I/O efficient Algorithms for Sparse Matrix Multiplication and Addition

Hashirul Quadir, Daniel Biediger, Carlos Ordonez

Department of Computer Science
University of Houston, USA

Abstract. Matrix multiplication and addition are ubiquitous in data science, with matrix multiplication being the most critical operation. Most systems and libraries handle these computations utilizing highly optimized C code designed for dense matrices, which is wrapped in Python libraries like NumPy, with frameworks like PyTorch layered on top. However, real-world applications produce a massive volume of sparse matrices, including text data, graphs, and binary-coded features. Most Python libraries manage these structures using compressed storage formats based on CSR and CSC layouts. While these layouts are highly efficient for main memory (RAM) and cache, they perform poorly for I/O operations. In this paper, we show that efficient external memory algorithms inspired by database query processing have promising performance, effectively balancing CPU and I/O while working well with accelerators like GPUs. We study how to program such algorithms utilizing standard Python libraries, including Pandas and NumPy. Experiments demonstrate that large sparse matrices strictly require specialized algorithms, and that database algorithms provide competitive performance compared to traditional high-performance computing (HPC) algorithms.

1 Introduction

The explosion of data in various fields such as social network analysis, recommendation engines, and scientific computing has led to the ubiquity of large-scale sparse matrices. Often characterized by their $90 - 99.9$ percent sparsity, these matrices represent relationships where the vast majority of potential interactions are zero. It is critical to emphasize that matrix multiplication is the most common operation in machine learning, acting as the fundamental computational engine for both training and inference. Efficient sparse matrix multiplication is an important area to study. Traditional Python programs often rely on manual iteration or nested loops, which fail to scale as dataset dimensions increase. Eventually, this simple approach hits a performance cliff characterized by the problem's quadratic time complexity. Traditional in-memory sparse formats like the CSR and Coordinate (COO) formats continue to be limited by the physical memory of the host system. While modern GPUs offer massive parallelism, accelerating neural network training, they are still constrained by VRAM limitations. When a matrix's dimensions scale into hundreds of thousands, it becomes mathematically and physically difficult to store this dense representation in RAM, due to the significant storage that is wasted on zero entries.

Our approach is motivated by the need to handle data sizes that reach the limits of available RAM. By using little main memory, our approach removes main memory limitations and allows a system to do other things at the same time. Outside of a database, data is usually stored in CSV files. We must also consider that the trend in high-performance computing is not moving toward shared-nothing architectures, but rather the opposite: toward large single servers equipped with fast Non-Volatile Memory (NVM), huge RAM capacities, and hardware accelerators.

Our goal is to improve Python with I/O efficient, parallelizable algorithms. We demonstrate that external memory algorithms inspired by database query optimization offer a highly scalable alternative to RAM-bound libraries for massive sparse matrices. We explicitly compare traditional Relational Database Management Systems (using SQL) against high-level Python libraries (using Pandas). We utilize Python to prove that relational algebra and database execution logic can be seamlessly programmed within standard data science workflows, bypassing the need for a dedicated DBMS. Central to this architecture is the Coordinate (COO) format, which we establish as the most important format for external memory algorithms. Its straightforward row-column-value mapping allows for sequential I/O streaming directly from standard CSV files without the complex decoding overhead of compressed structures. The core of our approach involves chunked ingestion, where we utilize block-reading workflows to handle massive matrices without triggering memory spikes during the parsing phase. Once the data is in main memory, we rely on vectorized execution and relational joins, where the multiplication is expressed as a sort-merge join followed by aggregation, and matrix addition is expressed as a full outer join. We structure the execution as a strict three-phase database-style I/O workflow (Read, Process, Write) utilizing the Coordinate (COO) format to map directly to relational CSV storage. Furthermore, this approach emphasizes algorithm versatility by exploiting computer science theory topics, like B-trees and merge sorts, that are traditionally associated with databases, demonstrating they are applicable to any large-scale data manipulation, even outside a DBMS. By bridging the gap between database systems and sparse linear algebra, our work aims to identify specific thresholds where database style algorithms provide the viable path for neural networks and large-scale graph analytics.

2 Definitions

2.1 Matrix Sizes

In this paper, we define a matrix A of size $n \times n$ and a vector v of dimension n. We explicitly focus on square matrices to make our definitions, matrix multiplication execution, and algorithm analysis more straightforward. However, it is important to note that the proposed algorithms work for any rectangular matrix.

Furthermore, our explicit focus on square matrices is fundamentally motivated by their direct mathematical correspondence to large-scale sparse graphs. In graph theory and network analysis, a network of n vertices is inherently represented by an $n \times n$ adjacency matrix, defined as e, where non-zero entries denote connections between nodes. Because real-world structures such as social networks, communication grids, and

web link topologies naturally map to these square sparse dimensions, optimizing the matrix computation $e*e$ provides the most direct benefit to modern graph analytics.

2.2 Sparsity

We state the assumption that one-dimensional vectors fit into main memory. This assumption is computationally justified by the linear space complexity $O(n)$ of a vector. This contrasts with our overarching goal of handling the corresponding two-dimensional matrices, which scale quadratically to $O(n^2)$ space and strictly require external memory techniques. To bypass this, we operate under the theoretical assumption of $O(1)$ non-zero entries per row. In graph theoretical terms, we assume that the number of edges $|E|$ is close to n (the number of vertices), rather than n^2, equating to $O(1)$ neighbors per vertex. While density in real-world applications can vary logarithmically ($\log n$) or by a small constant factor (e.g., 10 or 15 neighbors), it remains strictly bounded. Assuming bounded density, our space complexity remains strictly linear, $O(n)$, allowing our external memory techniques to function efficiently.

We denote the number of non-zero entries in a given matrix by the variable k (e.g., k_A for matrix A and k_B for matrix B). We formally define the primary operations used in our benchmarks as follows. Matrix multiplication evaluates a given matrix $A \in \mathbb{R}^{n \times n}$ and a matrix $B \in \mathbb{R}^{n \times n}$, calculating their product $C = A * B$ as an $n \times n$ matrix where each entry is computed as $C_{ij} = \sum_{l=1}^{n} A_{il} * B_{lj}$. Matrix addition processes two matrices $A, B \in \mathbb{R}^{n \times n}$, evaluating their sum $C = A + B$ element-wise as $C_{ij} = A_{ij} + B_{ij}$. Finally, vector dot product takes two vectors $u, v \in \mathbb{R}^n$ and computes their dot product as the scalar value $d = u * v = \sum_{i=1}^{n} u_i * v_i$.

$$density = k/n^2 \tag{1}$$

where k represents the total number of non-zero entries and n^2 represents the total number of structural elements. Consequently, sparsity represents the proportion of zero elements within the matrix, mathematically defined as:

$$\text{Sparsity} = 1 - \frac{k}{n^2} \tag{2}$$

Because we operate assuming bounded density of $O(1)$ non-zero entries per row, k is $O(n)$. Therefore, density is $\approx 1/n$. To realistically evaluate performance trade-offs, we evaluate our computational approaches across a wide spectrum of densities.

3 I/O Efficient Scalable Algorithms for Sparse Matrices

In our work, we develop a multi-tiered architecture to evaluate sparse operations across different memory constraints and hardware configurations. We do so by categorizing these into in-memory processing to achieve performance baselines and database-driven workflows for unlimited scalability.

3.1 Matrix Operations as Database Queries

To formalize the theoretical basis of our approach, we define the fundamental operation of sparse matrix multiplication not merely as arithmetic, but as a *matching* problem. Mathematically, a valid scalar multiplication occurs if and only if $\exists l$ such that $A_{il} \neq 0$ and $B_{lj} \neq 0$. By redefining the operation to process only the intersection of these non-zero subscripts, we can explicitly map linear algebra operations directly to database query plans, where this exact mathematical matching condition natively translates to an `INNER JOIN`. Furthermore, under our established $O(1)$ neighbor assumption, the number of successful subscript matches remains strictly bounded, theoretically guaranteeing the efficiency of these relational joins at scale.

SQL Queries Matrix multiplication ($A*B$) is executed as a Join-Group-Aggregate sequence. The first phase is an `INNER JOIN` on the condition $A.j = B.i$. The second phase is a `GROUP BY` operation on ($A.i$,$B.j$) utilizing a `SUM` aggregate function to compute the dot product of the joined values. Matrix addition ($A+B$) is conceptually executed as a `FULL OUTER JOIN` on the coordinate keys (i,j). The logical projection involves selecting the coalesced keys and the sum of the values ($A.v+B.v$), treating nulls as zero. Finally, matrix-vector multiplication ($A * v$) treats the vector as a relational table and joins on the column subscript. The SQL queries for multiplication and matrix-vector multiplication are respectively programmed below (with the JOIN clauses merged onto the FROM lines):

```
SELECT A.i, B.j, sum(A.v * B.v) /* matrix-matrix */
FROM A JOIN B ON A.j = B.i
GROUP BY A.i, B.j;
```

```
SELECT A.i, sum(A.v * V.v) /* matrix-vector */
FROM A JOIN V ON A.j = V.j
GROUP BY A.i;
```

Relational Algebra Query Plans

Relational Algebra Query Plan. We formally express the execution plan for matrix multiplication using relational algebra, where the operation consists of a Join followed by a Project-Aggregate sequence:

$$\pi_{A.i,B.j,\mathrm{sum}(A.v*B.v)}(A \bowtie_{A.j=B.i} B)$$

Algorithms derived from Query Plan In the subsequent mathematical and complexity analyses of these query plans, variables such as k_A, k_B, and k_C are explicitly defined as the number of non-zero entries in their respective matrices (A, B, and C). We use sort-merge algorithms for generality, but given our $O(1)$ neighbor assumption, we can use hashing in the future.

Standard Sort-Merge Multiplication Algorithm Our Python baseline utilizes the standard sort-merge multiplication algorithm for a pure COO format. The algorithm begins with a sorting phase, ordering matrix A primarily by column (with a secondary sort by row for stability) and matrix B primarily by row (with a secondary sort by column for stability). The second phase involves a two-subscript merge and accumulation, where the sorted matrices are traversed using a two-subscript merge to find matching columns in A and rows in B. For each match, nested loops are used to iterate through the non-zero entries sharing the matched subscript, accumulating the product $(A_{il} * B_{lj})$ directly into a hash map using the computed coordinate (i,j) as the key.

Standard Sort-Merge Addition Algorithm To formalize the physical execution plan, we outline the standard database sort-merge algorithm for computing the sum matrix $C = A+B$. The algorithm takes as input two relations, $A(i,j,v)$ and $B(i,j,v)$, representing the left and right matrices, and outputs a relation $C(i,j,v)$ representing the sum matrix. The first phase consists of sorting both relation A and relation B ascending by their coordinate pairs (i,j). The second phase performs the merge, effectively executing a full outer join. It initializes two subscripts, $I_A = 0$ and $I_B = 0$, to traverse the sorted relations. While both subscripts are within their respective bounds ($I_A < |A|$ and $I_B < |B|$), the algorithm compares the current coordinate pairs. If $A[I_A].(i,j) < B[I_B].(i,j)$, it emits $A[I_A]$ and advances I_A. If $A[I_A].(i,j) > B[I_B].(i,j)$, it emits $B[I_B]$ and advances I_B. If a match is found ($A[I_A].(i,j) = B[I_B].(i,j)$), it emits a combined tuple $(A[I_A].i, A[I_A].j, A[I_A].v + B[I_B].v)$ and advances both I_A and I_B. Once the loop completes, any remaining elements from A or B are appended to the output relation C.

Relational Matrix-Vector Multiplication (SQL SpMV) Our SQL-based approach for matrix-vector multiplication $(A * v)$ treats the vector as a relational table and relies on standard database merge-joins. The algorithm begins by subscripting and organizing the data, loading matrix A and vector B into raw tables before sorting matrix A primarily by its column subscript j and vector B by its coordinate j; B-Tree subscripts are created on these key columns to facilitate efficient relational matching. The second phase involves a sort-merge join and aggregation, executing an `INNER JOIN` between matrix A and vector B on the matching condition $A.j = B.j$. The intermediate relational output is then grouped by the row subscript $A.i$, and the `SUM()` aggregate function is applied on the element-wise product $(A.v * B.v)$ to compute the final vector result.

Pandas Relational Join Multiplication Our relational Python program leverages Pandas DataFrames to model the sparse matrices, replacing manual iteration with optimized database-style query execution. The algorithm executes the relational join by representing matrices A and B as relation tables with the schema (row, col, val) and performing a `pd.merge` (Inner Join) on the condition $A.\text{col} = B.\text{row}$. The second phase calculates element-wise multiplication by performing a vectorized operation on the joined values $(A.\text{val} * B.\text{val})$ to compute intermediate products before applying the `sum()` aggregate function to accumulate the final cell values.

C++ Development Analogue While our high-level evaluation utilizes Python and Pandas to simulate database operations, an equivalent program in C++ would

leverage the Standard Template Library (STL) to execute the relational logic at a bare-metal level. The COO triplets are natively modeled using `std::vector` of a custom `struct` containing `row`, `col`, and `val` members. The sorting phase accomplishes subscript matching using `std::sort` alongside custom lambda comparators to efficiently order matrix A by column and matrix B by row. Finally, the join and aggregation phase executes two-subscript merge logic via raw iterator traversal. To finalize the calculation, intermediate products can be accumulated dynamically via `std::unordered_map` (acting as the `GROUP BY` hash table) or stored in a temporary vector that is subsequently sorted and linearly reduced.

3.2 Database-Driven Multiplication (SQL vs. Relational Python)

When dealing with matrices that scale to the limits of our system memory, we shift to a formalized database workflow, treating the matrix operations as relational queries. In doing so, we prioritize I/O minimization over CPU cycles.

To illustrate these external memory constraints, consider a dimension of $n =$ 1,000,000. A dense 64-bit float vector requires only 8 MB of storage, which trivially fits into main memory. However, the corresponding two-dimensional dense $1M \times 1M$ matrix scales quadratically to $O(n^2)$ space, requiring roughly 8 terabytes of storage. When processing structures of this magnitude, we utilize the ijv (Coordinate) format stored within CSV files as our primary interchange medium. This format is highly natural for SQL databases, as it directly maps to the relational model of (row, column, value) tuples (records). Furthermore, these CSV files effectively act like "primitive SQL tables," allowing us to store and retrieve large-scale sparse data in a database-like manner before streaming it into our Python processing pipeline.

Since standard relational databases usually struggle with the specific scaling of sparse linear algebra, our custom Python workflow is developed to efficiently bridge secondary storage and memory. This is achieved through chunked vectorized ingestion, reading the matrices from CSV files in vectorized chunks instead of line-by-line parsing to prevent memory spikes during the initial load phase. The workflow employs data-parallel partitioning, dividing the data horizontally across available CPU workers so each worker evaluates a discrete subset of rows independently, maintaining a localized memory footprint. Finally, the execution pipeline adheres to a strict three-phase database-style I/O lifecycle: bulk reading the COO data from CSVs into memory, executing parallel relational joins and aggregations, and serializing the resulting sparse coordinates directly back to secondary storage.

3.3 Python Sparse CPU Baseline

When data fits into the available system RAM, we utilize optimized Python-based structures to establish a compute baseline. Rather than allocating a dense, zero-initialized two-dimensional NumPy array—which would immediately trigger an out-of-memory fault for our large-scale matrices—our baseline program leverages Numba's Just-In-Time (JIT) compilation to accelerate a pure COO sort-merge algorithm across available CPU cores.

To accurately measure performance and overhead, the complete execution follows a strict three-phase, chunk-based database I/O workflow. The first phase reads raw Coordinate (COO) format data from CSV files on secondary storage block by block (as chunks), instead of loading the entire matrix into memory beforehand. This avoids dense memory allocations entirely and limits RAM usage. The second phase processes the loaded chunks in RAM using multi-core parallel execution. The core logic matches non-zero elements via a sorted two-pointer merge across the chunks, circumventing the cubic time complexity associated with dense matrix multiplication. The final phase writes the resulting sparse matrix by serializing the computed chunks back to secondary storage in COO format.

3.4 Correctness of Database Algorithm

We establish the correctness of our relational approach by demonstrating that it computes the exact same result as the standard dense matrix algebraic computation ($C'_{i,j} = C_{i,j}$). Because our relational sparse matrices explicitly omit zero-value tuples, the `INNER JOIN` natively acts as a non-zero intersection filter. If an entry is zero, the join fails, adding nothing to the final sum, perfectly mirroring how multiplying by zero does not change an algebraic sum. For any valid non-zero pair, the grouped `SUM` aggregation accumulates the exact dot product $A_{i,l} * B_{l,j}$ for each coordinate (i,j), guaranteeing mathematical equivalence without requiring complex induction bounds.

3.5 Space and Time Complexity

Our relational approaches effectively bypass the performance bottlenecks of dense programs. For matrix multiplication (whether executed via SQL, Pandas, or C++), the sorting and relational matching phases operate in $O(n\log n)$ time, and we deduce the total bound by adding the M intermediate operations. Matrix-vector multiplication relies on similar merge-joins, yielding an $O(n\log n)$ time complexity. Matrix addition, however, evaluates to $O(n)$ time for the merge phase. In all cases, the total space complexity evaluates to $O(k_A + k_B + k_C)$. The storage of the input matrices requires $O(k_A + k_B)$ space, while the resulting output matrix requires $O(k_C)$ space. It is important to notice that the output matrix may be significantly larger than the input matrices.

Theoretical Sparse vs. Dense Acceleration Bounds The asymptotic performance advantage of sparse algorithms over dense implementations is strictly bounded by the matrix $density = k/n^2$. Standard dense matrix multiplication requires $O(n^3)$ floating-point operations. In contrast, the relational sparse algorithm operates in $O(n \log n)$ time, and we deduce the total bound by adding the M intermediate operations. For multiplication (the $O(n^3)$ advantage), substituting the density quotient causes the sparse time complexity to evaluate to $O(d * n^2 \log(d * n^2))$, plus M. For the sparse approach to asymptotically outperform the dense $O(n^3)$ bound, the overhead of sorting and relational matching must remain lower than the skipped zero computations. This dictates that d must be exceedingly small; as d increases, the $O(n\log n)$ sorting term quickly overtakes the arithmetic savings. For addition (the $O(n^2)$ constraint), standard dense matrix addition is an $O(n^2)$ linear operation.

Because the sparse equivalent evaluates to $O(n)$ time for the merge phase, the theoretical crossover point relies on the sparsity factor. Highly optimized dense SIMD instructions will theoretically process n^2 elements faster than a scalar processor can merge n elements unless the sparsity approaches absolute extremes.

Asymptotic Bottleneck Shifts When transitioning from main-memory imperative loops to external-memory relational algebra, the theoretical bottleneck of the system shifts fundamentally. Traditional sparse programs suffer from the scalar accumulation penalty, accumulating intermediate products M via hash maps and incurring an $O(1)$ random memory access per collision. As k scales, these random access penalties degrade cache coherence, severely limiting CPU throughput. This is mitigated through relational vectorization, which replaces iterative scalar loops with blocked, vectorized merge-joins. This applies relational algebra to transform random hash accumulations into sequential memory scans, lowering the constant time factor hidden in the Big-O notation. However, this optimization exposes I/O bound dominance; as the computational phase approaches theoretical optimal efficiency via vectorized joins, the CPU ceases to be the limiting factor. The algorithm's overall time complexity becomes dominated by the I/O bandwidth bounds defined by the sequential read and write speeds of the secondary storage medium, shifting the bottleneck entirely from compute to memory transfer.

3.6 Applications and Target Models

To ground our theoretical framework, it is crucial to highlight the specific models and algorithms that fundamentally rely on large-scale sparse matrix operations. Because our $O(1)$ bounded space complexity allows us to process data linearly rather than quadratically, our algorithms are highly applicable to evaluating social networks and large language models (LLMs), which are inherently super-sparse. In recommender systems, collaborative filtering algorithms, such as Alternating Least Squares (ALS), depend heavily on sparse matrix factorizations. In natural language processing (NLP), algorithms like support vector machines (SVMs) evaluate highly sparse TF-IDF or Bag-of-Words feature matrices.

To demonstrate how these domains utilize our relational framework, we map our evaluated mathematical operations to their physical data science pipeline counterparts:

Matrix-Matrix Multiplication ($A*B$): This operation corresponds to the **Training Phase** in machine machine learning (specifically batch gradient descent) where large datasets are processed simultaneously. In graph analytics, this represents graph transformations and message-passing in Graph Neural Networks (GNNs). For example, computing the 2-hop paths between nodes by squaring the adjacency matrix ($C = A * A$) evaluates to $C_{ij} = \sum_{k=1}^{n} A_{ik} * A_{kj}$. In our database-driven framework, this intermediate variable k is perfectly modeled as a relational join, discovering continuous paths and determining reachability across multiple hops.

Matrix-Vector Multiplication ($A*v$): This operation corresponds to the **Testing/Inference Phase**. For example, computing the prediction $\hat{Y} = \beta X$ in linear regression involves a matrix-vector product. In network analysis, this represents iterative traversal algorithms like PageRank, Breadth-First Search (BFS), or Single Source

Shortest Path (SSSP), where the sparse adjacency matrix is repeatedly multiplied by a state vector representing the current frontier of visited nodes.

Beyond traditional inference and graph traversal, sparse matrix-matrix multiplication (SpMM) is fundamental to **Model Compression and Network Pruning**. Once a dense neural network is pruned (removing less important weights to create sparsity), SpMM allows the hardware to skip multiplying and adding zeros, saving massive memory bandwidth and compute cycles during inference. However, because modern accelerators (like GPUs) are highly optimized for dense operations, relational SpMM only yields real-world inference speedups if the sparsity is extremely high (often $>80\%$) or heavily structured.

4 Experiments

In this section, we present a comprehensive experimental evaluation of our proposed database-inspired algorithms for sparse matrix operations. We begin by detailing the experimental setup, data characteristics, and hardware configurations. We then provide an extensive time comparison between our relational Python approach, traditional SQL database engines, and in-memory baselines. Finally, we evaluate execution performance across both CPU and GPU environments, explicitly identifying the fundamental I/O limitations and bottleneck shifts inherent in external memory execution.

4.1 Experimental Setup and Workloads

Hardware and Software. To properly contrast local server execution with cloud-based hardware acceleration, we evaluated our programs across two distinct hardware environments. CPU-bound benchmarking was performed on a high-performance local server to assess true multiprocessing capabilities (AMD Ryzen 9 8940HX CPU, 16 physical cores, 32 threads, 16GB RAM). To assess massive parallelism in sparse versus dense operations, we utilized a Google Colab instance provisioned with an NVIDIA Tesla T4 GPU (CUDA 13.0, 15GB VRAM). Our code leverages Python 3.13.1, NumPy 2.2.1, SciPy 1.15.0, Pandas, and PyTorch 2.9.1. It is important to justify why this experimental setup was not programmed entirely in C or C++. While Python does not natively possess a faster execution time than compiled languages, it acts as a highly efficient wrapper. When raw computational speed is required, Python libraries such as PyTorch or Pandas hand the COO data off to highly optimized, pre-compiled C/C++ backends under the hood. In doing so, we gain the bare-metal performance of C/C++ without having to write complex manual memory management code, allowing us to seamlessly evaluate relational logic at scale. To verify implementation correctness, code logic was successfully asserted against NumPy's dense C libraries on small scales. To ensure a fair comparative analysis, our PostgreSQL benchmarks explicitly include the I/O time required to load matrices from CSV files into the DBMS, providing an "end-to-end" view of database transition costs.

Data. In our evaluation, we define a "sparse" dataset as having a density of 1% or less, and an "ultra-sparse" dataset as having a density of 0.01% or less (ranging from 1 in 100,000 to 1 in 1,000,000 non-zero entries). Maintaining our established $O(1)$

non-zero per row assumption (specifically fixing the ratio $\frac{k}{n}=10$), we evaluate three-dimensional scales that fall strictly within these bounds: $n \in \{10^4, 10^5, 10^6\}$. Workloads encompass static operations ($C = A * B$) and dynamic updates ($A_{t+1} = A_t + \Delta A$). While benchmarked on square matrices to maintain scaling consistency, our relational operators natively map to rectangular domains without modification.

4.2 Time comparison: Python vs SQL

By programming matrix multiplication as a relational query, we compared a custom Python-based approach against a PostgreSQL-driven model. However, we must clarify that our current code evaluated in these experiments is not chunk-based; it loads the entire matrix into memory before processing. To accurately evaluate the impact of different hardware architectures, we explicitly separate our analysis into CPU and GPU execution contexts.

Table 1. Matrix-Vector Multiplication ($A*v$) on CPU (time in secs).

Method	10k	100k	1M
PostgreSQL	1.2	3.0	20.8
Pandas SpMV	0.2	0.8	6.2
Pandas Blocked	0.2	0.7	4.5

Table 2. Execution times (time in secs) for CPU-bound Matrix-Matrix Multiplication ($A*B$) on Local Server (16-Core CPU).

Method	10k	100k	1M
PostgreSQL (No Index)	2.8	16.0	163.8
Python (Pandas)	2.1	21.7	226.1
Python (Numpy COO)	3.7	38.1	2408.8

Table 3. Execution times (time in secs) for CPU-bound Matrix-Matrix Multiplication ($A*B$) on Cloud CPU (Google Colab).

Method	10k	100k	1M
PostgreSQL (No Index)	3.2	36.5	97.6
Python (Pandas)	3.6	45.4	109.0
Python (Numpy COO)	6.4	66.1	N/A

Table 4. Execution times (time in secs) for Ultra-Sparse CPU-bound Matrix Multiplication (Density 0.01%) on Cloud CPU.

Method	1k	10k	100k
PostgreSQL (No Index)	0.00	0.11	38.21
Python (Pandas)	0.30	0.21	48.14
Python (Numpy COO)	3.56	2.17	62.00

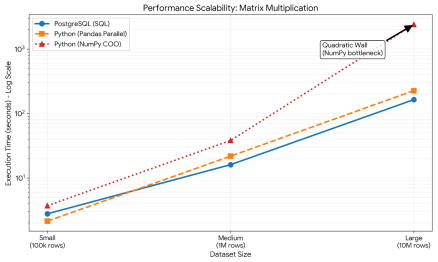

Fig. 1. Performance Scalability Analysis. The log-scale plot highlights the accumulation bottlenecks encountered by the Numba COO program (red dotted line) on large datasets, contrasting with the more linear scaling of the Database/Pandas approaches.

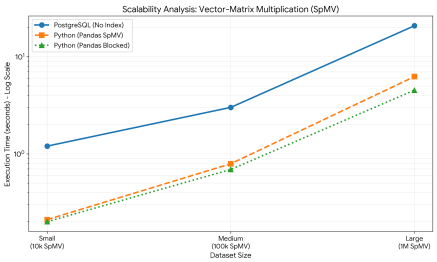

Fig. 2. Scalability Analysis of Vector-Matrix Multiplication ($C = A * v$). The plot demonstrates that Python-based approaches (Pandas and Blocked) significantly outperform PostgreSQL computationally, maintaining low latency even as dataset size increases.

CPU Execution Evaluation As detailed in Tables 2, 3, and 4 and Figure 1, the transition to a relational Pandas model executes the same workload in a fraction of the time as the Numba COO program, effectively bypassing the scalar iteration bottleneck. Computation time dropped by an order of magnitude, demonstrating that the relational model scales effectively. For the SpMV workload ($A * v$) shown in Table 1, the Python approaches demonstrate a significant performance advantage over the database engine by leveraging localized grouping and aggregation, bypassing PostgreSQL's transactional broadcast overhead. Most importantly, contrary to initial expectations, the compute phase no longer dominates the execution time in the optimized relational Python model; as parallel joins accelerate processing, sequential I/O (Read and Write) rises to consume the vast majority of execution time, firmly shifting the bottleneck to secondary storage bandwidth. Additionally, we observed that when running the experiment against a dense dataset on Google Colab, the program hangs during Phase 2 because it triggers a massive memory bottleneck.

Table 5. Execution times (time in secs) for GPU-bound Matrix-Matrix Multiplication ($A * B$).

Environment & Method	10k	100k	1M
Cloud GPU (Google Colab)			
PostgreSQL (No Index)	2.3	32.6	26.9
PyTorch (Sparse_COO)	2.0	19.1	42.0

GPU Execution Evaluation In the cloud environments, PyTorch's GPU acceleration unleashes thousands of CUDA cores via cuSPARSE. As detailed in Table 5, this results in significant speedups for smaller datasets. However, while the compute phase is drastically reduced, the GPU approach still struggles with I/O scaling at the highest limits.

4.3 Bottlenecks: Format Conversion and I/O Limitations

While modern GPUs offer massive parallelism for dense arithmetic, interfacing them with a database-style external memory workflow uncovers a severe format translation bottleneck. GPU hardware threads strictly demand contiguous memory access, forcing underlying libraries like cuSPARSE to require Compressed Sparse Row (CSR) formatting. However, external tabular data inherently streams in Coordinate (COO) format. During our $50{,}000 \times 50{,}000$ matrix benchmark, preparing the relational input (COO-to-CSR) and extracting the computed result (CSR-to-COO) back to secondary storage required 0.0394 seconds purely for memory reorganization, while the actual arithmetic compute only took 0.0398 seconds. Nearly half (49.74%) of the GPU's total processing time was wasted on formatting penalties. Our Python-based relational join approach operates natively and continuously on the COO tabular stream, structurally avoiding this penalty. Furthermore, it is critical to note that standard SQL relational database engines cannot natively interface with or execute on GPU hardware.

Ultimately, the viability of the database-driven approach hinges directly on the sparsity of the data. For large-scale sparse matrices executed on CPUs, the workflow is highly efficient because CPU computational cycles dominate the relatively small CSV footprint. However, when executing those same matrices on GPU accelerators, the arithmetic completes so rapidly that sequential CSV I/O scaling becomes the insurmountable dominant factor. Finally, for dense matrices, the database approach is fundamentally catastrophic regardless of hardware platform; the I/O operations strictly dominate execution, as parsing millions of relational tuples from storage is orders of magnitude slower than executing standard dense SIMD mathematical instructions.

5 Related Work

Research in sparse matrix operations fundamentally diverges into two main directions: High-Performance Computing (HPC) and Database/Big Data approaches.

HPC represents the fastest and by far the most widely used direction. Traditional HPC relies on compressed structures like Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) to optimize memory footprints. Bell and Garland [2] mapped these representations to throughput-oriented architectures like GPUs, while Buluç and Gilbert [3] established distributed sparse matrix multiplication using two-dimensional block layouts. However, these main-memory techniques encounter severe serialization and subscripting overheads when dataset sizes exceed physical RAM constraints.

Database and Big Data approaches to overcome these strict size limitations, a second direction of research leverages relational databases and distributed frameworks. For instance, Luo et al. [10] and Ordonez [11] demonstrated executing matrix operations and statistical models natively within an RDBMS via SQL. In database literature, researchers have sought to unify matrix operations with relational database principles. Declarative frameworks like SystemML [5] and SciDB [12] map high-level linear algebra directly to distributed engines, leveraging established query optimization strategies.

The matrix multiplication query can also be interpreted as a functional dataflow computation. The join on the shared matrix dimension corresponds to the distributed

shuffle and grouping step in MapReduce, while the multiplication of matched entries corresponds to the map phase, and the summation of partial products corresponds to the reduce phase. Although execution environments differ in how data is staged before computation, the core logic remains a sequential join followed by grouped aggregation over relational tables. This demonstrates how sparse matrix multiplication can be represented declaratively in SQL while also possessing a natural MapReduce and functional-programming interpretation. Unlike Hadoop/MapReduce-based graph-mining systems such as PEGASUS [7], whose GIM-V execution can incur shuffle, sorting, network-transfer, and disk-I/O overheads, our strict sequential-I/O sort-merge join approach is designed to reduce these latency costs by processing sparse COO-style tabular streams directly, complementing recent developments in parallel SQL/PGQ query processing for graphs [14].

Despite these relational integrations and advanced algorithmic optimizations for complex queries, such as large sparse matrix chain multiplications [9], a critical literature gap remains regarding the I/O bandwidth bottlenecks inherent in external memory execution. Irony et al. [6] established formal communication lower bounds for memory-hierarchy transfers, while Ali and Wrembel [1] and Varghese et al. [13] highlight the necessity of I/O-aware pipelines to manage secondary storage latency. Furthermore, the sequential bottleneck of ingesting massive plain-text formats (like CSVs) remains a barrier for analytical systems [4], especially given the complexity of semantic matching over matrix-style tables [8]. While the theoretical potential of relational linear algebra is well-documented, current literature lacks an end-to-end framework that seamlessly bridges high-level data science workflows (e.g., Python) with optimized, database-style I/O ingestion for massive, ultra-sparse matrices.

6 Conclusions

Our evaluation establishes that relational database-inspired algorithms offer a highly scalable alternative to traditional main-memory approaches for processing massive sparse matrices. Based on our experimental results, the relational Pandas model proved exceptionally effective for both matrix-matrix and matrix-vector operations. It consistently outperformed traditional iterative baselines by an order of magnitude and bypassed the heavy transactional broadcast overhead inherent to standard relational database engines like PostgreSQL. By operating directly on the native Coordinate (COO) tabular stream, our approach also structurally avoided the severe format conversion penalties that plagued GPU accelerators, which wasted nearly half of their processing time translating between COO and Compressed Sparse Row (CSR) formats. However, our evaluation also revealed significant limitations. The viability of the database-driven approach is strictly bounded by data sparsity. While the external memory workflow is highly efficient for large-scale ultra-sparse datasets where CPU cycles dominate, it becomes a severe limitation as matrix density increases. For dense datasets, the approach is fundamentally catastrophic, as the massive volume of secondary storage I/O operations entirely overwhelms the system, leading to memory faults and stalled execution. Furthermore, while GPUs offer massive arithmetic parallelism for smaller

datasets, their rigid memory requirements and inability to natively interface with external database streams limit their applicability in this specific architectural workflow.

To address remaining algorithmic constraints and further extend this research, our future work will strictly focus on several major directions. The first major direction involves extending our relational framework to efficiently process rectangular matrices, which are highly prevalent in feature extraction and collaborative filtering datasets. The second major direction will explore hybrid execution models where one matrix (such as a machine learning model) resides entirely in main memory, while the other matrix (such as training data or model summaries) streams continuously from secondary storage. The third major direction focuses on generalizing our architecture from two-dimensional matrices to multi-dimensional sparse tensors, specifically incorporating a time dimension to evaluate temporal networks and dynamic graph evolution. Finally, to further accelerate our external memory workflows and mitigate format conversion overheads, we plan to explore advanced GPU parallelism. Specifically, we need to consider utilizing highly optimized dense multiplication in main memory between aligned sparse vectors mapped directly onto a GPU, allowing these highly scalable computations to be effectively applied within the complex training and message-passing phases of Graph Neural Networks (GNNs). Additionally, a crucial theoretical direction involves transitioning our architecture into a fully external memory algorithm. By fully externalizing the intermediate computations, we aim to eliminate our strict reliance on the $O(1)$ non-zero entries per row assumption, potentially relaxing this constraint to efficiently process matrices that scale toward $O(n\log n)$ density bounds.

AI Acknowledgement We utilized Gemini for language editing, restructuring of arguments, and assistance with LaTeX and SQL formatting during manuscript preparation. All research contributions, experimental work, and final content decisions are entirely our own. We have verified all references and take full responsibility for the content of this paper.

References

1. Ali, S.M.F., Wrembel, R.: Framework to optimize data processing pipelines using performance metrics. In: Big Data Analytics and Knowledge Discovery - 22nd International Conference, DaWaK 2020. vol. 12393, pp. 131–140. Springer (2020)
2. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA. ACM (2009)
3. Buluç, A., Gilbert, J.R.: Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. SIAM J. Sci. Comput. **34**(4) (2012)
4. Fathollahzadeh, S., Boehm, M.: GIO: generating efficient matrix and frame readers for custom data formats by example. Proc. ACM Manag. Data **1**(2), 120:1–120:26 (2023)
5. Ghoting, A., Krishnamurthy, R., Pednault, E.P.D., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y., Vaithyanathan, S.: Systemml: Declarative machine learning on mapreduce. In: Proceedings of the 27th International Conference on Data Engineering, ICDE 2011. pp. 231–242 (2011)
6. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distributed Comput. **64**(9), 1017–1026 (2004)

7. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: mining peta-scale graphs. Knowl. Inf. Syst. **27**(2), 303–325 (2011)
8. Li, H., Yang, Q., Cao, Y., Zhou, G., Luo, P.: Semantic matching over matrix-style tables in richly formatted documents. In: Hartmann, S., Küng, J., Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) Database and Expert Systems Applications - 31st International Conference, DEXA 2020, Bratislava, Slovakia, September 14-17, 2020, Proceedings, Part I. pp. 369–384. Lecture Notes in Computer Science, Springer (2020)
9. Lin, C., Luo, W., Fang, Y., Ma, C., Liu, X., Ma, Y.: On efficient large sparse matrix chain multiplication. Proc. ACM Manag. Data **2**(3), 156 (2024)
10. Luo, S., Gao, Z.J., Gubanov, M.N., Perez, L.L., Jermaine, C.M.: Scalable linear algebra on a relational database system. IEEE Trans. Knowl. Data Eng. **31**(7), 1224–1238 (2019)
11. Ordonez, C.: Statistical model computation with udfs. IEEE Trans. Knowl. Data Eng. **22**(12), 1752–1765 (2010)
12. Stonebraker, M., Brown, P., Zhang, D., Becla, J.: SciDB: A Database Management System for Applications with Complex Analytics. Computing in Science and Engineering **15**(3), 54–62 (2013)
13. Varghese, R., Quadir, H., Bellatreche, L., Ordonez, C.: Accelerating Python code with parallel I/O. In: Database and Expert Systems Applications - 36th International Conference, DEXA 2025. pp. 360–366 (2025)
14. Yamasaki, K., Masuda, T., Amagasa, T.: Parallel and distributed SQL/PGQ query processing for property graphs. In: Leung, C.K., Dignös, A., Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) Big Data Analytics and Knowledge Discovery - 27th International Conference, DaWaK 2025, Bangkok, Thailand, August 25-27, 2025, Proceedings. pp. 167–181. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-032-02215-8_12, https://doi.org/10.1007/978-3-032-02215-8_12