PDSS Project: A Distributed Engine for Large-Scale Sparse Matrix and Tensor Algebra

November 12, 2025

IMPORTANT: All red italicized text below are placeholders/instructions that you should remove and replace with your actual content!

Abstract

Write your abstract here (150-200 words)

1 Introduction

1.1 Motivation and Problem Statement

Write 3-4 paragraphs explaining: why sparse matrix operations are important (applications), technical challenges in distributed environments, your solution approach, and your main contributions.

1.2 System Overview and Architecture

Operations Implemented: *You must specify exactly which operation combinations you chose to implement. Please check (✓) the combinations you've implemented.*

Table 1: Select the combinations you will implement.

Operation	Left Operand	Right Operand	Implemented? (✓)
SpMV	Sparse Matrix	Dense Vector	
SpMV	Sparse Matrix	Sparse Vector	
SpMM	Sparse Matrix	Dense Matrix	
SpMM	Sparse Matrix	Sparse Matrix	
Tensor Algebra	(e.g., MTTKRP)		

Data Flow Diagrams: *You must create separate flow diagrams for each operation you implemented, showing the complete data path through all system components*

2 Frontend Design

2.1 User-Facing API Design

Show your API design (class, functions, or interface) that accepts different operations as input and performs the actions. Focus on design and prototypes, not full implementation code. Explain how users specify operations and how you handle different operand types (Sparse & Dense).

3 Execution Engine

3.1 Data Layout and Representation

3.1.1 Data Loading Mechanisms

The system supports multiple input formats and performs fully distributed loading using Apache Spark. Matrices can be loaded from either:

- **COO triplet files:** Each line stores a non-zero entry as `(row, col, value)`. These are parsed directly into `RDD[(Int, Int, Double)]` tuples.
- **Dense CSV files:** Each line represents a row of the matrix. During parsing, zero values are filtered out to form a sparse COO representation automatically.
- **Vector files:** Stored as `(index, value)` pairs for distributed vectors.

All data are loaded using `SparkContext.textFile()`, which partitions files across workers. Each worker parses its own lines, so large CSV or TSV inputs are processed in parallel without collecting the full dataset. —

3.1.2 Dense Representations

Dense data are stored in row-based layouts for efficient local computation within each partition:

- **DenseMatrix:** `RDD[(Int, Array[Double])]`, one row per record, providing contiguous access to row elements.
- **DistVector:** `RDD[(Int, Double)]`, where each element is stored with its index, supporting distributed joins with matrix columns for `SpMV`.

These formats minimise per-element object overhead and are efficient for small or fully populated matrices and vectors.
—

3.1.3 Sparse Representations

The system's baseline layout is the **Coordinate (COO)** format:

$$\texttt{RDD[(Int, Int, Double)]} \Rightarrow (i, j, v)$$

Each record represents one non-zero element. This format maps directly to Spark's key–value model, enabling distributed joins on row or column indices for `SpMV` and `SpMM` operations.

3.1.4 Justification

COO is chosen as the default layout because it is simple, flexible, and directly compatible with distributed key-based operations. It supports both row- and column-keyed joins without preprocessing. **DenseMatrix** and **DistVector** provide contiguous layouts for non-sparse data.

Together, these formats balance scalability, memory efficiency, and performance across different matrix operations and workload patterns.

3.1.5 Data Distribution Example:

A 3×4 sparse matrix:

$$A = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 0 & 5 \end{bmatrix}$$

is stored in COO form as:

```
RDD[(i,j,v)] = {(0,0,1.0), (0,2,2.0), (1,2,3.0), (2,0,4.0), (2,3,5.0)}
```

and distributed across workers as independent partitions, for example:

$$P_0 = \{(0,0,1.0),(0,2,2.0)\}, \quad P_1 = \{(1,2,3.0)\}, \quad P_2 = \{(2,0,4.0),(2,3,5.0)\}.$$

Each partition performs local operations on its subset of non-zeros before global aggregation.

3.2 Runtime Implementation

3.2.1 Overview

The runtime engine is built entirely on **low-level RDD transformations** without using Spark `DataFrames` or `Datasets`. Each operation is expressed as a sequence of `map`, `flatMap`, `join`, and `reduceByKey` transformations, ensuring full control over data distribution and shuffle behaviour. No operation ever invokes `collect()` or any driver-side materialisation of large RDDs. All intermediate and final results remain distributed until a terminal action such as `count()` or `saveAsTextFile()` is called.

The runtime supports four distributed kernels:

- **SpMV (Sparse × Sparse Vector)**
- **SpMV (Sparse × Dense Vector)**
- **SpMM (Sparse × Sparse Matrix)**
- **SpMM (Sparse × Dense Matrix)**

3.2.2 SpMV: Sparse Matrix × Sparse Vector

Given $A(i,j,v)$ and $x(j,x_j)$, the product

$$y_i = \sum_j A_{ij} x_j$$

is computed as follows:

1. **Join:** The matrix is keyed by its column index j and joined with the vector on the same key: `A.entries.map((j,(i,v))).j`
2. **Map:** Each joined pair emits a partial result `(i, v * xj)`.
3. **Aggregation:** Partial values are summed per row index using `reduceByKey(_+_)`.

This join-based strategy scales efficiently with matrix sparsity and avoids broadcasting the vector across workers. Below is the code implementation:

```
def spmv(A: SparseMatrix, x: DistVector): RDD[(Int, Double)] = {
  val Akeyed = A.entries.map { case (i, j, v) => (j, (i, v)) }
  val joined = Akeyed.join(x.values)
  joined.map { case (_, ((i, v), xj)) => (i, v * xj) }
        .reduceByKey(_ + _)
}
```

3.2.3 SpMV: Sparse Matrix × Dense Vector

When the vector is dense, the runtime instead uses a `Broadcast` variable. Each partition accesses the dense array directly:

$$y_i = \sum_j A_{ij} \cdot x_j$$

- **Map:** Each non-zero (i, j, v) retrieves x_j locally from the broadcast array, multiplies, and emits $(i, v \cdot x_j)$.
- **Reduce:** Results are summed with `reduceByKey(_+_)` to form the output vector.

This approach eliminates the shuffle associated with joins and is more efficient when the dense vector comfortably fits in memory. Below is the code implementation:

```
def spmvCooWithDense(A: SparseMatrix, x_bcast: Broadcast[Array[Double]]): RDD[(Int, Double
  val x = x_bcast.value
  A.entries.map { case (i, j, v) =>
      val xj = if (j < x.length) x(j) else 0.0
      (i, v * xj)
  }.reduceByKey(_ + _)
}
```

3.2.4 SpMM: Sparse Matrix × Sparse Matrix

For $A(i, k, v_A)$ and $B(k, j, v_B)$:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

1. **Join:** Both matrices are keyed on k and co-partitioned using a `HashPartitioner` to reduce shuffle.
2. **Map:** Joined records emit `((i,j), vA * vB)`.
3. **Aggregation:** Partial products are accumulated via `reduceByKey(_+_)` to produce $C(i, j, v)$.

The COO layout aligns naturally with Spark's key–value model, allowing clean expression of distributed matrix multiplication. Below is the code implementation:

```
def spmm(A: SparseMatrix, B: SparseMatrix): RDD[((Int, Int), Double)] = {
  val AbyK = A.entries.map { case (i, k, vA) => (k, (i, vA)) }
  val BbyK = B.entries.map { case (k, j, vB) => (k, (j, vB)) }
  val joined = AbyK.join(BbyK)                    // (k, ((i,vA),(j,vB)))
  joined.map { case (_, ((i, vA), (j, vB))) =>
      ((i, j), vA * vB)
  }.reduceByKey(_ + _)
}
```

3.2.5 SpMM: Sparse Matrix × Dense Matrix

Given a sparse matrix $A(i, j, v)$ and a dense matrix $B(j, \mathbf{b}_j)$ stored as row arrays:

$$C_i = \sum_j A_{ij} \cdot \mathbf{b}_j$$

- **Join:** Each non-zero (i, j, v) is joined with the corresponding row vector $\mathbf{b}_j$ of B.
- **Map:** The dense row is scaled by v to yield a partial row contribution for i.
- **Reduce:** Partial rows are elementwise-summed using `reduceByKey`, producing `RDD[(i, Array[Double])]`.

This kernel extends the same join–map–reduce pattern to mixed sparse–dense scenarios. Below is the code implementation:

```
def spmm_dense(A: SparseMatrix, B: DenseMatrix): RDD[(Int, Array[Double])] = {
  val AkeyedByJ: RDD[(Int, (Int, Double))] =
    A.entries.map { case (i, j, v) => (j, (i, v)) }

  val joined: RDD[(Int, ((Int, Double), Array[Double]))] =
    AkeyedByJ.join(B.rows)

  val partials: RDD[(Int, Array[Double])] = joined.map {
    case (_, ((i, v), rowB)) =>
      val out = new Array[Double](rowB.length)
      var k = 0
      while (k < rowB.length) { out(k) = rowB(k) * v; k += 1 }
      (i, out)
  }

  partials.reduceByKey { (a, b) =>
    val len = math.max(a.length, b.length)
    val res = new Array[Double](len)
    var k = 0
    while (k < len) {
      val av = if (k < a.length) a(k) else 0.0
      val bv = if (k < b.length) b(k) else 0.0
      res(k) = av + bv
      k += 1
    }
    res
  }
}
```

3.2.6 Summary

Across all kernels, the engine translates high-level algebraic operations into explicit distributed pipelines built solely on RDD primitives. By combining careful keying, co-partitioning, and broadcast optimisation, the runtime executes efficiently at scale while adhering strictly to Spark's distributed dataflow model.

3.3 Distributed Optimizations

3.3.1 Overview

The runtime includes several distributed optimisation strategies aimed at reducing network communication, achieving better data locality, and maintaining load balance across Spark executors. All optimisations are implemented within the RDD abstraction layer, without relying on Spark's Catalyst optimizer or DataFrame-level physical plans. The focus is on explicit control of partitioning, co-location, persistence, and broadcast use to improve scalability and throughput for large sparse workloads.

—

3.3.2 Partitioning and Co-location

Matrix and vector RDDs are partitioned by integer keys corresponding to their join dimension. For operations such as `SpMM(A,B)`, both operands are re-keyed on the shared index k and then explicitly co-partitioned using a custom `HashPartitioner`:

$$(A_k, B_k) \Rightarrow \texttt{partitionBy(new HashPartitioner(p))}$$

This ensures that entries sharing the same k value reside on the same executor before the join. Co-location drastically reduces shuffle overhead by converting expensive all-to-all exchanges into partition-local joins. The partitioner size is tuned to `defaultParallelism × 5` to balance task granularity and scheduling overhead.

—

3.3.3 Shuffle Minimisation

Shuffle operations are the dominant cost in distributed matrix computations. The engine avoids unnecessary reshuffling by:

- Applying co-partitioning prior to joins.
- Using `reduceByKey` instead of `groupByKey`, performing partial aggregation on each partition before shuffle.
- Keeping intermediate results in the same key domain (e.g. (i, j) or (i)) between consecutive transformations.

These optimisations collectively reduce the volume of intermediate data transferred between executors and lower job stage counts.

—

3.3.4 Load Balancing for Irregular Sparsity

Sparse matrices with uneven non-zero distributions can cause workload imbalance. By using key-based partitioning on the join dimension rather than fixed row blocks, the engine spreads dense regions (hot columns or rows) across multiple executors. This dynamic hashing ensures that dense rows or columns do not create straggler tasks while keeping shuffle boundaries consistent.

—

3.3.5 Caching and Persistence

Matrices that are reused across multiple operations (e.g., chain multiplications or repeated SpMV calls) are explicitly persisted using Spark's `persist()` API with memory-level storage. Intermediate RDDs after partitioning or joining are also cached where reuse is expected. This reduces recomputation and I/O overhead, trading modest memory use for substantial performance gains.

—

3.3.6 Broadcast Variables

When one operand is small and dense (e.g., a dense vector or narrow dense matrix), it is broadcast to all executors using `sc.broadcast()`. This allows each worker to access the data locally within map transformations, avoiding shuffles entirely. The `spmvCooWithDense` kernel uses this strategy to multiply a sparse matrix by a dense vector efficiently.

—

3.3.7 Summary

Together, these optimisations—co-partitioning, shuffle minimisation, load-balanced hashing, caching, and broadcast use—enable efficient distributed execution while preserving full scalability. The runtime achieves reduced data movement, improved locality, and consistent task distribution across Spark's parallel architecture.

4 Advanced Optimizations

Once you have a working, optimized engine, explore more advanced techniques. These are excellent topics for demonstrating advanced understanding.

4.1 Advanced Data Layout Optimizations

The COO format is simple but not always the most efficient. If you implemented advanced data layouts, describe: CSR/DCSR formats (very efficient for SpMV), CSF format (if implementing tensor algebra), adaptive layout selection based on sparsity patterns, and any other advanced data layout techniques.

4.2 Algebraic Optimizations

Can you use mathematical properties to reduce the amount of computation or communication? If you implemented algorithmic optimizations, describe: matrix reordering for bandwidth reduction, symbolic preprocessing and pattern analysis, kernel fusion to reduce memory traffic, optimization of matrix multiplication chains (e.g., A·B·C order matters), distributivity law applications (A·(B+C) = A·B + A·C), and any other mathematical optimizations you applied.

5 Performance Evaluation

5.1 Experimental Setup

5.1.1 Hardware Platform

All experiments were executed on a local machine with the following specifications:

- CPU: Apple M4 (10-core, including 4 performance and 6 efficiency cores)
- Memory: 16 GB unified RAM
- Storage: SSD
- Operating System: MacOS 15.5

Each experiment was run in local[*] mode, utilising all available cores for Spark executions.
The system was kept idle during measurements to reduce background variability.

5.1.2 Software Environment

- Scala Version: 2.12.20
- Spark Version: 3.0.3
- Java Runtime: OpenJDK 11

5.1.3 Test Datasets

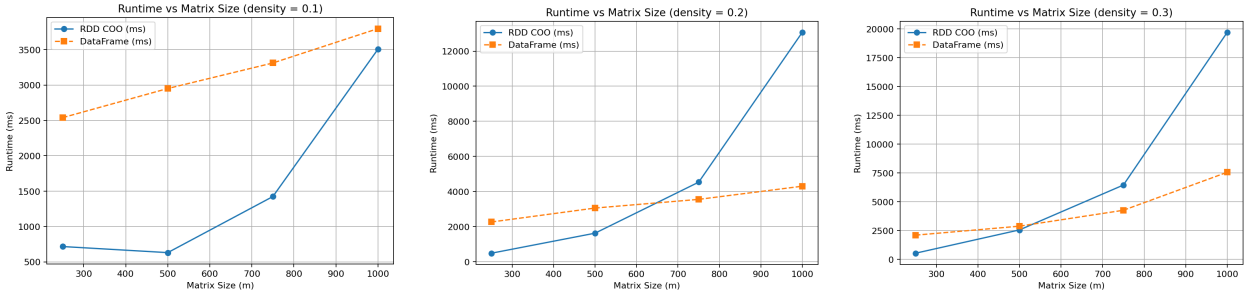
Synthetic matrices were generated using a custom Scala generation file (DatasetGen.Scala).

- Matrix Sizes ($m \times m$): 250,500,750,1000
- Densities: 0.1, 0.2, 0.3 (fraction of non-zero elements in the sparse matrix
- Format: COO (Coordinate List) representation, stored as CSV files.

5.1.4 Metrics

The performance metric measured was execution time in miliseconds for computing

$$C = A \times B$$



(a) Density=0.1 (b) Density=0.2 (c) Density=0.3

Figure 1: Effect of matrix size on runtime for different matrix densities.

5.2 Microbenchmark Results

Figure 1 compares the runtime of matrix multiplication using the RDD-COO implementation and Spark's DataFrame API as the matrix size increases (from 250 × 250 to 1000 × 1000) under three fixed sparsity levels: 0.1, 0.2, and 0.3. This design isolates the effect of matrix size on scalability while holding density constant.

The general trend for all densities is that while the RDD implementation excels at lower matrix sizes, the SQL DataFrame implementation works more efficiently with large matrices. This is especially evident in Figure 1a, where the DataFrame implementation is 4 times slower than RDD for $m = 250$, but quickly catches up, having almost the same runtime for $m = 1000$. Furthermore, it can also be seen that the RDD implementation is more efficient compared to DataFrame in lower densities. Looking at all three figures, the DataFrame implementation either gets close to, or beats the RDD implementation at a certain matrix size. However, the matrix size at which this happens decreases as the density increases, going from $m = 1000$ at $density = 0.1$ to $m = 750$ at $density = 0.2$ and $m = 500$ at $density = 0.3$. This means that for lower matrix sizes and densities, where there is less amount of data, RDD performs better, while DataFrame excels in higher matrix sizes and densities. This can be explained by the way each implementation works.

This behaviour can be explained by the execution characteristics of the two approaches. The RDD implementation relies on key-based joins and reduceByKey aggregations over sparse coordinate tuples. When the data is highly sparse, these operations are lightweight and parallelised efficiently across partitions, resulting in minimal shuffle overhead. However, as matrix size or density increases, the shuffle and join costs dominate, making the RDD execution increasingly expensive.

In contrast, the DataFrame API leverages Spark's Catalyst optimizer and Tungsten execution engine, which apply whole-stage code generation, vectorised processing, and optimized memory management. These features introduce fixed overheads at small scales but yield substantial performance benefits when handling large or dense datasets.

5.3 Impact of Distributed Optimizations

5.3.1 Impact of Number of Threads

Figure 2 shows the speedup of the RDD-based SpMM implementation with increasing thread count, for a 1000×1000 sparse matrix with 0.3 density. Looking at the plot, it can be seen that increasing the number of threds decreases the runtime, as a general trend. The performance improves up to 4 threads, reaching almost a 2.25x speedup. This indicates that the workload is successfully parallelised across cores. At 8 threads, however, the performance drops to a speedup of only 1.73x. This suggests that the overhead introduced by thread management and scheduling starts to outweigh the benefits of parallelism. For this matrix size and density, 4 threads seem to provide the optimal balance between computation and overhead.

5.3.2 Impact of Number of Partitions

Figure 3 shows the runtime of the RDD implementation with increasing number of partitions (8, 16, 32, 64, and 128). For this experiment, the thread count was kept constant at 8, and the input data was a 3000×3000 sparse matrix with $density = 0.2$. The experiment shows that while the runtime decreased with higher partitions, the improvements

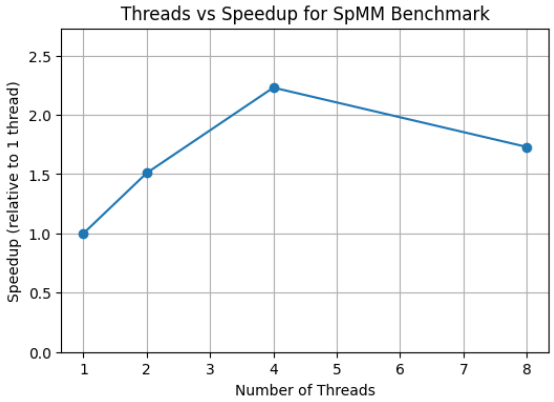


Figure 2: Speedup vs thread count.

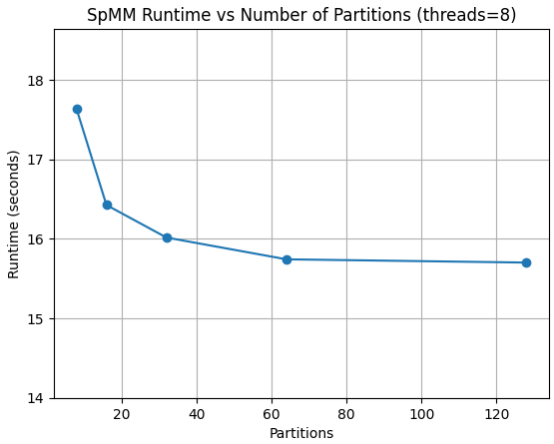


Figure 3: Runtime vs number of partitions.

diminished as the number of partitions increased, almost flattening at 128 partitions. Most of the gains were already realised by 32-64 partitions.

This means that initially, increasing the number of partitions significantly improved the load balancing across 8 cores, while too many partitions added scheduling and shuffle overhead, offering marginal benefits.

5.4 Further Ablation Studies

5.4.1 Chain Order Optimisation

Figure 4 shows the runtime for increasing matrix densities with the optimal and non-optimal chain ordering. The matrices used in this experiment are of the following size:

- $A : 5000 \times 10$
- $B : 10 \times 5000$
- $C : 5000 \times 10$

The experiments show that the optimal order remains consistently low in runtime, between 260-330 ms, across all densities. The non-optimal order, on the other hand, degrades dramatically as the density increases, going from 2.7s at 0.1 to 49s at 0.4 density.

The reason for this difference is that the optimal order first computes $B \times C$, producing a small intermediate of 10×10, which can be efficiently used in the second multiply. Conversely, the non-optimal order has an intermediate

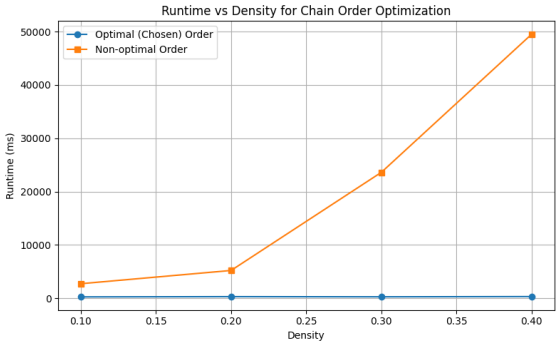


Figure 4: Runtime vs number of partitions.

of 5000×5000, which quickly becomes very dense as the input density rises, leading to a huge shuffle volume and slow second multiplication.

This experiment validates that the chain order optimisation correctly identifies the optimal plan, minimising the overall runtime.

5.5 End-to-End System Evaluation

5.5.1 Real World Application

To demonstrate the practical applicability of the execution engine beyond synthetic tensor operations, PageRank was implemented in Scala, which is a classic graph-based algorithm used in information retrieval and network analysis. The algorithm iteratively computes the distribution of a Markov chain representing web-page link structures, and it is widely used to rank web pages by importance, especially by Google until recently.

The PageRank implementation was implemented entirely using the sparse matrix-vector multiplication (SpMV) implementation.

The graph's adjacency structures was represented as a sparse matrix in COO format, where each edge (u, v) contributed an entry $(u, v, 1.0)$.

The transition matrix $M = D^{-1}A$ was constructed by normalising rows by number of outgoing edges (outdegree).

Each page rank iteration followed

$$r_{t+1} =^{T} r_t + (1 - \alpha)1/N$$

where $\alpha = 0.85$

Dangling nodes were handled directly in the engine to maintain mass conservation ($\Sigma r \approx 1$).

The algorithm was tested with a randomly generated directed graph of 100 nodes and 500 edges, with every node guaranteed to have at least five outgoing edges.

Below is the output from the experiment.

```
Iterations: 10
Sum of ranks  1,000000
Top 10 nodes by PageRank:
  node=   21   rank=2,067733e-02
  node=   80   rank=2,024477e-02
  node=   20   rank=1,950304e-02
  node=   39   rank=1,784775e-02
  node=   18   rank=1,658072e-02
  node=   87   rank=1,578205e-02
  node=    2   rank=1,551015e-02
  node=   74   rank=1,506549e-02
  node=   46   rank=1,502532e-02
  node=   76   rank=1,484480e-02
```

The experiment showed that the algorithm converged within 10 iterations, with the total rank mass ≈ 1.00, confirming numerical stability. Top nodes reached rank values of about 0.02.

This experiment demonstrates that the RDD implementation of matrix multiplication is not limited to standalone tensor multiplications but can also power higher-level workloads such as the PageRank algorithm.

Compare against alternative implementations or existing systems. Demonstrate the advantages of your approach.

Demonstrate your system with practical use cases. Show how your engine performs on realistic applications and workloads.

6 Conclusions

Summarize your main contributions: list 3-4 key achievements of your work, quantify the impact and significance of each contribution, include performance improvements and technical innovations, and highlight what makes your approach unique or effective.

Reflect on your development experience: technical insights about parallel/distributed computing, key design decisions and their implications, most difficult implementation aspects, and unexpected performance results.

Discuss potential improvements and future work: algorithm optimizations you could implement, how to scale your system to larger clusters/datasets, additional features that would enhance functionality, performance tuning opportunities you identified, and research directions for further development.

References

A Appendix

Include any additional material here