Programming Data Science at Scale

Lecture 5: Sparse Processing

Amir Noohi

University of Edinburgh

October 16, 2025

Today's Agenda (105 mins)

Part 1: Sparse Data Structures & Algorithms (60 mins)

- Motivation: Where is sparse data?
- Data Structures: COO, CSR, CSC, SELL
- COO Optimization Techniques
- SpMV: Problem & Solutions
- Performance Analysis & Scalability

Part 2: Distributed Sparse Processing (45 mins)

- Recap: RDDs & Key-Value Operations
- Sparse Primitives in MLlib
- SpGEMM Challenge & High-Level Solution
- Next Week Preview

Goal: *Master sparse data structures and distributed algorithms for large-scale sparse matrix operations.*

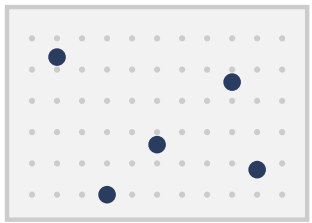
Motivation: The World is Mostly Empty

The Ubiquity of Sparsity

In big data, we often care more about what **isn't** there than what is. Most data points in high-dimensional spaces are zero.

Natural Language Processing

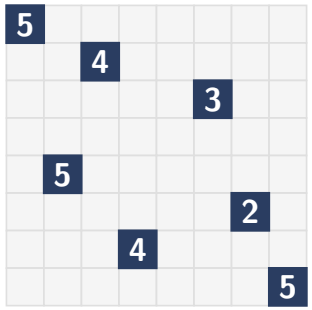
Vocabulary: 170K words



Doc uses 50 words
99.97% sparse

Recommender Systems

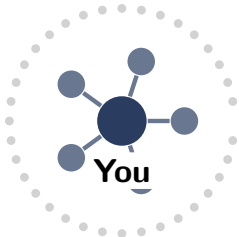
100K users $\times$ 50K movies



User rates 20 movies
99.96% sparse

Social Network

3 billion Facebook users



User has 338 friends
99.99999% sparse

The Problem with Dense Matrices

Imagine a recommender system for a small online store:

- 100,000 users
- 50,000 items

A dense matrix representation $(100,000 \times 50,000)$ would require:

- $10^5 \times 5 \times 10^4 = 5 \times 10^9$ entries.
- Assuming 8 bytes per entry (double): $5 \times 10^9 \times 8$ bytes $=$ **40 GB** of RAM.

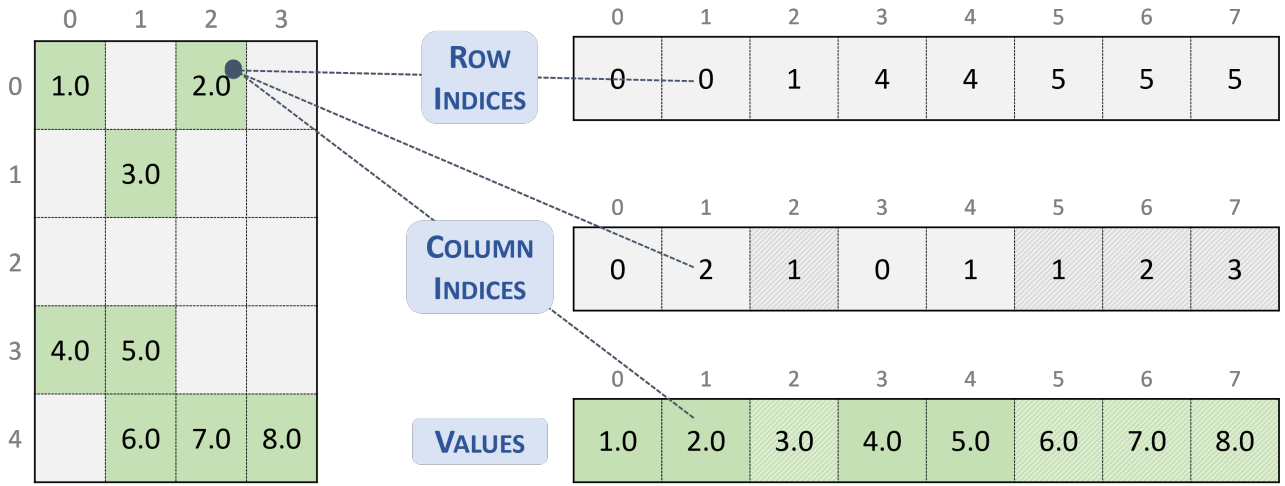
What if each user only rated 20 items?

- Non-zero entries: $100,000$ users $\times$ 20 ratings $= 2,000,000$
- Sparsity: $1 - \frac{2 \times 10^6}{5 \times 10^9} = 99.96\%$
- **99.96% of the 40 GB is wasted on storing zeros!**

Solution: Store only the non-zero values and their locations.

Data Structure 1: Coordinate List (COO)

The simplest approach: a list of ‘(row, column, value)‘ triplets.



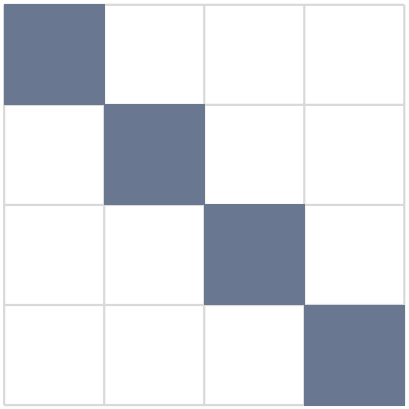
Pros & Cons

+ **Excellent for building a matrix.** Easy to append new non-zero entries.
- **Terrible for computation.** To find all elements in a row, you must scan the entire list.

COO: Visual Examples

Example 1: Identity Matrix

4×4 Identity Matrix



Only 4 entries!
75% sparse

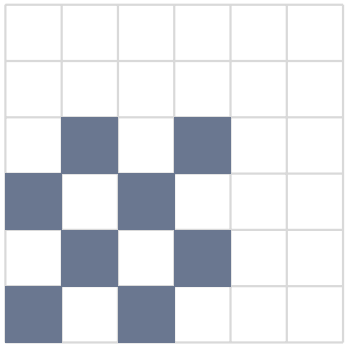
COO Representation:

Row	Col	Value
0	0	1.0
1	1	1.0
2	2	1.0
3	3	1.0

Storage: 4 entries

instead of 16!

Example 2: Checkerboard Pattern



6×6 Checkerboard
50% sparse

Pattern: Only squares where (i+j) is even have values!

- 18 entries instead of 36
- Perfect for game boards, image processing

COO Optimization Techniques

Technique 1: Row-Major Sorting

Before Sorting:

Row	Col	Value
2	1	7
0	2	5
1	0	9
1	3	2

After Row-Major Sorting:

Row	Col	Value
0	2	5
1	0	9
1	3	2
2	1	7

Benefit: Better cache locality, faster access patterns!

Memory Layout Optimization

Technique 2: Memory Layout Optimization

Structure of Arrays (SoA):

- Store all rows together, then all columns, then all values
- Better for vectorized operations on modern CPUs

Array of Structures (AoS):

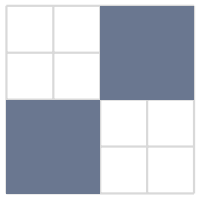
- Store (row, col, val) triplets together
- Better for sequential access patterns

Hybrid Approach:

- Use SoA for better vectorization in modern CPUs
- Switch between formats based on operation type

Advanced COO Optimizations

Technique 3: Block-Based COO



Block structure

Block COO: Group entries by blocks

- Better cache utilization
- Enables vectorized operations
- Used in high-performance libraries

Technique 4: Compressed COO (COO-C)

Compression Strategy

Store only the **differences** between consecutive entries! For sorted COO: $\Delta row = row[i] - row[i-1]$

Compressed COO Example

4×4 Sparse Matrix (Row-Major Sorted COO)

$$\begin{pmatrix} 4 & 0 & 9 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix}$$

Standard COO (Sorted):

- **rows:** [0, 0, 1, 3]
- **cols:** [0, 2, 1, 3]
- **vals:** [4, 9, 7, 5]

Compressed COO (Differences):

- **Δrows:** [0, 0, 1, 2]
- **cols:** [0, 2, 1, 3]
- **vals:** [4, 9, 7, 5]

How Δrows is calculated:

- **rows[0] = 0** $\rightarrow$ **Δrows[0] = 0** (first entry)
- **rows[1] = 0** $\rightarrow$ **Δrows[1] = 0 - 0 = 0** (same row)
- **rows[2] = 1** $\rightarrow$ **Δrows[2] = 1 - 0 = 1** (next row)
- **rows[3] = 3** $\rightarrow$ **Δrows[3] = 3 - 1 = 2** (skip 2 rows)

COO: Complex Real-World Examples

Example 1: Graph Adjacency Matrix

```
# Social network: 1M users, 10M connections
# Each edge = (user1, user2, weight)
edges = [
    (0, 1500, 0.8),    # User 0 connected to 1500
    (0, 2300, 0.6),    # User 0 connected to 2300
    (1500, 0, 0.8),    # Undirected: 1500 connected to 0
    (1500, 8900, 0.3), # User 1500 connected to 8900
    # ... 10M more edges
]
# COO: (row=user1, col=user2, value=connection_strength)

```

Example 2: Document-Term Matrix (NLP)

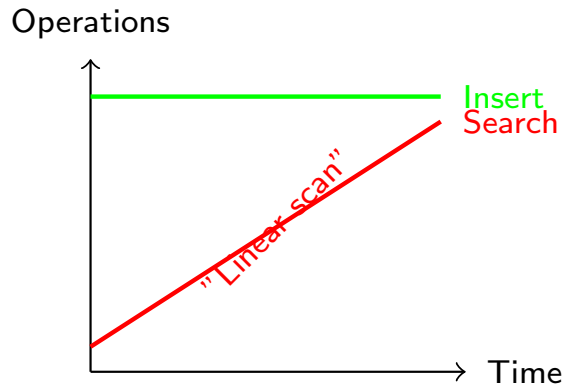
```
# 1M documents, 50K vocabulary
# Each entry = (doc_id, term_id, tfidf_score)
doc_terms = [
    (0, 1247, 0.85),   # Doc 0 contains term 1247
    (0, 8901, 0.42),   # Doc 0 contains term 8901
    (1, 1247, 0.91),   # Doc 1 also contains term 1247
    (1, 15000, 0.33),  # Doc 1 contains term 15000
    # ... millions more
]
# COO: (row=document, col=term, value=tfidf)

```

COO: Performance Characteristics

Storage: $O(nnz)$ where nnz = number of non-zeros

Operations on COO:

- **Insert:** $O(1)$ - just append!
- **Delete:** $O(nnz)$ - scan to find
- **Row access:** $O(nnz)$ - scan entire list
- **Column access:** $O(nnz)$ - scan entire list



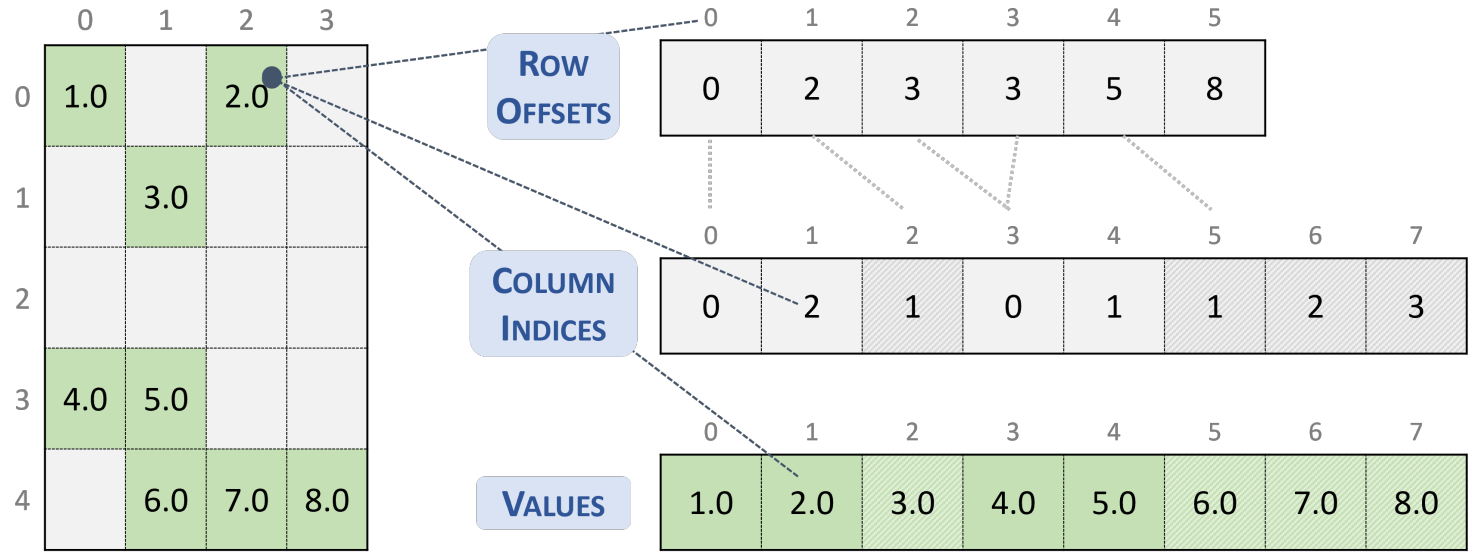
Key Insight

COO is **perfect for data ingestion** but **terrible for computation**. Always convert to CSR/CSC for algorithms!

CSR: Compressed Sparse Row

Three Arrays:

- **Values:** All non-zero elements (row-major order)
- **Column Indices:** Column position for each value
- **Row Pointers:** Starting index of each row

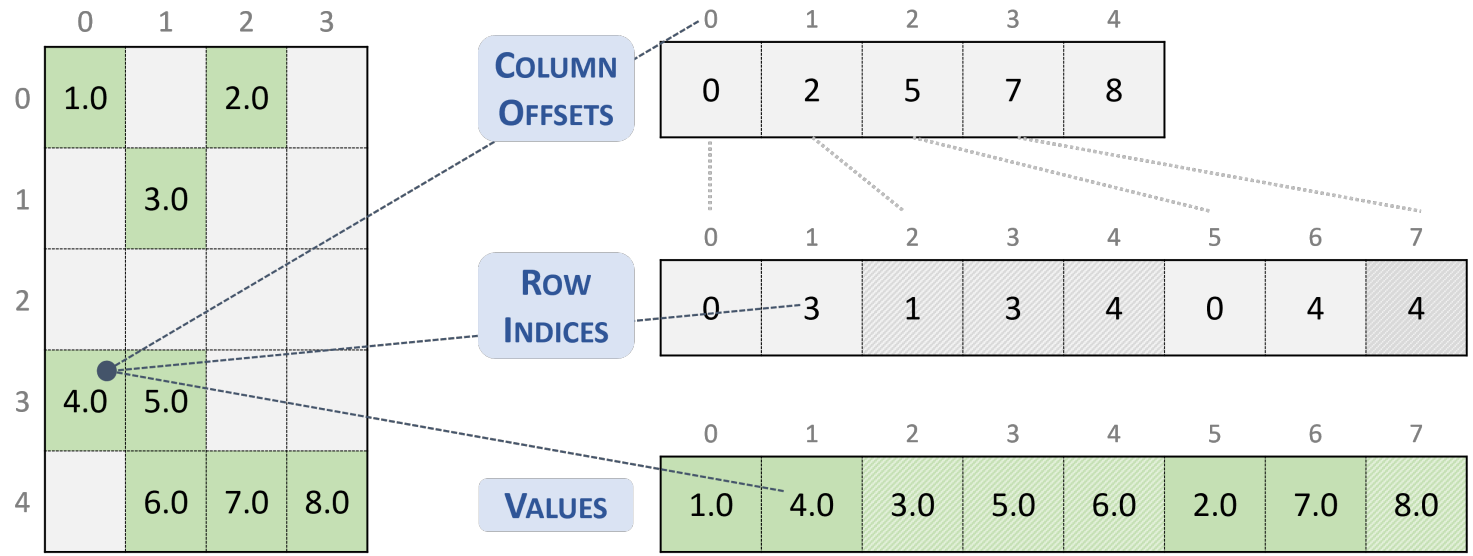


Key Insight: Use CSR for row-based operations like row slicing and row-wise computations.

Data Structure 3: Compressed Sparse Column (CSC)

Three Arrays:

- **Values:** All non-zero elements (column-major order)
- **Row Indices:** Row position for each value
- **Column Pointers:** Starting index of each column



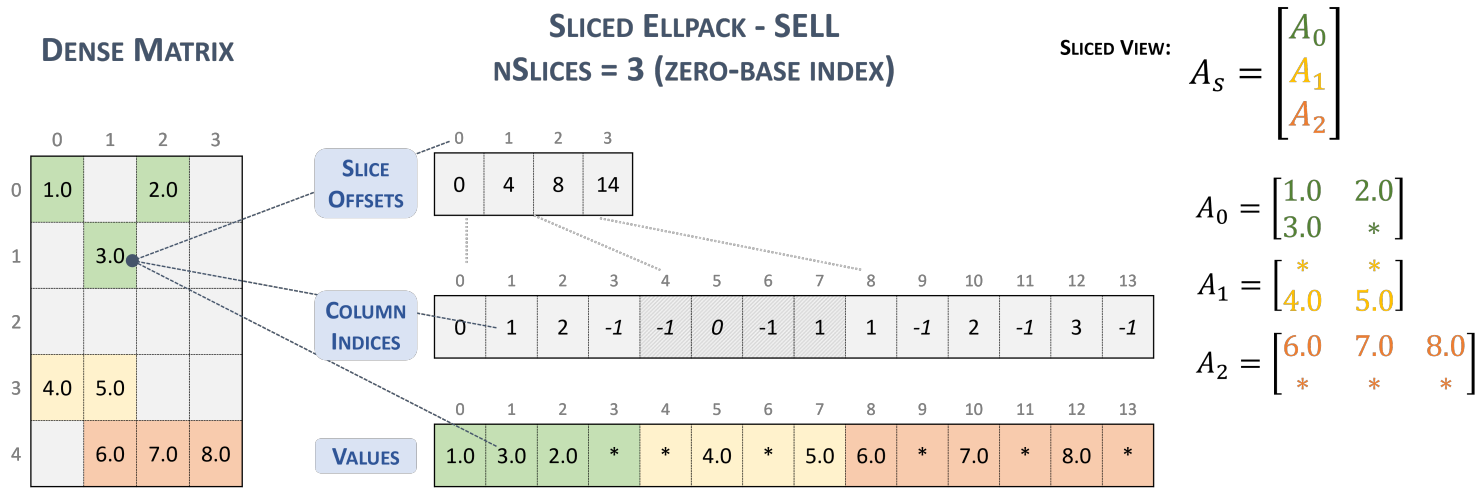
Key Insight: Use CSC for column-based operations like column slicing and column-wise computations.

Sliced Ellpack (SELL)

This format allows to significantly improve the performance of all problems that involve low variability in the number of nonzero elements per row.

Key Features:

- Matrix divided into slices of exact number of rows
- Each slice padded to maximum row length
- Value -1 used for padding
- Stored in column-major order for memory coalescing



Summary: Which Format to Use?

The choice depends entirely on your algorithm's access patterns.

Feature	**COO**	**CSR**	**CSC**	**SELL**
Storage	$O(nnz)$	$O(nnz + m)$	$O(nnz + n)$	$O(nnz + padding)$
Row Access	Slow	**Fast**	Slow	**Fast**
Column Access	Slow	Slow	**Fast**	Slow
Modification	**Fast**	Very Slow	Very Slow	Very Slow
Best For...	Building	Row ops	Column ops	GPU/Parallel

Common Workflow

Build with **COO**, convert to **CSR/CSC** for computation, use **SELL** for GPU acceleration.

Matrix-Vector Multiplication: The Problem

Example: $y = Ax$ where A is $m \times n$, x is $n \times 1$

Sparse Matrix A (4×4)

$$\begin{pmatrix} 2 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 14 \\ 2 \\ 4 \\ 15 \end{pmatrix}$$

Vector x **Result y**

Normal Algorithm: For each row i:

- $y[i] = A[i,0] \times x[0] + A[i,1] \times x[1] + A[i,2] \times x[2] + A[i,3] \times x[3]$
- **4 multiplications + 3 additions per row**
- **Total: 16 multiplications + 12 additions = 28 operations**

Problem

We do $0 \times 2 = 0$, $0 \times 3 = 0$, etc. - **Wasted operations on zeros!**

CSR Solution: Skip the Zeros!

Same Matrix in CSR Format:

Matrix A

$$\begin{pmatrix} 2 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 \end{pmatrix}$$

CSR Representation:

- **values:** [2, 3, 1, 4, 5]
- **col_indices:** [0, 3, 1, 0, 2]
- **row_ptr:** [0, 2, 3, 4, 5]

CSR Algorithm: For each row i:

- Start at `row_ptr[i]`, end at `row_ptr[i+1]`
- Only multiply non-zero values: `values[k] × x[col_indices[k]]`

Operations Count:

- **Row 0:** 2 operations (values[0]×x[0] + values[1]×x[3])
- **Row 1:** 1 operation (values[2]×x[1])
- **Row 2:** 1 operation (values[3]×x[0])
- **Row 3:** 1 operation (values[4]×x[2])
- **Total: 5 operations vs 28 operations!**

SpMV Scalability: Why CSR Matters

General Case: Matrix A is $m \times n$, Vector x is $n \times 1$

Normal Matrix Multiplication:

- **Operations:** $O(m \times n)$
- **Example:** 1000×1000 matrix
- **Operations:** 1,000,000
- **Problem:** Most are $0 \times x[j] = 0$

CSR Sparse Multiplication:

- **Operations:** $O(nnz)$
- **Example:** 1000×1000 matrix, 1000 non-zeros
- **Operations:** 1,000
- **Speedup:** 1000× faster!

SpMV-CSR Algorithm

1. For each row i from 0 to $m - 1$:
2. $y[i] = 0$
3. For k from `row_ptr[i]` to `row_ptr[i+1]-1`:
4. $y[i] += \texttt{values[k]} \times x[\texttt{col\_indices[k]}]$

5-Minute Break

Part 1 Complete!

- Sparse data structures (COO, CSR, CSC, SELL)
- Performance characteristics
- SpMV bottleneck analysis

Next: Part 2 - Distributed Processing with Spark

Recap: Spark Primitives

From Previous Lectures...

You've seen how Spark uses RDDs to represent distributed data.

- **RDDs:** Low-level, flexible collections. We use transformations like 'map', 'filter', 'reduceByKey'.
- **Pair RDDs:** RDDs of key-value pairs, enabling powerful operations like 'join' and 'groupByKey'.

Spark's Machine Learning library, **MLlib**, builds on these concepts to handle sparse data efficiently at scale.

Local Type: ‘SparseVector‘

‘SparseVector‘ is Spark’s primary way to represent a feature vector.

- Stores only non-zero values and their indices.
- Composed of three parts: ‘(size, indices, values)‘.
- Many feature transformers in Spark (e.g., ‘HashingTF‘, ‘CountVectorizer‘) output ‘SparseVector‘s automatically.

Creating a ‘SparseVector‘ in Scala

```
1 import org.apache.spark.ml.linalg.{Vector, Vectors}
2
3 // Create a sparse vector (1.0, 0.0, 3.0, 0.0)
4 // Format: Vectors.sparse(size, indices, values)
5 val sv: Vector = Vectors.sparse(4, Array(0, 2), Array(1.0,
      3.0))
6
7 // It can also be created from a sequence of tuples
8 val sv2: Vector = Vectors.sparse(4, Seq((0, 1.0), (2, 3.0)))
9
```

Local Type: ‘SparseMatrix‘

‘SparseMatrix‘ represents a local matrix on a single machine.

Key Design Choice

Spark’s ‘SparseMatrix‘ is stored in **CSC (Compressed Sparse Column)** format.

Why CSC?

- Many ML operations (like calculating statistics for a feature, or updating a model weight) are column-oriented.
- CSC format enables efficient column-wise access patterns.
- This design choice optimizes for the most common access patterns in ML pipelines.

Distributed Type: ‘CoordinateMatrix‘

A ‘CoordinateMatrix‘ is a distributed matrix backed by an ‘RDD[MatrixEntry]‘.

- Each ‘MatrixEntry‘ is just a ‘(row: Long, col: Long, value: Double)‘.
- This is a distributed version of the **COO format**.
- It’s perfect for building huge, very sparse matrices from raw data.

Creating a ‘CoordinateMatrix‘ in Scala

```
1  import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,
       MatrixEntry}
2
3  // Create an RDD of MatrixEntry objects
4  val entries = sc.parallelize(Seq(
5    MatrixEntry(0, 1, 1.2),
6    MatrixEntry(1, 2, 3.5),
7    MatrixEntry(2, 2, 0.5)
8  ))
9
10 // Create a CoordinateMatrix from the RDD
11 val coordMatrix = new CoordinateMatrix(entries)
12
```

Hands-On: Building User-Item Matrix from Logs

Scenario: Web click logs → Sparse user-item interaction matrix

CSV: user_id, item_id, rating

```
1  val logs = sc.textFile("hdfs://data/user_clicks.csv")
2
3  val entries = logs.map { line =>
4    val parts = line.split(",")
5    MatrixEntry(parts(0).toLong, parts(1).toLong, parts(2).toDouble)
6  }
7
8  val userItemMatrix = new CoordinateMatrix(entries)
9  println(s"Matrix: ${userItemMatrix.numRows()} x ${userItemMatrix.numCols()}")
10
```

100M users × 10M items = 1 quadrillion entries

- Dense storage: **8 TB**
- Sparse storage (0.02% non-zero): **16 GB**

Operations on CoordinateMatrix

Computing Matrix Statistics

```
1  // Count ratings per user (row-wise)
2  val ratingsPerUser = userItemMatrix.entries
3    .map(e => (e.i, 1)).reduceByKey(_ + _)
4
5  // Find most active user
6  val mostActive = ratingsPerUser.maxBy(_._2)
7
8  // Average rating per item (column-wise)
9  val avgRatingPerItem = userItemMatrix.entries
10   .map(e => (e.j, (e.value, 1)))
11   .reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2))
12   .mapValues { case (sum, cnt) => sum / cnt }
13
```

Key Insight: Each `reduceByKey` = shuffle = **data movement!**

The Big Challenge: Distributed SpGEMM

SpGEMM = Sparse General Matrix-Matrix multiplication ($C = AB$).

- Core operation: graph algorithms, collaborative filtering
- Challenge: A, B, C are too big for one machine

Performance Trap: '.multiply()' Method

Converting 'CoordinateMatrix' → 'RowMatrix/BlockMatrix' then calling '.multiply()':

- Internally converts to **dense** format
- Sparse matrix: 100 GB → Dense: **100 TB**
- Result: `OutOfMemoryError`

Solution: Implement SpGEMM using map-reduce primitives

SpGEMM: High-Level Solution

Challenge: Compute $C = A \times B$ where matrices are too large for one machine

Matrix A (2×3)

$$\begin{pmatrix} 2 & 0 & 3 \\ 0 & 1 & 0 \end{pmatrix}$$

$\times$

Matrix B (3×2)

$$\begin{pmatrix} 4 & 0 \\ 0 & 5 \\ 6 & 0 \end{pmatrix}$$

Solution Strategy

1. **Align:** Group elements that will be multiplied together
2. **Multiply:** Compute products of matching pairs
3. **Sum:** Add all products for each output position

Key Takeaways

1. **Sparsity is everywhere.** 99%+ zeros in NLP, recommender systems, graphs. Exploiting sparsity is essential for scalability.
2. **Data structure = performance.** COO for building, CSR for rows, CSC for columns. Choose based on access patterns.
3. **Distributed sparse ops use map-reduce.** SpGEMM shows how ‘map‘, ‘join‘, and ‘reduceByKey‘ enable scalable algorithms.

Critical Observation

The `join` operation in SpGEMM is the **most expensive** step. But what exactly happens during a join?

Next Week: Hash Partitioning & Shuffling

Today: Sparse operations rely on joins $\rightarrow$ shuffles

Next Week Topics:

1. **Hash Partitioning:** How Spark assigns keys to nodes
2. **Shuffle Internals:** Map output, network, reduce input
3. **Skew Problems:** When some keys dominate
4. **Optimizations:** Broadcast joins, pre-partitioning

Think About

- Why hash function vs random assignment?
- What if dimension j in SpGEMM is skewed (1M occurrences)?
- How to reduce shuffle for repeated A$\times$B operations?

Questions?