

MapReduce SSVD Working Notes.

MapReduce QR decomposition

Dmitriy Lyubimov, 2010

December 14, 2010

Contents

1	Base algorithm	3
1.1	Modified SSVD method	3
1.2	QR decomposition.	3
1.3	Thin decomposition.	4
2	Reordering base QR for MapReduce	4
2.1	Blocking the input.	4
2.2	Initialization step.	4
2.3	Iterative step.	5
2.4	Combining §§ 2.2 2.3	6
2.5	Combining results of multiple mappers	6
3	Collecting Q	8
3.1	Q collection in <i>Thin Streaming QR</i>	9
3.2	Merging Qs	9
4	MapReduce considerations and optimizations	14
4.1	QR MapReduce optimizations	14
4.2	Packing triangular matrices	14
4.3	Growing and trimming Q block on demand	16
5	SSVD Map Reduce steps	16
5.1	Q-Job	16
5.1.1	Mapper	16
5.2	\mathbf{B}^\top -Job	18
5.2.1	Mapper	18
5.2.2	Combiner	19
5.2.3	Reducer	19
5.3	$\mathbf{B}\mathbf{B}^\top$ -Job	19
5.4	Eigen Decomposition	19
5.5	U-, V- Jobs	19

5.5.1	U-Job	20
5.5.2	V-Job	20
6	Issues and future work	20
6.1	“Supersplits” in MapReduce	20
6.2	Lack of preprocessing capabilities of matrix data is wasteful on RAM with wider matrices when packed with DistributeRowMa- trix format	21

1 Base algorithm

Notations. I consider the problem in context of stochastic SVD [Halko, et al]. Hence some notation and matrix names are inherited from that work. Since this work encounters a lot of matrix blocks, I also use extensively matrix block notation from [Golub, Van Loan], i.e. notation $\mathbf{A}(i:j, k:p)$ means 'block of \mathbf{A} , rows i to j and columns k to p '. Another example accepted there is such as $\mathbf{A}(i,:)$ (i -th row). More mainstream linear algebra notations are also used: single subscript such as \mathbf{A}_i usually means 'i-th vertical or horizontal block of \mathbf{A} '. Double subscript usually means single element of a matrix (e.g. $\mathbf{A}_{i,j}$) or a row ($\mathbf{A}_{i,*}$) or a column ($\mathbf{A}_{*,i}$).

1.1 Modified SSVD method

Modified SSVD Algorithm. Given an $m \times n$ matrix \mathbf{A} , a target rank k , and an oversampling parameter p , this procedure computes an $m \times (k+p)$ SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ (some notations are adjusted):

1. Create seed for random $n \times (k+p)$ matrix $\mathbf{\Omega}$. The seed defines matrix $\mathbf{\Omega}$ using Gaussian unit vectors per one of suggestions in [Halko, et al].
2. $\mathbf{Y} = \mathbf{A}\mathbf{\Omega}$, $\mathbf{Y} \in \mathbb{R}^{m \times (k+p)}$.
3. Column-orthonormalize $\mathbf{Y} \rightarrow \mathbf{Q}$ by computing thin decomposition $\mathbf{Y} = \mathbf{Q}\mathbf{R}$. Also, $\mathbf{Q} \in \mathbb{R}^{m \times (k+p)}$, $\mathbf{R} \in \mathbb{R}^{(k+p) \times (k+p)}$.
4. $\mathbf{B} = \mathbf{Q}^\top \mathbf{A}$: $\mathbf{B} \in \mathbb{R}^{(k+p) \times n}$.
5. Compute Eigensolution of a small Hermitian $\mathbf{B}\mathbf{B}^\top = \hat{\mathbf{U}}\mathbf{\Lambda}\hat{\mathbf{U}}^\top$. $\mathbf{B}\mathbf{B}^\top \in \mathbb{R}^{(k+p) \times (k+p)}$.
6. Singular values $\mathbf{\Sigma} = \mathbf{\Lambda}^{0.5}$, or, in other words, $s_i = \sqrt{\sigma_i}$.
7. If needed, compute $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.
8. If needed, compute $\mathbf{V} = \mathbf{B}^\top \hat{\mathbf{U}}\mathbf{\Sigma}^{-1}$. Another way is $\mathbf{V} = \mathbf{A}^\top \mathbf{U}\mathbf{\Sigma}^{-1}$.

1.2 QR decomposition.

$$\mathbf{Y} = \mathbf{Q}\mathbf{R},$$

where $\mathbf{Y} \in \mathbb{R}^{m \times n}$, $\mathbf{Q} \in \mathbb{R}^{m \times m}$, $\mathbf{R} \in \mathbb{R}^{m \times n}$.

We find our base algorithm in [Golub, Van Loan].

Base QR algorithm.

```

for  $j = 1 : n$ 

    for  $i = m - 1 : j - 1$ 
         $[c, s] = \text{givens}(\mathbf{Y}(i - 1, j), \mathbf{Y}(i, j))$ 
         $\mathbf{Y}(i - 1 : i, j : n) \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \mathbf{Y}(i - 1 : i, j : n)$ 
    end

end

```

1.3 Thin decomposition.

Thin decomposition is defined as

$$\mathbf{Y} = \mathbf{Q}_1 \mathbf{R}_1,$$

where $\mathbf{Q}_1 = \mathbf{Q}(1 : m, 1 : n)$, $\mathbf{R}_1 = \mathbf{R}(1 : n, 1 : n)$. Also, $\text{range}(\mathbf{Q}_1) = \text{range}(\mathbf{Y})$ [Golub, Van Loan]. Further on when i speak of QR decomposition, i will imply the thin decomposition (unless explicitly said otherwise).

2 Reordering base QR for MapReduce

2.1 Blocking the input.

We start off by dividing \mathbf{Y} into z blocks, such that

$$\mathbf{Y} = \begin{pmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \\ \vdots \\ \mathbf{Y}_z \end{pmatrix}.$$

Each $\mathbf{Y}_i = \mathbf{Y}(r(i - 1) : r \cdot i, :)$ and $\mathbf{Y}_i \in \mathbb{R}^{r \times n}$ thus corresponds to a mapper process.

2.2 Initialization step.

We start Givens iterations in \mathbf{Y}_i by traversing it from bottom to top. We allocate buffer $\tilde{\mathbf{R}} \in \mathbb{R}^{(k+p+1) \times (k+p)}$ and initially populate it with last $k+p+1$ rows of \mathbf{Y}_i : $\tilde{\mathbf{R}} \leftarrow \mathbf{Y}_i(r - k - p : r, :)$. We assume $k + p \leq r$. There's a special case when $k + p = r$ in which case last row of $\tilde{\mathbf{R}}$ remains all zeros. Then we apply base thin **QR** algorithm thus zeroing out all subdiagonal entries:

$$\tilde{\mathbf{R}} = \begin{pmatrix} \times & \times & \times & \vdots & \times & \times & \times \\ 0 & \times & \times & \vdots & \times & \times & \times \\ 0 & 0 & \times & \vdots & \times & \times & \times \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \vdots & \times & \times & \times \\ 0 & 0 & 0 & \vdots & 0 & \times & \times \\ 0 & 0 & 0 & \vdots & 0 & 0 & \times \\ 0 & 0 & 0 & \vdots & 0 & 0 & 0 \end{pmatrix} \quad (1)$$

2.3 Iterative step.

At each iterative step, we keep “shifting” $\tilde{\mathbf{R}}$ one row up on \mathbf{Y} (thus new rows are added on top of it):

for $j = r - k - p - 1 : 1$

$\tilde{\mathbf{R}}(2 : k + p + 1, :) \leftarrow \tilde{\mathbf{R}}(1 : k + p, :)$

$\tilde{\mathbf{R}}(1, :) \leftarrow \mathbf{Y}_i(j, :)$ # step A

for $l = 1 : k + p - 1$ # step B

$[c, s] \leftarrow \text{givens}(\tilde{\mathbf{R}}(l, l), \tilde{\mathbf{R}}(l + 1, l))$

$\tilde{\mathbf{R}}(l : l + 1, l : k + p) \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \tilde{\mathbf{R}}(l : l + 1, l : k + p)$

end

end

In step A above the $\tilde{\mathbf{R}}$ obtains non-zero values in the sub diagonal:

$$\tilde{\mathbf{R}} = \begin{pmatrix} \times & \times & \times & \vdots & \times & \times & \times \\ \beta & \times & \times & \vdots & \times & \times & \times \\ 0 & \beta & \times & \vdots & \times & \times & \times \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \vdots & \times & \times & \times \\ 0 & 0 & 0 & \vdots & \beta & \times & \times \\ 0 & 0 & 0 & \vdots & 0 & \beta & \times \\ 0 & 0 & 0 & \vdots & 0 & 0 & \beta \end{pmatrix} \quad (2)$$

Step B brings $\tilde{\mathbf{R}}$ back to upper-triangular-plus-one form (1).

2.4 Combining §§ 2.2 2.3

Algorithm “Thin streaming QR”. Given thin tall matrix $\mathbf{Y} \in \mathbb{R}^{m \times (k+p)}$, this algorithm computes \mathbf{R} in thin decomposition $\mathbf{Y} = \mathbf{Q}\mathbf{R}$ with memory requirements $O \approx (k+p) \times (k+p)$.

```

for  $j = r - 1 : 1$ 
     $\tilde{\mathbf{R}}(2 : k + p + 1, :) \leftarrow \tilde{\mathbf{R}}(1 : k + p, :)$ 
     $\tilde{\mathbf{R}}(1, :) \leftarrow \mathbf{Y}_i(j, :)$ 
    added = r-j
    for  $l = 1 : \max(\text{added} - 1, k + p)$ 
         $[c, s] \leftarrow \text{givens}(\tilde{\mathbf{R}}(l, l), \tilde{\mathbf{R}}(l + 1, l))$ 
         $\tilde{\mathbf{R}}(l : l + 1, l : k + p) \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \tilde{\mathbf{R}}(l : l + 1, l : k + p)$ 
    end
end
collect  $\mathbf{R}_i \leftarrow \tilde{\mathbf{R}}(1 : k + p, :)$ 

```

Note that outer loop can be a Mapper iterator feeding rows of matrix \mathbf{Y} into $\tilde{\mathbf{R}}$ in a streaming fashion. Memory bounds do not depend on m .¹

2.5 Combining results of multiple mappers

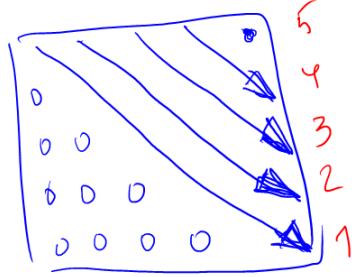
Next step is to define how to parallelize *thinGivensQR* algorithm. We split \mathbf{Y} into several blocks, and run additional Givens iterations pairwise on $\mathbf{R}_i, \mathbf{R}_j$ as:

Algorithm mergeR

¹In reality, mapper can't feed \mathbf{Y} rows in reverse. So i get my \mathbf{Q} rows in reverse. Which doesn't seem to be a problem if i account for it later.

$\text{mergeR}(R_1, R_2):$

R_2 traversal:



R_1 traversal

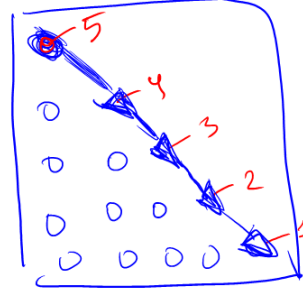


Figure 1: traversal order of matrices $\mathbf{R}_1, \mathbf{R}_2$ in algorithm mergeR .

```

R mergeR ( $\mathbf{R}_i, \mathbf{R}_j$ )
begin
  for  $v = 1 : k + p$ 
    for  $u = v : k + p$ 
       $[c, s] \leftarrow \text{givens}(\mathbf{R}_i(u, u), \mathbf{R}_j(u - v, u))$ 
       $\begin{pmatrix} \mathbf{R}_i(u, u : k + p) \\ \mathbf{R}_j(u - v, u : k + p) \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \begin{pmatrix} \mathbf{R}_i(u, u : k + p) \\ \mathbf{R}_j(u - v, u : k + p) \end{pmatrix}$ 
    end
  end
  return  $\mathbf{R}_i$ 
end

```

The traversal order for algorithm mergeR is outlined in fig. 1.
Thus final \mathbf{R} can be computed with the following:

Algorithm mergeAllRs

begin

for $i = 2 : z$

$\mathbf{R}_1 \leftarrow \text{merge}(\mathbf{R}_1, \mathbf{R}_i)$

end

return \mathbf{R}_1 as \mathbf{R} .

end

3 Collecting \mathbf{Q}

So far memory requirements to compute \mathbf{R} were very negligent: indeed, we only had to keep $\tilde{\mathbf{R}} \in \mathbb{R}^{(k+p+1) \times (k+p)}$ around. It is not so simple about collecting \mathbf{Q} . In the most common case, $\mathbf{Q} = \prod_i^N \mathbf{G}_i$, where $\mathbf{G}_i(i, j, \Theta)$ are Givens rotations accumulated in the order of the application. Since Givens operations are affecting only two columns of \mathbf{Q} , i and j , interesting facts emerge: a) since no mapper applies Givens to the same combination of (i, j) , it doesn't matter in which order to apply mapper results; b) memory requirements to collecting \mathbf{Q}_i in a mapper can be satisfied by $\mathbf{Q}_i \in \mathbb{R}^{r \times (k+p+1)}$ instead of $\mathbb{R}^{r \times r}$.

It is rather easy to modify “*Thin streaming QR*” to tug along partial Givens product for all Givens operations occurred in the given mapper which we denote as $\mathbf{Q}_i = \prod_j \mathbf{G}_{i,j}$ ². Problem arises when we approach collecting \mathbf{Q} during merge $(\mathbf{R}_i, \mathbf{R}_j)$. Indeed, merging \mathbf{Q}_1 and \mathbf{Q}_2 would result into interim accumulated \mathbf{Q} that densely requires $\mathbb{R}^{2r \times (k+p)}$. After l -th merge it grows to $\mathbb{R}^{lr \times (k+p)}$ and so on until ultimately we collect total thin $\mathbf{Q} \in \mathbb{R}^{m \times (k+p)}$. An observation is that if

$$\mathbf{Q} = \begin{pmatrix} \hat{\mathbf{Q}}_1 \\ \hat{\mathbf{Q}}_2 \\ \dots \\ \hat{\mathbf{Q}}_z \end{pmatrix}, \quad (3)$$

then intuition is that we can probably come up with some block algorithm computing $\hat{\mathbf{Q}}_i = \text{qMerge}(\mathbf{Q}_1, \mathbf{R}_1, \dots, \mathbf{Q}_z, \mathbf{R}_z)$, where $\{\mathbf{Q}_i, \mathbf{R}_i\}$ are results of running a *thinStreamingQR* over blocks \mathbf{Y}_i . If $\mathbf{Q}_1, \mathbf{R}_1, \dots, \mathbf{Q}_z, \mathbf{R}_z$ can be consumed in a streaming fashion, then memory requirement can be kept again in check the same as in mapper, since $\hat{\mathbf{Q}}_i \in \mathbb{R}^{r \times (k+p)}$. In addition, in context of stochastic SVD, collecting \mathbf{Q} column-wise is not a convenient outcome; since we'll have to do outer products with rows of \mathbf{A} during next pass, we'd rather collect it row-wise or in small horizontal blocks like in 3 above in order to avoid additional transpose.

²Here, $\mathbf{G}_{i,j}$ is denoting j^{th} Givens accumulated in mapper i

So all the said intuitively means that strategy in *mergeAllRs* is not really expected to work for the purposes of building final \mathbf{Q} .

3.1 \mathbf{Q} collection in *Thin Streaming QR*.

in order to collect thin \mathbf{Q}_i block, we'll define a rolling buffer similarly to $\tilde{\mathbf{R}}$:

$$\tilde{\mathbf{Q}} \in \mathbb{R}^{r \times (k+p+1)},$$

which would be collecting *columns* of \mathbf{Q}_i .

Algorithm “Thin streaming QR 1”:

for $j = r - 1 : 1$

$$\tilde{\mathbf{R}}(2 : k + p + 1, :) \leftarrow \tilde{\mathbf{R}}(1 : k + p, :)$$

$$\tilde{\mathbf{Q}}(2 : k + p + 1, :) \leftarrow \tilde{\mathbf{Q}}(1 : k + p, :)$$

$$\tilde{\mathbf{R}}(1, :) \leftarrow \mathbf{Y}_i(j, :)$$

added = r-j

for $l = 1 : \max(\text{added} - 1, k + p)$

$$[c, s] \leftarrow \text{givens}(\tilde{\mathbf{R}}(l, l), \tilde{\mathbf{R}}(l + 1, l))$$

$$\tilde{\mathbf{R}}(l : l + 1, l : k + p) \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \tilde{\mathbf{R}}(l : l + 1, l : k + p)$$

$$\tilde{\mathbf{Q}}(l : l + 1, :) \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \tilde{\mathbf{Q}}(l : l + 1, :)$$

end

end

$$\text{collect } \mathbf{R}_i \leftarrow \tilde{\mathbf{R}}(1 : k + p, :)$$

$$\text{collect } \mathbf{Q}_i^\top \leftarrow \tilde{\mathbf{Q}}(1 : k + p, :)$$

3.2 Merging Qs

So we got series of $(\mathbf{Q}_i, \mathbf{R}_i)$ pairs. I already demonstrated how to produce \mathbf{R} , but I am yet to demonstrate how to produce \mathbf{Q} in (3). We approach designing algorithm for $\hat{\mathbf{Q}}_i$ by induction. Let's assume, initially, we have only 2 blocks $(\mathbf{Q}_1, \mathbf{R}_1)$ and $(\mathbf{Q}_2, \mathbf{R}_2)$ which we now want to merge into final solution $\mathbf{Y} = \mathbf{QR}$. In general case, the number of rows in \mathbf{Q}_i coming from different mappers may not match. We will consider merge: $(\mathbf{Q}, \mathbf{R}) \leftarrow \text{QRmerge}(\mathbf{Q}_i, \mathbf{R}_i, \mathbf{Q}_j, \mathbf{R}_j)$, $\mathbf{Q}_i \in \mathbb{R}^{r_i \times (k+p)}$, $\mathbf{Q}_j \in \mathbb{R}^{r_j \times (k+p)}$, $\mathbf{Q}_m \in \mathbb{R}^{(r_i+r_j) \times (k+p)}$:

```

Algorithm QRmerge( $\mathbf{Q}_i, \mathbf{R}_i, \mathbf{Q}_j, \mathbf{R}_j$ )
begin
     $\tilde{\mathbf{Q}} \leftarrow \begin{pmatrix} \mathbf{Q}_i & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_j \end{pmatrix}$ 
    for  $v = 1 : k + p$ 
        for  $u = v : k + p$ 
             $[c, s] \leftarrow \text{givens}(\mathbf{R}_i(u, u), \mathbf{R}_j(u - v, u))$ 
             $\begin{pmatrix} \mathbf{R}_i(u, u : k + p) \\ \mathbf{R}_j(u - v, u : k + p) \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \begin{pmatrix} \mathbf{R}_i(u, u : k + p) \\ \mathbf{R}_j(u - v, u : k + p) \end{pmatrix}$ 
             $(\tilde{\mathbf{Q}}(:, u) \quad \tilde{\mathbf{Q}}(:, u - v + k + p)) \leftarrow$ 
                 $\leftarrow (\tilde{\mathbf{Q}}(:, u) \quad \tilde{\mathbf{Q}}(:, u - v + k + p)) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$ 
        end
    end
     $\mathbf{R} \leftarrow \mathbf{R}_i$ 
     $\mathbf{Q} \leftarrow \tilde{\mathbf{Q}}(:, 1 : (k + p))$ 
    return  $(\mathbf{Q}, \mathbf{R})$ 
end

```

The result of applying this algorithm to our trivial case of $(\mathbf{Q}_1, \mathbf{R}_1, \mathbf{Q}_2, \mathbf{R}_2)$ is final QR decomposition since it is equivalent to a Givens QR with a modified order of Givens operations.

Still working on the trivial case of induction: the algorithm *QRMerge* is still not good enough though because merged matrix is now approximately two times taller (not mentioning the quadrupled memory requirement to store temporary buffer $\tilde{\mathbf{Q}}$). All benefits of blocking in terms of memory requirements are thus gone. The next observation is that we actually can run the procedure 2 times and output \mathbf{Q} in 2 blocks. It will work since Givens operations are affecting two values in the same row independently of other rows. So the next step is to split algorithm into several subroutines:

Algorithm R mergeRonQ ($\mathbf{R}_i, \mathbf{R}_j, \mathbf{Q}_m$)

```

begin
  for  $v = 1 : k + p$ 
    for  $u = v : k + p$ 
       $[c, s] \leftarrow \text{givens}(\mathbf{R}_i(u, u), \mathbf{R}_j(u - v, u))$ 
       $\begin{pmatrix} \mathbf{R}_i(u, u : k + p) \\ \mathbf{R}_j(u - v, u : k + p) \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^\top \begin{pmatrix} \mathbf{R}_i(u, u : k + p) \\ \mathbf{R}_j(u - v, u : k + p) \end{pmatrix}$ 
       $(\mathbf{Q}_m(:, u) \quad \mathbf{Q}_m(:, u - v + k + p)) \leftarrow$ 
         $\leftarrow (\mathbf{Q}_m(:, u) \quad \mathbf{Q}_m(:, u - v + k + p)) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$ 
    end
  end
  return  $\mathbf{R}_i$ 
end

```

Algorithm (\mathbf{Q}, \mathbf{R}) mergeQrUp($\mathbf{Q}_d, \mathbf{R}_d, \mathbf{R}_s$):

```

begin
   $\tilde{\mathbf{Q}} \leftarrow (\mathbf{Q}_d \quad \mathbf{0})$ 
   $\mathbf{R} \leftarrow \text{mergeRonQ}(\mathbf{R}_d, \mathbf{R}_s, \tilde{\mathbf{Q}})$ 
   $\mathbf{Q} \leftarrow \tilde{\mathbf{Q}}(:, 1 : k + p)$ 
  return  $(\mathbf{Q}, \mathbf{R})$ 
end

```

Algorithm (\mathbf{Q}, \mathbf{R}) mergeQrDown($\mathbf{R}_d, \mathbf{Q}_s, \mathbf{R}_s$):

```

begin
   $\tilde{\mathbf{Q}} \leftarrow (\mathbf{0} \quad \mathbf{Q}_s)$ 
   $\mathbf{R} \leftarrow \text{mergeRonQ}(\mathbf{R}_d, \mathbf{R}_s, \tilde{\mathbf{Q}})$ 
   $\mathbf{Q} \leftarrow \tilde{\mathbf{Q}}(:, 1 : k + p)$ 
  return  $(\mathbf{Q}, \mathbf{R})$ 
end

```

So, in the trivial case of 2 blocks ($z = 2$), with respect to (3) we could execute

$$\hat{\mathbf{Q}}_1 = \text{mergeQrUp}(\mathbf{Q}_1, \mathbf{R}_1, \mathbf{Q}_2) \cdot \mathbf{Q}$$

$$\hat{\mathbf{Q}}_2 = \text{mergeQrDown}(\mathbf{R}_1, \mathbf{Q}_2, \mathbf{R}_2) \cdot \mathbf{Q}$$

which is equivalent to

$$\begin{pmatrix} \hat{\mathbf{Q}}_1 \\ \hat{\mathbf{Q}}_2 \end{pmatrix} = \text{mergeQR}(\mathbf{Q}_1, \mathbf{R}_1, \mathbf{Q}_2, \mathbf{R}_2)$$

above, which is in its turn a full Givens method solution for \mathbf{Q} .

Moving on. For number of Q-blocks $z = 3$ this will look like

$$\begin{aligned} \hat{\mathbf{Q}}_1 &= \text{mergeQrUp}(\text{mergeQrUp}(\mathbf{Q}_1, \mathbf{R}_1, \mathbf{R}_2), \mathbf{R}_3) \cdot \mathbf{Q}; \\ \hat{\mathbf{Q}}_2 &= \text{mergeQrUp}(\text{mergeQrDown}(\mathbf{R}_1, \mathbf{Q}_2, \mathbf{R}_2), \mathbf{R}_3) \cdot \mathbf{Q}; \\ \hat{\mathbf{Q}}_3 &= \text{mergeQrDown}(\text{mergeR}(\mathbf{R}_1, \mathbf{R}_2), \mathbf{Q}_3, \mathbf{R}_3) \cdot \mathbf{Q}. \end{aligned}$$

The pattern emerging from this is that in order to process Q blocks (3), one would need a sequential access to all \mathbf{R}_i (*always in the same sorted order which allows for great optimization under MR assumptions and which would require a rather small I/O requirement of $\sim \frac{z(k+p)^2}{2}$*), and only current Q block as initial value. By induction we infer our algorithm for $\hat{\mathbf{Q}}_i$ in (3) as:

Algorithm $\hat{\mathbf{Q}}_i$ computeQHatBlock (\mathbf{Q}_i , iterator ($\mathbf{R}_1, \dots, \mathbf{R}_z$)).
This algorithm computes a $\hat{\mathbf{Q}}_i$ block in (3), given \mathbf{Q}_i and complete series $\{\mathbf{R}_i\}$ from qr computations of blocks \mathbf{Y}_i :

```

begin
  let  $\tilde{\mathbf{R}} \leftarrow \text{iterator.fetch}(\mathbf{R}_1)$ 
  for  $j = 2 : (i - 1)$ 
     $\tilde{\mathbf{R}} \leftarrow \text{mergeR}(\tilde{\mathbf{R}}, \text{iterator.fetch}(\mathbf{R}_j))$ 
  end
  if  $(i > 1)$ 
     $(\mathbf{Q}_i, \tilde{\mathbf{R}}) \leftarrow \text{mergeQrDown}(\tilde{\mathbf{R}}, \mathbf{Q}_i, \text{iterator.fetch}(\mathbf{R}_i))$ 
  endif
  for  $j \leftarrow i + 1 : z$ 
     $(\mathbf{Q}_i, \tilde{\mathbf{R}}) \leftarrow \text{mergeQrUp}(\mathbf{Q}_i, \tilde{\mathbf{R}}, \text{iterator.fetch}(\mathbf{R}_j))$ 
  end
  return  $\mathbf{Q}_i$  as  $\hat{\mathbf{Q}}_i$ 
end

```

There is yet another observation. If we run *computeQBlock* in succession for an ordered sequence of Q blocks then we obviously keep running *mergeR* for

same subsequences of \mathbf{R} again and again. A little improvement on that would be reducing number of Rs in the R sequence as we go:

Algorithm computeQHatSequence.

This algorithm computes all $\{\hat{\mathbf{Q}}_i\}$ and final \mathbf{R} in (3) given individual series $\{\mathbf{Q}_i\}, \{\mathbf{R}_i\}$ from block-wise qr computations of blocks \mathbf{Y}_i :

computeQHatSequence($\{\mathbf{Q}_i\}, \{\mathbf{R}_i\}$):

begin

$s \leftarrow 1$

for $i=1:z$

$\mathbf{Q}_s \leftarrow \mathbf{Q}_i$

$\hat{\mathbf{Q}}_i \leftarrow \text{computeQHatBlock}(\mathbf{Q}_s, \text{iterator}(\{\mathbf{R}_i\}))$

if ($s==2$)

$\mathbf{R}_1 \leftarrow \text{mergeR}(\mathbf{R}_1, \mathbf{R}_2)$

$\{\mathbf{R}_i\} \leftarrow \{\mathbf{R}_i\} \setminus \{\mathbf{R}_2\}$

else $s++$

end

end

What's more, at the end $\{\mathbf{R}_i\} \equiv \{\mathbf{R}\} \square$.³

Thus, considering a potential for MapReduce implementation, we may find we need to re-distribute an individual copy of $\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_z$ to each process running series of QR merges. So sequence $\{\mathbf{R}_i\}$ may act as a side map info. Good thing is that this memory is only bound by number of Q blocks (that may quite high) and number of oversampled singular values $k+p$. Further memory scaling is aided by the fact that, as is demonstrated further, we can use hierarchical approach to computation of $\{\hat{\mathbf{Q}}_i\}$, thus essentially unbounding algorithm for memory for either z or m (algorithm is already theoretically unbounded for n in terms of memory requirements).

This is not to say that memory is completely not essential to the algorithm: there's a component in flops estimate, not terribly significant, that would depend on the number of blocks z . After all, computation path of every $\hat{\mathbf{Q}}_i$ has to go thru merging all Rs one way or another. So the higher \mathbf{Q}_i is, the smaller z . Flops function is not strictly linear to z (it's more like $\propto \frac{1}{2} \log z$) but it would help to have less blocks, given this algorithm is CPU bound as it is.

³We still go over Rs twice as really necessary since preceding call to *computeQHat* already merges the R internally... but it's gotta be good enough for now.

4 MapReduce considerations and optimizations

4.1 QR MapReduce optimizations

- Since source blocks for merging \mathbf{QR} decomposition are \mathbf{QR} decompositions themselves, $\hat{\mathbf{Q}}$ computation can be made hierarchically-recursive. E.g. each mapper can compute small blocks, then combiner would produce a larger block, and the larger blocks can again be input to the next level computation of $\hat{\mathbf{Q}}$. The only extension needed for existing *computeQHat* algorithm is that we need to invoke it multiple times for multiple subchunks of Q-block emanating from previous hierarchical merge, instead of combining them into one big Q. *TODO: add a formal algorithmic description of this process.* Essentially what this achieves is reducing number of R matrices to number of mappers which makes it more feasible to fit in memory at the next stage of the hierarchical merge (not to mention moving some significant computational load from reducers to combiners. Short version, if $\mathbf{Q}_{i,j}$ is a j -th block produced in i -th mapper, then final block $\hat{\mathbf{Q}}_{i,j} = \text{computeQBlock}_i \left(\text{computeQBlock}_j (\mathbf{Q}_{i,j}, \{\mathbf{R}_{i*}\}), \{\mathbf{R}_*\} \right)$ (here I omit details of merging intermediate Rs). The inner algorithm is run in combiner and outer algorithm is run in reducer/next step mappers. Quick illustration of the hierarchy is in fig. 2.
- It doesn't look like we have anything to gain from shuffle-and-sort. Makes sense to run 2 mapper-only jobs rather than one MR. Indeed, Rs are already coming in the order we need, out of map processing, and so are the Q blocks. We wouldn't even need to use combiners had we not wanted to revisit all Q blocks we just produced with the full R order available. The downside is that it is going to produce a lot of small partition files (one per mapper). But that's all temporary data. Also, in context of SSVD computation, the next mapper can proceed and finish with computation of \mathbf{B}^\top , so running it in the next mapper actually shortens overall execution time by skipping the need for one shuffle-sort-reduce parts (which are usually the most voluminous parts of any MR process).

4.2 Packing triangular matrices

We are packing an (upper) triangular matrix $\mathbf{R} \in \mathbb{R}^n$ into a vector $\boldsymbol{\alpha} \in \mathbb{R}^r$ and the following defines the transformations

$$r = n(n+1)/2$$

$$n = \frac{-1 + \sqrt{1 + 8 \cdot r}}{2}$$

Coordinates mapping: if $\forall i, j, i \in [0, n-1], j \in [0, n-1] \rightarrow \exists l \in [0, r-1] : \mathbf{R}_{i,j} = \boldsymbol{\alpha}_l$; and if $\forall l \exists i, j : \boldsymbol{\alpha}_l = \mathbf{R}_{i,j}$ then we denote such index transformation by $l = f(i, j)$. The solution for $f(\cdot)$ I used is:



2nd pass is using $\text{ComputeQHatBlock}()$
 Next level of aggregation is using
 $\text{ComputeQHatBlock} |_{\bar{i} = \text{Mapper\#}}$

So we can represent evolution of Q block
 as: ($\bar{i} = \text{Mapper\#}$, $j = \text{block\# in a mapper}$)

$$\hat{Q}_{i,j}^T = \underbrace{\text{ComputeQBlock}}_{\text{reducer or next step mapper}} \left(\underbrace{\text{ComputeQBlock}}_{\text{combiner}} (Q_{i,j}) \right)$$

Figure 2: illustration of hierarchical recursive application of computeQHatBlock routines.

$$l = f(i, j) = \frac{n + (n - i + 1)}{2}i + (j - i) = \frac{(2n - i + 1)i}{2} + (j - i)$$

and then I defined that

$$\mathbf{R}_{i,j} = \boldsymbol{\alpha}_{f(i,j)}$$

(assuming all indices start with 0, like they do in java).

4.3 Growing and trimming Q block on demand

When we reach end of an input corresponding to Q block computation, we need to make sure mapper still has at least another $k + p$ inputs for the next last block. If it turns out it doesn't, we need to add the remainder rows to the current block. That means that we may need to be able to grow $\tilde{\mathbf{Q}}$ buffer by $\Delta < k + p$ on the top to increase number of rows in the output Q block (Fig. 3).

Similarly, the solver can 'trim' the block if computation ends early (between $k + p$ and r rows).

5 SSVD Map Reduce steps

5.1 Q-Job

This is a map-only job. Shuffle and sort is unnecessary (well, it does have a reducer configured, although it doesn't send anything to it, since otherwise combiners would not fire). Hence this job is more scalable than a regular MapReduce job that involves shuffle-and-sort.

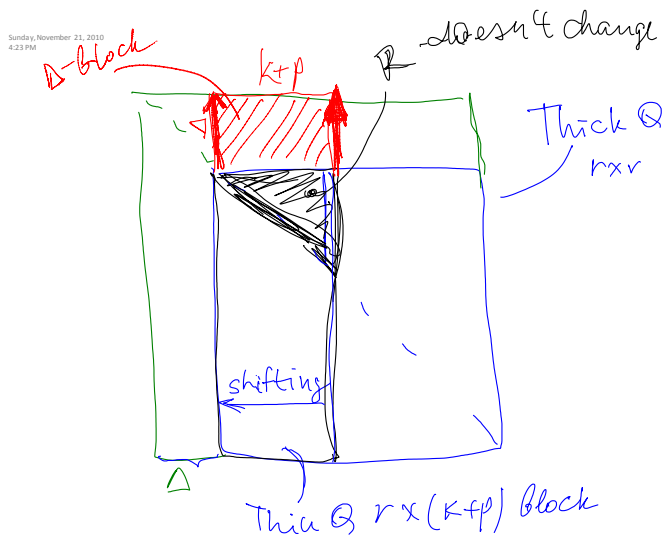
5.1.1 Mapper

The mappers compute Y rows as

$$(\mathbf{Y}_{i,*})^\top = (\mathbf{A}_{i,*})^\top \boldsymbol{\Omega}.$$

Each mapper then feeds Y rows into *ThinStreamingQR1* algorithm until block size of r is reached. Thus blocks of \mathbf{A} and \mathbf{Y} are only conceptual but are never formed in memory.

After r rows of \mathbf{Y} are consumed by algorithm, \mathbf{Q} and \mathbf{R} blocks are collected. In this version mapper does a second pass over \mathbf{Q} data, so \mathbf{Q} block is flushed to a local temp file but \mathbf{R} sequence is accumulated in the memory. for case of $k + p = 500$, \mathbf{R} matrix would occupy approximately 1Mb, uncompressed. Thus, optimal use of RAM is achieved by tweaking r parameter (block height) which



$$\Delta < k+p$$

We can grow Q_i^T on the left if we need to by $\Delta < (k+p)$.

Just use Arrays, copyOf + arrayCopy
Similarly, we can stop early by shrinking Q on the left (and be done),
As soon as we read r rows, we need to make sure that there's another $(k+p)$ available. If there isn't then we grow Q by Δ (remainder) and proceed with Q-flipping.

Figure 3: growing Q block

affects both length of $\{\mathbf{R}\}$ sharing memory with single \mathbf{Q} block at the peak RAM load.⁴

Once all \mathbf{Q} blocks are flushed into a local file, the stored \mathbf{Q} sequence is considered again in the same order. Processing of $\{\mathbf{Q}_i\}$, $\{\mathbf{R}_i\}$ is done per *ComputeQHatSequence* algorithm and HFDS *MultipleOutputs* are used to store one \mathbf{R}_i and $\{\hat{\mathbf{Q}}_{i,j}^\top\}$ where i is the mapper’s task id, as the mapper’s output. This is a first level of \mathbf{R} merging illustrated in fig. 2.

5.2 \mathbf{B}^\top -Job

This job completes \mathbf{Q} merge hierarchy in mappers, and produces two outputs: final \mathbf{Q} and \mathbf{B}^\top .⁵

5.2.1 Mapper

The mapper runs a variation of *ComputeQHatSequence* algorithm at the next level of hierarchy to produce final blocks $\{\hat{\mathbf{Q}}_i^\top\}$ and \mathbf{R} as has already been referred to fig. (2) previously. After $\hat{\mathbf{Q}}_i^\top$ fragment is computed, it is transposed and output row-wise into \mathbf{Q} side file⁶. Also, it is used to compute outer products per

$$\mathbf{B}^\top = \sum_i^m \mathbf{Q}_{i,*} (\mathbf{A}_{i,*})^\top. \quad (4)$$

The main input is \mathbf{A} with exactly the same split as was used in §5.1.1⁷. The second input is \mathbf{Q} -Job file corresponding to the same task id. This ensures that \mathbf{Q} rows are produced along with corresponding \mathbf{A} rows.

⁴A nuance here is that \mathbf{QR} can only be compiled if there are at least $k + p$ rows of \mathbf{Y} , so in order to ensure that, a lookahead buffer for \mathbf{Y} rows is maintained. If end of stream is encountered, the lookahead buffer is just added to current *ThinStreamingQR1* computation, thus ensuring the last block will have at least $k + p$ rows in it. If n becomes large enough that Hadoop splits do not include even $k + p$ rows of \mathbf{A} , then *minSplitSize* parameter should be used. Current Hadoop’s *FileInputFormat* implementation ensures that the last split is at least $1.1 \times \text{minSplitSize}$ large.

⁵Since all matrices are produced as row-wise sequence files, notation \mathbf{B}^\top and such just imply that matrix \mathbf{B} is produced column-wise.

⁶Since \mathbf{Y} rows are fed into *ThinStreamingQR1* in direct order, instead of inverse, the columns of $\hat{\mathbf{Q}}^\top$ are produced in inverse order. Hence, we need to flip $\hat{\mathbf{Q}}$ row-wise after transposition before outputting it and being able to use it to form outer products $\mathbf{Q}_{i,*} (\mathbf{A}_{i,*})^\top$. Also, $\hat{\mathbf{Q}}$ rows inherit labels from corresponding \mathbf{A} rows so we can use them later to label \mathbf{U} rows.

⁷This hugely relies the fact that \mathbf{A} is split exactly the same way in \mathbf{Q} -job and \mathbf{B}^\top -job and tasks with the same task ids in these jobs receive exactly the same input rows of \mathbf{A} , which at present seems to hold. I’d prefer to have a patch that would actually serialize splits produced for \mathbf{A} in the \mathbf{Q} -job and re-uses them for \mathbf{B}^\top -job to avoid unexpected surprises. But in the current implementation, it just relies on Hadoop to do this idempotently. I haven’t tested situation when \mathbf{A} might be presented by multiple files, which often might be the case if \mathbf{A} is a result of a MR job.

5.2.2 Combiner

5.2.3 Reducer

Reducer and combiner are exactly the same and doing regular reduction of outer products as in any matrix multiplication, to finalize partial sums in (4).

5.3 \mathbf{BB}^\top -Job⁸

This job (which is a MapReduce job) trivially produces

$$\mathbf{BB}^\top = \sum_i \mathbf{B}_{*,i} (\mathbf{B}_{*,i})^\top. \quad (5)$$

5.4 Eigen Decomposition

Eigen decomposition is run in frontend:

$$\mathbf{BB}^\top = \hat{\mathbf{U}} \mathbf{\Lambda} \hat{\mathbf{U}}^\top,$$

$$\mathbf{BB}^\top \in \mathbb{R}^{(k+p) \times (k+p)}.$$

Singular values are computed as $s_i = \sqrt{\sigma_i}$, where σ_i is i -th eigenvalue (from i -th row on the main diagonal of $\mathbf{\Lambda}$).

5.5 U-, V- Jobs

These jobs are optional (called when output is requested) and are trivial matrix multiplication jobs per formulations given in §(1.1). If both computations are requested, they are spawned as parallel multiplication jobs (similar to “parallel query” optimization used in Pig). The jobs use outputs of §5.2. \mathbf{U} rows receive correspondent labels from \mathbf{A} . \mathbf{V} rows just receive ordinals $[0, n)$. Only ints are supported as labels as of the time of this writing (which means \mathbf{A} can only have 4 billion uniquely labeled rows. This limitation is perhaps a *TODO*). Also, since thin SVD rank is meant to be k (not $k + p$), I use only parts of $\hat{\mathbf{U}}$ and $\mathbf{\Sigma}^{-1}$ as described below.

⁸It was brought to my attention that doing \mathbf{BB}^\top computation + subsequent Eigen may not necessarily be as numerically stable as doing stock SVD on the entire \mathbf{B} . However, in context of this algorithm i did not actually find that being the case so much. In a unit test with $m = 100,000$ and $n = 100$ (dense data) doing $\mathbf{BB}^\top = \hat{\mathbf{U}} \mathbf{\Lambda} \hat{\mathbf{U}}$ did not produce numerical errors in singular values when compared to stock SVD solution using error threshold $\varepsilon = 10^{-10}$. But errors produced for lack of oversampling ($k + p < n$) in stochastic projection produced much more noticable errors. So even if there’s a deficiency in computing $\mathbf{BB}^\top = \hat{\mathbf{U}} \mathbf{\Lambda} \hat{\mathbf{U}}$, it is dwarfed by errors caused by stochastic projection by several orders of magnitude. Therefore, there’s no practical deficiency in that approach but the benefit of dealing with much smaller $\mathbf{BB}^\top \in \mathbb{R}^{(k+p) \times (k+p)}$ in the front-end computation compared to $\mathbf{B} \in \mathbb{R}^{(k+p) \times n}$ is just very hard to ignore.

5.5.1 U-Job

U-Job is a mapReduce job that works row-wise thru

$$\mathbf{U}_{i,*} = (\mathbf{Q}_{i,*})^\top \hat{\mathbf{U}}(:, 1:k).$$

Hence, U-job is a map-only job.

5.5.2 V-Job

Similarly to U-Job, V-Job is a map-only mapReduce job that works row-wise via

$$\mathbf{V}_{i,*} = (\mathbf{B}_{i,*}^\top)^\top \hat{\mathbf{U}}(:, 1:k) \boldsymbol{\Sigma}^{-1}(1:k, 1:k).$$

6 Issues and future work

6.1 “Supersplits” in MapReduce

Row wise format poses fundamental limitation on block formation (vertical blocking only), thus putting forward a requirement for huge MR split sizes far exceeding what is deemed common and practical hdfs block sizes. Indeed, by my estimates, with $k + p = 500$, and assuming hdfs block size of 128Mb, expectation for mapper data downloads with this current approach starts to exceed 50% of the split size at about ~30,000-50,000 non-zero elements in rows of \mathbf{A} . With $n = 300,000$ for a dense data we can expect up to 90% of \mathbf{A} data to be moved around just be loaded in mappers (assuming a large machine cluster). With larger n these numbers only increase. It may not be a big problem for moderate size clusters and moderate size inputs, but for huge clusters (thousand nodes) it may prove to be a challenge (imo). I dont have first hand experience of working with hardware of this scale, so it’s hard for me to put forward any further estimates.

In terms of solution, intriguing possibility that might solve supersplit problem completely is blockwise input format.

Within the limitations of row-wise format and absent of blockwise format, the next best thing may be to devise a strategy of semi-automatic detection of the situations when $k + p$ blocks of \mathbf{A} start exceeding HDFS block size. Like it is shown above, it is actually expected to happen fairly quickly with growth either n or number of requested singular values $k + p$. When this happens, a slightly modified flow may kick in. In that flow, we preprocess \mathbf{A} in the mapper row-by-row and generate \mathbf{Y} rows and then send them to reducer for the first pass of QR hierarchical processing to happen there. We’d still be moving around a lot of data thru shuffle and sort, but it’s much easier to move \mathbf{Y} data around than \mathbf{A} data, since \mathbf{Y} is only $k + p$ wide as opposed to \mathbf{A} which may span millions in width. If we consider \mathbf{A} in a row by row fashion, then supersplit problem would not realistically cause more than 50% of \mathbf{A} data move up to a couple of million of non-zero elements in rows of \mathbf{A} . Another time matrix \mathbf{A} comes into

contact with our data is computation of \mathbf{B}^\top . At this point, a finer organization of Q data may be required to find starting point for map-side join with A data (HFile or MapFile come to mind as possible venues).⁹

6.2 Lack of preprocessing capabilities of matrix data is wasteful on RAM with wider matrices when packed with DistributeRowMatrix format

This algorithm may consider matrix \mathbf{A} element by element. However, as of the time of this writing implementation of *VectorWritable* doesn't allow sequential consumption of matrix elements without preloading the entire vector in memory. In case 1 million non-zero elements in incoming vector, the prebuffering RAM space reaches 8Mb or more, for 100 million vector it would require 1G just to get access to single next element of matrix A. Considering the fact that this SSVD implementation is theoretically memory-unbound for n (width) of \mathbf{A} , this situation strikes me as being severely off-balance between effort going into SSVD algorithm efficiency and serialization framework.

However one doesn't have to operate with 1Gb long vectors to start feeling the pain. Consider situation (which was confirmed to be somewhat typical with recommenders) where occasionally a 100mb long vector for an extremely popular item may occur. With default mapper setting in hadoop of -Xmx200m that means 50% of mapper RAM has to be dedicated to prebuffering occurring in *VectorWritable*, with the rest being available to the algorithm. What's worse, we have to do that allocation even though long vectors are occurring only 0.1% or so, so that memory is wasted while algorithm running time suffers. That may become quite a showstopper in some computations dealing with unevenly sparse data whereas it doesn't have to be.

In terms of solution, I believe this deficiency can be relatively easily rectified though so it should be a low-hanging fruit for furthering memory efficacy effort in SSVD solution.

References

- [Golub, Van Loan] Golub, Van Loan. Matrix computations, 3rd edition.
- [Halko, et al] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions

⁹Another possible solution is to have a separate cluster solely dedicated for algorithms working on "supersplits" with 1G or something block size, but i would be walking out of the door the next minute i proposed something like that to our VP. So i don't consider this really a solution in my case.