Compressed Linear Algebra in Apache SystemML

Spark Technology Center

Fred Reiss
Arvind Surve
IBM Spark Technology Center

Strata San Jose 2017
Let us introduce ourselves.
Who we are

Fred Reiss
Apache SystemML Committer.

2014-present: Chief Architect, IBM Spark Technology Center.

Arvind Surve
Apache SystemML Committer.

Software developer at IBM Spark Technology Center.
Has worked at IBM for 17 years.
M.S. in digital electronics and communication systems.
MBA in finance and marketing.
Spark Technology Center

Founded in 2015.

Location:
Physical: 505 Howard St., San Francisco CA
Web: http://spark.tc Twitter: @apachespark_tc

Mission:
Contribute intellectual and technical capital to the Apache Spark community.
Make the core technology enterprise- and cloud-ready.
Build data science skills to drive intelligence into business applications — http://bigdatauniversity.com

Key statistics:
About 50 developers, co-located with 25 IBM designers.
Major contributions to Apache Spark http://jiras.spark.tc
Apache SystemML is now an Apache Incubator project.
Founding member of UC Berkeley AMPLab and RISE Lab
Member of R Consortium and Scala Center
Let’s talk about Compressed Linear Algebra.
Let’s talk about Linear Algebra.
Algebra \( \text{æl·dʒə·brə} \) \( n \). The study of mathematical symbols and the rules for manipulating these symbols\(^1\).

\[ y = mx + b \]

\[ x = \frac{y - b}{m} \]

In high school algebra, symbols represent numbers.

\(^1\) [https://en.wikipedia.org/wiki/Algebra](https://en.wikipedia.org/wiki/Algebra)
In linear algebra, the symbols represent arrays of numbers called **vectors** (1D) and **matrices** (2D). Many of the same operations, but extended from individual numbers to arrays of numbers.

**High School Algebra:** \( y = mx + b \)

**Linear Algebra:** \( y = X \times m + b \)
Linear Algebra is the Language of Machine Learning.

Linear algebra is powerful,
Express complex transformations over large arrays of data...

precise,
...in a clear and unambiguous way

and high-level.
...using a small number of operations.
Example: Linear Regression

Widely-used technique in machine learning. Given a matrix $X$ (feature values) and a vector $y$ (labels), find the values of $m$ and $b$ that come “closest” to satisfying the equation:

$$y = X \times m + b$$

$L_2$ regularized linear regression:

Given $X$ and $y$,
Minimize $||y - X \times m - b|| + ||X||$
S.t. $\forall i, j \ b[i] = b[j]$
L₂ regularized linear regression by the conjugate gradient method in R:

```r
X = read.csv(...)  # n x m feature matrix
y = read.csv(...)  # n x 1 feature vector
maxi = 50; lambda = 0.001; ...

r = -(t(X) %*% y)
norm_r2 = sum(r * r); p = -r  # initial gradient
w = matrix(0, ncol(X), 1); i = 0
while(i < maxi & norm_r2 > norm_r2_trgt) {
    # compute conjugate gradient
    q=((t(X)%*%(X%*%p))+lambda*p);
    # compute step size
    alpha = norm_r2 / sum(p * q);
    # update model and residuals
    w=w+alpha*p; r=r+alpha*q
    old_norm_r2 = norm_r2
    norm_r2 = sum(r^2); i = i + 1
    p = -r + norm_r2/old_norm_r2 * p
}
write.csv(w, ...);
```

Variables are matrices
Expressions use linear algebra operations
Linear Algebra-Based Programming Languages

Linear regression by the conjugate gradient method in R syntax:

```r
X = read($1)  # n x m feature matrix
y = read($2)  # n x 1 feature vector
maxi = 50; lambda = 0.001; ...
p = matrix(1, ncol(X), 1)  # initial gradient
w = matrix(0, ncol(X), 1);
i = 0

while(i < maxi & norm_r2 > norm_r2_trgt) {
    # compute conjugate gradient
    q = ((t(X) %*% (X %*% p)) + lambda * p);
    # compute step size
    alpha = norm_r2 / sum(p * q)
    # update model and residuals
    w = w + alpha * p; r = r + alpha * q
    old_norm_r2 = norm_r2
    norm_r2 = sum(r^2);
    i = i + 1
    p = -r + old_norm_r2 / norm_r2 * p
}
write(w, $3, format="text");
```

**Linear Algebra-Based Programming Languages**

- **Matrix**
- **Vector**
- **Number**
- **Matrix**
- **Vector**
- **Matrix**
- **Vector**

- Matrix transposition
- Matrix-vector multiplication
- Cellwise multiplication
Goal: Take the linear algebra formulation of an algorithm and make it scale automatically.

IBM Research technology, now an open-source Apache Incubating project.
Try SystemML online on the IBM Data Science Experience: http://datascience.ibm.com
Benefits of the SystemML Approach

Simplifies algorithm development.

Makes experimentation easier.

Your code gets faster as the system improves.

Focus of this talk
Let’s talk about Compressed Linear Algebra.
Key Insight

Common pattern: repeated memory-bound read-only access to a large matrix of lightly-transformed raw data

Usually, each row represents the features of a particular training example

Example: Linear regression algorithm from before

```java
while(...) {
    ...  
    # compute conjugate gradient
    q=(((t(X)*%*(X*%*p))+lambda*p);
    ... 
}
```

Iterative algorithm repeats until convergence

X is a large matrix of raw data

p is a small vector of coefficients
Other Examples

Principal Component Analysis

\[ C = \frac{(t(A) \times A)}{(N-1)} - \frac{(N/(N-1)) \times t(mu) \times mu;}{\text{Raw Data}} \]

Linear Regression (Direct Solve Method)

\[ \text{Raw Data} \]

Multinomial Logistic Regression

\[ \text{Raw Data} \]

Generalized Linear Models

\[ \text{Raw Data} \]

Pearson’s Correlation Coefficient

\[ \text{Raw Data} \]
Optimizing Repeated Read-Only Access

SystemML automatically chooses between three strategies, depending on data characteristics:

- Data fits on one node’s memory: Single-node, multithreaded.
- Data fits in cluster memory: Distributed, main-memory computation on Spark.
- Data doesn’t fit in memory: Stream data off of disk and back to disk.

Orders of magnitude performance difference between these modes of operation.

Key insight #1: We can fit more data in memory if we compress the raw data
  Allows us to stay in a faster mode longer as data volume grows.
Strawman Approach #1: Spark + Snappy

Compression is already built into Spark
Ostensibly for exactly this purpose.

How it’s supposed to work:
Tell Spark to use Snappy compression on cached RDD data.
Keep more data in memory.
Faster algorithm performance!

In practice:
Poor compression ratios on ML data.
Compression ratio half that of gzip in our experiments.
Decompressing data is about as fast as streaming it off of disk.
Between 232MB/s and 638MB/s in our experiments.
Sparse linear algebra: Special data structures and operators for matrices where most values are zero.

SystemML already uses sparse linear algebra.
Optimizer chooses the best representation (dense or sparse) for each matrix based on data characteristics.

But we can do better!
Sparse matrix representations often don’t achieve good compression.
Mixtures of dense and sparse fields
Repeated values (e.g. categorical data or sensor readings)
Performance degrades as data becomes less sparse.
Our Approach: Compressed Linear Algebra

Borrow techniques from database compression.

Perform critical linear algebra operations directly over compressed data, without decompression.

Up to 26X better end-to-end performance.
Compression Technique

Key insight #2: A matrix of training data is similar to a table of relational database data:

- More rows than columns.
- Each row represents a training example.
- Each column represents a feature value.

We use techniques from relational database compression:

- Dictionary coding
- Bitmap indexes
- Co-Coding
Dictionary Coding

Original Dataframe

```
+----------+--------+
| House-   | Size   |
| hold Size|       |
| 1        |       |
| 2        |       |
| 2        |       |
| 2        |       |
| 5        |       |
| 1        |       |
| 2        |       |
| 2        |       |
+----------+--------+
```

Uncompressed Matrix

```
[... 1.0 ...]
[: 2.0 :]
[: 2.0 :
[: 5.0 :
[: 1.0 :
[: 2.0 :
[: 2.0 :
```

7×64 = 448 bits

Dictionary

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1.0</td>
</tr>
<tr>
<td>01</td>
<td>2.0</td>
</tr>
<tr>
<td>10</td>
<td>5.0</td>
</tr>
</tbody>
</table>
```

Column Compressed with Dictionary Coding

```
00
01
01
10
00
01
01
```

3×2 + 3×64 + 7×2
= 212 bits
**Bitmap Indexes**

<table>
<thead>
<tr>
<th>Household Size</th>
<th>Value Bitmap-Encoded Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 0 0 0 1 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0 0 1 1 0 0</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0 0 1 1 0 0</td>
</tr>
</tbody>
</table>

Original Dataframe: `| House- | Uncompressed Matrix | Bitmap-Encoded Column | Compressed Bitmaps |
|hold Size| 1 2 2 2 5 1 1 | | 1 0 1 0 | 5.0 1 0 1 0 | 2.0 1 0 0 1 | |

Bitmaps: 1’s at positions 1, 6, and 7

Offsets Lists: 1 6 7

Run-Length Encoding:
- One “0” followed by 3 “1”’s

Compressed Bitmaps: 1 3
Co-Coding

Also works with bitmap indexes

Correlated Columns

<table>
<thead>
<tr>
<th>Household Size</th>
<th># Cars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Uncompressed Matrix

```
... 1.0 1.0 ...
.: 2.0 2.0 :.
: 2.0 1.0 :
.: 1.0 1.0 :.
: 1.0 1.0 :
: 2.0 2.0 :
: 2.0 2.0 :
```

Dictionary

<table>
<thead>
<tr>
<th>Code</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>{1.0, 1.0}</td>
</tr>
<tr>
<td>01</td>
<td>{2.0, 2.0}</td>
</tr>
<tr>
<td>10</td>
<td>{2.0, 1.0}</td>
</tr>
<tr>
<td>11</td>
<td>{5.0, 3.0}</td>
</tr>
</tbody>
</table>

Dictionary-Coded Column Group

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
Hybrid Data Structure

Can freely intermix all supported column group types Dictionary, RLE Bitmap, OLE Bitmap, CSR (lightly-compressed sparse), and Uncompressed

Example:

We support linear algebra operations directly over all types of compressed column group.
Some operations turn into *metadata operations* in the compressed domain.

Orders of magnitude faster!

**Example:** Time to compute $X^2$ (square the value of every cell in a matrix) over six different ML data sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Time to compute $X^2$ (normalized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higgs</td>
<td>Uncompressed: 0.2, Compressed: 0</td>
</tr>
<tr>
<td>Census</td>
<td>Uncompressed: 1.0, Compressed: 1.0</td>
</tr>
<tr>
<td>Covtype</td>
<td>Uncompressed: 1.0, Compressed: 1.0</td>
</tr>
<tr>
<td>ImageNet</td>
<td>Uncompressed: 1.0, Compressed: 1.0</td>
</tr>
<tr>
<td>MNIST</td>
<td>Uncompressed: 1.0, Compressed: 1.0</td>
</tr>
<tr>
<td>Airline</td>
<td>Uncompressed: 1.0, Compressed: 1.0</td>
</tr>
</tbody>
</table>

Higgs has a large uncompressed column group.
Linear Algebra in the Compressed Domain

Some operations turn into metadata operations in the compressed domain.

Orders of magnitude faster!

Example: Time to compute $X^2$ (square the value of every cell in a matrix) over six different ML data sets.
Performance-Critical Operations

Other operations require touching the compressed data.
Analysis of common algorithms showed four “heavy hitter” operations dominate running time.

Example of MVChain:

\[ q = ((t(X)^\top (X^\top p)) + \text{lambda} \cdot p); \]

We have cache-conscious, parallel algorithms for all four operations.
See our paper or code (links later) for algorithm details.
Example: Matrix-Vector Multiplication

Performance is comparable to uncompressed across six different machine learning data sets.
End-to-End Implementation

Integrated into Apache SystemML 0.13 as an experimental feature.

Instructions to enable CLA:
Update key compressed.linalg with value true in configuration file SystemML-config.xml:

```xml
<compressed.linalg>true</compressed.linalg>
```

Otherwise, completely transparent to the user
No code modifications!
Spark Technology Center

High-Level Algorithm

SystemML Optimizer

Parallel Spark Program
High-Level Algorithm

Parallel Spark Program
New operators that compress and decompress data

New kernel functions for key linear algebra ops

New rewrite rules that insert compression operations
Major Code Changes

- New compress logical plan node
- Rewrite adds compress operators
- New compress physical operator
Data structures and CLA kernel functions.

Column group partitioner

Size estimation for compression planning

New compress instruction (SMP runtime)

New compress instruction (Spark runtime)
Compression Process

1. Decide whether to compress
2. Divide matrix into blocks
3. For each block:
   - Gather sample statistics
   - Choose column groups
   - Choose compression type for each group
   - Compress each column group
End-to-End Performance

We did a detailed experimental study with six different data sets and six different machine learning algorithms. Example of typical results: (L2 SVM over MNIST data)

![Graph showing performance improvements](image)

Note that these measurements are end-to-end, including:
- Reading raw data
- Compressing raw data
- Compilation, job startup, and communication delays

5X faster. (Some algorithm and data combinations up to 26X faster)
Try Apache SystemML!
http://systemml.apache.org

Read our VLDB 2016 conference paper.
Best Paper award!

Coming soon: VLDB Journal paper
More algorithm details, better performance results
Thank You!

Thanks to the Apache SystemML Team!

Matthias Boehm
Rich Bowen
Joseph Bradley
Michael Dusenberry
Deron Eriksson
Alexandre Evfimievski
Faraz Makari
Nakul Jindal
Holden Karau
Xiangrui Meng
Madison Myers
Niketan Pansare
Berthold Reinwald
Fred Reiss
Luciano Resende
Felix Schueler
Prithviraj Sen
Henry Saputra
Arvind Surve
Shirish Tatikonda
DB Tsai
Glenn Weidner
Patrick Wendell
Imran Younus
Reynold Xin