Column Subset Selection on Terabyte-sized Scientific Data

Michael W. Mahoney

ICSI and Department of Statistics, UC Berkeley

(Joint work with Alex Gittens and many others.)

December 2015

Overview

Comments on scientific data and choosing good columns as features.

Linear Algebra in Spark

- CX and SVD/PCA implementations and performance

Two scientific applications of Spark

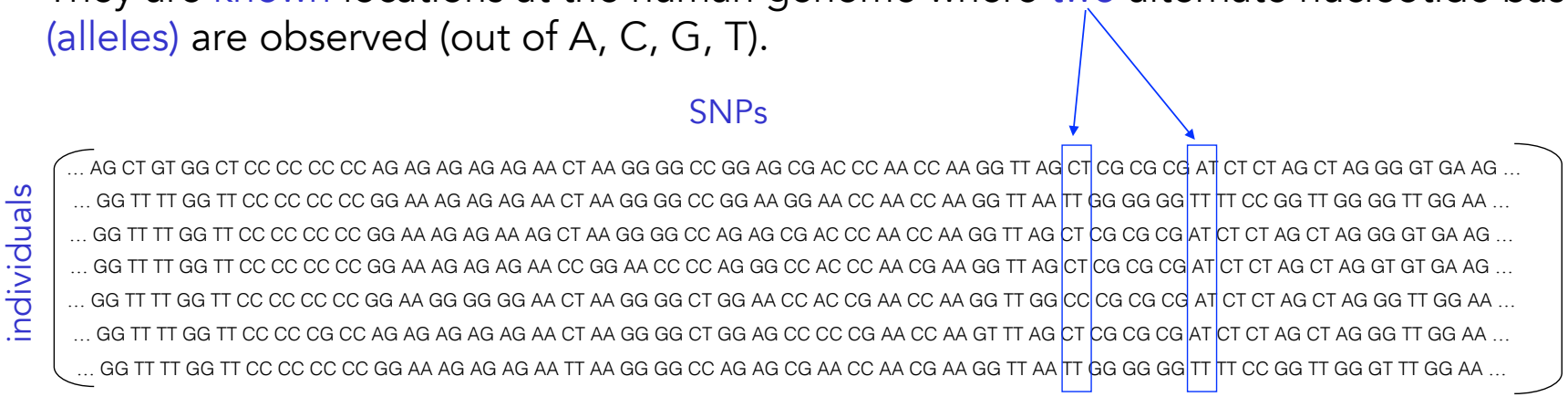
- Applications of the CX and PCA matrix decompositions
- To mass spec imaging, climate science (etc.)

Scientific data and choosing good columns as features

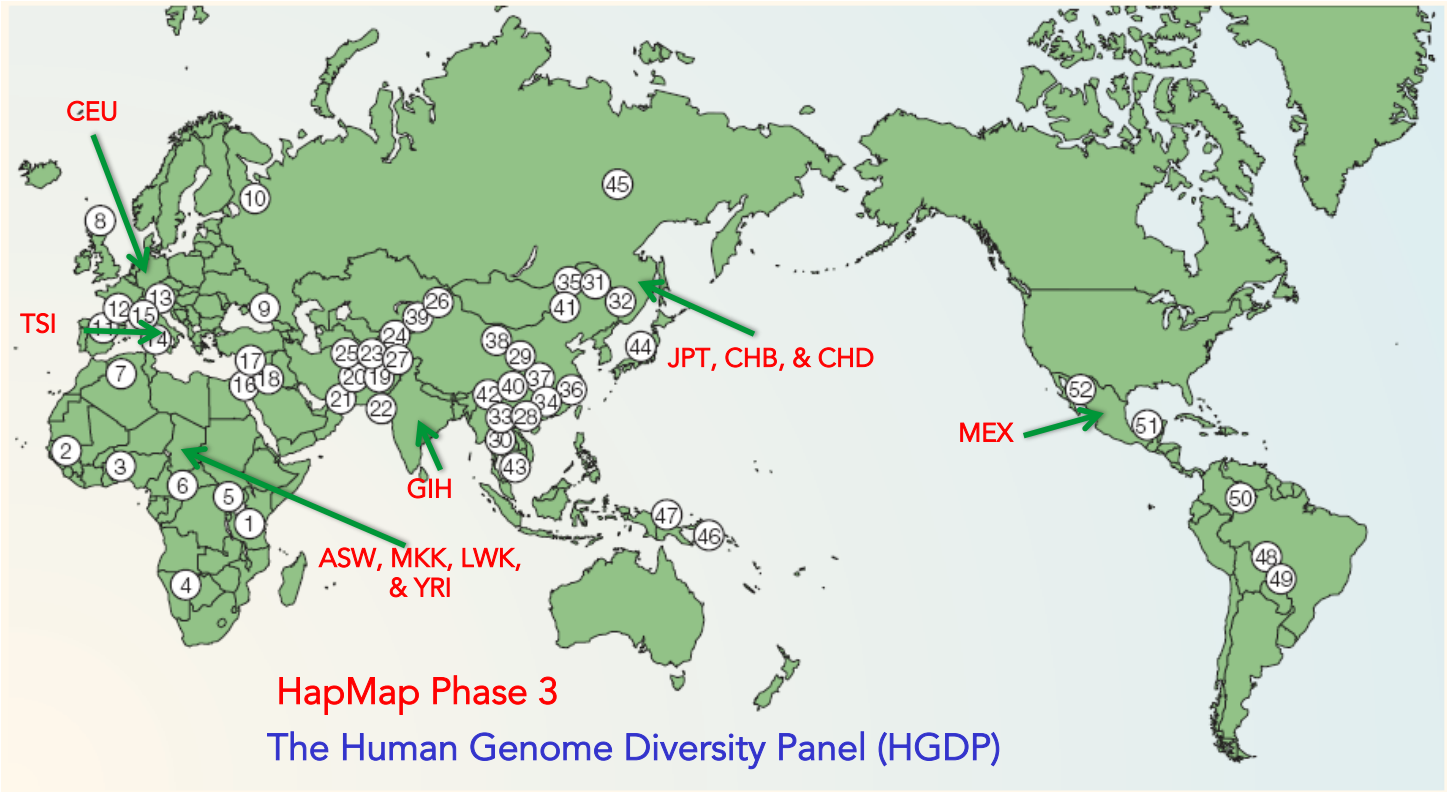
E.g., application in: Human Genetics

Single Nucleotide Polymorphisms: the most common type of genetic variation in the genome across different individuals.

They are known locations at the human genome where two alternate nucleotide bases (alleles) are observed (out of A, C, G, T).



Matrices including thousands of individuals and hundreds of thousands (large for some people, small for other people) if SNPs are available.



Africans
1 Bantu
2 Mandenka
3 Yoruba
4 San
5 Mbuti pygmy
6 Biaka
7 Mozabite

Europeans
8 Orcadian
9 Adygei
10 Russian
11 Basque
12 French
13 North Italian
14 Sardinian
15 Tuscan

Western Asians
16 Bedouin
17 Druze
18 Palestinian

Central and Southern Asians
19 Balochi
20 Brahui
21 Makrani
22 Sindhi
23 Pathan
24 Burusho
25 Hazara
26 Uygur
27 Kalash

Eastern Asians
28 Han (S. China)
29 Han (N. China)
30 Dai
31 Daur
32 Hezhen
33 Lahu
34 Miao
35 Oroqen
36 She
37 Tujia
38 Tu
39 Xibo
40 Yi
41 Mongola
42 Naxi
43 Cambodian
44 Japanese
45 Yakut

Oceanians
46 Melanesian
47 Papuan

Native Americans
48 Karitiana
49 Surui
50 Colombian
51 Maya
52 Pima

Cavalli-Sforza (2005) *Nat Genet Rev*

Rosenberg et al. (2002) *Science*

Li et al. (2008) *Science*

The International HapMap Consortium (2003, 2005, 2007) *Nature*

HGDP data

- 1,033 samples
- 7 geographic regions
- 52 populations

HapMap Phase 3 data

- 1,207 samples
- 11 populations

Apply SVD/PCA on the (joint) HGDP and HapMap Phase 3 data.

Matrix dimensions:

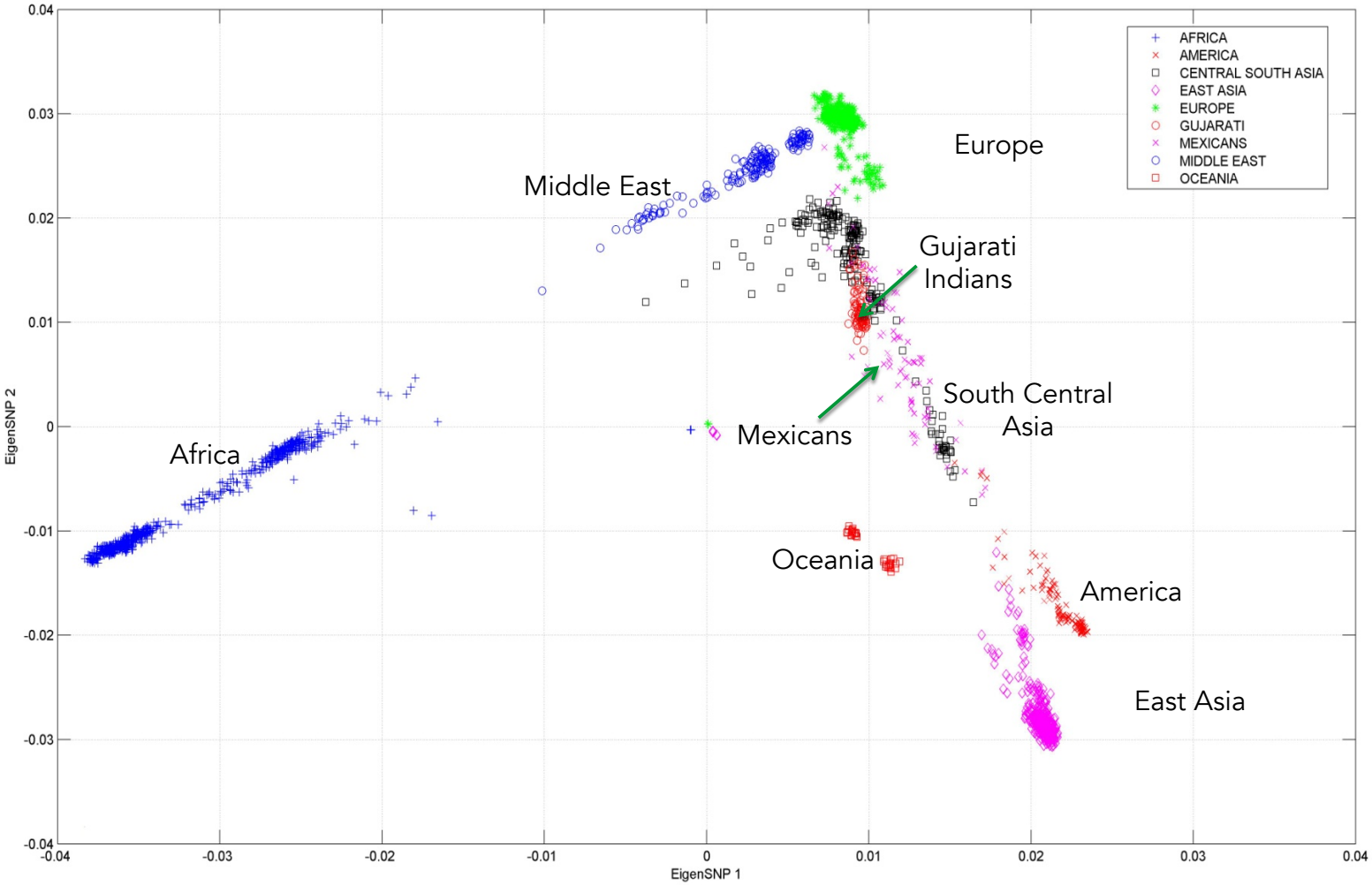
2,240 subjects (rows)

447,143 SNPs (columns)

Dense matrix:

over one billion entries

Paschou, et al (2010) J Med Genet



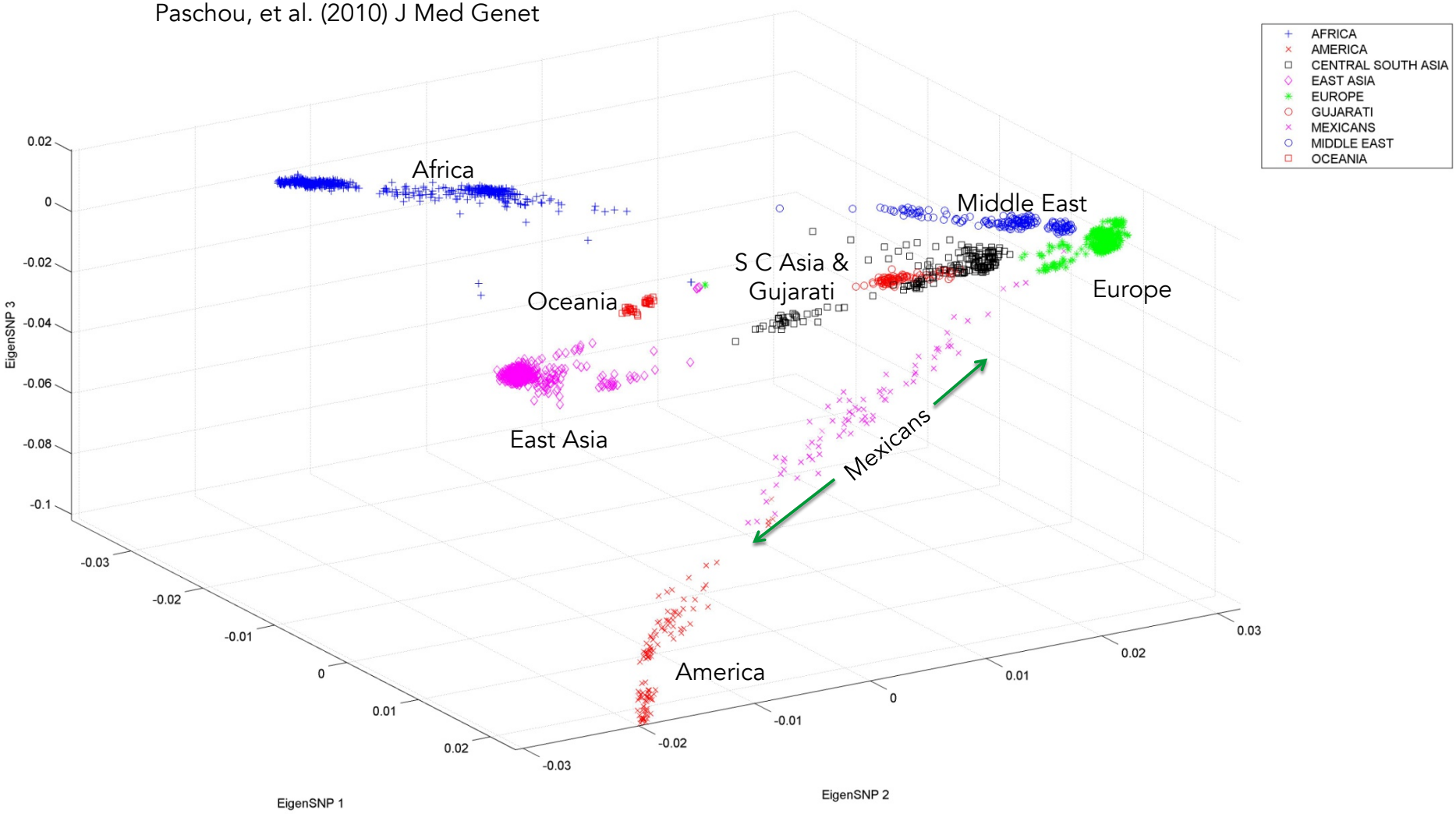
• Top two Principal Components (PCs or eigenSNPs)

(Lin and Altman (2005) *Am J Hum Genet*)

• The figure renders visual support to the "out-of-Africa" hypothesis.

• Mexican population seems out of place: we move to the top three PCs.

Paschou, et al. (2010) J Med Genet



- Not altogether satisfactory: the principal components are linear combinations of all SNPs, and – of course – can not be assayed!
- Can we find actual SNPs that capture the information in the singular vectors?
- Relatedly, can we compute them and/or the truncated SVD "efficiently."

Two related issues with eigen-analysis

Computing large SVDs: computational time

- In commodity hardware (e.g., a 4GB RAM, dual-core laptop), using MatLab 7.0 (R14), the computation of the SVD of the dense 2,240-by-447,143 matrix A takes ca 20 minutes.
- Computing this SVD is not a one-liner, since we can not load the whole matrix in RAM (runs out-of-memory in MatLab).
- Instead, compute the SVD of AA^T.
- In a similar experiment, compute **1,200 SVDs** on matrices of dimensions (approx.) 1,200-by-450,000 (roughly, a full leave-one-out cross-validation experiment) (DLP2010)

Selecting *actual columns* that "capture the structure" of the top PCs

- Combinatorial optimization problem; hard even for small matrices.
- Often called the Column Subset Selection Problem (CSSP).
- Not clear that such "good" columns even exist.
- Avoid "reification" problem of "interpreting" singular vectors!
- (Solvable in "random projection time" with CX/CUR decompositions! (PNAS, MD09))

Linear Algebra in Spark: CX and SVD/PCA implementations and performance

Alex Gittens, Jey Kottalam, Jiyan Yang, Michael F. Ringenburg, Jatin Chhugani, Evan Racah, Mohitdeep Singh, Yushu Yao, Curt Fischer, Oliver Ruebel, Benjamin Bowen, Norman Lewis, Michael Mahoney, Venkat Krishnamurthy, Prabhat

December 2015

Why do linear algebra in Spark?

Con: Classical MPI-based linear algebra algorithms will be faster and more efficient

Potential Pros:

- Faster development
- One abstract uniform interface
- An entire ecosystem that can be used before and after the NLA computations
- To some extent, Spark can take advantage of the available linear algebra codes
- Automatic fault-tolerance
- Transparent support for out of memory calculations

The Decompositional Approach

"The underlying principle of the decompositional approach to matrix computation is that it is not the business of matrix algorithmicists to solve particular problems but to construct computational platforms from which a wide variety of problems can be solved"

- A decomposition solves a multitude of problems
- They are expensive to compute, but can be reused
- Different algorithms can produce the same product
- Facilitates rounding-error analysis
- Can be updated efficiently
- Well-engineered black-box solutions are available

[G.W. Stewart, "The decompositional approach to matrix computation" (2000)]

The Big 6 Decompositions

- Cholesky Decomposition
  $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ — solving positive-definite linear systems
- LU Decomposition
  $\mathbf{A} = \mathbf{L}\mathbf{U}$ — solving general linear systems
- QR Decomposition
  $\mathbf{A} = \mathbf{Q}\mathbf{R}$ — least squares problems; dimensionality reduction
- Spectral Decomposition
  $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$ — analysis of physical systems
- Schur Decomposition
  $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H$ — more stable alternative to eigenvectors
- Singular Value Decomposition
  $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ — low-rank approximation

SVD and PCA

The SVD decomposes a matrix into a basis for its column space (U), a basis for its row space (V) , and singular values (Σ)

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where

$$\mathbf{A} \in \mathbf{R}^{m \times n}$$

$$\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_r] \in \mathbb{R}^{m \times r} \qquad \mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_r] \in \mathbb{R}^{n \times r}$$

$$\mathbf{\Sigma} = \mathrm{Diag}(\sigma_1, \ldots, \sigma_r) \qquad \sigma_1 \geq \cdots \geq \sigma_r > 0$$

If the matrix has zero-mean rows, then its SVD is called the *Principal Components Analysis (PCA)*, and U, V, and Σ are interpreted as capturing modes/directions and amounts of variation.

Truncated SVD

The computation time of the full SVD decomposition scales like $O(mn^2)$ so it can be infeasible to compute the full SVD.

Often (for dimensionality reduction, physical interpretation, etc.) instead it suffices to compute the rank-k truncated SVD (PCA)

$$\mathbf{A}_k = \text{argmin}_{\text{rank}(\mathbf{B})=k} \|\mathbf{A} - \mathbf{B}\|_F^2$$

which is given by

$$\begin{aligned}
\mathbf{A}_k &= \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T \\
\mathbf{U}_k &= [\mathbf{u}_1, \ldots, \mathbf{u}_k] \\
\mathbf{\Sigma}_k &= \text{Diag}(\sigma_1, \ldots, \sigma_k) \\
\mathbf{V}_k &= [\mathbf{v}_1, \ldots, \mathbf{v}_k]
\end{aligned}$$

and can be computed in $O(mnk)$

Computing the Truncated SVD (I)

To get the right singular vectors of A, we can compute the eigenvectors of A^TA, because

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T \Rightarrow \mathbf{A}^T\mathbf{A} = \mathbf{V}\boldsymbol{\Sigma}^2\mathbf{V}^T$$

Once we have V_k, we can use its orthogonality to recover Σ_k and U_k from

$$\mathbf{A}\mathbf{V}_k = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T\mathbf{V}_k = \mathbf{U}_k\boldsymbol{\Sigma}_k$$

Thus the two steps in computing the truncated SVD of A are:

requires only matrix vector multiplies

1. Compute the truncated SVD of A^TA to get V_k
2. Compute the SVD of AV_k to get Σ_k and V_k

assume this is small enough that the SVD can be computed locally

Computing the Truncated SVD (II)

To compute the truncated SVD of $M = A^TA$, we use the Lanczos algorithm

The idea is to restrict M to *Krylov subspaces* of increasing dimensionality:

$$\begin{aligned}\mathcal{K}_s &= \text{span}(\mathbf{x}_0, \mathbf{M}\mathbf{x}_0, \ldots, \mathbf{M}^{s-1}\mathbf{x}_0) \\ &= \mathbf{Q}_s\mathbf{R}_s \\ \mathbf{H}_s &= \mathbf{Q}_s^T\mathbf{M}\mathbf{Q}_s \in \mathbb{R}^{s\times s}\end{aligned}$$

As s increases, the eigenvalues/vectors of H_s approximate the extreme eigenvalues/vectors of M and H_s is much smaller.

Because of the special structure of the Krylov subspace and the fact M is symmetric, going from H_s to H_{s+1} is very efficient and requires only the cost of a matrix-vector multiply by $M=A^TA$

Implementing the truncated SVD algorithm in Spark

Our Scala-Spark implementation assumes:

1. A is a (tall-skinny) dense matrix of Doubles given as an `spark.mllib.linalg.distributed.IndexedRowMatrix`
2. k is small enough that AV_k fits in memory on the executor *and* is small enough not to violate the JVM array size restriction ($k*m < 2^{32}$) e.g. for k = 100, this means m must be less than 43 billion.

Recall the overall algorithm

1. Use Lanczos on A^TA to get V_k
2. Compute the SVD of AV_k to get Σ_k and U_k

The second step is done by using Breeze on the driver

Computing the Lanczos iterations using Spark (I)

We call the `spark.mllib.linalg.EigenvalueDecomposition` interface to the ARPACK implementation of the Lanczos method

This requires a function which computes a matrix-product against A^TA

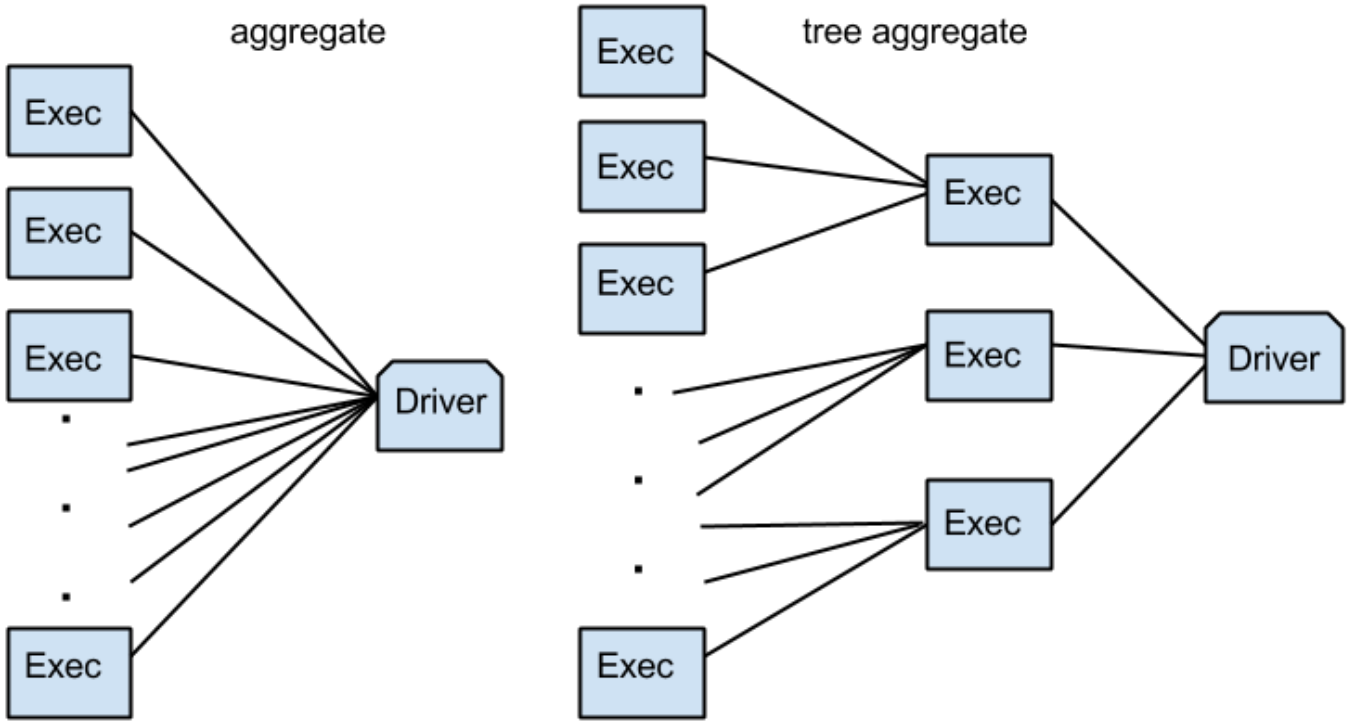
If $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_2^T \end{bmatrix}$ then the product can be computed as

$$(\mathbf{A}^T\mathbf{A})\mathbf{x} = \sum_{i=1}^{m} \mathbf{a}_i(\mathbf{a}_i^T\mathbf{x})$$

Computing the Lanczos iterations using Spark (II)

$$(\mathbf{A}^T\mathbf{A})\mathbf{x} = \sum_{i=1}^{m} \mathbf{a}_i(\mathbf{a}_i^T\mathbf{x})$$

is computed using a treeAggregate operation over the RDD



[src: https://databricks.com/blog/2014/09/22/spark-1-1-mllib-performance-improvements.html]

Spark SVD performance (I)

Experimental Setup:

- A 30-node EC2 cluster of r3.8xlarge instances (960 nodes with 7.2 TB RAM)
- A is a 6,349,676-by-46,715 dense matrix of Doubles (about 1.2 Tb)
- A is stored in Parquet format, row-wise
- A is zero-meaned and the columns are standardized and is stored in memory
- k = 20

Spark SVD performance (II)

	Run 1	Run 2	Run 3
Mean/Std of A^TAx	18.2s (1.2s)	18.2s (2s)	19.2s (1.9s)
Lanczos iterations	70	70	70
Time in Lanczos	21.3 min	21.3 min	22.4 min
Time to collect AV_k	29s	34s	30s
Time to load A in mem*	4.2 min	4.1 min	3.9 min
Total Time	26 min	26 min	26.8 min

* we zero-mean and standardize the columns of A to compute a variant of the PCA

The CX Decomposition (I)

- Dimensionality reduction is a ubiquitous tool in science (bio-imaging, neuro-imaging, genetics, chemistry, climatology, …), typical approaches include PCA and NMF which give approximations that rely on nonlinear combinations of the datapoint in A

- PCA, NMF, etc. lack reifiability. Instead, CX matrix decompositions identify **exemplar** data points (columns of A) that capture the same information as the top singular vectors, and give approximations of the form

$$\mathbf{A} \approx \mathbf{CX}$$

The CX Decomposition (II)

- To get accuracy comparable to the truncated rank-k SVD, the CX algorithm *randomly* samples O(k) columns with replacement from A according to the *leverage score* pmf

$$p_i = \frac{\|\mathbf{v}_i\|_2^2}{k}, \qquad \text{where} \qquad \mathbf{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_n]$$

- Since the algorithm is randomized, we can use a randomized algorithm to approximate V_k in o(mnk) time

CXDECOMPOSITION

Input: $A \in \mathbb{R}^{m \times n}$, rank parameter $k \leq \text{rank}(A)$, number of power iterations q.

Output: C.

1: Compute an approximation of the top-k right singular vectors of A denoted by $\tilde{V}_k$, using RANDOMIZEDSVD with q power iterations.
2: Let $\ell_i = \sum_{j=1}^{k} \tilde{\mathbf{v}}_{ij}^2$, where $\tilde{\mathbf{v}}_{ij}^2$ is the (i,j)-th element of $\tilde{V}_k$, for $i = 1, \ldots, n$.
3: Define $p_i = \ell_i / \sum_{j=1}^{d} \ell_j$ for $i = 1, \ldots, n$.
4: Randomly sample c columns from A in i.i.d. trials, using the importance sampling distribution $\{p_i\}_{i=1}^{n}$.

The Randomized SVD algorithm

The matrix analog of the power method:

$$\mathbf{x}_{t+1} = \frac{\mathbf{A}^T\mathbf{A}\mathbf{x}_t}{\|\mathbf{A}^T\mathbf{A}\mathbf{x}_t\|_2} \rightarrow \mathbf{v}_1$$

$$\mathbf{Q}_{t+1}, \_ = \mathrm{QR}(\mathbf{A}^T\mathbf{A}\mathbf{Q}_t) \rightarrow \mathbf{V}_k$$

RANDOMIZEDSVD Algorithm

Input: $A \in \mathbb{R}^{m\times n}$, number of power iterations $q \geq 1$, target rank $k > 0$, slack $p \geq 0$, and let $\ell = k + p$.

Output: $U\Sigma V^T \approx A_k$.

1: Initialize $B \in \mathbb{R}^{n\times\ell}$ by sampling $B_{ij} \sim \mathcal{N}(0, 1)$.
2: **for** q times **do**
3: $\quad B \leftarrow A^T AB$
4: $\quad (B, \_) \leftarrow \textsc{ThinQR}(B)$
5: **end for**
6: Let Q be the first k columns of B.
7: Let $M = AQ$.
8: Compute $(U, \Sigma, \tilde{V}^T) = \textsc{ThinSVD}(M)$.
9: Let $V = Q\tilde{V}$.

---

requires only matrix-matrix multiplies against A^TA

assumes B fits on one machine

Implementing the CX algorithm in Spark

Our Scala-Spark implementation assumes:

1. A is a fat sparse matrix of Doubles given as an `spark.mllib.linalg.distributed.IndexedRowMatrix`
2. l = k + p is small enough that B fits in memory on the executor *and* is small enough not to violate the JVM array size restriction ($l*m < 2^{32}$) e.g. for k = 100, this means m must be less than 43 billion.

The overall algorithm

1. Use the Randomized SVD to approximate V_k
2. Sample the columns of A according to the leverage probabilities

Computing the Randomized SVD using Spark

As before, if $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_2^T \end{bmatrix}$ then the product can be computed as

$$(\mathbf{A}^T\mathbf{A})\mathbf{B} = \sum_{i=1}^{m} \mathbf{a}_i(\mathbf{a}_i^T\mathbf{B})$$

and we use treeAggregation for efficiency

```
def multiplyGramian(A: RowMatrix, B: LocalMatrix) = {
  A.rows.treeAggregate(LocalMatrix.zeros(n, k))(
    seqOp = (X, row) => X += row * row.t * B,
    combOp = (X, Y) => X += Y
  )
}
```

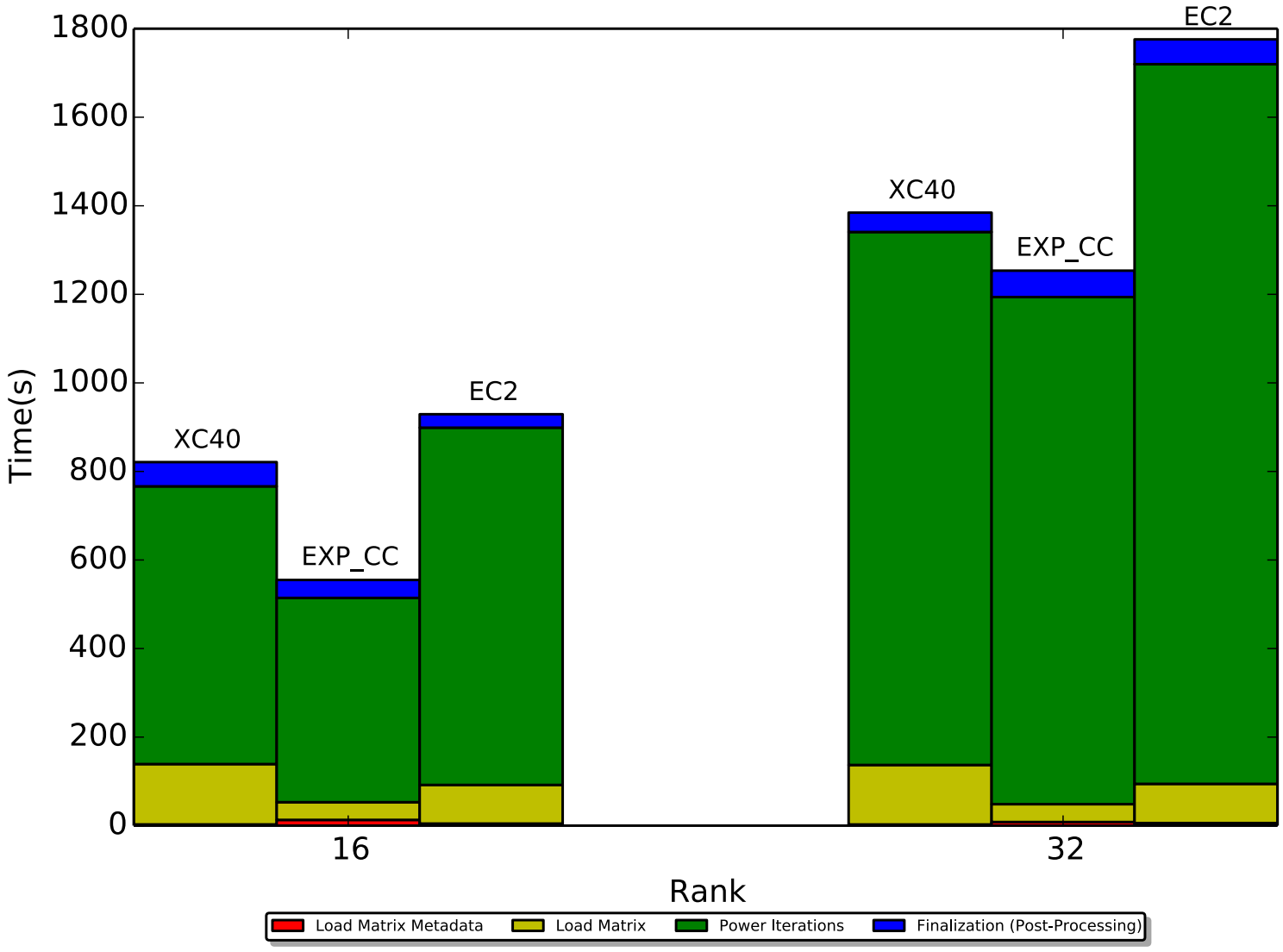
Spark CX performance (I)

Dataset: A is a 131,048-by-8,258,911 sparse matrix (1 TB)

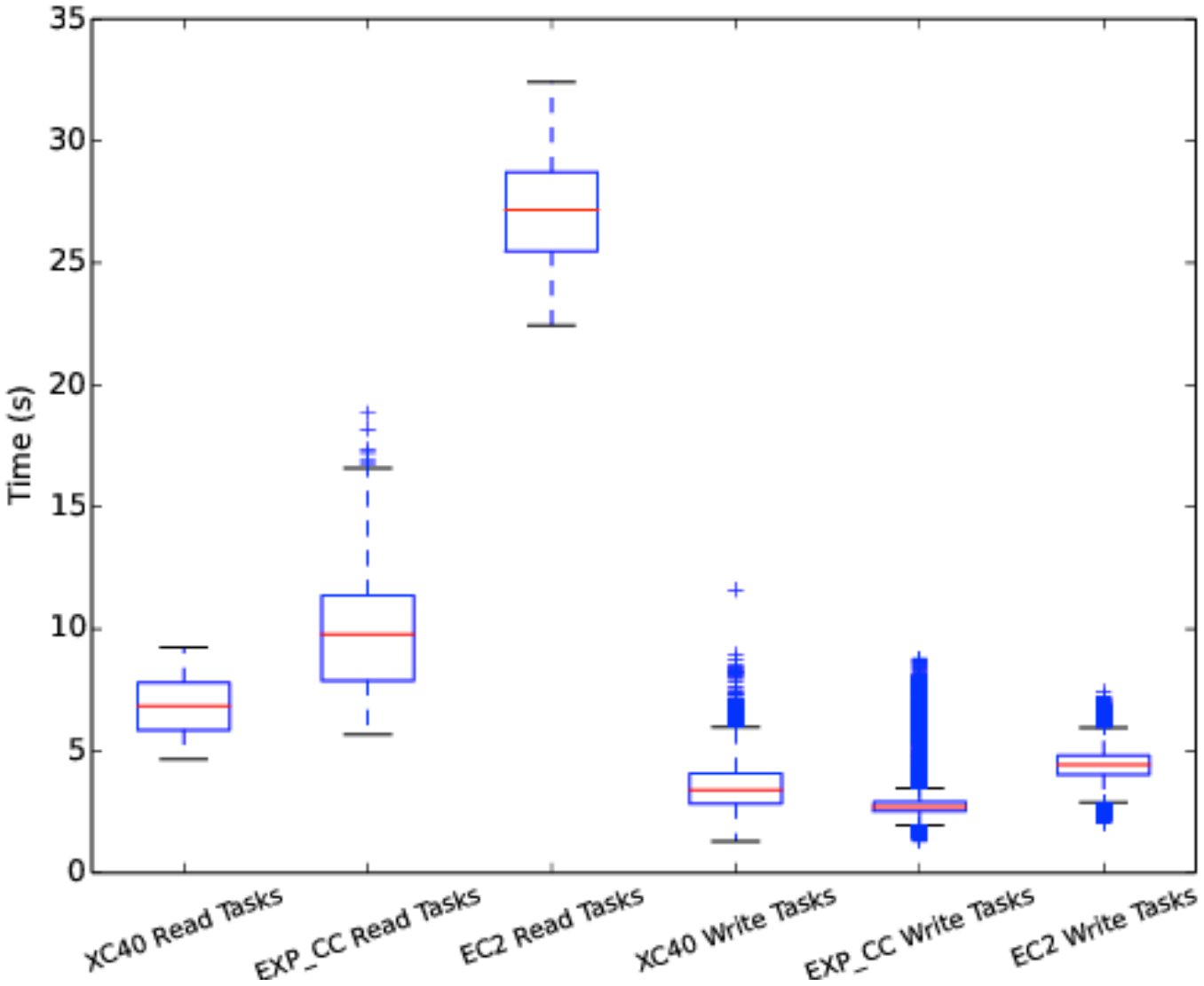
Platform	Total Cores	Core Frequency	Interconnect	DRAM	SSDs
Amazon EC2 `r3.8xlarge`	960 (32 per-node)	2.5 GHz	10 Gigabit Ethernet	244 GiB	2 x 320 GB
Cray XC40	960 (32 per-node)	2.3 GHz	Cray Aries [20], [21]	252 GiB	None
Experimental Cray cluster	960 (24 per-node)	2.5 GHz	Cray Aries [20], [21]	126 GiB	1 x 800 GB

Platform	Total Runtime	Load Time	Time Per Iteration	Average Local Task	Average Aggregation Task	Average Network Wait
Amazon EC2 `r3.8xlarge`	24.0 min	1.53 min	2.69 min	4.4 sec	27.1 sec	21.7 sec
Cray XC40	23.1 min	2.32 min	2.09 min	3.5 sec	6.8 sec	1.1 sec
Experimental Cray cluster	15.2 min	0.88 min	1.54 min	2.8 sec	9.9 sec	2.7 sec

Spark CX performance (II)



Spark CX performance (III)



Lessons learned

- the *main challenge is converting the data* to a format Spark can read
- *treeAggregation is key* (and don't be shy with changing the `depth` option) for more efficient row-based linear algebra
- increase the worker timeouts and network timeouts with
  `--conf spark.worker.timeout=1200000`
  `--conf spark.network.timeout=1200000`
  when passing around large vectors

What next

- use optimized NLA libraries under Breeze
- get the truncated SVD code to scale successfully when the RDD cannot be held in memory, or identify the culprit
- characterize the performance on EC2 and NERSC, Cray platforms of the truncated SVD code
- characterize the performance of Spark vs parallel ARPACK
- investigate how much can be gained by using block-Lanczos and communication-avoiding algorithms

CX code and IPDPS submission: https://github.com/rustandruin/sc-2015.git

Large-scale SVD/PCA code: https://github.com/rustandruin/large-scale-climate.git

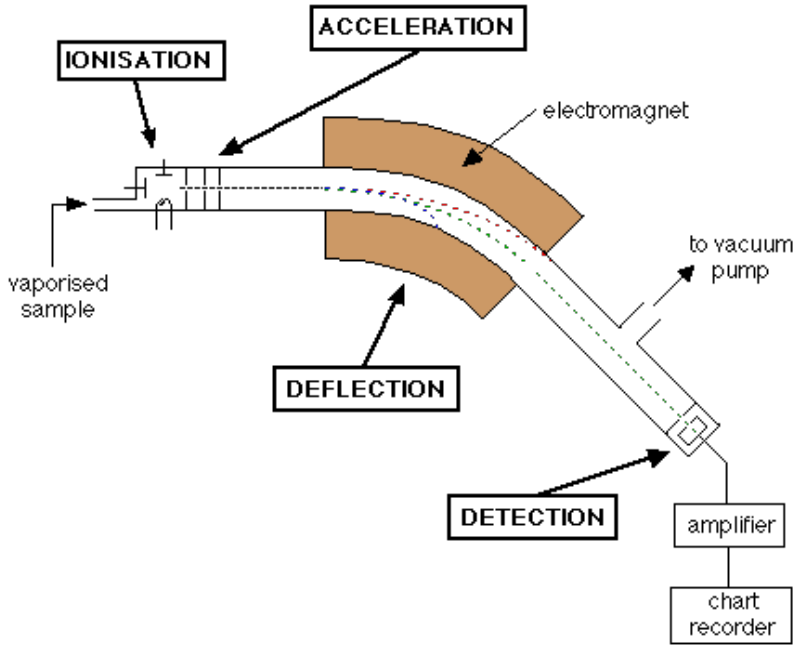
Two scientific applications of Spark implementations of the CX and PCA matrix decompositions

Alex Gittens, Jey Kottalam, Jiyan Yang, Michael F. Ringenburg, Jatin Chhugani, Evan Racah, Mohitdeep Singh, Yushu Yao, Curt Fischer, Oliver Ruebel, Benjamin Bowen, Norman Lewis, Michael Mahoney, Venkat Krishnamurthy, Prabhat

December 2015

Mass Spectrometry Imaging (I)

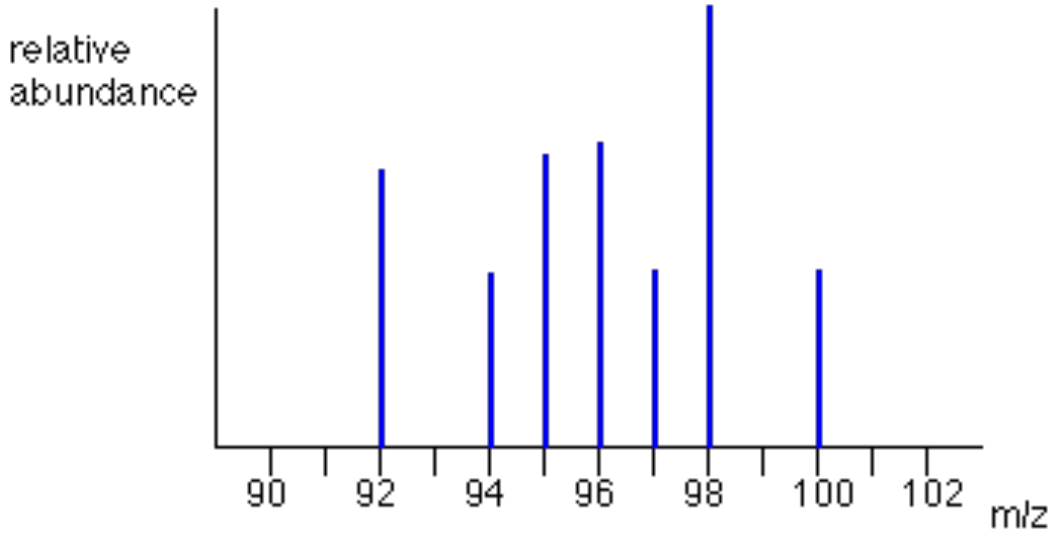
- Mass spectrometry measures ions that are derived from the molecules present in a biological sample



[src: http://www.chemguide.co.uk/analysis/masspec/howitworks.html]

Mass Spectrometry Imaging (II)

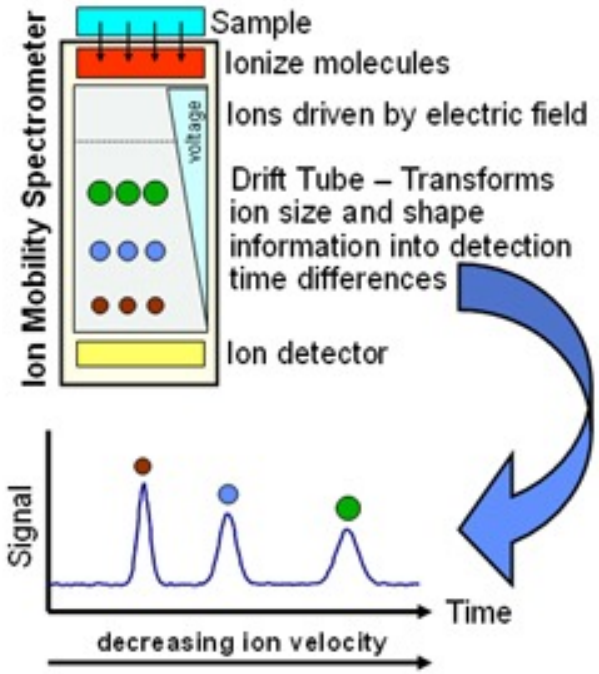
- Scanning over the 2D sample gives a 3D dataset r(x,y,m/z) where m/z is the mass-to-charge ratio and r is the relative abundance



[src: http://www.chemguide.co.uk/analysis/masspec/howitworks.html]

Ion-Mobility Mass Spectrometry Imaging

- Different ions can have the same m/z signature. Ion-mobility mass spectrometry further supplements dataset to include drift times $\boldsymbol{\tau}$, which assist in differentiating ions, giving a 4D dataset r(x,y,m/z,$\boldsymbol{\tau}$)



[src: http://www.technet.pnnl.gov/sensors/chemical/projects/ES4_IMS.stm

Ion-Mobility Mass Spectrometry Imaging in Spark (CX)

- A single mass spec image may be many gigabytes; further exacerbated by using ion-mobility mass spec imaging
- Scientists use MSI to find ions corresponding to chemically and biogically interesting compounds
- Question: can the CX decomposition, which identifies a few columns in a dataset that reliably explain a large portion of the variance in the dataset, help pinpoint important ions and locations in MSI images?

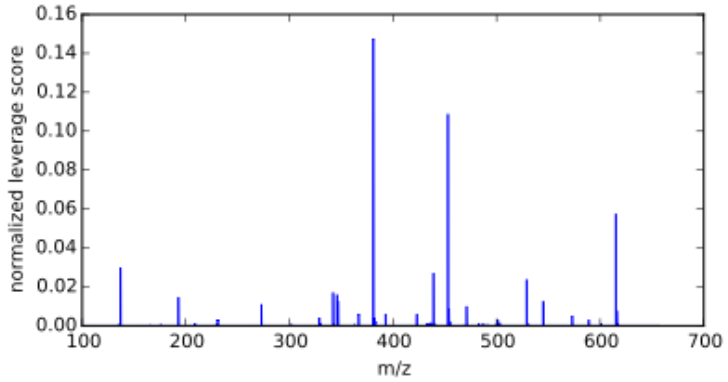
CX for Ion-Mobility MSI Results (I)

- One of the largest available Ion-Mobility MSI scans: 100GB scan of a sample of *Lewis Dalisay Peltatum* (a plant)
- A is a 8,258,911-by-131,048 matrix; with rows corresponding to pixels and columns corresponding to ($\boldsymbol{\tau}$, m/z) values
- k = 16, l = 18, p = 1

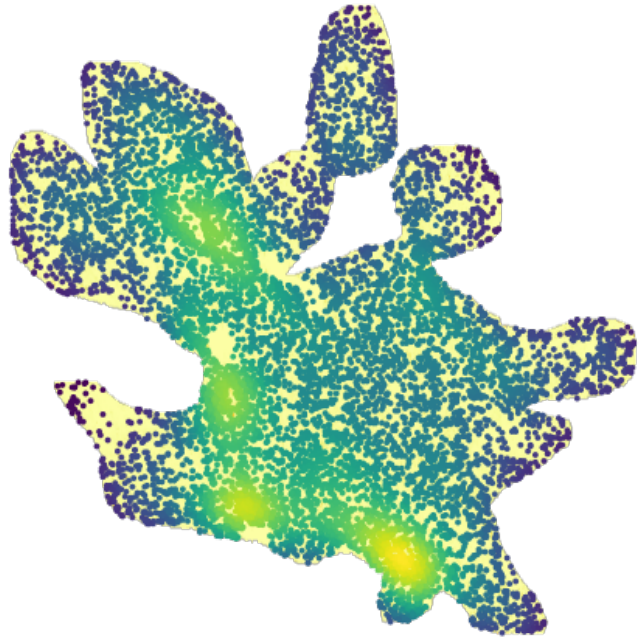
Platform	**Total Cores**	**Core Frequency**	**Interconnect**	**DRAM**	**SSDs**
Amazon EC2 `r3.8xlarge`	960 (32 per-node)	2.5 GHz	10 Gigabit Ethernet	244 GiB	2 x 320 GB
Cray XC40	960 (32 per-node)	2.3 GHz	Cray Aries [20], [21]	252 GiB	None
Experimental Cray cluster	960 (24 per-node)	2.5 GHz	Cray Aries [20], [21]	126 GiB	1 x 800 GB

Table I: Specifications of the three hardware platforms used in these performance experiments.

CX for Ion-Mobility MSI Results (II)



Normalized leverage scores (sampling probabilities) for the ions. Three regions account for 59.3% of the total probability mass. These regions correspond to ions which are chemically related, so may have similar biological origins, but have different spatial distributions within the sample.



10000 points sampled by leverage score. Color and luminance of each point indicates density of points at that location as determined by a Gaussian kernel density estimate.

Climate Analysis (PCA) in Spark

- In climate analysis, PCA (EOF analysis) is used to uncover *potentially meaningful* spatial and temporal modes of variability. Given A containing zero-mean i.i.d. observations in its rows, one column per observation interval,

$$\mathbf{A}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T \in \mathbb{R}^{m \times n}$$

- The columns of U_k capture the dominant modes of spatial variation in the anomaly field, and the columns of V_k capture the dominant modes of temporal variation

- Despite the fact that fully 3D climate fields (temperature, velocity, etc.) are available, and their usefulness, EOFs have historically only been calculated on 2D slices of these fields

- Question of interest: *Is there any scientific benefit to computing the EOFs on full 3D climate fields?*

CFSRA Datasets

- Consists of multiyear (1979—2010) global gridded representations of atmospheric and oceanic variables, generated using constant data assimilation and interpolation using a fixed model

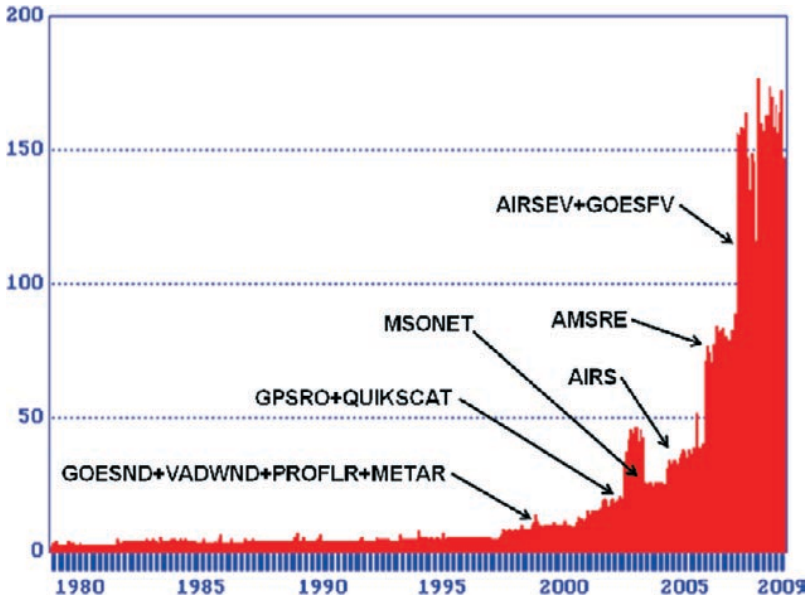


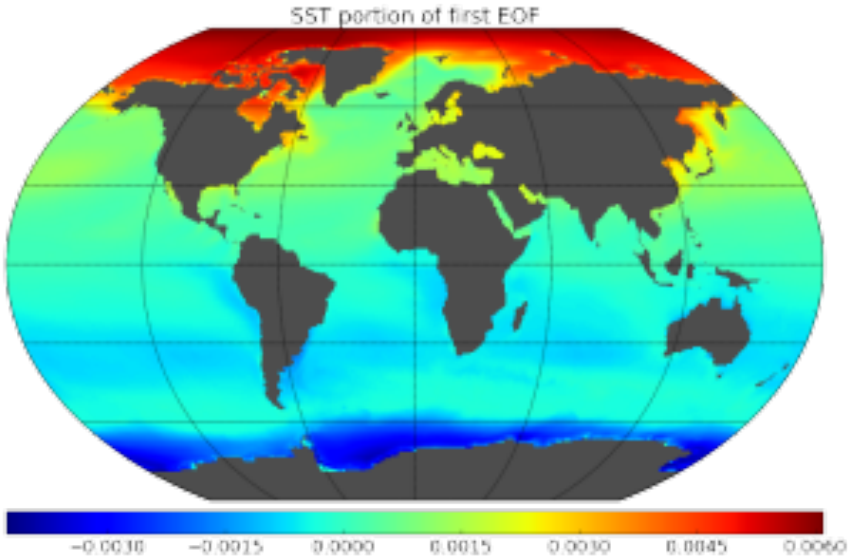
FIG. 2. Diagram illustrating CFSR data dump volumes, 1978–2009 (GB month^{-1}).

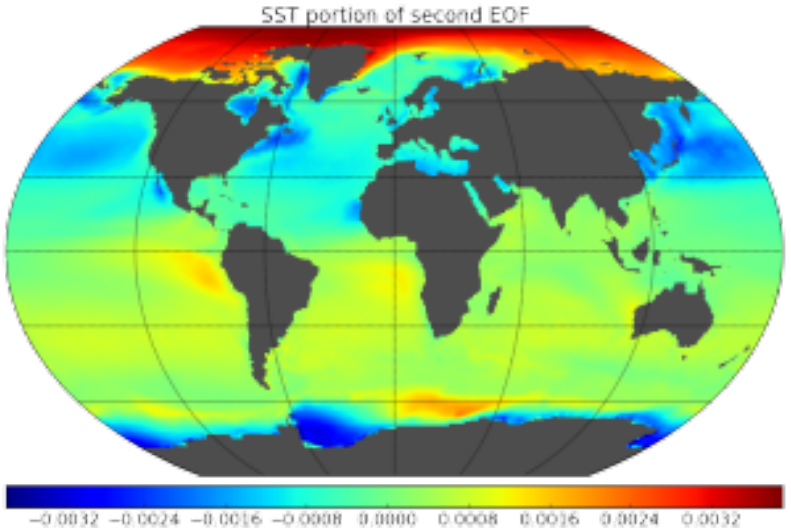
[src: http://cfs.ncep.noaa.gov/cfsr/docs/]

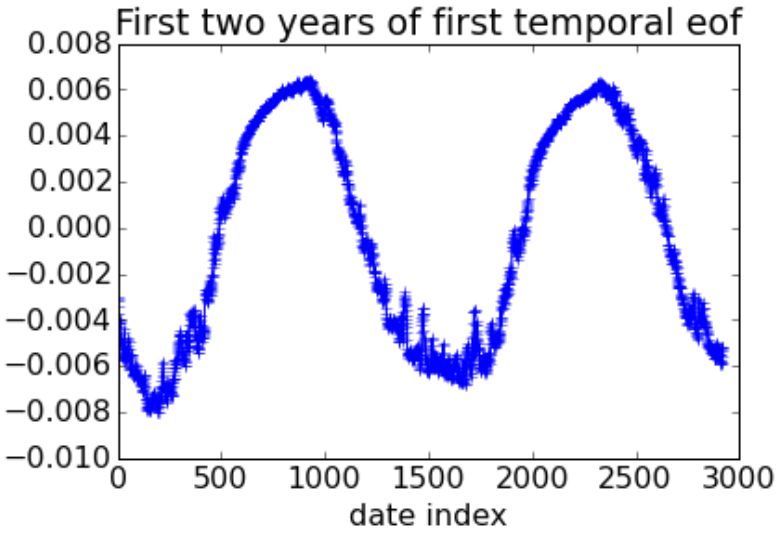
CFSR Ocean Temperature Dataset (I)

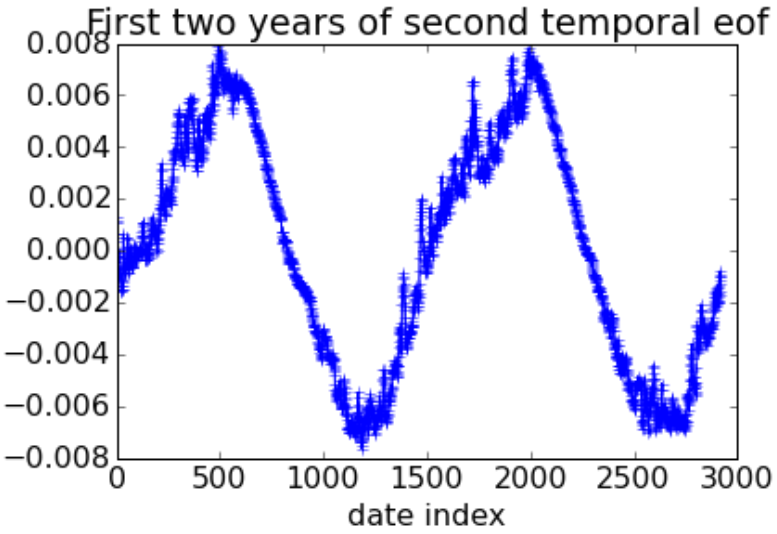
- Ocean temperature (K) observations from 1979—2010 at 6 hours intervals at 40 different depths in the ocean, on 360-by-720-by-40 grid.
- The data was provided in the form of one GRB2 file per 6 hour observation, and were converted to CSV format, then converted to Parquet format using Spark
- The subsequent analysis was conducted on this dataset
- A is a 6,349,676-by-46,715 matrix (about 1.2TB)
- computed the dominant 20 modes (captures about 81% of the variance)

CFSR Ocean Temperature Dataset (II)









CFSR Ocean Temperature Dataset (III)

- Run on a 30-node r3.8xlarge EC2 cluster (960 2.5GHz cores, 7.2TB memory) — CFSR-O cached in memory

	Run 1	Run 2	Run 3
Mean/Std of A^TAx	18.2s (1.2s)	18.2s (2s)	19.2s (1.9s)
Lanczos iterations	70	70	70
Time in Lanczos	21.3 min	21.3 min	22.4 min
Time to collect AV_k	29s	34s	30s
Time to load A in mem*	4.2 min	4.1 min	3.9 min
Total Time	26 min	26 min	26.8 min

CFSR Atmospheric Dataset

- Consists of 26 2D fields and 5 3D fields, e.g. total cloud cover (%), several types of fluxes (Wm^{-2}), convective precipitation rate (kg m^{-2} s^{-1}), …
- A is a 54,843,120-by-46,720 matrix (about 10.2 TB); because the fields are measured in different units, must normalize each row by its standard deviation
- Conversion is still a work in progress. Getting Parquet to successfully read in the data when the rows have > 54 million entries is challenging.
- This dataset will not fit in memory, so expect runtime to be much slower

Latest Point of Failure

Try to multiply against A, which is stored in Parquet format

```
val rows = {
  sqlContext.read.parquet(datafname).rdd.map {
    case SQLRow(rowname: String, values: WrappedArray[Float]) =>
    // DenseVectors have to be doubles
      val vector = new DenseVector(values.toArray.map(v => v.toDouble))
      new IndexedRow(indexLUT(rowname), vector)
    }
}

val nrows : Long = 46752
val ncols = 54843120
val A = new IndexedRowMatrix(rows, nrows, ncols)
A.rows.unpersist() // doesn't help avoid OOM

val x = new DenseMatrix(ncols, 1, BDV.rand(ncols).data)
A.multiply(x).rows.collect
```

throws an OOM error in the ParquetFileReader

```
15/12/06 05:23:36 WARN TaskSetManager: Lost task 950.0 in stage 4.0
(TID 28398, 172.31.34.233): java.lang.OutOfMemoryError: Java heap space
at org.apache.parquet.hadoop.ParquetFileReader$ConsecutiveChunkList.readAll(ParquetF
at org.apache.parquet.hadoop.ParquetFileReader.readNextRowGroup(ParquetFileReader.ja
at org.apache.parquet.hadoop.InternalParquetRecordReader.checkRead(InternalParquetRe
at org.apache.parquet.hadoop.InternalParquetRecordReader.nextKeyValue(InternalParque
at org.apache.parquet.hadoop.ParquetRecordReader.nextKeyValue(ParquetRecordReader.ja
...
```

[see http://stackoverflow.com/questions/34114571/parquet-runs-out-of-memory-on-reading]

Conclusion

Sophisticated analytics involves strong control over linear algebra.

Most workflows/applications currently do not demand much of the linear algebra.

Low-rank matrix algorithms for interpretable scientific analytics on scores of terabytes of data!

What is the “right” way to do linear algebra for large-scale data analysis?