# CSE 6040
# Computing for Data Analytics: Methods and Tools

# Low-level matrix storage

RICH VUDUC, SARA KARAMATI

DA KUANG, POLO CHAU

GEORGIA TECH

# Outline

Dense matrix
- Column-major order vs. Row-major order

Computational complexity of dense *matvec*

Sparse matrix
- Term-document matrix
- Graph adjacency matrix

Storage of a sparse matrix
- (row, column, value) -format
- Compressed sparse row (CSR) format

Sparse matrix operations

# Recall: Tibbles

A tibble is a two-dimensional table of numerical values where:

- Each row is an observation, e.g., data point in $\mathbb{R}^n$.

- Columns are variables or features.

In Python, a Pandas DataFrame is a convenient way to store a tibble. In R, a dataframe is the equivalent.

A matrix is a linear algebraic interpretation of a tibble. There are two common scenarios:

- Rows are observations and columns are variables, i.e., the tibble *is* the matrix.

- Each entry of the matrix is a tuple (row, column, value), given by one (or more) observations. That is, the tibble defines the matrix.

| depth | table | price | x | y | z |
|-------|-------|-------|------|------|------|
| 61.5 | 55.0 | 326 | 3.95 | 3.98 | 2.43 |
| 59.8 | 61.0 | 326 | 3.89 | 3.84 | 2.31 |
| 56.9 | 65.0 | 327 | 4.05 | 4.07 | 2.31 |
| 62.4 | 58.0 | 334 | 4.20 | 4.23 | 2.63 |
| 63.3 | 58.0 | 335 | 4.34 | 4.35 | 2.75 |
| 62.8 | 57.0 | 336 | 3.94 | 3.96 | 2.48 |
| 62.3 | 57.0 | 336 | 3.95 | 3.98 | 2.47 |
| 61.9 | 55.0 | 337 | 4.07 | 4.11 | 2.53 |
| 65.1 | 61.0 | 337 | 3.87 | 3.78 | 2.49 |
| 59.4 | 61.0 | 338 | 4.00 | 4.05 | 2.39 |
| 64.0 | 55.0 | 339 | 4.25 | 4.28 | 2.73 |
| 62.8 | 56.0 | 340 | 3.93 | 3.90 | 2.46 |
| 60.4 | 61.0 | 342 | 3.88 | 3.84 | 2.33 |
| 62.2 | 54.0 | 344 | 4.35 | 4.37 | 2.71 |
| 60.2 | 62.0 | 345 | 3.79 | 3.75 | 2.27 |
| 60.9 | 58.0 | 345 | 4.38 | 4.42 | 2.68 |
| 62.0 | 54.0 | 348 | 4.31 | 4.34 | 2.68 |
| 63.4 | 54.0 | 351 | 4.23 | 4.29 | 2.70 |

# Matrices

The mathematical (linear algebraic) object of a *matrix* is also a two-dimensional table, with, say, *m* rows, *n* columns, and *m\*n* values.

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix}$$

A matrix might be stored as a tibble. There are two common scenarios:

- Rows are observations and columns are variables, i.e., the tibble *is* the matrix. This option is good when the matrix is *dense*, i.e., most or all entries are non-zero.

- Each entry of the matrix is a tuple (row, column, value), given by one (or more) observations. That is, the tibble *lists* the matrix entries. This option is best when the matrix is *sparse*, i.e., most values are zero and, therefore, need not be stored explicitly.

| depth | table | price | x | y | z |
|-------|-------|-------|------|------|------|
| 61.5 | 55.0 | 326 | 3.95 | 3.98 | 2.43 |
| 59.8 | 61.0 | 326 | 3.89 | 3.84 | 2.31 |
| 56.9 | 65.0 | 327 | 4.05 | 4.07 | 2.31 |
| 62.4 | 58.0 | 334 | 4.20 | 4.23 | 2.63 |
| 63.3 | 58.0 | 335 | 4.34 | 4.35 | 2.75 |
| 62.8 | 57.0 | 336 | 3.94 | 3.96 | 2.48 |
| 62.3 | 57.0 | 336 | 3.95 | 3.98 | 2.47 |
| 61.9 | 55.0 | 337 | 4.07 | 4.11 | 2.53 |
| 65.1 | 61.0 | 337 | 3.87 | 3.78 | 2.49 |
| 59.4 | 61.0 | 338 | 4.00 | 4.05 | 2.39 |
| 64.0 | 55.0 | 339 | 4.25 | 4.28 | 2.73 |
| 62.8 | 56.0 | 340 | 3.93 | 3.90 | 2.46 |
| 60.4 | 61.0 | 342 | 3.88 | 3.84 | 2.33 |
| 62.2 | 54.0 | 344 | 4.35 | 4.37 | 2.71 |

# Dense matrix storage

Dense matrices in R and Matlab are stored in the **column-major** order: Entries in the same column are continuous in memory.
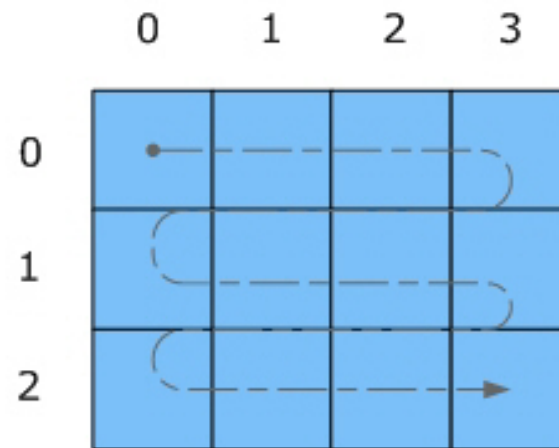
◦ Reason for column-major storage is historical: Numerical software developed in Fortran in the early days used the column-major order.

In Numpy:

◦ numpy.array([1, 2, 3], order='C') # row-major order; the default

◦ numpy.array([1, 2, 3], order='F') # column-major order
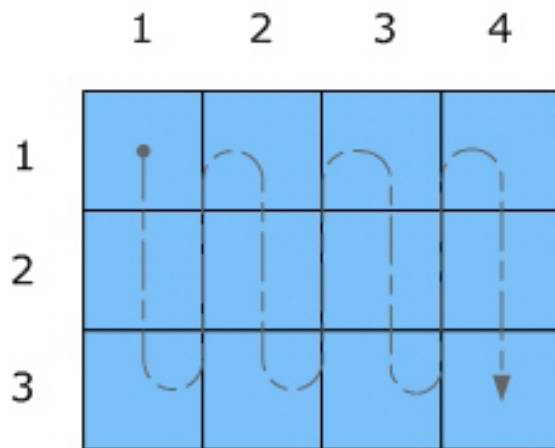


A: Column-major order (Fortran-style)     B: Row-major order (C-style)

Source: software.intel.com

# Operating in the direction of major dimension

Accessing continuous memory is efficient.

Example: For matrices in the column-major order, scaling each column is fast, and scaling each row is slow.

```
> import numpy as np
> A = np.ones((10000, 10000), order='F')

> t0 = time.time()
> for i in range(10000):
>   A[:, i] *= 2.0          # scaling each column
> print time.time() - t0   # 0.119 second

> t0 = time.time()
> for i in range(10000):
>   A[i, :] *= 2.0          # scaling each row
> print time.time() - t0   # 1.574 seconds
```

# Computational complexity of dense *matvec*

Complexity of a dense matrix $A \in \mathbb{R}^{n \times n}$ multiplying a vector $x \in \mathbb{R}^n$: $O(n^2)$

$$Ax = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n \\ A_{21}x_1 + A_{22}x_2 + \cdots + A_{2n}x_n \\ \vdots \\ A_{n1}x_1 + A_{n2}x_2 + \cdots + A_{nn}x_n \end{bmatrix}$$

How to get this number?

$n$ elements in the result vector, each requiring $n$ scalar multiplications and $n - 1$ scalar additions. Total count of flops (floating point operations): $(2n - 1)n = 2n^2 - n$
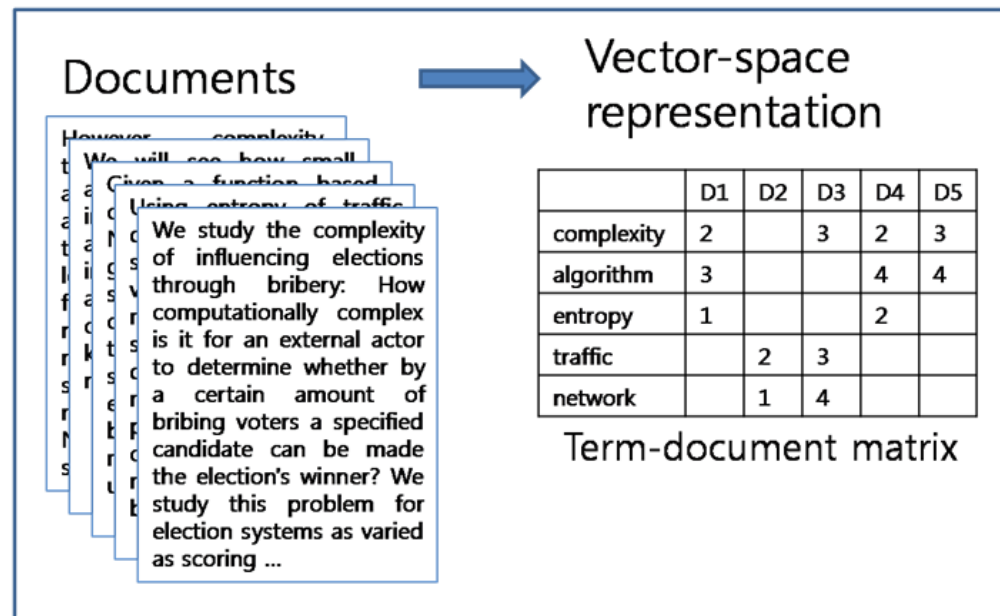
**What if the majority of $A_{ij}$ are zero??**

# Sparse matrix: Term-document matrix

How to represent a text document?

We often ignore the ordering of words and keep only the counts of each unique word in a document, resulting in a *term-document matrix*.

◦ Rows correspond to documents
◦ Columns correspond to terms

Example:

**Documents**

We study the complexity of influencing elections through bribery: How computationally complex is it for an external actor to determine whether by a certain amount of bribing voters a specified candidate can be made the election's winner? We study this problem for election systems as varied as scoring ...

**Vector-space representation**

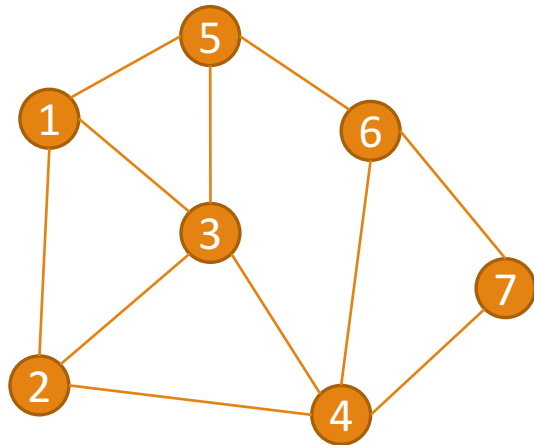|            | D1 | D2 | D3 | D4 | D5 |
|------------|----|----|----|----|----|
| complexity | 2  |    | 3  | 2  | 3  |
| algorithm  | 3  |    |    | 4  | 4  |
| entropy    | 1  |    |    | 2  |    |
| traffic    |    | 2  | 3  |    |    |
| network    |    | 1  | 4  |    |    |

Term-document matrix

Note:
Lots of zeros because all the terms cannot appear in one document.

Source: mlg.postech.ac.kr/research/nmf

# Sparse matrix: Graph adjacency matrix

How to represent a graph?



|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **1** |   | 1 | 1 |   | 1 |   |   |
| **2** | 1 |   | 1 | 1 |   |   |   |
| **3** | 1 | 1 |   | 1 | 1 |   |   |
| **4** |   | 1 | 1 |   |   | 1 | 1 |
| **5** | 1 |   | 1 |   |   | 1 |   |
| **6** |   |   |   | 1 | 1 |   | 1 |
| **7** |   |   |   | 1 |   | 1 |   |

A node in a graph is typically connected to only a small fraction of nodes.

Source: www.cs.umn.edu/~metis

# Sparse matrix is often very sparse

Term-document matrix for 4.5M English Wikipedia articles:

0.05% nonzeros


DBLP co-authorship network for 300,000 academic authors:

0.0007% nonzeros




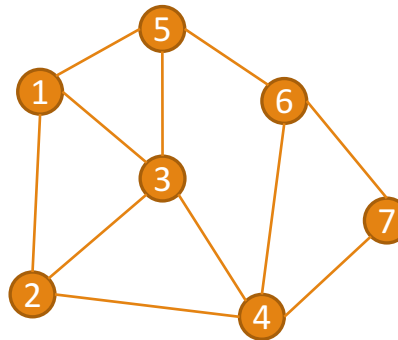We need efficient storage for sparse matrices.

# Storage of a sparse matrix

We store only the nonzeros and their positions
- ◦ (row, column, value)-triplet

Use the same example:

(1, 2, 1) (1, 3, 1) (1, 5, 1)
(2, 1, 1) (2, 3, 1) (2, 4, 1)
(3, 1, 1) (3, 2, 1) (3, 4, 1) (3, 5, 1)
(4, 2, 1) (4, 3, 1) (4, 6, 1) (4, 7, 1)
(5, 1, 1) (5, 3, 1) (5, 6, 1)
(6, 4, 1) (6, 5, 1) (6, 7, 1)
(7, 4, 1) (7, 6, 1)



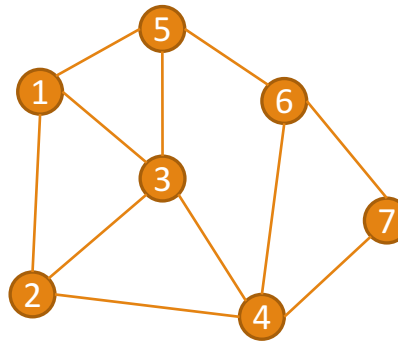|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 |   | 1 | 1 |   | 1 |   |   |
| 2 | 1 |   | 1 | 1 |   |   |   |
| 3 | 1 | 1 |   | 1 | 1 |   |   |
| 4 |   | 1 | 1 |   |   | 1 | 1 |
| 5 | 1 |   | 1 |   |   | 1 |   |
| 6 |   |   |   | 1 | 1 |   | 1 |
| 7 |   |   |   | 1 |   | 1 |   |

Viewing indices of the matrix as graph nodes, these triplets are edges.

Symmetric sparse matrix ($A = A^T$) ⇔ Undirected graph

Non-symmetric sparse matrix ⇔ Bipartite graph

# Coordinate (COO) format

The triplets can be stored as 3 arrays: rows, cols, values.



|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 |   | 1 | 1 |   | 1 |   |   |
| 2 | 1 |   | 1 | 1 |   |   |   |
| 3 | 1 | 1 |   |   | 1 | 1 |   |
| 4 |   | 1 | 1 |   |   | 1 | 1 |
| 5 | 1 |   | 1 |   |   | 1 |   |
| 6 |   |   |   | 1 | 1 |   | 1 |
| 7 |   |   |   | 1 |   | 1 |   |

`rows` = [0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6]

`cols` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Note: 0-based arrays

# Compressed sparse row (CSR) format

Suppose a sparse matrix has `nnz` nonzero entries.

`rows` = [0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6]

`cols` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

The COO format needs $3nnz$ elements to store the matrix. Can we do better?

When the nonzeros are stored row by row, we can compress the above storage:

`rowptr` = [0, 3, 6, 10, 14, 17, 20, 22]        | Row pointer |

`colind` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]        | Column index |

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]        | Values |

This CSR format needs $2nnz+n$ elements to store the matrix.

Similary, we have compressed sparse column (CSC) format.

# Sparse formats in Scipy

```
> import scipy.sparse as sp
> rows = [0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6]
> cols = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]
> values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
> A = sp.coo_matrix( (values, (rows, cols)) )
> A
<7x7 sparse matrix of type '<type 'numpy.int32'>'
        with 22 stored elements in COOrdinate format>
> B = A.tocsr()
> B
<7x7 sparse matrix of type '<type 'numpy.int32'>'
with 22 stored elements in Compressed Sparse Row format>
```

Typical and efficient way to construct a sparse matrix:
Initialize and fill in the three arrays (rows, cols, values), then call coo_matrix().

# Sparse formats in Scipy

```
> import numpy as np
> import scipy.sparse as sp
> rowptr = [0, 3, 6, 10, 14, 17, 20, 22]
> colind = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]
> values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
> A = sp.csr_matrix( (values, colind, rowptr), dtype=np.float64 )
> A
<7x7 sparse matrix of type '<type 'numpy.float64'>'
        with 22 stored elements in Compressed Sparse Row format>
> A.todense()
matrix([[ 0.,  1.,  1.,  0.,  1.,  0.,  0.],
        [ 1.,  0.,  1.,  1.,  0.,  0.,  0.],
        [ 1.,  1.,  0.,  1.,  1.,  0.,  0.],
        [ 0.,  1.,  1.,  0.,  0.,  1.,  1.],
        [ 1.,  0.,  1.,  0.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  1.,  1.,  0.,  1.],
        [ 0.,  0.,  0.,  1.,  0.,  1.,  0.]])
```

# Sparse formats in Scipy

## COO FORMAT

Facilitates fast conversion among sparse formats

Very fast conversion to and from CSR/CSC formats

Does not directly support arithmetic operations and slicing (indexing)

Intended usage:

- COO is a fast format for constructing sparse matrices
- Once a matrix has been constructed, convert to CSR or CSC format for fast arithmetic and matrix vector operations

## CSR FORMAT

Efficient arithmetic operations CSR + CSR, CSR * CSR, etc.

efficient row slicing

Fast matrix vector products

Slow column slicing operations (consider CSC)

Changes to the sparsity structure (adding/deleting nonzeros) are expensive (consider LIL or DOK)

- **(Can you figure out why?)**

Source: docs.scipy.org

# Sparse matrix operations

Operations involving two sparse matrices A, B yield a sparse matrix:

- Addition: `A+B`
- Multiplication: `A*B` or `A.dot(B)`
- Entrywise multiplication: `A.multiply(B)`

Operations involving a sparse matrix A and a dense matrix B (2D array in Numpy) yield a dense matrix:

- Addition: `A+B`
- Multiplication: `A*B` or `A.dot(B)`
- Entrywise multiplication: `A.multiply(B)`

# Computational complexity of sparse *matvec*

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **1** |   | 1 | 1 |   | 1 |   |   |
| **2** | 1 |   | 1 | 1 |   |   |   |
| **3** | 1 | 1 |   | 1 | 1 |   |   |
| **4** |   | 1 | 1 |   |   | 1 | 1 |
| **5** | 1 |   | 1 |   |   | 1 |   |
| **6** |   |   |   | 1 | 1 |   | 1 |
| **7** |   |   |   | 1 |   | 1 |   |

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1 \cdot x_2 + 1 \cdot x_3 + 1 \cdot x_5 \\ 1 \cdot x_1 + 1 \cdot x_3 + 1 \cdot x_4 \\ 1 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_4 + 1 \cdot x_5 \\ 1 \cdot x_2 + 1 \cdot x_3 + 1 \cdot x_6 + 1 \cdot x_7 \\ 1 \cdot x_1 + 1 \cdot x_3 + 1 \cdot x_6 \\ 1 \cdot x_4 + 1 \cdot x_5 + 1 \cdot x_7 \\ 1 \cdot x_4 + 1 \cdot x_6 \end{bmatrix}$$

Scalar multiplication is needed for nonzeros only.

Complexity: O(nnz)