 1 & 1 & 0 \\ - & - & 2 & 0 \\ - & - & - & 0 \end{pmatrix}$$

$X_1 \longrightarrow X_1^T X_1$

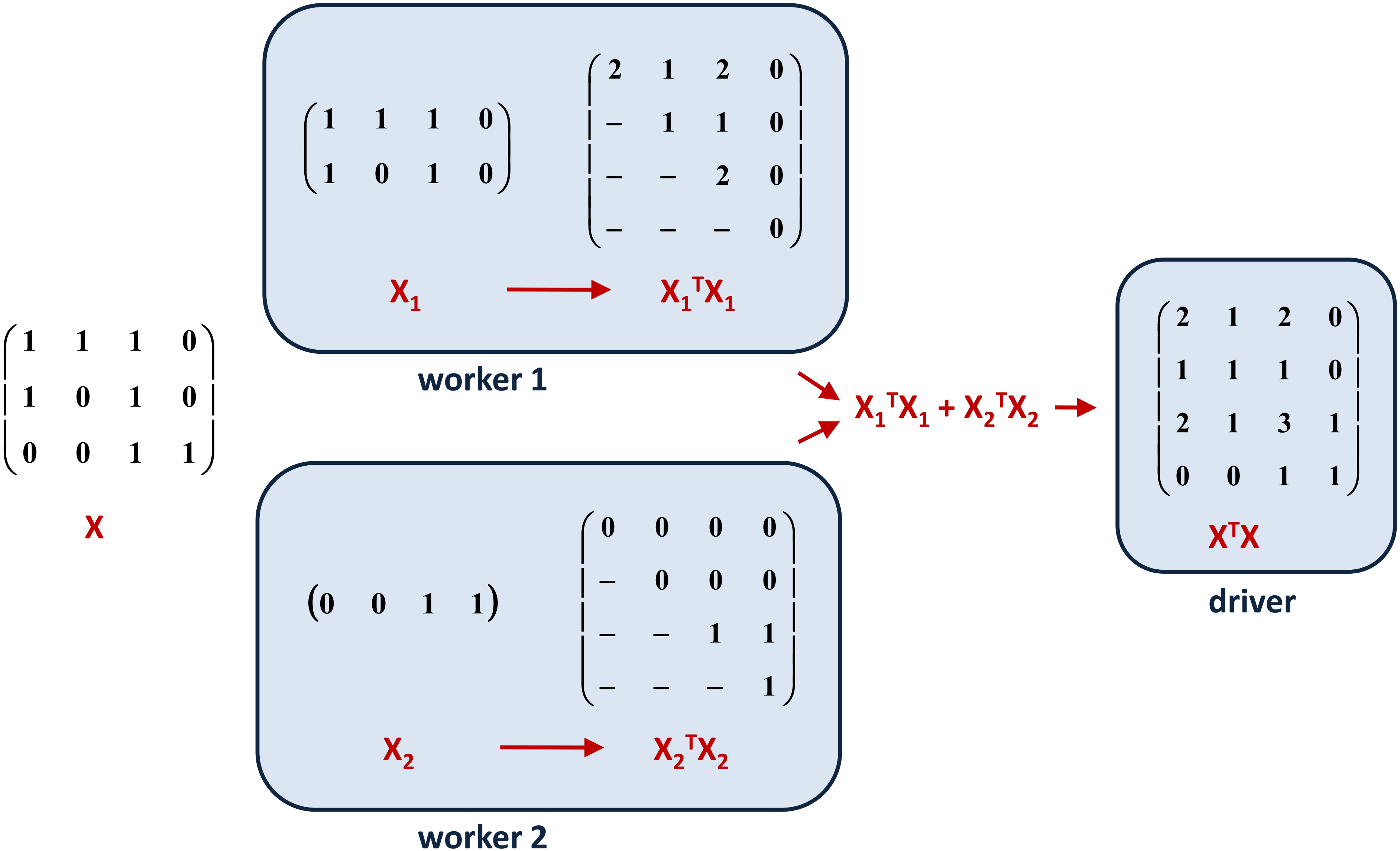
worker 1

$$\begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 \\ - & 0 & 0 & 0 \\ - & - & 1 & 1 \\ - & - & - & 1 \end{pmatrix}$$

$X_2 \longrightarrow X_2^T X_2$

worker 2

Physical operator *„AtA_slim“*



Summary

- **MapReduce outdated** as abstraction for distributed machine learning
- R/Matlab-like DSL for **declarative implementation of algorithms**
- **Automatic compilation, optimization and parallelization** of programs written in this DSL
- Execution on novel distributed engines like **Apache Spark** and **Apache Flink**

Thank you. Questions?

Tutorial for playing with the new Mahout DSL:
http://mahout.apache.org/users/sparkbindings/play-with-shell.html

Apache Flink Meetup in Berlin:
http://www.meetup.com/Apache-Flink-Meetup/

Follow me on twitter
@sscdotopen