MapReduce SSVD Working Notes.
MapReduce QR decomposition

Dmitriy Lyubimov, 2010

November 25, 2011

Contents

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 $\mathbf{A}\left(i:j,k:p\right)$ means 'block of $\mathbf{A}$, rows i to j and columns k to p'. Another example accepted there is such as $\mathbf{A}\left(i,:\right)$ (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 $i = m - 1 : j - 1$

    $[c, s] = \text{givens}\left(\mathbf{Y}\left(i-1, j\right), \mathbf{Y}\left(i, j\right)\right)$

    $\mathbf{Y}\left(i-1 : i, j : n\right) \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^{\top} \mathbf{Y}\left(i-1 : i, j : n\right)$

  end

end

1.3 Thin decomposition.

Thin decomposition is defined as

$$\mathbf{Y} = \mathbf{Q_1}\mathbf{R_1},$$

where $\mathbf{Q_1} = \mathbf{Q}(1 : m, 1 : n)$, $\mathbf{R}_1 = \mathbf{R}\left(1 : n, 1 : n\right)$. Also, $\text{range}\left(\mathbf{Q_1}\right) = \text{range}\left(\mathbf{Y}\right)$ [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 $\mathbf{Y}_i = \mathbf{Y}\left(r\left(i-1\right) : r \cdot i, :\right)$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:

$$\tilde{\mathbf{R}} = \begin{pmatrix} \times & \times & \times & \vdots & \times & \times & \times \\ 0 & \times & \times & \vdots & \times & \times & \times \\ 0 & 0 & \times & \vdots & \times & \times & \times \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \vdots & \times & \times & \times \\ 0 & 0 & 0 & \vdots & 0 & \times & \times \\ 0 & 0 & 0 & \vdots & 0 & 0 & \times \\ 0 & 0 & 0 & \vdots & 0 & 0 & 0 \end{pmatrix} \tag{1}$$

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

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

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

end

end

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

$$\tilde{\mathbf{R}} = \begin{pmatrix} \times & \times & \times & \vdots & \times & \times & \times \\ \beta & \times & \times & \vdots & \times & \times & \times \\ 0 & \beta & \times & \vdots & \times & \times & \times \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \vdots & \times & \times & \times \\ 0 & 0 & 0 & \vdots & \beta & \times & \times \\ 0 & 0 & 0 & \vdots & 0 & \beta & \times \\ 0 & 0 & 0 & \vdots & 0 & 0 & \beta \end{pmatrix} \tag{2}$$

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$\mathbf{Y} = \mathbf{QR}$ 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 $l = 1 : \max(added - 1, k+p)$

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

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

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]

[1] In 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.

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

$\mathbf{R}$ mergeR $(\mathbf{R}_i, \mathbf{R}_j)$

begin

for $v = 1 : k + p$

for $u = v : k + p$

$[c, s] \leftarrow \text{givens}\left(\mathbf{R}_i\left(u, u\right), \mathbf{R}_j\left(u - v, u\right)\right)$

$$\begin{pmatrix} \mathbf{R}_i\left(u, u : k + p\right) \\ \mathbf{R}_j\left(u - v, u : k + p\right) \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^{\top} \begin{pmatrix} \mathbf{R}_i\left(u, u : k + p\right) \\ \mathbf{R}_j\left(u - v, u : k + p\right) \end{pmatrix}$$

end

end

return $\mathbf{R}_i$

end

The traversal order for algorithm *mergeR* is outlined in fig. 1.

Thus final R can be computed with the following:

Algorithm mergeAllRs

begin

for $i = 2 : z$

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

end

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

end

3 Collecting Q

So far memory requirements to compute 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

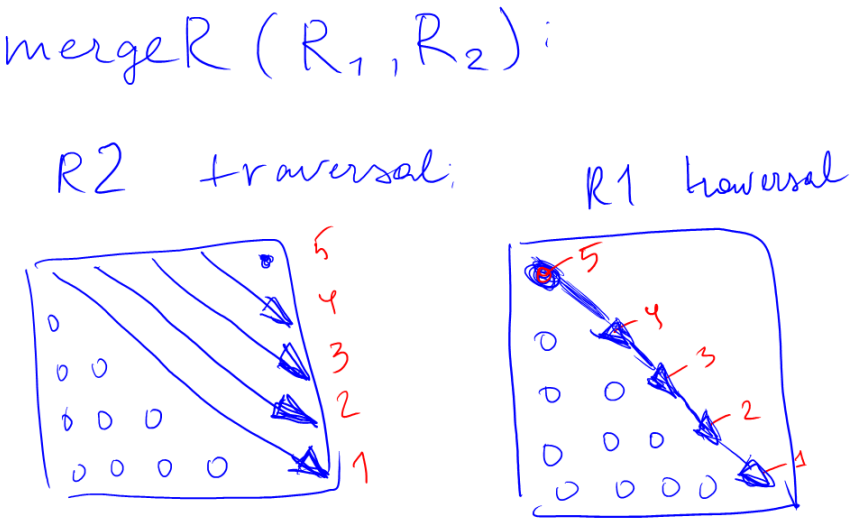

Figure 1: traversal order of matrices $\mathbf{R}_1, \mathbf{R}_2$ in algorithm *mergeR*.

$\mathbf{Q}$. 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

$$\mathbf{Q} = \begin{pmatrix} \hat{\mathbf{Q}}_1 \\ \hat{\mathbf{Q}}_2 \\ \cdots \\ \hat{\mathbf{Q}}_z \end{pmatrix}, \tag{3}$$

then intuition is that we can probably come up with some block algorithm computing $\hat{\mathbf{Q}}_i = \text{qMerge}\,(\mathbf{Q}_1, \mathbf{R}_1, ..., \mathbf{Q}_z, \mathbf{R}_z)$, where $\{\mathbf{Q}_i, \mathbf{R}_i\}$ are results of running a *thinStreamingQR* over blocks $\mathbf{Y}_i$. If $\mathbf{Q}_1, \mathbf{R}_1, ..., \mathbf{Q}_z, \mathbf{R}_z$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.

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 Q collection in *Thin Streaming QR.*

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

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

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

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 $l = 1:\max(added-1, k+p)$

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

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

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

end

end

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

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 $u = v : k + p$

$[c, s] \leftarrow \text{givens}\left(\mathbf{R}_i\left(u, u\right), \mathbf{R}_j\left(u - v, u\right)\right)$

$\begin{pmatrix} \mathbf{R}_i\left(u, u : k + p\right) \\ \mathbf{R}_j\left(u - v, u : k + p\right) \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^{\top} \begin{pmatrix} \mathbf{R}_i\left(u, u : k + p\right) \\ \mathbf{R}_j\left(u - v, u : k + p\right) \end{pmatrix}$

$\left(\tilde{\mathbf{Q}}\left(:, u\right) \quad \tilde{\mathbf{Q}}\left(: u - v + k + p\right)\right) \leftarrow$

$\leftarrow \left(\tilde{\mathbf{Q}}\left(:, u\right) \quad \tilde{\mathbf{Q}}\left(:, u - v + k + p\right)\right) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$

end

end

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

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

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 $u = v : k + p$

$[c, s] \leftarrow \text{givens}\left(\mathbf{R}_i\left(u, u\right), \mathbf{R}_j\left(u - v, u\right)\right)$

$$\begin{pmatrix} \mathbf{R}_i\left(u, u : k + p\right) \\ \mathbf{R}_j\left(u - v, u : k + p\right) \end{pmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^{\top} \begin{pmatrix} \mathbf{R}_i\left(u, u : k + p\right) \\ \mathbf{R}_j\left(u - v, u : k + p\right) \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{Q}_m\left(:, u\right) & \mathbf{Q}_m\left(: u - v + k + p\right) \end{pmatrix} \leftarrow$$

$$\leftarrow \begin{pmatrix} \mathbf{Q}_m\left(:, u\right) & \mathbf{Q}_m\left(:, u - v + k + p\right) \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

end

end

return $\mathbf{R}_i$

end

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

begin

$\tilde{\mathbf{Q}} \leftarrow \begin{pmatrix} \mathbf{Q}_d & \mathbf{0} \end{pmatrix}$

$\mathbf{R} \leftarrow \text{mergeRonQ}\left(\mathbf{R}_d, \mathbf{R}_s, \tilde{\mathbf{Q}}\right)$

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

return $(\mathbf{Q}, \mathbf{R})$

end

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

begin

$\tilde{\mathbf{Q}} \leftarrow \begin{pmatrix} \mathbf{0} & \mathbf{Q}_s \end{pmatrix}$

$\mathbf{R} \leftarrow \text{mergeRonQ}\left(\mathbf{R}_d, \mathbf{R}_s, \tilde{\mathbf{Q}}\right)$

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

return $(\mathbf{Q}, \mathbf{R})$

end

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

$$\hat{\mathbf{Q}}_1 = \text{mergeQrUp}\left(\mathbf{Q}_1, \mathbf{R}_1, \mathbf{Q}_2\right).\mathbf{Q}$$
$$\hat{\mathbf{Q}}_2 = \text{mergeQrDown}\left(\mathbf{R}_1, \mathbf{Q}_2, \mathbf{R}_2\right).\mathbf{Q}$$

which is equivalent to

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

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

$$\hat{\mathbf{Q}}_1 = \text{mergeQrUp}\left(\text{mergeQrUp}\left(\mathbf{Q}_1, \mathbf{R}_1, \mathbf{R}_2\right), \mathbf{R}_3\right).\mathbf{Q};$$
$$\hat{\mathbf{Q}}_2 = \text{mergeQrUp}\left(\text{mergeQrDown}\left(\mathbf{R}_1, \mathbf{Q}_2, \mathbf{R}_2\right), \mathbf{R}_3\right).\mathbf{Q};$$
$$\hat{\mathbf{Q}}_3 = \text{mergeQrDown}\left(\text{mergeR}\left(\mathbf{R}_1, \mathbf{R}_2\right), \mathbf{Q}_3, \mathbf{R}_3\right).\mathbf{Q}.$$

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

end

There is yet another observation. If we run *computeQ̂Block* 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 $\left\{\hat{\mathbf{Q}}_i\right\}$ 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

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

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

if (s==2)

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

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

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 $\mathbf{R}_1, \mathbf{R}_2, ..., \mathbf{R}_z$ 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 $\left\{\hat{\mathbf{Q}}_i\right\}$, 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 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.

[3] We 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.

3.3 Hierarchical QR

ComputeQHatSequence algorithm is a one-pass algorithm which requires preloading the entire $\{\mathbf{R}_i\}$ sequence into memory. Even though matrices $\mathbf{R}_i$ are not directly bound to m (number of input rows), they are still bound by the same process memory space as the height of Q-block r. This means that at some point, for a given amount of RAM in a process, a maximum for length of $\{\mathbf{R}_i\}$ is going to be reached and upper bound for m can be presented by a function $m = f(r)$ where $f(r)$ is a function with a single global maximum. Bottom line, m is still RAM-bound at this point, even maximum might be quite high for practical purposes.

There's another observation that allows to unbind m for RAM with additional passes over Q data.

We can employ bottom-up r-indegree divide-and-conquer aproach to evolve Q thru multiple sequences of R as follows:

$$\mathbf{Y} = \begin{pmatrix} \cdots \\ \mathbf{Y}_i \\ \mathbf{Y}_{i+1} \\ \mathbf{Y}_{i+2} \\ \cdots \end{pmatrix} = \begin{matrix} \cdots \\ \begin{bmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{Q} & \mathbf{R} \\ \cdots & \cdots \\ \mathbf{Q} & \mathbf{R} \end{bmatrix} \\ \begin{bmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{Q} & \mathbf{R} \\ \cdots & \cdots \\ \mathbf{Q} & \mathbf{R} \end{bmatrix} \\ \cdots \end{matrix} \Rightarrow \begin{matrix} \cdots \\ \begin{bmatrix} \mathbf{Q} \\ \mathbf{Q} \\ \cdots \\ \mathbf{Q} \end{bmatrix} \\ \begin{bmatrix} \mathbf{Q} \\ \mathbf{Q} \\ \cdots \\ \mathbf{Q} \end{bmatrix} \\ \cdots \end{matrix} \begin{matrix} \cdots \\ \mathbf{R} \\ \mathbf{R} \\ \cdots \end{matrix} \Rightarrow \cdots \Rightarrow \begin{matrix} \cdots \\ \hat{\mathbf{Q}} \\ \hat{\mathbf{Q}} \\ \cdots \\ \hat{\mathbf{Q}} \\ \cdots \end{matrix} \hat{\mathbf{R}}$$

Indeed, since source blocks for merging **QR** decomposition are **QR** decompositions themselves, $\hat{\mathbf{Q}}$ computation can be made hierarchically-recursive.

We start at the bottom of tree and merge smaller **QR** sequences into longer ones. Each pass is rougly a map-only MapReduce job. Nodes in the tree have rather high outdegree (~1000). Thus a tree corresponding to 2 passes would have depth tree and can merge up more than a million leaves, which each leaf

being a rather high Q block. If Q block can be at least 1000 rows high (and our memory resources can easily allow much taller Q blocks) then we can scale current 2-pass approach to over than a billion rows.

Suppose initially we have same input as for the *computeQHatSequence* algorithm, i.e. two sequences $\{\mathbf{Q}_i\}$, $\{\mathbf{R}_i\}$. The next step is to split them further into subsequences $\left\{\{(\mathbf{Q}_j, \mathbf{R}_j)\}_i\right\}$. For simplicity, let's denote members of new subsequences with a double subscript notation such as $\mathbf{Q}_{j,i}$ in order to denote j-th q-block in i-th group. How we group them into new subsequences is not important, what important is that $\forall i: \mathbf{Q}_i \equiv \mathbf{Q}_{j,s} \rightarrow \mathbf{R}_i \equiv \mathbf{R}_{j,s}$ and the length of each subsequence group is small enough so that we can preload the entire subsequence $\{\mathbf{R}_j\}_i$ into memory of a single process. Let's assume the split algorithm is available as a subroutine that posseses the aforementioned properties.

The next step is to apply *computeQHatSequence* algorithm to each of the subsequences. In this algorithm, we only will have to preload R subsequences for individual split only. The output of this hence will have only one $\hat{\mathbf{R}}$ and the same number of q-blocks which we can denote as $\left\{\left\{\hat{\mathbf{Q}}_j\right\}_i\right\}, \left\{\hat{\mathbf{R}}_i\right\}$. Each subsequence $\left\{\left\{\hat{\mathbf{Q}}_j\right\}_i\right\}$ then can be regarded as $\{\mathbf{Q}_i\}$ and the split-and-merge steps repeat until there's only one $\hat{\mathbf{R}}$ left (i.e., more formally, until $\left\|\left\{\hat{\mathbf{R}}_i\right\}\right\| = 1$).

Now we can try to produce a formal description of hierarchical QR.

First, we need a slightly modified version of *ComputeQHatSequence* algorithm that allows to work with groups of q-blocks:

Algorithm computeQHatSequence2.
This algorithm computes all $\left\{\hat{\mathbf{Q}}_i\right\}$ and final $\mathbf{R}$ in (3) given individual series $\left\{\{\mathbf{Q}_j\}_i\right\}$, $\{\mathbf{R}_i\}$ of intermediate QR products

$\left(\left\{\hat{\mathbf{Q}}_j\right\}, \mathbf{R}\right)$**computeQHatSequence2**$\left(\left\{\{\mathbf{Q}_j\}_i\right\}, \{\mathbf{R}_i\}\right)$**:**

begin

$u \leftarrow 1, v \leftarrow 1$
$z = \|\{\mathbf{R}_s\}\|$
for $l = 1 : z$
for $k = 1 : \left\|\{\mathbf{Q}_j\}_l\right\|$
begin
$\hat{\mathbf{Q}}_v \leftarrow \text{computeQHatBlock}\left(\mathbf{Q}_{k,l}, \text{iterator}\left(\{\mathbf{R}_i\}\right)\right)$
$v \leftarrow v + 1$
end
if (u==2)
then
$\mathbf{R}_1 \leftarrow \text{mergeR}\left(\mathbf{R}_1, \mathbf{R}_2\right)$
$\{\mathbf{R}_i\} \leftarrow \{\mathbf{R}_i\} \setminus \{\mathbf{R}_2\}$
else
$u \leftarrow u + 1$
end
$\mathbf{R} \leftarrow \mathbf{R}_1$
return $\left(\left\{\hat{\mathbf{Q}}\right\}, \mathbf{R}\right)$

end

Next, we can define a pseudo code for hierarchical QR merge process. Note that this is pseudocode that is logical only and never really implemented sequentially as presented:

Algorithm HierarchicalQR. This algorithm compiles full reordered Givens QR of initial blocked input per (3).

$\left\{\hat{\mathbf{Q}}_i\right\}, \mathbf{R}$ **HierarchicalQR** $\left(\left\{\{\mathbf{Q}_j\}_i\right\}, \{\mathbf{R}_i\}, \text{maxRseqLen}\right)$

begin

if $\|\{\mathbf{R}_i\}\| > \text{maxRseqLen}$

then

groups = split input into maxRSeqLen groups; *// this will set up MR pass*[a]

input $\leftarrow \emptyset$

foreach group in **groups**

input =input $\cup$ HierarchicalQR(group)

end

else

input=$\left(\left\{\{\mathbf{Q}_i\}_j\right\}, \{\mathbf{R}_i\}\right)$

end

return computeQHatSequence2(input, maxRseqLen)

end

[a] actually we only need to split R sequences between passes.

The idea behind multiple passes is that we keep splitting R sequences into groups such that group of Rs is sufficiently small to be loaded into a map process, and run *computeQHatSequence2* on each group in parallel so that each group will produce a single R. Thus, new R-sequence will have significantly smaller number of Rs. Realistically, if we can hope to fit 1000 $k+p \times k+p$ triangular matrices in memory, then with each pass we can reduce length of $\{\mathbf{R}_i\}$ 1000 times. With current implementation illustrated in fig. (2) we have only 2 passes in which we can reasonably expect to be able to compute 1,000,000 Rs (and also Q blocks) assuming some reasonable block height r. Suppose under these arrangements the height of the Q block can be at least $1,000 \times (k+p \approx 500) = 4$Mb then we can orthogonalize 1 billion rows of input matrix $\mathbf{Y}$. But of course expectation is that r can be quite more than 1,000 to maximize m, in which case we can easily overshoot 1 bln rows scale for some combination of reasonable RAM and r.

4 MapReduce considerations and optimizations

4.1 QR MapReduce optimizations

- Since source blocks for merging **QR** decomposition are **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 $\hat{\mathbf{Q}}$. 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 $\left\{\hat{\mathbf{Q}}_{i,j}\right\} = \text{computeQ̂Block}|_i \left(\text{computeQ̂Block}|_j \left(\mathbf{Q}_{i,j,} \left\{\mathbf{R}_{i*}\right\}\right), \left\{\mathbf{R}_{*}\right\}\right)$ (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. More formal definition of this process for any number of hierarchical passes was given in § 3.3.
- 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

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

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

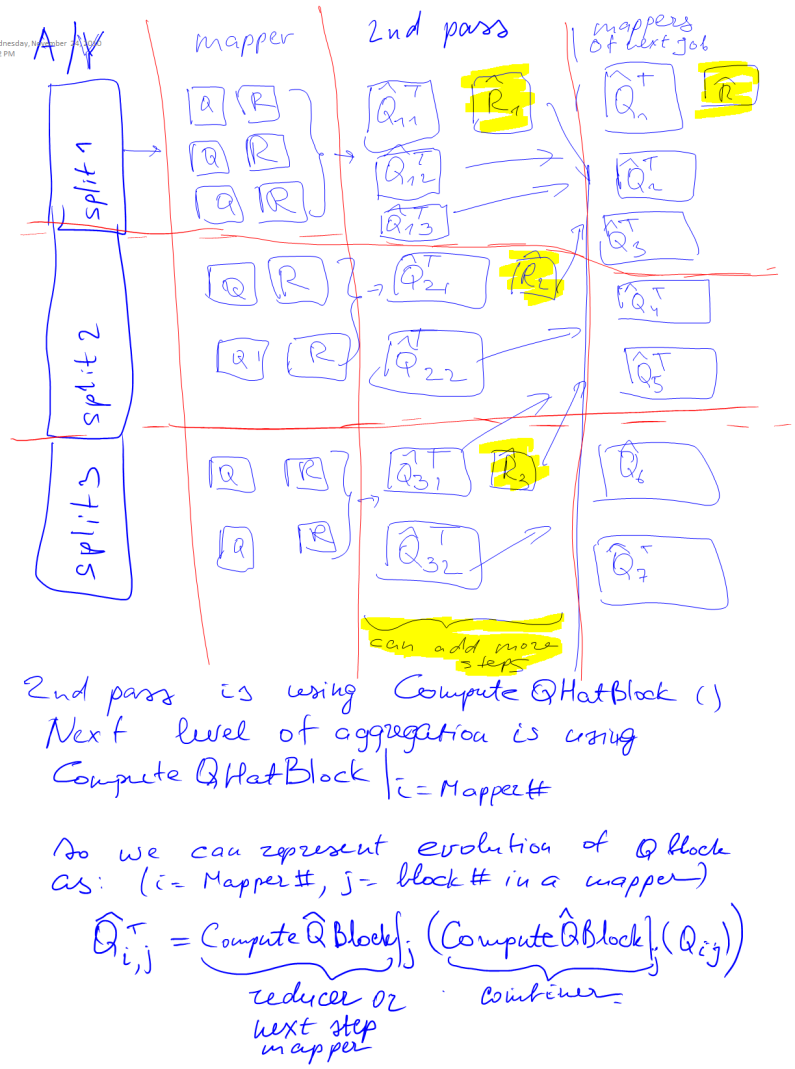

Figure 2: illustration of hierarchical recursive application of *computeQHatSequence* routines.

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 $\forall\, l\, \exists\, i, j : \rightarrow \boldsymbol{\alpha}_l = \mathbf{R}_{i,j}$ then we denote such index transformation by $l = f(i, j)$. The solution for $f(\cdot)$ I used is:

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

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

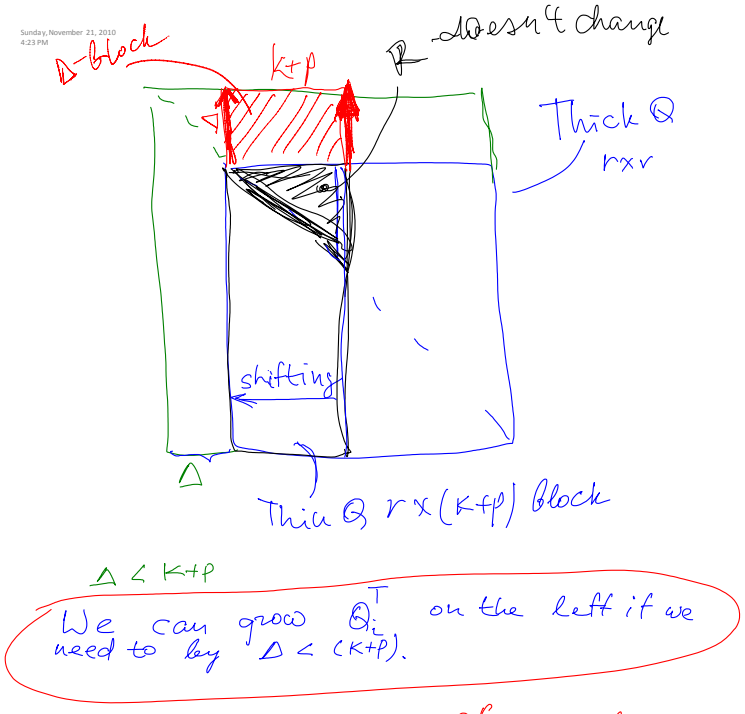

Figure 3: growing Q block

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

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 (unless there's exactly one $\mathbf{Q}$ block has been computed[4]) 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 affects both length of $\{\mathbf{R}\}$ sharing memory with single $\mathbf{Q}$ block at the peak RAM load. [5]

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

5.2 $\mathbf{B}^\top$-Job

This job completes Q merge hierarchy in mappers, and produces two outputs: final $\mathbf{Q}$ and $\mathbf{B}^\top$.[6] .

[4] In reality, if we don't increase split sizes using –minsplitsize, then we always can load 1 hdfs block worth of data in memory. But since we are forming Q blocks which presumably require even less memory than blocks of $\mathbf{A}$, then in reality our Q blocks should always be able to fit into memory for the entire mapper input and the case of exactly 1 Q block formed in a mapper pass should be the norm. Which, it it's turn, means there's no need for a second pass in the mapper mentioned here. Indeed, optimization is such that no temporary side Q file is created in mappers unless we encounter more than 1 Q block in a mapper pass. Bottom line, the second pass mentioned here is largely expected to be rather an exception than a rule and is necessary mostly with unusually high `--minsplitsize` parameter. But by the time this unusally large split happens, we should be able to switch to more ad hoc efficient reducer-side first phase QR which will be available in the *ssvd-wide* branch as an alternative execution flow. Which means, in theory, we should never have a need for second passes here.

[5] A nuance here is that 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 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 $1.1 \times minSplitSize$ large.

[6] Since 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.

5.2.1 Mapper

The mapper runs a variation of *ComputeQHatSequence* algorithm at the next level of hierarchy to produce final blocks $\left\{\hat{\mathbf{Q}}_i^\top\right\}$ and $\mathbf{R}$ as has already been refered to fig. (2) previously. After $\hat{\mathbf{Q}}_i^\top$ fragment is computed, it is transposed and output row-wise into Q side file[7]. Also, it is used to compute outer products per

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

The main input is $\mathbf{A}$ with exactly the same split as was used in §5.1.1[8]. The second input is Q-Job file corresponding to the same task id. This ensures that Q rows are produced along with corresponding A rows.

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).

Reducer now also produces upper triangular sums of $\mathbf{BB}^\top$. Since $\mathbf{BB}^\top$ is symmetrical, it is sufficient to sum up just an upper triangular part of all the outer products. After a reducer finishes summing up a row of $\mathbf{B}^\top$ matrix $\mathbf{b}_i$, outer product $\mathbf{b}_i\mathbf{b}_i^\top$ is added to accumulator $\left[\mathbf{BB}^\top\right]_l$:

$$\left[\mathbf{BB}^\top\right]_l \leftarrow \left[\mathbf{BB}^\top\right]_l + \mathbf{B}_{i,*}{\mathbf{B}_{i,*}}^\top.$$

At the end of the reducing, L partial sums are created, one per reducer.

[7]Since $\mathbf{Y}$ 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 outputing it and being able to use it to form outer products $\mathbf{Q}_{i,*}\left(\mathbf{A}_{i,*}\right)^\top$. Also, $\hat{\mathbf{Q}}$ rows inherit labels from correpsonding $\mathbf{A}$ rows so we can use them later to label $\mathbf{U}$ rows.

[8]This hugely relies the fact that $\mathbf{A}$ is split exactly the same way in 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 **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 A might be presented by multiple files, which often might be the case if A is a result of a MR job.

5.3 Front-end step

5.3.1 Finishing summing up $\mathbf{BB}^\top$

This takes individual partial sums which are $(k+p)\times(k+p)$ upper-triangulars, and sums them up

$$\mathbf{BB}^\top = \sum_{l=1}^{L} \left[\mathbf{BB}^\top\right]_l$$

Since input matrices are tiny, running it on frontend even on output of couple of thousands of reducers should not be a performance bottleneck. Perhaps we could improve this a little by parallelizing the I/O part but at this point it is just done sequentially with no noticeable impact.

5.3.2 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 $\mathbf{\Lambda}$).

5.4 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.

5.4.1 U-Job

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

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

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

Alternatively, solver may compute a shared similarity space output for U as

$$\mathbf{U}_\sigma = \mathbf{U}\mathbf{\Sigma}^{0.5}$$

if requested by a command line option. Here, notation $\mathbf{U}_\sigma$ implies, for lack of a better one, that U columns are scaled using singular values to a similarity space shared by both $\mathbf{U}_\sigma$ and $\mathbf{V}_\sigma$. Notation $\mathbf{\Sigma}^{0.5}$ implies

$$\mathbf{\Sigma}^{0.5} = \begin{pmatrix} \sqrt{\sigma_1} & 0 & \cdots & 0 \\ 0 & \sqrt{\sigma_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sqrt{\sigma_k} \end{pmatrix}.$$

5.4.2 V-Job

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

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

Alternatively, solver may compute a shared similarity space output for $\mathbf{V}$ as

$$\mathbf{V}_\sigma = \mathbf{V}\mathbf{\Sigma}^{0.5}$$

if requested by a command line option.

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 A. With $n = 300,000$ for a dense data we can expect up to 90% of 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 possiblity 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 A in the mapper row-by-row and generate 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 Y data around than A data, since Y is only $k+p$ wide as opposed to A which may span millions in width. If we consider A in a row by row fashion, then supersplit problem would not realistically cause more than 50% of A data move up to a couple of million of non-zero elements in rows of A. Another time matrix 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]

[9] "ssvd-wide" git branch is slated to address this solution in ssvd-lsi github repo.

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 **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 **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 occuring 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.

Another opportunity to improve is that *VectorWritable* to a significant degree goes against philosophy of the *Writable* use case. Indeed, during read operation Hadoop mappers create one same *Writable* instance and keep reusing it for every record as a preallocated buffer so that batch doesn't generate stream of one-time references that then directly go to garbage colleciton, thus creating signficant garbage collection overhead and, in situations of low memory, even a phenomenon known as GC thrashing. Opportunity to do the same is thus presented to *Writable.readFields()*. However, *VectorWritable* actually doesn't follow that pattern and as of the time of this writing allocates a new Vector for *every new record*. Even if GC thrashing does not really occur in a particular case, GC still occupies significantly more CPU ticks than it actually has to, had *Writable* philosophy been actually followed. [10]

In terms of solution, I beleive this deficiency can be relatively easily rectified though so it should be a low-hanging fruit for furthering memory efficacy effort in SSVD solution.[11]

[10] *TODO: get mapper running time comparison when running with and without vector preprocessing.*

[11] This is now alpha-ready in branch "ssvd-preprocessing" of my github account as of this writing, see https://github.com/dlyubimov/ssvd-lsi. Q-Job is now forming Y rows without

6.3 Additional passes over QR data would allow to unbind m from per-process RAM

As described in § 3.3, each additional Map-only pass over Q data would increase bound for m approx. 1000 times or more. Such additional pass might be implemented rather easily by shuffling some code around and adding some more properties for tuning this up.[12]

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

forming A row vector in memory, and $\mathbf{B}^\top$-job is forming both partial $\mathbf{B}^\top$ sums and final $\mathbf{Q}$ rows without forming A row, so at any given time it is pretty much just single Q block in memory (+ one partial product consisting of $k+p$ double precision values).

[12] "ssvd-tall" branch is slated for this in ssvd-lsi repo.