Mahout Scala Bindings
and
Mahout Spark Bindings
for Linear Algebra Subroutines

Working Notes and Manual

Dmitriy Lyubimov*

Abstract

In recent years significant effort was spent to produce semantically friendly environments for linear algebra. Working with vector, matrix and tensor data structures as a single data type offers essential qualities necessary for rapid prototyping of algebraically defined mathematical problems. The other wanted quality is the convenience of the same environment as a programming language. Yet another one is doing things at scale. Yet another highly desirable capability of the same environment is plotting and visualization. Without bringing any detailed review of existing environments here, the author however offers an opinion that while a lot of environments succeed in one or more of these aspects, none of them however adequately addresses all of them at the same time and at a reasonable cost.

Unlike many other environments, Mahout-math Matrix model was targeting both dense and sparse data structures from the very beginning both in type modeling and cost-based optimized computations.

In this work we are trying to bring semantic explicitness to Mahout’s in-core and out-of-core linear algebra subroutines, while adding benefits of strong programming environment of scala, and capitalizing on scalability benefits of Spark and GraphX.

Overview

Mahout Scala and Spark Bindings is a package aiming to provide a R-like look and feel to Mahout’s in-core and out-of-core Spark-backed linear algebra. It is built in the image of R’s base package. So if you are familiar with basic R matrix primitives, you should feel right at home.

There are, at the moment, 3 major types to be operated on: in-core vectors, in-core matrices (including numerous specialized types), and distributed row matrices (DRM). SparkBindings expressions can mix in all three types of things.

The manual is organised by giving DSL features by example. That means that capabilities are wider than those shown, and may change behind the scenes as the work develops. However, the authors try to facilitate and encourage particular style given, and retain behind-the-scenes compatibility with the examples given. §1 enumerates in-core DSL operators only. §2 describes operators involving combinations of DRM, in-core vectors or matrices requiring out-of-core optimization and processing, as well conceptual notes about checkpointing, caching, broadcasting and behind-the-scenes optimizing.

* dlyubimov at apache dot org

1 initially, an adaptation of Colt linear algebra library
Contents

1 Mahout in-core algebraic Scala Bindings 4

1.1 Imports ....................................... 4
1.2 Inline initialization .......................... 4
1.3 Slicing and Assigning ....................... 5
1.4 BLAS-like operations ....................... 6
1.5 Decompositions .................. ........... 8
   - Cholesky decomposition .................. 8
   - SVD ...................................... 8
   - EigenDecomposition ...................... 8
   - QR decomposition ......................... 8
   - Rank .................................... 8
   - In-core SSVD .............................. 9
   - Solving linear equation systems and matrix inversion ......... 9
1.6 Misc ........................................... 9
1.7 Random matrices ........................... 9
   - Functional matrix views .................. 9
   - $U(0, 1)$ random matrix view .......... 10
   - $U(-1, 1)$ random matrix view .......... 10
   - Univariate $N(0, 1)$ matrix view ....... 10
1.8 Iterators .................................... 10
   - Iterating over rows in a matrix ...... 10
   - Iterating over non-zero and all elements of a vector or matrix .. 10
1.9 Bringing it all together: in-core SSVD ....... 10
1.10 Stochastic PCA ................. .............. 11
1.11 Pitfalls ................................... 12

2 Out-of-core linalg bindings 13

2.1 Initializing Mahout/Spark context ...... 13
2.2 Recommended imports for Scala & Spark Bindings ......... 13
2.3 DRM Persistence operators ................ 14
   - Loading DRM off HDFS .................. 14
   - Parallelizing from an in-core matrix .... 14
   - Empty DRM ................................ 14
   - Collecting to driver’s jvm in-core ....... 14
1 Mahout in-core algebraic Scala Bindings

In-core DSL is hardly much more than just a syntactic sugar over `org.apache.mahout.math.Matrix(Vector)` trait implementations. As such, all originally implemented operation signatures of Mahout are also retained.

1.1 Imports

The following two scala imports are typically used to enable Mahout Scala DSL bindings for Linear Algebra:

```scala
import org.apache.mahout.math._
import scalabindings._
import RLikeOps._
```

Another option is to use “matlab like” dialect by doing

```scala
import MatlabLikeOps._
```

However, Matlab-like DSL dialect adherence to original Matlab dialect is far less optimal that R dialect due to the specifics of operator support in scala, so we just will limit ourselves to R-like dialect here.

1.2 Inline initialization

Dense vectors

```scala
val denseVec1: Vector = (1.0, 1.1, 1.2)
val denseVec2 = dvec(1, 0, 1.1, 1.2)
```

Sparse vectors

```scala
val sparseVec = svec((5 -> 1) :: (10 -> 2.0) :: Nil)
val sparseVec2: Vector = (5 -> 1.0) :: (10 -> 2.0) :: Nil
```

matrix inline initialization, either dense or sparse, is always row-wise:
dense matrices :

```scala
val A = dense((1, 2, 3), (3, 4, 5))
```
sparse matrices

```scala
val A = sparse(
    (1, 3) :: Nil,
    (0, 2) :: (1, 2.5) :: Nil
)
```

\(^2\)See link: [original proposal](#)
1.3 Slicing and Assigning

diagonal matrix with constant diagonal elements

\[
\text{diag}(10, 3.5)
\]
diagonal matrix with main diagonal backed by a vector

\[
\text{diagv}((1, 2, 3, 4, 5))
\]
Identity matrix

\[
\text{eye}(10)
\]

Obviously, direct initialization of any vector or matrix type in Mahout is still available with regular operation `new`.

1.3 Slicing and Assigning

getcng vector element

\[
\text{val } d = \text{vec}(5)
\]
setting vector element

\[
\text{vec}(5) = 3.0
\]
getting matrix element

\[
\text{val } d = m(3, 5)
\]
setting matrix element (setQuick() behind the scenes)

\[
\text{M}(3, 5) = 3.0
\]

Getting matrix row or column

\[
\text{val rowVec} = \text{M}(3, ::)
\]
\[
\text{val colVec} = \text{M}(::, 3)
\]

Setting matrix row or column

\[
\text{M}(3, ::) = (1, 2, 3)
\]
\[
\text{M}(::, 3) = (1, 2, 3)
\]

thru vector assignment also works
1.4 **BLAS-like operations**

**plus/minus**, either vector or matrix or numeric, with assignment or not

\[
\begin{align*}
  a + b \\
  a - b \\
  a + 5.0 \\
  a - 5.0
\end{align*}
\]

Hadamard (elementwise) product, either vector or matrix or numeric operands
1.4 BLAS-like operations

\[ a \times b \]
\[ a \times 5 \]

same things with assignment, matrix, vector or numeric operands

\[ a \mathbin{+}= b \]
\[ a \mathbin{-}= b \]
\[ a \mathbin{+}= 5.0 \]
\[ a \mathbin{-}= 5.0 \]
\[ a \mathbin{*=} b \]
\[ a \mathbin{*=} 5 \]

One nuance here is associativity rules in scala. E.g. \(1/x\) in R (where \(x\) is vector or matrix) is elementwise inverse operation and in scala realm would be expressed as

```scala
val xInv = 1 /: x
```

and R's \(5.0 - x\) would be

```scala
val x1 = 5.0 -: x
```

Even trickier and really probably not so obvious stuff:

\[ a \mathbin{-}= b \]

assigns \(a - b\) to \(b\) (in-place) and returns \(b\). Similarly for \(a \mathbin{/}= b\) or \(1 \mathbin{/}= v\).

(all assignment operations, including \(=\), return the assignee argument just like in C++)

dot product (vector operands)

\[ a \mathbin{dot} b \]

matrix /vector equivalency (or non-equivalency). Dangerous, exact equivalence is rarely useful, better use norm comparisons with admission of small errors

\[ a \mathbin{===} b \]
\[ a \mathbin{!===} b \]

Matrix multiplication (matrix operands)

\[ a \mathbin{%*%} b \]

for matrices that explicitly support optimized right and left multiply (currently, diagonal matrices)

right-multiply (for symmetry, in fact same as \(\%*\%\))

\[ \text{diag}(5,5) :\mathbin{%*%} b \]
optimized left multiply with a diagonal matrix:

\[ A \  %*%: \ \text{diag}(5, 5) \ # \ i.e. \ same \ as \ (\text{diag}(5, 5) :\%*\%\ A.t) \ t \]

Second norm, vector or matrix argument:

\[ a.norm \]

Finally, transpose

\[ \text{val } Mt = M.t \]

Note: Transposition currently is handled via \text{view}, i.e. updating a transposed matrix will be updating the original. Also computing something like \( X^\top X \)

\[ \text{val } XtX = X.t \ %*% X \]

will not therefore incur any additional data copying.

### 1.5 Decompositions

All arguments in the following are matrices.

- **Cholesky decomposition** (as an object of a CholeskyDecomposition class with all its operations)

  \[ \text{val } ch = \text{chol}(M) \]

- **SVD**

  \[ \text{val } (U, V, s) = \text{svd}(M) \]

- **EigenDecomposition**

  \[ \text{val } (V, d) = \text{eigen}(M) \]

- **QR decomposition**

  \[ \text{val } (Q, R) = \text{qr}(M) \]

- **Rank** Check for rank deficiency (runs rank-revealing QR)

  \[ M.\text{isFullRank} \]
In-core SSVD

\[
\text{val (U, V, s) = ssvd(A, k=50, p=15, q=1)}
\]

Solving linear equation systems and matrix inversion This is fully similar to R semantics. There are three forms of invocation:

\[
\begin{align*}
\text{solve(A, B)} & \quad \text{// solves } AX = B \\
\text{solve(A, b)} & \quad \text{// solves } Ax = b \\
\text{solve(A)} & \quad \text{// computes inverse } A^{-1}
\end{align*}
\]

1.6 Misc

vector cardinality

\[
a.length
\]

matrix cardinality

\[
m.nrow \\
m.ncol
\]

means and sums

\[
m.colSums \\
m.colMeans \\
m.rowSums \\
m.rowMeans
\]

a copy-by-value (vector or matrix)

\[
\text{val b = a cloned}
\]

1.7 Random matrices

See org.apache.mahout.math.Matrices for up-to-date information

1.7.1 Functional matrix views.

On Mahout-math side (i.e. java side) there is a concept of a “functional view”. Java side has a type, IntInt-Function. An argument of that functional type could be provided to construct a (dense) matrix readonly view via Matrices.functionalMatrixView(m, n, gf, denseLike). ‘gf’ function is expected to be idempotent (i.e. return the same matrix element for the same combination of indices. Specialization of the functional views are used for a transposed matrix view, as well as the following random matrices views.
1.7.2 $U(0,1)$ random matrix view
val inCoreA = Matrices.uniformView(m, n, seed)

1.7.3 $U(-1,1)$ random matrix view
val inCoreA = Matrices.symmetricUniformView(m, n, seed)

1.7.4 Univariate $N(0,1)$ matrix view
val inCoreA = Matrices.gaussianView(m, n, seed)

1.8 Iterators
Mahout-math already exposes a number of iterators. Scala code just needs to import collection.JavaConversions._ to enable implicit conversions to scala iterators.

import collection._
import JavaConversions._

1.8.1 Iterating over rows in a matrix
for (row <- m) {
    ... do something with row
}

1.8.2 Iterating over non-zero and all elements of a vector or matrix
Note that Vector.Element also has some implicit syntactic sure, e.g. to add 5.0 to every non-zero element the following code may be used:

for (row <- m; el <- row.nonZero) el = 5.0 + el
... or
for (row <- m; el <- row.nonZero) el := 5.0 + el

Similarly, row.all produces iterator over all elements in a row (vector).

1.9 Bringing it all together: in-core SSVD
Just to illustrate semantic clarity, we will adduce a source for in-core SSVD code.

/**
 * In-core SSVD algorithm.
 * @param a input matrix A
 * @param k request SSVD rank
 */
* @param p oversampling parameter
* @param q number of power iterations
* @return (U,V,s)
*/

def ssvd(a: Matrix, k: Int, p: Int = 15, q: Int = 0) = {
  val m = a.nrow
  val n = a.ncol
  if (k > min(m, n))
    throw new IllegalArgumentException(
      "k cannot be greater than smaller of m,n")
  val pfxed = min(p, min(m, n) - k)
  // actual decomposition rank
  val r = k + pfxed
  val rnd = RandomUtils.getRandom
  val omega = Matrices.symmetricUniformView(n, r, rnd.nextInt)
  var y = a %*% omega
  var yty = y.t %*% y
  var at = a.t
  var ch = chol(yty)
  var bt = ch.solveRight(at %*% y)
  // power iterations
  for (i <- 0 until q) {
    y = a %*% bt
    yty = y.t %*% y
    ch = chol(yty)
    bt = ch.solveRight(at %*% y)
  }
  val bbt = bt.t %*% bt
  val (uhat, d) = eigen(bbt)
  val s = d.sqrt
  val u = ch.solveRight(y) %*% uhat
  val v = bt %*% (uhat %*%: diagv(1 /: s))
  (u(::, 0 until k), v(::, 0 until k), s(0 until k))
}

1.10 Stochastic PCA

/**
  * PCA based on SVD that runs without forming an always-dense A-(colMeans(A)) input for SVD. This
  * follows the solution outlined in MAHOUT-817. For in-core version it, for most part, is supposed
  * to save some memory for sparse inputs by removing direct mean subtraction.<p>
  *
  * Hint: Usually one wants to use AV which is approximately USigma, i.e.<code>u %*%: diagv(s)</code>.
  * If retaining distances and original scaled variances not that important, the normalized PCA space
  * is just U.
  *
  * Important: data points are considered to be rows.
  *
  * @param a input matrix A
  * @param k request SVD rank
  * @param p oversampling parameter
  * @param q number of power iterations
  * @return (U,V,s)
  */

def spca(a:Matrix, k: Int, p: Int = 15, q: Int = 0)
Stochastic PCA is a re-flow of MAHOUT-817 for in-core DSL. One usually needs output $\mathbf{A}\mathbf{V} \approx \mathbf{U}\mathbf{\Sigma}$:

```scala
val (inCoreU, _, s) = spca(a = input, k = 30, q = 1)
val uSigma = inCoreU %*%: diagv(s)
```

### 1.11 Pitfalls

This one the people who are accustomed to writing R linear algebra will probably find quite easy to fall into. R has a nice property, a copy-on-write, so all variables actually appear to act as no-side-effects scalar-like values and all assignment appear to be by value. Since scala always assigns by reference (except for AnyVal types which are assigned by value), it is easy to fall prey to the following side effect modifications:

```scala
val m1 = m
m1 += 5.0 // modifies m as well
```

A fix is as follows:

```scala
val m1 = m cloned
m1 += 5.0 // now m is intact
```
2 Out-of-core linalg bindings

The subject of this section are solely operations applicable to Mahout’s DRM (distributed row matrix).

Spark Binding’s DRM persistence to HDFS is compatible with all Mahout’s MR algorithms using DRM such as ssvd or seq2sparse.\(^3\)

Once loaded into spark, DRM is represented by Spark partitions initially consisting of handful of row vectors.

Here and on, I will denote spark-backed DRM references as A, whereas in-core matrices as inCoreA.

2.1 Initializing Mahout/Spark context

Many (if not all) operations will require a Spark context. Spark context can be passed in two ways: (1) as an implicit value; and as passed down from a parent source (DRM’s backing RDD).

To initialize Mahout/Spark session, just create an implicit value of a specifically prepped Spark context:

```scala
implicit val mahoutCtx = mahoutSparkContext(
   masterUrl = "local",
   appName = "MahoutLocalContext"
   // [...]
)
```

Parameter masterUrl points to Spark’s master. Note that Mahout expects either MAHOUT_HOME environment or -Dmahout.home=... java system variable to point to Mahout home directory in order to collect relevant jars for the Spark sessions.

From there on, as long as Mahout-initialized Spark context is exposed thru implicit variable, attribute or paremeter, there’s no need to specify it explicitly for any of the successive operations.

Note that as of the time of this writing Spark sessions cannot coexist in the same jvm, even though a single spark session is reentrant and can handle requests from more than one thread.

2.2 Recommended imports for Scala & Spark Bindings

For seamless in-core & distributed algebraic DSL:

```scala
// Import matrix, vector types, etc.
import org.apache.mahout.math._

// Import scala bindings operations
import scalabindings._

// Enable R-like dialect in scala bindings
import RLikeOps._
```

---

\(^3\)This statement needs comprehensive QA throughout; but intent is true.
2.3 DRM Persistence operators

2.3.1 Loading DRM off HDFS

val A = drmFromHDFS(path = hdfsPath)

2.3.2 Parallelizing from an in-core matrix

val inCoreA = dense((1, 2, 3), (3, 4, 5))
val A = drmParallelize(inCoreA)

2.3.3 Empty DRM

val A = drmParallelizeEmpty(100, 50)

2.3.4 Collecting to driver’s jvm in-core

val inCoreA = A.collect()

Warning: Collection of distributed matrices is now happening implicitly whenever conversion to in-core
(o.a.m.math.Matrix) type is required:

val inCoreA:Matrix = ...
val drmB:DrmLike[Int] = ..
val inCoreC:Matrix = inCoreA %*% drmB
// implied: (incoreA %*%) drmB).collect

All subsequent snippets assume the relevant packages are imported.
Mahout shell does all these imports automatically.
2.4 Logical algebraic operators on DRM matrices

We will define a logical set of operators that are familiar to users of R, which are elementwise +, -, *, / as well as matrix multiplication `%*%` and transposition. General rule is that we try to do a subset of the same things found in the in-core DSL. In particular, since all distributed matrices are immutable, there are no assignment versions (e.g. `A += B`).

Logical operators comprised into expression do not however mean that concrete physical plan is materialized until the expression is “checkpointed” – directly or indirectly. In terms of Spark, this is called “action”.

Unlike with Spark, we want to discern two types of “actions”: optimizer action and computational action.

**Optimizer actions.** *Optimizer action* triggers materialization of a physical plan (concrete RDD graph with result marked for Spark caching), backed by CheckpointedDRM. CheckpointedDRM serves as a cutoff boundary for optimizer action. Optimizer action does not trigger actual computation of result data set. Right now optimizer action is triggered explicitly by DRMLike#checkpoint().

Let consider two examples:

```scala
val A = drmParallelize ()
val B = drmParallelize ()
val C = A %*% B.t
val D = C.t
val E = C.t %*% C
D.writeDRM(path =)
E.writeDRM(path =)
```

In this example, optimizer optimizes separately 2 pipelines: \( D = BA^T \) and \( E = (AB^T)^T (AB^T) \) using same matrices \( A \) and \( B \) as root of both computations. Now let’s consider the following modified example:

```scala
val A = drmParallelize ()
val B = drmParallelize ()
val C = (A %*% B.t).checkpoint
val D = C.t
val E = C.t %*% C
D.writeDRM(path =)
E.writeDRM(path =)
```

In this case, which is functionally equivalent to the previous one, the optimizer considers 3 separate pipelines: \( C = AB^T, D = C^T \) and \( E = C^T C \) while caching optimized plan and intermediate result for \( C \) into the Spark cache. Introducing checkpoints may improve “wall time” (since matrices \( D \) and \( E \) will be triggered for action

---

4 if you see an error here along the lines “no implicit view available from A => org.apache.hadoop.io.Writable” most likely you need just to import SparkContext._.
at different time and optimizer wouldn’t be able to consider computational graph that includes both at the same time). But even in the first example optimizer will be able to figure to optimize $E = \left(AB^\top\right)^\top \left(AB^\top\right)$ as $t\_\text{square}(\text{product}(A, B^\top))$ pipeline, i.e. into only two sequential physical operators.

In either of the examples, nothing happens in the backend until a computational action is triggered for either of $E$ or $D$.

It doesn’t matter how many times checkpointing is called on a logical operator, same logical operator will be optimized and set for caching policy only once.

**Computational actions.** *Computational action* leads to result being computed and (optionally?) placed into Spark cache. Such actions will also lazily and implicitly trigger linalg optimizer checkpointing. Currently, computational actions include writeDrm(), collect(), blockify() and sometimes could also be triggered implicitly by optimizer activity beyond current checkpoint cut-off (if checkpointed but not computed and cached yet) to run some cost estimates necessary for the optimizer beyond checkpointing, potentially future actions associated with DRM sub-blocking.

E.g. in the second example, running

$$E\text{.writeDrm(path)}$$

will trigger computational actions for $E$ and, implicitly, for $C$.

All these rules follow the same patterns as for the in-core arguments.

**Caching in Spark’s Block Manager.** Every checkpoint can be, and by default, is, pushed into Spark’s memory block manager. Default policy is MEMORY__ONLY, but storage level can be specified explicitly as a parameter to checkpoint() call. Actual push of data to memory block manager happens no sooner that actual partition computation occurs for the first time (i.e. at the first occurrence of a computational action of the pipeline involving the result in question). Checkpointed DRMs may later be explicitly uncached from block manager (asynchronously) if desired, e.g.:

```scala
val drmA = (/*..drm expression..*/).checkpoint(CacheHint.MEMORY_AND_DISK)
... some computational actions involving drmA
... drmA is not needed anymore
drmA.uncache()
```

If argument is not cached by the time the uncache() call has occurred, nothing of substance happens.

### 2.4.1 Transposition

$$A.t$$

---

5See Spark manual to understand interaction with Block Manager and storage levels in detail.
2.4.2 Elementwise +, -, *, /

\[ M = A + B \]

\[ M = A - B \]

\[ M = A \odot B \text{ (Hadamard)} \]

\[
M = \begin{pmatrix}
a_{11} & a_{12} & \ldots & a_{1n} \\
b_{11} & b_{12} & \ldots & b_{1n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m1} & a_{m2} & \ldots & a_{mn} \\
b_{m1} & b_{m2} & \ldots & b_{mn}
\end{pmatrix}
\]

(elementwise division)

All this operations require identical geometry of operands and row keying types that will be asserted at optimizer checkpointing time.

A + B
A - B
A * B
A / B

Binary operators involving in-core argument (only on int-keyed DRMs)

A + inCoreB
A - inCoreB
A * inCoreB
A / inCoreB
A :+ inCoreB
A :- inCoreB
A :* inCoreB
A :/ inCoreB
inCoreA +: B
inCoreA -: B
inCoreA *: B
inCoreA /: B

Note spark associativity change (e.g. inCoreA +: B means B.leftMultiply(A), just like with both in-core arguments). Important thing here is that whenever operator arguments include both in-core and out-of-core arguments, operator can only be associated with the out-of-core argument to support distributed implementation.

2.4.3 Matrix-matrix multiplication %*%

\[ M = AB \]

A %*% B
A %*% inCoreB
A %*% inCoreDiagonal (i.e. things like A %*% diagv(d))
A :%*% inCoreB
inCoreA %*%: B
2.5  Slicing

Same as above, when both in-core and out-of-core arguments used, associativity of operation must follow the out-of-core (DRM) argument in the expression.

2.4.4 Matrix-vector multiplication %*%

Currently, we support a right-multiply product of a DRM and an in-core Vector ($Ax$), resulting in a distributed single-column DRM, which then of course could be collected in front (usually that’s the desired outcome):

```scala
val Ax = A %*% x
val inCoreX = Ax.collect(::,0)
```

There are 2 physical operators associated with this: $Ax$ and $A^\top x$.

2.4.5 Matrix-scalar +,-,*,/  

In this context, matrix-scalar operations mean element-wise operations of every matrix element and a scalar.

```scala
A + 5.0  
A - 5.0  
A :- 5.0  
5.0 -:/ A  
A * 5.0  
A / 5.0  
5.0 /:/ A
```

Note that $5.0 -:/ A$ means $m_{ij} = 5 - a_{ij}$ and $5.0 /:/ A$ means $m_{ij} = \frac{5}{a_{ij}}$ for all elements of the result.

2.5 Slicing

Slicing (without assigning) is supported mostly identically to in-core slicing. Slicing row or range is of `Range scala type, which typically can be inlined as $x$ to $y$ or $x$ until $y$. All-range is given by ::.

General slice

```scala
A(100 to 200, 100 to 200)
```

Horizontal block

```scala
A(::, 100 to 200)
```

Vertical block

```scala
A(100 to 200, ::)
```

Note: if row range is not all-range (::) then the DRM must be Int-keyed. Row slicing in general case is not supported for key types other than Int.
2.6 Stitching (cbind and rbind)

We can stitch two matrices side by side (cbind R semantics):

```scala
val drmAnextToB = drmA cbind drmB
```

Or, which is the same in Scala,

```scala
val drmAnextToB = drmA.cbind(drmB)
```

Obviously, our variation of `cbind` is accepting only two arguments, but we can stitch more matrices by chaining the operation.

Analogously, vertical distributed concatenation is available since MAHOUT-1596 via `rbind`.

2.7 Custom pipelines on blocks

Pretty often there's a need to do something with the matrix expressed as blocks. Some physical operators are also more effective once working with matrix blocks rather than individual rows. Internally, Mahout's matrix pipeline (lazily) blockifies every data partition into `BlockifiedDrmTuple` blocks whenever first physical operator requiring blocking is encountered. After that, any row-wise physical operators work on row vector views of the blocks.

Here is definition for DRM block tuple type:

```scala
/** Drm block-wise tuple: 
 Array of row keys and the matrix block. */
type BlockifiedDrmTuple[K] = (Array[K], _ <: Matrix)
```

DRM operator `mapBlock` provides transformational access to the vertical blockified tuples of the matrix. (Current implementation also guarantees that there's exactly one block per map task).

Here is unit test that demonstrates use of `mapBlock` operator by producing $A + 1.0$:

```scala
val inCoreA = dense((1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6))
val A = drmParallelize(m = inCoreA, numPartitions = 2)
val B = A.mapBlock(/* Inherit width */) {
  case (keys, block) => keys -> (block += 1.0)
}
val inCoreB = B.collect
val inCoreBControl = inCoreA + 1.0
println(inCoreB)
// Assert they are the same
(inCoreB - inCoreBControl).norm should be < 1E-10
```

The constrain is that operator `mapBlock` should not attempt to change the height of the block, in order to provide correct total matrix row count estimate to optimizer after application of the operator. `MapBlock` operator may change `width` (i.e. column count) of the matrix; if it does so, it needs to supply it to first `ncol` parameter of the `mapBlock(newNCol)` call. Otherwise, it is assumed operator has inherited the width of the original matrix. The geometry of the block returned is asserted at run time, as geometry is vitally important for the coherence of linear operators.

Another note is that it is ok to return a reference to a modified same in-core block. This is actually recommended whenever possible (note the `+=` operator in the example) to avoid matrix copying.
2.8 Doing something completely custom

If flexibility of Drm api is not enough, it is always possible to exit out of opimizer-based algebra pipeline into pure spark RDD environment. The exit is possible at optimizer checkpoints, which are presented by CheckpointedDrmBase[K] trait. This trait has an rdd:DrmRdd[K] method, which returns a row-wise RDD of DrmTuple[K] type.

The row-wise tuple types and RDDs are defined as following:

```scala
/** Drm row-wise tuple */
type DrmTuple[K] = (K, Vector)
/** Row-wise organized DRM rdd type */
type DrmRdd[K] = RDD[DrmTuple[K]]
```

(type Vector here is org.apache.mahout.math.Vector).

E.g.:

```scala
val myRdd = (A %*% B).checkpoint().rdd
...
```

Similarly, an Rdd conforming to a type of DrmRdd, can be re-wrapped into optimizer checkpoint via

```scala
val rdd:DrmRdd[K] = ... //
val A = drmWrap(rdd = rdd, nrow = 50000, ncol = 100)
... // use A in a DRM pipeline
```

Parameters ncol and nrow (geometry) are optional. If not supplied, they will be recomputed off cached dataset. But if supplied, they must be accurate!

A note about serialization: the Spark bindings for Mahout support serialization of Vector and Matrix types (including their views and slices) via Kryo serialization. Hence, Spark context for Mahout is initialized with kryo serializer for all objects. This is something to keep in mind (Vector and Matrix objects can be broadcasted/collected, but there’s no way to revert to java-serialized-only support in spark session and use Mahout objects at the same time). This generally should not be a problem in Spark 0.9 since there’s a kryo serialization back for practically anything of interest in the twitter/chill that is used by Spark since 0.8.

2.9 Broadcasting vectors and matrices to closures.

Generally, one can create and use one-way closure attributes and use them at backend, e.g we can implement scalar matrix multiplication by a variable factor the following way:

```scala
val factor:Int = ...
val drm2 = drm1.mapBlock() {
    case (keys, block) => block *= factor
    keys -> block
}
```
As we can see, even though factor is initialized in front end, it can be easily used at backend closures running on every matrix vertical block in parallel. Very easy and elegant.

A slight wrinkle with that is, closure attributes must be java-serializable. This is, as it stands, not currently the case with in-core matrices and vectors. And even if they were, java serialization would be less compact on the wire than a custom serialization that Mahout in-core matrices use. E.g. the following fragment, implementing a vector subtraction from every matrix row, will fail with NotSerializableException:

```scala
val v:Vector = ...
val drm2 = drm1.mapBlock() {
  case (keys, block) =>
    for (row <- 0 until block.nrow) block(row,:) -= v
    keys -> block
}
```

Spark and similar execution backends supports “broadcast” feature which ensures that a broadcast variable is available to all backend running code. We abstract that away too. The fix to the previous fragment would be:

```scala
val v:Vector = ...
val bcastV = drmBroadcast(v)
val drm2 = drm1.mapBlock() {
  case (keys, block) =>
    for (row <- 0 until block.nrow) block(row,:) -= bcastV
    keys -> block
}
```

### 2.10 Computations providing ad-hoc summaries

There’s a number of operators that do not return a new distributed matrix. As such, some of them may or will trigger computational action. This is something to keep in mind.

#### 2.10.1 nrow, ncol

For example, matrix geometry properties (nrow, ncol) will trigger a summary computation if the geometry is not already inferred thru optimizer. If they do, they will checkpoint with storage level MEMORY_ONLY automatically.

#### 2.10.2 colSums, colMeans

```scala
val acs = A.colSums
val amean = A.colMeans
```

Those will always trigger a computational action. There’s no lazy behavior for these (vector properties are considered to be too bulky to be a lazy property). I.e. if one calls colSums() n times, then back end will actually recompute colMeans n times.
2.10.3  rowMeans, rowSums

2.10.4  Matrix norm

\[
\text{val rmse} = (A - U \times V.t).\text{norm} / \sqrt{A.nrow \times A.ncol}
\]

2.11  Distributed Decompositions

2.11.1  Imports for decompositions package

import org.apache.mahout.math._
...
import decompositions._

2.11.2  Distributed thin QR

For the classic QR decomposition of the form \( A = QR \), \( A \in \mathbb{R}^{m \times n} \), a distributed version is fairly easily achieved if \( A \) is tall and thin such that \( A^\top A \) fits in memory, i.e. \( m \) is large, but \( n \leq 5000 \). Under such circumstances, only \( A \) and \( Q \) are distributed matrices, and \( A^\top A \) and \( R \) are in-core products. We just compute in-core version of Cholesky decomposition in the form of \( LL^\top = A^\top A \). After that we take \( R = L^\top \) and \( Q = A (L^\top)^{-1} \). The latter is easily achieved by multiplying each vertical block of \( A \) by \( (L^\top)^{-1} \). (There’s no actual matrix inversion happening).

Corollary to this design are two facts: (1) rows of \( Q \) retain the same indexing type as rows of \( A \) (not necessarily int-keyed); and (2) \( A \) and \( Q \) are identically partitioned. Therefore, \( A \) and \( Q \) subsequently can be trivially zipped together if join of rows is needed (used in d-ssvd).

\[
\text{val (drmQ, inCoreR) = dqrThin(drmA)}
\]

The source of this method as of the time of this writing is extremely simple (probably too simple):

```scala
def dqrThin[K: ClassTag](A: DrmLike[K], checkRankDeficiency: Boolean = true): (DrmLike[K], Matrix) = {
  if (A.ncol > 5000)
    log.warn("A is too fat. A'A must fit in memory and easily broadcasted.\")
  implicit val ctx = A.context
  val AtA = (A.t \times A).checkpoint()
  val inCoreAtA = AtA.collect
  if (log.isDebugEnabled) log.debug("A'A=
  \%s\n".format(inCoreAtA))
  val ch = chol(inCoreAtA)
  val inCoreR = (ch.getL cloned) t
  if (log.isDebugEnabled) log.debug("R=\n  \%s\n".format(inCoreR))
```
if (checkRankDeficiency && !ch.isPositiveDefinite)
    throw new IllegalArgumentException("R is rank-deficient.")

val bcastAtA = drmBroadcast(inCoreAtA)

// Unfortunately, I don’t think Cholesky decomposition is serializable to backend. So we re-
// decompose A’A in the backend again.

// Compute Q = A*inv(L’) -- we can do it blockwise.
val Q = A.mapBlock() {
    case (keys, block) => keys -> chol(bcastAtA).solveRight(block)
}

Q -> inCoreR

Since we see that is navigated twice, it is recommended that it is checkpointed before calling this method to
avoid recomputation.

2.11.3 Distributed Stochastic SVD

Usage example:

val (drmU, drmV, s) = dssvd(drmA, k = 40, q = 1)

As a side effect of checkpointing, U and V values returned as logical operators (i.e they are neither check-
pointed nor computed). Therefore, there’s no physical work actually done to compute final U or V until they are actually used in a subsequent expression. So unlike the SSVDSolver, this does not require additional
parameters to configure which set of product combinations is actually computed in the end. Neat, isn’t it.

Source (for those who likes counting lines):

```scala
/**
 * Distributed Stochastic Singular Value decomposition algorithm.
 *
 * @param A input matrix A
 * @param k request SSVD rank
 * @param p oversampling parameter
 * @param q number of power iterations
 * @return (U,V,s). Note that U, V are non-checkpointed matrices
 * (i.e. one needs to actually use them
 * e.g. save them to hdfs in order to trigger their computation.
 */
def dssvd[K: ClassTag](A: DrmLike[K], k: Int, p: Int = 15, q: Int = 0):
    (DrmLike[K], DrmLike[Int], Vector) = {

    val drmA = A.checkpoint()
    val m = drmA.nrow
    val n = drmA.ncol
```
assert(k <= (m min n), "k cannot be greater than smaller of m, n.")

val pfxed = safeToNonNegInt((m min n) - k min p)

// Actual decomposition rank
val r = k + pfxed

// We represent Omega by its seed.
val omegaSeed = Random.nextInt()

// Compute Y = A*Omega.
var drmY = drmA.mapBlock(ncol = r) {
  case (keys, blockA) =>
    val blockY = blockA %*% Matrices.symmetricUniformView(blockA.ncol, r, omegaSeed)
    keys -> blockY
}

var drmQ = dqrThin(drmY.checkpoint())._1

// Checkpoint Q if last iteration
if (q==0) drmQ = drmQ.checkpoint()

// This actually is optimized as identically
// partitioned map-side A'B since A and Q should
// still be identically partitioned.
var drmBt = drmA.t %*% drmQ

// Checkpoint B' if last iteration
if (q==0) drmBt = drmBt.checkpoint()

for (i <- 1 to q) {
  drmY = drmA %*% drmBt
  drmQ = dqrThin(drmY.checkpoint())._1

  // Checkpoint Q if last iteration
  if (i == q) drmQ = drmQ.checkpoint()

  drmBt = drmA.t %*% drmQ

  // Checkpoint B' if last iteration
  if (i == q) drmBt = drmBt.checkpoint()
}

val inCoreBBt = (drmBt.t %*% drmBt).checkpoint(StorageLevel.NONE).collect
val (inCoreUHat, d) = eigen(inCoreBBt)
val s = d.sqrt
val drmU = drmQ %*% inCoreUHat
val drmV = drmBt %*% (inCoreUHat %*%: diagv(1 /: s))
(drmU(::, 0 until k), drmV(::, 0 until k), s(0 until k))

Done!
2.12 Adjusting parallelism of computations

2.11.4 Distributed Stochastic PCA

```scala
/**
 * Distributed Stochastic PCA decomposition algorithm. A logical reflow of the "SSVD-PCA options.pdf"
 * document of the MAHOUT-817.
 * @param A input matrix A
 * @param k request SSVD rank
 * @param p oversampling parameter
 * @param q number of power iterations (hint: use either 0 or 1)
 * @return (U,V,s). Note that U, V are non-checkpointed matrices (i.e. one needs to actually use them
 * e.g. save them to hdfs in order to trigger their computation.
 */
def dspca[K: ClassTag](A: DrmLike[K], k: Int, p: Int = 15, q: Int = 0):
  (DrmLike[K], DrmLike[Int], Vector) = <...>
```

Stochastic PCA is a re-flow of MAHOUT-817 for sparkbindings algebra. One usually needs output $AV ≈ UΣ$:

```scala
val (drmU, _, s) = dspca(a = drmA, k = 30, q = 1)
val drmUSigma = drmU %*% diagv(s)
```

2.11.5 Distributed regularized ALS

It is extremely easy now to alternate regularized ALS in distributed way:

```scala
for (i <- 0 until maxIterations) {
  drmV = (drmAt %*% drmU %*% solve(drmU.t %*% drmU -: diag(lambda, k))).checkpoint()
  drmU = (drmA %*% drmV %*% solve(drmV.t %*% drmV -: diag(lambda, k))).checkpoint()
}
```

Actual implementation is doing a tad more than just alternations, of course. To run convenience implementation of distributed ALS in Mahout (see scaladoc), use `dals(...)`:

```scala
val (drmU, drmV, _) = dals(A, ...).toTuple
```

2.12 Adjusting parallelism of computations

Sometimes there’s a need to adjust parallelism of computations. For most part it happens when we want to increase parallelism of computations. With Spark Bindings, parallelism value is construed as the number of splits.

---

6Here, note that rightmost term of the expression

$\text{drmV.t \%*% drmV -: diag(\lambda, k)}$

is an in-core matrix, so expression $\text{drmV.t \%*% drmV}$ is computed in distributed way, but matrix subtraction and solving the linear system $\text{solve(...)}$ is performed in the front-end. This is because matrix subtraction is associated with right hand side operand $\text{diag(\lambda, k)}$ which is an in-core matrix and requires an in-core left operand; so implicit matrix collection kicks in before doing subtraction.

If we used just `-` then operation of subtraction would have been associated with a tiny but distributed matrix product $V^T V$ and be carried in distributed way, which, given tiny size of the operands, would actually be just a waste of time communicating to machine nodes.

7Code untested for performance at this point.

---
First, minimum parallelism can be established by passing `parMin` parameter to `drmFromHdfs()` during load from DFS.

Second, parallelism operators could be inserted into expressions.

\[
\text{drmA.par(min = 100)}
\]

will establish minimum parallelism for outgoing computations of `drmA`.

\[
\text{drmA.par(exact = 100)}
\]

will establish exact parallelism; and

\[
\text{drmA.par(auto = true)}
\]

will try to do engine-specific automatic parallelism adjustment (with Spark, it is based on current default parallelism property for Spark). Auto parallelism will not decrease already established parallelism though.

### 2.13 Mahout shell for Spark

Use something like the following to run Mahout Spark shell:

\[
\text{MASTER=<spark-master-url> mahout/bin spark-shell}
\]

Assuming startup has completed successfully, all necessary imports are already done, and implicit instance of `DistributedContext` is already created (sdc).

See `simple.mscala` for an example of test script/script invocation.

Prerequisites to run Mahout Spark shell:

- Spark is installed, compiled and `SPARK_HOME` is set. Spark version should be that of what Mahout has been compiled with.
- Mahout is compiled, and `MAHOUT_HOME` is set.
- Other usual Spark requirements (e.g. it is not easy, if possible at all, to run Spark shell from behind NAT, etc.)

\[8\text{as of time of this writing, spark-0.9.1}\]
3 Optimizer notes

3.1 Physical operators

\(AB^\top\) This is using cartesian combination of vertical blocks of both sides.

\(A^\top B\) There are two implementations for this. If \(A\) and \(B\) are identically partitioned (which is tracked by optimizer throughout), then this is computed by zip+combine. If not, then inner join+combine is used.

\(A^\top\) This requires int-keyed input only. A direct transposition operator.

\(A^\top A\) This operator compiles squared matrix with arbitrary row keys. Result is always int-keyed. There are two different implementations here depending on whether \(n \times n\) upper triangular matrix fits in memory.

\(AB\) right-multiply where \(B\) is in-core. Broadcast for \(B\) is used in this map-block implementation.

... and some more less important. No time to list all.

3.2 Tracking the partitioning

Optimizer tracks identicity of partitioning of products. E.g. in SSVD for \(B^\top = A^\top Q\) the optimizer figures \(A\) and \(Q\) are identically partitioned in 0-th iteration, whereas in power iterations they are not, so it picks different physical operators in these cases.

3.3 Tracking the DRM row key values and their types

Optimizer automatically tracks non-integral row keys thoughout expression. E.g. dssvd's \(U\) rows are keys identically to rows of \(A\) even if they are non-integers; whereas rows of \(V\) are automatically granted with Int keys due to logical transformations of expression by the optimizer.
4 Notations

COMMENT

TENTATIVE or NOT YET IMPLEMENTED