Multi-Node Multi-GPU Datalog

Ahmedur Rahman Shovon
ashov@uic.edu
University of Illinois, Chicago
Chicago, Illinois, USA

Yihao Sun
ysun67@syr.edu
Syracuse University
Syracuse, New York, USA

Thomas Gilray
thomas.gilray@wsu.edu
Washington State University
Pullman, Washington, USA

Kristopher Micinski
kkmicins@syr.edu
Syracuse University
Syracuse, New York, USA

Sidharth Kumar
sidharth@uic.edu
University of Illinois, Chicago
Chicago, Illinois, USA

Abstract

Datalog, a declarative logic programming language that operates bottom-up, has experienced increasing popularity due to its natural handling of recursive queries. Its applications span diverse fields, including graph mining, program analysis, deductive databases, and neuro-symbolic reasoning. While Datalog shares similarities with SQL in using relational algebra kernels, it uniquely employs iterative execution until reaching a fixed point to support recursion. Current Datalog engines like SLOG, LogicBlox, and Soufflé work well with multi-core and multi-threaded systems, but none have yet tackled multi-node, multi-GPU architectures. Our research addresses this gap by developing the first multi-GPU, multi-node Datalog engine. This advancement is particularly significant for high-performance computing (HPC) systems, which typically feature multiple GPUs per node. Our implementation combines MPI for inter-node communication with CUDA for GPU parallelization, enabling the processing of massive datasets in real time. We have created novel data-parallel implementations of core relational algebra operations (join), while also optimizing deduplication and tuple materialization. To handle iterative execution requirements, we have developed two novel GPU-accelerated methods for non-uniform all-to-all data exchange. Evaluating on Argonne National Lab's Polaris supercomputer demonstrated our engine's effectiveness, achieving performance improvements of up to 31.9× against state-of-the-art multi-node Datalog engine.

'ICS 2025', June 9–11, 2025, Salt Lake City, USA

ACM ISBN 000-0-0000-0000-0/00/00
https://doi.org/XXXXXXX.XXXXXXX

Keywords

Datalog, Multi-GPU, Analytic Databases

1 Introduction

Datalog is a declarative logic programming language notable for its elegant handling of recursive queries. Its power is exemplified by its ability to express complex algorithms succinctly–for instance, computing a graph's transitive closure for path finding requires merely two lines of Datalog code. Similarly, other graph algorithms like same graph generation, connected component analysis, and single-source shortest path calculations can be implemented in just two to three lines of Datalog code [37]. This remarkable expressiveness has led to Datalog's adoption across diverse domains, from bioinformatics and graph mining [16, 30, 32, 35] to program analysis [3, 6, 11, 12], and neuro-symbolic reasoning [26].

Datalog operates by translating queries into relational algebra operations–such as joins, projections, and unions. The system architecture consists of two main components: a frontend compiler that transforms Datalog queries into iterative relational algebra kernels and a backend that executes these kernels [21] until a fixed-point is reached. This design enables a powerful combination of high-level expressiveness through Datalog's syntax while achieving performance through parallel implementation of the relational algebra kernels. Several engines have implemented this approach: Soufflé [20] leverages OpenMP for multi-threaded CPU processing, SLOG [14] employs MPI for distributed CPU computation, GPUJoin [34] and GPULog [38] utilize CUDA for GPU acceleration.

Among parallel Datalog engines, the GPU-based approach significantly outperforms others, with reported speedups exceeding 10× in complex tasks such as program analysis [38]. This advantage comes from Datalog's inherently data-intensive and memory-bound nature, where each iteration requires deduplication, indexing, and aggregation of

large volumes of generated data. Datacenter-grade GPUs provide much higher memory bandwidth than CPUs of the same class; for instance, the Nvidia H100 reaches 3.35 TB/s [28], whereas the high-end AMD Zen 5 achieves only 576 GB/s [2].

The performance advantages of GPU acceleration are predominantly realized in single-node configurations. When scaling GPU-based methodologies across parallel nodes, a fundamental challenge emerges due to the substantial disparity between the high-bandwidth memory (HBM) available for intra-GPU communication and the relatively constrained bandwidth of inter-node networks. To mitigate this potential bottleneck, multi-node multi-GPU Datalog execution engines must implement advanced workload partitioning strategies that substantially reduce communication requirements while simultaneously incorporating relational algebra operators specifically optimized for the iterative computational patterns characteristic of Datalog evaluation.

In this paper, we introduce MNMGDATALOG, the first multi-node, multi-GPU Datalog engine designed to address the above mentioned challenges. To the best of our knowledge, ours is the first Datalog engine that fully harnesses the potential of modern GPU-based supercomputers. Our engine employs a radix-hash-based data partitioning scheme to balance computation across GPUs while minimizing data exchange. We design novel distributed relational algebra operators that consider both data partitioning strategies and the SIMT nature of GPUs [29]. Our framework relies on non-uniform all-to-all data exchanges to facilitate fixed point iteration. We explore two different implementations of all-to-all data exchanges that also leverage the computational capabilities of the GPU. In addition to core features found in other GPU-based Datalog engines, we implement recursive aggregation, a widely used Datalog feature, and scale it efficiently across multiple nodes. We evaluate MNMGDATALOG's performance up to 32 NVIDIA A100 GPUs on the Polaris supercomputer, benchmarking its performance against state-of-the-art CPU and GPU-based engines on diverse graph analytic queries. Our main contributions are summarized as follows:

- We present a radix-hash-based data partitioning strategy for Datalog, optimized for indexing and iterative computation.
- We implement CUDA-aware non-uniform all-to-all exchanges for tuple materialization to facilitate iterative relational algebra.
- We implement and scale recursive aggregation on GPU.
- On a single GPU, MNMGDATALOG achieves up to 7× speedup over GPULog, and up to 33× over Soufflé.
- In a multi-node, multi-GPU setting, MNMGDATALOG outperforms the state-of-the-art HPC-based engine SLOG by up to 31.9×.

2 Declarative Analytics using Datalog

Datalog operates through a lightweight bottom-up evaluation approach and is widely used in deductive database systems [7, 8, 10]. A Datalog program consists of an *extensional database* containing explicit input facts and an *intensional database* of derived facts inferred through rules [5]. These rules are expressed as first-order Horn clauses, where each rule takes the form:

$$Head \leftarrow Body_1, Body_2, \ldots, Body_n$$

The head contains a single predicate atom which represents the inferred fact, while the body specifies the conditions for its derivation using a set of predicate atoms. The implication symbol $\leftarrow$ connects the head with the body. Commas in the body represent logical AND ($\wedge$) that performs a *join* operation between the predicate atoms.

Datalog excels at handling recursive queries through *fixed-point evaluation*, where rules are repeatedly applied until no new facts can be derived. For instance, the following rules can be used to derive the transitive closure of an input graph, which finds all reachable paths from each node:

$$\begin{aligned} \texttt{TC}(x, y) &\leftarrow \texttt{Edge}(x, y). \\ \texttt{TC}(x, z) &\leftarrow \texttt{TC}(x, y), \texttt{Edge}(y, z). \end{aligned}$$

The first rule establishes that a direct edge from x to y implies reachability. The second rule, which is recursive, extends this reachability by chaining existing paths: if x can reach y and there is an edge from y to z, then x can also reach z. This recursive expansion is achieved through a *join* operation between the TC and Edge relations, where previously derived reachability facts from TC are joined with new edges from Edge to infer additional paths. This process continues iteratively until no new paths can be inferred. Such recursive capabilities allow Datalog to efficiently solve problems like transitive closure, connected components, same generation, and other queries that traditional SQL struggles to handle. Though traditional SQL supports recursive queries through common table expressions (CTEs), Datalog's expressive syntax and bottom-up evaluation make recursive queries simpler and more efficient by automatically iterating until a fixed point is reached, without requiring explicit control structures.

Semi-naïve evaluation. Modern datalog engines achieve performance improvements through the semi-naïve evaluation [1], an incremental evaluation technique. Our framework, MNMGDATALOG, likewise implements this optimization technique, which improves the efficiency of iterative queries by exclusively utilizing only the newly derived facts during each successive iteration. This strategy avoids redundant computation by ensuring that only newly derived facts are used to infer additional facts in the iterative process.

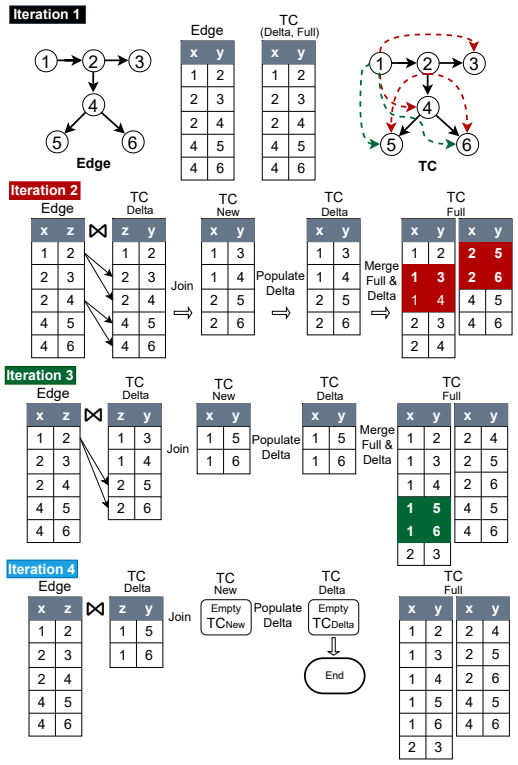

Figure 1: Iterations of transitive closure computation.

In semi-naïve evaluation, non-static relations (such as TC in path-finding applications) are strategically decomposed into three distinct components: (1) `full`, which encompasses all facts discovered prior to the most recent iteration; (2) `delta`, containing exclusively those facts identified during the immediately preceding iteration; and (3) `new`, which stores facts newly derived during the current computational iteration. Throughout each iteration, join operations are selectively applied using only the `delta` component of a relation, with resultant tuples stored in the `new` component. Upon iteration completion, the algorithm executes a three-phase transition process: it transfers all tuples from `delta` into the `full` relation, exchanges pointers between `delta` and `new` to prepare `delta` for the subsequent iteration, and reinitializes `new` to accommodate upcoming derivations. Figure 1 provides a visual representation of this iterative execution process for transitive closure rules under the semi-naïve evaluation framework.

The top left presents the input graph, while the top right depicts the fully computed TC. In the first iteration, the TC relation is initialized using the direct edges from the Edge relation. Since TC is initially empty, the recursive rule does not contribute any new facts at this stage. Both the `Delta` and `Full` versions are set to be identical to TC, ensuring that all direct connections are established before applying recursive expansions in subsequent iterations. In the second iteration, the recursive rule computes `New` := `Edge` $\bowtie$ `Delta`, yielding $\{(1,3),(1,4),(2,5),(2,6)\}$. Since these tuples are not present in the existing `Full` version, they are added to `Full` (highlighted in red) and also stored in `Delta` for the next iteration. In the third iteration, the join produces unique set of tuples $\{(1,5),(1,6)\}$, which are merged into `Full` (highlighted in green) and retained in `Delta`. In the fourth iteration, `Edge` $\bowtie$ `Delta` produces no new tuples, leaving `Delta` empty and signaling fixpoint termination. This stepwise expansion optimizes TC computation by restricting joins to newly derived facts, eliminating redundant computation while ensuring correctness.

Recursive Aggregation. Recursive aggregation extends standard Datalog semantics by allowing aggregate functions such as `MIN`, `MAX`, `SUM`, and `COUNT` to be applied dynamically during recursive evaluation. This feature is particularly useful in graph algorithms, including connected components, shortest paths, and PageRank, where values must be iteratively propagated rather than computed post-fixpoint using stratification [1].

One such application is the Weakly Connected Components (WCC) problem, where recursive aggregation enables the efficient propagation of component representatives. The WCC query can be formulated in Datalog as follows:

$$\begin{aligned} \texttt{WCC}(n, n) &\leftarrow \texttt{Edge}(n, \_). \\ \texttt{WCC}(y, \texttt{MIN}(z)) &\leftarrow \texttt{WCC}(y, z),\ \texttt{Edge}(x, y). \end{aligned}$$

The first rule initializes each node as its own component, while the second rule propagates the smallest representative node ID across connected nodes using the `MIN` aggregate. Unlike traditional Datalog engines that materialize all possible component memberships, recursive aggregation ensures that only the minimal component representative is maintained, reducing both space complexity and redundant computations.

3 Challenges and requirements

This section outlines the critical requirements for building a multi-node multi-GPU datalog engine. We focus on three fundamental components necessary to achieve scalable performance: workload partitioning, data representation, and inter-node data exchange.

3.1 Workload partitioning

Parallelizing algorithms necessitates identifying an appropriate partitioning axis for workload distribution. Two fundamental approaches exist for problem partitioning: model/task-level partitioning and data-level partitioning. Datalog programs inherently support both paradigms. While task parallelism is program-dependent, allowing complex Datalog programs to be decomposed into independent, concurrently executable task groups – our research exclusively addresses data-level parallelism. We propose a data-parallel framework that effectively distributes the computational workload of Datalog programs across multiple GPUs. This approach requires strategic partitioning of all relations within the Datalog program and subsequent allocation of these partitioned segments across available GPUs.

Conventional GPU-accelerated algorithms are engineered primarily for dense computational patterns, exemplified by matrix multiplication operations. These dense workloads facilitate uniform partitioning into equal-sized computational units, thereby optimizing memory bandwidth utilization and cache efficiency. This characteristic extends advantageously to parallel computing environments, enabling straightforward and balanced data distribution across multiple GPU devices. In contrast, Datalog computation exhibits inherent irregularity and sparsity, as the relations involved in Datalog programs vary considerably in size, characteristics, and topological properties. This intrinsic heterogeneity renders efficient data partitioning across multiple GPUs substantially more complex.

Naive approaches that uniformly distribute relations across available GPUs prove inadequate, as effective partitioning must specifically accommodate the fundamental operations underlying Datalog execution– such as low-level relational algebra kernels including joins, unions, projections, and other tasks like deduplication and merging procedures. In Figure 1, when distributing Edge relation with tuples (1,2), (2,3), (2,4), (4,5), and (4,6) across two GPUs, a join-preserving approach must ensure that all tuples with identical join keys—such as those beginning with "2"—are allocated to the same GPU. This locality-preserving allocation is essential for ensuring that join operations, which in this case occur on the first column, can be performed efficiently within a single GPU without cross-device communication. A naive uniform distribution strategy that allocates an equal number of tuples (e.g., three tuples per GPU) would compromise this crucial locality property. While such an approach might achieve nominal data balance across GPUs, it inevitably leads to highly imbalanced computational workloads during execution and introduces significant inter-GPU communication overhead. To address this distribution challenge, we implement a hash-based partitioning methodology wherein relational data is systematically distributed across all available GPUs according to the hash value of the join column (detailed in Section 4.1). Thus, the first key challenge we propose to address in this paper is designing a data partitioning strategy that respects Datalog computation while minimizing computation, memory, and communication overhead.

3.2 Data representation

A prerequisite for constructing a scalable multi-node, multi-GPU Datalog execution engine is the optimization of single-GPU performance. Fundamental to achieving peak single-GPU efficiency is the underlying data representation — specifically, how relational data is organized and stored within GPU memory. The critical design challenge lies in developing data structures that simultaneously achieve multiple performance objectives: minimizing memory footprint, enabling high-throughput data retrieval operations (essential for join execution), and effectively supporting auxiliary operations such as deduplication. These memory-resident data structures form the foundation upon which the entire distributed computation framework depends, directly influencing overall system scalability and performance characteristics.

For efficient lookup operations, hash tables represent the predominant data structure upon which hash join algorithms are constructed. Conventional CPU-oriented hash join implementations frequently utilize hash tables built on linked-list architectures. However, these structures demonstrate poor performance characteristics when deployed on GPU architectures, as pointer-chasing operations inherently generate non-coalesced memory access patterns, resulting in significant latency penalties. To overcome this architectural limitation, our implementation employs a specialized open-addressing hash map with linear probing techniques. This design modification substantially optimizes memory access patterns and enhances overall execution performance on GPU hardware. A comprehensive detail of this implementation approach, including performance characteristics and design considerations, is presented in Section 4.

Historically, GPU-based Datalog execution engines have been underused primarily due to their insufficient support for seminaïve evaluation strategies, resulting in substantial computational redundancy– a limitation exemplified by the first GPU Datalog implementation, GPUDatalog [25]. Consequently, beyond facilitating efficient join operations, our proposed data representation must systematically eliminate duplicate computations throughout all phases of query evaluation. The data structures must not only enable high-performance join processing but also provide mechanisms to avoid computational redundancy. In numerous recursive

Datalog queries, such as those computing transitive closure or identifying connected components, the inner relation frequently remains invariant across iterations. By precomputing and persisting this static relation in an optimized storage format, the join operation can selectively process only newly derived tuples. This targeted approach substantially reduces unnecessary memory operations and markedly enhances computational throughput.

3.3 Efficient communication

Relational algebra (RA) kernels—including join and other operations—executed locally within individual GPUs generate new tuples that typically serve as input for subsequent iterations of the semi-naive evaluation process. However, these newly generated tuples do not necessarily belong to the GPU/process where they were originally produced. To materialize the newly generated facts and consequently to facilitate iterative parallel relational algebra execution, processes must participate in a non-uniform all-to-all inter-process exchange of generated tuples to their appropriate destination (GPU). The materialization of a tuple in an output relation involves hashing its join column to identify the target GPU, followed by transmission to that specified GPU. Since tuples produced during the local computation phase may each correspond to arbitrary GPUs in the output relation, an all-to-all communication phase becomes necessary to redistribute these output tuples to their managing processes. Due to inherent variations in both the number of tuples generated across different processes and their destination distributions, the all-to-all communication phase exhibits a fundamentally *non-uniform* characteristic. We explain the need for a *non-uniform all-to-all* run using a real example.

In Figure 1, consider performing transitive closure of the Edge relation using two GPUs with hash-based data distribution on the first column. Under this scheme, *GPU 0* stores $\{(2,3),(2,4)\}$ and *GPU 1* holds $\{(4,5),(4,6),(1,2)\}$ based on hashing on the first column. During TC computation, a join operation is performed, followed by a projection that eliminates the common column, resulting in newly derived tuples after the first iteration: *GPU 0* produces $\{(1,3),(1,4)\}$, while *GPU 1* derives $\{(2,5),(2,6)\}$. Since the projection step eliminates the join column, a global redistribution of all new tuples is necessary. This necessitates all-to-all communication, where each GPU sends and receives derived tuples according to the hash partitioning scheme, ensuring that subsequent join operations are performed locally with minimal redundant computation. Without an optimized communication strategy, this redistribution introduces significant overhead, limiting scalability. An efficient communication mechanism must address three key challenges: structured data movement, minimal memory overhead, and scalable GPU-to-GPU communication.

4 MNMGDATALOG: multi-node multi-GPU Datalog

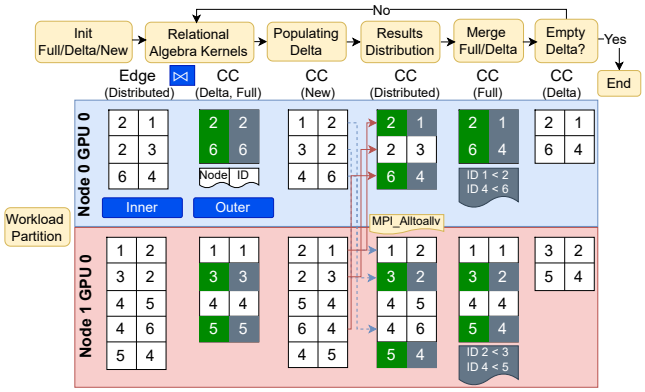

Figure 2: First iteration of semi-naïve evaluation with local aggregation on Weakly Connected Component (WCC) query using MNMGDATALOG.

This section describes the implementation of MNMGDATALOG, the first multi-node, multi-GPU Datalog engine. Figure 2 provides a structural overview of our engine, illustrating the execution flow of the first iteration of Weakly Connected Components (WCC) query on a multi node multi GPU setup with semi-naïve evaluation strategy. It links directly to the key components of our implementation: workload partitioning with hash-based data distribution (Sec 4.1), efficient communication for distributed result propagation (Sec 4.2), and GPU-optimized relational algebra kernels (Sec 4.3).

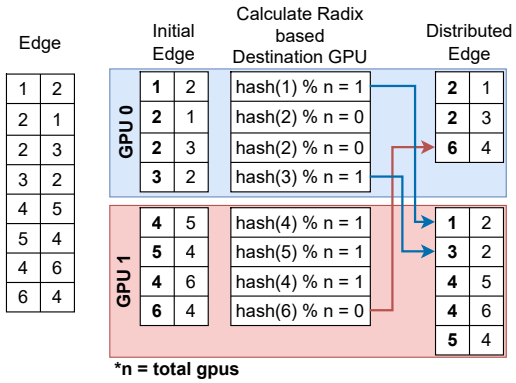

Figure 3: Data distribution based on radix based hashing technique within the available GPUs

Algorithm 1 Sorting-Based buffer preparation and All-to-All communication

Input: Local GPU buffer D, total GPUs R (1 MPI rank per GPU)
Output: Distributed GPU buffer *receive_data*
for each tuple *(key, value)* in D **parallel do**
    row_mapping ← *get_rank(key, R)*
end for
StableSortByKey(*row_mapping, D*)
(unique_rank, send_count) ← ReduceByKey(*row_mapping, D*)
ExclusiveScan(*send_count, send_displacement*)
receive_count ← `MPI_Alltoall` on *send_count*
ExclusiveScan(*receive_count, receive_displacement*)
if CUDA-aware MPI is supported **then**
    receive_data ← `MPI_Alltoallv` on *(D, send_displacement, receive_displacement)*
else
    Copy D to CPU buffer *send_data_host*
    receive_data_host ← `MPI_Alltoallv` on *(send_data_host, send_displacement, receive_displacement)*
    Copy *receive_data_host* to GPU buffer *receive_data*
end if
Return: *receive_data*

Algorithm 2 Two-Pass buffer preparation and for All-to-All communication

Input: Local GPU buffer D, total GPUs R (1 MPI rank per GPU)
Output: Distributed GPU buffer *receive_data*
for each tuple *(key, value)* in D **parallel do** ▷ First pass
    destination_rank ← *get_rank(key, R)*
    AtomicAdd(*send_count[destination_rank]*, 1)
end for
ExclusiveScan(*send_count, send_offset*)
Copy *send_offset* to *send_displacement*
for each tuple *(key, value)* in D **parallel do** ▷ Second pass
    destination_rank ← *get_rank(key, R)*
    position ← AtomicAdd(*send_offset[destination_rank]*, 1)
    send_data[position] ← *(key, value)*
end for
receive_count ← `MPI_Alltoall` on *send_count*
ExclusiveScan(*receive_count, receive_displacement*)
if CUDA-aware MPI is supported **then**
    receive_data ← `MPI_Alltoallv` on *(send_data, send_displacement, receive_displacement)*
else
    Copy *send_data* to CPU buffer *send_data_host*
    receive_data_host ← `MPI_Alltoallv` on *(send_data_host, send_displacement, receive_displacement)*
    Copy *receive_data_host* to GPU buffer *receive_data*
end if
Return: *receive_data*

4.1 Hash based data distribution

Inspired by the classic parallel processing algorithm GRACE Hash Join [13] in distributed RDBMS, and the distributed Datalog processing layout BPRA [22], mnmgDatalog creates local data storage in each GPU's VRAM and partitions each relation based on the join columns. We illustrate this process using the partitioning of the *edge* relation, described in Section 2.

Figure 3 shows how the *edge* relation is distributed in mnmgDatalog on two-GPU devices located on two nodes. The assignment of GPUs for each tuple in the *Edge* relation is determined by applying a hash function to the second first column of the tuple and taking the result modulo the total number of GPUs, which in this case is 2. Assuming the hash function used is the identity function, which hashes an integer value to itself, all tuples with an odd value in the join column are assigned to Node 1, GPU 0. In contrast, tuples with an even value in the join column are assigned to node 0, GPU 0. This partitioning strategy guarantees that tuples sharing the same join column value are co-located on the same GPU. As a result, the join computation can be performed locally on each GPU once all relations have been partitioned, provided that all relations are partitioned in this manner, significantly reducing the need for cross-GPU communication and improving computational efficiency.

4.2 Data communication

As established in Section 3.3, our framework necessitates non-uniform all-to-all data exchanges to materialize newly generated tuples during each iteration of the fixed-point computation loop. Within the MPI programming model, non-uniform all-to-all communication is typically implemented using the `MPI_Alltoallv` collective operation. This function

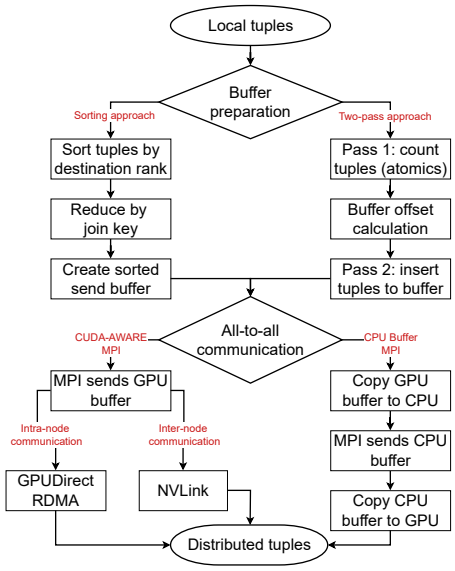

Figure 4: Communication phases of mnmgDatalog

requires all processes to participate synchronously, with the first argument specifying a contiguous buffer containing the concatenated data segments destined for all participating processes. To correctly interpret this buffer during transmission, the function requires supplementary offset and count arrays that precisely delineate the boundaries of individual data segments targeted to specific processes. Consequently, prior to invoking these MPI collective operations, the system must construct the all-to-all send buffer and calculate accurate offsets for the data segments destined for each GPU process within the consolidated buffer. The preparation of this buffer in the required format can incur significant computational overhead. To address this challenge, we leverage the parallel processing capabilities of GPUs to efficiently prepare these communication buffers. MNMGDATALOG implements and comparatively evaluates two distinct buffer preparation methodologies—sorting-based preparation and two-pass preparation—each offering different performance characteristics under varying workload conditions outlined in Figure 4.

The sorting-based approach, outlined in Algorithm 1. This method begins by copying all data into the send buffer and computing the destination GPU rank for each tuple using the data partitioning mechanism described in the previous section. The tuples in the send buffer are then sorted by their associated rank numbers, which automatically groups all tuples destined for the same rank together. Next, a histogram of the rank numbers in the send buffer is computed to determine the data offsets for each rank. The advantage of this approach is that all the operations involved, such as sorting and histogram computation, can be efficiently accelerated on GPUs using well-established algorithms [31, 36]. These algorithms are specifically designed to maximize GPU memory bandwidth utilization. This approach minimizes memory fragmentation, but incurs additional computational overhead due to sorting.

An alternative method, the two-pass buffer preparation technique described in Algorithm 2 eliminates the need for sorting by performing a two-step counting and writing process. CUDA kernels are used to directly count the number of tuples destined for each rank. The first kernel pass scans the input tuples and uses atomic operations to track the send count for each rank. The second kernel pass prepares the send buffer based on these counts by writing the tuples to their respective positions in memory. This approach eliminates the need for sorting, but the resulting buffers may be more fragmented, potentially affecting memory access efficiency during the communication phase. While the two-pass method avoids sorting overhead, we found that the sorting-based approach consistently yields slightly better performance due to improved memory coalescing and reduced fragmentation (detailed in Section 5.4). However, to support diverse dataset characteristics and communication patterns, our system allows configurable selection between the two methods.

All-to-all communication. Once buffer preparation is complete, the second stage performs all-to-all communication to exchange data among GPUs. MNMGDATALOG supports two communication mechanisms: CUDA-aware MPI and CPU buffer-based MPI communication. CUDA-aware MPI enables direct communication between GPU buffers without requiring intermediate copies to CPU memory. This technique leverages advanced technologies like `GPUDirect RDMA` and `NVLink` to achieve high bandwidth and low-latency data transfers between GPUs across nodes. `GPUDirect RDMA` allows GPUs to communicate directly with the network interface card, bypassing the host CPU and reducing communication latency. Similarly, `NVLink` provides high-speed interconnects between GPUs on the same node, enabling faster data movement during intra-node communication. In systems that support CUDA-aware MPI, MNMGDATALOG's communication calls directly transfer GPU-resident buffers between processes, minimizing overhead and improving the performance of iterative relational algebra operations. To ensure compatibility across a broader range of systems, we also provide a CPU buffer-based communication mode, where GPU data is first copied to CPU memory before MPI communication using CPU buffers. After the communication, the data are copied back to the GPU. While this approach incurs additional data movement overhead, it guarantees that MNMGDATALOG can run on systems without CUDA-aware MPI support. MNMGDATALOG provides a runtime configuration to select between these two communication modes. This flexibility ensures that MNMGDATALOG can maximize performance on CUDA-aware MPI-enabled systems by utilizing direct GPU communication, while also maintaining portability across diverse HPC environments.

4.3 GPU-optimized data representation

MNMGDATALOG employs an open-addressing hash table with linear probing for efficient join execution. Hash joins dominate computational cost in recursive Datalog queries, making their optimization crucial for scalability. Each GPU builds a static hash table using the partitioned `Edge` relation as the inner relation. In iterative queries such as Weakly Connected Components (WCC), the inner relation remains unchanged across iterations, making static hash tables an ideal choice. By constructing the hash table once and reusing it in subsequent iterations, MNMGDATALOG eliminates unnecessary recomputation. The hash join consists of two phases: build phase and probe phase. During the build phase, the inner relation keys are inserted into the hash table, with collisions handled via linear probing. Contiguous memory storage ensures high

cache locality, reducing memory latency. In the probe phase, the Delta relation serves as the outer relation, querying the hash table for matches to propagate new facts. Although our engine is optimized for iterative join during Datalog evaluation, many of these join strategies such as static hash table construction and GPU-friendly memory layouts can also benefit traditional non-iterative join workloads. Traditional joins share the same underlying hash-based probing mechanism. Therefore, mnmgDatalog's GPU-accelerated join implementation is general enough to improve performance in both iterative and non-iterative relational joins.

In the WCC example shown in Figure 2, each GPU builds a static hash table using the distributed Edge relation as the inner relation. In many Datalog queries, such as WCC, the inner relation remains static across iterations. A static hash table is an ideal choice for such scenarios, as it can be built once and reused in subsequent iterations without modification. The local join operation involves matching the current connected component (CC_{Delta}) relation with the Edge relation to propagate the smallest component ID across connected nodes. The hash join process consists of two phases: the build phase and the probe phase. In the build phase, keys from the inner relation (distributed *Edge*) are inserted into the hash table, with collisions handled through linear probing. Linear probing ensures that keys are stored in contiguous memory spaces, which improves cache locality and reduces memory latency—a key advantage on GPUs where memory bandwidth is often the bottleneck. In the probe phase, the outer relation (CC_{Delta}) is used to query the hash table for matches. When a match is found, the corresponding tuples are combined to propagate the smallest component ID across connected nodes.

To further optimize performance, mnmgDatalog leverages grid-stride loops, ensuring that each GPU thread processes multiple tuples in a row-major order. This approach minimizes memory divergence by aligning memory accesses with the GPU's warp execution model, reducing unnecessary stalls and maximizing cache hits. This design choice ensures that the local join operations are fast, memory-efficient, and scalable. By avoiding redundant hash table constructions and reducing memory access overhead, mnmgDatalog achieves high throughput in iterative Datalog queries.

4.4 Recursive aggregation in mnmgDatalog

Our engine, mnmgDatalog, supports recursive aggregation through an optimized join strategy that ensures the smallest representative node is efficiently propagated across recursive iterations. In the WCC query, the recursive use of the MIN aggregate prevents unnecessary materialization of intermediate results while ensuring convergence. Specifically, mnmgDatalog implements a semi-na"ive evaluation strategy that tracks changes across iterations using the CCDelta and CCFull relations. The MIN aggregator is applied incrementally, ensuring that only newly discovered smaller IDs are propagated in each iteration. This approach significantly reduces memory usage and computational overhead by minimizing redundant joins and intermediate results. By overcoming the limitations of modern Datalog engines, mnmgDatalog enables efficient recursive aggregation, making it well-suited for graph queries like WCC that require fixed-point evaluation with aggregation.

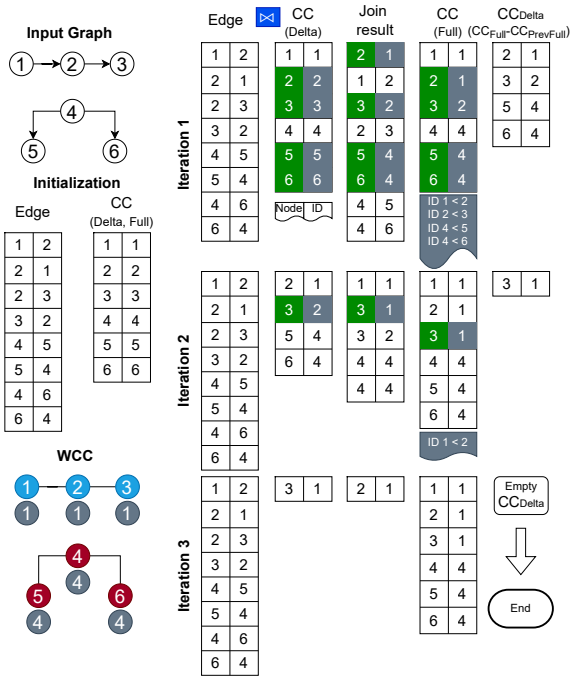

Figure 5: All iterations of Weakly Connected Components (WCC) calculation using the semi-naïve evaluation technique incorporating recursive aggregation.

Figure 5 shows the execution of the WCC query for an input graph using the semi-naïve evaluation technique, incorporating recursive aggregation. The input graph is shown on the top-left, followed by the step-by-step execution of the WCC query through multiple iterations. The initial step assigns each node to its own component, forming the CCDelta and CCFull relations, where Delta contains the newly derived component IDs in the current iteration, and Full accumulates all component IDs derived so far.

In the first iteration, the CC_{Delta} relation is joined with the Edge relation to propagate component IDs across connected nodes. The join result shows how smaller IDs are propagated to neighboring nodes, which is essential for identifying the

smallest representative node within each connected component. This propagation process leverages recursive aggregation using the MIN function, ensuring that each node's component ID is minimized efficiently. For example, initially, all nodes are assigned their same value as ID (node 1 - ID 1, node 2 - ID 2, and so on). After the aggregation, node 2 receives a smaller ID from node 1, and similarly, nodes 5 and 6 are updated with smaller IDs from their neighbors, as highlighted in the figure by the gray boxes. Following the semi-naïve evaluation strategy, the CCDelta relation for the next iteration is computed as the set difference between the updated CCFull and the previous CCFull. This ensures that only the newly discovered component IDs are used in the next iteration, reducing redundant computations. In the second iteration, the process repeats with the updated CCDelta, further propagating the smallest component IDs across connected nodes. The recursive aggregation continues to compress the connected components by applying the MIN function in each iteration. By the third iteration, all nodes have converged to their final component IDs, and the CC_{Delta} relation becomes empty, indicating that no new facts are being derived. At this point, the process terminates. The final connected components are shown in the bottom-left of Figure 5. Nodes with the same component ID belong to the same connected component in the graph. For instance, nodes 1, 2, 3 form one connected component, while nodes 4, 5, 6 form another.

This example demonstrates how the combination of recursive aggregation and semi-naïve evaluation enables efficient propagation of component IDs while minimizing redundant computations. The MIN aggregator ensures that the smallest representative node is identified for each component, and the incremental propagation through CC_{Delta} prevents the re-evaluation of previously derived facts, making the overall process scalable and efficient.

5 Evaluation

In this section, we present a comprehensive performance evaluation of mnmgDatalog, demonstrated by three sets of experiments: (1) evaluation with real applications on one single GPU, (2) evaluation of single join operation in multi-GPU environment, and (3) evaluation with real applications in multi-GPU environment.

5.1 Environment

We conducted all our experiments on the Polaris supercomputer at the Argonne Leadership Computing Facility. Each compute node is equipped with an AMD EPYC Milan 7543P 32-core CPU, 512 GB of DDR4 RAM, and four NVIDIA A100 GPUs interconnected via NVLink. Multi-node communication is facilitated by Slingshot 11 high-speed interconnects. For GPU-based Datalog systems, including mnmgDatalog, GPUlog, GPUJoin, and cuDF, we used a single GPU on a single node on Polaris. For multi-node experiments, we compared mnmgDatalog against SLOG, a distributed CPU-based Datalog engine, using the same number of nodes for both systems. Each SLOG node was configured to utilize 32 CPU threads to match Polaris' CPU architecture. Both SLOG and Soufflé programs were compiled into C++ using the -O3 optimization flag. CUDA-aware MPI was enabled by using MPI-GPU support to allow the MPI library to send and receive data directly from GPU buffers. To evaluate mnmgDatalog's portability, we also benchmarked its performance using CPU buffer-based MPI communication, where GPU data was first transferred to host memory, sent via MPI, and copied back to the GPU post-communication.

5.2 Test programs and datasets

To evaluate the performance and scalability of mnmgDatalog, we designed experiments that assess both single join performance and full iterative query execution. The single join benchmark isolates the core computational step of recursive Datalog execution, while the transitive closure (TC) (Section 2), same generation (SG) [38], and weakly connected components (WCC)(Section 2) benchmarks evaluate full multi-iteration workloads. For the single join benchmark, we used a synthetic dataset with 10M tuples per rank for weak scaling and a total of 10M tuples for the strong scaling experiment. For recursive queries (TC, SG, WCC), we used large real-world graphs from the Stanford Network Analysis Project (SNAP), SuiteSparse, and road network datasets [9, 23, 24]. These graphs span diverse domains, with output sizes ranging from millions to several billion edges, providing a comprehensive scalability assessment across varying data distributions and computational complexities. As the data are large-scale, parallel I/O is employed, where each MPI rank independently reads and writes its assigned partition from disk, ensuring efficient data distribution, reducing I/O contention, and enabling scalable processing across multiple nodes.

5.3 Single GPU Benchmark

To develop an optimized multi-node multi-GPU Datalog engine, we first evaluate its performance on a single GPU. This benchmark assesses the efficiency of mnmgDatalog in executing transitive closure (TC) and same generation (SG) queries compared to state-of-the-art GPU-based Datalog engines (GPULog, GPUJoin, cuDF) and a multi-threaded CPU-based solver (Soufflé). Table 1 and Table 2 present execution times for a single NVIDIA A100 GPU across mnmgDatalog, GPULog, and GPUJoin, while Soufflé is executed with 32 CPU threads to leverage its multi-threaded capabilities.

Table 1: Transitive Closure (TC) execution time comparison: MNMGDATALOG vs. GPULOG, Soufflé (AMD Milan CPU 32 cores), and GPUJoin on large graphs (OOM: out of memory).

Dataset name	*TC* edges	Time (s) MNMGDATALOG	GPULOG	Soufflé	GPUJoin
com-dblp	1.91B	**13.58**	26.95	232.99	OOM
fe_ocean	1.67B	**66.34**	72.74	292.15	100.30
usroads	871M	**75.07**	78.08	222.76	364.55
vsp_finan	910M	**81.14**	82.75	239.33	125.94
Gnutella31	884M	**4.75**	7.64	96.82	OOM

Table 2: Same Generation (*SG*) execution time comparison: MNMGDATALOG vs. GPULOG, Soufflé and cuDF. Soufflé running on 32 core AMD Milan CPU.

Dataset name	*SG* size	Time (s) MNMGDATALOG	GPULOG	Soufflé	cuDF
fe_body	408M	**9.08**	18.41	74.26	OOM
loc-Brightkite	92.3M	**1.66**	11.67	48.18	OOM
fe_sphere	205M	**3.55**	7.88	48.12	OOM
CA-HepTH	74M	**0.60**	4.79	20.12	21.24

For TC queries (Table 1), MNMGDATALOG demonstrates competitive performance against GPULog, outperforming it with up to 1.98× speedup. Compared to Soufflé, MNMGDATALOG achieves up to 20× speedup. Additionally, MNMGDATALOG is up to 4.8× faster than GPUJoin, which fails to process large datasets due to out-of-memory (OOM) errors, highlighting its limited scalability in recursive query execution. For SG queries (Table 2), MNMGDATALOG outperforms GPULog by up to 7× and achieves a speedup of up to 33.5× over Soufflé. Compared to cuDF, MNMGDATALOG demonstrates up to 35.4× higher performance, as cuDF fails to process most SG queries due to memory limitations. The comparison highlights the superior scalability of MNMGDATALOG over CPU-based approaches while demonstrating its robust handling of large graphs compared to existing GPU-based Datalog engines.

5.4 Iterative single join benchmark

We executed the single join operation using both sorting-based and two-pass buffer generation approaches to measure their respective impact on buffer preparation time, communication overhead, and overall execution time. Each approach was evaluated under both CUDA-aware MPI (CAM) and CPU buffer-based MPI communication to analyze the effect of direct GPU-to-GPU transfers versus CPU-mediated communication. This test emphasizes the efficiency of iterative join operations, as recursive queries rely heavily on repeated joins across iterations. We conduct both strong scaling and weak scaling benchmarks, providing a granular breakdown of individual operations, including join operation, buffer preparation, communication (both pre-join and post-join), and deduplication.

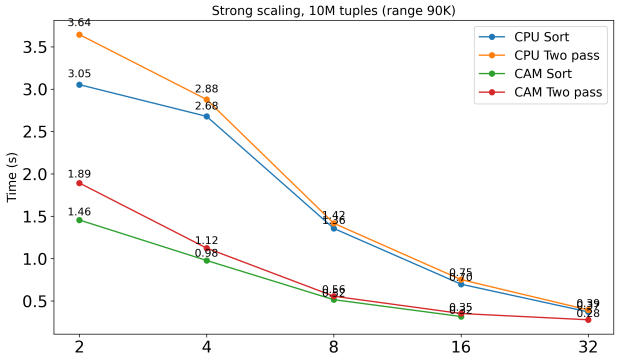

Figure 6: Strong scaling performance of the single iterative join operation in MNMGDATALOG scaling from 2 to 32 GPUs while keeping the total workload fixed at 10M tuples.

Strong scaling performance. For strong scaling, we fixed the total dataset size to 10 million tuples and increased the number of GPUs from 2 to 32, effectively reducing the workload per rank. Figure 6 illustrates the execution time across different configurations. As the number of GPUs increases beyond 2, execution time decreases due to improved workload distribution and parallel processing. For both CUDA-Aware MPI (CAM) and CPU buffer-based MPI, the sorting-based buffer preparation shows better scaling than the two-pass approach. When comparing CAM to CPU buffer-based MPI, CAM achieves lower execution times across all scales due to direct GPU-to-GPU transfers, whereas CPU-based MPI incurs additional memory copies between host and device. Figure 7 provides a breakdown of execution time for strong scaling. The join operation remains relatively constant in time across all configurations, whereas buffer preparation and communication contribute the most significant overheads. The two-pass approach suffers from higher buffer preparation time, while the sorting method reduces this cost significantly. The transition from 4 to 8 GPU achieves the largest performance gain, as it enables workload distribution and parallelism, but further increasing the number of GPUs beyond 16 results in diminishing returns due to less workload on GPUs.

Weak scaling performance. Across both strong and weak scaling, the sorting based buffer preparation consistently

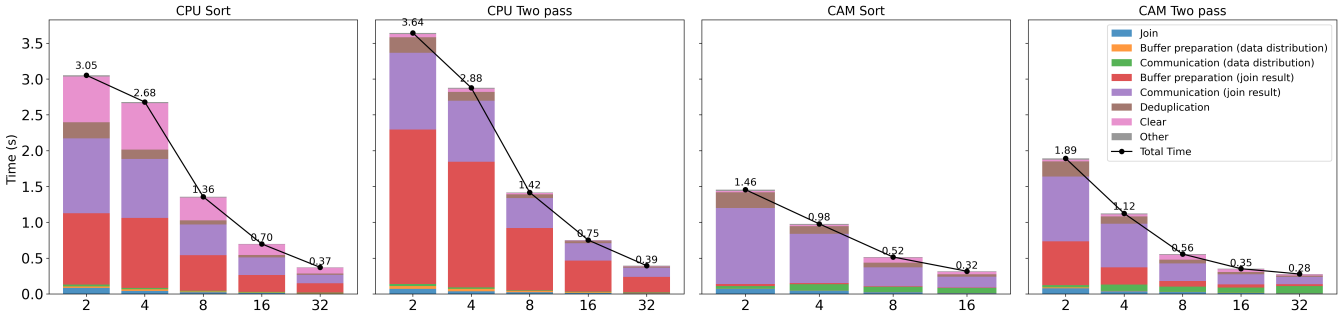

Figure 7: Strong scaling time breakdown of the single iterative join operation across key computational stages for both sorting-based and two-pass buffer preparation methods, evaluated under CPU buffer-based MPI and CUDA-aware MPI (CAM).

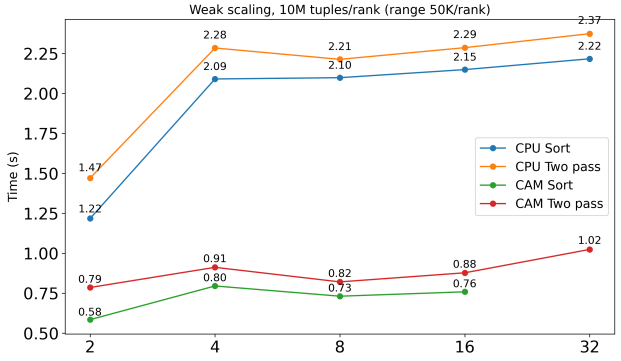

Figure 8: Weak scaling performance of the single iterative join operation in MNMGDATALOG.

outperforms two pass approaches, demonstrating the benefits of avoiding expensive atomic operations from two pass approach. CUDA-aware MPI (CAM) provides substantial performance gains over CPU buffer-based MPI by eliminating redundant host-device memory transfers, making it the preferred choice for GPU-accelerated join processing. The performance bottlenecks shift from join computation to buffer preparation and communication, indicating that optimizing these stages is crucial for improving the efficiency of iterative joins in distributed GPU environments.

5.5 Multi-node multi-GPU benchmark

We benchmark MNMGDATALOG on transitive closure (TC), same generation (SG), and weakly connected components (WCC) using up to 32 GPUs spanning multiple nodes on the Polaris supercomputer. As no existing multi-node, multi-GPU Datalog engine is available, we compare MNMGDATALOG against the state-of-the-art distributed CPU-based Datalog engine, SLOG, for transitive closure computation.

We intentionally used host-side (CPU) buffers for MPI communication to ensure a fair and neutral comparison with SLOG, which also relies on CPU buffers for data exchange. By configuring MNMGDATALOG in the same way, we isolated the impact of algorithmic design from hardware acceleration, allowing us to assess the raw algorithmic advantage under equivalent conditions. The performance benefits of CUDA-Aware MPI were evaluated separately in the single-join experiments (Section 5.4), where we demonstrated how our engine could further accelerate communication when GPU-to-GPU transfers are enabled.

Transitive closure. Table 3 presents the performance comparison of TC execution across multiple GPUs and nodes. MNMGDATALOG consistently outperforms SLOG. MNMGDATALOG achieves up to 31.97× speedup at 1 GPU and 13.89× speedup at 32 GPUs over SLOG. While MNMGDATALOG maintains a clear advantage across all configurations, the performance gap between MNMGDATALOG and SLOG narrows as the number of GPUs increases. This is due to the decreased workload per GPU on higher scales. However, even with this diminishing gap, MNMGDATALOG continues to exhibit superior scalability due to its optimized join processing and reduced memory overhead in recursive query execution whereas SLOG experiences significant overhead from CPU-bound relational operations. Figure 9 top row further illustrates the scaling trends, emphasizing that MNMGDATALOG exhibits near-linear scaling as the number of GPUs increases from 1 GPU to 32 GPUs. The speedup for *fe_ocean* ranges from 7.9× to 13.9×, *vsp_finan* from 3.6× to 9.9×, and *usroad* from 2.4× to 6.8× when scaling from 1 to 32 GPUs, compared to SLOG on the same nodes.

Figure 10 provides a detailed breakdown of the execution time for TC across different GPU configurations. The results show that join time and communication time decrease significantly as the number of GPUs increases, demonstrating

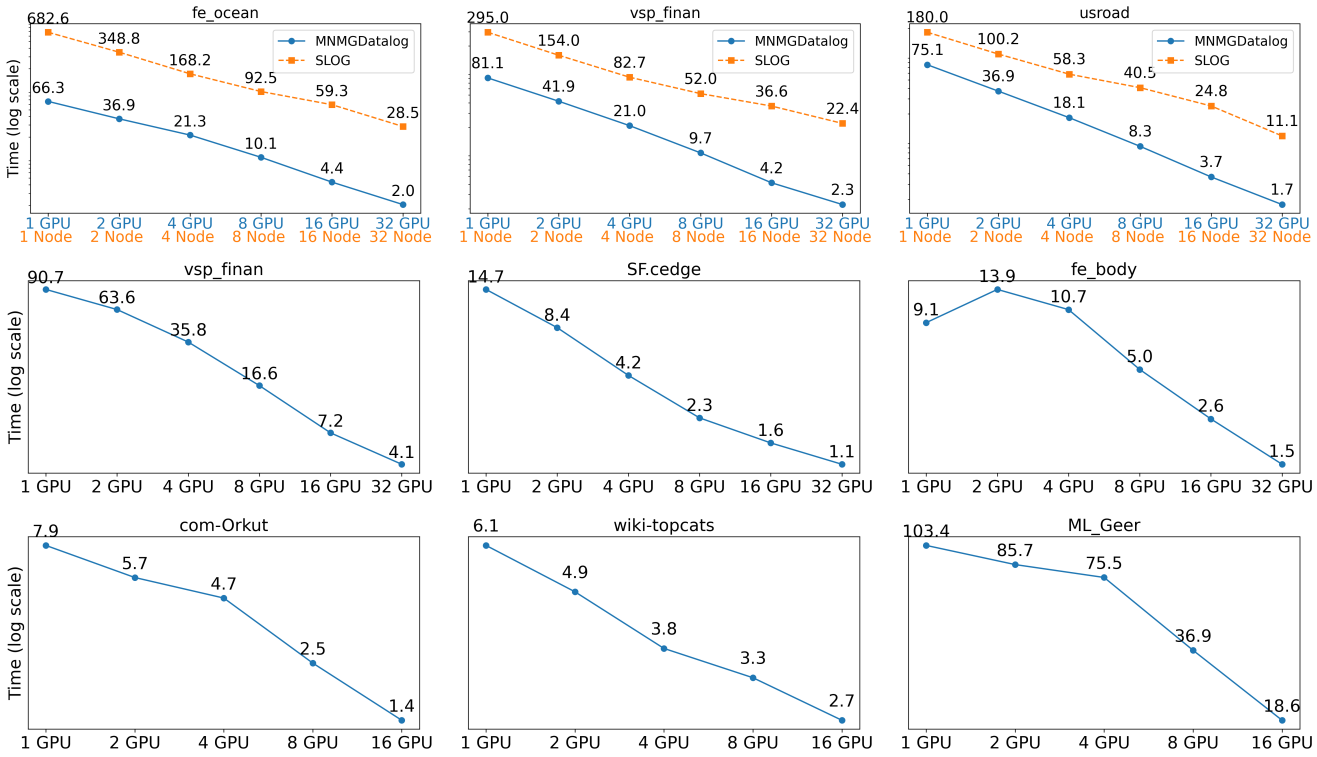

Figure 9: Execution time comparison of transitive closure (TC), same generation (SG), and weakly connected components (WCC) on MNMGDatalog. (Top) TC comparison between MNMGDatalog and SLOG, where SLOG uses 32 CPU threads per node, and MNMGDatalog employs GPU acceleration with CPU-based MPI buffer communication. (Middle) SG scaling from 1 to 32 GPUs. (Bottom) WCC scaling from 1 to 16 GPUs.

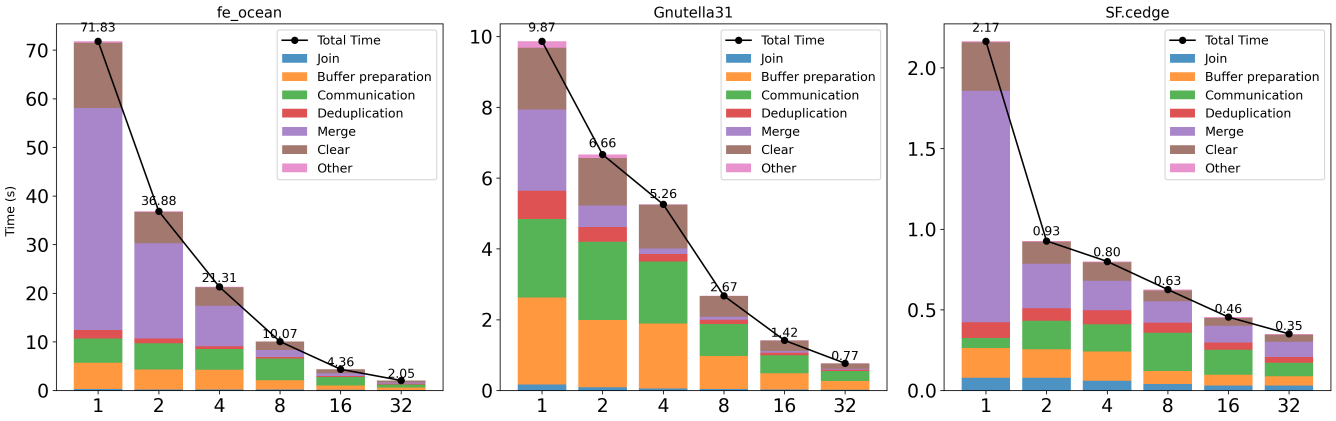

Figure 10: Breakdown analysis of Transitive Closure (TC) execution on MNMGDatalog illustrating the time distribution across key operations as the number of GPUs scales from 1 to 32.

the effectiveness of workload distribution across multiple nodes. Join operations benefit from parallelism, while communication overhead is reduced as individual GPU workloads become smaller with increasing GPU counts. However, merge and memory clearing time are the dominant contributors to the total execution time. This is expected, as merging the intermediate results in each iteration requires

allocations/deallocations of GPU memory during the iterations. Additionally, deduplication time reduces at higher GPU counts, as smaller partitions per GPU lead to more efficient duplicate elimination. This highlights the capability of mnmgDatalog to effectively scale distributed Datalog queries in a multi-node, multi-GPU environment, making it a robust solution for large-scale recursive query processing.

Table 3: Transitive Closure (TC) runtime (s) comparison: SLOG vs. mnmgDatalog. Scaling SLOG from 1–32 nodes (32 CPU threads per node) and mnmgDatalog from 1–32 GPUs via CPU-buffered MPI (sorting-based buffer preparation).

Dataset Name	TC Edges	Datalog Engine	1 Node 1 GPU	2 Nodes 2 GPUs	4 Nodes 4 GPUs	8 Nodes 8 GPUs	16 Nodes 16 GPUs	32 Nodes 32 GPUs
vsp_finan	910M	SLOG	294.99	154.04	82.69	52.01	36.61	22.40
		mnmgDatalog	**81.14**	**41.94**	**21.02**	**9.73**	**4.18**	**2.26**
usroads	871M	SLOG	180.05	100.20	58.28	40.49	24.75	11.06
		mnmgDatalog	**75.07**	**36.92**	**18.06**	**8.35**	**3.67**	**1.74**
fe_ocean	1.669B	SLOG	682.62	348.85	168.23	92.53	59.35	28.48
		mnmgDatalog	**66.34**	**36.88**	**21.31**	**10.07**	**4.36**	**2.05**
Gnutella31	884M	SLOG	315.52	143.85	58.32	31.70	16.69	10.56
		mnmgDatalog	**4.75**	**6.66**	**5.26**	**2.67**	**1.42**	**0.77**

Same generation benchmarking. The Same Generation (SG) query determines whether two nodes in a directed graph belong to the same hierarchical level. It is defined recursively as follows:

$$SG(u, v) \leftarrow Edge(p, u),\ Edge(p, v),\ u \neq v.$$
$$SG(u, v) \leftarrow Edge(x, u),\ SG(x, y),\ Edge(y, v), u \neq v.$$

The first rule captures direct relationships where two nodes share a common predecessor, while the second rule extends this recursively by checking for intermediate connections. Figure 9 middle row illustrates the SG execution times across multiple GPUs using CPU-buffered MPI communication with sorting-based buffer preparation. We achieve 6× to 22× speedup from 1 to 32 GPUs. For the *fe_body* dataset, we observe an anomaly where execution time increases from 1 GPU to 2 GPUs before improving with additional GPUs. This behavior is likely due to communication and partitioning overhead outweighing the benefits of parallelism at this scale. When moving from 1 GPU to 2 GPUs, data redistribution introduces inter-GPU communication, which incurs latency, especially for datasets where the computation-to-communication ratio is not sufficiently high.

Weakly connected component. Figure 9 bottom row presents the WCC execution time as the number of GPUs increases from 1 to 16. *com-Orkut* achieves 5.7 × speedup, while *ML_Geer* scales 5.5×, *wiki-topcats* shows a 2.3× speedup. The performance gain is largely attributed to local materialization, where each GPU retains its partial connected component state, reducing inter-GPU communication. This prevents redundant updates from being exchanged in every iteration, ensuring that only minimal data is transferred. The hash-based partitioning strategy further optimizes execution by keeping most component updates local, limiting global synchronization overhead. Scaling efficiency varies across datasets due to differences in graph connectivity. Graphs with denser connectivity require frequent inter-GPU communication, impacting performance; by contrast, sparser graphs benefit from localized processing, yielding better scalability.

6 Related work

GPU-based Datalog. Accelerating Datalog engines with GPUs has been a long-standing goal of the Datalog community. Early attempts, such as GPUDatalog [25] and RedFox [40], failed to gain traction due to their inability to efficiently handle iterative queries using semi-naïve evaluation, a core feature of high-performance Datalog engines. Additionally, limited GPU VRAM (under 16GB) required frequent host-to-device data transfers, further constraining performance and scalability. Consequently, GPU-based Datalog was largely overlooked for years. Recent advancements in GPU VRAM and computational power have revived interest in GPU-based Datalog systems. GPUJoin [33] demonstrated that GPU-based Datalog could outperform optimized CPU engines, though it was limited to binary relations and a narrow set of queries. Inspired by GPUJoin, GPULog [38] became the first GPU Datalog engine to support all fundamental relational algebra operations and semi-naïve evaluation, leveraging the novel HISA data structure for efficient execution.

Modern datacenter GPUs typically support advanced GPU-to-GPU interconnects, offering significantly higher memory bandwidth compared to GPU-to-host data transfers. We view mnmgDatalog as a critical extension of GPU-based Datalog systems to leverage this trend. By scaling the number of GPUs in the system, mnmgDatalog can handle significantly larger databases while maintaining low communication overhead.

Distributed Datalog. There has been significant progress in scaling Datalog-like languages to large machine clusters. Systems such as RDFox[27], BigDatalog[32], SociaLite[30], Myria[17], Nexus[18], and Radlog [16] have effectively utilized Apache Spark clusters to achieve scalability in data size. However, the query performance of these systems often can't scale beyond ten nodes. A more recent MPI-based distributed Datalog engine, SLOG [14], has demonstrated promising results in performance scaling, achieving near-linear scaling up to 64 nodes and gradual saturation up to 256 nodes on ANLF's Theta supercomputer. The design of mnmgDatalog draws inspiration from SLOG, adapting its data partitioning

and communication techniques to the GPU. By integrating CUDA-aware MPI, mnmgDatalog is optimized to scale efficiently on modern high-performance computing clusters, leveraging the computational power and interconnect capabilities of GPUs.

Monotonic Aggregation and Semiring Provenance. Datalog's basic semantics are simple, but extensions are needed for complex real-world queries in domains like program and graph analysis. Recursive aggregation, which allows monotonically updating existing tuples, is a widely adopted extension supported by CPU-based engines like BigDatalog[32], RecStep[11], and Logica [35]. The GPU-based system Lobster [4] formalizes monotonic aggregation using semiring provenance [15, 41], supporting various neural-symbolic reasoning queries, but it is limited to single-GPU setups. Efficient multi-GPU monotonic aggregation remains an open problem. In mnmgDatalog, we experimentally support multi-GPU monotonic aggregation with a specialized comparator during deduplication. While limited to single-integer column aggregation, this represents a step toward general multi-GPU support.

7 Conclusion

We presented mnmgDatalog, the first-ever multi-node, multi-GPU Datalog engine, designed for efficient execution of recursive queries over internet-scale datasets at unprecedented levels of scalability. Our approach integrates a radix-hash-based data partitioning strategy with CUDA-aware non-uniform all-to-all communication. Our benchmarks demonstrate that mnmgDatalog is the highest-performance Datalog engine to date, beating the previously-SOTA GPU-based competitor (GPULog) by 7×, the SOTA CPU-based engine (Soufflé) by up to 33×, and the distributed supercomputing competitor (SLOG) by up to 32×.

Our current implementation supports checkpointing only at the end of execution. As part of future work, we plan to integrate full checkpoint/restart capabilities to capture the application state at arbitrary points during execution at event of failures to enable recovery from mid-execution faults. We also plan to explore work-stealing-based load balancing techniques from sparse data processing to support dynamic rebalancing when GPUs finish computation unevenly. We are also expanding mnmgDatalog to support HPC systems with AMD and Intel GPUs leveraging GPU-aware MPI support in ROCm [39] and OneAPI [19]. To improve the load balancing, we are exploring two complementary strategies. For skew arising in intermediate joins (such as star or snowflake patterns), we plan to integrate GPU-accelerated worst-case optimal join algorithms that are inherently more robust to skewed input. For skew in outer relation distributions, we aim to implement sub-bucketing with double hashing, a technique adopted by other state-of-the-art engines to improve load balancing. These extensions will enhance the scalability and robustness of mnmgDatalog across diverse datasets.

Acknowledgments

This work was funded in part by NSF PPoSS large grants CCF-2316159 and CCF-2316157. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the Polaris supercomputer located at the Argonne National Laboratory. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] AMD. 5TH GEN AMD EPYC™ PROCESSOR ARCHITECTURE. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/5th-gen-amd-epyc-processor-architecture-white-paper.pdf, 2024.

[3] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*, pages 84–104. Springer, 2016.

[4] PAUL BIBERSTEIN, ZIYANG LI, JOSEPH DEVIETTI, and MAYUR NAIK. Lobster: A gpu-accelerated framework for neurosymbolic programming.

[5] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[6] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, 2009.

[7] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[8] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, sep 2001.

[9] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[10] Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702. Springer, 2012.

[11] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. Scaling-up in-memory datalog processing: Observations and techniques. *Proc. VLDB Endow.*, 12(6):695–708, Feb 2019.

[12] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092, 2020.

[13] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB*, volume 86, pages 209–219, 1986.

[14] Thomas Gilray, Arash Sahebolamri, Yihao Sun, Sowmith Kunapaneni, Sidharth Kumar, and Kristopher Micinski. Datalog with first-class facts. *Proc. VLDB Endow.*, 18(3):651–665, November 2024.

[15] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

[16] Jiaqi Gu, Yugo H Watanabe, William A Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. In *Proceedings of the 2019 International Conference on Management of Data*, pages 467–484, 2019.

[17] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 881–884, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Muhammad Imran, Gábor E Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. Fast datalog evaluation for batch and stream graph processing. *World Wide Web*, 25(2):971–1003, 2022.

[19] Intel. Intel MPI for GPU Clusters. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-2/intel-mpi-for-gpu-clusters.html, Jul 2023.

[20] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 422–430. Springer, 2016.

[21] Sidharth Kumar and Thomas Gilray. Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE*, volume 1, 2019.

[22] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*, pages 288–308. Springer, 2020.

[23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[24] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *International symposium on spatial and temporal databases*, pages 273–290. Springer, 2005.

[25] Carlos Alberto Martínez-Angeles, Inês Dutra, Vítor Santos Costa, and Jorge Buenabad-Chávez. A datalog engine for gpus. In *International Conference on Applications of Declarative Programming and Knowledge Management*, pages 152–168. Springer, 2013.

[26] Adithya Murali, Atharva Sehgal, Paul Krogmeier, and P Madhusudan. Composing neural learning and symbolic reasoning with an application to visual discrimination. *arXiv preprint arXiv:1907.05878*, 2019.

[27] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable rdf store. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II 14*, pages 3–20. Springer, 2015.

[28] NVIDIA. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/, 2022.

[29] NVIDIA. CUDA C Programming Guide: SIMT. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture, 2024.

[30] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE, 2013.

[31] Ramtin Shams, RA Kennedy, et al. Efficient histogram algorithms for nvidia cuda compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422. Citeseer, 2007.

[32] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Ahmedur Rahman Shovon, Landon Richard Dyken, Oded Green, Thomas Gilray, and Sidharth Kumar. Accelerating datalog applications with cudf. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 41–45. IEEE, 2022.

[34] Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. Towards iterative relational algebra on the {GPU}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1009–1016, 2023.

[35] Evgeny Skvortsov, Yilin Xia, and Bertram Ludäscher. Logica: Declarative data science for mere mortals. In *EDBT*, pages 842–845, 2024.

[36] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 417–432, 2017.

[37] Yihao Sun, Sidharth Kumar, Thomas Gilray, and Kristopher Micinski. Communication-avoiding recursive aggregation. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 197–208. IEEE, 2023.

[38] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Optimizing datalog for the gpu. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 762–776, New York, NY, USA, 2025. Association for Computing Machinery.

[39] Yıltan Hassan Temuçin, Mahdieh Gazimirsaeed, Ryan E. Grant, and Ahmad Afsahi. Rocm-aware leader-based designs for mpi neighbourhood collectives. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–12, 2024.

[40] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 44–54, 2014.

[41] Hangdong Zhao, Shaleen Deep, Paraschos Koutris, Sudeepa Roy, and Val Tannen. Evaluating datalog over semirings: A grounding-based approach. *Proc. ACM Manag. Data*, 2(2), May 2024.