Distributed Join Processing

Estella Xu

June 2025

1 Introduction

Efficiently processing large-scale data often requires distributing workload across multiple compute nodes. Single-node databases can struggle with performing expensive operations like joins on large tables. To address this, distributed database systems partition data across nodes and coordinate computations in parallel to both speed up computation time and balance load. In this project, we explore a lightweight, distributed join implementation using DuckDB, a fast in-process analytical database, while simulating a multi-node environment. We implement this system in two ways: first using Docker containers and sockets to simulate a distributed environment, and then adapting the implementation to run on Northwestern University's Quest high-performance computing cluster using MPI and SLURM.

The join operation is split into three main phases: partitioning and distribution, local joining, and collection. By using Parquet files for data exchange, we avoid costly Python object serialization and maintain compatibility with DuckDB's efficient columnar I/O. In the MPI implementation, we take advantage of non-blocking communication to reduce idle time and improve performance.

This paper describes the design and implementation of our system, compares performance across different environments and node counts, and evaluates the scalability of our approach in an HPC setting.

2 Background

2.1 Docker

Docker is a platform that enables developers to build, ship, and run applications in lightweight, portable units called **containers**. It abstracts the underlying infrastructure and provides a consistent environment across different stages of software development and deployment.

A **container** is an isolated, executable environment that includes the application code, runtime, system tools, libraries, and configurations — all bundled into a single unit. Containers share the host OS kernel but remain isolated from each other using Linux namespaces and control groups (cgroups), which ensures security, resource control, and minimal overhead compared to traditional virtual machines.

Docker supports two primary execution models to manage containers beyond simple docker run commands:

Compose Mode: This is used for local multi-container applications. With docker-compose up, users can define and run multiple interconnected services (e.g., a web server, database, cache) using a docker-compose.yml file. It's ideal for development or testing on a single machine.

Swarm Mode: This enables Docker-native clustering. Using commands like docker swarm init and docker service create, users can deploy containers across multiple hosts as a unified cluster.

In this project, since our primary focus is on implementing and optimizing distributed join algorithms, and given the limitation in available hardware resources, we have chosen to use the Compose model as our development environment. Compose allows us to simulate a distributed setup locally by orchestrating multiple containers on a single machine, making it suitable for development and testing without the overhead of managing a full-scale cluster.

2.2 TCP Socket

In this project, we use TCP(Transmission Control Protocol) sockets as the underlying communication mechanism between distributed nodes. A TCP socket is a connection-oriented and reliable communication method that enables bidirectional and ordered data transmission once a connection is established. Unlike UDP, which does not guarantee delivery, ordering, or integrity, which would risk incorrect or incomplete join results, TCP includes built-in mechanisms for error checking, acknowledgment, and retransmission, making it suited for our distributed join algorithm where data correctness and consistency are critical.

However, using TCP does come with trade-offs. The connection setup involves a three-way handshake, which adds overhead and resource consumption, especially under high concurrency. TCP also typically incurs slightly higher latency due to its reliability mechanisms.

2.3 DuckDB

DuckDB is an in-process analytical DBMS optimized for OLAP workloads. It runs within the host application (e.g., Python/C++), avoiding inter-process overhead and

supporting efficient, ACID-compliant SQL queries, including joins and window functions. Its native support for formats like Parquet and Arrow and zero-dependency design make it ideal for lightweight, embedded analytics.

In this project, we use DuckDB as the underlying engine for executing local joins on each node. Since we're not focused on its internal physical operators, our emphasis is on implementing the logic for distributed coordination.

2.4 MPI

The Message Passing Interface (MPI) is a standardized system designed to allow processes to communicate in a parallel computing environment. MPI enables efficient communication between distributed processes using point-to-point (e.g., `send`, `recv`) and collective (e.g., `broadcast`, `gather`) operations. Unlike shared-memory parallelism, MPI assumes each process has its own local memory, making it ideal for high-performance computing (HPC) environments such as multi-node clusters. In this project, we use MPI to coordinate communication among distributed DuckDB instances, allowing us to implement partitioning, local computation, and result collection in a scalable and efficient manner.

MPI's non-blocking communication primitives (e.g., `Isend`, `Irecv`) are particularly useful for overlapping computation with communication, allowing for asynchronous execution and less idle time.

2.5 Northwestern's Quest Cluster

Quest is Northwestern's high-performance computing cluster. Quest consists of thousands of nodes, each with its own memory and storage. Users submit jobs via the SLURM resource manager, which schedules resources and queues jobs based on available capacity and priorities.

In this project, we used Quest to evaluate the performance of our MPI-based implementation at various scales. We were able to experiment with different node configurations (e.g. 1, 3, 8, and 16 nodes), making it a suitable platform for analyzing the performance and scalability of the distributed joins on different levels of parallelism.

3 Implementation

3.1 Docker and Socket Implementation

3.1.1 Docker Setting

Each node is encapsulated as an independent container. All nodes operate within a shared virtual network with static IPs to ensure predictable communication.

3.1.2 Communication Mechanism

- The Coordinator node is responsible for **distributing data and SQL** to the workers, also **collecting results** via sockets.
- Each node acts as both **server** and **client** to enable P2P communication.
- **Concurrent multithreading:** one thread continuously listens for incoming connections, accepts them, and creates new threads to handle each connection.

3.1.3 Data Partitioning and Management

- The coordinator loads the full TPC-H dataset and range-partitions it by a specified column.
- Each partition is sent to a node via socket; the coordinator retains one partition.
- A catalog tracks each table's partitioning column and value ranges per node.
- Each node has a dedicated directory with isolated access permissions.
- Data is stored in compressed **Parquet format** to reduce storage size and transmission volume.
- Upon receiving a completion signal, each worker loads its Parquet files and builds a local database.

3.1.4 Workflow

- Each node initializes its local database from the received partitioned data.
- Upon receiving a SQL query, the Coordinator analyzes the join condition to identify the join key.
- If the join key matches the current partitioning key and value ranges are aligned across nodes, the Coordinator instructs all Workers to execute the join operation locally and in parallel.
- If the join key differs from the existing partitioning key, a global repartitioning is initiated, after which all nodes execute the query independently on their updated local datasets. [1]
- Final results from all nodes are collected and aggregated by the Coordinator.

3.2 MPI Implementation

Code available here.

We then implemented the join using MPI to be used in a SLURM cluster. Similar to the Docker and Socket implementation, the join is split into three phases: *partitioning and distribution*, *local joining*, and *collection*.

3.2.1 Partitioning and Distribution

The partitioning and distribution implementation is available in `partition_utils.py`.

The coordinator node (rank 0) partitions the desired tables into equally-sized sections using a modulo operation based on the join key column. Each of the subsets of rows is loaded into a Parquet file, and is asynchronously sent to the worker nodes with the non-blocking MPI `Isend` communicator (note that we must first send the size of the byte stream to dynamically allocate space in the receiver's buffer). In turn, worker nodes receive the byte streams, write them to their own memory, and reconstruct the tables locally into DuckDB. The coordinator node processes its own subset locally in its memory without any sending or receiving.

3.2.2 Local Joining

The local joining phase is written as part of the collection phase to allow for asynchronous sending of intermediate results.

3.2.3 Collection

The collection and merging of local join results is found in `collect_utils.py`.

Each node executes the SQL query on its own DuckDB instance and writes the results to a temporary Parquet file. Each worker node sends the resulting file bytes with MPI's `send`. The coordinator receives the byte streams from each of the worker nodes, reconstructs their results files, and merges them into a final output. We chose to use `send` over a collective operation such as `Gather` to avoid unnecessary waiting of intermediate results.

4 Experimental Evaluation

4.1 Dataset and Query

TPC-H[2] is a standardized benchmark dataset developed by the Transaction Processing Performance Council (TPC) for evaluating the performance of database systems on decision support workloads. It consists of a set of relational tables representing a product-order-supplier business schema and a suite of parameterized SQL queries. The dataset is scalable (e.g., SF=0.5, SF=1), allowing generation of arbitrarily large data volumes, and is widely used to assess query execution performance, especially for complex joins and aggregations in analytical settings.

In our experiment, for simplicity, we select a simplified version of TPC-H Query 3 as the test workload

```
SELECT c.c_custkey, c.c_name, o.o_orderkey,
o.o_orderdate, o.o_totalprice
FROM customer c
JOIN orders o ON c.c_custkey = o.o_custkey
WHERE o.o_orderdate < DATE '1995-03-15'
ORDER BY o.o_orderdate DESC;
```

4.2 Correctness Verification

To obtain the ground truth, we execute the query on a single-node setup. Prior to result comparison, we sort the outputs and reset their indices to ensure consistent column order and alignment. After performing programmatic comparison, the outputs are identical, thereby validating the correctness of our distributed join implementation.

4.3 Baseline Experiments

We tested both implementations with one coordinator node and two worker nodes. To account for system variability on Quest such as load balancing and queue congestion, we ran each script multiple times and recorded both the average and the best execution times. We chose to include the best time in addition to the average time since it illustrates the distributed join's potential under minimal external interference.

4.3.1 Docker and Socket Results

In our Docker-based implementation, we adopt a network topology consisting of **one coordinator** and **two workers**.

Initially, no explicit resource constraints are set on the Docker containers. Upon inspection using system commands, the default Docker configuration on each container is: 7.654 GB Memory; 1 vCPU; 10,000 Mb/s Network bandwidth; and 2.4 GB/s I/O throughput.

To begin with, we evaluated the system under an idealized setting—where the *partitioning key perfectly matches the join key*, the *range partitions are aligned across nodes*, and the *resources are unconstrained*, with *no artificial network delay*. As shown in Table 1, the **Local Join** time refers to the duration each worker takes to execute the query on its own partition after receiving the SQL command. The **Distributed Join** time, on the other hand, includes the entire end-to-end process: from the coordinator dispatching the query to all workers, to collecting and merging the final results.

The difference between these two timings reflects the *communication overhead* incurred during the distributed join. For reference, we also measured the execution time of the same query on a single node with the entire dataset loaded locally (**Join on Single Node**).

Despite the favorable conditions, we observed that the distributed execution is still noticeably slower than the single-node baseline. However, the *local computation time* itself is relatively fast, indicating that the performance gap is primarily due to **TCP communication overhead**. Specifically, the proportion of time spent on communication (*Distributed Join* −

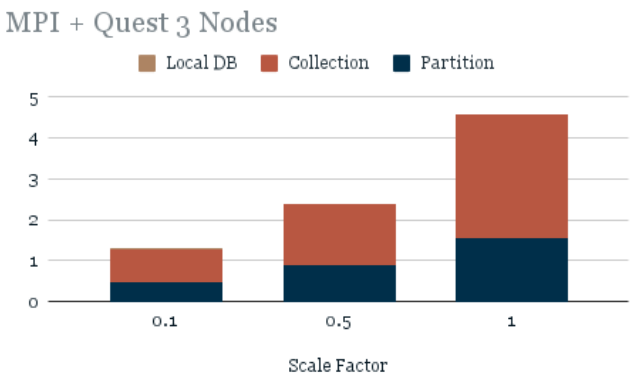

Figure 1: MPI and Quest Results on 3 Nodes

Local Join)/*Distributed Join* ranges from **88% to 96%** across different scale factors. This suggests that for relatively lightweight analytical queries, the cost of distributed coordination outweighs the benefit of parallel computation.

To address this bottleneck, we explored the use of the **Quest cluster infrastructure** with **MPI-based communication**. MPI offers high-bandwidth, low-latency interconnects and provides a more abstract and efficient interface for orchestrating distributed tasks across nodes. This enables more effective parallelism while reducing manual effort in managing communication semantics.

4.3.2 MPI and Quest Results

Since all nodes in the MPI cluster execute the same program (with behavior varying based on whether a node acts as the coordinator or a worker), we report timing measurements from the coordinator's perspective. This is appropriate since the coordinator is the one that produces the final output.

Figure 1 displays the best result for running the same query using the MPI and Quest implementation on 3 nodes. For each run, the chart is split into three sections:

1. Partition: this measures the total time spent in loading subsets of rows into Parquet files and confirming receipt from all worker nodes.

2. Collection: this measures the time spent by the coordinator in receiving intermediate results from worker nodes.

3. Local DB: this includes the local join time and the time spent on I/O between Parquet and DuckDB.

4.4 Scale-Out Experiments with Quest

In addition to comparing the implementations with 3 total nodes, we timed join execution on Quest using 1, 8, and 16 nodes. Since Quest is a shared cluster with many active users, we interleaved our job submissions to ensure that each experiment experienced a similar distribution of queue wait times and system-induced delays. From there, we recorded the results for the best execution time in Figure 2. The exact values are included in Tables 2- 4.

As expected, the time required to perform the local join decreases as nodes increase, since each partition is now smaller. The 8-node configuration appears the be the best total execution time, suggesting that it is potentially the best balance between parallel computation benefits and coordination overhead.

In the cases with 1, 8, and 16 nodes, the partitioning phase has the largest bottleneck. To further analyze this, we also measured more fine-grained times.

Shown in Figure 3 is the breakdown of the partitioning time into the following:

1. The coordinator's partitioning time: the time it takes for the coordinator node to load the subsets of the rows to Parquet files and initiate the send.

2. The partition communication barrier time: the time the coordinator must wait until the worker nodes have received the tables (i.e., the time the MPI `Waitall` takes, since using `Isend` results in asynchronous execution).

3. The local database setup time: the time the coordinator takes to load its portion of the table into a local DuckDB.

Overall, time segment 1 (the coordinator's partitioning time) increases as the number of nodes grows, which is expected due to the additional overhead of generating more table splits. In contrast, the partition communication barrier time remains minimal across all configurations. For the single-node case, this value is effectively zero since no communication is needed. In the multi-node configurations, the barrier time is still small and relatively consistent. This is because the actual transmission of Parquet files occurs asynchronously in the background while the coordinator continues partitioning for other nodes, resulting in very little idle waiting during the `Waitall` stage.

In the multi-node experiments (3, 8, and 16 nodes), the dominant component of the partition phase is the coordinator's partitioning (segment 1), as the coordinator must repeatedly generate multiple partitioned Parquet files. For the single-node setup, time segment 3 (loading data into DuckDB locally) becomes the primary contributor. This is because the single node is responsible for reconstructing the full tables needed for the join, which is a more intensive operation when not distributed.

This presents a potential area of optimization – instead of the coordinator handling all splitting and distribution, we could consider using hierarchical distribution and result collection.

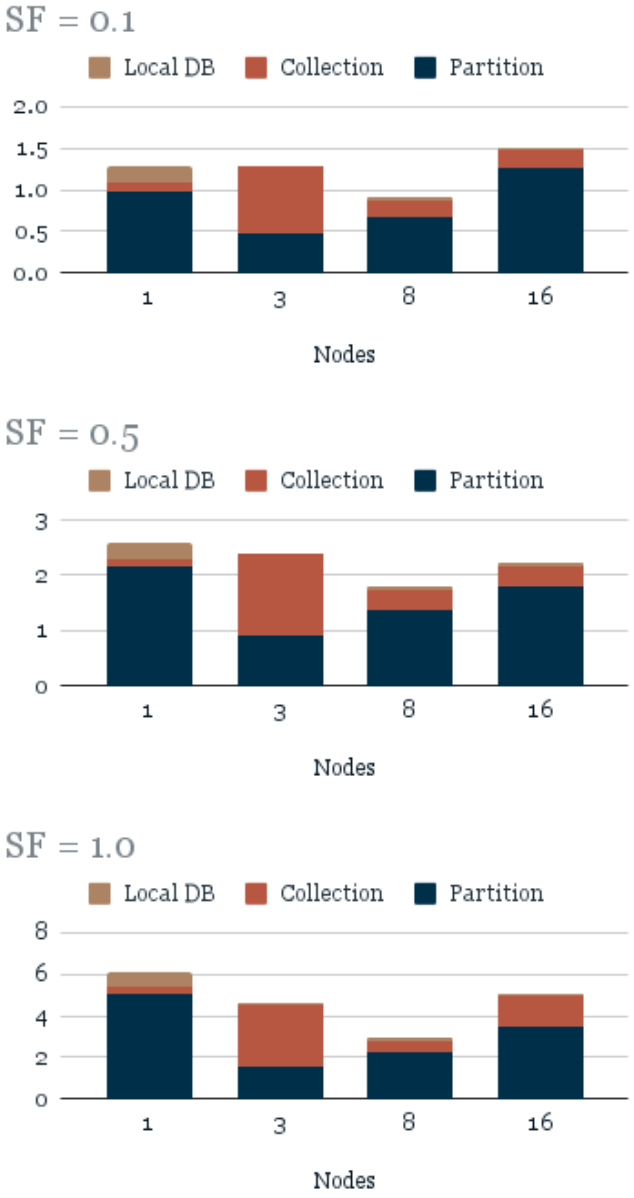

Figure 2: Best execution time on 1, 3, 8, and 16 nodes

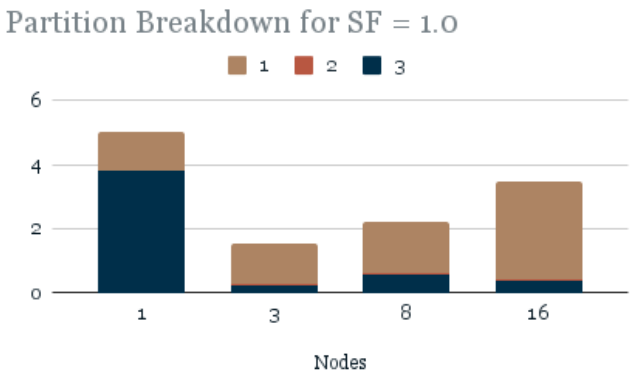

Figure 3: Partition Breakdown for SF = 1.0

5 Conclusions

Our experiments demonstrate that the Docker & TCP-based approach incurs prohibitive network communication costs. On the other hand, the MPI-based join implementation on the Quest cluster scales well to a degree, with performance benefits primarily from reduced local join and I/O time per node. As the number of nodes increased, the local join time decreased due to smaller data partitions per node. However, the coordinator's partitioning workload grew, introducing significant overhead. The 8-node configuration provided the best overall performance, likely giving the best trade-off between parallelization and coordination cost.

The idle time waiting for the asynchronized MPI `isend` calls is minimal, showing the effectiveness of using MPI in this scenario. The partitioning phase remains a bottleneck, which we can further explore using hierarchical partitioning and data distribution strategies to better distribute coordination load.

References

[1] Carnegie Mellon University. *15-445/645: Distributed OLAP (Lecture 24)*, `https://15445.courses.cs.cmu.edu/fall2024/slides/24-distributedolap.pdf`.

[2] Transaction Processing Performance Council. *TPC Benchmark H (TPC-H)*, `http://www.tpc.org/tpch/`.

[3] Open MPI Project. *Open MPI: Open Source High Performance Computing*, `https://www.open-mpi.org/`.

[4] Northwestern IT. *Quest: Northwestern's High Performance Computing Cluster*, `https://services.northwestern.edu/TDClient/30/Portal/KB/ArticleDet?ID=1964`.

Appendix: Data Tables

Table 1: Join Performance Under Different Scale Factors (in seconds)

	SF = 0.1	**SF = 0.5**	**SF = 1**	**SF = 5**
Local Join	0.01000	0.04000	0.09000	0.72000
Distributed Join	0.193192	0.439090	0.787724	17.91353
Join on Single Node	0.03169	0.07697	0.10595	1.02600

Table 2: Best Times for TPC-H SF = 0.01 across Different Node Counts (in seconds)

Nodes	Total	Partition	Coord Partition	Partition Communication	DB Setup	Collection	DB I/O & Join
1	1.2888	0.9884	0.2213	0.0034	0.7637	0.1092	0.1912
3	1.2930	0.4740	0.2924	0.0396	0.1420	0.8008	0.0181
8	0.9158	0.6661	0.4267	0.0331	0.2063	0.2064	0.0433
16	1.5068	1.2731	0.9840	0.0340	0.2552	0.2029	0.0309

Table 3: Best Times for TPC-H SF = 0.5 across Different Node Counts (in seconds)

Nodes	Total	Partition	Coord Partition	Partition Communication	DB Setup	Collection	DB I/O & Join
1	2.5858	2.1600	0.4544	0.0045	1.7011	0.1327	0.2931
3	2.4001	0.8971	0.6345	0.0495	0.2130	1.4858	0.0172
8	1.7899	1.3854	0.9621	0.0412	0.3821	0.3315	0.0730
16	2.2113	1.8018	1.5338	0.0460	0.2221	0.3629	0.0465

Table 4: Best Times for TPC-H SF = 1.0 across Different Node Counts (in seconds)

Nodes	Total	Partition	Coord Partition	Partition Communication	DB Setup	Collection	DB I/O & Join
1	6.0945	5.0268	1.1872	0.0095	3.8301	0.3356	0.7321
3	4.5795	1.5353	1.2564	0.0327	0.2463	3.0241	0.0200
8	2.9192	2.2227	1.5956	0.0340	0.5932	0.5946	0.1019
16	5.0393	3.4908	3.0785	0.0377	0.3746	1.4759	0.0727